ElevateDB Version 2 SQL Manual

Table Of Contents

Chapter 1 - Getting Started **1**

1.1 Adherence to the SQL Standard 1

1.2 Architecture 2

1.3 Creating a Local Database 7

1.4 Migrating a Database 15

1.5 Starting and Configuring the ElevateDB Server 22

1.6 Creating a Client-Server Database 38

1.7 Internationalization 46

1.8 Identifiers 49

1.9 NULLs 51

1.10 User Security 53

1.11 Buffering and Caching 57

1.12 Change Detection 63

1.13 Locking and Concurrency 64

1.14 Transactions 68

1.15 External Modules 72

1.16 Migrating Databases 74

1.17 Text Indexing 76

1.18 Optimizer 81

1.19 Result Set Cursor Sensitivity 87

1.20 Compression 89

1.21 Encryption 90

1.22 Stores 91

1.23 Replication 92

1.24 Row Value Constructors 95

1.25 Object Versioning 97

1.26 Custom Attributes 98

Chapter 2 - Operators **99**

2.1 Introduction 99

2.2 Boolean Operators 100

2.3 Comparison Operators 101

2.4 Arithmetic Operators 104
2.5 String Operators 107
2.6 Text Index Operators 108

Chapter 3 - Types 111

3.1 Introduction 111
3.2 Exact Numeric Types 112
3.3 Approximate Numeric Types 114
3.4 String Types 115
3.5 Binary Types 117
3.6 Date and Time Types 119
3.7 Interval Types 121
3.8 Boolean Types 124
3.9 Type Promotion 125

Chapter 4 - System Information 127

4.1 Introduction 127
4.2 Configuration Database 128
4.3 Collations Table 129
4.4 DataTypes Table 130
4.5 Modules Table 131
4.6 TextFilters Table 132
4.7 WordGenerators Table 133
4.8 Migrators Table 134
4.9 MigratorParams Table 135
4.10 LogEvents Table 136
4.11 Backups Table 137
4.12 Updates Table 138
4.13 FileIOStatistics Table 139
4.14 SessionStatistics Table 140
4.15 LoggedStatements Table 142
4.16 ServerSessions Table 143
4.17 ServerSessionLocks Table 144
4.18 ServerSessionStatistics Table 146
4.19 Users Table 148
4.20 Roles Table 149
4.21 UserRoles Table 150
4.22 Databases Table 151

4.23 DatabasePrivileges Table 153
4.24 Jobs Table 154
4.25 Stores Table 156
4.26 StorePrivileges Table 158
4.27 Files Table 159
4.28 Information Schema 160
4.29 Tables Table 161
4.30 TablePrivileges Table 162
4.31 TableColumns Table 163
4.32 TemporaryTables Table 165
4.33 Constraints Table 166
4.34 ConstraintColumns Table 168
4.35 Indexes Table 169
4.36 IndexColumns Table 171
4.37 Triggers Table 172
4.38 TriggerColumns Table 174
4.39 Views Table 175
4.40 ViewPrivileges Table 176
4.41 ViewColumns Table 177
4.42 ViewIndexes Table 178
4.43 TemporaryViews Table 179
4.44 Procedures Table 180
4.45 ProcedurePrivileges Table 181
4.46 ProcedureParams Table 182
4.47 Functions Table 183
4.48 FunctionPrivileges Table 184
4.49 FunctionParams Table 185
4.50 Dependencies Table 186
4.51 SchemaObjects Table 188
4.52 SchemaDifference Table 190
Chapter 5 - DDL Statements 193
5.1 Introduction 193
5.2 CREATE DATABASE 194
5.3 ALTER DATABASE 196
5.4 DROP DATABASE 197
5.5 RENAME DATABASE 198

5.6 CREATE USER 199
5.7 ALTER USER 200
5.8 DROP USER 202
5.9 RENAME USER 203
5.10 CREATE ROLE 204
5.11 ALTER ROLE 205
5.12 DROP ROLE 206
5.13 RENAME ROLE 207
5.14 GRANT PRIVILEGES 208
5.15 REVOKE PRIVILEGES 210
5.16 GRANT ROLES 212
5.17 REVOKE ROLES 213
5.18 CREATE JOB 214
5.19 ALTER JOB 217
5.20 DROP JOB 220
5.21 RENAME JOB 221
5.22 ENABLE JOB 222
5.23 DISABLE JOB 223
5.24 ENABLE JOBS 224
5.25 DISABLE JOBS 225
5.26 RESET JOB 226
5.27 CREATE STORE 227
5.28 ALTER STORE 229
5.29 DROP STORE 231
5.30 RENAME STORE 232
5.31 CREATE MODULE 233
5.32 ALTER MODULE 234
5.33 DROP MODULE 235
5.34 RENAME MODULE 236
5.35 CREATE TEXT FILTER 237
5.36 ALTER TEXT FILTER 238
5.37 DROP TEXT FILTER 239
5.38 RENAME TEXT FILTER 240
5.39 CREATE WORD GENERATOR 241
5.40 ALTER WORD GENERATOR 242
5.41 DROP WORD GENERATOR 243

5.42 RENAME WORD GENERATOR 244
5.43 CREATE MIGRATOR 245
5.44 ALTER MIGRATOR 246
5.45 DROP MIGRATOR 247
5.46 RENAME MIGRATOR 248
5.47 CREATE TABLE 249
5.48 ALTER TABLE 254
5.49 DROP TABLE 259
5.50 RENAME TABLE 260
5.51 CREATE TRIGGER 261
5.52 ALTER TRIGGER 266
5.53 DROP TRIGGER 268
5.54 RENAME TRIGGER 269
5.55 ENABLE TRIGGER 270
5.56 DISABLE TRIGGER 271
5.57 ENABLE TRIGGERS 272
5.58 DISABLE TRIGGERS 273
5.59 ENABLE DEFAULTS 274
5.60 DISABLE DEFAULTS 275
5.61 ENABLE GENERATED 276
5.62 DISABLE GENERATED 277
5.63 CREATE INDEX 278
5.64 CREATE TEXT INDEX 280
5.65 ALTER INDEX 282
5.66 ALTER TEXT INDEX 284
5.67 DROP INDEX 286
5.68 RENAME INDEX 287
5.69 CREATE VIEW 288
5.70 ALTER VIEW 290
5.71 DROP VIEW 292
5.72 RENAME VIEW 293
5.73 CREATE FUNCTION 294
5.74 ALTER FUNCTION 296
5.75 DROP FUNCTION 299
5.76 RENAME FUNCTION 300
5.77 CREATE PROCEDURE 301

5.78 ALTER PROCEDURE 304

5.79 DROP PROCEDURE 307

5.80 RENAME PROCEDURE 308

Chapter 6 - DML Statements **309**

6.1 Introduction 309

6.2 SELECT 310

6.3 INSERT 318

6.4 UPDATE 320

6.5 DELETE 322

Chapter 7 - SQL/PSM Statements **323**

7.1 Introduction 323

7.2 BEGIN..END 324

7.3 EXCEPTION 326

7.4 FINALLY 328

7.5 DECLARE 330

7.6 RAISE 334

7.7 IF 336

7.8 CASE 338

7.9 LOOP 340

7.10 REPEAT 342

7.11 WHILE 344

7.12 ITERATE 346

7.13 LEAVE 347

7.14 SET 348

7.15 CALL 350

7.16 USE 351

7.17 EXECUTE IMMEDIATE 353

7.18 PREPARE 355

7.19 UNPREPARE 357

7.20 EXECUTE 359

7.21 OPEN 361

7.22 CLOSE 363

7.23 FETCH 364

7.24 START TRANSACTION 367

7.25 COMMIT 369

7.26 ROLLBACK 371

7.27 INSERT 373

7.28 UPDATE 375

7.29 DELETE 377

7.30 REFRESH 379

7.31 SET LOG MESSAGE 380

7.32 SET PROGRESS 382

7.33 SET STATUS MESSAGE 384

7.34 ABORT 386

7.35 RETRY 388

7.36 LOG EVENT 390

7.37 SET STATEMENT CACHE 392

7.38 SET PROCEDURE CACHE 394

Chapter 8 - Administrative Statements 397

8.1 Introduction 397

8.2 ENABLE STATEMENT LOGGING 398

8.3 DISABLE STATEMENT LOGGING 400

8.4 MIGRATE DATABASE 401

8.5 SET MIGRATOR 403

8.6 BACKUP DATABASE 404

8.7 RESTORE DATABASE 406

8.8 SET BACKUPS STORE 408

8.9 PUBLISH DATABASE 409

8.10 UNPUBLISH DATABASE 411

8.11 SET INFORMATION COLLATE 412

8.12 COMPARE DATABASE 413

8.13 SAVE UPDATES 415

8.14 LOAD UPDATES 417

8.15 SET UPDATES STORE 419

8.16 COPY FILE 420

8.17 RENAME FILE 421

8.18 DELETE FILE 422

8.19 SET FILES STORE 423

8.20 VERIFY TABLE 424

8.21 REPAIR TABLE 425

8.22 OPTIMIZE TABLE 427

8.23 IMPORT TABLE 429

8.24 EXPORT TABLE 432
8.25 EMPTY TABLE 435
8.26 DISCONNECT SERVER SESSION 436
8.27 REMOVE SERVER SESSION 437
Chapter 9 - Numeric Functions 439
9.1 Introduction 439
9.2 ABS 440
9.3 ACOS 441
9.4 ASIN 442
9.5 ATAN 443
9.6 ATAN2 444
9.7 CEILING 445
9.8 COS 446
9.9 COT 447
Chapter 9 - String Functions 448
9.10 CURRENT_SESSIONID 448
Chapter 9 - Numeric Functions 449
9.11 DEGREES 449
9.12 EXP 450
9.13 FLOOR 451
9.14 LASTIDENTITY 452
9.15 LOG 454
9.16 LOG10 455
9.17 PI 456
9.18 POWER 457
9.19 RADIANS 458
9.20 RAND 459
9.21 ROUND 460
9.22 SIGN 462
9.23 SIN 463
9.24 SQRT 464
9.25 TAN 465
9.26 TRUNCATE 466
Chapter 10 - String Functions 469
10.1 Introduction 469
10.2 CHARACTER_LENGTH 470

10.3 CONCAT 471
10.4 CURRENT_GUID 472
10.5 CURRENT_USER 473
10.6 CURRENT_DATABASE 474
10.7 CURRENT_COMPUTER 475
10.8 LEFT 477
10.9 LENGTH 478
10.10 LOWER 479
10.11 LTRIM 480
10.12 OCCURS 481
10.13 POSITION 482
10.14 REPEAT 483
10.15 REPLACE 484
10.16 RIGHT 485
10.17 RTRIM 486
10.18 SUBSTRING 487
10.19 TRIM 489
10.20 UPPER 491
10.21 QUOTEDSTR 492
Chapter 11 - Array Functions **495**
11.1 Introduction 495
11.2 CARDINALITY 496
Chapter 12 - Date and Time Functions **499**
12.1 Introduction 499
12.2 CURRENT_DATE 500
12.3 CURRENT_TIME 501
12.4 CURRENT_TIMESTAMP 502
12.5 EXTRACT 503
Chapter 13 - Interval Functions **505**
13.1 Introduction 505
13.2 ABS 506
13.3 EXTRACT 507
Chapter 14 - Conversion Functions **511**
14.1 Introduction 511
14.2 CAST 512
14.3 COALESCE 527

14.4 IF 529
14.5 IFNULL 531
14.6 NULLIF 533
14.7 CASE 535
Chapter 15 - Aggregate Functions 537
15.1 Introduction 537
15.2 AVG 538
15.3 COUNT 540
15.4 MAX 542
15.5 MIN 544
15.6 RUNSUM 546
15.7 STDDEV 548
15.8 SUM 549
15.9 LIST 551
Chapter 16 - Boolean Functions 553
16.1 Introduction 553
16.2 EXISTS 554
Chapter 17 - SQL/PSM Functions 555
17.1 Introduction 555
17.2 ABORTED 556
17.3 BOF 558
17.4 EOF 560
17.5 ERRORCODE 562
17.6 ERRORMSG 564
17.7 ROWCOUNT 566
17.8 ROWSAFFECTED 568
17.9 SENSITIVE 570
17.10 LOADINGUPDATES 572
17.11 INTRANSACTION 574
17.12 OPERATION 576
17.13 COLUMNCOUNT 577
17.14 COLUMNNAME 579
17.15 STMTRESULT 581
Appendix A - Error Codes and Messages 583
Appendix B - System Capacities 591

Chapter 1
Getting Started

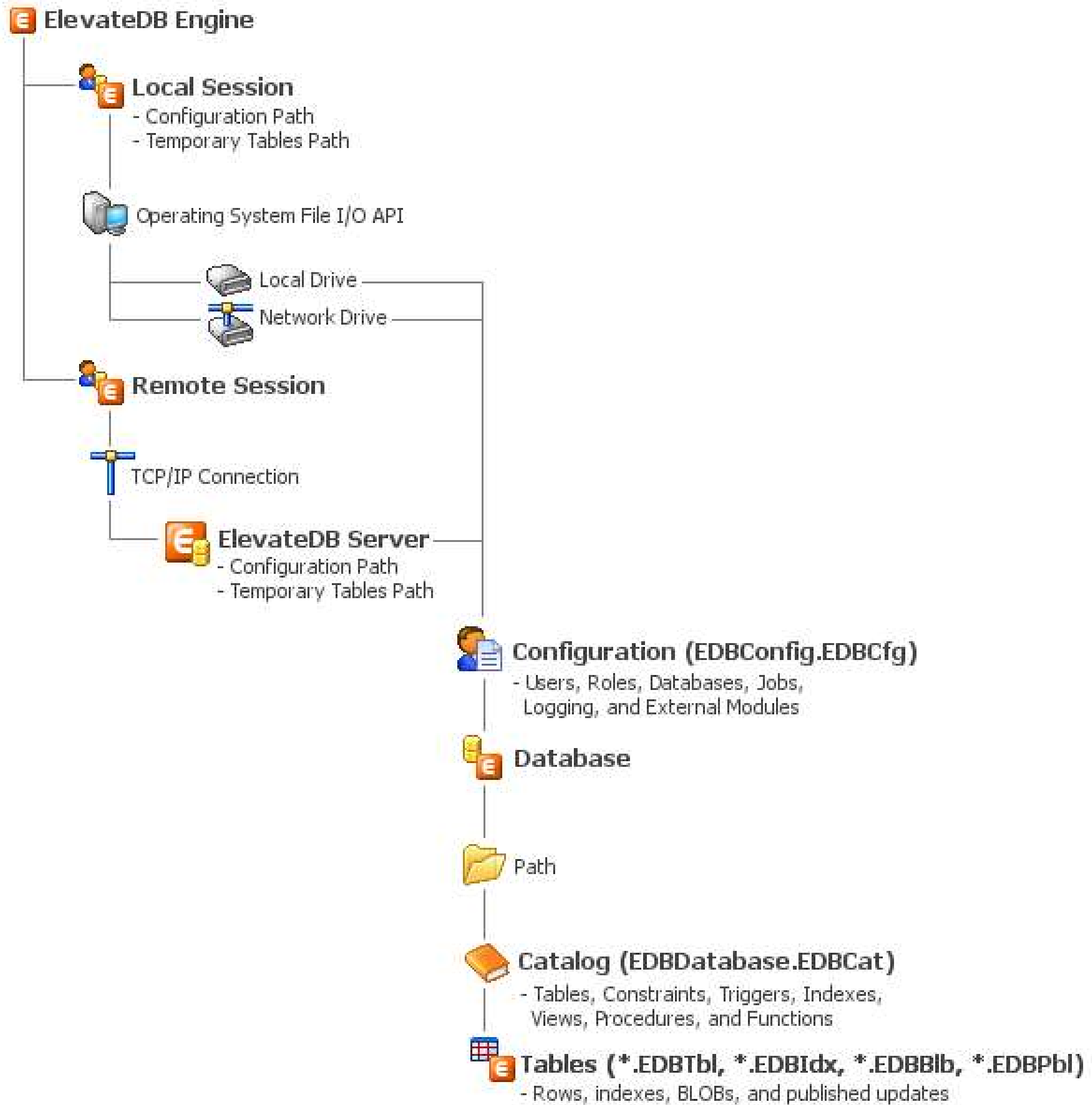
1.1 Adherence to the SQL Standard

ElevateDB was developed according to the SQL 2003 standard (ANSI ISO/IEC 9075:2003), and every effort was made to make sure that the product adheres to this standard as much as possible with no deviations. However, there are certain areas where ElevateDB does deviate from the standard. Each type, operator, statement, or function reference in this manual includes a summary of any deviation from the SQL 2003 standard at the end of the reference entitled SQL 2003 Standard Deviations that will detail any deviations from the SQL 2003 standard.

1.2 Architecture

ElevateDB is an embedded SQL database engine that can be compiled directly into your application and offers local single and multi-user access (file-sharing) and client-server access with the provided ElevateDB server. ElevateDB can switch between these modes of operation quickly, requiring just a few application changes.

The following image illustrates the general architecture of ElevateDB:



The various areas of the architecture are detailed next.

Engine

The ElevateDB engine can act as either a client or a server:

Engine Type	Description
Client	When acting as a client, the ElevateDB engine can create and manage both local and remote sessions (see below). These sessions can be multi-threaded, and there is no set limit to the number of sessions that can be created. The engine is automatically started whenever a new session is connected.
Server	When acting as a server, the ElevateDB engine can also create and manage both local and remote sessions. More importantly, it can listen for and respond to incoming connections and requests from remote sessions. An ElevateDB server can listen for incoming connections on all IP addresses or a specific IP address. By default, the ElevateDB server listens for incoming connections on port 12010. The ElevateDB Server is multi-threaded and uses one thread per session connection. ElevateDB can cache threads and keep a pool of unused threads available in order to improve connect/disconnect times. You may have an ElevateDB Server (or several) accessing the same configuration file and databases at the same time as other local applications such as CGI or ISAPI web server applications. This allows you to put critical server-side processing on the server where it belongs without incurring a lot of unnecessary overhead that would be imposed by the transport protocol of the ElevateDB Server. This can improve the performance of server-based local applications significantly, especially when they reside on the same machine as the ElevateDB Server and the databases being accessed are local to the server machine.

Sessions

ElevateDB is session-based, where a session is equivalent to a virtual user. In multi-threaded applications ElevateDB requires a separate session for each thread performing database access.

A ElevateDB session can be either local or remote:

Session Type	Description

Local	A local session gains direct access to database tables via the operating system API to a given storage medium, which can literally be any such medium that is accessible from the operating system in use. This means that a local session on the Windows operating system could access database tables on a Windows or Linux file server. ElevateDB automatically provides for the sharing of database tables using a local session. For example, an application can use local sessions on a small peer-to-peer network to provide a low-cost, multi-user solution without the added expense of using the ElevateDB Server. A local session has all of the capabilities of a remote session. Before a local session can perform any operation, it must be logged in with a proper user name and password.
Remote	A remote session uses sockets to communicate to an ElevateDB Server over a network (or on the same physical machine) using the TCP/IP protocol. ElevateDB allows all remote session communications to be encrypted. Compression is also available for remote sessions and can be changed whenever it is deemed necessary in order to improve the data transfer speed. This is especially important with low-bandwidth connections like a dial-up Internet connection. A remote session connects to a given ElevateDB Server via an IP address or host name and a port or service name. Before a remote session can perform any operation on an ElevateDB Server, it must be logged in with a proper user name and password.

Note
A developer can mix as many local and remote sessions in one application as needed, thus enabling a single application to access data from a local hard drive, a shared file server, or an ElevateDB Server. Also, local and remote sessions are completely identical from a usage standpoint, offering both navigational and SQL access methods.

A local ElevateDB session relies on a couple of important configuration items:

Item	Description

Configuration Path	This is the path where the configuration file is created and stored, and can be specified as being located in the process memory or on-disk. By default, the configuration file is called EDBConfig.EDBCfg, and is used by ElevateDB to store the information for the Configuration Database. ElevateDB can only access and use one configuration file per session, and the session cannot be connected when the configuration path is modified. Because the configuration file stores users, roles, and databases, among other things, it is very important that the configuration path is set properly for the local session, and that the local session uses the correct configuration file. Also, the configuration path is where all external modules are located, and it is where the system log (EDBConfig.EDBLog) is created and stored. ElevateDB creates a single hidden file called "EDBConfig.EDBLck" (by default) in the configuration path that is used for locking on the Configuration Database. It is created as needed and may be deleted if not in use by ElevateDB. However, if ElevateDB cannot write to this file it will treat the Configuration Database as read-only, thus preventing any modifications.
Temporary Tables Path	This is the path where ElevateDB creates any temporary tables used by the local session internally for SQL result sets, or for temporary tables created by the user via the CREATE TABLE statement. By default, ElevateDB uses the local user temporary files path in the operating system for this setting.

Please see your product-specific manual for more information on modifying the above configuration items.

Databases

ElevateDB stores all defined databases in the configuration file (see above). A database can be created using the CREATE DATABASE statement. The path specified when a database is created is subsequently used by ElevateDB to store both the catalog file and the database table files for the database. All metadata for a database is stored in the catalog file, and is represented in ElevateDB via the Information Schema. By default, the catalog file name for a database is EDBDatabase.EDBCat. Also, ElevateDB creates a single hidden file called "EDBDatabase.EDBLck" (by default) in the database path that is used for locking. It is created as needed and may be deleted if not in use by ElevateDB. However, if ElevateDB cannot write to this file it will treat the database as read-only. Please see the Locking and Concurrency topic for more information.

Tables

ElevateDB tables are divided into up to 4 physical files, one for the table rows, one for indexes, one for BLOBs (if there are BLOB columns present in the table), and one for published updates (if the table is published):

File Type	Description

Table Rows (.EDBTbl)	Used to store a fixed-length header for table-specific statistics along with the fixed-length rows. The use of a fixed-length header and rows allows for easier repair of tables in the case of physical table corruption. All rows contain a small row header and then the actual row data. BLOB columns contain a link to the BLOB file where the actual variable-length BLOB is stored in a block format.
Table Indexes (.EDBIdx)	Used to store a fixed-length header for index statistics along with the fixed-length index pages. The index page size is variable and can be set between 1024 bytes and 16 kilobytes on a per-table basis. All indexes defined for the table are stored in this file.
Table BLOBs (.EDBBlb)	Used to store a fixed-length header for BLOB statistics along with the fixed-length BLOB blocks. The BLOB block size is variable and can be set between 64 bytes and 64 kilobytes on a per-table basis. All BLOBs for all BLOB columns defined for the table are stored in this file.
Table Published Updates (.EDBPbl)	Used to store a fixed-length header for published updates statistics along with the fixed-length published updates blocks. The published updates block size is variable and can be set between 64 bytes and 64 kilobytes on a per-table basis.

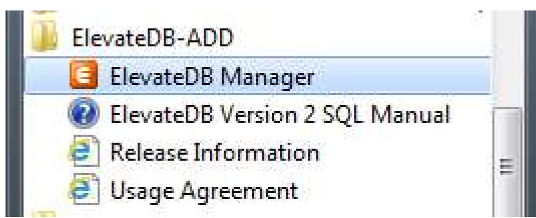
The file extensions used for these physical files can be changed. Please see your product-specific manual for more information.

1.3 Creating a Local Database

The following steps will guide you through creating the Tutorial database using the ElevateDB Manager.

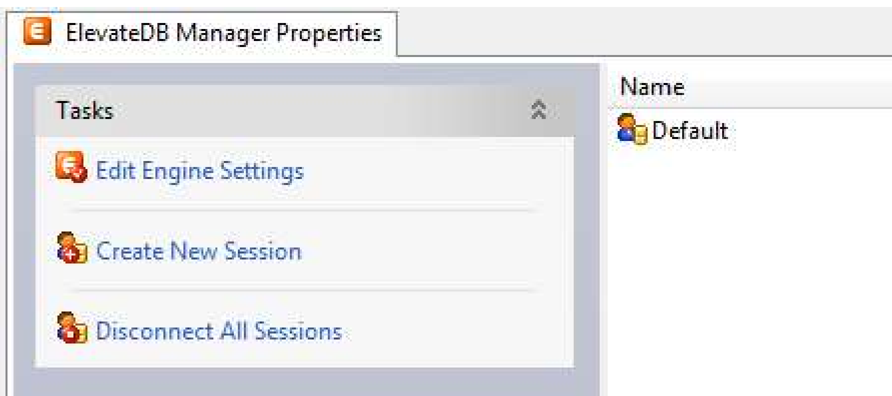
1. Start the ElevateDB Manager (edbmgr.exe) by clicking on the ElevateDB Manager link in the Start menu.

> **Note**
> The ElevateDB Manager is installed with the ElevateDB Additional Software and Utilities (EDB-ADD) installation available from the Downloads page of the web site.

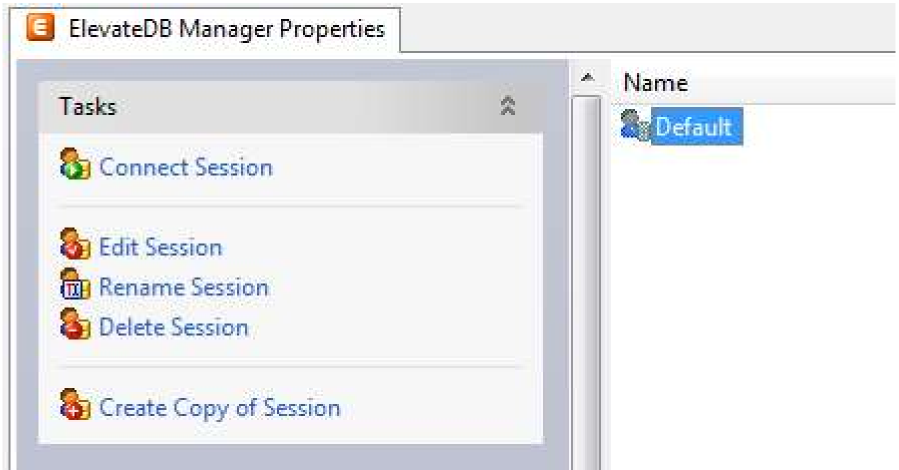


2. Make sure that the session is using the desired character set and configuration file folder (**C:\Tutorial**).

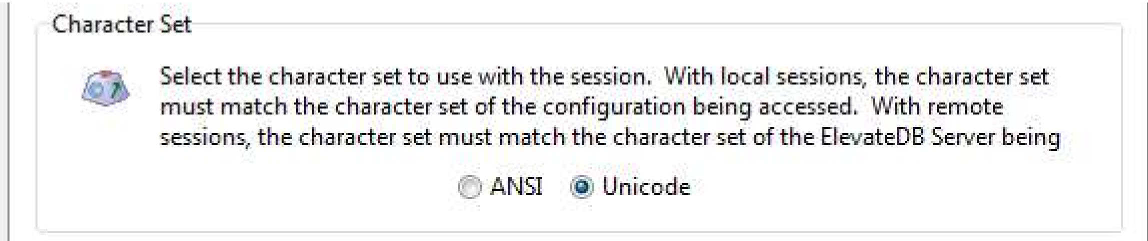
a. Select the **Default** session from the list of available sessions.



b. In the Tasks pane, click on the **Edit Session** link.

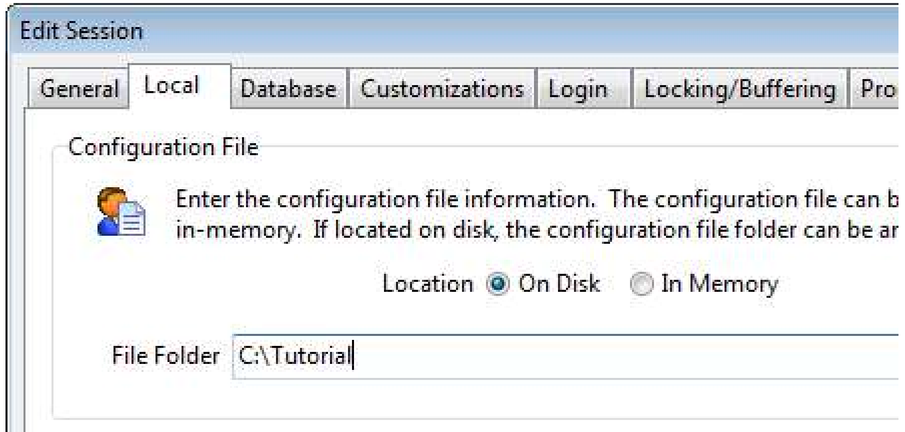


c. On the **General** page of the Edit Session dialog, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



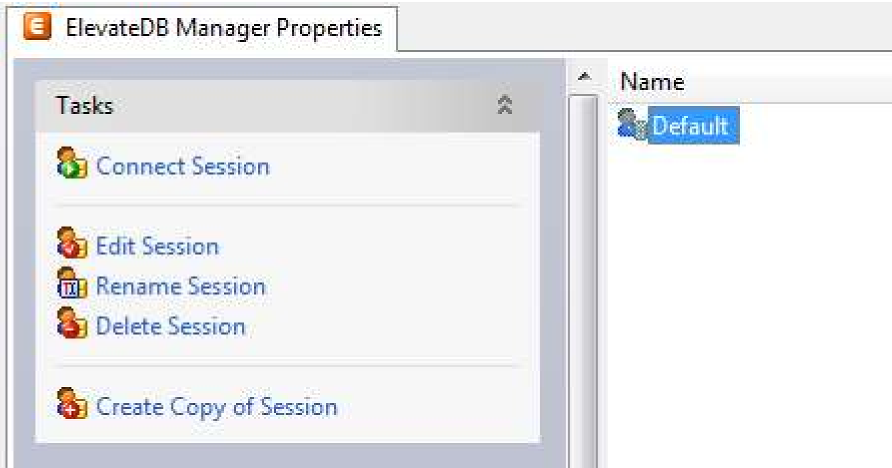
Note
If you're not sure which character set to select and this is the first time using the ElevateDB Manager, then leave the character set at the default of Unicode.

d. On the **Local** page of the Edit Session dialog, make sure that the Configuration File - File Folder is set to the desired folder.

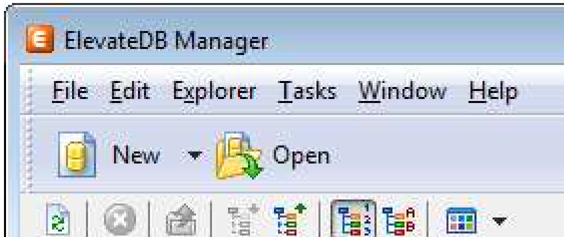


e. Click on the **OK** button.

3. Double-click on the **Default** session in the Properties window in order to connect the session.



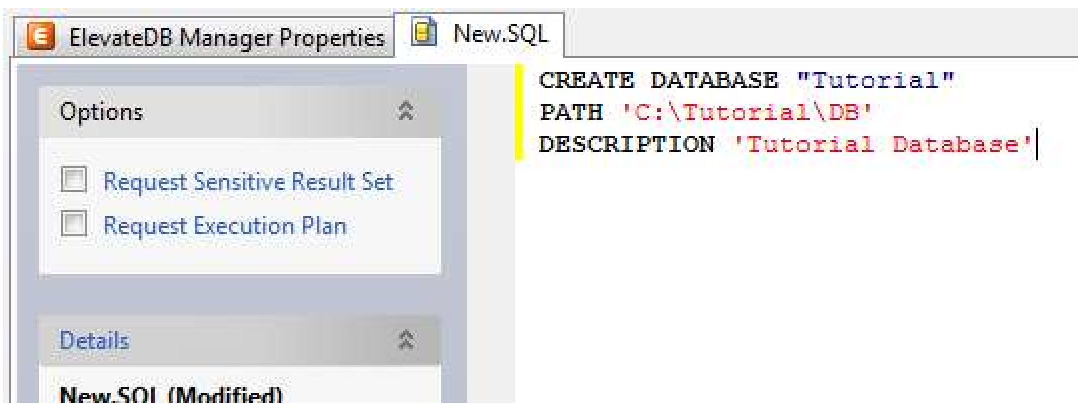
4. Click on the **New** button on the main toolbar.



5. Paste in the following CREATE DATABASE SQL statement in the new SQL window:

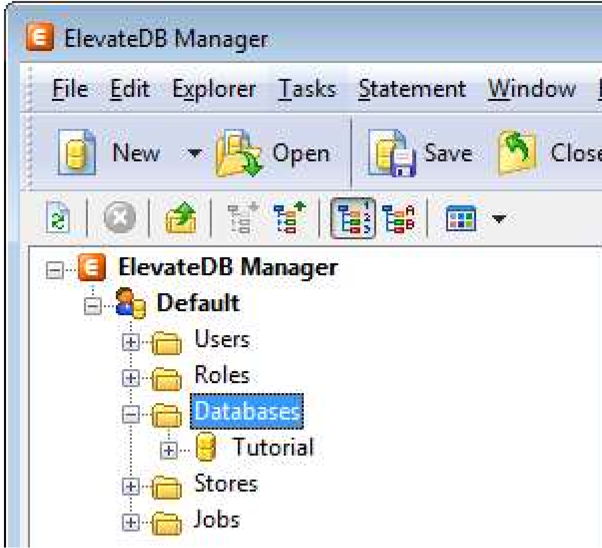
```
CREATE DATABASE "Tutorial"
PATH 'C:\Tutorial\DB'
DESCRIPTION 'Tutorial Database'
```

6. Press the **F9** key to execute the SQL statement.

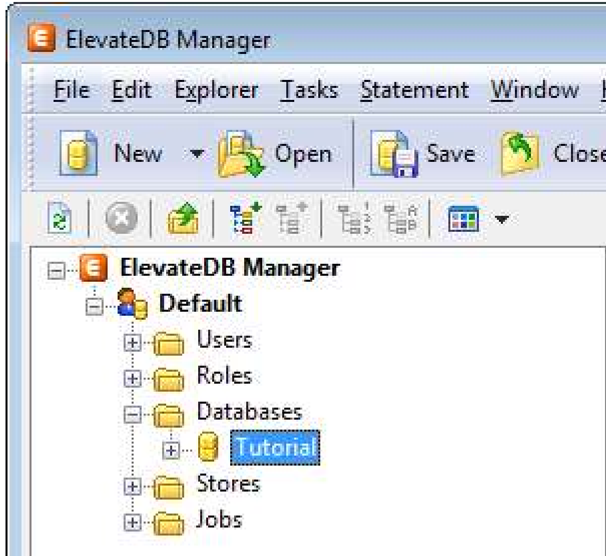


7. Press the **F5** key to refresh the explorer contents for the session.

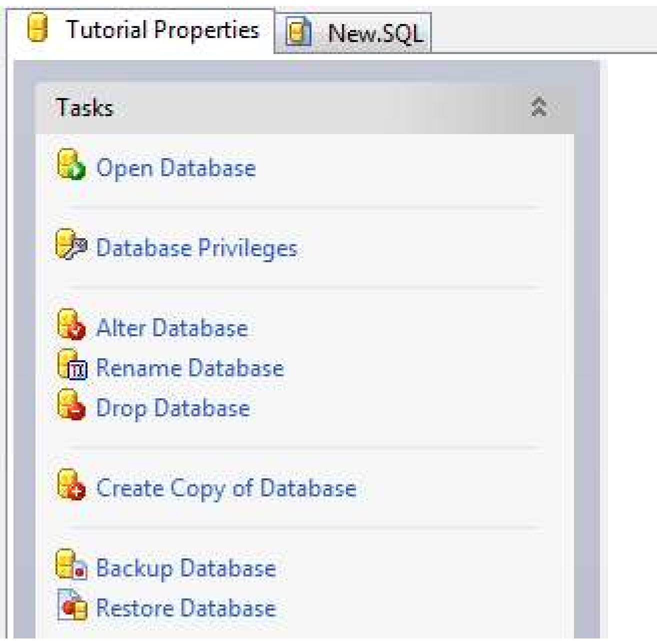
8. Click on the **+** sign next to the **Databases** node in the treeview.



9. Click on the new **Tutorial** database that you just created.



10. Press the **F6** key to make the Properties window the active window, and then click on the **Open Database** link in the Tasks pane.



11. Click on the **New.SQL** tab to bring forward the SQL window.

12. Paste in the following CREATE TABLE SQL statement. If you are using a Unicode session (see Step 2 above), then you should use the Unicode version of the CREATE TABLE statement. If you are using an ANSI session, then you should use the ANSI version of the CREATE TABLE statement:

ANSI

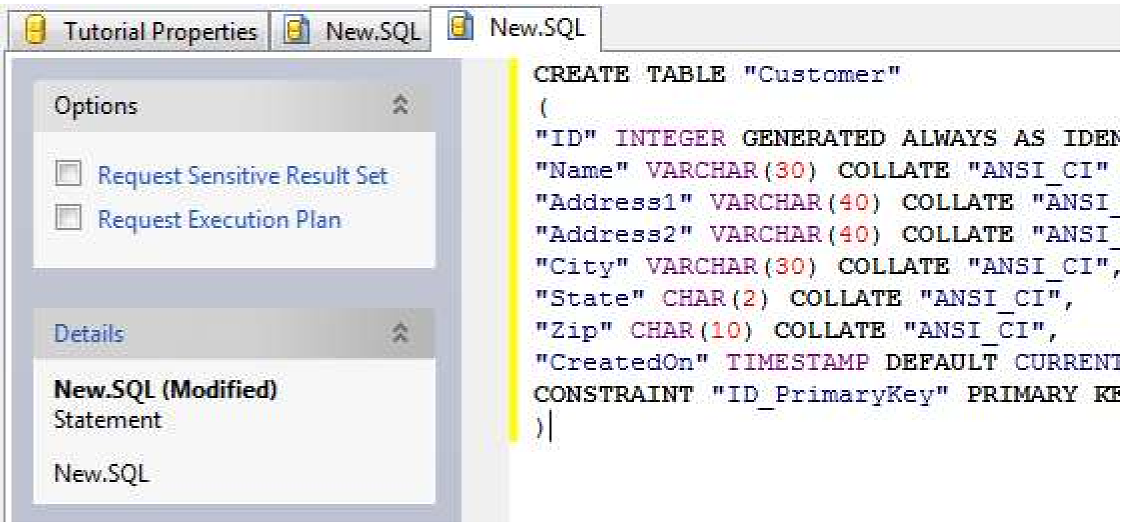
```
CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "ANSI_CI" NOT NULL,
"Address1" VARCHAR(40) COLLATE "ANSI_CI",
"Address2" VARCHAR(40) COLLATE "ANSI_CI",
"City" VARCHAR(30) COLLATE "ANSI_CI",
"State" CHAR(2) COLLATE "ANSI_CI",
"Zip" CHAR(10) COLLATE "ANSI_CI",
"CreatedOn" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID")
)
```

Unicode

```
CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "UNI_CI" NOT NULL,
"Address1" VARCHAR(40) COLLATE "UNI_CI",
"Address2" VARCHAR(40) COLLATE "UNI_CI",
"City" VARCHAR(30) COLLATE "UNI_CI",
"State" CHAR(2) COLLATE "UNI_CI",
"Zip" CHAR(10) COLLATE "UNI_CI",
"CreatedOn" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

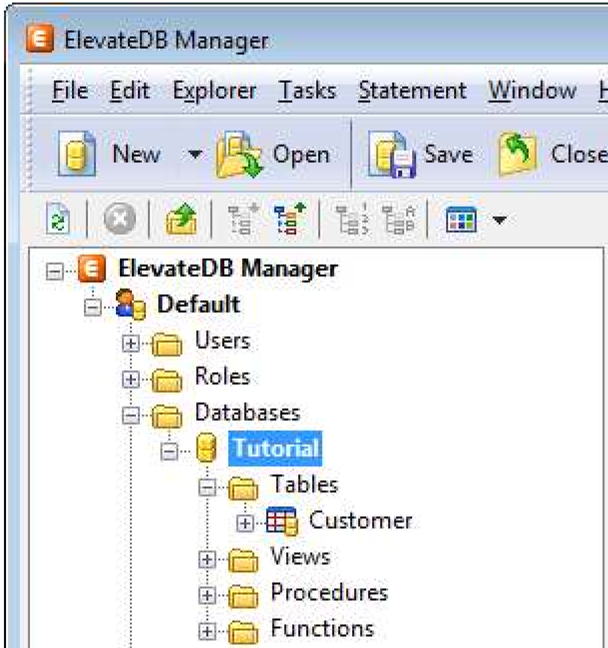
```
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID")
)
```

13. Press the **F9** key to execute the SQL statement.



14. Press the **F5** key to refresh the explorer contents for the session.

15. The table should now show up in the list of tables for the Tutorial database.



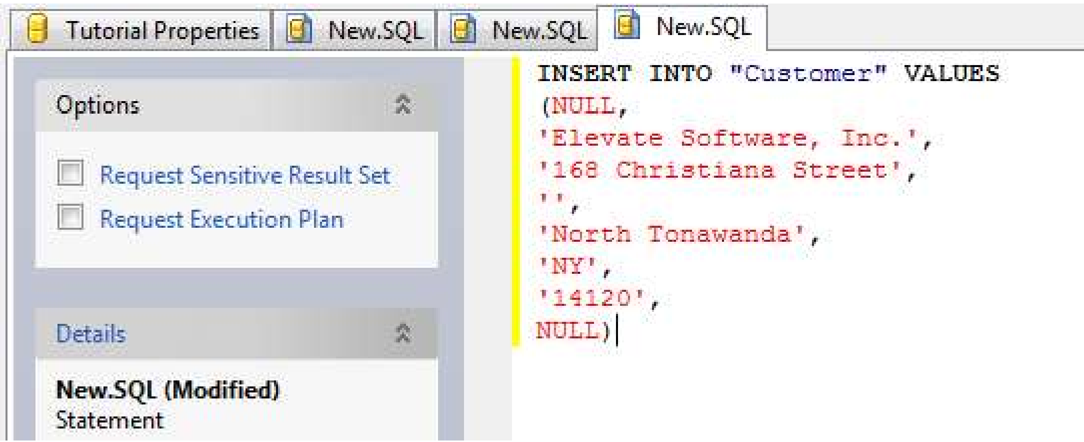
16. Click on the **New.SQL** tab to bring forward the SQL window.

17. Paste in the following INSERT SQL statement:

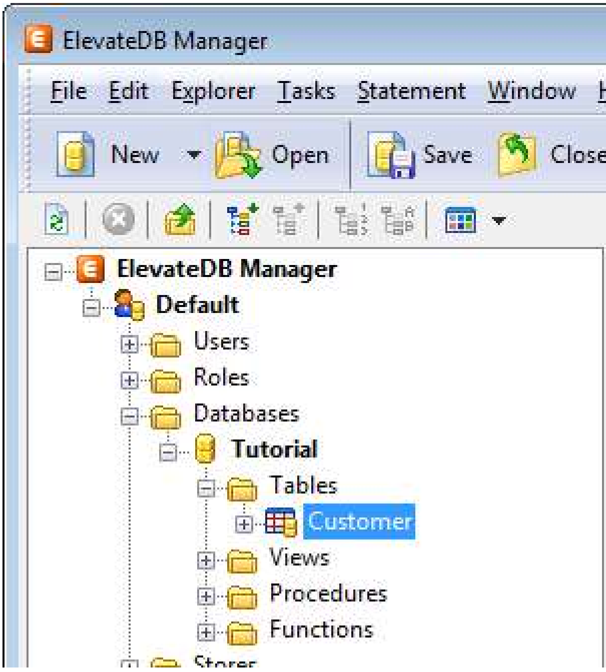
```
INSERT INTO "Customer" VALUES
(NULL,
```

```
'Elevate Software, Inc.',
'168 Christiana Street',
'',
'North Tonawanda',
'NY',
'14120',
NULL)
```

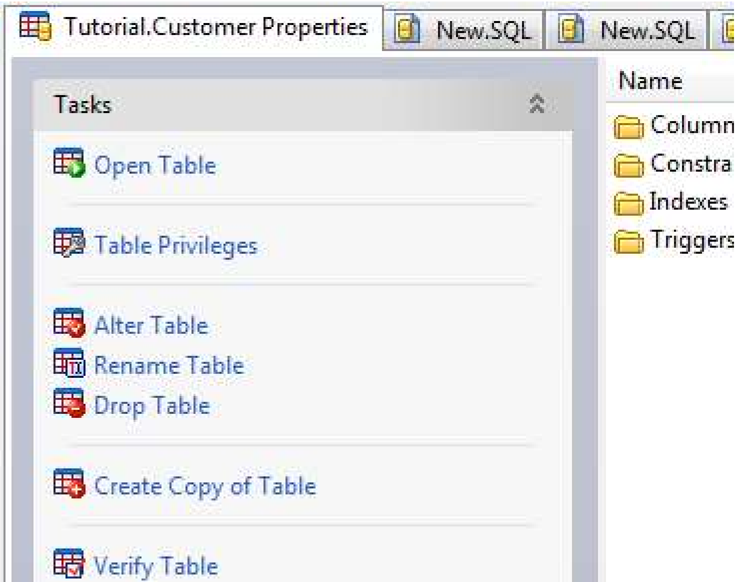
18. Press the **F9** key to execute the SQL statement.



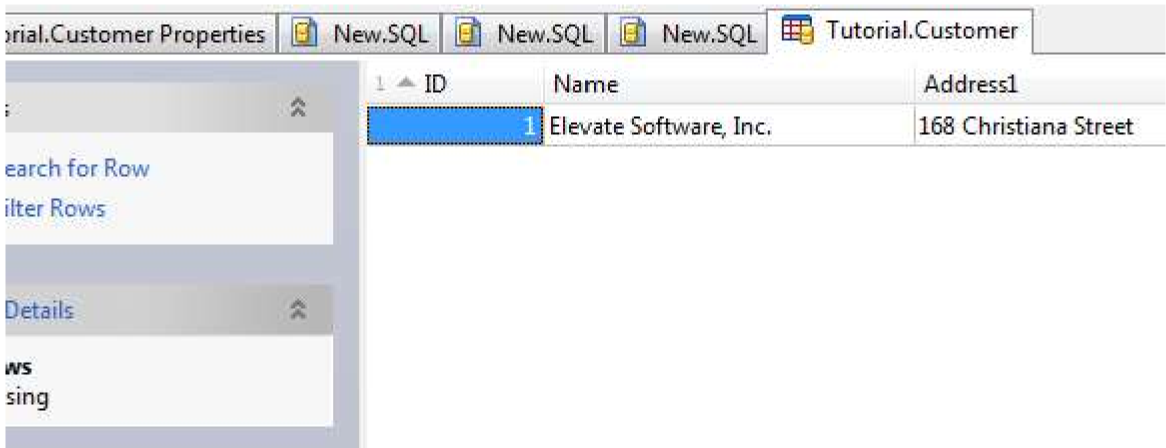
19. Click on the **Customer** table that you just created.



20. Press the **F6** key to make the Properties window the active window, and then click on the **Open Table** link in the Tasks pane.



21. You will now see the row that you just inserted.



You have now successfully created the Tutorial database.

1.4 Migrating a Database

The following steps will guide you through migrating a database from another format to ElevateDB format using the ElevateDB Manager.

1. The migrator modules provided with ElevateDB are:

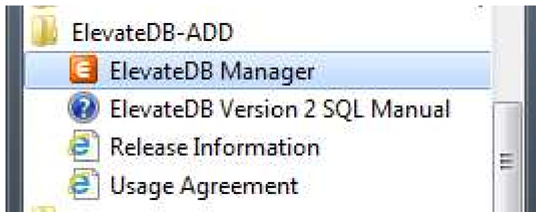
Module	Description
edbmigrate	ElevateDB migrator module
edbmigratedbisam1	DBISAM Version 1.x migrator module
edbmigratedbisam2	DBISAM Version 2.x migrator module
edbmigratedbisam3	DBISAM Version 3.x migrator module
edbmigratedbisam4	DBISAM Version 4.x migrator module
edbmigratebde	BDE (Borland Database Engine) migrator module
edbmigrateado	ADO (Microsoft ActiveX Data Objects) migrator module
edbmigratendb	NexusDB migrator module
edbmigrateads	ADS (Advantage Database Server) migrator module

You can find these migrator modules as part of the ElevateDB Additional Software and Utilities (EDB-ADD) installation in the \libs subdirectory under the main installation directory. There are ANSI and Unicode versions of each of the migrator modules that will work with both ANSI or Unicode sessions, and the ElevateDB Manager will automatically select the correct migrator modules for the session being used.

> **Note**
> You can download the ElevateDB Additional Software and Utilities (EDB-ADD) installation from the Downloads page of the web site.

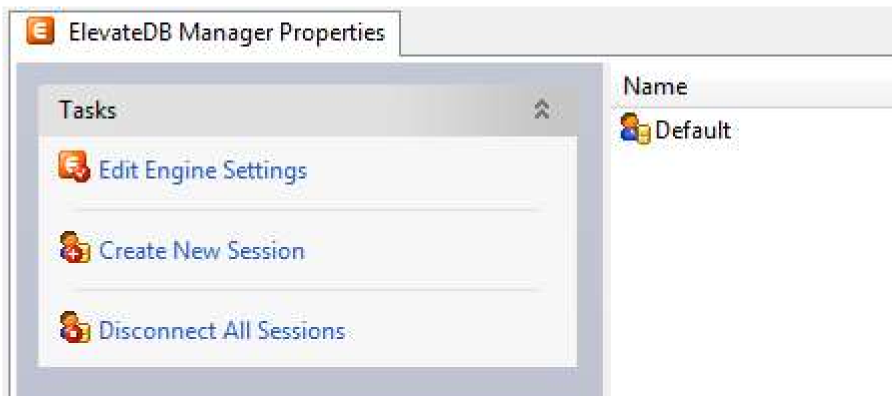
2. Start the ElevateDB Manager (edbmgr.exe) by clicking on the ElevateDB Manager link in the Start menu.

> **Note**
> The ElevateDB Manager is installed with the ElevateDB Additional Software and Utilities (EDB-ADD) installation available from the Downloads page of the web site.

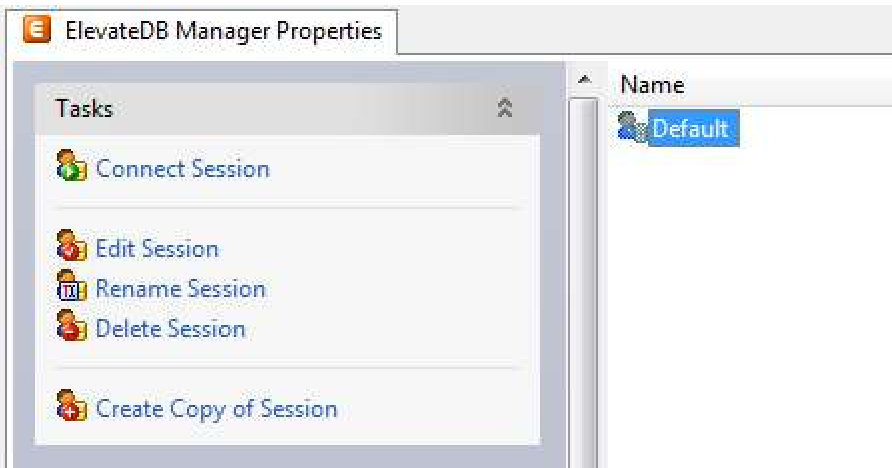


3. Make sure that the session is using the desired character set and configuration file folder (**C:\Tutorial**).

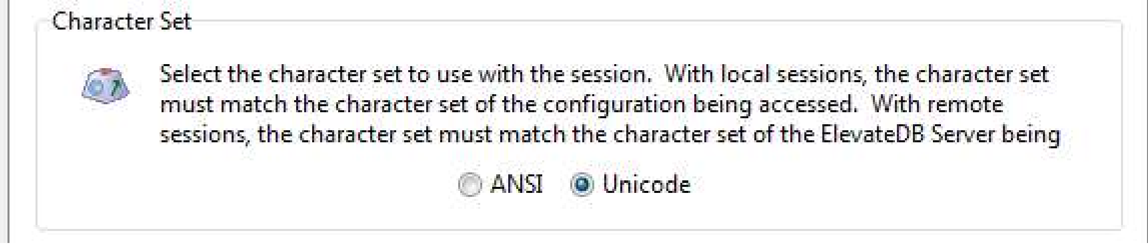
a. Select the **Default** session from the list of available sessions.



b. In the Tasks pane, click on the **Edit Session** link.

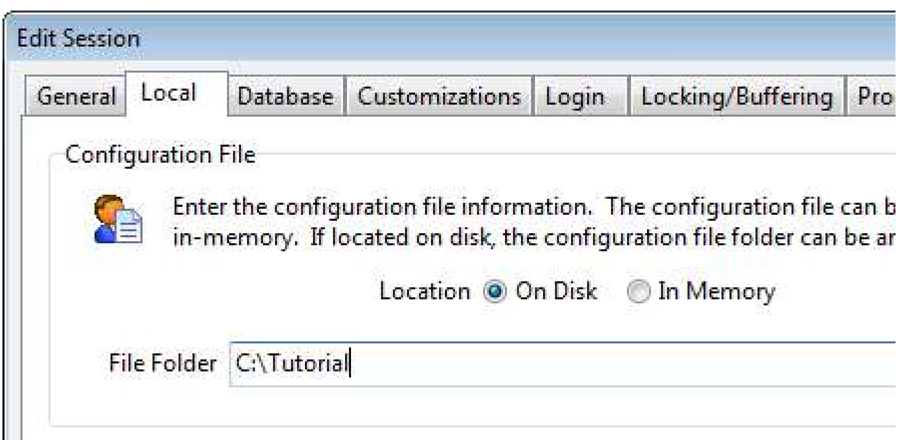


c. On the **General** page of the Edit Session dialog, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



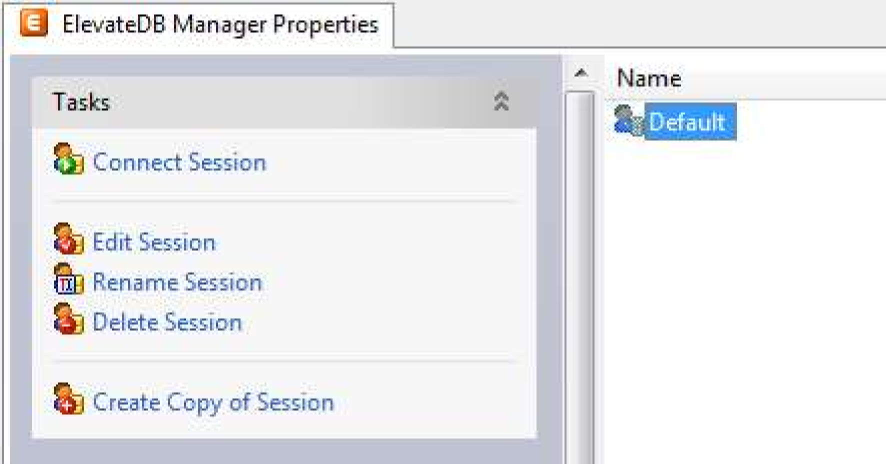
Note
If you're not sure which character set to select and this is the first time using the ElevateDB Manager, then leave the character set at the default of Unicode.

d. On the **Local** page of the Edit Session dialog, make sure that the Configuration File - File Folder is set to the desired folder.

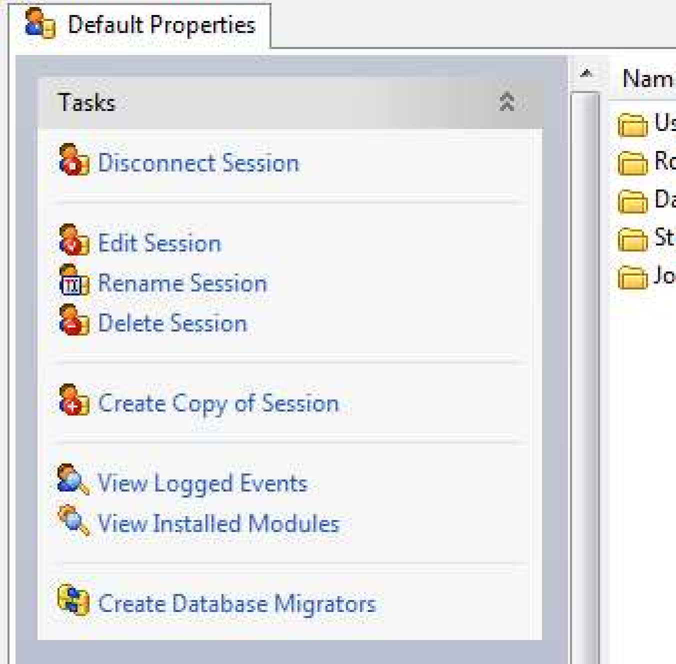


e. Click on the **OK** button.

4. Double-click on the **Default** session in the Properties window in order to connect the session.

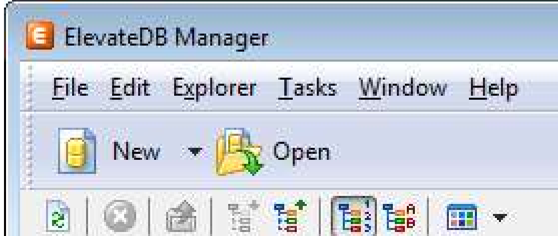


5. In the Tasks pane, click on the **Create Database Migrators** link. This will automatically create all of the database migrators that are shipped with the ElevateDB Manager.



> **Note**
> If the character set of the session is changed in the future (Step 3 above), just re-execute this step in the ElevateDB Manager and the database migrators will be updated so that they use the correct migrator modules that match the character set of the session.

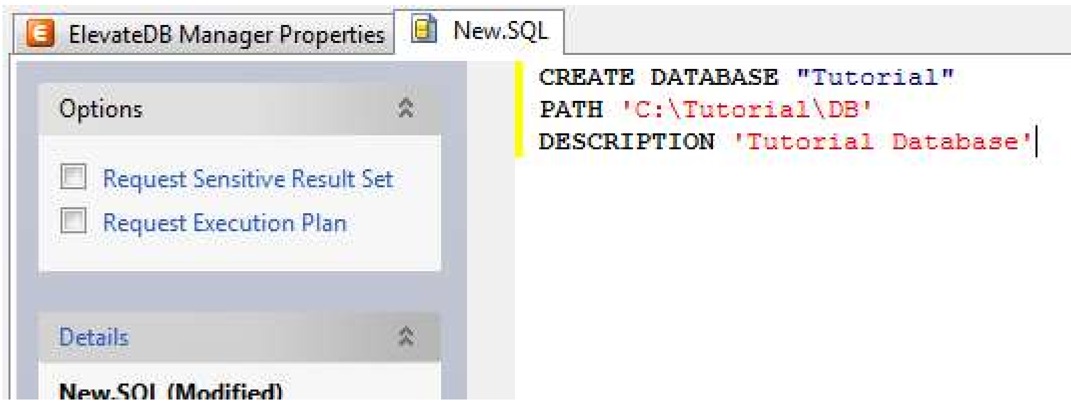
6. Click on the **New** button on the main toolbar.



7. Paste in the following CREATE DATABASE SQL statement in the new SQL window:

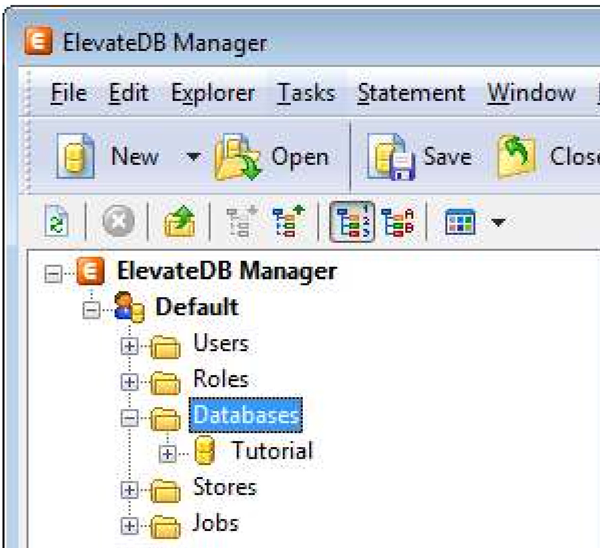
```
CREATE DATABASE "Tutorial"
PATH 'C:\Tutorial\DB'
DESCRIPTION 'Tutorial Database'
```

8. Press the **F9** key to execute the SQL statement.

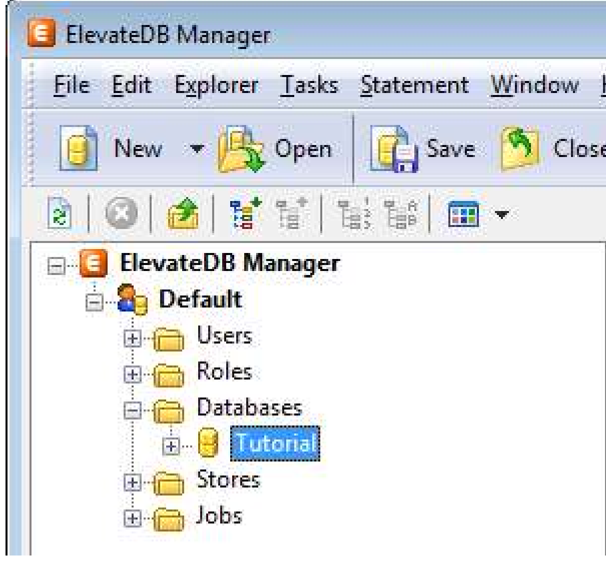


9. Press the **F5** key to refresh the explorer contents for the session.

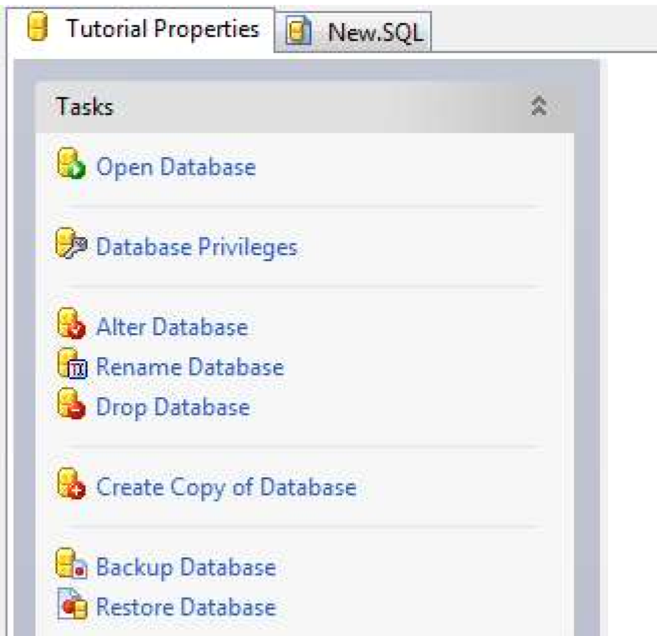
10. Click on the **+** sign next to the **Databases** node in the treeview.



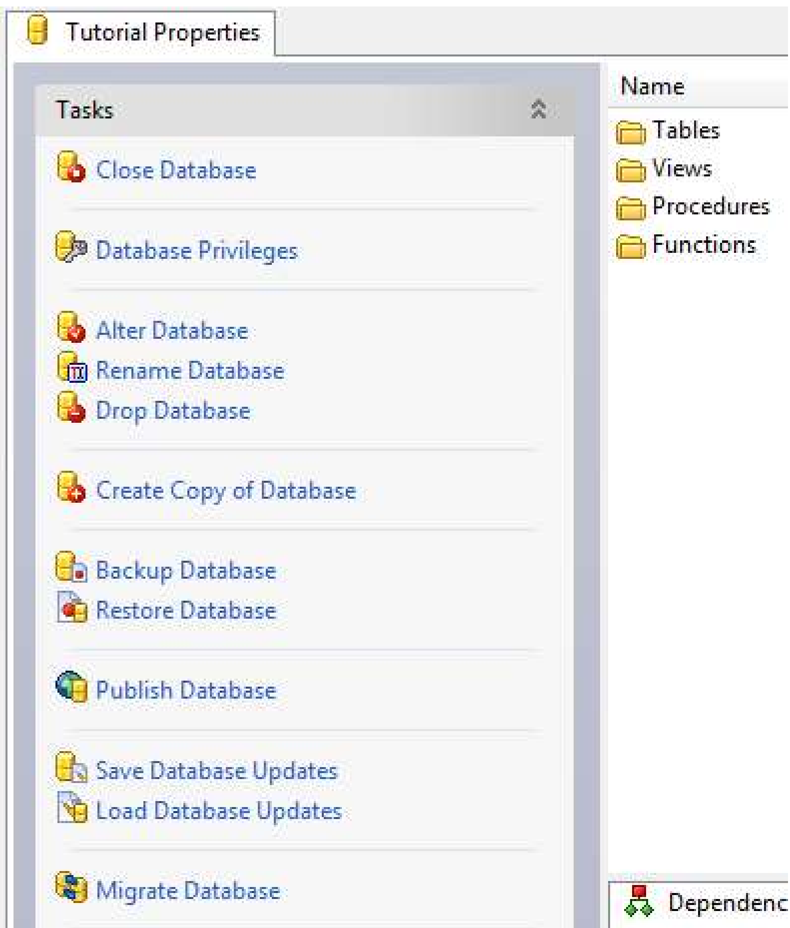
11. Click on the new **Tutorial** database that you just created.



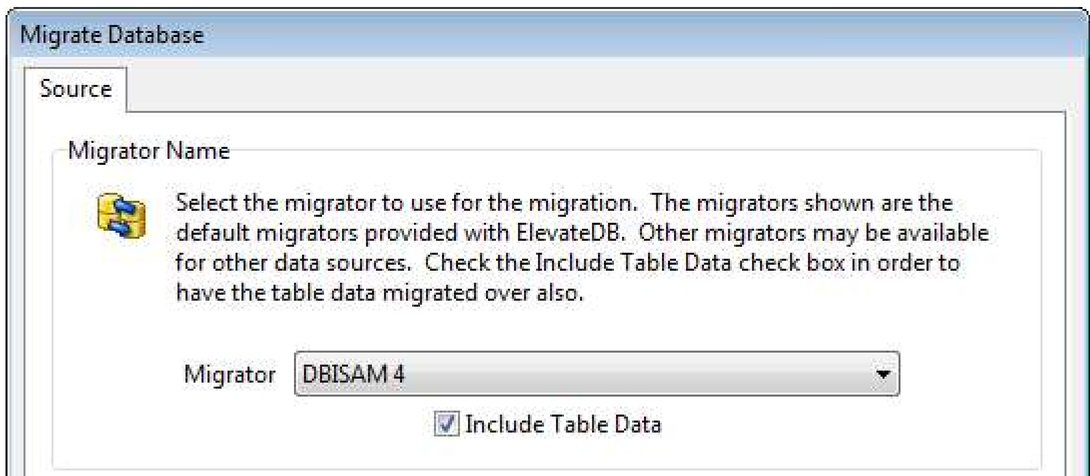
12. Press the **F6** key to make the Properties window the active window, and then click on the **Open Database** link in the Tasks pane.



13. Click on the **Migrate Database** link in the Tasks pane for the database.



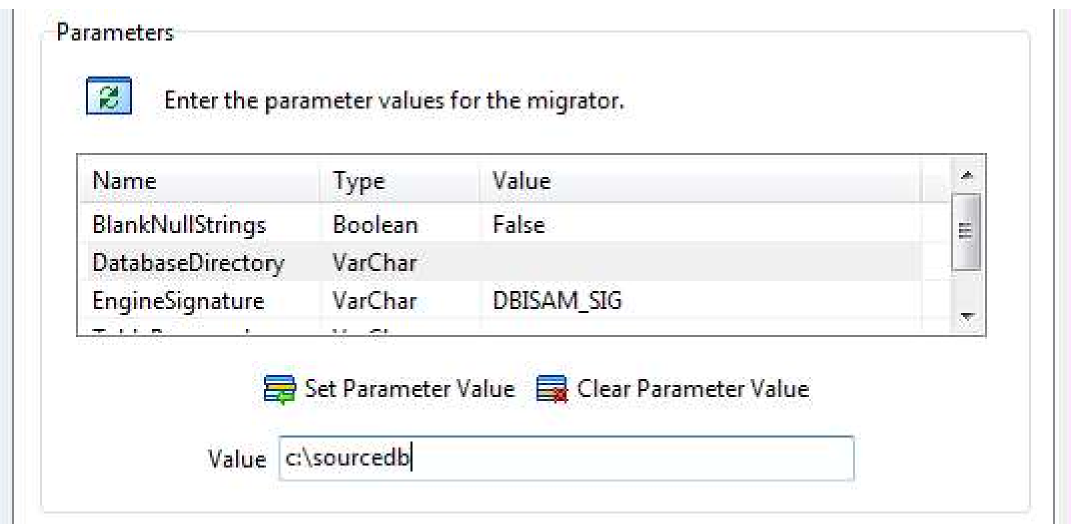
14. Select the desired migrator from the list of migrators.



15. Each migrator will have various parameters that control how the migration process executes, and these parameters are expressed in terms that are easily understood. Usually, at a minimum, the source database name or directory parameter will need to be set. To set the source database parameters:

a. Click on the desired parameter in the list of parameters.

b. Type in the parameter value in the parameter edit control, and click on the **Set Parameter** button.



15. Click on the **OK** button, and the migration process will begin and progress information will be present in the bottom status bar of the ElevateDB Manager.

You have now successfully migrated your database to ElevateDB.

1.5 Starting and Configuring the ElevateDB Server

ElevateDB comes with a version of the ElevateDB Server called **edbsrvr.exe** for Windows and a command-line version of the ElevateDB Server called **edbsrvr** for Linux. The ElevateDB Server for Windows can be run as a normal application (with a GUI) or as a service. The ElevateDB Server for Linux can be run as a normal command-line application or as a service/daemon.

If running the ElevateDB Server as a normal application there is nothing else to do besides start up the ElevateDB Server from the directory in which the ElevateDB Server binary is located. You can find the ElevateDB Server binaries in the \servers\edbsrvr sub-directory under the main installation directory. There are separate subdirectories for 32-bit Windows, 64-bit Windows, and 64-bit Linux binaries.

> **Note**
> Before starting, please make note of the fact that there are two types of "configuration" files being discussed here. The first is the ElevateDB Server configuration file (EDBConfig.EDBCfg, by default). This file contains system-wide user, role, database, store, and job definitions and is used by the ElevateDB engine in all modes of operation. The second is the edbsrvr.ini file (Windows) or edbsrvr.cnf file (Linux), which is used to store the configuration of the ElevateDB Server itself.

Installing the ElevateDB Server as a Windows Service

If you wish to run the ElevateDB Server as a Windows service you must install it as a service by running the ElevateDB Server with the /install command-line switch set. For example, to install the ElevateDB Server as a service using the Run command window under Windows you would specify the following command:

```
edbsrvr.exe /install
```

To uninstall the ElevateDB Server as a Windows service you must run the ElevateDB Server with the /uninstall command-line switch set. For example, to uninstall the ElevateDB Server as a service using the Run command window under Windows you would specify the following command:

```
edbsrvr.exe /uninstall
```

Finally, by default the service will display a "Service installed" dialog box when the service is installed successfully. This is sometimes not desired during installations, and in these cases you can use the /silent command-line switch to suppress the dialog box:

```
edbsrvr.exe /install /silent
```

Installing the ElevateDB Server as a Linux Service

If you wish to run the ElevateDB Server as a Linux service you must install it as a service so that it can be managed by the systemd service manager. This can be accomplished by completing the following steps:

1. Copy the edbsrvr binary from the installation directory:

```
<InstallDir>\servers\edbsrvr\linux64
```

to the following target Linux system's binaries directory:

```
/usr/sbin
```

2. Create the systemd service file for the ElevateDB Server by using the following commands from a terminal window:

```
cd /etc/systemd/system

sudo gedit edbsrvr.service
```

Copy and paste the following information into the edbsrvr.service file being edited:

```
[Unit]
Description=ElevateDB Server

[Service]
Type=forking
ExecStart=/usr/sbin/edbsrvr

[Install]
WantedBy=multi-user.target
```

Save the service file using Ctrl-S and exit the gedit text editor.

3. Reload the service configurations for systemd using the following command in the terminal window:

```
sudo systemctl daemon-reload
```

4. Enable the service so that it will be loaded at boot time:

```
sudo systemctl enable edbsrvr
```

Starting the ElevateDB Server

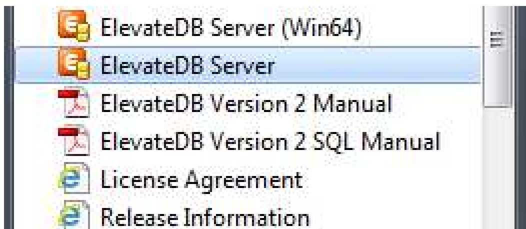
The main difference between starting the ElevateDB Server as a normal application and starting the ElevateDB Server as a service is that the normal application can be started just like any other application

while the service must be started using the operating-system-specific methods for doing so.

Starting the ElevateDB Server as a Normal Application Under Windows

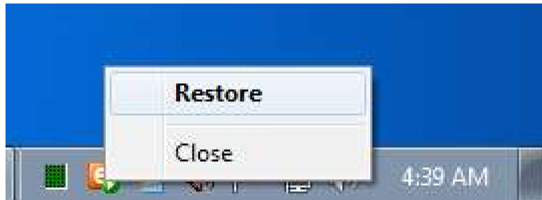
You can start and configure the ElevateDB Server as a normal application by completing the following steps.

1. Start the ElevateDB Server (edbsrvr.exe) by clicking on the ElevateDB Server link in the Start menu.

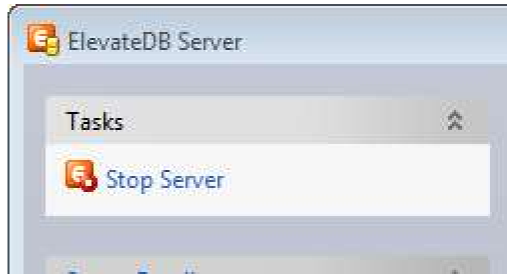


2. Make sure that the server is using the desired character set and configuration file folder (**C:\Tutorial**).

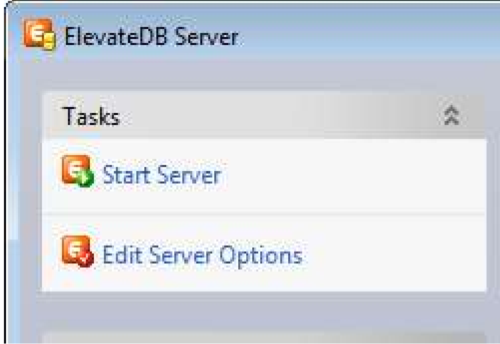
a. In the system tray, right-click on the ElevateDB Server icon to bring up the server menu, and click on the **Restore** option on the server menu.



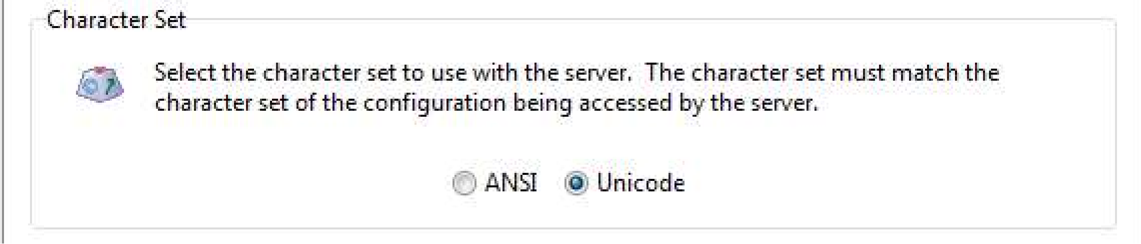
b. In the Tasks pane, click on the **Stop Server** link.



c. In the Tasks pane, click on the **Edit Server Options** link.

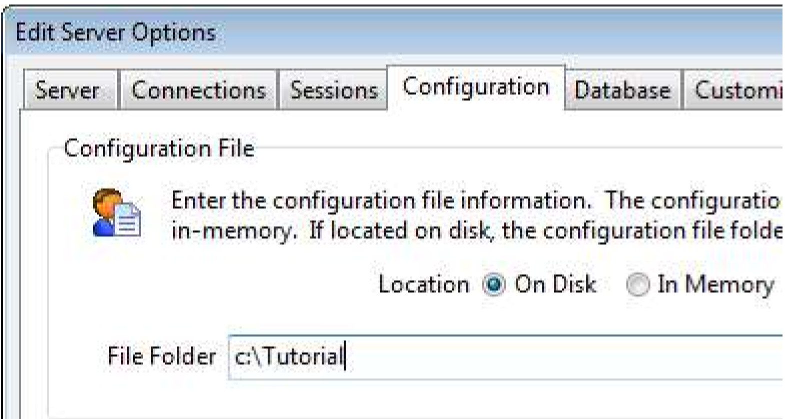


d. On the **Server** page, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



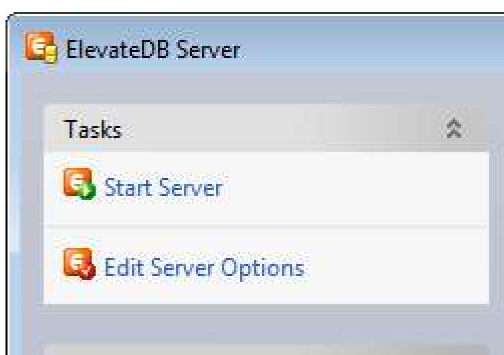
Note
If you're not sure which character set to select and this is the first time using the ElevateDB Server, then leave the character set at the default of Unicode.

e. On the **Configuration** page, make sure that the Configuration File - File Folder is set to the desired folder for the ElevateDB Server configuration file (EDBConfig.EDBCfg).

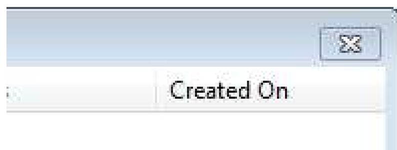


f. Click on the **OK** button.

g. In the Tasks pane, click on the **Start Server** link.



e. Click on the close button in the upper-right-hand corner of the ElevateDB Server window to close the server window.



Starting the ElevateDB Server as a Normal Application Under Linux

You can start and configure the command-line ElevateDB Server as a normal application by completing the following steps.

1. Make sure that the ElevateDB Server command-line server binary (edbsrvr) for Linux is copied into the desired location on the target system.

2. Create the edbsrvr.cnf configuration information file using the following commands from a terminal window:

```
cd /etc

mkdir elevate

cd elevate

mkdir edbsrvr

cd edbsrvr

sudo gedit edbsrvr.cnf
```

Copy and paste the following information into the edbsrvr.cnf file being edited:

```
[Server]
Configuration Folder=<Configuration File Location>
Configuration Name=EDBConfig
Large File Support=1
Maximum Log File Size=1048576
Log Information Events=0
Log Warning Events=1
Log Error Events=1
Catalog Name=EDBDatabase
Configuration File Extension=.EDBCfg
Lock Files Extension=.EDBLck
Log File Extension=.EDBLog
Backup Files Extension=.EDBBkp
Catalog Files Extension=.EDBCat
Table Files Extension=.EDBTbl
Table Index Files Extension=.EDBIdx
Table BLOB Files Extension=.EDBBlb
Temporary Tables Folder=/tmp
Server Name=EDBSrvr
Server Description=ElevateDB Server
Server Run Jobs=1
Server Job Category=
Server Job Retries=10
Server Address=
```

```
Server Port=12010
Server Thread Cache Size=128
Server Encrypted Only=0
Server Session Timeout=60
Server Dead Session Interval=30
Server Dead Session Expiration=30
Server Maximum Dead Sessions=64
Server Authorized Addresses=*<#CR#><#LF#>
Server Blocked Addresses=<#CR#><#LF#><#CR#><#LF#>
Update Files Extension=.EDBUpd
Table Publish Files Extension=.EDBPbl
Signature=edb_signature
Licensed Sessions=4096
Encryption Password=elevatesoft
Configuration In Memory=0
Show User Passwords=0
Character Set=0
Show Database Catalog Information=1
Server Encryption Password=elevatesoft
Cache Modules=1
Buffered File IO=0
Buffered File IO Settings=
Buffered File IO Flush Check Interval=60
Trace=0
Trace File Name=edbtrace.log
Auto-Increment Trace File Name=1
Max Trace File Size=134217728
Max Auto-Increment Trace File Name=64
```

Save the file using Ctrl-S and exit the gedit text editor.

Note
Be sure to modify the Configuration Folder item so that it points to the desired location for the ElevateDB configuration file (EDBConfig.EDBCfg). Please see the Configuration Reference section below for more information on the various settings in the edbsrvr.cnf file.

3. Execute the ElevateDB Server binary using the following commands from a terminal window:

```
cd <Target Location>

./edbsrvr
```

where <Target Location> is the location where the ElevateDB Server binary was copied in step 1.

Starting the ElevateDB Server as a Windows Service

To start the ElevateDB Server as a Windows service, you can use the following command from the command-line:

```
net start edbsrvr
```

Note
In order to start the ElevateDB Server as a Windows service, the ElevateDB Server must have already been installed as a service using the steps in the **Installing the ElevateDB Server as a Windows Service** section above.

Starting the ElevateDB Server as a Linux Service

To start the ElevateDB Server as a Linux service, you can use the following commmands from a terminal window:

```
net start edbsrvr
```

Note
In order to start the ElevateDB Server as a Linux service, the ElevateDB Server must have already been installed as a service using the steps in the **Installing the ElevateDB Server as a Linux Service** section above.

Configuration Reference

On Windows, the ElevateDB Server stores its configuration information in an .ini file that is, by default, located in the following directory:

```
C:\ProgramData\Elevate Software\ElevateDB Server
```

On Linux, the ElevateDB Server stores its configuration information in a .cnf file that is, by default, located in the following directory:

```
/etc/elevate/edbsrvr
```

The name of the .ini or .cnf configuration file is determined by the name of the binary. For example, for the edbsrvr.exe Windows binary, the name of the .ini file would be edbsrvr.ini, and for the edbsrvr Linux binary, the name of the .cnf file would be edbsrvr.cnf.

Note
As of the 2.09 release of ElevateDB, if the ElevateDB Server finds an .ini or .cnf file with the proper name in the same directory as the ElevateDB Server binary, it will use it instead of the .ini or .cnf file in the above directories.

All of the configuration entries in the ElevateDB Server .ini or .cnf configuration files are stored under a section called "Server" (see below for how multiple server instances can change this). Each of the individual configuration entries in this section are as follows:

Configuration Entry	Description
Encryption Password	Specifies the encryption password used by the ElevateDB Server for all file encryption purposes. The ElevateDB Server uses this password for all configuration file, database catalog, and table files encryption (for encrypted tables). ElevateDB uses the Blowfish block cipher encryption algorithm with 128-bit MD5 hash keys for encryption. Please see the Encryption topic for more information.
Signature	Specifies the signature used by the ElevateDB Server for all communications and database access. The ElevateDB Server uses this signature for all configuration file access, table files access, and for all communications with a remote session. A signature is useful for "branding" a server so that it only communicates with sessions that are using a specific signature, rejecting any that do not use that signature. The default value is 'edb_signature'.
Licensed Sessions	Specifies that a certain maximum number of concurrent licensed sessions be allowed. The default value is 4096 sessions. Specifying a lower figure will allow no more than the specified number of sessions to concurrently access the same configuration.
Character Set	Specifies the character set to use for the ElevateDB Server. The valid values are 0 for the ANSI character set, or 1 for the Unicode character set. If the ElevateDB Server is accessing an existing configuration file and the specified character set does not match the character set of the configuration file, then an error message will be displayed and/or logged when the ElevateDB Server is started. The default is 0 (ANSI) under Linux and 1 (Unicode) under Windows.
Configuration Folder	Specifies the path where the ElevateDB Server should look for the configuration file. The configuration file is used to store the information in the Configuration Database in ElevateDB. If the path specified does not exist, then an error will be raised when the ElevateDB Server is started. If the path exists, but the configuration file does not exist in the path, then the ElevateDB Server will create the configuration file as necessary. **Note** It is very important that you do not have more than one instance of the ElevateDB Server using different configuration files and accessing the same database(s). Doing so will cause locking errors. All instances of the ElevateDB Server must use the same configuration file if they will be accessing the same database(s). The default value is the current folder where the server application is running.

Configuration In Memory	Specifies that the configuration file will be "virtual" for all sessions in the ElevateDB Server, and reside only in the process's memory. The default value is 0 (False).
Configuration Name	Specifies the root name (without extension) used by the ElevateDB Server for the configuration file. The extension used for the configuration file is determined by the "Configuration File Extension" configuration entry (below). The location of the configuration file is determined by the "Configuration Folder" configuration entry (above). The default value is 'EDBConfig'.
Configuration File Extension	Specifies the extension to be used for the configuration file. The default value is '.EDBCfg'.
Lock Files Extension	Specifies the extension to be used for the configuration and catalog lock files. The default value is '.EDBLck'.
Log File Extension	Specifies the extension to be used for the log file. The default value is '.EDBLog'.
Maximum Log File Size	Specifies the maximum log file size. The default value is 1048576 bytes.
Log Information Events	Specifies that information events should be logged in the log file. The default value is 1 (True).
Log Warning Events	Specifies that warning events should be logged in the log file. The default value is 1 (True).
Log Error Events	Specifies that error events should be logged in the log file. The default value is 1 (True).
Catalog Name	Specifies the root name (without extension) used by the ElevateDB Server for all database catalog files. The extension used for the catalog files is determined by the "Catalog Files Extension" configuration entry (below). The location of the catalog file is determined by the path designated for the applicable database when the database was created. Please see the CREATE DATABASE topic for more information. The default value is 'EDBDatabase'.
Catalog Files Extension	Specifies the extension to be used for database catalog files. The default value is '.EDBCat'.
Backup Files Extension	Specifies the extension to be used for database backup files. The default value is '.EDBBkp'.
Update Files Extension	Specifies the extension to be used for database update files. The default value is '.EDBUpd'. Update files are used to store logged updates for the purposes of synchronizing two different copies of the same database.
Table Files Extension	Specifies the extension to be used for database table files. The default value is '.EDBTbl'.
Table Index Files Extension	Specifies the extension to be used for database table index files. The default value is '.EDBIdx'.
Table BLOB Files Extension	Specifies the extension to be used for database table BLOB files. The default value is '.EDBBlb'.

Table Publish Files Extension	Specifies the extension to be used for database table publish files. The default value is '.EDBPbl'. Publish files are used to store the logged updates for a table.
Temporary Tables Folder	Specifies where the ElevateDB Server creates any temporary tables that are required for storing query result sets. The default value is the user-specific temporary tables path for the operating system.
Show User Passwords	Specifies whether the server will include user passwords when populating the Users system information table. The default value is 1 (True).
Show Database Catalog Information	Specifies whether the server will include database catalog character set and version information when populating the Databases system information table. The default value is 1 (True). **Note** Setting this configuration item to 0 (False) can significantly improve the performance of the loading of the Databases system information table when there are a lot of databases in a configuration. This is because ElevateDB has to open the database catalog for each database in order to read the character set and version number.
Cache Modules	Specifies whether the server will load external modules once into memory per session and cache them until the session is closed. The default value is 0 (False). **Note** Setting this configuration entry to 1 (True) can result in significant performance improvements. This is especially true for configurations with many different external modules.
Buffered File IO	Specifies whether buffered file I/O should be enabled. The default value is 0 (False). Please see the Buffering and Caching topic for more information on buffered file I/O in ElevateDB.
Buffered File IO Settings	Specifies the buffered file I/O settings for various file specifications. Each setting is a comma-delimited list of values that make up the buffer settings: the file specification, enclosed in double-quotes (") (String), the block size, in KB (Integer), the buffer size, in MB (Integer), the flush age, in seconds (Integer), and a flush to disk flag (Boolean). Please see the Buffering and Caching topic for more information on each of these settings and their default values.

	Note All of the values for each setting must be specified or an error will occur during server startup. Also, due to the way that .ini or .cnf file entries must be specified, multiple settings must be separated with the following literal value instead of actual line feeds: <#CR#><#LF#>
Buffered File IO Flush Check Interval	If buffered file I/O is enabled, specifies how often ElevateDB will check buffered files to see if there are any dirty buffers that need to be written. The default value is 60 seconds. Please see the Buffering and Caching topic for more information on how the buffered file I/O flush check interval works.
Server Name	Identifies the ElevateDB Server to external clients once they have connected to the ElevateDB Server. The default value is 'edbsrvr'. This configuration item is not used for named server instances (see below Multiple Server Instances for more information on named server instances).
Server Description	Used in conjunction with the "Server Name" configuration entry to give more information about the ElevateDB Server to external clients once they have connected to the ElevateDB Server. The default value is 'ElevateDB Server'.
Server Address	Specifies the IP address that the ElevateDB Server should bind to when listening for incoming connections from remote sessions. The default value is blank (""), which specifies that the ElevateDB Server should bind to all available IP addresses.
Server Port	Specifies the port that the ElevateDB Server should bind to when listening for incoming connections from remote sessions. The default value is 12010.
Server Thread Cache Size	Specifies the number of threads that the ElevateDB Server should actively cache for connections. When a thread is terminated in the server it will be added to this thread cache until the number of threads cached reaches this value. This allows the ElevateDB Server to re-use the threads from the cache instead of having to constantly create/destroy the threads as needed, which can improve the performance of the ElevateDB Server if there are many connections and disconnections occurring. The default value is 10.
Server Encryption Password	Specifies the encryption password used by the ElevateDB Server for encrypting all communications with remote sessions. The default value is 'elevatesoft'. ElevateDB uses the Blowfish block cipher encryption algorithm with 128-bit MD5 hash keys for encryption. Please see the Encryption topic for more information.
Server Encrypted Only	Specifies whether all incoming connections from remote sessions should be encrypted or not. If this configuration entry is set to 1 (True), then all incoming connections to the ElevateDB Server that are not encrypted will be rejected with

	an error. The default value is 0 (False). **Note** If you intend to use encrypted connections to an ElevateDB Server over a public network then you should always use a different "Server Encryption Password" configuration entry (above) from the default password.
Server Session Timeout	Specifies how long the ElevateDB Server should wait for a request from a connected remote session before it disconnects the session. This is done to keep the number of concurrent connections at a minimum. Once a session has been disconnected by the ElevateDB Server, the session is then considered to be "dead" until either the remote session reconnects to the session in the server, or the server removes the session according to the parameters specified by the "Server Dead Session Interval", "Server Dead Session Expiration", and "Server Maximum Dead Sessions" configuration entries (below). A remote session may enable pinging in order to prevent the ElevateDB Server from disconnecting the remote session due to this configuration entry. The default value is 180 seconds, or 3 minutes.
Server Dead Session Interval	Specifies how often the ElevateDB Server will poll the disconnected sessions to see if any need to be removed according to the "Server Dead Session Expiration" or "Server Maximum Dead Sessions" configuration entries (below). The default value is 30 seconds.
Server Dead Session Expiration	Specifies how long a session can exist in the ElevateDB Server in a disconnected, or "dead", state before the server removes the session. This is done to prevent a situation where "dead" sessions accumulate from client applications whose network connections were permanently interrupted. **Note** If all of the remote sessions accessing the ElevateDB Server are using pinging, then you should set this configuration entry to the minimum value of 10 seconds so that sessions are removed as soon as they stop pinging the server. The default value is 300 seconds, or 5 minutes.
Server Maximum Dead Sessions	Specifies how many "dead" sessions can accumulate in the ElevateDB Server before the server begins to remove them immediately, irrespective of the "Server Dead Session Expiration" configuration entry (above). If the "Server Maximum Dead Sessions" configuration entry is exceeded, then the server removes the "dead" sessions in oldest-to-

	youngest order until the number of "dead" sessions is at or under the setting for this configuration entry. The default value is 64.
Server Authorized Addresses	Specifies which IP addresses are authorized to access the ElevateDB Server. This is commonly referred to as a "white list". There is no limit to the number of addresses that can be specified, and the IP address entries may contain the asterisk (*) wildcard character to represent any portion of an address. **Note** Due to the way that .ini or .cnf file entries must be specified, multiple addresses must be separated with the following literal value instead of actual line feeds: <#CR#><#LF#>
Server Blocked Addresses	Specifies which IP addresses are not allowed to access the ElevateDB Server. This is commonly referred to as a "black list". There is no limit to the number of addresses that can be specified, and the IP address entries may contain the asterisk (*) wildcard character to represent any portion of an address. **Note** Due to the way that .ini or .cnf file entries must be specified, multiple addresses must be separated with the following literal value instead of actual line feeds: <#CR#><#LF#>
Server Run Jobs	Specifies whether the ElevateDB Server is allowed to schedule and run jobs that are defined in the Configuration Database. If this configuration entry is set to 1 (True, and the default), then the "Server Job Category" configuration entry (below) determines which category of jobs that the server will schedule and run.
Server Job Category	Specifies which job category the ElevateDB Server will schedule and run if the "Server Run Jobs" configuration entry is set to 1 (True). This configuration entry can contain any value, and the default value is blank (''), which indicates that the server can run all job categories. A job category is assigned to each job when it is created via the CREATE JOB DDL statement.
Server Job Retries	Specifies how many times the ElevateDB Server will attempt to execute a given job before disabling the job. The default value is 10.
Trace	Specifies whether tracing is enabled in the ElevateDB Server. If this configuration entry is set to 1 (True), then the ElevateDB Server will log every request/response to/from the server to the trace file name indicated by the "Trace File Name" configuration entry. The default value is 0 (False).

	Warning Do not enable tracing in production without being aware of the consequences. Tracing can generate a large number of trace files that can easily consume large amounts of disk space on a busy server.
Trace File Name	Specifies the trace file name to use when tracing is enabled (see above). The "Max Trace File Size" and "Auto-Increment Trace File Name" configuration entries control how the trace file is managed (see below). The default value is "edbtrace.log". **Note** Do not specify a path in the trace file name. The ElevateDB Server will use the system-defined temporary files directory for storing the trace files to ensure that it has proper write permissions.
Max Trace File Size	Specifies the maximum allowed size of the trace file. The default value is 128MB (134217728 bytes).
Auto-Increment Trace File Name	Specifies how to handle the trace file when the maximum allowed trace file size is reached or exceeded. If this configuration entry is set to 0 (False), then the trace file name will renamed with a ".bak" extension and a new trace file will be started with the value of the "Trace File Name". If this configuration entry is set to 1 (True), then the trace file name will be renamed to <Trace File Name> + <Auto-Incrementing Number> + Trace File Name Extension> (starting at 1 for the <Auto-Incrementing Number> portion of the trace file name), and a new trace file will be started with the value of the "Trace File Name". The default value is 0 (False).
Max Auto-Increment Trace File Name	Specifies the maximum number of auto-incrementing trace files that will be created before the auto-incrementing trace file name is reset to 1. This value, in conjunction with the Max Trace File Size setting, determines the maximum amount of disk space that will be used when using auto-incrementing trace files. The default value is 64.

Multiple Server Instances

Multiple instances of the ElevateDB Server can be run on the same physical machine through named server instances. Named server instances are simply instances of the ElevateDB Server that were executed using two special command-line switches:

```
edbsrvr.exe /name=<Server Instance Name> /desc=<Server Instance Description>
```

Named server instances use the passed name and description to provide the name of the ElevateDB Server instance, as well as the description. The name parameter is also used to determine which section of

the edbsrvr.ini (Windows) or edbsrvr.cnf (Linux) file is used for configuration purposes. Instead of just the normal "Server" section being used, the section is named using the provided server name. For example, if the named server instance is called "MyServer", then the section where the configuration is stored will be the following:

```
[Server_MyServer]
```

The description parameter, if also specified, is immediately written to the named server instance section. All other configuration options described above in the Configuration Reference must be modified by running the ElevateDB Server as a normal application on Windows and using the Edit Server Options in the ElevateDB Server's user interface. You can run the ElevateDB Server as a normal application on Windows in order to modify the configuration of a named server instance. For example, to modify the MyServer configuration you would use the following from the command-line:

```
edbsrvr.exe /name=MyServer"
```

In order to use a named server instance as a Windows service, the /name parameter must be specified during the installation of the service. For example, if the named server instance is called "MyServer", then the service installation would be accomplished using the following from the command-line:

```
edbsrvr.exe /install /name=MyServer /desc="My Server"
```

When you want to start the named server instance as a Windows service, you would simply just use the following from the command-line:

```
net start MyServer
```

The following example shows how you would install two ElevateDB Server named server instances as Windows services, and then start them:

```
edbsrvr.exe /install /name=MyFirstServer /desc="My First Server"

edbsrvr.exe /install /name=MySecondServer /desc="My Second Server"

net start MyFirstServer

net start MySecondServer
```

Warning
You will need to verify that the port being used by each named server instance is unique, or one or more named server instance will not start due to a port conflict. As mentioned above, you can use the ElevateDB Server run as a normal application to modify the configuration of any named server instance.

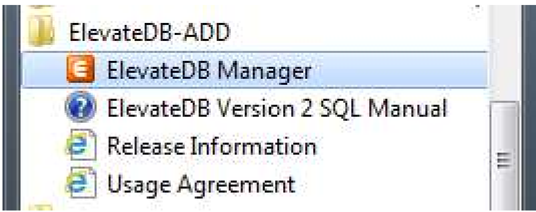
1.6 Creating a Client-Server Database

The following steps will guide you through creating the Tutorial database using the ElevateDB Manager and ElevateDB Server.

> **Note**
> It is assumed that you have already configured and started the ElevateDB Server using the steps outlined in the Starting and Configuring the ElevateDB Server topic.

1. Start the ElevateDB Manager (edbmgr.exe) by clicking on the ElevateDB Manager link in the Start menu.

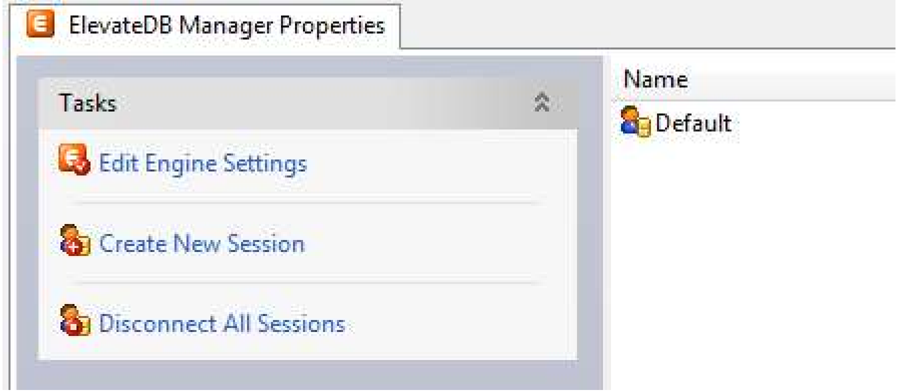
> **Note**
> The ElevateDB Manager is installed with the ElevateDB Additional Software and Utilities (EDB-ADD) installation available from the Downloads page of the web site.



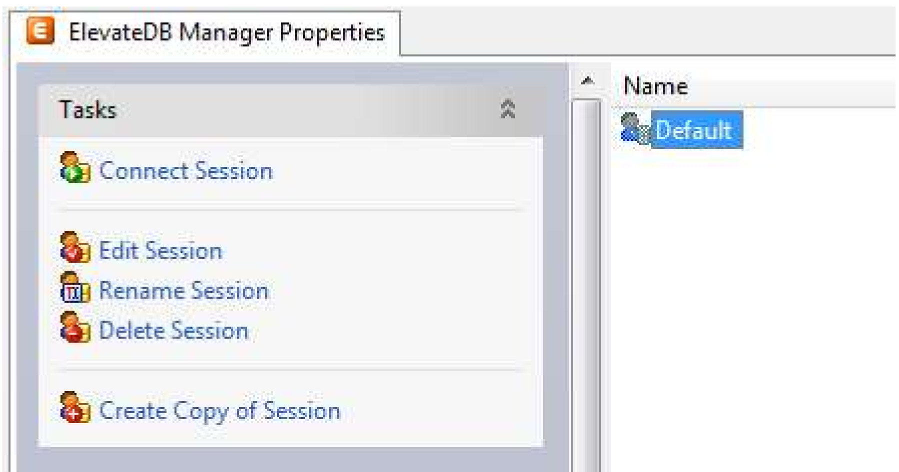
2. Make sure that the session is using the correct session type (**Remote**) and desired character set.

> **Note**
> The character set for the session must match the character set being used by the ElevateDB Server being accessed. Using a different character set will result in you not being able to connect to the ElevateDB Server.

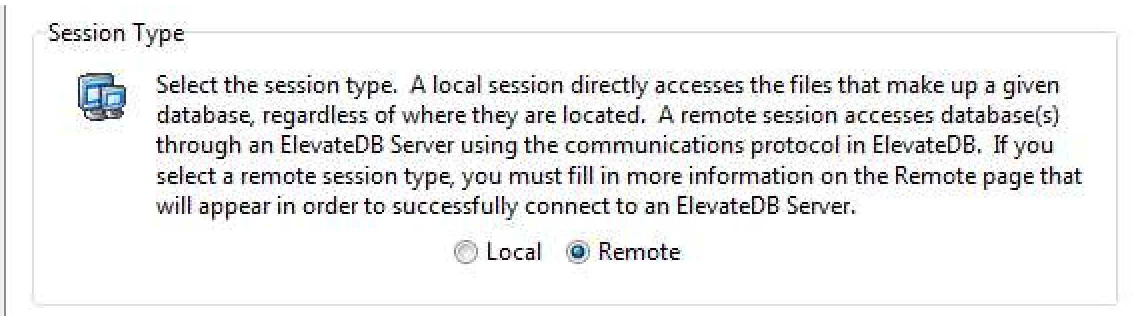
a. Select the **Default** session from the list of available sessions.



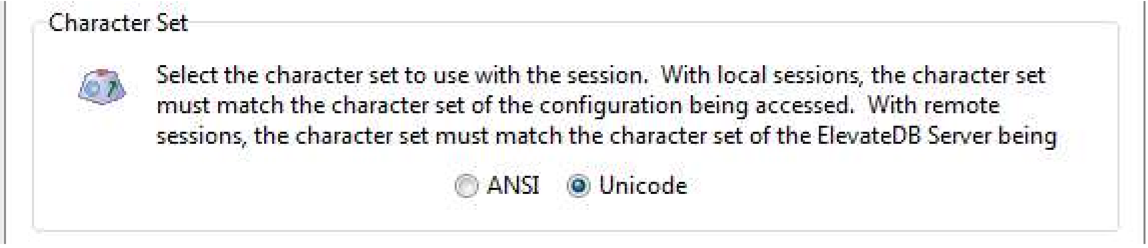
b. In the Tasks pane, click on the **Edit Session** link.



c. On the **General** page of the Edit Session dialog, make sure that the Session Type is set to **Remote**.



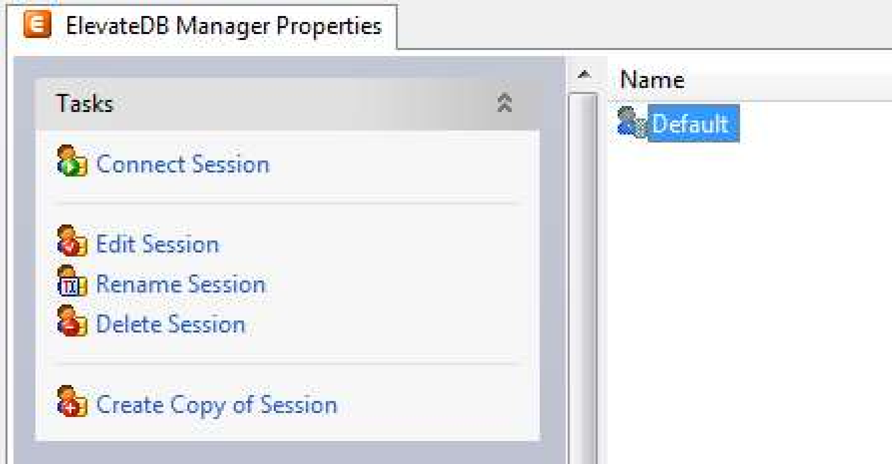
d. On the **General** page of the Edit Session dialog, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



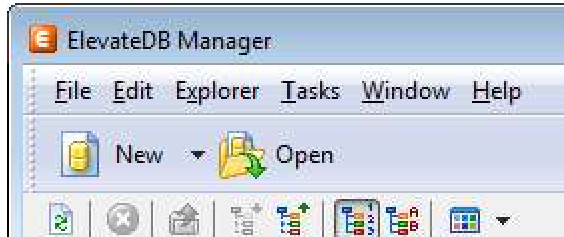
Note
If you're not sure which character set to select and this is the first time using the ElevateDB Manager, then leave the character set at the default of Unicode.

e. Click on the **OK** button.

3. Double-click on the **Default** session in the Properties window in order to connect the session.



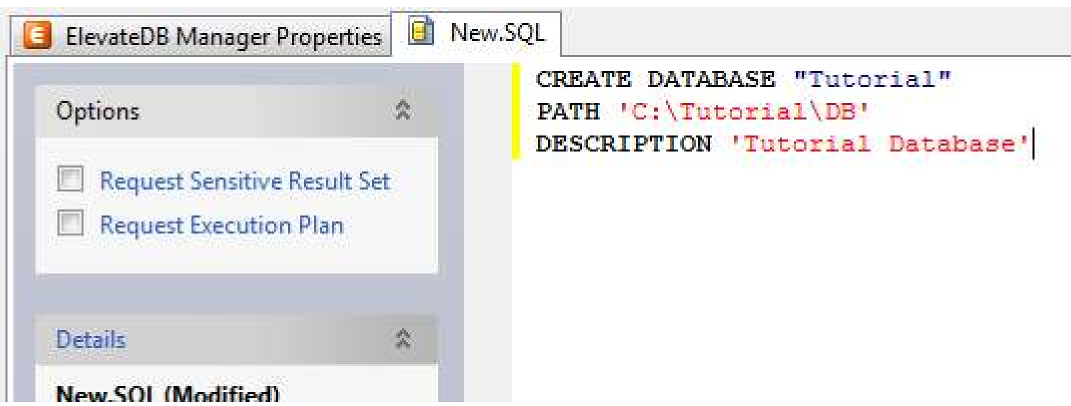
4. Click on the **New** button on the main toolbar.



5. Paste in the following CREATE DATABASE SQL statement in the new SQL window:

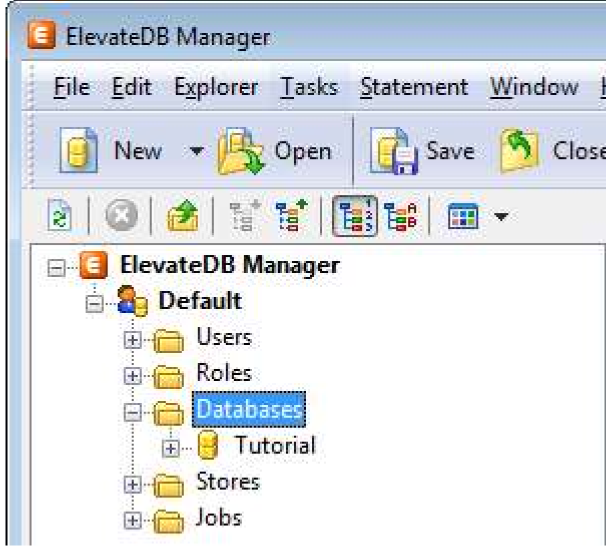
```
CREATE DATABASE "Tutorial"
PATH 'C:\Tutorial\DB'
DESCRIPTION 'Tutorial Database'
```

6. Press the **F9** key to execute the SQL statement.

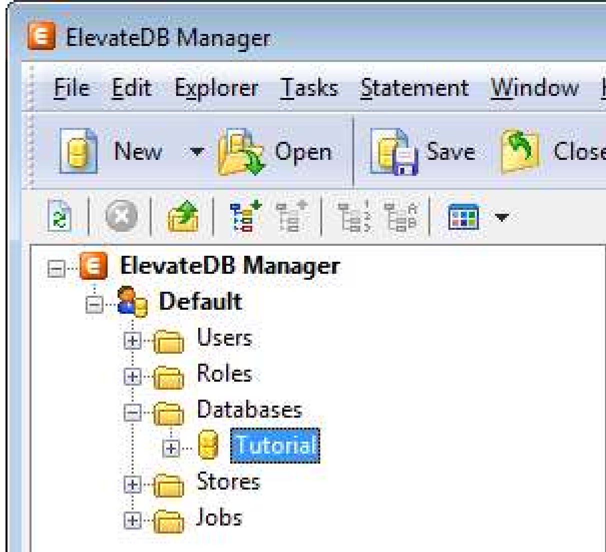


7. Press the **F5** key to refresh the explorer contents for the session.

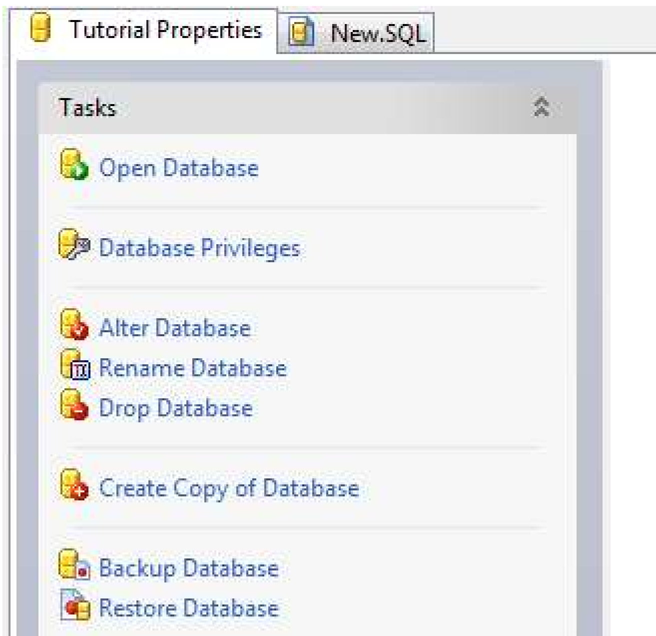
8. Click on the **+** sign next to the **Databases** node in the treeview.



9. Click on the new **Tutorial** database that you just created.



10. Press the **F6** key to make the Properties window the active window, and then click on the **Open Database** link in the Tasks pane.



11. Click on the **New.SQL** tab to bring forward the SQL window.

12. Paste in the following CREATE TABLE SQL statement. If you are using a Unicode session (see Step 2 above), then you should use the Unicode version of the CREATE TABLE statement. If you are using an ANSI session, then you should use the ANSI version of the CREATE TABLE statement:

ANSI

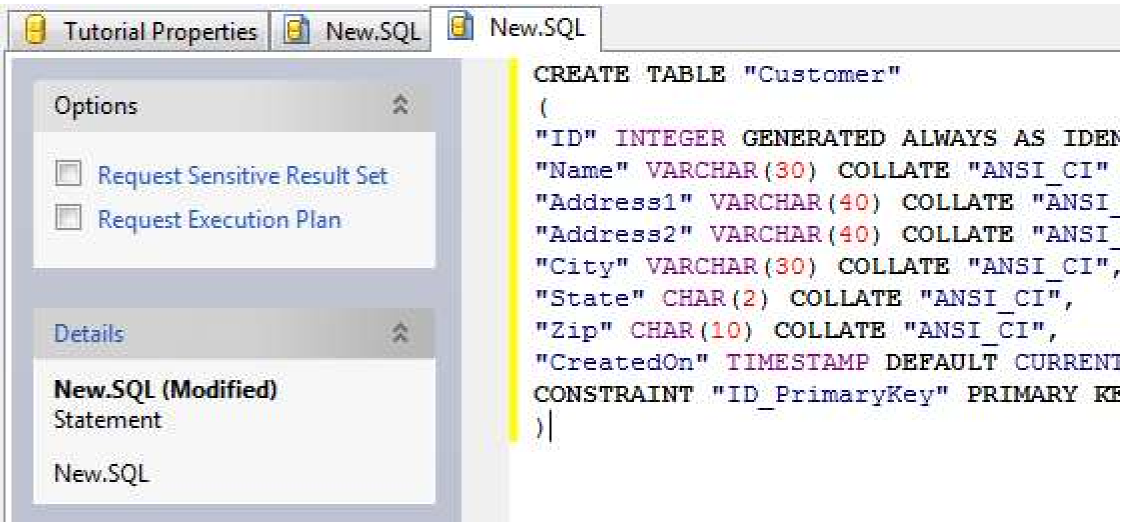
```
CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "ANSI_CI" NOT NULL,
"Address1" VARCHAR(40) COLLATE "ANSI_CI",
"Address2" VARCHAR(40) COLLATE "ANSI_CI",
"City" VARCHAR(30) COLLATE "ANSI_CI",
"State" CHAR(2) COLLATE "ANSI_CI",
"Zip" CHAR(10) COLLATE "ANSI_CI",
"CreatedOn" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID")
)
```

Unicode

```
CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "UNI_CI" NOT NULL,
"Address1" VARCHAR(40) COLLATE "UNI_CI",
"Address2" VARCHAR(40) COLLATE "UNI_CI",
"City" VARCHAR(30) COLLATE "UNI_CI",
"State" CHAR(2) COLLATE "UNI_CI",
"Zip" CHAR(10) COLLATE "UNI_CI",
"CreatedOn" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

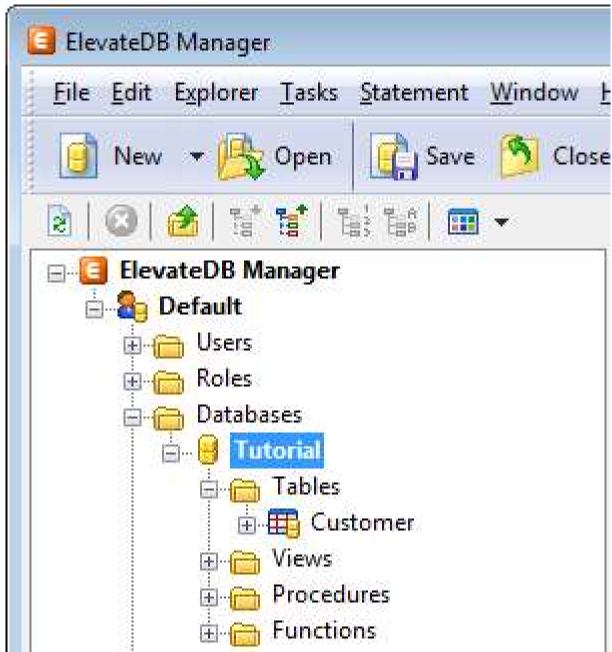
```
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID")
)
```

13. Press the **F9** key to execute the SQL statement.



14. Press the **F5** key to refresh the explorer contents for the session.

15. The table should now show up in the list of tables for the Tutorial database.



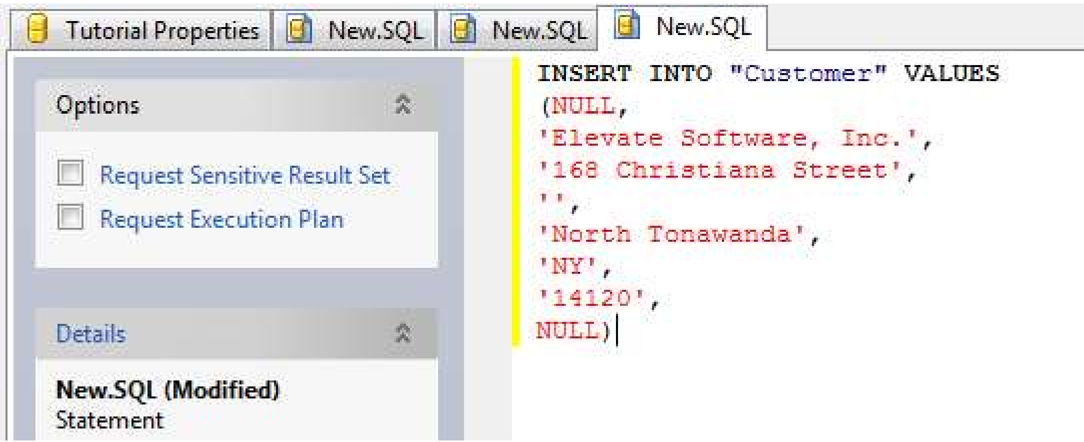
16. Click on the **New.SQL** tab to bring forward the SQL window.

17. Paste in the following INSERT SQL statement:

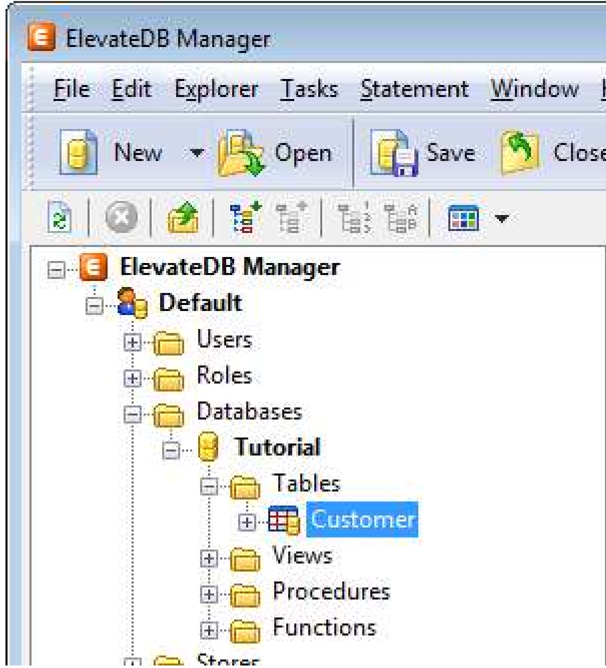
```
INSERT INTO "Customer" VALUES
(NULL,
```

```
'Elevate Software, Inc.',
'168 Christiana Street',
'',
'North Tonawanda',
'NY',
'14120',
NULL)
```

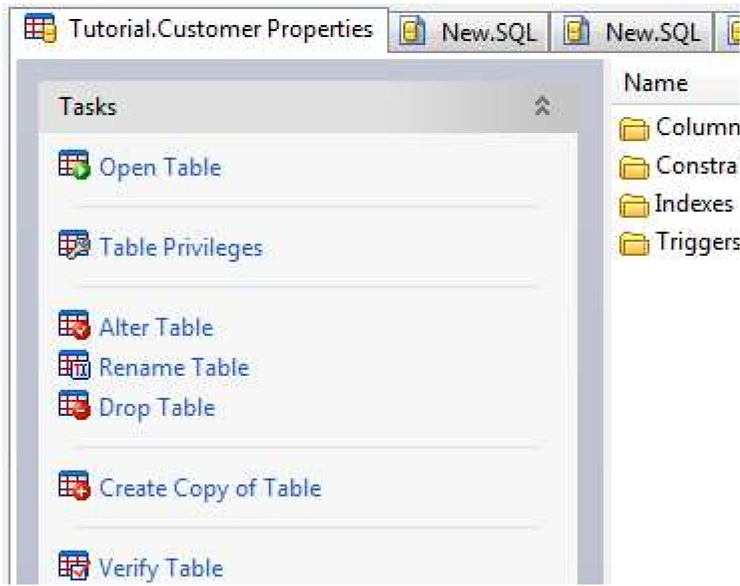
18. Press the **F9** key to execute the SQL statement.



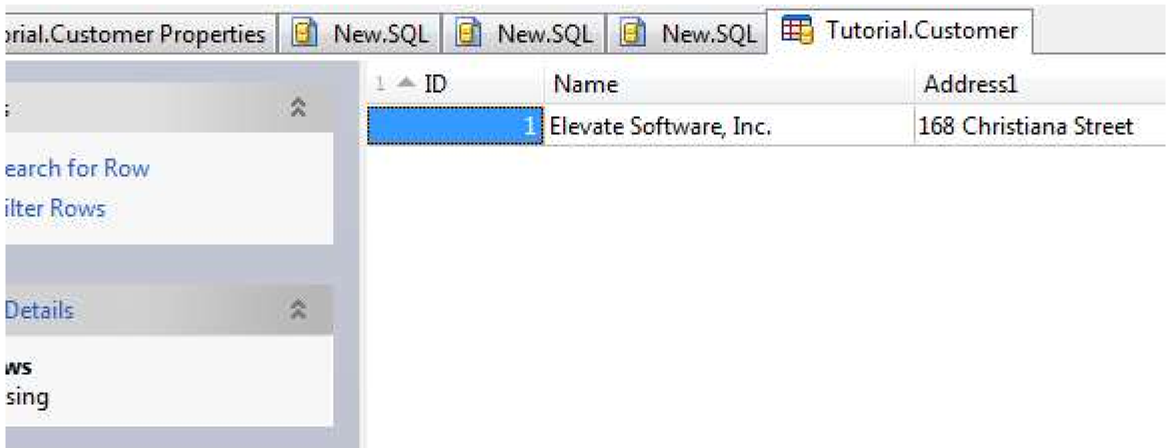
19. Click on the **Customer** table that you just created.



20. Press the **F6** key to make the Properties window the active window, and then click on the **Open Table** link in the Tasks pane.



21. You will now see the row that you just inserted.



You have now successfully created the Tutorial database.

1.7 Internationalization

Character Sets Supported

ElevateDB supports the Windows ANSI and Unicode character sets. The Windows ANSI character set is based upon an applicable locale's code page and is an 8-bit character set. The Unicode character set is the standard ISO 10646 character set and is a 16-bit character set.

ElevateDB supports these two character sets as both an engine-level and session-level option, with the session-level option inheriting the engine-level setting, by default. A session can only access configuration files and databases that use the same character set that is specified for the engine/session. However, you can mix sessions using different character sets within the same application or server.

Character Encodings

At this time, ElevateDB does not completely support using double-byte character set encodings (DBCS) with the ANSI character set, nor does it completely support using the UTF-16 encoding (Unicode surrogate pairs) with the Unicode character set. Specifically, operators such as the LIKE operator may not work properly. This means that one should assume straight single-character comparisons for both the ANSI character set and the Unicode character set, effectively making UCS-2 the only Unicode encoding completely supported.

This will change in the near future and complete DBCS and UTF-16 support will be made available.

Collations

ElevateDB supports table column and indexed column collations. Collations specified for a table column affect all column comparisons. When a table column is indexed, by default the table column collation is used for the index. If the table column collation is overridden when the index is created using the CREATE INDEX or CREATE TEXT INDEX statement, then the new indexed column collation in conjunction with the a specific SQL JOIN or WHERE expression determines whether the index column can be used by the ElevateDB SQL optimizer to optimize the expression by using the index. Please see the Optimizer topic for more information on how collations affect index selection in the optimizer. Also, the index column collation affects whether the index can be used to return a sensitive result set cursor for SELECT statements with an ORDER BY. Please see the Result Set Cursor Sensitivity topic for more information.

To specify a collation for a table or index column, you must use the COLLATE clause:

```
COLLATE <CollationName>
```

The COLLATE clause is supported for any CHAR, VARCHAR, or CLOB column. In addition, the collation name can be specified with additional modifiers. Each of the modifiers is added to the collation name using the underscore as a separator

Collation Modifier	Description

CI	Specifies that any case differences between characters are ignored.
AI	Specifies that accents are ignored and only base characters are considered.
KI	Specifies that any Japanese hiragana/katakana character differences are ignored.
WI	Specifies that the equivalent single-byte and double-byte character should be considered equal, even though they are encoded differently.

For example, to specify a case-insensitive English (United States) collation for a column, you would use the following SQL:

```
COLLATE "ENU_CI"
```

The available collations in ElevateDB are dynamic and reflect the available installed locales in the operating system. In addition, ElevateDB includes one default collation:

ANSI   ANSI (Binary)
or
UNI   Unicode (Binary)

depending upon whether the engine and/or session is using the ANSI or Unicode character set.

Warning
Linux implementations of ElevateDB only support the default ANSI and UNI collations, and any attempts to use an existing database that references other collations available on Windows only will result in errors.

This default collation uses the ANSI or Unicode ordinal values for each character comparison. Upper-casing and lower-casing is done using US and Western European casing rules.

Note
The AI (accent-insensitive), KI (Kana-insensitive), and WI (width-insensitive) collation modifiers described above are not applicable to the default ANSI or Unicode collations.

You may query the system-created Configuration database to get a list of available collations. All available collations in the operating system are stored in the Collations table in the Configuration database. For example, the following SELECT statement returns all of the available collations:

```
SELECT * FROM Configuration.Collations
```

Note
You will receive an error if you try to open a database that has table columns or index columns that reference a collation that is not available on the operation system being used.

1.8 Identifiers

What Constitutes an Identifier

An identifier is the name of any object that resides in the catalog for a database, such as a table, column, constraint, index, trigger, stored procedure, function, etc. as well as any predefined ElevateDB objects such as a collation, module, etc.

Valid Identifiers

Identifiers may contain any non-symbolic or non-punctuation character in the ANSI character set, if using the ANSI character set with the current engine/session, or the lower 256 characters of the Unicode character set, if using the Unicode character set with the current engine/session. Identifiers cannot begin with a digit and must begin with a valid alphabetic character or underscore (_). Identifers may contain undercores (_), dashes (-), pound signs (#), and right and left parentheses (()), in addition to alpha-numeric characters. Identifiers can include spaces also, but if they do then the identifier must be enclosed in double-quotes (""). For example, the following SELECT statement contains a normal table identifier:

```
SELECT * FROM MyTable
```

However, the following SELECT statement contains a table name with embedded spaces:

```
SELECT * FROM "My Table"
```

and, therefore, must be enclosed in double-quotes.

Table Qualifiers

Table identifiers can optionally be prefaced with a database identifier and/or schema identifier. For example, the following SELECT statement contains a table identifier that has been prefaced with a database identifier:

```
SELECT * FROM MyDatabase.MyTable
```

Note
If a database identifier is specified, but a schema identifier is not specified, then ElevateDB assumes the use of the default schema Default for the database.

Also, a table identifier can be prefaced with just a schema identifier:

```
SELECT * FROM Default.MyTable
```

Finally, a table identifier can be prefaced with both a database identifier and a schema identifier:

```
SELECT * FROM MyDatabase.Default.MyTable
```

See the System Information topic for more information on databases and schemas, including the default and information schemas.

1.9 NULLs

Definition of a NULL

NULL is the term used in the SQL standard and database management systems to describe a value that is not known. It is important to note that while a NULL column value is an unknown value, it does still have a type. There is no such thing as an unknown column value that also has an unknown type. The only time a NULL value can also have an unknown type is in the case of a NULL constant:

```
NULL
```

NULL Assignments

It is important to remember that a NULL is not the same as a zero value with numeric columns such as INTEGER columns, and that a NULL is not the same as an empty string value with string columns such as VARCHAR columns. Assigning any non-NULL value to a column will result in the column value being known and not NULL. Likewise, assigning a NULL to a column is the only way to set a column's value to NULL. For example, the following UPDATE statement will result in a State column value that is empty, but still not NULL:

```
UPDATE Customer SET State=''
```

In order to set the State column to NULL, you would need to use this UPDATE statement:

```
UPDATE Customer SET State=NULL
```

NULLs and Operators

The primary rule to remember with NULLs is that any operator that uses a NULL as an operand will result in a NULL. In other words, it is impossible for any operator using an unknown value to return a known value. For example, in the following UPDATE statement any rows with a NULL in the Quantity column will still have a NULL in the Quantity column after the statement is executed:

```
UPDATE Orders SET Quantity=(Quantity + 10)
```

This is also the case with aggregate functions like MIN, MAX, SUM, COUNT, etc. that operate on an individual column. Any row values having a NULL in the column being operated on will be ignored for the purposes of the operation. In addition, any aggregate functions that operate on an individual column will also return a NULL value as the result of the operation if no rows are visited while executing the aggregate function. This can be the case when the JOIN or WHERE clauses filter the available rows so that the aggregate operation is not executed.

Note
There is a "special" case, however, with respect to the boolean AND and OR operators. The following examples illustrate these special cases:

```
FALSE AND NULL results in FALSE
TRUE OR NULL results in TRUE
```

These results occur because in each case the NULL, or unknown value, is irrelevant to the outcome of the operation. ElevateDB already knows enough to be able to give an accurate answer, and it simply wouldn't matter if the NULL was actually a known value. In each case the result would be the same even with a known value instead of the NULL.

Preventing NULLs in Columns

In order to prevent NULLs from being allowed in a given column, you may use the NOT NULL check constraint on a column when creating it via a CREATE TABLE or ALTER TABLE statement. This will prevent any row from being added or updated with a NULL specified for the column.

NULLs and Primary and Unique Key Constraints

Primary key constraints require that all of the columns that make up the constraint contain a non-NULL value, irregardless of any NOT NULL check constraints defined for the column(s).

If all of the columns that make up a unique key constraint contain NULLs, then the unique key constraint is not enforced. In other words, unique key constraints allow multiple rows with NULLs in the unique key columns. Only known values are used to enforce the unique key constraint.

1.10 User Security

ElevateDB supports most of the SQL security model that is specified in the SQL 2003 standard. This includes support for users and roles (authorizations), as well as the granting and revoking of privileges on database objects. However, ElevateDB only allows users that have been granted the special system-created Administrators role to create, alter, or drop users and roles, or grant or revoke privileges for either. The SQL statements that apply to user security in ElevateDB are as follows:

CREATE USER
ALTER USER
DROP USER
RENAME USER
CREATE ROLE
ALTER ROLE
DROP ROLE
RENAME ROLE
GRANT ROLES
REVOKE ROLES
GRANT PRIVILEGES
REVOKE PRIVILEGES

Users and Roles

ElevateDB supports the creation of both users and roles, and both are considered authorizations in that they can be granted privileges on database objects. Roles can be granted to users, which allows for easier administration of the privileges for a given application and/or database by organizing the granting and revoking of privileges based upon the tasks required by a certain group of users. For example, in a point-of-sale application there would possibly be the following roles:

```
Cashiers
Managers
```

which would be created using the following SQL statements:

```
CREATE ROLE Cashiers DESCRIPTION 'Store Cashiers'

CREATE ROLE Managers DESCRIPTION 'Store Managers'
```

One could then grant privileges on the various database objects to these roles instead of directly to the users like this:

```
GRANT SELECT, INSERT ON Transactions
TO Cashiers

GRANT SELECT, INSERT, UPDATE, DELETE ON Inventory
TO Managers
```

Finally, granting these roles to new users can be done using the GRANT ROLE statement:

```
CREATE USER Jenny PASSWORD '34IJT199'
DESCRIPTION 'Jenny Myers'

GRANT Cashiers TO Jenny
```

Default Users and Roles

There are two system-created users and two system created roles in every ElevateDB configuration. They are as follows:

System User

The System user is created automatically for each new ElevateDB configuration, and cannot be dropped or altered. The System user is used in the following contexts:

Context	Description
Job Execution	By default, the System User is used as the current user within the actual execution context of a job. However, this can be changed on a per-job basis in order to use a different user for the execution context.
Routine Execution	Once ElevateDB verifies that the current user has the proper execution privileges for a function or procedure, the System User is used as the current user within the actual execution context of the function or procedure. This is to allow for the routine to access resources that the executing user may or may not have access to.
Triggers	Triggers always use the System user for the current user within the execution context of the trigger. This is to allow for the trigger to access resources that the executing user may or may not have access to. This is especially useful, for example, in situations where transactions cause inventory to be updated but you don't want the person entering the transactions into the transactions table to have access to the inventory table.
Foreign Key Constraints	Any tables referenced in a foreign key constraint are opened using the System User. This is necessary because not all users may have the proper privileges required to open up tables that have been declared the target of a foreign key constraint.

Administrator User

The Administrator user is created automatically for each new ElevateDB configuration, and can be dropped and altered. The default password for the Administrator user is:

```
EDBDefault (case-sensitive)
```

The default Administrator user is automatically granted the system-created Administrators role (see below).

Administrators Role

The Administrators role is created automatically for each new ElevateDB configuration, and cannot be dropped or altered. Only users that have been granted the Administrators role can execute the following statements:

CREATE USER
ALTER USER
DROP USER
RENAME USER
CREATE ROLE
ALTER ROLE
DROP ROLE
RENAME ROLE
GRANT ROLES
REVOKE ROLES
GRANT PRIVILEGES
REVOKE PRIVILEGES
CREATE DATABASE
ALTER DATABASE
DROP DATABASE
RENAME DATABASE
CREATE JOB
ALTER JOB
DROP JOB
RENAME JOB
CREATE STORE
ALTER STORE
DROP STORE
RENAME STORE
CREATE MODULE
ALTER MODULE
DROP MODULE
RENAME MODULE
CREATE MIGRATOR
ALTER MIGRATOR
DROP MIGRATOR
RENAME MIGRATOR
CREATE TEXT FILTER
ALTER TEXT FILTER
DROP TEXT FILTER
RENAME TEXT FILTER
CREATE WORD GENERATOR
ALTER WORD GENERATOR
DROP WORD GENERATOR
RENAME WORD GENERATOR
DISCONNECT SERVER SESSION
REMOVE SERVER SESSION

> **Note**
> The one exception is the ALTER USER statement, which can also be used by the current user to change his or her password at any time.

Public Role

The Public role is created automatically for each new ElevateDB configuration, and cannot be dropped or altered. By default, all users are automatically granted the Public role, but the role can be revoked at any time as necessary.

Privileges

The GRANT PRIVILEGES and REVOKE PRIVILEGES statements can be used by any user that has been granted the Administrators role, and are used to specify the database object privileges that are available to the various users and/or roles that are defined in the configuration. The following table shows which privileges may be granted for the various database objects:

Database Object	Privileges
DATABASE	SELECT (Determines Visibility) CREATE (Tables, Views, Functions, Procedures) ALTER (Tables, Views, Functions, Procedures) DROP (Tables, Views, Functions, Procedures) MAINTAIN (Tables) BACKUP RESTORE
STORE	SELECT (Determines Visibility) CREATE (Files) ALTER (Files) DROP (Files)
TABLE	SELECT (Determines Visibility) INSERT UPDATE DELETE CREATE (Triggers, Indexes) ALTER (Triggers, Indexes) DROP (Triggers, Indexes)
VIEW	SELECT (Determines Visibility) INSERT UPDATE DELETE
FUNCTION	EXECUTE (Determines Visibility)
PROCEDURE	EXECUTE (Determines Visibility)

As you can see, the privileges granted on a given object usually dictate whether another object contained within the object can be accessed or altered in some way. For example, a user or role must have been granted CREATE privileges on a given table in order for that user or role to be able to use the CREATE TRIGGER statement to create a new trigger on the table.

1.11 Buffering and Caching

ElevateDB uses caching and buffering algorithms internally to ensure that data is cached for as long as possible and is accessible in the fastest possible manner when needed to perform an operation. ElevateDB offers several different types of buffering, each having a specific purpose for optimizing performance: global file I/O buffering, per-session table buffering, and per-session SQL statement and function/procedure caching.

Global File I/O Buffering

Global file I/O buffering is used to cache as much of the configuration, log, database catalog, and table files as possible in order to maximize I/O throughput. This is accomplished by utilizing heuristics and settings that are specific to ElevateDB, allowing for more control over the caching than what is available when leaving the file caching to the operating system.

Warning
Enabling global file I/O buffering can cause the ElevateDB Server, or any process using ElevateDB that has the I/O buffering enabled, to be more susceptible to experiencing data loss if the process is terminated unexpectedly. There are ways to minimize the chances of such an occurrence, but it is always a possibility at this time. Fail-safe writes will be available at some point in a future update, so this is not a permanent situation.

Global file I/O buffering can be enabled in ElevateDB at the engine level and specific file I/O buffering settings can be also specified at the engine level for any configuration, log, database catalog, and table files that are accessed by ElevateDB. Please see your product-specific manual for more information on enabling and configuring file I/O buffering in code for ElevateDB, and the Starting and Configuring the ElevateDB Server topic for more information on enabling and configuring file I/O buffering in the ElevateDB Server.

Note
When global file I/O buffering is enabled, ElevateDB will exclusively open any configuration, log, database catalog, and table files so that no other processes can open them. Doing this allows ElevateDB to buffer as much data as it needs to without worrying about changes being made by other processes. These files are only accessible through the current ElevateDB process, which means that the global I/O buffering is not usable with multiple processes that need to share configurations/databases using direct, local access. In such a case, one can only use the per-session table buffering form of caching.

After global file I/O buffering has been enabled, buffering settings must also be specified so that they provide the optimal caching for your specific ElevateDB installation. You can also adjust these settings at a later time so that they stay current with any system configuration changes, such as adding more physical memory, or with any major changes to the underlying file sizes. Each buffering setting consists of the following properties:

Setting	Description

File Specification	The file specification is a file name mask and can contain wildcards (*). The file specification mask can include paths, or one can use a wildcard to match on all paths. There is no default value for this setting and you must specify a file mask.
Block Size	This setting controls the size, in KB, of file blocks that will be used for buffering any file that matches the file specification mask. The default value is 4KB.
Buffer Size	This setting controls the maximum amount of memory, in MB, that will be used for buffering any file that matches the file specification mask. The default value is 8MB.
Flush Age	This setting controls how long, in seconds, a dirty file block buffer will stay in the buffer pool before ElevateDB automatically writes the dirty buffer to the file that matches the file specification mask. This setting helps to alleviate issues with dirty buffers not being written to the file on a regular basis because the buffer size is configured too large for the current file size. The default value is 120 seconds.
Flush to Disk	This setting controls whether any writes to any file that matches the file specification mask will be followed by a disk flush call to the operating system. The default value is False.

Note
These settings are evaluated by ElevateDB from back-to-front, so you should specify the settings from general file specifications to very specific file specifications.

In addition to the file mask buffering settings, there is an additional flush check interval setting that specifies how often, in seconds, ElevateDB will scan the buffer pools for each file in order to write any dirty buffers that are past their flush age to the file. The default value is 60 seconds.

As mentioned above, the file I/O buffering is susceptible to causing data loss if the process running ElevateDB is terminated unexpectedly. You can minimize the possibility of this issue for selected files by:

- specifying a low (30 seconds) flush check interval and
- specifying a low flush age (30 seconds) for all applicable files.

You can view the file I/O buffer settings, as well as current statistics for the active file I/O buffer pools, by querying the following table:

FileIOStatistics Table

File Block Buffer Replacement Policy

Any file block buffer maintained within the global file I/O buffer pool is replaced using an LRU, or least-recently-used, algorithm. For example, if the buffer pool is full when reading a file block, ElevateDB will discard the least-recently-used file block in order to make room for the new file block. The "age" of a given buffered file block is determined by the access patterns at the time. Every time a file block buffer is accessed, it is moved so it is the first file block buffer in the LRU list of file block buffers. This would make it the "youngest" buffer present in the LRU list of file block buffers, and all other file block buffers would be moved down the LRU list. As a particular file block buffer moves down the LRU list, it becomes "older" and is more likely to be removed and discarded from the LRU list of file block buffers.

Optimized Writes with File I/O Buffering

When ElevateDB writes file block buffers to a file, the file blocks are ordered according to their offset and ElevateDB attempts to write contiguous file blocks in the fewest number of write operations as possible. This reduces the number of I/O calls and can greatly improve write throughput, especially on hard disk drives that benefit from fewer drive seeks.

File I/O Buffering and OS Buffering

In addition to the file I/O buffering in ElevateDB, additional buffering may be provided by the operating system. When ElevateDB writes data using operating system calls, there is no guarantee that the data will be immediately written to disk. On the contrary, it may be several seconds or minutes until the operating system lazily flushes the data to disk. This has implications in terms of data corruption if the computer is improperly shut down after updates have taken place in ElevateDB. You can minimize the possibility of this issue for selected files by:

- specifying a low (30 seconds) flush check interval,
- specifying a low flush age (30 seconds) for all applicable files,
- specifying that all applicable files follow all flush checks with a disk flush call to the operating system if any file block buffers were written to the file.

Per-Session Table Buffering

At a level above the global file I/O buffering, if enabled, is the per-session table buffering. The per-session table buffering buffers rows, index pages, BLOB blocks, and published update blocks for each open table. There are separate buffer pools for each class of buffer - rows, index pages, BLOB blocks, and published update blocks. If global file I/O buffering is enabled in ElevateDB, any data that isn't available in the per-session table buffers will require a read operation to the file block buffer pool for the applicable file. If global file I/O buffering is not enabled in ElevateDB, any data that isn't available in the per-session table buffers will require a read operation to the operating system.

The amount of memory used for the per-session table buffers is typically very small and only used for improving the locality of access for rows, index pages, and BLOB/published update blocks that are being currently accessed/updated. In most cases the default memory settings for the per-session table buffers will suffice. If necessary, ElevateDB will increase the amount of memory that is being used for the table buffers for a particularly table.

The only exception to this is when an application wants to use direct access to a shared configuration and database(s) located on a file server. In such a case, one can modify the per-session table buffers so that they are larger than the default values. These modifications can be performed when the table is created via the CREATE TABLE statement, or after the table is created via the ALTER TABLE statement. The applicable clauses are as follows:

```
MAX ROW BUFFER SIZE <MaxRowBufferSize>
MAX INDEX BUFFER SIZE <MaxIndexBufferSize>
MAX BLOB BUFFER SIZE <MaxBLOBBufferSize>
```

The default amount of memory used for each is detailed below:

Cache Type	Amount
Rows	32768 bytes
Index Pages	65536 bytes
BLOB Blocks	32768 bytes

You can view the per-session table buffer settings, as well as current statistics for the active table buffers, by querying the following tables:

Type	Table
Server Sessions	ServerSessionStatistics Table
Local Sessions	SessionStatistics Table

Table Buffer Replacement Policy

Any table buffer maintained within the per-session table buffer pool is replaced using an LRU, or least-recently-used, algorithm. Each class of table buffer maintains its own buffer pool. Subsequently, each buffer pool has its own LRU list. For example, if the table row buffer pool is full when reading a row, ElevateDB will discard the least-recently-used row in order to make room for the new row. The "age" of a given buffered row is determined by the access patterns at the time. Every time a row buffer is accessed, it is moved so it is the first row buffer in the LRU list of row buffers. This would make it the "youngest" buffer present in the LRU list of row buffers, and all other row buffers would be moved down the LRU list. As a particular row buffer moves down the LRU list, it becomes "older" and is more likely to be removed and discarded from the LRU list of row buffers.

Read-Ahead Buffering with Table Buffering

ElevateDB performs intelligent read-ahead when reading rows and BLOB/published update blocks:

- For read-ahead on rows, this intelligence is gathered from information in the active index for a given table when accessing a table using a specific ordering, or using raw row information for non-ordered access, and allows ElevateDB to determine how rows physically align with one another on disk.

- For read-ahead on BLOB/published update blocks, this intelligence is gathered from information in the row about the size of the BLOB, or from information about the size of the published updates.

Performing read-ahead in this manner can reduce the number of read calls that ElevateDB has to make to the global file I/O buffering or the operating system and can significantly speed up sequential read operations such as those found in SQL queries and other bulk operations.

Optimized Writes with Table Buffering

When ElevateDB writes table buffers, the table buffers are ordered according to their offset and ElevateDB attempts to write contiguous table buffers in the fewest number of write operations as possible.

Table Buffering and OS Buffering

The effect of operating system buffering on per-session table buffering depends upon whether the global file I/O buffering is enabled or not. If the global file I/O buffering is not enabled, then writes using the per-session table buffers go directly to the operating system. There are session-level settings in ElevateDB that will allow you to specify that such writes are followed by a disk flush call to the operating system. In addition, there are transaction commit options to do the same, as well as specific methods/function calls for explicitly performing disk flush calls. Please see your product-specific manual for more information on

enabling session-level table buffer disk flushing or explicitly performing disk flush calls.

Per-Session SQL Statement and Function/Procedure Caching

At a level above both the per-session table buffering and and the global file I/O buffering is the SQL statement and function/procedure caching. A session can be configured to cache a specified maximum of SQL statements, as well a specified maximum of functions/procedures, per open database in the session.

> **Note**
> The maximum number of open SQL statements and functions/procedures per connection is 2048, so you should not set the statement or function/procedure cache size that high. Typically, values higher than 32 or 64 will exhibit diminishing returns on improved performance.

This level of caching is used to eliminate costly prepare/unprepare cycles with SQL statements and functions/procedures without requiring the developer to explicitly keep statements and functions/procedures prepared. In a lot of cases, such as within scripts, triggers, and other forms of SQL/PSM routines in ElevateDB, it is impossible for a developer to manage the prepared state of various SQL statements and functions/procedures being used.

The SQL statement and function/procedure caching works as follows:

- Each cached SQL statement is managed using a checksum of the SQL statement, and each function/procedure is managed using a checksum of the function/procedure name. Additionally, once an object has been added to the cache, it stays present in the cache until is is ejected due to the LRU replacement policy (see below for the replacement policy details) or explicitly freed from the cache. Each cached object contains an in-use flag that is used to track whether the object can be used or whether a new object must be created. This allows the cache to work in the face of recursive triggers and other functions/procedures, and prevents the cache from incurring an inordinate amount of overhead due to constant modifications to the internal list of objects in the cache.

- When an SQL statement or function/procedure is prepared, ElevateDB checks the cache for the open database in which the SQL statement or function/procedure is being prepared. If the same SQL statement or function/procedure is already present in the cache, then ElevateDB will use the cached object instead of creating a new object. If the SQL statement or function/procedure cannot be found in the cache, then a new object is created and added to the cache. If the maximum number of cached objects has been exceeded, then the oldest (see below for the replacement policy details) cached object is ejected from the cache and freed before the new object is added to the cache.

- When an SQL statement or function/procedure is un-prepared, ElevateDB checks to see if the object was previously cached. If it was, then ElevateDB simply marks the cached object as available for re-use in the cache. If it wasn't, then the object is simply un-prepared as normal, releasing all memory and resources associated with the object.

Possible Cached SQL Statement and Function/Procedure Conflicts

Within a given session, ElevateDB automatically manages freeing cached SQL statements and functions/procedures whenever the session performs an operation that may conflict with any of the cached SQL statements and functions/procedures. This resolves situations where the same session may try to perform operations that may conflict, but does not address issues with cached SQL statements and functions/procedures that may conflict with operations being attempted by other sessions. In order to handle such situations, ElevateDB provides session-level calls that can be used to manually free any cached SQL statements and/or functions/procedures within the session. There are separate calls for both

SQL statements and functions/procedures, and the calls allow you to free objects within a specific open database, or for all open databases within the session. Please see your product-specific manual for more information on performing these operations.

Cached SQL Statement and Function/Procedure Replacement Policy

Any cached SQL statement or function/procedure is maintained within a separate pool for each open database in a session. Each SQL statement or function/procedure is replaced using a LRU, or least-recently-used, algorithm.

1.12 Change Detection

ElevateDB automatically performs change detection when either reading or updating tables.

Reads and Change Detection

When reading from a table, ElevateDB only checks for changes by other sessions when it cannot find the desired data locally in its cache and must physically read the data from the table. The data can be a row, index page, or BLOB block, and the actual check for changes is very quick. If changes are found in the table, ElevateDB will dump its per-session table buffers for the table and retry the read operation that it was in the process of executing when it found that it needed more data from the table.

Note
The amount of memory used for per-session table buffering can affect how often ElevateDB detects changes within tables, and ElevateDB allows you to change these settings on a per-table basis. Please see the Buffering and Caching topic for more information on modifying the per-session table buffering settings for a table.

Updates and Change Detection

When performing updates using the INSERT, UPDATE, or DELETE statements, ElevateDB will automatically make sure that it's per-session table buffers contain the most up-to-date data before performing the actual update operation. ElevateDB performs a row buffer comparison when updating or deleting rows to ensure that the row has not been deleted by another session. If this is the case, then ElevateDB will raise a 1007 (EDB_ERROR_ROWDELETED) error indicating that the row has been deleted by another session and the operation will be aborted.

ElevateDB can also perform a row buffer comparison when updating or deleting rows to ensure that the row that is now present in it's cache contains the same values as the row that was intended to be updated or deleted before the operation was initiated (i.e. it's what the user sees when the row is selected). If the row is not the same due to a change by another session, ElevateDB will raise a 1008 (EDB_ERROR_ROWMODIFIED) error indicating that the row has been modified by another session and the operation will be aborted. By default, this behavior is turned off, but it can be enabled if needed. Please see your product-specific documentation for more information on enabling row change detection.

1.13 Locking and Concurrency

ElevateDB manages most locking and concurrency issues without requiring any action on the part of the user or developer. The following information details the steps that ElevateDB takes internally in order to maximize concurrency while still resolving conflicts for shared resources using locking.

ElevateDB performs locking in two different ways, depending upon whether global file I/O buffering is enabled in the ElevateDB engine. Please see the Buffering and Caching topic for more information on how the file I/O buffering works in ElevateDB.

How ElevateDB Performs Locking when Global File I/O Buffering is Disabled

When global file I/O buffering is disabled, all locks in ElevateDB are performed using calls to the operating system on the configuration lock file (EDBConfig.EDBLck), the database lock file (EDBDatabase.EDBLck), or the database table files themselves (*.EDBTbl). The *.EDBLck files are used for managing shared or exclusive object locks on users, jobs, databases, tables, views, and functions/procedures. The database lock file (EDBDatabase.EDBLck) is also used for managing table read, write, and transaction locks for all tables within the database. The *.EDBTbl files are use for both storing the rows of a table and locking the rows. If using a local session accessing an ElevateDB database on a network file server, these calls are then routed by the operating system to the file server's operating system.

> **Note**
> If either the configuration lock file (EDBConfig.EDBLck) or the database lock file (EDBDatabase.EDBLck) does not exist and cannot be created due to issues with security permissions or read-only media, then the configuration or database will be treated as read-only and you will not be able to modify any objects contained within them.

ElevateDB takes advantage of the fact that modern operating systems allow an application to lock portions of a file beyond the actual size of the file. This process is known as virtual byte offset locking. ElevateDB restricts the size of any physical file that is part of a table to 128,000,000,000 bytes, or slightly below the maximum file size of 128GB. ElevateDB does this so it can reserve the space available between the 128GB mark and the 128,000,000,000 byte mark for row locks in the table.

How ElevateDB Performs Locking when Global File I/O Buffering is Enabled

When global file I/O buffering is enabled, ElevateDB will exclusively open any configuration, log, database catalog, and table files so that no other processes can open them. Doing this allows ElevateDB to buffer as much data as it needs to without worrying about changes being made by other processes. In addition, all locks in ElevateDB are performed using lock structures that are internal to the engine along with the configuration lock file (EDBConfig.EDBLck), the database lock file (EDBDatabase.EDBLck), or the database table files themselves (*.EDBTbl). The internal lock structures are used for managing shared or exclusive object locks on users, jobs, databases, tables, views, and functions/procedures. The database lock file (EDBDatabase.EDBLck) is also used for managing table read, write, and transaction locks for all tables within the database. The *.EDBTbl files are use for both storing the rows of a table and locking the rows using additional internal lock structures.

> **Note**
> If either the configuration lock file (EDBConfig.EDBLck) or the database lock file (EDBDatabase.EDBLck) does not exist and cannot be created due to issues with security permissions or read-only media, then the configuration or database will be treated as read-only and you will not be able to modify any objects contained within them.

Row Locking Protocols

ElevateDB offers two types of row locking protocols, pessimistic (default) and optimistic locking.

Locking Protocol	Description
Pessimistic	The pessimistic row locking protocol specifies that a row should be locked when the row is retrieved for updating.
Optimistic	The optimistic locking protocol specifies that a row should be locked when any row modifications are posted back to the table. Using the optimistic row locking protocol for remote sessions removes the possibility that dangling row locks will be left on the ElevateDB Server if the application is terminated unexpectedly. However, even with the pessimistic row locking protocol, an ElevateDB server can clean up dead sessions and remove any row locks that they may be holding.

The two row locking protocols can safely and reliably be used among multiple sessions on the same database, although it is not recommended due to the potential for confusion for the developer or user of the application.

Row Locks

Row locks are used to enforce ElevateDB's pessimistic or optimistic row locking protocols and prevent the same or multiple sessions from updating the same row at the same time. Row locks block other row lock attempts, but do not block any reads of the locked rows. The following details what happens in the various scenarios that use row locks:

Action	Description

Inserting	When inserting a row, no row locks are acquired until the row is actually inserted. During the insertion of a new row, a row lock is only implicity acquired by ElevateDB on the new row if the insertion is taking place inside of a transaction.
Updating	When updating a row, a row lock is implicity acquired by ElevateDB. This row lock will fail if the row is already locked by the same session or a different session. If the row lock fails, then an exception will be raised. The error code that is raised when a row lock fails is 1005 (EDB_ERROR_LOCKROW). If the row locking protocol for the session is set to optimistic then ElevateDB will not attempt to implicitly acquire a row lock when the row is retrieved, but will instead wait until the row is actually updated to implicitly acquire the row lock. This means that another session is capable of updating or deleting the row before the current session actually completes the update. If either of these actions occur, an exception will be raised. The error code that is raised when an update fails because the row has been altered is 1008 (EDB_ERROR_ROWMODIFIED). The error code that is raised when an update fails because the row has been deleted is 1007 (EDB_ERROR_ROWDELETED).
Deleting	When deleting a row, a row lock is implicity acquired by ElevateDB. This row lock will fail if the row is already locked by the same session or a different session. If the row lock fails, then an exception will be raised. The error code that is raised when a row lock fails is 1005 (EDB_ERROR_LOCKROW).

The number of row lock retries and the amount of time between each retry can be controlled on a per-session basis. In a busy multi-user application it may be necessary to increase these values in order to relieve lock contention and provide for smoother concurrency between multiple users. However, in most cases the default values should work just fine. Please see your product-specific manual for more information on changing these settings for the session.

Table Locks

There are three types of table locks used by ElevateDB:

Type	Description

Table Read Locks	Table read locks allow ElevateDB to accurately treat reads on internal table structures, such as the indexes or BLOB columns, as atomic, or a single unit of work. Table read locks ensure that no other session writes to the table by blocking any table write locks. Table read locks are the most widely-used locks in ElevateDB and are the cornerstone of correct multi-user operation. They especially play a large role in change detection. Please see the Change Detection topic for more information. Table read locks are also acquired during table scans for un-optimized query conditions. Please see the Optimizer topic for more information on optimizing query conditions.
Table Write Locks	Table write locks allow ElevateDB to accurately treat writes on internal table structures, such as the indexes or BLOB columns, as atomic, or a single unit of work. Table write locks ensure that no other session reads from or writes to the table by blocking any table read lock or write locks.
Table Transaction Locks	Table transaction locks allow ElevateDB to treat single or multi-table updates within a transaction as atomic, or a single unit of work. Table transaction locks ensure that no other session begins a transaction on the table by blocking any other table transaction locks. Table read locks are allowed, however, and other sessions can read the rows from tables and acquire row locks. When a transaction is ready to be committed to disk, additional table write locks are acquired in order to block other table reads or writes while the data is being committed.

1.14 Transactions

ElevateDB allows for transactions in order to provide the ability to execute single or multi-table updates and have them treated as an atomic unit of work. Transactions are implemented logically in the same fashion as most other database engines, however at the physical level there are some important considerations to take into account and these will be discussed here.

Executing a Transaction

A transaction is started and committed or rolled back by using the START TRANSACTION, COMMIT, and ROLLBACK statements within an ElevateDB job, procedure, function, or trigger.

A typical transaction block of code looks like this:

```
BEGIN
   START TRANSACTION;
   -- Perform some updates to the table(s) in this database
   COMMIT;
EXCEPTION
   ROLLBACK;
END
```

Note
It is very important that you always ensure that the transaction is rolled back if there is an exception of any kind during the transaction. This will ensure that the row and table locks held by the transaction are released and other sessions can continue to update data while the exception is dealt with.

Restricted Transactions

It is also possible with ElevateDB to start a restricted transaction. A restricted transaction is one that specifies only certain tables be part of the transaction. The START TRANSACTION statement accepts an optional list of tables that can be used to specify what tables should be involved in the transaction and, subsequently, locked as part of the transaction (see below regarding locking). If this list of tables is not specified (the default), then the transaction will encompass the entire database.

The following example shows how to use a restricted transaction on two tables, the Customer and Orders table:

```
BEGIN
   START TRANSACTION ON TABLES 'Customer', 'Orders';
   -- Perform some updates to the tables
   COMMIT;
EXCEPTION
   ROLLBACK;
END
```

Flushing Data to Disk During a Commit

By default, the COMMIT statement will cause a flush of all data to disk within the operating system. The COMMIT statement has the optional keywords NO FLUSH that will prevent the OS flush from occurring. This will improve the performance of a commit operation at the expense of possible data corruption if the application is improperly terminated after the commit takes place. This is due to the fact that the operating system may wait several minutes before it lazily flushes any modified data to disk. Please see the Buffering and Caching topic for more information.

Locking During a Transaction

When a transaction on the entire database is started, ElevateDB acquires a table transaction lock on all tables in the database. This prevents any other sessions from inserting, updating, or deleting any rows from the tables in the database while the current transaction is active. When a restricted transaction is started on a specific set of tables, ElevateDB will only acquire this table transaction locks on the tables specified as part of the transaction. It is very important with ElevateDB that all transactions be kept as short as possible.

Note
Table transaction locks do not prevent other sessions from reading rows from the tables involved in the transaction or acquiring row locks on the tables involved in the transaction while the current transaction is active. This means that it is still possible for other sessions to cause a row update or delete within the transaction to fail due to not being able to acquire the necessary row lock. Also, any row locks acquired during a transaction will remain locked until the transaction is rolled back or committed. This can have some adverse side affects with some network operating systems that only permit a fixed number of locks per connection. These row locks can accumulate over the course of a lengthy transaction and you can run into this limit rather quickly, ending up with an OS locking error that is seemingly coming from nowhere. If you plan on executing many inserts, updates, or deletes within a single transaction then you should make sure to check your network operating system documentation in order to verify that you won't run into any limitations such as this.

Opening and Closing Tables

If a transaction on the entire database (not a restricted transaction) is active and a new table is opened, that table will automatically become part of the active transaction. Unlike a transaction on the entire database, if a table involved in a restricted transaction is not currently open at the time that the START TRANSACTION statement is executed, then an attempt will be made to open it at that time. Also, any tables that are opened during the restricted transaction and not initially specified as part of the restricted transaction will be excluded from the transaction. If a table involved in a transaction, either restricted or not, is closed while the transaction is still active, the table will be kept open internally by ElevateDB until the transaction is committed or rolled back, at which point the table will then be closed.

SQL and Transactions

The INSERT, UPDATE, or DELETE statements implicitly use a restricted transaction on the involved tables if a transaction is not already active. The interval at which the implicit transaction is committed is internally calculated to be optimal for the table being updated. If a transaction was explicitly started by the user or developer, then ElevateDB will not commit any of the effects of the SQL statement automatically, leaving the committing up to the explicit transaction.

> **Note**
> By default, commits that occur during the execution of SQL statements do not force an operating system flush to disk.

Incompatible Operations

The following statements are not compatible with transactions and will cause an exception if encountered during a transaction.

BACKUP DATABASE
RESTORE DATABASE

SAVE UPDATES
LOAD UPDATES

CREATE TABLE
ALTER TABLE
DROP TABLE
RENAME TABLE
REPAIR TABLE
OPTIMIZE TABLE

CREATE VIEW
ALTER VIEW
DROP VIEW
RENAME VIEW

CREATE INDEX
CREATE TEXT INDEX
ALTER INDEX
DROP INDEX
RENAME INDEX

CREATE TRIGGER
ALTER TRIGGER
DROP TRIGGER
RENAME TRIGGER

CREATE FUNCTION
ALTER FUNCTION
DROP FUNCTION
RENAME FUNCTION

CREATE PROCEDURE
ALTER PROCEDURE
DROP PROCEDURE
RENAME PROCEDURE

> **Note**
> There is an exception to the following statements for temporary tables:

CREATE TABLE
ALTER TABLE
DROP TABLE
RENAME TABLE

CREATE INDEX
CREATE TEXT INDEX
ALTER INDEX
DROP INDEX
RENAME INDEX

These statements can be executed for temporary tables, even inside of a transaction.

Isolation Level

The default and only isolation level for transactions in ElevateDB is serializable. This means that only the session in which the transaction is taking place will be able to see any inserts, updates, or deletes made during the transaction. All other sessions will see the data as it existed before the transaction began. Only after the transaction is committed will other sessions see any new row inserts, updates, or deletes.

Data Integrity

A transaction in ElevateDB is buffered, which means that all row inserts, updates, or deletes that take place during a transaction are cached in memory for the current session and are not physically applied to the tables involved in the transaction until the transaction is committed. If the transaction is rolled back, then the updates are discarded. With a local session this allows for a fair degree of stability in the case of a power failure on the local workstation, however it will not prevent a problem if a power failure happens to occur while the commit operation is taking place. Under such circumstances it's very likely that physical and/or logical corruption of the tables involved in the transaction could take place. The only way corruption can occur with a remote session is if the ElevateDB Server itself is terminated improperly during the middle of a transaction commit. This type of occurrence is much more rare with a server than with a workstation.

1.15 External Modules

External modules can used to extend the functionality of ElevateDB to external code in the form of DLLs. The functionality that can be extended via external modules includes:

Functionality	Description
Procedures and Functions	A procedure or function can be defined to use an external module for its implementation via the EXTERNAL NAME keywords in the CREATE PROCEDURE or CREATE FUNCTION statements.
Text Indexing	Text indexing can be customized for a given index by specifying an external module for the text filtering via the MODULE keyword in the CREATE TEXT FILTER statement, and/or by specifying an external module for the word generation via the MODULE keyword in the CREATE WORD GENERATOR statement. Please see the Text Indexing topic for more information.
Migration	Migrators used by ElevateDB to migrate databases from external data sources use external modules to provide their implementation via the MODULE keyword in the CREATE MIGRATOR statement. Please see the Migrating Databases topic for more information.

Creating External Modules

External modules can be created in any language that can generate a DLL (Dynamic Link Library) with a C-style calling convention, which is the standard calling convention for DLLs under Windows. In addition, ElevateDB provides template projects for the various types of external modules in every specific compiler or IDE that it supports. Please see your product-specific manual for more information.

Note
Although all external modules in ElevateDB are DLLs, each type of external module has a different set of calling conventions and identifies itself differently so that ElevateDB can verify whether the proper type of external module is being used with the proper type of functionality in ElevateDB.

Installing External Modules

In order to use an external module in ElevateDB, you must make sure that the module is registered in the current configuration file by using the CREATE MODULE statement. You can verify that this is done by using the following SELECT statement on the special system-defined Configuration Database:

```
SELECT * FROM Modules
```

If the rows returned from the above query include the module that you wish to use with ElevateDB, then external module has been registered successfully in the configuration file. Please see the Architecture topic for more information on the configuration file.

Once an external module has been registered correctly, it can then be used with ElevateDB with procedures and functions, text indexing, and/or database migration.

1.16 Migrating Databases

ElevateDB provides an open migration interface so that migrators can be written to migrate data from literally any external data source. Migrators are defined using the CREATE MIGRATOR statement and refer to migrator modules (DLLs) that implement the actual migration interface. Please see your product-specific manual for more information on creating migrator modules. The MIGRATE DATABASE statement is used to actually execute the migration for the external data source.

Standard Migrator Modules

ElevateDB includes the following migrator modules:

Module	Description
edbmigrate	ElevateDB migrator module
edbmigratedbisam1	DBISAM Version 1.x migrator module
edbmigratedbisam2	DBISAM Version 2.x migrator module
edbmigratedbisam3	DBISAM Version 3.x migrator module
edbmigratedbisam4	DBISAM Version 4.x migrator module
edbmigratebde	BDE (Borland Database Engine) migrator module
edbmigrateado	ADO (Microsoft ActiveX Data Objects) migrator module
edbmigratendb	NexusDB migrator module
edbmigrateads	ADS (Advantage Database Server) migrator module

You can find these migrator modules as part of the ElevateDB Additional Software and Utilities (EDB-ADD) installation in the \libs subdirectory under the main installation directory. There are ANSI and Unicode versions of each of the migrator modules that will work with both ANSI or Unicode sessions.

Note
You can download the ElevateDB Additional Software and Utilities (EDB-ADD) installation from the Downloads page of the web site.

In order to reference these migrator modules from within a migrator in ElevateDB, you must make sure that the migrator modules (DLLs) are registered in the configuration file by using the CREATE MODULE statement. You can verify that this is done by using the following SELECT statement on the special system-defined Configuration Database:

```
SELECT * FROM Modules
```

If the rows returned from the above query include the five migrator modules listed above, then the migrator modules have been registered successfully in the configuration file. Please see the Architecture topic for more information on the configuration file.

Creating a Migrator

To create a migrator that uses the desired migrator module, you can use the CREATE MIGRATOR statement. For example, you would use the following statement to create a migrator for use with DBISAM 1 data sources:

```
CREATE MIGRATOR "DBISAM1"
MODULE "edbmigratedbisam1"
DESCRIPTION 'DBISAM 1 Migrator'
```

Migrating the External Data

The first step in migrating an external data source is to query the parameters required for the migrator. You can do so by querying the MigratorParams Table table in the Configuration database. This table contains the parameters for each migrator that are retrieved from the migrator module, along with their type and any default values. These parameters are important because they will be used with the MIGRATE DATABASE statement to populate the parameters as required by the migrator. Usually, the most important parameter is the name of the external database, or database directory, or a connection string that indicates the proper values used to connect to the external data source.

The target database for a migration must be present before the migration takes place, and the migration is always executed from within the target database. You can use the CREATE DATABASE statement in order to create the target database.

To perform the migration, you can execute the MIGRATE DATABASE statement from the ElevateDB database that you just created (or already existed):

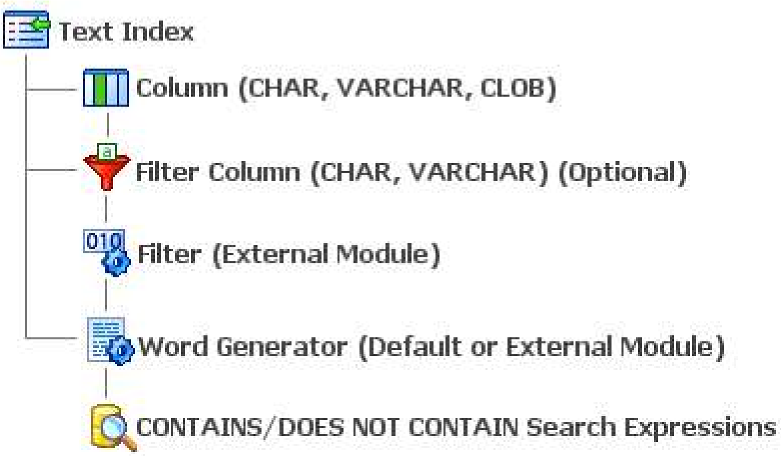
```
MIGRATE DATABASE FROM "DBISAM1"
USING DatabaseDirectory = 'c:\dbisamdata'
```

When the MIGRATE DATABASE statement is executed, the external data source should migrate to the current ElevateDB database. Any errors that are encountered will be raised as an exception with an ElevateDB error code.

1.17 Text Indexing

ElevateDB provides the ability to index CHAR, VARCHAR, or CLOB columns so that they may be quickly searched for a given word or words. This is known as text indexing since it results in the indexing of every word in a specified column.

The following image illustrates the general architecture of the text indexing in ElevateDB:



You can use the CREATE TEXT INDEX statement to create a new text index on a given column. When creating a text index, you may specify the CHAR, VARCHAR, or CLOB column to index, the indexed word length, optionally another CHAR or VARCHAR column to use as a text filter type indicator to ElevateDB, and optionally a specific word generator module to use for generating the actual words that are added to the index.

Specifying the Indexed Column

Each text index can index one, and only one, CHAR, VARCHAR, or CLOB column. By default, ElevateDB always uses the collation of the column that is being indexed and modifies it so that all comparisons are case-insensitive. However, if a collation is explicitly specified along with the indexed column, then ElevateDB will use that collation instead of a case-insensitive version of the column being indexed. For example, the following text index overrides the default collation of the Notes column (ANSI_CI) so that the text index uses a case-sensitive version:

```
CREATE TEXT INDEX "Notes" ON "Customer"
(Notes COLLATE ANSI)
INDEXED WORD LENGTH 20
```

Note
It is generally recommended that you always use a case-insensitive collation with any text index in order to reduce the size of the text index and to make searching easier.

Specifying the Indexed Word Length

The indexed word length controls how long each index key is in the actual text index. It does not affect which words are indexed in any way. However, if a word that is being indexed is longer than the indexed word length specified when the text index was created, then the word will be truncated to the indexed word length. If the indexed word length is not specified, then the default indexed word length of 30 characters is used. You should try to keep the indexed word length as small as possible in order to minimize the size of the text index.

Note
The minimum word length indexed by the default word generator is 3 characters. Any word smaller than 3 characters will not be included in the text index.

Specifying a Filter Type Column

The contents of a filter type column indicate to ElevateDB what type of data is in the column being indexed. This means that you can store text with various types of formatting in the same column and still have the text index only index the non-formatting information. The filter type indicator is used to look up the applicable text filter in the defined Text Filters in the current Configuration Database. If a matching text filter is found, then the text to be indexed is first passed to the text filter before being passed on to the word generator (see below). If a matching text filter is not found, then the text is passed on directly to the word generator without being filtered.

For example, suppose that you have a column in your table called Notes and a column called TypeOfNotes. The Notes column may contain either plain text, HTML-formatted text, or RTF-formatted text, and the type of text is indicated by the TypeOfNotes column, which will contain either a NULL (plain text), 'HTML' (HTML Text), or 'RTF' (RTF Text) value in each row. In addition, you have defined two text filters that use external modules to parse out all non-formatting text and return it to ElevateDB for use in the word generation:

```
CREATE TEXT FILTER HTMLFilter
TYPE 'HTML'
MODULE HTMLTextFilterModule
DESCRIPTION 'HTML Text Filter'

CREATE TEXT FILTER RTFFilter
TYPE 'RTF'
MODULE RTFTextFilterModule
DESCRIPTION 'RTF Text Filter'
```

Whenever the Notes column is updated, the appropriate text filter will be called with the new contents of the Notes column, and the filtered text that is returned will be passed on to the word generation process.

Please see your product-specific manual for information on creating external modules that can implement text filtering.

Specifying a Word Generator

By default, ElevateDB will use the following parameters when parsing and generating words from text:

Space Characters

Space characters are used to determine which characters should be treated as whitespace. Word breaks

always occur at any character that is considered whitespace.

```
#0..#47,#58..#64,
#91..#96,#123..#130,
#132..#137,#139,#141,
#143..#153,#155,#157,
#160..#191,#215,#247
```

All numeric values represent the ordinal character value in the 256 characters of the Windows ANSI Code Page 1252 character set.

Include Characters

Include characters are used to determine which characters should be included in the words that are generated. Any character that isn't an include character or space character is simply ignored.

```
'A'..'Z',
'a'..'z',#131,
#138,#140,#142,
#154,#156,#158..#159,
#192..#214,#216..#246,
#248..#255
```

All numeric values represent the ordinal character value in the 256 characters of the Windows ANSI Code Page 1252 character set.

Stop Words

Stop words are words that are so common in most text that they provide no value in terms of narrowing the search process and increase the size of the text index. Stop words are sometimes also referred to as noise words.

```
'ABOUT','ABOVE','AFAIK','ALL','ALONG','ALSO','ALTHOUGH','AND','ARE','ARENT',
'BECAUSE','BEEN','BTW','BUT','CAN','CANNOT','CANT','COULD','COULDNT','DID',
'DIDNT','DOES','DOESNT','DUH','EITHER','ETC','EVEN','EVER','FOR','FROM',
'FURTHERMORE','FYI','GET','GETS','GOT','GOTTEN','HAD','HADNT','HARDLY',
'HAS','HASNT','HAVING','HENCE','HER','HERE','HERS','HEREBY','HEREIN',
'HEREOF','HEREON','HERETO','HEREWITH','HIM','HIS','HOW','HOWEVER','IMHO',
      'IMO',
'INTO','ISNT','ITS','LOL','MINE','NOR','NOT','ONTO','OTHER','OTOH','OUR',
'OURS','OUT','OVER','REALLY','ROTFL','SAID','SAME','SHE','SHOULD','SHOULDNT',
'SINCE','SOMEWHAT','SUCH','THAN','THAT','THATLL','THATS','THE','THEIR',
'THEIRS','THEM','THEN','THERE','THEREBY','THEREFORE','THEREFROM',
'THEREIN','THEREOF','THEREON','THERETO','THEREWITH','THESE','THEY',
'THEYLL','THEYRE','THIS','THOSE','THROUGH','THROUGHOUT','THUS','TIA','TOO',
'UNDER','UNTIL','UNTO','UPON','VERY','WAS','WASNT','WERE','WERENT','WHAT',
'WHEN','WHERE','WHEREBY','WHEREIN','WHETHER','WHICH','WHILE','WHO','WHOM',
'WHOS','WHOSE','WHY','WITH','WITHIN','WITHOUT','WONT','WOULD','WOULDNT',
'YOU','YOULL','YOUR','YOURE','YOURS'
```

In order to override the default word generation, one must specify a different word generator when

creating a text index.

Please see your product-specific manual for information on creating external modules that can implement custom word generation.

Performing a Text Index Search

ElevateDB includes CONTAINS, DOES NOT CONTAIN, CONTAINS ANY, and DOES NOT CONTAIN ANY operators for searching a text index for a series of words. The difference between CONTAINS and CONTAINS ANY (and their negatives) is that the CONTAINS operator performs an ANDed search of all specified search words, while the CONTAINS ANY operator performs an ORed search of all specified search words. The following is an example of using the CONTAINS operator to search for the word 'Development':

```
SELECT *
FROM Customer
WHERE Notes CONTAINS 'Development'
```

Note
The CONTAINS, CONTAINS ANY, DOES NOT CONTAIN, and DOES NOT CONTAIN ANY operators can only be used with columns that have been indexed with a text index. Using them with a non-text-indexed column will result in an error.

If multiple search words are specified with the CONTAINS or DOES NOT CONTAIN operators, then ElevateDB will return all rows that contain all of the search words. If you want to return all rows that contain only some of the search words, then you will need to use the CONTAINS ANY or DOES NOT CONTAIN ANY operators. For example, if you want to return all rows that contain either the word 'Development' or 'Vacation' in the Notes column, then you would use the following SELECT statement:

```
SELECT *
FROM Customer
WHERE Notes CONTAINS ANY 'Development Vacation'
```

You can also specify partial-word searches by using an asterisk (*) anywhere in any of the search words. The following is an example of using the DOES NOT CONTAIN operator to find all rows that don't contain any version of the word 'Develop' in the Notes column:

```
SELECT *
FROM Customer
WHERE Notes DOES NOT CONTAIN 'Develop*'
```

The following is an example of using the CONTAINS operator to find all rows that contain 'invest' in any words in the Notes column:

```
SELECT *
FROM Customer
WHERE Notes CONTAINS '*invest*'
```

You can mix and match search words with wildcards and whole search words in the same search string.

1.18 Optimizer

ElevateDB uses available indexes and I/O cost estimates when optimizing SQL queries so that they execute in the least amount of time possible. In addition, joins are re-arranged whenever possible so that the smallest number of actual join operations occur during the execution of a query.

Index Selection

ElevateDB will use an available index to optimize any expression in the JOIN or WHERE clause of an SELECT, UPDATE, or DELETE statement. It will also use an available index to optimize any join expressions between multiple tables. This index selection is based on the following rules:

1) ElevateDB can only optimize expressions that resolve to the following formats:

```
<ColumnReference> [<CollationReference>] <Operator> <Expression>

<RowValueConstructor> <Operator> <Expression>

<RowValueConstructor> =

(<ColumnReference> [<CollationReference>],
 <ColumnReference> [<CollationReference>]
 [,<ColumnReference> [<CollationReference>]])
```

The <ColumnReference> or <RowValueConstructor> and <Expression> items are not order-dependent, and ElevateDB will reverse them as necessary so that the <ColumnReference> or <RowValueConstructor> is on the left-hand side of the <Operator>. The only requirement is that the <Expression> be appropriate for what is being compared against, so if a comparison is being made against a row value constructor, then the expression must also use a row value constructor.

For more information on row value constructors, please see the Row Value Constructors topic.

2) ElevateDB only uses the first column of any given index for the optimization of single column references. This means that if you have an index containing the columns LastName and FirstName, ElevateDB can only use the this index for optimizing any expressions that contain a reference to the LastName column. When optimizing a row value constructor that is comprised of column references, ElevateDB will try to find a matching index by comparing the column references in the row value from left-to-right.

3) ElevateDB can use an index for optimization irrespective of the ascending or descending status of a given column in the index.

4) ElevateDB can only use an index for optimization if columns in the index match both the column references and their collation references, if specified, in the expression that ElevateDB is attempting to optimize. If the collation references are not specified, then columns in the index must match the defined collations for the column references.

5) ElevateDB can mix and match the optimization of expressions so that it is possible to have one expression be optimized and the other not. This is known as a partially-optimized query.

For example, suppose that you have a Customer table with a State column that was defined with the ANSI_CI (ANSI collation, case-insensitive). An index was created on the State column using the following

CREATE INDEX statement:

```
CREATE INDEX State ON Customer (State)
```

To execute an optimized search for any rows where the State column contains 'FL', one would use the following SELECT statement:

```
SELECT *
FROM Customer
WHERE State = 'FL'
```

Since the collation defined for the State column is case-insensitive, you could also use the following SELECT statement and get the same result:

```
SELECT *
FROM Customer
WHERE State = 'fl'
```

However, suppose that the State column was defined with simply the ANSI collation (case-sensitive), but the index was created using the following CREATE INDEX statement:

```
CREATE INDEX State ON Customer
(State COLLATE ANSI_CI)
```

In order to allow ElevateDB to use this index to optimize any searches on the State column, you must now specifically reference the ANSI_CI collation in the actual search expression:

```
SELECT *
FROM Customer
WHERE State COLLATE ANSI_CI = 'FL'
```

Please see the Internationalization topic for more information on collations.

How ElevateDB Selects the Rows

Once an index is selected for the optimization of an expression in a JOIN or WHERE clause, a range is set on the index in order to limit the index keys to those that match the current expression being optimized. The index keys that satisfy the expression are then scanned, and during the scan a bitmap is built in row order. A bit is turned on if the index key satisfies the expression, and a bit is turned off if it doesn't. This method of using bitmaps works well because it can represent sets of data with minimal memory consumption. Also, ElevateDB is able to quickly determine how many rows are in a given set by how many bits are turned on, and it can easily AND, OR, and NOT bitmaps together to fulfill boolean logic between multiple expressions joined by the AND, OR, and NOT boolean operators. Finally, because the bitmap is in row order, accessing the rows using a bitmap is very direct since ElevateDB uses fixed-length rows with

directly-addressable offsets in the table.

Optimizing Before-Join Expressions

When optimizing queries that contain both JOIN and WHERE expressions, ElevateDB always processes the WHERE expressions first if the expressions do not reference the target table(s) in any of the join(s). The target table in a join is the table on the right side of a LEFT OUTER JOIN or the table on the left side of a RIGHT OUTER JOIN. WHERE expressions that reference the target table in a join are called after-join expressions (see below).

Note
INNER JOINs also have target tables, but due to the nature of an INNER JOIN, both the driver and target table can be optimized as before-join expressions.

Evaluating WHERE expressions as before-join expressions can speed up the joins tremendously since the joins will only need to take into account the rows in the source tables based upon the before-join WHERE expressions. For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
OrderDet.Model_Num,
OrderDet.Cust_Item
FROM OrderHdr INNER JOIN OrderDet ON
OrderHdr.Order_Num=OrderDet.Order_Num
WHERE OrderHdr.Cust_ID = 'C901'
ORDER BY OrderHdr.Cust_ID,
OrderHdr.Order_Num
```

In this example, the WHERE expression:

```
OrderHdr.Cust_ID = 'C901'
```

will be evaluated first before the INNER JOIN expression:

```
OrderHdr.Order_Num = OrderDet.Order_Num
```

so that the INNER JOIN only needs to evaluate a small number of rows in the OrderHdr table.

Optimizing During-Join Expressions

When optimizing SELECT queries that contain INNER JOINs that contain non-join expressions in addition to join expressions, the non-join expressions are always processed at the same time as the join expression, even if they affect the target table of the INNER JOIN. This can speed up join operations tremendously since the join expressions will only take into account the rows existing in the target table based upon the non-join expression(s). For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
OrderDet.Model_Num,
OrderDet.Cust_Item
FROM OrderHdr INNER JOIN OrderDet ON
OrderHdr.Order_Num = OrderDet.Order_Num AND OrderHdr.Cust_ID = 'C901'
ORDER BY OrderHdr.Cust_ID,
OrderHdr.Order_Num
```

In this example, the non-join expression:

```
OrderHdr.Cust_ID = 'C901'
```

will be evaluated first before the join expression:

```
OrderHdr.Order_Num = OrderDet.Order_Num
```

so that the joins only need to process a small number of rows in the OrderHdr table.

After-Join Expressions

After-join expressions are expressions that must be processed after any joins have executed because they contain a column reference to a column in a table that is the target table of a right or left outer join. After-join expressions are always evaluated in an un-optimized manner, meaning that they are processed after all joins have been executed. Therefore, they are not useful in limiting the amount of work or costs involved in a particular query, but rather only useful in filtering the resultant rows of the query based upon a specific expression. For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
FROM OrderHdr LEFT OUTER JOIN OrderDet ON
OrderHdr.Order_Num = OrderDet.Order_Num
WHERE OrderDet.Order_Num IS NULL
ORDER BY OrderHdr.Cust_ID,
OrderHdr.Order_Num
```

In this example, the non-join expression:

```
OrderHdr.Order_Num IS NULL
```

will be evaluated after all joins have been executed so that ElevateDB can accurately assess whether the LEFT OUTER JOIN has caused any NULL Order_Num columns to be generated from the join.

How Joins are Executed

Joins in a SELECT statement are executed in ElevateDB using a technique known as nested-loop evaluation. This means that ElevateDB recursively processes the source tables in a master-detail, master-detail, etc. arrangement with a driver table and a target table (which then becomes the driver table for any subsequent joins). When using this technique, it is very important that the table with the smallest row count, after any non-join expressions have been evaluated, is specified as the first driver table in the join execution. ElevateDB's optimizer will automatically optimize the join ordering so that the table with the smallest row count is placed as the first driver table, as long as the joins are INNER JOINS. LEFT OUTER JOINs and RIGHT OUTER JOINs cannot be re-ordered in such a fashion and must be left as-is.

The following is an example that illustrates nested-loop joins in ElevateDB:

```
SELECT c.Company,
o.OrderNo,
e.LastName,
p.Description,
v.VendorName
FROM Customer c
INNER JOIN Orders o ON c.CustNo=o.CustNo
INNER JOIN Employee e ON o.EmpNo=e.EmpNo
INNER JOIN Items i ON o.OrderNo=i.OrderNo
INNER JOIN Parts p ON i.PartNo=p.PartNo
INNER JOIN Vendors v ON p.VendorNo=v.VendorNo
ORDER BY e.LastName
```

In this example, ElevateDB would process the joins in this order:

1) The Customer table is joined to Orders table on the CustNo column.

2) The Orders table is joined to the Items table on the OrderNo column and the Orders table is joined to Employee table on the EmpNo column (this is also known as a multi-way, or star, join).

3) The Items table is joined to the Parts table on the PartNo column.

4) The Parts table is joined to the Vendors table on the VendorNo column.

In this case the Customer table is the smallest table in terms of its row count, so making it the driver table in this case is a good choice.

Note
You can use the NOJOINOPTIMIZE keyword at the end of a SELECT statement in order to tell ElevateDB not to reorder the joins. Also, you can use the JOINOPTIMIZECOSTS clause to force the ElevateDB optimizer to use I/O cost projections to determine the most efficient way to process the joins. If you have a join with multiple join expressions in it, then using this clause may help improve the performance of the join, especially if it is already executing very slowly.

Execution Plans

ElevateDB can generate an execution plan for any DML statement. Please see your product-specific

manual for more information on retrieving an execution plan for a SELECT, INSERT, UPDATE, or DELETE statement.

Limitations to the Optimizer

ElevateDB does not currently optimize multiple JOIN or WHERE expressions joined by an AND operator by mapping them to multiple columns in an available index. To illustrate this point, consider a table with the following structure:

```
Employee

Column       Data Type      Index
----------------------------------------------------------------------
LastName     VARCHAR(30)    Primary Key  (both columns are part of the
FirstName    VARCHAR(20)    Primary Key   primary key constraint)
```

And consider the following SELECT statement:

```
SELECT *
FROM Employee
WHERE (LastName = 'Smith') and (FirstName = 'John')
```

Logically you would assume that ElevateDB can use the one index available for the enforcement of the primary key constraint in order to optimize the entire WHERE clause. Unfortunately, this is not the case, and instead ElevateDB will only use the index created for the primary key constraint for optimizing the LastName expression and resort to reading the resultant rows in order to evaluate the FirstName expression.

However, you can overcome this limitation by using a row value constructor instead of two expressions combined with the AND operator:

```
SELECT *
FROM Employee
WHERE (LastName,FirstName) = ('Smith','John')
```

With the WHERE clause specified using a row value constructor, ElevateDB will be able to use the entire primary key to optimize the expression.

Note
ElevateDB automatically uses a system-defined index to enforce primary key, unique, and foreign key constraints, so the presence of an index for such a constraint can always be assumed.

1.19 Result Set Cursor Sensitivity

ElevateDB generates two types of query result set cursors depending upon the makeup of a given SELECT statement:

Type	Description
Sensitive	The result set cursor is editable and all inserts, updates, and deletes performed via the cursor are performed directly on the source table. Also, any changes made by any other sessions on the source table are reflected in the cursor, subject to ElevateDB Change Detection. This type of result set cursor is sometimes also referred to as "Dynamic".
Insensitive	The result set cursor is read-only and cannot be edited.. This type of result set cursor is sometimes also referred to as "Static".

The following rules determine whether a result set cursor will be sensitive or insensitive.

Single-table queries

Queries that retrieve data from a single table will generate a sensitive result set provided that:

1) The user or developer requests a sensitive result set cursor. Please see the DECLARE statement for more information on requesting a a sensitive or insensitive result set cursors in SQL/PSM routines, and your product-specific manual for requesting sensitive or insensitive cursors in client applications.

2) There is no DISTINCT keyword in the SELECT statement.

3) All SELECT expressions are either a column reference or a computed column that does not contain any aggregate functions (MIN, MAX, SUM, etc.). Computed columns are read-only in the sensitive result set cursor and cannot be modified.

4) There is no GROUP BY clause in the SELECT statement.

5) There is no ORDER BY clause in the SELECT statement, or there is an ORDER BY clause that minimally matches the columns, and the collations defined for the columns, in an existing index in the source table.

6) There are no correlated sub-queries in the WHERE clause of the SELECT statement.

Note
For sensitive query result set cursors with computed columns, the update of any column in a given row causes the update of any dependent computed columns in that same row.

A query containing sub-queries in the SELECT column expressions can return a sensitive result set in ElevateDB. This means that a query like the following can return a sensitive result set:

```
SELECT CustNo,
(SELECT Company FROM Customer WHERE Customer.CustNo=Orders.CustNo) AS Company
```

```
FROM Orders
```

If the sensitive result set is updated and the CustNo column is changed, the "looked-up" Company value will automatically change as necessary. This is extremely useful for situations where, in the past, you would normally use a join and get an insensitive result set.

Multi-table queries

All queries that join two or more tables or merge two or more SELECT statements via the UNION/INTERSECT/EXCEPT operators will automatically produce an insensitive result set cursor, irrespective of the requested result set cursor type.

Temporary Tables

If a SELECT statement generates an insensitive result set cursor, then a temporary table will created in order to hold the rows that make up the result set. This temporary table is stored in a location specified by either the ElevateDB Server or the client application. By default, ElevateDB uses the local user temporary files path in the operating system for this setting. Please see your product-specific manual for more information on modifying the temporary tables path for either the ElevateDB Server or the client application.

Identifying the Result Set Cursor Type

You may use the SENSITIVE function to identity the type of a result set cursor in an SQL/PSM routine after it has been opened via the OPEN statement. Please see your product-specific manual for more information on determining the type of a result set cursor in a client application.

1.20 Compression

ElevateDB uses the standard ZLib compression algorithm for compressing data such as BLOB columns and remote session requests and responses to and from an ElevateDB Server.

Copyright and Credits

The ZLib implementation in ElevateDB was originally contributed by David Martin for use in DBISAM and was modified extensively for use with ElevateDB. The following are the citations and copyrights for both the code that was contributed as well as for the ZLib algorithm itself.

© Copyright 1995-98 Jean-loup Gailly & Mark Adler
© Copyright 1998-00 Jacques Nomssi Nzali
© Copyright 2000-2001 David O. Martin

These units build upon a pascal port of the ZLib compression routines by Jean-loup Gailly and Mark Adler. The original pascal port was performed by Jacques Nomssi Nzali as contained in PasZLib which is based on ZLib 1.1.2. There are some errors in that port which have been fixed in this version. Although most of the code in this unit is derivative, there are some important changes (bug fixes). Nevertheless, this code is released as freeware with the same permissions as granted by the preceding authors (Gailly, Adler, Nzali).

1.21 Encryption

ElevateDB uses the Blowfish symmetric block cipher encryption algorithm along with the RSA Data Security, Inc. MD5 message-digest algorithm for encrypting configuration files, tables and remote session requests and responses to and from an ElevateDB Server.

Copyright and Credits

Both the Blowfish and MD5 implementations in ElevateDB were developed internally. The following are the citations and copyrights for the Blowfish and MD5 algorithms.

Blowfish Algorithm © Copyright 1993 Bruce Schneier
MD5 Algorithm © Copyright 1991-1992, RSA Data Security, Inc.

ElevateDB uses the MD5 message-digest algorithm to generate 128-bit MD5 hashes from plain-text passwords. These hashes are then used with the Blowfish 8-byte symmetric block cipher algorithm to encrypt the actual data.

1.22 Stores

Stores are used in ElevateDB to define storage areas where files can be located. Like databases, stores are defined in the ElevateDB configuration and are privileged objects, so you can control access to stores based upon the privileges that you grant or revoke from other users or roles defined in the configuration. Stores are created, altered and dropped via the CREATE STORE, ALTER STORE, DROP STORE, and RENAME STORE statements.

Types of Stores

Stores can be created as either local or remote, and they are defined as follows:

Type	Description
Local	A local store simply points to a local path that is accessible from the current process.
Remote	A remote store is a "virtual" store that is defined locally but actually points to another store on a remote ElevateDB Server. This abstraction of remote stores make the stores very useful because you can transfer files between different machines by simply copying a file from a local store to a remote store, and vice-versa.

Working with Files in Stores

Adding files to a local store can be done via the operating system itself by copying or moving files into the local path used by the local store. However, many times the files will be created using statements such as the BACKUP DATABASE, SAVE UPDATES, or EXPORT TABLE statements. These statements require a local store as the location where the files generated by these operations will be created.

You can also use the COPY FILE, RENAME FILE, and DELETE FILE statements to manipulate files in a given local or remote store. This makes stores very useful because they use the existing ElevateDB remote communications facilities and don't require any extension configuration of the operating system to set up virtual private networks (VPNs) or other elaborate setups.

For example, here's an example of using the COPY FILE statement to copy a backup file from a local store to a remote store.

```
COPY FILE "MyBackup.EDBkp" IN STORE "LocalStore"
TO "MyBackup.EDBBkp" IN STORE "RemoteStore"
```

When used in conjunction with the SAVE UPDATES, LOAD UPDATES, stores can be used for replicating updates from a local location to a remote location. Please see the Replication topic for more information on replicating updates.

1.23 Replication

Replication in ElevateDB is accomplished using several different aspects of the product and includes the capability to replicate virtually any type of data, including but not limited to, updates to databases.

The following image illustrates the general architecture of the replication of database updates in ElevateDB:

Publishing a Database

The first step in configuring a replication system for database updates in ElevateDB is to publish the database(s) that you wish to replicate updates for. You can publish a database by using the PUBLISH DATABASE statement. Once a database has been published, it will begin to log all inserts, updates, and deletes that take place so that they can be saved to an update file at a later time for replicating to other copies of the same database.

Creating a Local Store

The next step is to create a local store where the updates can be saved. You can use the CREATE STORE statement to create the local store.

Saving the Updates to a Database

Once the local store has been created, you can now save the logged updates for the database to an update file in the local store using the SAVE UPDATES statement. The frequency with which the updates are saved and replicated is completely up to you, but the general rule is that you want to save the updates more frequently as the volume of updates increases in order to minimize the synchronization time.

> **Note**
> The SAVE UPDATES statement only works with local stores. Any attempt to use a remote store with this statement will result in an error.

Creating a Remote Store

The next step is to create a remote store (or stores, for multiple remote locations) where the update file can be copied in order to replicate it to the remote location. You can use the CREATE STORE statement to create the local store. However, in order for this new remote store to be accessible, the store at the remote location that is pointed to by the remote store must have already been created at the remote location. The replication will not work until this step has been completed at each remote location.

Replicating the Update File

Once the update file has been created in the local store, you can now begin the process of replicating it to other copies of the same database. Typically, there are two types of replication used:

Replication Type	Description
Push	A push replication involves the master location copying a master update file from a local store to several remote stores, thus effectively replicating the updates to all of the remote copies of the database. The remote locations can then load the update file to complete the process of synchronizing their database copies so that they now match the database at the master location.
Pull	A pull replication involves the master location copying a master update file from a local store to several other local stores. The remote locations are then responsible for copying the update file from their designated store to their own local store. The remote locations can then load the update file to complete the process of synchronizing their database copies so that they now match the database at the master location.

One method is not necessarily superior to the other, and the choice of which to use usually revolves around whether the remote locations are constantly accessible to the master location. If they are not, such as is the case with salesman on the road with laptops, then the pull replication is the better choice.

Replicating an update file from a local store to a remote store, or vice-versa, is accomplished via the COPY FILE statement. Copying an update file from a local store to a remote store will cause ElevateDB to automatically log into the remote ElevateDB Server designated for the remote store, and transfer the update file to the remote store.

Loading the Updates into a Database

At the remote location, we will use the LOAD UPDATES statement to load the update file from a local store into the copy of the database. The update file is assumed to already be present in the local store as a result of the previous COPY FILE operation. Once the updates are loaded, the copy of the database at the remote location is now considered to be synchronized with the database at the master location.

If there are multiple update files in a local store that need to be loaded for a database, it is very important that you do not confuse the ordering of the update files and attempt to load them out-of-order. Each update file has a creation timestamp that can be used to determine which should be loaded first, second, etc. To retrieve information about the update files in a specific store, you can use the SET UPDATES STORE statement to specify the store where the update files are located, and then use a SELECT statement to query the Updates Table in the Configuration Database. The Updates table contains information about all of the update files in the store specified by the SET UPDATES STORE statement, with one row per update file.

> **Note**
> The LOAD UPDATES statement only works with local stores. Any attempt to use a remote store with this statement will result in an error.

You can use the FROM UPDATES clause of the CREATE TABLE statement to examine the contents of any update file. This is useful when you are trying to load an update file and cannot do so due to constraint violations or other types of problems.

Bi-Directional Replication

In addition to a scenario where a master location replicates its updates to remote locations, you can also publish the copies of the databases at the remote locations and replicate their updates back to the database at the master location. This is called bi-directional replication, and is very common. In order to set up a bi-directional replication system, you would execute the same steps as before, but you would also repeat the same steps for each of the remote locations.

Scheduling Replication

The actual processing of saving and loading updates, and copying the update files between different locations, can be scheduled to run as a job by using the CREATE JOB statement.

Replicating Other Types of Data

The replication capabilities in ElevateDB are not limited solely to database updates. Backups and other types of files can be replicated in a similar manner, and this is very useful for situations where you want remote locations to send backups and other important information to a master location where they will be stored in a more secure fashion than what might be possible at the remote location. It is also sometimes more efficient to use backups for the initial population of new remote locations instead of using one large update file, or a series of many update files. Being able to replicate backups is very useful in such a situation.

1.24 Row Value Constructors

Row value constructors are a special syntax used to aggregate basic expressions into a special row value that can be compared against other row values, or used as a whole in certain DML statments like INSERT and UPDATE. The syntax is as follows:

```
(<Expression>,<Expression>[,<Expression>])
```

Each expression in the row value constructor is separated by a comma (,), and a row value constructor requires that at least two expressions be specified in order for the parser to recognize that it is dealing with a row value, as opposed to a simple scalar value enclosed in parentheses. The only exception to this is when a row value constructor is used in an INSERT statement (see below).

Using Row Value Constructors in SELECT, UPDATE, and DELETE Statements

Row value constructors are very useful in SELECT, UPDATE, and DELETE statements for comparing multiple expressions in a single operation. Row value constructors can be used with any comparison operator except for the LIKE/NOT LIKE operators, as the following examples show:

```
SELECT *
FROM Orders
WHERE (CustNo,OrderNo)=(2156,1020)

SELECT *
FROM Orders INNER JOIN Items ON
(Orders.CustNo,Orders.OrderNo) = (Items.CustNo,Items.OrderNo)
```

Note
Row values are always compared from left-to-right, so all comparison operators work by comparing the first scalar value in the row value, followed by the second, and so on. The IS NULL/IS NOT NULL comparison operators work on an all-or-nothing basis, meaning that the entire row value must be NULL or NOT NULL in order for these operators to return True.

Please see the Comparison Operators topic for more information on the available comparison operators, and the Optimizer topic for more information on how ElevateDB optimizes row value constructors in expressions.

UPDATE statements can also use row value constructors in order to update more than one column at a time in the SET clause. For example:

```
UPDATE Orders
SET (ShipToState,ShipToCountry)=
   (SELECT State,Country FROM Customer WHERE CustNo=Orders.CustNo)
```

Using Row Value Constructors in INSERT Statements

Row value constructors can be used in INSERT statements to insert multiple rows in a single statement execution. In order to accomplish this, just separate each row value with a comma (,):

```
INSERT INTO Orders (OrderNo, ItemNo, QtyOrdered, UnitPrice)
VALUES (1200, 23478, 10, 30.00),
       (1200, 15453, 4, 23.00),
       (1200, 14545, 1, 89.00)
```

Note
You should be careful not to specify too many row values in a single INSERT statement. It is quite possible to exceed the parsing and memory limitations of ElevateDB if you specify hundreds of thousands of row values in a single INSERT statement.

1.25 Object Versioning

User-defined version numbers can be assigned to to certain objects in an ElevateDB configuration or database using the VERSION clause. These version numbers are specified as a DECIMAL number in the form of:

```
<MajorVersion>.<MinorVersion>
```

The major version may contain up to 19 digits, and the minor version may contain up to 4 digits.

The following DDL statements allow you to define version numbers for the associated object being created or altered:

CREATE JOB
ALTER JOB

CREATE TABLE
ALTER TABLE

CREATE VIEW
ALTER VIEW

CREATE FUNCTION
ALTER FUNCTION

CREATE PROCEDURE
ALTER PROCEDURE

1.26 Custom Attributes

Custom attributes can be assigned to most objects in an ElevateDB configuration or database using the ATTRIBUTES clause. These attributes are simply stored as a block of text along with the object to which they are assigned, and can later be retrieved by querying the system information tables in ElevateDB. It is recommended that you use a structured text format such as XML or INI (key-value pairs) in order to allow for quick and easy reading of any structured data that you wish to store as custom attributes.

The following DDL statements allow you to define custom attributes for the associated object being created or altered:

CREATE DATABASE
ALTER DATABASE

CREATE STORE
ALTER STORE

CREATE USER
ALTER USER

CREATE ROLE
ALTER ROLE

CREATE JOB
ALTER JOB

CREATE TABLE
ALTER TABLE

CREATE VIEW
ALTER VIEW

CREATE FUNCTION
ALTER FUNCTION

CREATE PROCEDURE
ALTER PROCEDURE

Chapter 2
Operators

2.1 Introduction

ElevateDB supports most standard SQL operators, which are organized by the following categories:

Boolean Operators
Comparison Operators
Arithmetic Operators
String Operators
Text Index Operators

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

2.2 Boolean Operators

The following are the boolean operators in ElevateDB, ordered by their operator precedence:

Operator	Description
NOT	Flips a boolean expression so that True becomes False, or vice-versa.
AND	Returns True if both the left and right boolean expressions are True.
OR	Returns True if either the left or right boolean expression is True.

Examples

```
-- The following SQL uses the AND Boolean
-- operator to select all shipped orders
-- placed within the last 100 days

SELECT *
FROM Orders
WHERE OrderDate >= CURRENT_DATE-100 AND
ShipDate IS NOT NULL
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

2.3 Comparison Operators

The following are the comparison operators in ElevateDB, ordered by their operator precedence:

Operator	Description
=	Returns True if both the left and right expressions are equal.
<>	Returns True if both the left and right expressions are not equal.
>	Returns True if the left expression is greater than the right expression.
>=	Returns True if the left expression is greater than or equal to the right expression.
<	Returns True if the left expression is less than the right expression.
<=	Returns True if the left expression is less than or equal to the right expression.
LIKE	Returns True if the left string expression matches the right pattern expression. The percent (%) character can be used to represent multiple unknown characters in the pattern expression, while the underline (_) character can be used to represent a single unknown character in the pattern expression. In addition, the ESCAPE clause can be used after the pattern expression to specify a single character to be used as an escape character in front of any percent or underline literal characters in the pattern expression itself. This is useful when you wish to treat the percent or underline characters as literal characters for the purposes of the comparison.
NOT LIKE	Returns True if the left string expression does not match the right pattern expression. It is the inverse of the LIKE operator. **Note** Both LIKE and NOT LIKE can correctly handle situations where a specific collation treats two characters as collating as one character, such as is the case with 'ss' in the German collations.
BETWEEN	Returns True if the left expression is between the two right expressions. The right expressions must be separated by the AND keyword.
NOT BETWEEN	Returns True if the left expression is not between the two right expressions. It is the inverse of the BETWEEN operator.
IN	Returns True if the left expression equals one of the right expressions. The right expressions are specified as a comma-delimited list of expressions enclosed in parentheses.

NOT IN	Returns True if the left expression does not equal one of the right expressions. It is the inverse of the IN operator.
IS NULL	Returns True if the left expression is null.
IS NOT NULL	Returns True if the left expression is not null. It is the inverse of the IS NULL operator.

Examples

```
-- The following uses the LIKE operator
-- to search the Notes column for the
-- occurrence of the string 'angry'

SELECT *
FROM customers
WHERE Notes LIKE '%angry%'

-- This is the same as the last search
-- except this search has been modified
-- to use a case-insensitive comparison
-- using the COLLATE clause

SELECT *
FROM customers
WHERE Notes COLLATE ANSI_CI LIKE '%angry%'

-- The following uses the LIKE operator's
-- ESCAPE clause to perform a search on a
-- phrase that includes the percent (%)
-- character

SELECT *
FROM Orders
WHERE AdjustmentNotes LIKE '20/%' ESCAPE '/'

-- The following query searches for all
-- orders placed between January 1st and
-- January 31st of 2007

SELECT *
FROM Orders
WHERE OrderDate BETWEEN DATE '2007-01-01' AND
DATE '2007-01-31'

-- The following query searches for all
-- orders placed by someone in the states
-- of California (CA) or Nevada (NV)

SELECT *
FROM Orders
WHERE State IN ('CA','NV')
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

2.4 Arithmetic Operators

The following are the arithmetic operators in ElevateDB, ordered by their operator precedence:

Operator	Description
*	Multiplies the left numeric or interval expression by the right numeric or interval expression.
/	Divides the left numeric or interval expression by the right numeric expression.
-	Subtracts the right numeric, date, time, timestamp, or interval expression from the left numeric, date, time, timestamp, or interval expression.
+	Adds the right numeric, date, time, timestamp, or interval expression to the left numeric, date, time, timestamp, or interval expression.
MOD	Returns the remainder derived from dividing the left numeric or interval expression by the right numeric or interval expression.

Date, Time, and Timestamp Subtraction

When subtracting dates, times, and timestamps, the result is always an interval value. The type of interval value depends upon what is being subtracted. The following details the interval type that you can expect with the various input types:

Expression	Interval Type
DATE - DATE	INTERVAL DAY
TIME - TIME	INTERVAL HOUR TO MSECOND
TIMESTAMP - TIMESTAMP TIMESTAMP - DATE	INTERVAL DAY TO MSECOND

In addition, you can force the resulting interval value to a specific type by enclosing the expression in parentheses and appending the interval type to the expression:

```
(<Value> - <Value>) <Interval Type>
```

Note
Unlike in most scenarios in SQL where an interval type is prefaced with the keyword INTERVAL, specifying an interval type for DATE/TIME/TIMESTAMP subtraction only requires the actual interval type without the INTERVAL keyword. It is done this way due to the fact that ElevateDB already knows that the resulting type is an interval. The only question is which type of interval is desired by the application.

The valid interval types for each type of expression are as follows:

Expression	Interval Type
DATE - DATE	YEAR MONTH YEAR TO MONTH DAY
TIME - TIME	HOUR HOUR TO MINUTE HOUR TO SECOND HOUR TO MSECOND MINUTE MINUTE TO SECOND MINUTE TO MSECOND SECOND SECOND TO MSECOND MSECOND
TIMESTAMP - TIMESTAMP TIMESTAMP - DATE	YEAR MONTH YEAR TO MONTH DAY DAY TO HOUR DAY TO MINUTE DAY TO SECOND DAY TO MSECOND HOUR HOUR TO MINUTE HOUR TO SECOND HOUR TO MSECOND MINUTE MINUTE TO SECOND MINUTE TO MSECOND SECOND SECOND TO MSECOND MSECOND

Examples

```
-- The following uses the subtraction
-- operator to show all customers that
-- have not ordered within the last 12
-- months

SELECT *
FROM Customers
WHERE (CURRENT_DATE-LastOrderDate) MONTH > 12
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

2.5 String Operators

The following are the string operators in ElevateDB, ordered by their operator precedence:

Operator	Description
+	Concatenates the right string expression to the left string expression.
||	Concatenates the right string expression to the left string expression.

Examples

```
-- The following uses the +
-- operator to format the output
-- of one of the SELECT column
-- expressions

SELECT LastName+', '+FirstName+' '+Initial AS FullName,
DateOfBirth
FROM Employees
ORDER BY FullName
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

2.6 Text Index Operators

The following are the string operators in ElevateDB, ordered by their operator precedence:

Operator	Description
CONTAINS	Returns True if the left column reference contains all of the word values specified in the right string expression (not necessarily next to each other). The asterisk (*) can be used to specify a trailing wildcard.
DOES NOT CONTAIN	Returns True if the left column reference does not contain all of the word values specified in the right string expression. It is the inverse of the CONTAINS operator.
CONTAINS ANY	Returns True if the left column reference contains any of the word values specified in the right string expression (not necessarily next to each other). The asterisk (*) can be used to specify a trailing wildcard.
DOES NOT CONTAIN ANY	Returns True if the left column reference does not contain any of the word values specified in the right string expression. It is the inverse of the CONTAINS ANY operator.

Examples

```
-- The following uses the text index
-- on the Notes column to find any rows
-- where the word 'angry', 'anger', or 'angered'
-- appears.

SELECT *
FROM customers
WHERE Notes CONTAINS ANY 'angry anger angered'

-- The following uses the text index
-- on the Text column to find any rows
-- where the words 'little', 'red',
-- 'riding', and 'hood' appear

SELECT *
FROM Documents
WHERE Text CONTAINS 'little red riding hood'
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details

CONTAINS DOES NOT CONTAIN CONTAINS ANY DOES NOT CONTAIN ANY	The CONTAINS, CONTAINS ANY, DOES NOT CONTAIN, and DOES NOT CONTAIN ANY operators are ElevateDB extensions

This page intentionally left blank

Chapter 3
Types

3.1 Introduction

ElevateDB supports most standard SQL types, which are organized by the following categories:

Exact Numeric Types
Approximate Numeric Types
String Types
Binary Types
Date and Time Types
Interval Types
Boolean Types

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
Collection/Array Types	ElevateDB does not support the creation or use of collection types, and only supports the use of array types in SQL/PSM routines.
RowTypes	ElevateDB does not support the creation or use of row types.
User-Defined Types	ElevateDB does not support the creation or use of user-defined types.
Reference Types	ElevateDB does not support the creation or use of reference types.
Locators	ElevateDB does not support the creation or use of locators for large object types.

3.2 Exact Numeric Types

Exact numeric types are used when you wish to store a numeric value in its exact representation without accumulating rounding errors. Specifically, NUMERIC and DECIMAL types allow you to specify the scale so that any numeric values with a greater scale are automatically rounded to the specified scale using the bankers rounding algorithm, which simply says that any digits past the specified scale are rounded to the specified scale using the following logic:

1) Round towards the nearest integer.

2) If there are two nearest integers, then round the value towards the even integer value.

Type	Description
INTEGER INT	A 32-bit,signed integer value
SMALLINT	A 16-bit,signed integer value
BIGINT	A 64-bit,signed integer value
NUMERIC[(<Precision>[,<Scale>])]	A 64-bit exact numeric value a specifc precision and scale, or fractional digits. Regardless of the precision specified, ElevateDB always uses an implementation-defined precision of 19 digits. The maximum scale is 4 digits. This type is ideal for representing monetary values. If the scale is not specified, then the default is 0. If the precision is not specified, then the default is 19 digits.
DECIMAL[(<Precision>[,<Scale>])]	An 64-bit exact numeric value a specifc precision and scale, or fractional digits. Regardless of the precision specified, ElevateDB always uses an implementation-defined precision of 19 digits. The maximum scale is 4 digits. This type is ideal for representing monetary values. If the scale is not specified, then the default is 0. If the precision is not specified, then the default is 19 digits.

Literals

Exact numeric literals use the period (.) as the decimal point character, the minus (-) as the negative sign character, the plus (+) as the positive sign character, and scientific notation is not supported.

Literal Examples

```
-- This example specifies an INTEGER literal

SELECT * FROM Customer WHERE CustNo=1206

-- This example specifies a DECIMAL literal

SELECT * FROM Customer WHERE BalanceDue > 1000.00
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
NUMERIC Type	ElevateDB translates any NUMERIC into the equivalent DECIMAL type. Also, it always uses the implementation-defined precision of 19 digits instead of the precision that is specified in the type definition.

3.3 Approximate Numeric Types

Approximate numeric types are used when you wish to store a numeric value in an approximate representation with a floating decimal point. Using approximate numeric types can cause rounding errors due to the fact that certain numbers such as 0.33 cannot be accurately represented using floating-point precision.

Type	Description
DOUBLE PRECISION	A 64-bit, floating-point numeric value with a maximum precision of 16 digits.
FLOAT[(<Precision>)]	A 64-bit, floating-point numeric value with a maximum precision of 16 digits. The precision is ignored if specified.

Literals

Approximate numeric literals use the period (.) as the decimal point character, the minus (-) as the negative sign character, the plus (+) as the positive sign character, and scientific notation is supported via E (e or E) as the exponent character followed by a plus (+) or minus (-) character and the actual exponent value.

Literal Examples

```
-- This example specifies a FLOAT literal

SELECT * FROM Orders WHERE Amount > 100.00

-- This example specifies a FLOAT literal using
-- scientific notation

SELECT * FROM Planets WHERE Distance > 100E+10
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
REAL Type	ElevateDB does not support the REAL type. Use the DOUBLE PRECISION or FLOAT type instead.

3.4 String Types

String types are used when you wish to store a character string of a fixed or variable length, including very large character strings.

Type	Description
CHARACTER[(<Length>)] CHAR[(<Length>)]	A string value with a fixed number of characters. If the length of the string value is not specified, then a length of 1 is used. The maximum length is 1024 characters. When assigning a value to a CHAR type value that is smaller in length than the specified length, the value being assigned will be padded with spaces to the specified length. For example, if you have a column defined as: MyColumn CHAR(20) If you were to assign the value 'Test' to the column, then the MyColumn column would contain the value 'Test'+<16 Spaces> after the assignment.
CHARACTER VARYING(<Length>) VARCHAR(<Length>)	A string value with a variable number of characters. The length of the string value must always be specified. The maximum length is 1024 characters. Contrary to the CHARACTER type, this type does not pad the string value with spaces.
GUID	A string value that has an exact length of 38 characters. A GUID value is treated the same as a VARCHAR value.
CHARACTER LARGE OBJECT CLOB	A large, variable-length string value with a maximum size of 2GB.

Literals

String literals use the single quote (') character to identify themselves as such. Any single quotes enclosed inside of the literal must be escaped by prefacing them with another single quote. In addition, single character constants may be specified using their literal value or by prefacing their ordinal character set position with the pound sign (#) character.

Literal Examples

```
-- This example specifies a VARCHAR literal

SELECT * FROM Customer WHERE Name = 'Acme Boot Makers'

-- This example specifies a VARCHAR literal with
-- embedded quotes

SELECT * FROM Customer WHERE Name = 'Bill''s Shoes'

-- This example specifies two CHAR literals (a carriage
-- return and line feed) using pound sign (#) notation
```

```
SELECT * FROM Documentation WHERE Notes LIKE '%'+#13+#10+'%'
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
NATIONAL Types	ElevateDB does not support the NATIONAL versions of the CHARACTER, CHARACTER VARYING (VARCHAR), or CHARACTER LARGE OBJECT (CLOB) types. See the Internationalization topic for more information.
CLOB Type	ElevateDB does not support the specification of a default size in KB, MB, or GB for large object types.
GUID Type	ElevateDB adds the GUID type as an extended type.

3.5 Binary Types

Binary types are used when you wish to store a series of bytes with a fixed or variable length, including very large series of bytes.

Type	Description
BYTE[(<Length>)]	A binary value with a fixed number of bytes. If the length of the binary value is not specified, then a length of 1 is used. The maximum length is 1024 bytes. When assigning a value to a BYTE type value that is smaller in length than the specified length, the value being assigned will be padded with NULL bytes (0) to the specified length. For example, if you have a column defined as: MyColumn BYTE(8) If you were to assign the value 0x00 0x01 0x02 0x03 to the column, then the MyColumn column would contain the value 0x00 0x01 0x02 0x03 0x00 0x00 0x00 0x00 after the assignment.
BYTE VARYING(<Length>) VARBYTE(<Length>)	A binary value with a variable number of bytes. The length of the binary value must always be specified. The maximum length is 1024 bytes. Contrary to the BYTE type, this type does not pad the binary value with NULL bytes (0).
BINARY LARGE OBJECT BLOB	A large, variable-length binary value with a maximum size of 2GB.

Literals

Binary literals use the 'X' character along with the single quote (') character to identify themselves as such. The contents of a binary literal consists of the bytes that make up the value encoded in hexadecimal form, such that each byte is represented by two characters from the range of '00' (0) to 'FF' (255).

Literal Examples

```
-- This example specifies a BYTE literal

SELECT * FROM Instruments WHERE Data = X'01F21028'
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details

BLOB Type	ElevateDB does not support the specification of a default size in KB, MB, or GB for large object types.
BYTE and BYTE VARYING Types	ElevateDB adds the BYTE and BYTE VARYING (VARBYTE) types as extended types.

3.6 Date and Time Types

Date and time types are used when you wish to store a date, time, or timestamp (date and time) value.

Type	Description
DATE	A date value containing a year, month, and day.
TIME	A time value containing an hour, minute, second, and millisecond.
TIMESTAMP	A combined date and time value containing a year, month, and day along with an hour, minute, second, and millisecond.

Literals

Date and time literals use the IS0 8601 standard which dictates the following formats:

Format	Description
DATE 'YYYY-MM-DD'	YYYY is the 4-digit year, MM is the 2-digit month (1-based), and DD is the 2-digit day (1-based).
TIME 'HH:MM [:SS [.ZZZ AM/PM]]'	HH is the 2-digit hour (0-based), MM is the 2-digit minute (0-based), SS is the 2-digit second (0-based), ZZZ is the 3-digit millisecond, or fraction of a second, and AM/PM is the 12-hour time format specifier (as opposed to the default 24-hour time format). The SS, ZZZ, and AM/PM portions of times are optional.
TIMESTAMP '<Date> <Time>'	Timestamp literals use the date and time formats with a space between them.

Literal Examples

```
-- This example specifies a date literal

SELECT * FROM Orders
WHERE OrderDate BETWEEN DATE '2006-01-01' AND DATE '2006-01-31'

-- This example specifies a TIME literal
-- using 24-hour format

SELECT * FROM TimeClockEntries
WHERE PunchInTime > TIME '16:00'

-- This example specifies a TIME literal
-- using 12-hour format

SELECT * FROM TimeClockEntries
WHERE PunchInTime > TIME '4:00 PM'
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
TIME and TIMESTAMP Types	ElevateDB supports including the AM or PM (case-insensitive) indicator for indicating 12-hour time formats.
TIME and TIMESTAMP Types	ElevateDB does not support specifying the precision of a time or timestamp value and always includes the millisecond portion.
WITH TIME ZONE Types	ElevateDB does not support the time zone versions of the TIME and TIMESTAMP types.

3.7 Interval Types

Interval types are used to represent the difference between two dates, times, or timestamps. There are two classes of interval values:

Year-Month Intervals
Day-Time Intervals

These two classes are not type-compatible and cannot be assigned to each other or cast between each other.

Type	Description
INTERVAL YEAR	An interval value representing the number of years between two date values.
INTERVAL MONTH	An interval value representing the number of months between two date values.
INTERVAL YEAR TO MONTH	An interval value representing the number of years and months (remainder) between two date values.
INTERVAL DAY	An interval value representing the number of days between two date values.
INTERVAL HOUR	An interval value representing the number of hours between two time values.
INTERVAL MINUTE	An interval value representing the number of minutes between two time values.
INTERVAL SECOND	An interval value representing the number of seconds between two time values.
INTERVAL MSECOND	An interval value representing the number of milliseconds between two time values.
INTERVAL DAY TO HOUR	An interval value representing the number of days and hours between two timestamp values.
INTERVAL DAY TO MINUTE	An interval value representing the number of days, hours, and minutes between two timestamp values.
INTERVAL DAY TO SECOND	An interval value representing the number of days, hours, minutes, and seconds between two timestamp values.
INTERVAL DAY TO MSECOND	An interval value representing the number of days, hours, minutes, seconds, and milliseconds between two timestamp values.
INTERVAL HOUR TO MINUTE	An interval value representing the number of hours and minutes between two time values.
INTERVAL HOUR TO SECOND	An interval value representing the number of hours, minutes, and seconds between two time values.
INTERVAL HOUR TO MSECOND	An interval value representing the number of hours, minutes, seconds, and milliseconds between two time values.
INTERVAL MINUTE TO SECOND	An interval value representing the number of minutes and seconds between two time values.

INTERVAL MINUTE TO MSECOND	An interval value representing the number of minutes, seconds, and milliseconds between two time values.
INTERVAL SECOND TO MSECOND	An interval value representing the number of seconds and milliseconds between two time values.

Literals

Interval literals are specified using the following formats:

Format	Description
INTERVAL 'Y' YEAR	Y is the years.
INTERVAL 'M' MONTH	M is the months.
INTERVAL 'Y-M' YEAR TO MONTH	Y is the years and M is the months.
INTERVAL 'D' DAY	D is the days.
INTERVAL 'H' HOUR	H is the hours.
INTERVAL 'M' MINUTE	M is the minutes.
INTERVAL 'S' SECOND	S is the seconds.
INTERVAL 'Z' MSECOND	Z is the milliseconds.
INTERVAL 'D H' DAY TO HOUR	D is the days and H is the hours.
INTERVAL 'D H:M' DAY TO MINUTE	D is the days, H is the hours, and M is the minutes.
INTERVAL 'D H:M:S' DAY TO SECOND	D is the days, H is the hours, M is the minutes, and S is the seconds.
INTERVAL 'D H:M:S.Z' DAY TO MSECOND	D is the days, H is the hours, M is the minutes, S is the seconds, and Z is the milliseconds.
INTERVAL 'H:M' HOUR TO MINUTE	H is the hours and M is the minutes.
INTERVAL 'H:M:S' HOUR TO SECOND	H is the hours, M is the minutes, and S is the seconds.
INTERVAL 'H:M:S.Z' HOUR TO MSECOND	H is the hours, M is the minutes, S is the seconds, and Z is the milliseconds.
INTERVAL 'M:S' MINUTE TO SECOND	M is the minutes and S is the seconds.
INTERVAL 'M:S.Z' MINUTE TO MSECOND	M is the minutes, S is the seconds, and Z is the milliseconds.
INTERVAL 'S.Z' SECOND TO MSECOND	S is the seconds, and Z is the milliseconds.

Literal Examples

```
-- This example specifies a YEAR interval literal
```

```
SELECT * FROM Orders
WHERE (OrderDate + INTERVAL '1' YEAR) BETWEEN
DATE '2006-01-01' AND DATE '2006-01-31'

-- This example specifies a DAY interval literal

SELECT * FROM Orders
WHERE (ShipDate - OrderDate) > INTERVAL '2' DAY

-- This example specifies an HOUR interval literal

SELECT * FROM TimeClockEntries
WHERE (PunchOutTime - PunchInTime) > INTERVAL '8' HOUR
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
INTERVAL MSECOND Type	This is an ElevateDB extension to the day-time interval data types.
Interval Precisions	ElevateDB does not support specifying the precision of interval values and always uses 4 digits for years, 1-2 digits for months, 1-2 digits for days, 1-2 digits for hours, 1-2 digits for minutes, 1-2 digits for seconds, and 1-3 digits for milliseconds.

3.8 Boolean Types

Boolean types are used to represent the values of True or False.

Type	Description
BOOLEAN BOOL	A logical true/false value.

Literals

Boolean literals are expressed as the literals TRUE and FALSE (case-insensitive) or 1 and 0 for TRUE and FALSE, respectively.

Literal Examples

```
-- This example specifies a BOOLEAN literal

SELECT * FROM Customer WHERE SpecialAttention=TRUE
```

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

3.9 Type Promotion

Type promotion can occur when the following operators or functions are used:

Operators

UNION, INTERSECT, or EXCEPT in a SELECT statement

When one of these operators is used, the type, collation, length, and scale of the column in the result set is determined by the type promotion rules outlined below using the corresponding columns of each SELECT statement involved as the inputs to the type promotion process.

Functions

CASE
IF
IFNULL
NULLIF
COALESCE

> **Note**
> CASE is an operator, not a function, but it behaves like a function.

When one of these functions is used, the type, collation, length, and scale of their result values are determined by the type promotion rules outlined below using their multiple input arguments as the inputs to the type promotion process.

Rules

The rules for type promotion are as follows:

The precedence for CHAR, VARCHAR, and CLOB types is:

CLOB (highest)
VARCHAR
CHAR

If the resulting type is a CHAR or VARCHAR, then the resulting length is the greatest of all of the input lengths.

The resulting collation is determined by the input that was selected according to the type precedence above.

The precedence for BYTE, VARBYTE, and BLOB types is:

BLOB (highest)
VARBYTE
BYTE

If the resulting type is a BYTE or VARBYTE, then the resulting length is the greatest of all of the input

lengths.

The precedence for SMALLINT, INTEGER, LARGEINT, DECIMAL, and FLOAT types is:

FLOAT (highest)
DECIMAL
LARGEINT
INTEGER
SMALLINT

If the resulting type is a DECIMAL, then the resulting scale is the greatest of all of the input scales.

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
None	

Chapter 4
System Information

4.1 Introduction

ElevateDB maintains information about both the system configuration and each of the databases contained within a given configuration. The information about the system configuration is stored in a special system-generated Configuration database.

Each database, including the Configuration database, contains two schemas:

Schema	Description
Information	Contains tables describing the objects contained within the database. See the Information Schema topic for more information on the tables contained with the Information schema for each database.
Default	Contains the actual objects contained within the database.

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
Configuration Database	The SQL standard does not specify any system-generated databases holding system information.
Schemas	The SQL standard dictates a different name for the information schema along with the ability to define more than two schemas. In addition, the tables contained within the ElevateDB Informaton schema are different from the standard.

4.2 Configuration Database

The tables that make up the Configuration database are as follows:

DataTypes Table
Collations Table
Modules Table
TextFilters Table
WordGenerators Table
Migrators Table
MigratorParams Table
LogEvents Table
Backups Table
Updates Table
ServerSessions Table
ServerSessionLocks Table
ServerSessionStatistics Table
Users Table
Roles Table
UserRoles Table
Databases Table
DatabasePrivileges Table
Jobs Table
Stores Table
StorePrivileges Table
Files Table

The contents of these tables are stored on disk in the configuation file for ElevateDB. You can find out more information on how to modify the configuration file settings for ElevateDB in your product-specific manual.

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
Extended Objects	Databases, modules, text filters, word generators, log events, migrators, backups, updates, server sessions, server session locks, server session statistics, jobs, stores, and files are ElevateDB extensions, and these objects are not defined in the SQL 2003 standard.

4.3 Collations Table

Structure

```
CREATE TABLE "Collations"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI"
)

CREATE INDEX "Name" ON "Collations"
("Name")
```

Description

The collations in ElevateDB are dynamic and this table reflects the available installed collations (locales) in the operating system. See the Internationalization topic for more information on collations.

Related DDL Statements

Statement	Description
None	

4.4 DataTypes Table

Structure

```
CREATE TABLE "DataTypes"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"BinarySize" INTEGER
)

CREATE INDEX "Name" ON "DataTypes"
("Name")
```

Description

This table contains the ElevateDB data types. The data types in ElevateDB are fixed and user-defined data types are not permitted. See the Types topic for more information on the available data types in ElevateDB.

Related DDL Statements

Statement	Description
None	

4.5 Modules Table

Structure

```
CREATE TABLE "Modules"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"FilePath" VARCHAR(255) COLLATE "ANSI_CI",
"FileVersion" VARCHAR(15) COLLATE "ANSI_CI",
"FileDescription" CLOB COLLATE "ANSI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"CharacterSet" VARCHAR(30) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Modules"
("Name")
```

Description

This table contains all of the defined external modules in an ElevateDB configuration. These external modules consist of the compiled implementation of text filters, word generators, migrators, or modules used for stored functions or procedures.

The Type column indicates the purpose of the module.

The CharacterSet column represents the character set used by the module, and the values are as follows:

Character Set	Description
ANSI	The module uses the ANSI character set
Unicode	The module uses the Unicode character set

You can find out more information on how to create such external modules for ElevateDB in your product-specific manual.

Related DDL Statements

Statement	Description
CREATE MODULE	Creates (registers) a new external module
ALTER MODULE	Alters an existing external module
DROP MODULE	Drops an existing external module
RENAME MODULE	Renames an existing external module

4.6 TextFilters Table

Structure

```
CREATE TABLE "TextFilters"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"ModuleName" VARCHAR(255) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "TextFilters"
("Name")
```

Description

This table contains the defined text filters in an ElevateDB configuration. The text filters are used by the full-text indexing functionality in ElevateDB to filter out text before it is indexed. Text filters are defined with a specific type, such as 'html' or 'rtf'. In addition, each text index defined for a given table can specify a filter column. The values in this filter column are used by the text index in conjunction with the type assigned to each text filter to determine which text filter to use for filtering the text before it is indexed. You can find out more about text indexing in the Text Indexing.

Related DDL Statements

Statement	Description
CREATE TEXT FILTER	Creates a new text filter
ALTER TEXT FILTER	Alters an existing text filter
DROP TEXT FILTER	Drops an existing text filter
RENAME TEXT FILTER	Renames an existing text filter

4.7 WordGenerators Table

Structure

```
CREATE TABLE "WordGenerators"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"ModuleName" VARCHAR(255) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "WordGenerators"
("Name")
```

Description

This table contains the defined word generators in an ElevateDB configuration. Word generators are used by the full-text indexing functionality in ElevateDB to parse text and separate the text into distinct words. This is important when you wish to parse the text in a different fashion than the default in ElevateDB. You can find out more about text indexing in the Text Indexing topic.

Related DDL Statements

Statement	Description
CREATE WORD GENERATOR	Creates a new word generator
ALTER WORD GENERATOR	Alters an existing word generator
DROP WORD GENERATOR	Drops an existing word generator
RENAME WORD GENERATOR	Renames an existing word generator

4.8 Migrators Table

Structure

```
CREATE TABLE "Migrators"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"ModuleName" VARCHAR(255) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Migrators"
("Name")
```

Description

This table contains the defined migrators in an ElevateDB configuration. Migrators are used by ElevateDB to migrate a database from an external database to ElevateDB. You can find out more about migrating databases in the Migrating Databases topic.

Related DDL Statements

Statement	Description
CREATE MIGRATOR	Creates a new migrator
ALTER MIGRATOR	Alters an existing migrator
DROP MIGRATOR	Drops an existing migrator
RENAME MIGRATOR	Renames an existing migrator

4.9 MigratorParams Table

Structure

```
CREATE TABLE "MigratorParams"
(
"MigratorName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Mode" VARCHAR(15) COLLATE "ANSI_CI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"Length" INTEGER,
"Precision" INTEGER,
"Scale" INTEGER,
"DefaultValue" VARCHAR(60) COLLATE "ANSI",
"OrdinalPos" INTEGER
)

CREATE INDEX "MigratorName" ON "MigratorParams"
("MigratorName")

CREATE INDEX "Name" ON "MigratorParams"
("Name")
```

Description

The migrator parameters are dynamic in ElevateDB and this table reflects the migrator parameters present in the current migrator. The migrator can be changed or modified using the SET MIGRATOR statement. You can find out more about migrating databases in the Migrating Databases topic.

The Mode column values are as follows:

Mode	Description
Unknown	The parameter type is unknown
In	The parameter is an input parameter
Out	The parameter is an output parameter
InOut	The parameter is both an input and output parameter

Related DDL Statements

Statement	Description
None	

4.10 LogEvents Table

Structure

```
CREATE TABLE "LogEvents"
(
"Category" VARCHAR(15) COLLATE "ANSI_CI",
"Function" VARCHAR(40) COLLATE "ANSI_CI",
"ErrorCode" INTEGER,
"Description" CLOB COLLATE "ANSI",
"LogTimeStamp" TIMESTAMP,
"Version" DECIMAL(19,2),
"Build" INTEGER,
"ProductType" VARCHAR(40) COLLATE "ANSI_CI",
"User" VARCHAR(40) COLLATE "ANSI_CI",
"Process" VARCHAR(40) COLLATE "ANSI_CI",
"ThreadID" INTEGER,
"SessionID" INTEGER,
"SessionName" VARCHAR(40) COLLATE "ANSI_CI",
"SessionDescription" CLOB COLLATE "ANSI",
"IPAddress" VARCHAR(16) COLLATE "ANSI_CI",
"Encrypted" BOOLEAN
)

CREATE INDEX "Category" ON "LogEvents"
("Category")

CREATE INDEX "Version" ON "LogEvents"
("Version")

CREATE INDEX "User" ON "LogEvents"
("User")

CREATE INDEX "Process" ON "LogEvents"
("Process")
```

Description

This table contains the events currently logged by ElevateDB. The log events are generated by ElevateDB at various times, and can be informational messages, warning messages, or error messages. The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

Related DDL Statements

Statement	Description
None	

4.11 Backups Table

Structure

```
CREATE TABLE "Backups"
(
"Name" VARCHAR(255) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"DatabaseName" VARCHAR(40) COLLATE "ANSI_CI",
"DatabasePath" VARCHAR(255) COLLATE "ANSI_CI",
"CreatedOn" TIMESTAMP,
"CreatedBy" VARCHAR(40) COLLATE "ANSI",
"CompressionLevel" INTEGER,
"Size" BIGINT,
"IncludesCatalog" BOOLEAN,
"NumTables" INTEGER,
"Tables" CLOB COLLATE "ANSI"
)

CREATE INDEX "Name" ON "Backups"
("Name")
```

Description

The backups are dynamic in ElevateDB and this table reflects the backup files present in the current backups store. The backups store can be changed or modified using the SET BACKUPS STORE statement.

Related DDL Statements

Statement	Description
SET BACKUPS STORE	Sets the current backups store

4.12 Updates Table

Structure

```
CREATE TABLE "Updates"
(
"Name" VARCHAR(255) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"DatabaseName" VARCHAR(40) COLLATE "ANSI_CI",
"DatabasePath" VARCHAR(255) COLLATE "ANSI_CI",
"CreatedOn" TIMESTAMP,
"CreatedBy" VARCHAR(40) COLLATE "ANSI",
"CompressionLevel" INTEGER,
"Size" BIGINT,
"NumTables" INTEGER,
"Tables" CLOB COLLATE "ANSI"
)

CREATE INDEX "Name" ON "Updates"
("Name")
```

Description

The updates are dynamic in ElevateDB and this table reflects the update files present in the current updates store. The updates store can be changed or modified using the SET UPDATES STORE statement.

Related DDL Statements

Statement	Description
SET UPDATES STORE	Sets the current updates store

4.13 FileIOStatistics Table

Structure

```
CREATE TABLE "FileIOStatistics"
(
"FileName" VARCHAR(40) COLLATE "ANSI_CI",
"BlockSize" INTEGER,
"MaxSize" BIGINT,
"Hits" BIGINT,
"Misses" BIGINT,
"HitRatio" DECIMAL(19,2),
"Reads" BIGINT,
"BytesRead" BIGINT,
"AvgRead" DECIMAL(19,2),
"Writes" BIGINT,
"BytesWritten" BIGINT,
"AvgWrite" DECIMAL(19,2),
"TotalAllocated" BIGINT,
"TotalDirty" BIGINT,
"TotalFlushes" BIGINT
)

CREATE INDEX "FileName" ON "FileIOStatistics"
("FileName")
```

Description

This table reflects the global file I/O buffering statistics for ElevateDB. This information is useful in determining the efficiency of the global file I/O buffering in ElevateDB and whether the file I/O buffering settings may need to be changed. For example, a large number of reads for a file may indicate that you may need to increase the amount of memory allocated for the file. Please see the Buffering and Caching topic for more information.

The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

Related DDL Statements

Statement	Description
None	

4.14 SessionStatistics Table

Structure

```
CREATE TABLE "SessionStatistics"
(
"SessionID" INTEGER,
"SessionName" VARCHAR(40) COLLATE "ANSI_CI",
"DatabaseName" VARCHAR(40) COLLATE "ANSI_CI",
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"BufferMgr" VARCHAR(40) COLLATE "ANSI_CI",
"MaxBufferSize" INTEGER,
"CurrentBufferSize" INTEGER,
"Hits" BIGINT,
"Misses" BIGINT,
"HitRatio" DECIMAL(19,2),
"Reads" BIGINT,
"BytesRead" BIGINT,
"Writes" BIGINT,
"BytesWritten" BIGINT
)

CREATE INDEX "SessionID" ON "SessionStatistics"
("SessionID")

CREATE INDEX "SessionName" ON "SessionStatistics"
("SessionName")

CREATE INDEX "DatabaseName" ON "SessionStatistics"
("DatabaseName")

CREATE INDEX "TableName" ON "SessionStatistics"
("TableName")
```

Description

This table reflects the table buffer manager statistics for all of the sessions that are present in the ElevateDB engine. This information is useful in determining how efficient the ElevateDB engine is with respect to I/O, and also how much I/O is actually taking place. For example, a large number of reads for the row buffer manager for a table may indicate that you have an un-optimized query or filter present in your application that is causing an inordinate number of row reads. The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

The BufferMgr column values are as follows:

BufferMgr	Description

Row	The statistics are for the row buffer manager for the table
Index page	The statistics are for the index page buffer manager for the table
BLOB block	The statistics are for the BLOB block buffer manager for the table
Published updates log block	The statistics are for the published updates log block buffer manager for the table

Related DDL Statements

Statement	Description
None	

4.15 LoggedStatements Table

Structure

```
CREATE TABLE "LoggedStatements"
(
"ExecutedOn" TIMESTAMP,
"User" VARCHAR(40) COLLATE "ANSI_CI",
"Process" VARCHAR(40) COLLATE "ANSI_CI",
"SessionID" INTEGER,
"SessionName" VARCHAR(40) COLLATE "ANSI_CI",
"SessionDescription" CLOB COLLATE "ANSI",
"DatabaseName" VARCHAR(40) COLLATE "ANSI_CI",
"SQL" CLOB COLLATE "ANSI_CI",
"ExecutionTime" FLOAT,
"RowsAffected" INTEGER
)
```

Description

This table contains the SQL statements that have been logged while SQL statement logging was enabled for ElevateDB via the ENABLE STATEMENT LOGGING statement. This information is useful in determining which SQL statements are causing performance issues in the ElevateDB engine or ElevateDB Server.

> **Note**
> The SQL statement logging is engine-wide functionality, which means that this table represents the slowest SQL statements across all sessions, not just the current session that is querying this table.

The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

Related DDL Statements

Statement	Description
ENABLE STATEMENT LOGGING	Enables SQL statement logging
DISABLE STATEMENT LOGGING	Disables SQL statement logging

4.16 ServerSessions Table

Structure

```
CREATE TABLE "ServerSessions"
(
"ID" INTEGER,
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Created" TIMESTAMP,
"LastConnected" TIMESTAMP,
"Connected" BOOLEAN,
"Encrypted" BOOLEAN,
"IPAddress" VARCHAR(16) COLLATE "ANSI_CI",
"User" VARCHAR(40) COLLATE "ANSI_CI",
"Process" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "ID" ON "ServerSessions"
("ID")

CREATE INDEX "Name" ON "ServerSessions"
("Name")
```

Description

This table reflects the sessions that are present on the ElevateDB server that the current session is logged in to. The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

Related DDL Statements

Statement	Description
DISCONNECT SERVER SESSION	Disconnects an existing server session
REMOVE SERVER SESSION	Removes an existing server session

4.17 ServerSessionLocks Table

Structure

```
CREATE TABLE "ServerSessionLocks"
(
"SessionID" INTEGER,
"SessionName" VARCHAR(40) COLLATE "ANSI_CI",
"DatabaseName" VARCHAR(40) COLLATE "ANSI_CI",
"ObjectName" VARCHAR(40) COLLATE "ANSI_CI",
"ObjectType" VARCHAR(15) COLLATE "ANSI_CI",
"LockType" VARCHAR(15) COLLATE "ANSI_CI",
"Number" INTEGER
)

CREATE INDEX "SessionID" ON "ServerSessionLocks"
("SessionID")

CREATE INDEX "SessionName" ON "ServerSessionLocks"
("SessionName")

CREATE INDEX "DatabaseName" ON "ServerSessionLocks"
("DatabaseName")
```

Description

This table reflects all locks currently acquired by all of the sessions that are present on the ElevateDB server that the current session is logged in to. The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

The LockType column values are as follows:

LockType	Description
Shared	The lock is a shared table open lock
Exclusive	The lock is an exclusive table open lock
Read	The lock is a table read lock
Write	The lock is a table write lock
Transaction	The lock is a transaction table lock
Row	The lock is a table row lock

Related DDL Statements

Statement	Description
DISCONNECT SERVER SESSION	Disconnects an existing server session
REMOVE SERVER SESSION	Removes an existing server session

4.18 ServerSessionStatistics Table

Structure

```
CREATE TABLE "ServerSessionStatistics"
(
"SessionID" INTEGER,
"SessionName" VARCHAR(40) COLLATE "ANSI_CI",
"DatabaseName" VARCHAR(40) COLLATE "ANSI_CI",
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"BufferMgr" VARCHAR(40) COLLATE "ANSI_CI",
"MaxBufferSize" INTEGER,
"CurrentBufferSize" INTEGER,
"Hits" BIGINT,
"Misses" BIGINT,
"HitRatio" DECIMAL(19,2),
"Reads" BIGINT,
"BytesRead" BIGINT,
"Writes" BIGINT,
"BytesWritten" BIGINT
)

CREATE INDEX "SessionID" ON "ServerSessionStatistics"
("SessionID")

CREATE INDEX "SessionName" ON "ServerSessionStatistics"
("SessionName")

CREATE INDEX "DatabaseName" ON "ServerSessionStatistics"
("DatabaseName")

CREATE INDEX "TableName" ON "ServerSessionStatistics"
("TableName")
```

Description

This table reflects the table buffer manager statistics for all of the sessions that are present on the ElevateDB server that the current session is logged in to. This information is useful in determining how efficient the ElevateDB Server is with respect to I/O, and also how much I/O is actually taking place. For example, a large number of reads for the row buffer manager for a table may indicate that you have an un-optimized query or filter present in your application that is causing an inordinate number of row reads. The current user must be granted the system-defined Administrators role in order to view this table. Please see the User Security topic for more information.

The BufferMgr column values are as follows:

BufferMgr	Description

Row	The statistics are for the row buffer manager for the table
Index page	The statistics are for the index page buffer manager for the table
BLOB block	The statistics are for the BLOB block buffer manager for the table
Published updates log block	The statistics are for the published updates log block buffer manager for the table

Related DDL Statements

Statement	Description
DISCONNECT SERVER SESSION	Disconnects an existing server session
REMOVE SERVER SESSION	Removes an existing server session

4.19 Users Table

Structure

```
CREATE TABLE "Users"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Password" VARCHAR(40) COLLATE "ANSI",
"PasswordLastChanged" TIMESTAMP,
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Users"
("Name")
```

Description

This table contains the defined users in an ElevateDB configuration. Each user may be assigned various roles as well as granted individual privileges. In each new ElevateDB configuration, there is one pre-defined System user that cannot be modified at all, and there is one pre-defined Administrator user that can be modified or deleted. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
CREATE USER	Creates a new user
ALTER USER	Alters an existing user
DROP USER	Drops an existing user
RENAME USER	Renames an existing user

4.20 Roles Table

Structure

```
CREATE TABLE "Roles"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Roles"
("Name")
```

Description

This table contains the defined roles in an ElevateDB configuration. Each role may be assigned to any defined user except for the System user, and each role can be granted privileges. Roles can make administration must simpler by allowing one to grant privileges to a fixed set of roles, and then assign the roles to individual users as necessary. The privileges granted to an individual user along with the privileges inherited from roles that have been granted to the same user, are collectively referred to as the effective privileges for the user. You can find out more information on effective privileges in the User Security topic.

Related DDL Statements

Statement	Description
CREATE ROLE	Creates a new role
ALTER ROLE	Alters an existing role
DROP ROLE	Drops an existing role
RENAME ROLE	Renames an existing role

4.21 UserRoles Table

Structure

```
CREATE TABLE "UserRoles"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "UserRoles"
("Name")

CREATE INDEX "GrantedTo" ON "UserRoles"
("GrantedTo")
```

Description

This table contains the roles assigned to the users in an ElevateDB configuration. You can find out more about users and roles in the User Security topic.

Related DDL Statements

Statement	Description
GRANT ROLES	Grants roles to an existing user
REVOKE ROLES	Revokes roles from an existing user

4.22 Databases Table

Structure

```
CREATE TABLE "Databases"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Path" VARCHAR(255) COLLATE "ANSI_CI",
"InMemory" BOOLEAN,
"EncryptedCatalog" BOOLEAN,
"CatalogCharacterSet" VARCHAR(30) COLLATE "ANSI_CI",
"CatalogVersion" DECIMAL(19,2),
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Databases"
("Name")
```

Description

This table containes the defined databases in an ElevateDB configuration. Each database in ElevateDB is defined with path information that determines the folder where the database will store its catalog and table files.

The CatalogCharacterSet column represents the character set used by the database, and the values are as follows:

Character Set	Description
ANSI	The module database the ANSI character set
Unicode	The database uses the Unicode character set

> **Note**
> The CatalogCharacterSet and CatalogVersion columns are populated by reading the actual database catalog on disk. Please see your product-specific manual for the relevant settings that allow you to turn this feature on or off.

Related DDL Statements

Statement	Description

CREATE DATABASE	Creates a new database
ALTER DATABASE	Alters an existing database
DROP DATABASE	Drops an existing database
RENAME DATABASE	Renames an existing database

4.23 DatabasePrivileges Table

Structure

```
CREATE TABLE "DatabasePrivileges"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Privilege" VARCHAR(15) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "DatabasePrivileges"
("Name")

CREATE INDEX "GrantedTo" ON "DatabasePrivileges"
("GrantedTo")
```

Description

This table contains the database privileges assigned to the users in an ElevateDB configuration. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
GRANT PRIVILEGES	Grants privileges to an existing user
REVOKE PRIVILEGES	Revokes privileges from an existing user

4.24 Jobs Table

Structure

```
CREATE TABLE "Jobs"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Version" DECIMAL(19,2),
"Category" VARCHAR(15) COLLATE "ANSI_CI",
"UserName" VARCHAR(40) COLLATE "ANSI_CI",
"StartDate" DATE,
"EndDate" DATE,
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"Interval" INTEGER,
"Days" VARCHAR(60) COLLATE "ANSI_CI",
"DayNumber" INTEGER,
"DayOfMonth" VARCHAR(15) COLLATE "ANSI_CI",
"Months" VARCHAR(60) COLLATE "ANSI_CI",
"StartTime" TIME,
"EndTime" TIME,
"Definition" CLOB COLLATE "ANSI",
"Enabled" BOOLEAN,
"LastRun" TIMESTAMP,
"NextRun" TIMESTAMP,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Jobs"
("Name")
```

Description

This table contains the defined jobs in an ElevateDB configuration. Jobs are stored procedures that don't accept or output any parameters and are not contained within a specific database. Jobs are scheduled when they are defined and can only be executed by an ElevateDB server. In addition, each job can be assigned a category so that it is run on a specific ElevateDB server that is serving a particular category of jobs.

The Type column values are as follows:

Type	Description

Once	The job is scheduled to execute only once
Hourly	The job is scheduled to execute every hour
Daily	The job is scheduled to execute every day
Weekly	The job is scheduled to execute every week
Monthly	The job is scheduled to execute every month
Every X Minutes	The job is scheduled to execute every X number of minutes
Every X Hours	The job is scheduled to execute every X number of hours
Every X Days	The job is scheduled to execute every X number of days
Every X Weeks	The job is scheduled to execute every X number of weeks
At Server Start	The job is scheduled to execute when the ElevateDB Server is started

Note
The NextRun column will be NULL for any jobs that are disabled.

Related DDL Statements

Statement	Description
CREATE JOB	Creates a new job
ALTER JOB	Alters an existing job
DROP JOB	Drops an existing job
RENAME JOB	Renames an existing job
ENABLE JOB	Enables an existing job
DISABLE JOB	Disables an existing job
ENABLE JOBS	Enables all existing jobs
DISABLE JOBS	Disables all existing jobs

4.25 Stores Table

Structure

```
CREATE TABLE "Stores"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Path" VARCHAR(255) COLLATE "ANSI_CI",
"Address" VARCHAR(16) COLLATE "ANSI_CI",
"Host" VARCHAR(60) COLLATE "ANSI_CI",
"Port" INTEGER,
"Service" VARCHAR(60) COLLATE "ANSI_CI",
"UserName" VARCHAR(40) COLLATE "ANSI_CI",
"Password" VARCHAR(60) COLLATE "ANSI_CI",
"RemoteStore" VARCHAR(40) COLLATE "ANSI_CI",
"Signature" VARCHAR(60) COLLATE "ANSI_CI",
"Encrypted" BOOLEAN,
"EncryptionPassword" VARCHAR(60) COLLATE "ANSI_CI",
"Compression" INTEGER,
"Timeout" INTEGER,
"Ping" BOOLEAN,
"PingInterval" INTEGER,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Stores"
("Name")
```

Description

This table containes the defined stores in an ElevateDB configuration. Each store in ElevateDB is defined with path information that determines the folder where the store will manage its files.

The Type column values are as follows:

Type	Description
Local	The store references a local path
Remote	The store references another store defined on an ElevateDB Server

Related DDL Statements

Statement	Description

CREATE STORE	Creates a new store
ALTER STORE	Alters an existing store
DROP STORE	Drops an existing store
RENAME STORE	Renames an existing store

4.26 StorePrivileges Table

Structure

```
CREATE TABLE "StorePrivileges"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Privilege" VARCHAR(15) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "StorePrivileges"
("Name")

CREATE INDEX "GrantedTo" ON "StorePrivileges"
("GrantedTo")
```

Description

This table contains the store privileges assigned to the users in an ElevateDB configuration. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
GRANT PRIVILEGES	Grants privileges to an existing user
REVOKE PRIVILEGES	Revokes privileges from an existing user

4.27 Files Table

Structure

```
CREATE TABLE "Files"
(
"Name" VARCHAR(255) COLLATE "ANSI_CI",
"CreatedOn" TIMESTAMP,
"ModifiedOn" TIMESTAMP,
"Size" LARGEINT)

CREATE INDEX "Name" ON "Files"
("Name")

CREATE INDEX "CreatedOn" ON "Files"
("CreatedOn")

CREATE INDEX "ModifiedOn" ON "Files"
("ModifiedOn")
```

Description

The files are dynamic in ElevateDB and this table reflects the files present in the current files store. The files store can be changed or modified using the SET FILES STORE statement.

Related DDL Statements

Statement	Description
SET FILES STORE	Sets the current files store
COPY FILE	Copies a file from one file to another file, either in the same store or in a different store
RENAME FILE	Renames a file to a different name in a store
DELETE FILE	Deletes a file from a store

4.28 Information Schema

The tables that make up the Information schema for each database are as follows:

Tables Table
TablePrivileges Table
TableColumns Table
TemporaryTables Table
Constraints Table
ConstraintColumns Table
Indexes Table
IndexColumns Table
Triggers Table
TriggerColumns Table
Views Table
ViewPrivileges Table
ViewColumns Table
TemporaryViews Table
Procedures Table
ProcedurePrivileges Table
ProcedureParams Table
Functions Table
FunctionPrivileges Table
FunctionParams Table
Dependencies Table
SchemaObjects Table
SchemaDifference Table

The metadata that these tables are based upon is stored in the catalog file (EDBDatabase.EDBCat, by default) located in the database folder where the database was created. See the CREATE DATABASE statement for more information on creating a database.

SQL 2003 Standard Deviations

The following areas are where ElevateDB deviates from the SQL 2003 standard:

Deviation	Details
Extended Objects	Indexes are an ElevateDB extension, and these objects are not defined in the SQL 2003 standard.

4.29 Tables Table

Structure

```
CREATE TABLE "Tables"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI"
"Attributes" CLOB COLLATE "ANSI",
"Version" DECIMAL(19,2),
"ReadOnly" BOOLEAN,
"Encrypted" BOOLEAN,
"RowSize" INTEGER,
"IndexPageSize" INTEGER,
"BlobBlockSize" INTEGER,
"PublishBlockSize" INTEGER,
"PublishCompression" INTEGER,
"MaxRowBufferSize" INTEGER,
"MaxIndexBufferSize" INTEGER,
"MaxBlobBufferSize" INTEGER,
"MaxPublishBufferSize" INTEGER,
"Published" BOOLEAN,
"PublishedOn" TIMESTAMP,
"PublishID" VARCHAR(60) COLLATE "ANSI_CI",
"DefaultsEnabled" BOOLEAN,
"GeneratedEnabled" BOOLEAN,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Tables"
("Name")
```

Description

This table contains the defined tables in an ElevateDB database.

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
ALTER TABLE	Alters an existing table
DROP TABLE	Drops an existing table
RENAME TABLE	Renames an existing table

4.30 TablePrivileges Table

Structure

```
CREATE TABLE "TablePrivileges"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Privilege" VARCHAR(15) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "TablePrivileges"
("Name")

CREATE INDEX "GrantedTo" ON "TablePrivileges"
("GrantedTo")
```

Description

This table contains the table privileges assigned to the users in an ElevateDB configuration for the tables contained within an ElevateDB database. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
GRANT PRIVILEGES	Grants privileges to an existing user
REVOKE PRIVILEGES	Revokes privileges from an existing user

4.31 TableColumns Table

Structure

```
CREATE TABLE "TableColumns"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"Length" INTEGER,
"Precision" INTEGER,
"Scale" INTEGER,
"BlobCompression" INTEGER,
"Nullable" BOOLEAN,
"ErrorCode" INTEGER,
"ErrorMessage" CLOB COLLATE "ANSI",
"Generated" BOOLEAN,
"GeneratedWhen" VARCHAR(15) COLLATE "ANSI_CI",
"GenerateExpr" CLOB COLLATE "ANSI",
"Identity" BOOLEAN,
"IdentitySeed" INTEGER,
"IdentityIncrement" INTEGER,
"Computed" BOOLEAN,
"ComputeExpr" CLOB COLLATE "ANSI",
"DefaultExpr" CLOB COLLATE "ANSI",
"OrdinalPos" INTEGER,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "TableName" ON "TableColumns"
("TableName")

CREATE INDEX "Name" ON "TableColumns"
("Name")
```

Description

This table contains the defined columns for the tables in an ElevateDB database.

The GeneratedWhen column values are as follows:

GeneratedWhen	Description
Always	The generated column is always updated when a row is inserted or updated, overwriting any value that may exist for the column
By Default	The generated column is only updated when a row is inserted if no value currently exists for the column (NULL)

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
ALTER TABLE	Alters an existing table

4.32 TemporaryTables Table

Structure

```
CREATE TABLE "TemporaryTables"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Version" DECIMAL(19,2),
"ReadOnly" BOOLEAN,
"Encrypted" BOOLEAN,
"RowSize" INTEGER,
"IndexPageSize" INTEGER,
"BlobBlockSize" INTEGER,
"MaxRowBufferSize" INTEGER,
"MaxIndexBufferSize" INTEGER,
"MaxBlobBufferSize" INTEGER
)

CREATE INDEX "Name" ON "TemporaryTables"
("Name")
```

Description

This table contains the defined temporary tables in an ElevateDB database.

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new temporary table
ALTER TABLE	Alters an existing temporary table
DROP TABLE	Drops an existing temporary table
RENAME TABLE	Renames an existing temporary table

4.33 Constraints Table

Structure

```
CREATE TABLE "Constraints"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI"
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"EnforcingIndex" VARCHAR(40) COLLATE "ANSI_CI",
"TargetTable" VARCHAR(40) COLLATE "ANSI_CI",
"TargetTableConstraint" VARCHAR(40) COLLATE "ANSI_CI",
"UpdateAction" VARCHAR(15) COLLATE "ANSI_CI",
"DeleteAction" VARCHAR(15) COLLATE "ANSI_CI",
"CheckExpr" CLOB COLLATE "ANSI",
"ErrorCode" INTEGER,
"ErrorMessage" CLOB COLLATE "ANSI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "TableName" ON "Constraints"
("TableName")

CREATE INDEX "Name" ON "Constraints"
("Name")

CREATE INDEX "Type" ON "Constraints"
("Type")

CREATE INDEX "TargetTable" ON "Constraints"
("TargetTable")
```

Description

This table contains the defined constraints for the tables in an ElevateDB database.

The Type column values are as follows:

Type	Description
Primary Key	The constraint is a primary key constraint
Foreign Key	The constraint is a foreign key constraint
Unique	The constraint is a unique constraint
Check	The constraint is a check constraint

The UpdateAction and DeleteAction column values are as follows:

Action	Description

No Action	This is the same as Restrict
Cascade	Not supported currently
Set Null	Not supported currently
Set Default	Not supported currently
Restrict	If a primary key or unique constraint column is updated, then the update will be rejected if it violates any foreign key constraints

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
ALTER TABLE	Alters an existing table

4.34 ConstraintColumns Table

Structure

```
CREATE TABLE "ConstraintColumns"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"ConstraintName" VARCHAR(40) COLLATE "ANSI_CI",
"ColumnName" VARCHAR(40) COLLATE "ANSI_CI",
"OrdinalPos" INTEGER
)

CREATE INDEX "TableName" ON "ConstraintColumns"
("TableName")

CREATE INDEX "ConstraintName" ON "ConstraintColumns"
("ConstraintName")

CREATE INDEX "ColumnName" ON "ConstraintColumns"
("ColumnName")
```

Description

This table contains the columns that make up the defined primary, unique, or foreign key constraints for the tables in an ElevateDB database.

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
ALTER TABLE	Alters an existing table

4.35 Indexes Table

Structure

```
CREATE TABLE "Indexes"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"OwnerConstraint" VARCHAR(40) COLLATE "ANSI_CI",
"IndexedWordLength" INTEGER,
"FilterTypeColumn" VARCHAR(40) COLLATE "ANSI_CI",
"WordGenerator" VARCHAR(40) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "TableName" ON "Indexes"
("TableName")

CREATE INDEX "Name" ON "Indexes"
("Name")

CREATE INDEX "Type" ON "Indexes"
("Type")
```

Description

This table contains the defined indexes for the tables in an ElevateDB database.

The Type column values are as follows:

Type	Description
Index	The index is a normal index
Primary Key	The index is used to enforce a primary key constraint, and is maintained automatically by ElevateDB
Foreign Key	The index is used to enforce a foreign key constraint, and is maintained automatically by ElevateDB
Unique	The index is used to enforce a unique constraint, and is maintained automatically by ElevateDB
Text Index	The index is a text index

Related DDL Statements

Statement	Description

CREATE INDEX	Creates a new index
CREATE TEXT INDEX	Creates a new text index
ALTER INDEX	Alters an existing index
DROP INDEX	Drops an existing index
RENAME INDEX	Renames an existing index

4.36 IndexColumns Table

Structure

```
CREATE TABLE "IndexColumns"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"IndexName" VARCHAR(40) COLLATE "ANSI_CI",
"ColumnName" VARCHAR(40) COLLATE "ANSI_CI",
"Descending" BOOLEAN,
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"OrdinalPos" INTEGER
)

CREATE INDEX "TableName" ON "IndexColumns"
("TableName")

CREATE INDEX "IndexName" ON "IndexColumns"
("IndexName")

CREATE INDEX "ColumnName" ON "IndexColumns"
("ColumnName")
```

Description

This table contains the columns that make up the defined indexes for the tables in an ElevateDB database.

Related DDL Statements

Statement	Description
CREATE INDEX	Creates a new index
CREATE TEXT INDEX	Creates a new text index

4.37 Triggers Table

Structure

```
CREATE TABLE "Triggers"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"ActionTime" VARCHAR(15) COLLATE "ANSI_CI",
"ActionType" VARCHAR(15) COLLATE "ANSI_CI",
"Condition" CLOB COLLATE "ANSI",
"Definition" CLOB COLLATE "ANSI",
"Enabled" BOOLEAN,
"OrdinalPos" INTEGER,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "TableName" ON "Triggers"
("TableName")

CREATE INDEX "Name" ON "Triggers"
("Name")
```

Description

This table contains the defined triggers for the tables in an ElevateDB database.

The ActionTime column values are as follows:

ActionTime	Description
Before	The trigger will be executed before the type of operation for which the trigger is defined
After	The trigger will be executed after the type of operation for which the trigger is defined
Error	The trigger will be executed after any error with the type of operation for which the trigger is defined

The ActionType column values are as follows:

ActionType	Description
Insert	The trigger will be executed for any insert operation
Update	The trigger will be executed for any update operation
Delete	The trigger will be executed for any delete operation
All	The trigger will be executed for all operations (universal trigger)

Related DDL Statements

Statement	Description
CREATE TRIGGER	Creates a new trigger
ALTER TRIGGER	Alters an existing trigger
DROP TRIGGER	Drops an existing trigger
RENAME TRIGGER	Renames an existing trigger

4.38 TriggerColumns Table

Structure

```
CREATE TABLE "TriggerColumns"
(
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"TriggerName" VARCHAR(40) COLLATE "ANSI_CI",
"ColumnName" VARCHAR(40) COLLATE "ANSI_CI",
"OrdinalPos" INTEGER
)

CREATE INDEX "TableName" ON "TriggerColumns"
("TableName")

CREATE INDEX "TriggerName" ON "TriggerColumns"
("TriggerName")

CREATE INDEX "ColumnName" ON "TriggerColumns"
("ColumnName")
```

Description

This table contains the columns that make up the defined update triggers for the tables in an ElevateDB database. Update triggers can be created so that they fire only when specific columns are updated.

Related DDL Statements

Statement	Description
CREATE TRIGGER	Creates a new trigger

4.39 Views Table

Structure

```
CREATE TABLE "Views"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Version" DECIMAL(19,2),
"Definition" CLOB COLLATE "ANSI",
"Updateable" BOOLEAN,
"WithCheck" BOOLEAN,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Views"
("Name")
```

Description

This table contains the defined views in an ElevateDB database.

Related DDL Statements

Statement	Description
CREATE VIEW	Creates a new view
ALTER VIEW	Alters an existing view
DROP VIEW	Drops an existing view
RENAME VIEW	Renames an existing view

4.40 ViewPrivileges Table

Structure

```
CREATE TABLE "ViewPrivileges"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Privilege" VARCHAR(15) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "ViewPrivileges"
("Name")

CREATE INDEX "GrantedTo" ON "ViewPrivileges"
("GrantedTo")
```

Description

This table contains the view privileges assigned to the users in an ElevateDB configuration for the views contained within an ElevateDB database. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
GRANT PRIVILEGES	Grants privileges to an existing user
REVOKE PRIVILEGES	Revokes privileges from an existing user

4.41 ViewColumns Table

Structure

```
CREATE TABLE "ViewColumns"
(
"ViewName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Origin" VARCHAR(90) COLLATE "ANSI_CI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"Length" INTEGER,
"Precision" INTEGER,
"Scale" INTEGER,
"OrdinalPos" INTEGER,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "ViewName" ON "ViewColumns"
("ViewName")

CREATE INDEX "Name" ON "ViewColumns"
("Name")
```

Description

This table contains the defined columns for the views in an ElevateDB database.

Related DDL Statements

Statement	Description
CREATE VIEW	Creates a new view

4.42 ViewIndexes Table

Structure

```
CREATE TABLE "ViewIndexes"
(
"ViewName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "TableName" ON "Indexes"
("TableName")

CREATE INDEX "Name" ON "Indexes"
("Name")
```

Description

This table contains the defined indexes for the non-updateable views in an ElevateDB database.

The Type column values are as follows:

Type	Description
Index	The index is a normal index

Related DDL Statements

Statement	Description
CREATE INDEX	Creates a new index
ALTER INDEX	Alters an existing index
DROP INDEX	Drops an existing index
RENAME INDEX	Renames an existing index

4.43 TemporaryViews Table

Structure

```
CREATE TABLE "TemporaryViews"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Definition" CLOB COLLATE "ANSI",
"Updateable" BOOLEAN
)

CREATE INDEX "Name" ON "TemporaryViews"
("Name")
```

Description

This table contains the defined temporary views in an ElevateDB database. Temporary views are used to implement derived tables. The temporary view used to implement a derived table is dropped as soon as the SELECT statement containing the derived table is unprepared. Please see the SELECT statement for more information on derived tables.

Related DDL Statements

Statement	Description
SELECT	Creates a new temporary view via a derived table definition in the FROM clause

4.44 Procedures Table

Structure

```
CREATE TABLE "Procedures"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Version" DECIMAL(19,2),
"Implementation" VARCHAR(15) COLLATE "ANSI_CI",
"Definition" CLOB COLLATE "ANSI",
"ModuleName" VARCHAR(60) COLLATE "ANSI_CI",
"NumParams" INTEGER,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Procedures"
("Name")
```

Description

This table contains the defined procedures in an ElevateDB database.

The Implementation column values are as follows:

Implementation	Description
SQL	The procedure is implemented in SQL
External	The procedure is implemented in an external module

Related DDL Statements

Statement	Description
CREATE PROCEDURE	Creates a new procedure
ALTER PROCEDURE	Alters an existing procedure
DROP PROCEDURE	Drops an existing procedure
RENAME PROCEDURE	Renames an existing procedure

4.45 ProcedurePrivileges Table

Structure

```
CREATE TABLE "ProcedurePrivileges"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Privilege" VARCHAR(15) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "ProcedurePrivileges"
("Name")

CREATE INDEX "GrantedTo" ON "ProcedurePrivileges"
("GrantedTo")
```

Description

This table contains the procedure privileges assigned to the users in an ElevateDB configuration for the procedures contained within an ElevateDB database. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
GRANT PRIVILEGES	Grants privileges to an existing user
REVOKE PRIVILEGES	Revokes privileges from an existing user

4.46 ProcedureParams Table

Structure

```
CREATE TABLE "ProcedureParams"
(
"ProcedureName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Mode" VARCHAR(15) COLLATE "ANSI_CI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"Length" INTEGER,
"Precision" INTEGER,
"Scale" INTEGER,
"OrdinalPos" INTEGER
)

CREATE INDEX "ProcedureName" ON "ProcedureParams"
("ProcedureName")

CREATE INDEX "Name" ON "ProcedureParams"
("Name")
```

Description

This table contains the defined parameters for the procedures in an ElevateDB database.

The Mode column values are as follows:

Mode	Description
Unknown	The parameter type is unknown
In	The parameter is an input parameter
Out	The parameter is an output parameter
InOut	The parameter is both an input and output parameter

Related DDL Statements

Statement	Description
CREATE PROCEDURE	Creates a new procedure

4.47 Functions Table

Structure

```
CREATE TABLE "Functions"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Attributes" CLOB COLLATE "ANSI",
"Version" DECIMAL(19,2),
"Implementation" VARCHAR(15) COLLATE "ANSI_CI",
"Definition" CLOB COLLATE "ANSI",
"ModuleName" VARCHAR(60) COLLATE "ANSI_CI",
"NumParams" INTEGER,
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"Length" INTEGER,
"Precision" INTEGER,
"Scale" INTEGER,
"CreateSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "Functions"
("Name")
```

Description

This table contains the defined functions in an ElevateDB database.

The Implementation column values are as follows:

Implementation	Description
SQL	The function is implemented in SQL
External	The function is implemented in an external module

Related DDL Statements

Statement	Description
CREATE FUNCTION	Creates a new function
ALTER FUNCTION	Alters an existing function
DROP FUNCTION	Drops an existing function
RENAME FUNCTION	Renames an existing function

4.48 FunctionPrivileges Table

Structure

```
CREATE TABLE "FunctionPrivileges"
(
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Privilege" VARCHAR(15) COLLATE "ANSI_CI",
"GrantedTo" VARCHAR(40) COLLATE "ANSI_CI",
"GrantedBy" VARCHAR(40) COLLATE "ANSI_CI"
)

CREATE INDEX "Name" ON "FunctionPrivileges"
("Name")

CREATE INDEX "GrantedTo" ON "FunctionPrivileges"
("GrantedTo")
```

Description

This table contains the function privileges assigned to the users in an ElevateDB configuration for the functions contained within an ElevateDB database. You can find out more about users and privileges in the User Security topic.

Related DDL Statements

Statement	Description
GRANT PRIVILEGES	Grants privileges to an existing user
REVOKE PRIVILEGES	Revokes privileges from an existing user

4.49 FunctionParams Table

Structure

```
CREATE TABLE "FunctionParams"
(
"FunctionName" VARCHAR(40) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Description" CLOB COLLATE "ANSI",
"Mode" VARCHAR(15) COLLATE "ANSI_CI",
"Type" VARCHAR(30) COLLATE "ANSI_CI",
"Collation" VARCHAR(40) COLLATE "ANSI_CI",
"Length" INTEGER,
"Precision" INTEGER,
"Scale" INTEGER,
"OrdinalPos" INTEGER
)

CREATE INDEX "FunctionName" ON "FunctionParams"
("FunctionName")

CREATE INDEX "Name" ON "FunctionParams"
("Name")
```

Description

This table contains the defined parameters for the functions in an ElevateDB database.

The Mode column values are as follows:

Mode	Description
Unknown	The parameter type is unknown
In	The parameter is an input parameter
Out	The parameter is an output parameter
InOut	The parameter is both an input and output parameter

Related DDL Statements

Statement	Description
CREATE FUNCTION	Creates a new function

4.50 Dependencies Table

Structure

```
CREATE TABLE "Dependencies"
(
"ParentName" VARCHAR(40) COLLATE "ANSI_CI",
"ParentType" VARCHAR(15) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"DependentParentName" VARCHAR(40) COLLATE "ANSI_CI",
"DependentParentType" VARCHAR(15) COLLATE "ANSI_CI",
"DependentName" VARCHAR(40) COLLATE "ANSI_CI",
"DependentType" VARCHAR(15) COLLATE "ANSI_CI"
)

CREATE INDEX "ParentName" ON "Dependencies"
("ParentName")

CREATE INDEX "ParentType" ON "Dependencies"
("ParentType")

CREATE INDEX "Name" ON "Dependencies"
("Name")

CREATE INDEX "Type" ON "Dependencies"
("Type")

CREATE INDEX "DependentParentName" ON "Dependencies"
("DependentParentName")

CREATE INDEX "DependentParentType" ON "Dependencies"
("DependentParentType")

CREATE INDEX "DependentName" ON "Dependencies"
("DependentName")

CREATE INDEX "DependentType" ON "Dependencies"
("DependentType")
```

Description

This table contains the dependencies for all objects in an ElevateDB database.

The ParentType, Type, DependentParentType, and DependentType column values are as follows:

Type	Description

Schema	The object is a schema
Table	The object is a table
Column	The object is a column
Constraint	The object is a constraint
Trigger	The object is an trigger
Index	The object is an index
View	The object is a view
Procedure	The object is a procedure
Function	The object is a function

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
CREATE TRIGGER	Creates a new trigger
CREATE INDEX	Creates a new index
CREATE VIEW	Creates a new view
CREATE PROCEDURE	Creates a new procedure
CREATE FUNCTION	Creates a new function

4.51 SchemaObjects Table

Structure

```
CREATE TABLE "SchemaObjects"
(
"ParentName" VARCHAR(40) COLLATE "ANSI_CI",
"ParentType" VARCHAR(15) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"CreateSQL" CLOB COLLATE "ANSI_CI",
"PostCreateSQL" CLOB COLLATE "ANSI_CI",
"PreDropSQL" CLOB COLLATE "ANSI_CI",
"DropSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "ParentName" ON "SchemaObjects"
("ParentName")

CREATE INDEX "ParentType" ON "SchemaObjects"
("ParentType")

CREATE INDEX "Name" ON "SchemaObjects"
("Name")

CREATE INDEX "Type" ON "SchemaObjects"
("Type")
```

Description

This table contains all objects in an ElevateDB database in their dependency-sensitive creation order. This table can be used to generate a script for creating/dropping all objects in a database in a manner that will not generate any dependency errors.

In order to generate a proper CREATE script, simply navigate this table from the first row to the last row, using the CreateSQL column for the CREATE SQL statements for each object. After this initial pass, navigate this table again from the first row to the last row, using the PostCreateSQL column for any remaining ALTER SQL statements that may exist for dealing with objects that contain mutual depdendencies between each other, such as the case with two tables that have foreign keys that refer to the other table.

Note
If you wish to include populating tables with existing rows in your CREATE script, you should do so **before** including any PostCreateSQL statements that may exist.

In order to generate a proper DROP script, simply navigate this table in reverse order from the last row to the first row, using the PreDropSQL column for any ALTER SQL statements that may exist for dealing with objects that contain mutual dependencies between each other, and need to have these dependencies removed before the objects can be dropped. After this initial pass, navigate this table again in reverse order from the last row to the first row, using the DropSQL column for the DROP SQL statements for each

object.

> **Note**
> Sub-objects such as indexes and triggers for tables are not specified in this table. Because such objects are not dependency-sensitive, they can be created according to the order that they appear in in their corresponding system information table such as the Indexes and Triggers tables. Simply query these tables for a given parent table name to retrieve the list of sub-objects.

The ParentType and Type column values are as follows:

Type	Description
Table	The object is a table
View	The object is a view
Procedure	The object is a procedure
Function	The object is a function

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
CREATE TRIGGER	Creates a new trigger
CREATE INDEX	Creates a new index
CREATE VIEW	Creates a new view
CREATE PROCEDURE	Creates a new procedure
CREATE FUNCTION	Creates a new function
DROP TABLE	Drops an existing table
DROP TRIGGER	Drops an existing trigger
DROP INDEX	Drops an existing index
DROP VIEW	Drops an existing view
DROP PROCEDURE	Drops an existing procedure
DROP FUNCTION	Drops an existing function

4.52 SchemaDifference Table

Structure

```
CREATE TABLE "SchemaDifference"
(
"ParentName" VARCHAR(40) COLLATE "ANSI_CI",
"ParentType" VARCHAR(15) COLLATE "ANSI_CI",
"Name" VARCHAR(40) COLLATE "ANSI_CI",
"Type" VARCHAR(15) COLLATE "ANSI_CI",
"AlterSQL" CLOB COLLATE "ANSI_CI"
)

CREATE INDEX "ParentName" ON "SchemaDifference"
("ParentName")

CREATE INDEX "ParentType" ON "SchemaDifference"
("ParentType")

CREATE INDEX "Name" ON "SchemaDifference"
("Name")

CREATE INDEX "Type" ON "SchemaDifference"
("Type")
```

Description

This table is populated for the source database when the COMPARE DATABASE statement is executed. The contents of the table reflect the minimal difference between the source database and the target database, and are in the proper dependency order. This table can be used to generate a script for altering objects in a database so that they are equivalent to the target databaes, and in a manner that will not generate any dependency errors.

Note
This table is re-populated every time the COMPARE DATABASE statement is executed. If an object requires multiple statements for creating, altering, or dropping sub-objects (such as indexes/triggers for tables), then each statement will be separated by a blank line and terminated with the statement terminator character specified in the COMPARE DATABASE statement (or '!', if no statement terminator character is specified).

The ParentType and Type column values are as follows:

Type	Description
Table	The object is a table
View	The object is a view
Procedure	The object is a procedure
Function	The object is a function

Related DDL Statements

Statement	Description
CREATE TABLE	Creates a new table
CREATE TRIGGER	Creates a new trigger
CREATE INDEX	Creates a new index
CREATE VIEW	Creates a new view
CREATE PROCEDURE	Creates a new procedure
CREATE FUNCTION	Creates a new function
ALTER TABLE	Alters an existing table
ALTER TRIGGER	Alters an existing trigger
ALTER INDEX	Alters an existing index
ALTER VIEW	Alters an existing view
ALTER PROCEDURE	Alters an existing procedure
ALTER FUNCTION	Alters an existing function
DROP TABLE	Drops an existing table
DROP TRIGGER	Drops an existing trigger
DROP INDEX	Drops an existing index
DROP VIEW	Drops an existing view
DROP PROCEDURE	Drops an existing procedure
DROP FUNCTION	Drops an existing function

This page intentionally left blank

Chapter 5
DDL Statements

5.1 Introduction

DDL (data definition language) statements are used to create, alter, or drop objects in or from an ElevateDB configuration or database. This section of the manual details the available DDL statements in ElevateDB.

Notation

The notation used in the syntax section for each DDL statement is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

5.2 CREATE DATABASE

Creates a new database.

Syntax

```
CREATE DATABASE <Name>
PATH <Path>|IN MEMORY
[ENCRYPTED CATALOG|UNENCRYPTED CATALOG]
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]

<Path> = Any valid operating system path
```

Usage

Use this statement to create a new database. If the path specified for the database is not valid, then ElevateDB will attempt to create the path. Specifying the IN MEMORY keywords instead of a path will result in the database being created in memory.

Note
As of 2.03 Build 14, if you specify a relative path for an on-disk (not in-memory) database, the relative path will be interpreted as relative to the current configuration file path setting for the current session. For example, given a configuration file path of 'C:\MyApplication', the database path 'MyDatabase' will be interpreted as 'C:\MyApplication\MyDatabase'.

The ENCRYPTED CATALOG clause can be used to indicate that the catalog for the database should be encrypted on disk.

Note
Using the UNENCRYPTED CATALOG clause is the same as not specifying the ENCRYPTED CATALOG clause, and is present for compatibility with the CREATE DATABASE syntax.

Examples

```
-- The following statement creates a Support database using a
-- path name without a drive letter.

CREATE DATABASE "Support"
PATH '\support\data'
DESCRIPTION 'Support Database'

-- The following statement creates an Accounting database using a
-- path name with a drive letter.

CREATE DATABASE "Accounting"
```

```
PATH 'g:\acctng\data'
DESCRIPTION 'Accounting Database'

-- The following statement creates an Accounting database using a
-- relative path name.  The database will be located in the 'data'
-- subdirectory under the defined configuration path.

CREATE DATABASE "Accounting"
PATH 'data'
DESCRIPTION 'Accounting Database'

-- The following statement creates an Accounting database using a
-- relative path name.  The database will be located in the same
-- directory as the defined configuration path

CREATE DATABASE "Accounting"
PATH '.'
DESCRIPTION 'Accounting Database'

-- The following statement creates a Tracks database in memory

CREATE DATABASE "Tracks"
IN MEMORY
DESCRIPTION 'Song Tracks Database'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.3 ALTER DATABASE

Alters an existing database.

Syntax

```
ALTER DATABASE <Name>
PATH <Path>|IN MEMORY
[ENCRYPTED CATALOG|UNENCRYPTED CATALOG]
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]
```

Usage

Use this statement to alter an existing database. The options are the same as those for the CREATE DATABASE statement.

> **Note**
> All clauses after the PATH clause are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER DATABASE statement was executed.

Examples

```
-- The following statement changes the description of the Support database.

ALTER DATABASE "Support"
PATH '\support\data'
DESCRIPTION 'Support Database for All Applications'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.4 DROP DATABASE

Drops an existing database.

Syntax

```
DROP DATABASE <Name>
[KEEP CONTENTS]
```

Usage

Use this statement to drop a database.

Warning
Dropping a database will drop all tables and the entire catalog for the database specified. This means that all data and metadata for the database will be permanently deleted. However, you can keep the catalog and tables but remove the database from the configuration by specifying the KEEP CONTENTS clause.

Examples

```
-- The following statement drops the Support database.

DROP DATABASE "Support"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.5 RENAME DATABASE

Renames an existing database.

Syntax

```
RENAME DATABASE <Name> TO <Name>
```

Usage

Use this statement to rename a database.

Examples

```
-- The following statement renames the Support
-- database to MainSupport

RENAME DATABASE "Support" TO "MainSupport"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.6 CREATE USER

Creates a new user.

Syntax

```
CREATE USER <Name>
PASSWORD <Password>
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]
```

Usage

Use this statement to create a new user.

Warning
The password is sent over a network as plain text when this statement is used with a non-encrypted connection to the ElevateDB server.

Examples

```
-- The following statement creates a new user "Joe Smith".

CREATE USER "JoeSmith"
PASSWORD 'Test1043'
DESCRIPTION 'Joe Smith'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.7 ALTER USER

Alters an existing user.

Syntax

```
ALTER USER <Name>
PASSWORD <Password>
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]
```

Usage

Use this statement to alter an existing user.

Warning
The password is sent over a network as plain text when this statement is used with a non-encrypted connection to the ElevateDB server.

Note
All clauses after the PASSWORD clause are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER USER statement was executed.

Examples

```
-- The following statement changes the password of
-- the "Joe Smith" user.

ALTER USER "Joe Smith"
PASSWORD 'New1030'
```

Required Privileges

The current user must be granted the system-defined Administrators role or be logged in as the same user that is being altered in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Extension	This SQL statement is an ElevateDB extension

5.8 DROP USER

Drops an existing user.

Syntax

```
DROP USER <Name>
```

Usage

Use this statement to drop a user.

Examples

```
-- The following statement drops the "Joe Smith" user.

DROP USER "Joe Smith"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.9 RENAME USER

Renames an existing user.

Syntax

```
RENAME USER <Name> TO <Name>
```

Usage

Use this statement to rename a user.

Examples

```
-- The following statement renames the "Joe Smith"
-- user as "John Doe"

RENAME USER "Joe Smith" TO "John Doe"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.10 CREATE ROLE

Creates a new role.

Syntax

```
CREATE ROLE <Name>
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]
```

Usage

Use this statement to create a new role. Roles can be granted privileges and then granted to users. This makes it very quick and easy to modify the privileges for a group of users without being force to modify the privileges for each user individually.

Examples

```
-- The following statement creates a Cashier role.

CREATE ROLE "Cashier"
DESCRIPTION 'Cashier Role'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.
WITH ADMIN	The WITH ADMIN clause is not supported.

5.11 ALTER ROLE

Alters an existing role.

Syntax

```
ALTER ROLE <Name>
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]
```

Use this statement to alter an existing role.

Note
All clauses are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER ROLE statement was executed.

```
-- The following statement changes the description of the Cashier role.

ALTER ROLE "Cashier"
DESCRIPTION 'Cashier role'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.12 DROP ROLE

Drops an existing role.

Syntax

```
DROP ROLE <Name>
```

Usage

Use this statement to drop a role.

Warning
Dropping a role which has been assigned to existing users may mean that the users will cease to have the proper privileges required to complete their necessary tasks.

Examples

```
-- The following statement drops the Cashier role.

DROP ROLE "Cashier"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

5.13 RENAME ROLE

Renames an existing role.

Syntax

```
RENAME ROLE <Name> TO <Name>
```

Usage

Use this statement to rename a role.

Examples

```
-- The following statement renames the Cashier
-- role to the Clerk role

RENAME ROLE "Cashier" TO "Clerk"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.14 GRANT PRIVILEGES

Grants privileges on a given object to a user or role.

Syntax

```
GRANT <PrivilegeName> [,<PrivilegeName>]
ON <ObjectName>
TO <Authorization> [,<Authorization>]

<PrivilegeName> =

ALL PRIVILEGES|
SELECT|
INSERT|
UPDATE|
DELETE|
CREATE|
ALTER|
DROP|
MAINTAIN|
BACKUP|
RESTORE|
EXECUTE

<ObjectName> =

DATABASE <DatabaseName>|
STORE <StoreName>|
TABLE <TableName>|
VIEW <ViewName>|
PROCEDURE <ProcedureName>|
FUNCTION <FunctionName>

<Authorization> = <UserName>|<RoleName>
```

Usage

Use this statement to grant privileges to a user or role.

Examples

```
-- The following statement grants SELECT privileges
-- on the EmployeesList view to the system-defined
-- Public role which, by default, includes all users.

GRANT SELECT ON VIEW "EmployeesList"
TO "Public"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
WITH HIERARCHY OPTION	ElevateDB does not support the WITH HIERARCHY OPTION clause.
WITH GRANT OPTION	ElevateDB does not support the WITH GRANT OPTION clause. Only administrators can grant privileges and roles in ElevateDB.
GRANTED BY	ElevateDB does not support the GRANTED BY clause. The grantor in ElevateDB is always the administrator executing the GRANT statement.
Privileges	ElevateDB does not support the REFERENCES, USAGE, TRIGGER, and UNDER privileges, and adds the CREATE, ALTER, DROP, MAINTAIN, BACKUP, and RESTORE privileges as extensions.

5.15 REVOKE PRIVILEGES

Revokes privileges on a given object from a user or role.

Syntax

```
REVOKE <PrivilegeName> [,<PrivilegeName>]
ON <ObjectName>
FROM <Authorization> [,<Authorization>]

<PrivilegeName> =

ALL PRIVILEGES|
SELECT|
INSERT|
UPDATE|
DELETE|
CREATE|
ALTER|
DROP|
MAINTAIN|
BACKUP|
RESTORE|
EXECUTE

<ObjectName> =

DATABASE <DatabaseName>|
STORE <StoreName>|
TABLE <TableName>|
VIEW <ViewName>|
PROCEDURE <ProcedureName>|
FUNCTION <FunctionName>

<Authorization> = <UserName>|<RoleName>
```

Usage

Use this statement to revoke privileges from a user or role.

Examples

```
-- The following statement revokes SELECT privileges
-- on the EmployeesList view from the system-defined
-- Public role which, by default, includes all users.

REVOKE SELECT ON VIEW "EmployeesList"
FROM "Public"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
HIERARCHY OPTION FOR	ElevateDB does not support the HIERARCHY OPTION FOR clause.
GRANT OPTION FOR	ElevateDB does not support the GRANT OPTION FOR clause. Only administrators can revoke privileges and roles in ElevateDB.
GRANTED BY	ElevateDB does not support the GRANTED BY clause. The grantor in ElevateDB is always the administrator executing the GRANT statement.
RESTRICT or CASCADE	ElevateDB does not support the RESTRICT or CASCADE clauses.
Privileges	ElevateDB does not support the REFERENCES, USAGE, TRIGGER, and UNDER privileges, and adds the CREATE, ALTER, DROP, MAINTAIN, BACKUP, and RESTORE privileges as extensions.

5.16 GRANT ROLES

Grants roles to a given user.

Syntax

```
GRANT <RoleName> [,<RoleName>]
TO <Authorization> [,<Authorization>]

<Authorization> = <UserName>
```

Usage

Use this statement to grant a role to a user.

Examples

```
-- The following statement grants the Administrators
-- role to the user "Joe Smith".

GRANT "Administrators" TO "Joe Smith"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
WITH ADMIN OPTION	ElevateDB does not support the WITH ADMIN OPTION clause. Only administrators can grant privileges and roles in ElevateDB.
GRANTED BY	ElevateDB does not support the GRANTED BY clause. The grantor in ElevateDB is always the administrator executing the GRANT statement.

5.17 REVOKE ROLES

Revokes roles from a given user.

Syntax

```
REVOKE <RoleName> [,<RoleName>]
FROM <Authorization> [,<Authorization>]

<Authorization> = <UserName>
```

Usage

Use this statement to revoke a role from a user.

Examples

```
-- The following statement revokes the Administrators
-- role from the user "Joe Smith".

REVOKE "Administrators" FROM "Joe Smith"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
ADMIN OPTION FOR	ElevateDB does not support the ADMIN OPTION FOR clause. Only administrators can revoke privileges and roles in ElevateDB.
GRANTED BY	ElevateDB does not support the GRANTED BY clause. The grantor in ElevateDB is always the administrator executing the GRANT statement.
RESTRICT or CASCADE	ElevateDB does not support the RESTRICT or CASCADE clauses.

5.18 CREATE JOB

Creates a new job.

Syntax

```
CREATE JOB <Name>
RUN AS <UserName>
FROM <StartDate> TO <EndDate>
<IntervalDefinition>
[CATEGORY <CategoryName>]
<BodyDefinition>
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION
   [<Statement>;]]
END

<IntervalDefinition> =

<IntervalType>|<SpecificInterval>

<IntervalType> =

ONCE|HOURLY|DAILY|WEEKLY|MONTHLY|AT SERVER START
[ON <DaysDefinition>|ON <DaysDefinition> OF <MonthsDefinition>
BETWEEN <StartTime> AND <EndTime>]

<SpecificInterval> =

EVERY <Interval> MINUTES|HOURS|DAYS|WEEKS
ON <DaysDefinition>
BETWEEN <StartTime> AND <EndTime>

HOURLY/DAILY/WEEKLY/EVERY <Interval> MINUTES/HOURS/WEEKS Interval

<DaysDefinition> =

[MON] [,TUE] [,WED]....

MONTHLY Interval

<DaysDefinition> =

DAY <DayNumber>|
FIRST|SECOND|THIRD|FOURTH|LAST MON|TUE|WED|THU|FRI|SAT|SUN
```

```
<DayNumber> = 1..31

<MonthsDefinition> =

[JAN] [,FEB] [,MAR] [,APR]...
```

Usage

Use this statement to create a new job. Jobs are configuration-level procedures that are executed, by default, within the context of the system-defined Configuration database as the current database. Jobs accept no parameters and cannot return cursors or values.

Examples

```
-- The following job backs up all tables in all databases
-- defined in the current system at 11:00 PM every evening.

CREATE JOB Backup
RUN AS "System"
FROM DATE '2006-01-01' TO DATE '2010-12-31'
DAILY
BETWEEN TIME '11:00 PM' AND TIME '11:30 PM'
CATEGORY 'Backup'
BEGIN
   DECLARE DBCursor CURSOR FOR DBStmt;
   DECLARE DBName VARCHAR DEFAULT '';

   PREPARE DBStmt FROM 'SELECT * FROM Databases';

   OPEN DBCursor;

   FETCH FIRST FROM DBCursor ('Name') INTO DBName;

   WHILE NOT EOF(DBCursor) DO
      IF (DBName <> 'Configuration') THEN
         EXECUTE IMMEDIATE 'BACKUP DATABASE "' + DBName + '" AS "' +
                           CAST(CURRENT_DATE AS VARCHAR(10)) +
                           '-' + DBName + '" TO STORE "Backups" INCLUDE
      CATALOG';
      END IF;
      FETCH NEXT FROM DBCursor ('Name') INTO DBName;
   END WHILE;

   CLOSE DBCursor;
END
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.19 ALTER JOB

Alters an existing job.

Syntax

```
ALTER JOB <Name>
RUN AS <UserName>
FROM <StartDate> TO <EndDate>
<IntervalDefinition>
[CATEGORY <CategoryName>]
<BodyDefinition>
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION
   [<Statement>;]]
END

<IntervalDefinition> =

<IntervalType>|<SpecificInterval>

<IntervalType> =

ONCE|HOURLY|DAILY|WEEKLY|MONTHLY|AT SERVER START
[ON <DaysDefinition>|ON <DaysDefinition> OF <MonthsDefinition>
BETWEEN <StartTime> AND <EndTime>]

<SpecificInterval> =

EVERY <Interval> MINUTES|HOURS|DAYS|WEEKS
ON <DaysDefinition>
BETWEEN <StartTime> AND <EndTime>

HOURLY/DAILY/WEEKLY/EVERY <Interval> MINUTES/HOURS/WEEKS Interval

<DaysDefinition> =

[MON] [,TUE] [,WED]....

MONTHLY Interval

<DaysDefinition> =

DAY <DayNumber>|
FIRST|SECOND|THIRD|FOURTH|LAST MON|TUE|WED|THU|FRI|SAT|SUN
```

```
<DayNumber> = 1..31

<MonthsDefinition> =

[JAN] [,FEB] [,MAR] [,APR]...
```

Usage

Use this statement to alter an existing job.

> **Note**
> All clauses after the job definition are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER JOB statement was executed.

Examples

```
-- The following statement changes the run time
-- of the Backup job.

ALTER JOB Backup
RUN AS "System"
FROM DATE '2006-01-01' TO DATE '2010-12-31'
DAILY
BETWEEN TIME '12:00 AM' AND TIME '12:30 AM'
CATEGORY 'Backup'
BEGIN
   DECLARE DBCursor CURSOR FOR DBStmt;
   DECLARE DBName VARCHAR DEFAULT '';

   PREPARE DBStmt FROM 'SELECT * FROM Databases';

   OPEN DBCursor;

   FETCH FIRST FROM DBCursor ('Name') INTO DBName;

   WHILE NOT EOF(DBCursor) DO
      IF (DBName <> 'Configuration') THEN
         EXECUTE IMMEDIATE 'BACKUP DATABASE "' + DBName + '" AS "' +
                           CAST(CURRENT_DATE AS VARCHAR(10)) +
                           '-' + DBName + '" TO STORE "Backups" INCLUDE
      CATALOG';
      END IF;
      FETCH NEXT FROM DBCursor ('Name') INTO DBName;
   END WHILE;

   CLOSE DBCursor;
END
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this

statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.20 DROP JOB

Drops an existing job.

Syntax

```
DROP JOB <Name>
```

Usage

Use this statement to drop a job.

Examples

```
-- The following statement drops the Backup job.

DROP JOB "Backup"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.21 RENAME JOB

Renames an existing job.

Syntax

```
RENAME JOB <Name> TO <Name>
```

Usage

Use this statement to rename a job.

Examples

```
-- The following statement renames the Backup
-- job to BackupDB

RENAME JOB "Backup" TO "BackupDB"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.22 ENABLE JOB

Enables an existing job.

Syntax

```
ENABLE JOB <Name>
```

Usage

Use this statement to enable a job (if disabled).

Note
This is an in-memory operation and does not persist in the ElevateDB Server configuration. Once the ElevateDB Server is restarted, all jobs will be enabled, by default.

Examples

```
-- The following statement enables the Backup job

ENABLE JOB "Backup"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.23 DISABLE JOB

Disables an existing job.

Syntax

```
DISABLE JOB <Name>
```

Usage

Use this statement to disable a job (if enabled).

> **Note**
> This is an in-memory operation and does not persist in the ElevateDB Server configuration. Once the ElevateDB Server is restarted, all jobs will be enabled, by default.

Examples

```
-- The following statement disables the Backup job

DISABLE JOB "Backup"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.24 ENABLE JOBS

Enables all existing jobs.

Syntax

```
ENABLE JOBS
```

Usage

Use this statement to enable all jobs. If any of the jobs are already enabled, then this statement does nothing for those jobs.

> **Note**
> This is an in-memory operation and does not persist in the ElevateDB Server configuration. Once the ElevateDB Server is restarted, all jobs will be enabled, by default.

Examples

```
-- The following statement enables all jobs

ENABLE JOBS
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.25 DISABLE JOBS

Disables all existing jobs.

Syntax

```
DISABLE JOBS
```

Usage

Use this statement to disable all jobs. If any of the jobs are already disabled, then this statement does nothing for those jobs.

> **Note**
> This is an in-memory operation and does not persist in the ElevateDB Server configuration. Once the ElevateDB Server is restarted, all jobs will be enabled, by default.

Examples

```
-- The following statement disables all jobs

DISABLE JOBS
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.26 RESET JOB

Resets an existing job.

Syntax

```
RESET JOB <Name>
```

Usage

Use this statement to reset a job. When a job is reset, the last run timestamp for the job is cleared. Subsequently, ElevateDB treats the job as never having been executed for scheduling purposes.

Note
This is a persistent change that is saved in the current ElevateDB Server configuration file.

Examples

```
-- The following statement resets the Backup job

RESET JOB "Backup"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.27 CREATE STORE

Creates a new store.

Syntax

```
CREATE STORE <StoreName> AS <LocalStoreDefinition>|<RemoteStoreDefinition>
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]

<LocalStoreDefinition> =

LOCAL PATH <Path>

<Path> = Any valid operating system path

<RemoteStoreDefinition> =

REMOTE ADDRESS <IPAddress>|HOST <Host>
PORT <Port>|SERVICE <Service>
USER <UserName>
PASSWORD <Password>
STORE <RemoteStoreName>
[SIGNATURE <Signature>]
[ENCRYPTED]
[ENCRYPTION PASSWORD <EncryptionPassword>]
[COMPRESSION <Compression>]
[TIMEOUT <Timeout (seconds)>]
[PING <PingInterval (seconds)>]

<Compression> = 0..9
```

Usage

Use this statement to create a new store. Use the LOCAL keyword to create a local store that references a path accessible from the local process. If the path specified for a local store is not valid, then ElevateDB will attempt to create the path. Use the REMOTE keyword to create a remote store that references a store located on an ElevateDB Server specified by the ADDRESS or HOST, and PORT or SERVICE, keywords. The remote store name is not validated until the store is opened. If the remote store does not exist when the store is opened, then an appropriate error message will be displayed.

Note
As of 2.03 Build 14, if you specify a relative path for a store, the relative path will be interpreted as relative to the current configuration file path setting for the current session. For example, given a configuration file path of 'C:\MyApplication', the store path 'MyStore' will be interpreted as 'C:\MyApplication\MyStore'.

Examples

```
-- The following statement creates a local Backups
-- store using a path name without a drive letter.

CREATE STORE "Backups" AS
LOCAL PATH '\support\backups'
DESCRIPTION 'Support Database Backups'

-- The following statement creates a remote RemoteOffice store
-- that references the SavedUpdates store on a remote ElevateDB
-- Server.

CREATE STORE "RemoteOffice" AS
REMOTE ADDRESS '64.65.248.118' PORT 12010
USER "MainOffice"
PASSWORD 'LogMeIn'
STORE "SavedUpdates"
DESCRIPTION 'Remote Office Updates'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.28 ALTER STORE

Alters an existing store.

Syntax

```
ALTER STORE <StoreName> AS <LocalStoreDefinition>|<RemoteStoreDefinition>
[DESCRIPTION <Description>]
[ATTRIBUTES <CustomAttributes>]

<LocalStoreDefinition> =

LOCAL PATH <Path>

<Path> = Any valid operating system path

<RemoteStoreDefinition> =

REMOTE ADDRESS <IPAddress>|HOST <Host>
PORT <Port>|SERVICE <Service>
USER <UserName>
PASSWORD <Password>
STORE <RemoteStoreName>
[SIGNATURE <Signature>]
[ENCRYPTED]
[ENCRYPTION PASSWORD <EncryptionPassword>]
[COMPRESSION <Compression>]
[TIMEOUT <Timeout (seconds)>]
[PING <PingInterval (seconds)>]

<Compression> = 0..9
```

Usage

Use this statement to alter an existing store.

> **Note**
> All clauses after the local or remote store definition are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER STORE statement was executed.

Examples

```
-- The following statement alters a local Backups
-- store to use a new path.

ALTER STORE "Backups" AS
LOCAL PATH 'c:\backups'
DESCRIPTION 'Support Database Backups'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.29 DROP STORE

Drops an existing store.

Syntax

```
DROP STORE <Name>
[KEEP CONTENTS]
```

Usage

Use this statement to drop a store.

Warning
Dropping a store will delete all of the files present in the store along with the store directory if the store is a local store. However, you can keep the files but remove the store from the configuration by specifying the KEEP CONTENTS clause.

Examples

```
-- The following statement drops the Backups store.

DROP STORE "Backups"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.30 RENAME STORE

Renames an existing store.

Syntax

```
RENAME STORE <Name> TO <Name>
```

Usage

Use this statement to rename a store.

Examples

```
-- The following statement renames the Backups
-- store to DBBackups

RENAME STORE "Backups" TO "DBBackups"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.31 CREATE MODULE

Creates (registers) a new external module.

Syntax

```
CREATE MODULE <Name>
PATH <ExternalModuleFile>
[DESCRIPTION <Description>]

<ExternalModuleFile> = Path/file name of DLL
```

Usage

Use this statement to create (register) a new external module. An external module is a compiled DLL that contains specific code that can be used as a text filter, word generator, migrator, or external procedure/function. Please see the External Modules topic for more information.

Examples

```
-- The following statement registers a
-- text filter external module

CREATE MODULE "TextFilterModule"
PATH 'c:\myapplication\modules\txtfilter.dll'
DESCRIPTION 'Text Filter Module'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.32 ALTER MODULE

Alters an existing external module.

Syntax

```
ALTER MODULE <Name>
PATH <ExternalModuleFile>
[DESCRIPTION <Description>]

<ExternalModuleFile> = Path/file name of DLL
```

Usage

Use this statement to alter an existing external module.

Note
All clauses after the PATH clause are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER MODULE statement was executed.

Examples

```
-- The following statement changes the path
-- of the TextFilterModule module

ALTER MODULE "TextFilterModule"
PATH 'c:\myapplication\dlls\txtfilter.dll'
DESCRIPTION 'Text Filter Module'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.33 DROP MODULE

Drops an existing external module.

Syntax

```
DROP MODULE <Name>
```

Usage

Use this statement to drop an external module.

Examples

```
-- The following statement drops the TextFilterModule
-- external module.

DROP MODULE "TextFilterModule"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.34 RENAME MODULE

Renames an existing external module.

Syntax

```
RENAME MODULE <Name> TO <Name>
```

Usage

Use this statement to rename an external module.

Examples

```
-- The following statement renames the
-- TextFilterModule external module to
-- TextFilterDLL

RENAME MODULE "TextFilterModule" TO "TextFilterDLL"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.35 CREATE TEXT FILTER

Creates a new text filter.

Syntax

```
CREATE TEXT FILTER <Name>
TYPE <FilterType>
MODULE <ExternalModuleName>
[DESCRIPTION <Description>]
```

Usage

Use this statement to create a new text filter. A text filter associates a particular filter name with a text filter module that will be used to filter the text prior to the text being indexed by ElevateDB and added to a text index. The referenced external text filter module must already be created and available to the current session. Please see the Text Indexing topic for more information.

Examples

```
-- The following statement creates a text filter for HTML

CREATE TEXT FILTER "HTML"
TYPE 'HTML'
MODULE "HTMLFilter"
DESCRIPTION 'Filters HTML'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.36 ALTER TEXT FILTER

Alters an existing text filter.

Syntax

```
ALTER TEXT FILTER <Name>
[DESCRIPTION <Description>]
```

Usage

Use this statement to alter an existing text filter.

Note
All clauses are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER TEXT FILTER statement was executed.

Examples

```
-- The following statement changes the description of the HTML
-- text filter.

ALTER TEXT FILTER "HTML"
DESCRIPTION 'HTML Filter'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.37 DROP TEXT FILTER

Drops an existing text filter.

Syntax

```
DROP TEXT FILTER <Name>
```

Usage

Use this statement to drop a text filter.

Warning
Dropping a text filter that is in use may cause large amounts of text to be included in the text index. For example, if you included a CLOB column containing HTML in a text index that relied on an HTML text filter to remove all formatting prior to indexing, then removing the HTML text filter will cause the formatting to be subsequently included in the text index whenever the CLOB column is updated.

Examples

```
-- The following statement drops the HTML text filter.

DROP TEXT FILTER "HTML"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.38 RENAME TEXT FILTER

Renames an existing text filter.

Syntax

```
RENAME TEXT FILTER <Name> TO <Name>
```

Usage

Use this statement to rename a text filter.

Examples

```
-- The following statement renames the HTML
-- text filter to XML

RENAME TEXT FILTER "HTML" TO "XML"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.39 CREATE WORD GENERATOR

Creates a new word generator.

Syntax

```
CREATE WORD GENERATOR <Name>
MODULE <ExternalModuleName>
[DESCRIPTION <Description>]
```

Use this statement to create a new word generator. A word generator is used by ElevateDB to parse and extract the words from a column that are indexed via a text index on that column. The referenced external word generator module must already be created and available to the current session. Please see the Text Indexing topic for more information.

Examples

```
-- The following statement creates a word generator for
-- use with Pascal code.

CREATE WORD GENERATOR "Pascal"
MODULE "PascalWordGenerator"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.40 ALTER WORD GENERATOR

Alters an existing word generator.

Syntax

```
ALTER WORD GENERATOR <Name>
[DESCRIPTION <Description>]
```

Usage

Use this statement to alter an existing word generator.

Note
All clauses are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER WORD GENERATOR statement was executed.

Examples

```
-- The following statement changes the description of the Pascal
-- word generator.

ALTER WORD GENERATOR "Pascal"
DESCRIPTION 'Pascal language word generator'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.41 DROP WORD GENERATOR

Drops an existing word generator.

Syntax

```
DROP WORD GENERATOR <Name>
```

Usage

Use this statement to drop a word generator.

Warning
Dropping a word generator that is in use may cause text to be included in a text index that is not desired. For example, if you included a CLOB column containing German text in a text index that relied on a German word generator to properly parse all words prior to indexing, then removing the German word generator will cause different words to be subsequently included in the text index whenever the CLOB column is updated.

Examples

```
-- The following statement drops the Pascal word generator.

DROP WORD GENERATOR "Pascal"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.42 RENAME WORD GENERATOR

Renames an existing word generator.

Syntax

```
RENAME WORD GENERATOR <Name> TO <Name>
```

Usage

Use this statement to rename a word generator.

Examples

```
-- The following statement renames the Pascal
-- word generator to ObjectPascal

RENAME WORD GENERATOR "Pascal" TO "ObjectPascal"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.43 CREATE MIGRATOR

Creates a new migrator.

Syntax

```
CREATE MIGRATOR <Name>
MODULE <ExternalModuleName>
[DESCRIPTION <Description>]
```

Use this statement to create a new migrator. A migrator is used by ElevateDB to migrate a database from an external source to an ElevateDB database. The referenced external migrator module must already be created and available to the current session. Please see the Migrating Databases topic for more information.

Examples

```
-- The following statement creates a migrator for
-- use with the Borland Database Engine (BDE).

CREATE MIGRATOR "BDE"
MODULE "edbmigratebde"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.44 ALTER MIGRATOR

Alters an existing migrator.

Syntax

```
ALTER MIGRATOR <Name>
[DESCRIPTION <Description>]
```

Usage

Use this statement to alter the an existing migrator.

> **Note**
> All clauses are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER MIGRATOR statement was executed.

Examples

```
-- The following statement changes the description of the BDE
-- migrator.

ALTER MIGRATOR "BDE"
DESCRIPTION 'Borland Database Engine (BDE)'
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.45 DROP MIGRATOR

Drops an existing migrator.

Syntax

```
DROP MIGRATOR <Name>
```

Usage

Use this statement to drop a migrator.

Examples

```
-- The following statement drops the BDE migrator.

DROP MIGRATOR "BDE"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.46 RENAME MIGRATOR

Renames an existing migrator.

Syntax

```
RENAME MIGRATOR <Name> TO <Name>
```

Usage

Use this statement to rename a migrator.

Examples

```
-- The following statement renames the BDE
-- migrator to Borland Database Engine.

RENAME MIGRATOR "BDE" TO "Borland Database Engine"
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.47 CREATE TABLE

Creates a new table.

Syntax

```
CREATE [TEMPORARY] TABLE <Name>
[[(
<ColumnName> <ColumnDefinition>|
LIKE <LikeDefinition>|
[CONSTRAINT <ConstraintName>] <ConstraintDefinition>]
[,<ColumnName> <ColumnDefinition>|
LIKE <LikeDefinition>|
[CONSTRAINT <ConstraintName>] <ConstraintDefinition>]
)]
[AS <QueryExpression> WITH DATA|WITH NO DATA]
[FROM PUBLISHED UPDATES [TABLES <TableName> [,<TableName>]]] | [FROM UPDATES
      <UpdateName> IN STORE <StoreName>]]
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[READONLY|READWRITE]
[ENCRYPTED|UNENCRYPTED]
[INDEX PAGE SIZE <IndexPageSize>]
[BLOB BLOCK SIZE <BLOBBlockSize>]
[PUBLISH BLOCK SIZE <PublishBlockSize>]
[PUBLISH COMPREESSION <Compression>]
[MAX ROW BUFFER SIZE <MaxRowBufferSize>]
[MAX INDEX BUFFER SIZE <MaxIndexBufferSize>]
[MAX BLOB BUFFER SIZE <MaxBLOBBufferSize>]
[MAX PUBLISH BUFFER SIZE <MaxPublishBufferSize>]
[ATTRIBUTES <CustomAttributes>]

<ColumnDefinition> =

<DataType>
[COMPRESSION <Compression>]
[GENERATED <GenerationOptions>|COMPUTED <ComputationOptions>|
DEFAULT <DefaultExpression>]
[NOT NULL [ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]]
[<ColumnConstraintDefinition>] [<ColumnConstraintDefinition>]
[DESCRIPTION <Description>]

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code

<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
```

```
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<Compression> = 0..9

<GenerationOptions> =
ALWAYS AS <GenerateExpression>|
ALWAYS AS IDENTITY (START WITH <SeedValue>, INCREMENT BY <IncrementValue>)|
BY DEFAULT AS IDENTITY (START WITH <SeedValue>, INCREMENT BY
      <IncrementValue>)

<GenerateExpression> =
Any valid SQL expression that does not include any sub-queries

<ComputationOptions> =
ALWAYS AS <ComputeExpression>

<ComputeExpression> =
Any valid SQL expression that does not include any sub-queries
or user-defined function references

<DefaultExpression> =
Any valid SQL expression that does not include any sub-queries

<ColumnConstraintDefinition> =

CHECK <CheckExpression>
PRIMARY KEY|
UNIQUE|
REFERENCES <TableName> [(<ColumnName> [,<ColumnName>])]
   [ON UPDATE RESTRICT|ON DELETE RESTRICT]
[ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]

<CheckExpression> =
Any valid SQL expression that does not include any sub-queries

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code

<LikeDefinition> =

<TableName>
[INCLUDING|EXCLUDING IDENTITY]
[INCLUDING|EXCLUDING DEFAULTS]
[INCLUDING|EXCLUDING GENERATED]

<ConstraintDefinition> =

CHECK <CheckExpression>
PRIMARY KEY (<ColumnName> [,<ColumnName>])|
```

```
UNIQUE (<ColumnName> [,<ColumnName>])|
FOREIGN KEY (<ColumnName> [,<ColumnName>])
   REFERENCES <TableName> [(<ColumnName> [,<ColumnName>])]
   [ON UPDATE RESTRICT|ON DELETE RESTRICT]
[ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]
[DESCRIPTION <Description>]

<CheckExpression> =
Any valid SQL expression that does not include any sub-queries

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code
```

Usage

Use this statement to create a new table. Use the TEMPORARY clause to specify that the table should be created as a local temporary table that is only visible to the current session. You may only use the following DDL statements on temporary tables:

DROP TABLE
RENAME TABLE
EMPTY TABLE
CREATE INDEX
CREATE TEXT INDEX
DROP INDEX
RENAME INDEX

Use the READONLY clause to specify that a table is always read-only. Doing so can improve multi-user performance on a table because ElevateDB will not need to perform any locking on such a table.

FROM PUBLISHED UPDATES and FROM UPDATES Clauses

Use the FROM PUBLISHED UPDATES or FROM UPDATES version of the CREATE TABLE statement to create a table that contains the contents of pending published updates that have not been saved, or the contents of an existing replication update file in a store. These two clauses are mutually-exclusive, and only one can be used at a time.

The format of the created table will be as follows:

```
"TableName" VARCHAR(40) COLLATE "ANSI_CI",
"UpdateType" VARCHAR(15) COLLATE "ANSI_CI",
"UpdateTimeStamp" TIMESTAMP,
"Manifest" CLOB COLLATE "ANSI_CI",
"KeyData" CLOB COLLATE "ANSI_CI",
"RowData" CLOB COLLATE "ANSI_CI"
```

The UpdateType column will contain one of the following values:

UpdateType	Description

Insert	The update is an insert operation. The RowData column will contain a CRLF-delimited list of column:value pairs.
Update	The update is an update operation. The KeyData column will contain a CRLF-delimited list of primary key column:value pairs, and the RowData column will contain a CRLF-delimited list of column:value pairs.
Delete	The update is an update operation. The KeyData column will contain a CRLF-delimited list of primary key column:value pairs.

The Manifest column contains a CRLF-delimited list of published table IDs that serve to tell ElevateDB which published tables have loaded this update already.

With the PUBLISHED UPDATES clause, you can further limit the published tables that are used to generate the table by using the TABLES clause. When using the TABLES clause, only the pending published updates for the specified tables will be included in the created table.

For more information on replication, please see the Replication topic.

Examples

```
-- The following statement creates the Customer table

CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "ANSI_CI",
"Address1" VARCHAR(40) COLLATE "ANSI_CI",
"Address2" VARCHAR(40) COLLATE "ANSI_CI",
"City" VARCHAR(30) COLLATE "ANSI_CI",
"State" CHAR(2) COLLATE "ANSI_CI",
"Zip" CHAR(10) COLLATE "ANSI_CI",
"CreatedOn" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID"),
CONSTRAINT "ID_Check" CHECK (ID IS NOT NULL),
CONSTRAINT "Name_Check" CHECK (Name IS NOT NULL)
)

-- The following statement creates a temporary table containing
-- the contents of an update file

CREATE TEMPORARY TABLE "MainOfficeUpdates"
FROM UPDATES "MainOffice-2010-12-14 17-57-12.0306"
IN STORE MainUpdates
```

Required Privileges

The current user must be granted the CREATE privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Column COMPUTED	The column COMPUTED clause is an ElevateDB extension.
Column ERROR CODE/MESSAGE	The column ERROR CODE/MESSAGE clause is an ElevateDB extension.
Column DESCRIPTION	The column DESCRIPTION clause is an ElevateDB extension.
Column COMPRESSION	The column COMPRESSION clause is an ElevateDB extension.
Constraint ON UPDATE	The only option supported for the ON UPDATE clause is the RESTRICT option.
Constraint ON DELETE	The only option supported for the ON DELETE clause is the RESTRICT option.
Constraint ERROR CODE/MESSAGE	The constraint ERROR CODE/MESSAGE clause is an ElevateDB extension.
Constraint DESCRIPTION	The constraint DESCRIPTION clause is an ElevateDB extension.
DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ENCRYPTED	The ENCRYPTED and UNENCRYPTED clauses are an ElevateDB extension.
INDEX PAGE SIZE	The INDEX PAGE SIZE clause is an ElevateDB extension.
BLOB BLOCK SIZE	The BLOB BLOCK SIZE clause is an ElevateDB extension.
PUBLISH BLOCK SIZE	The PUBLISH BLOCK SIZE clause is an ElevateDB extension.
PUBLISH COMPRESSION	The PUBLISH COMPRESSION clause is an ElevateDB extension.
MAX ROW BUFFER SIZE	The MAX ROW BUFFER SIZE clause is an ElevateDB extension.
MAX INDEX BUFFER SIZE	The MAX INDEX BUFFER SIZE clause is an ElevateDB extension.
MAX BLOB BUFFER SIZE	The MAX BLOB BUFFER SIZE clause is an ElevateDB extension.
MAX PUBLISH BUFFER SIZE	The MAX PUBLISH BUFFER SIZE clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.
FROM PUBLISHED UPDATES	The FROM PUBLISHED UPDATES clause is an ElevateDB extension.
FROM UPDATES	The FROM UPDATES clause is an ElevateDB extension.

5.48 ALTER TABLE

Alters an existing table.

Syntax

```
ALTER TABLE <Name>
[ADD [COLUMN] <ColumnName> <ColumnDefinition>]
[ALTER [COLUMN] <ColumnName> <ColumnAlterOptions>|AS <ColumnDefinition>
[RENAME [COLUMN] <ColumnName> TO <ColumnName>]
[DROP [COLUMN] <ColumnName>]
[ADD [CONSTRAINT <ConstraintName>] <ConstraintDefinition>]
[ALTER CONSTRAINT <ConstraintName> <ConstraintAlterOptions>|AS
      <ConstraintDefinition>
[RENAME CONSTRAINT <ConstraintName> TO <ConstraintName>
[DROP CONSTRAINT <ConstraintName>]
[,ADD|ALTER|DROP]
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[READONLY|READWRITE]
[ENCRYPTED|UNENCRYPTED]
[INDEX PAGE SIZE <IndexPageSize>]
[BLOB BLOCK SIZE <BLOBBlockSize>]
[PUBLISH BLOCK SIZE <PublishBlockSize>]
[PUBLISH COMPREESSION <Compression>]
[MAX ROW BUFFER SIZE <MaxRowBufferSize>]
[MAX INDEX BUFFER SIZE <MaxIndexBufferSize>]
[MAX BLOB BUFFER SIZE <MaxBLOBBufferSize>]
[MAX PUBLISH BUFFER SIZE <MaxPublishBufferSize>]
[ATTRIBUTES <CustomAttributes>]
[NO BACKUP FILES]

<ColumnDefinition> =

<DataType>
[COMPRESSION <Compression>]
[GENERATED <GenerationOptions>|COMPUTED <ComputationOptions>|
DEFAULT <DefaultExpression>]
[<ColumnConstraintDefinition>] [<ColumnConstraintDefinition>]
[NOT NULL [ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]]
[<ColumnConstraintDefinition>] [<ColumnConstraintDefinition>]
[DESCRIPTION <Description>]
[AT <ColumnPos>]

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code

<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
```

```
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<Compression> = 0..9

<GenerationOptions> =
ALWAYS AS <GenerateExpression>|
ALWAYS AS IDENTITY (START WITH <SeedValue>, INCREMENT BY <IncrementValue>)|
BY DEFAULT AS IDENTITY (START WITH <SeedValue>, INCREMENT BY
      <IncrementValue>)

<GenerateExpression> =
Any valid SQL expression that does not include any sub-queries

<ComputationOptions> =
ALWAYS AS <ComputeExpression>

<ComputeExpression> =
Any valid SQL expression that does not include any sub-queries
or user-defined function references

<DefaultExpression> =
Any valid SQL expression that does not include any sub-queries

<ColumnConstraintDefinition> =

CHECK <CheckExpression>
PRIMARY KEY|
UNIQUE|
REFERENCES <TableName> [(<ColumnName> [,<ColumnName>])]
   [ON UPDATE RESTRICT|ON DELETE RESTRICT]
[ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]

<CheckExpression> =
Any valid SQL expression that does not include any sub-queries

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code

<ColumnAlterOptions> =

[SET <DefaultExpression>|DROP DEFAULT|DESCRIPTION <Description>|
MOVE TO <ColumnPos>|RESTART WITH <SeedValue>]

<ConstraintDefinition> =

CHECK <CheckExpression>
PRIMARY KEY (<ColumnName> [,<ColumnName>])|
```

```
UNIQUE (<ColumnName> [,<ColumnName>])|
FOREIGN KEY (<ColumnName> [,<ColumnName>])
   REFERENCES <TableName> [(<ColumnName> [,<ColumnName>])]
   [ON UPDATE RESTRICT|ON DELETE RESTRICT]
[ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]
[DESCRIPTION <Description>]

<CheckExpression> =
Any valid SQL expression that does not include any sub-queries

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code

<ConstraintAlterOptions> =

[DESCRIPTION <Description>]
```

Usage

Use this statement to alter the structure of an existing table. You may add new columns or constraints, alter existing columns, or drop existing columns or constraints.

To alter an existing column in a table, use the ALTER COLUMN AS clause. To alter an existing constraint in a table, use the ALTER CONSTRAINT AS clause. These clauses allow for the complete re-definition of a column or constraint.

Note
If you alter an existing column that was previously defined with column-level constraints, then you should not specify the column-level constraints again when altering the column. Doing so will result in a duplicate constraint being added again, possibly causing an error. This is due to the fact that column-level constraints are, except for the NOT NULL constraint, defined as table-level constraints internally in ElevateDB.

The AT clause is 1-based, with 1 being the first column and the column count being the last column.

Note
All clauses after the ADD, ALTER, or DROP clauses are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER TABLE statement was executed.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of any physical table files that were altered during the execution of the statement. Also, this clause does **not** apply to physical backup files created for the database catalog, which are always created and retained.

Examples

```
-- The following statement alters the structure of the
-- Customer table by adding a new Notes column.

ALTER TABLE "Customer"
ADD COLUMN Notes CLOB
```

```
-- The following statement alters the structure of the
-- Customer table by adding a new foreign key constraint
-- on the State column that establishes a referential
-- integrity link to the State table.

ALTER TABLE "Customer"
ADD CONSTRAINT "State_ForeignKey" FOREIGN KEY REFERENCES "State"

-- The following statement alters the structure of the
-- Customer table by renaming the State column to StateProvince.

ALTER TABLE "Customer"
RENAME COLUMN State TO StateProvince
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Column AS	The AS column alteration clause is an ElevateDB extension.
Column COMPUTED	The column COMPUTED clause is an ElevateDB extension.
Column ERROR CODE/MESSAGE	The column ERROR CODE/MESSAGE clause is an ElevateDB extension.
Column DESCRIPTION	The column DESCRIPTION and SET DESCRIPTION clauses are ElevateDB extensions.
Column COMPRESSION	The column COMPRESSION clause is an ElevateDB extension.
Column AT	The column AT clause for adding columns at a specific position is an ElevateDB extension.
Column MOVE TO	The column MOVE TO clause for altering columns and moving them to a specific position is an ElevateDB extension.
RENAME COLUMN	The RENAME COLUMN clause is an ElevateDB extension.
Constraint AS	The AS constraint alteration clause is an ElevateDB extension.
Constraint ON UPDATE	The only option supported for the ON UPDATE clause is the RESTRICT option.
Constraint ON DELETE	The only option supported for the ON DELETE clause is the RESTRICT option.
Constraint ERROR CODE/MESSAGE	The constraint ERROR CODE/MESSAGE clause is an ElevateDB extension.
Constraint DESCRIPTION	The constraint DESCRIPTION and SET DESCRIPTION clauses are ElevateDB extensions.

RENAME CONSTRAINT	The RENAME CONSTRAINT clause is an ElevateDB extension.
DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ENCRYPTED	The ENCRYPTED and UNENCRYPTED clauses are an ElevateDB extension.
INDEX PAGE SIZE	The INDEX PAGE SIZE clause is an ElevateDB extension.
BLOB BLOCK SIZE	The BLOB BLOCK SIZE clause is an ElevateDB extension.
PUBLISH BLOCK SIZE	The PUBLISH BLOCK SIZE clause is an ElevateDB extension.
PUBLISH COMPRESSION	The PUBLISH COMPRESSION clause is an ElevateDB extension.
MAX ROW BUFFER SIZE	The MAX ROW BUFFER SIZE clause is an ElevateDB extension.
MAX INDEX BUFFER SIZE	The MAX INDEX BUFFER SIZE clause is an ElevateDB extension.
MAX BLOB BUFFER SIZE	The MAX BLOB BUFFER SIZE clause is an ElevateDB extension.
MAX PUBLISH BUFFER SIZE	The MAX PUBLISH BUFFER SIZE clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.
NO BACKUP FILES	The NO BACKUP FILES clause is an ElevateDB extension.

5.49 DROP TABLE

Drops an existing table.

Syntax

```
DROP TABLE <Name>
```

Usage

Use this statement to drop a table from a database.

NOTE

You cannot drop a table that is involved in any referential integrity links to other tables via foreign keys. You must first alter the table and drop the foreign key constraints before you will be allowed to drop the table.

Warning
Dropping a table can cause other jobs, functions, procedures, and triggers to generate an error if they refer to the table being dropped.

Examples

```
-- The following statement drops the Customer table.

DROP TABLE "Customer"
```

Required Privileges

The current user must be granted the DROP privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
RESTRICT and CASCADE	ElevateDB does not support the RESTRICT or CASCADE clauses.

5.50 RENAME TABLE

Renames an existing table.

Syntax

```
RENAME TABLE <Name> TO <Name>
```

Usage

Use this statement to rename a table in a database.

> **Warning**
> Renaming a table can cause other jobs, functions, procedures, and triggers to generate an error if they refer to the table being renamed.

Examples

```
-- The following statement renames the Customer
-- table to Cust.

RENAME TABLE "Customer" TO "Cust"
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.51 CREATE TRIGGER

Creates a new trigger on a given table.

Syntax

```
CREATE TRIGGER <Name> <ActionTime> <Type> ON <TableName>
[AT <ExecutionPos>]
[WHEN <Condition>]
<BodyDefinition>
[DESCRIPTION <Description>]

<ActionTime>=BEFORE|AFTER|ERROR

<Type>=INSERT|UPDATE [OF <UpdateColumns>]|DELETE|LOAD UPDATE|ALL

<UpdateColumns>=

<ColumnName> [,<ColumnName>]

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
END
```

Usage

Use this statement to create a new trigger on a table. Triggers can be created to respond to any INSERT, UPDATE, DELETE, or LOAD UPDATE of a row. The AT clause can be used to specify the position (1-based) of the new trigger in relation to any existing triggers on the same table. You can use the WHEN condition to restrict when the trigger will fire as well as the OF clause to restrict the trigger to firing only when certain columns are updated.

OLDROW/NEWROW Row Values

You may refer to the special row identifiers OLDROW and NEWROW anywhere within the WHEN condition or the trigger body itself. They identify the row being inserted, updated, or deleted in the state prior to the action (OLDROW) and after the action (NEWROW).

> **Note**
> NEWROW row values may only be assigned new values from within a BEFORE INSERT, BEFORE UPDATE, BEFORE LOAD UPDATE, ERROR INSERT, ERROR UPDATE, or ERROR LOAD UPDATE trigger body definition. OLDROW row values may only be assigned new values from within a BEFORE LOAD UPDATE or ERROR LOAD UPDATE trigger body. You can use the SET statement to assign a value to any of the OLDROW or NEWROW row values.

For any type of trigger, you can use the LOADINGUPDATES function to determine whether the trigger is executing during the execution of a LOAD UPDATES statement. This is useful for situations where you only want triggers to execute when loading updates, or want to conditionally execute different code depending upon whether the operation is due to a LOAD UPDATES statement execution.

ERROR Triggers

Error triggers are a special kind of trigger that can be defined for insert, update, or delete operations and are called whenever an error occurs during these operations. Normally, the ERRORCODE and ERRORMSG functions are accessible only from within EXCEPTION blocks. However, they are also accessible from anywhere within an error trigger. In addition, the RETRY statement is provided for allowing the trigger to attempt to correct the exception and retry the operation that originally caused the error.

Universal Triggers

Starting in 2.04, you may define a universal trigger using the ALL keyword instead of a specific INSERT, UPDATE, DELETE, or LOAD UPDATE trigger type. This will cause the trigger to be fired for all operations, and you can use the OPERATION function to determine the current operation that caused the trigger to be fired. In the case of a LOAD UPDATE trigger, the current operation is always the type of operation for the update that the LOAD UPDATES statement is currently trying to load.

LOAD UPDATE Triggers

Starting in 2.05, you can also create LOAD UPDATE triggers that are fired for each update being loaded during the execution of the LOAD UPDATES statement. This is useful for being able to respond to update load errors due to constraint violations or missing rows, as well as controlling the update loading process itself by choosing which updates should or should not be loaded (see next section on aborting an operation). Please see the Replication topic for more information on loading updates for a database.

LOAD UPDATE triggers occur before any triggers for the actual update operation occurring. The following shows the order in which the operations that make up the loading of an update occur:

- If the update being loaded is not an INSERT, then the primary key values for the UPDATE or DELETE are loaded.

- The BEFORE LOAD UPDATE triggers are executed, giving you the chance to modify the primary key values in the OLDROW column values before ElevateDB searches for the row in an UPDATE or DELETE operation.

- For UPDATE or DELETE operations, ElevateDB searches for the row using the primary key values.

- The INSERT, UPDATE, or DELETE operation is performed:

  If the update being loaded is an INSERT, then the BEFORE INSERT triggers, INSERT operation, AFTER INSERT triggers, or ERROR INSERT triggers (if any errors occur) are executed.

  If the update being loaded is an UPDATE, then the BEFORE UPDATE triggers, UPDATE operation, AFTER UPDATE triggers, or ERROR UPDATE triggers (if any errors occur) are executed.

  If the update being loaded is a DELETE, then the BEFORE DELETE triggers, DELETE operation, AFTER DELETE triggers, or ERROR DELETE triggers (if any errors occur) are executed.

- The AFTER LOAD UPDATE triggers are executed, with the NEWROW column values representing the column values after the operation has been executed.

- If any errors occur during this entire sequence of operations, then the ERROR LOAD UPDATE triggers are executed.

The OLDROW and NEWROW row values have a specific usage when accessed from within LOAD UPDATE triggers, depending upon the operation being performed. As mentioned above, you can use the OPERATION function to determine the actual operation being performed.

Operation	OLDROW/NEWROW Usage
INSERT	OLDROW values are the column values for the INSERT operation. NEWROW values are the column values after the INSERT operation, and are all NULL for any BEFORE LOAD UPDATE triggers.
UPDATE	OLDROW values are the primary key values used to find the row for the UPDATE operation. NEWROW values are the column values after the UPDATE operation, and are all NULL for any BEFORE LOAD UPDATE triggers.
DELETE	OLDROW values are the primary key values used to find the row for the DELETE operation. NEWROW values are the column values after the DELETE operation, and are all NULL for any BEFORE LOAD UPDATE triggers.
ERROR	OLDROW and NEWROW values depend upon the operation being performed during the loading of the update. Use the OPERATION function to determine how to modify or examine the column values.

Using a Trigger to Abort an Operation

Starting in 2.05, you can use the ABORT statement to abort any INSERT, UPDATE, DELETE, or LOAD UPDATE operation. Aborting an operation sets the aborted flag for the current operation and, after the current trigger is done executing, will cause the operation and any subsequent triggers to be silently ignored. For example, calling ABORT in a BEFORE LOAD trigger will cause the current load operation to stop and the LOAD UPDATES execution to continue on the next update to be loaded, if any more updates are present in the incoming update file.

Note
Calling ABORT does not cause the trigger execution to stop immediately. Any statements after the ABORT statement will continue to execute. If you want to abort the current operation and immediately exit the current trigger being executed, then you should use the ABORT statement followed immediately by the LEAVE statement.

Examples

```
-- This trigger calls the external
-- SendMail procedure with which group to
-- send the email to along with the new
-- value of the Notes column for the customer
-- being updated

CREATE TRIGGER "NotesUpdate" AFTER UPDATE OF "Notes"
ON "Customer"
BEGIN
   CALL SendEmail('CustomerReps',NEWROW.Notes);
END

-- This trigger logs any insert errors that
-- occur during a LOAD UPDATES for
-- the Customer table into a table called
-- LoadErrors

CREATE TRIGGER "LogInsertError" ERROR INSERT ON "customer"
WHEN LOADINGUPDATES()
BEGIN
   DECLARE ErrorData VARCHAR DEFAULT '';

   SET ErrorData = 'Cust #: ' + CAST(NEWROW.CustNo AS VARCHAR);
   SET ErrorData = ErrorData + 'Name: ' + NEWROW.Company;
   SET ErrorData = ErrorData + 'Error #: ' + CAST(ERRORCODE() AS VARCHAR);
   SET ErrorData = ErrorData + 'Error Msg: ' + ERRORMSG();

   EXECUTE IMMEDIATE 'INSERT INTO LoadErrors (''Customer'',''INSERT'',
                                              ''' + ErrorData + '''';
END

-- This trigger updates any new row with
-- a timestamp of when the row was inserted
-- into the Customer table.  The AT clause
-- is used to ensure that this trigger always
-- fires first before any other triggers

CREATE TRIGGER "SetTimeStamp" BEFORE INSERT ON "customer"
AT 1
BEGIN
   SET NEWROW.CreatedOn = CURRENT_TIMESTAMP();
END

-- This trigger examines the primary key
-- values for an update being loaded into
-- the Customer table.  If the SiteID column
```

```
-- value for the update does not match the
-- SiteID column in the System table in the
-- same database, then the loading of the update
-- is aborted using the ABORT statement.
-- NOTE: the column being filtered on, in this case
-- the SiteID column, must be part of the primary
-- key in order for it to be non-NULL in the
-- OLDROW column values for UPDATE and DELETE
-- operations.

CREATE TRIGGER "FilterUpdates" BEFORE LOAD UPDATE ON "customer"
BEGIN
   DECLARE SiteID INTEGER DEFAULT 0;

   EXECUTE IMMEDIATE 'SELECT SiteID INTO ? FROM System' USING SiteID;

   IF OLDROW.SiteID <> SiteID THEN
      ABORT;
   END IF;
END
```

Required Privileges

The current user must be granted the CREATE privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
REFERENCING	The REFERENCING clause is not supported in ElevateDB and the old and new row values are always referred to as OLDROW and NEWROW.
FOR EACH	The FOR EACH clause is not supported. ElevateDB triggers are always fired on a row basis and never on a statement basis.
DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.

5.52 ALTER TRIGGER

Alters an existing trigger on a given table.

Syntax

```
ALTER TRIGGER <Name> <ActionTime> <Type> ON <TableName>
[AT <ExecutionPos>]
[WHEN <Condition>]
<BodyDefinition>
[DESCRIPTION <Description>]

<ActionTime>=BEFORE|AFTER|ERROR

<Type>=INSERT|UPDATE [OF <UpdateColumns>]|DELETE|LOAD UPDATE|ALL

<UpdateColumns>=

<ColumnName> [,<ColumnName>]

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
END
```

Usage

Use this statement to alter an existing trigger.

> **Note**
> All clauses after the body definition are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER TRIGGER statement was executed.

Examples

```
-- The following statement changes the description of the
-- NotesUpdate trigger.

ALTER TRIGGER "NotesUpdate" AFTER UPDATE OF "Notes"
ON "Customer"
BEGIN
   CALL SendEmail('CustomerReps',NEWROW.Notes);
END
DESCRIPTION 'Sends an email to all customer reps when Notes is updated'
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.53 DROP TRIGGER

Drops an existing trigger from a given table.

Syntax

```
DROP TRIGGER <Name> FROM <TableName>
```

Usage

Use this statement to drop a trigger from a table.

Examples

```
-- The following statement drops the UpdateNotes trigger.

DROP TRIGGER "UpdateNotes" FROM "Customer"
```

Required Privileges

The current user must be granted the DROP privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
FROM Clause	The FROM clause is an ElevateDB extension. Triggers are a table-level object in ElevateDB, whereas they are a schema-level object in the standard.

5.54 RENAME TRIGGER

Renames an existing trigger on a given table.

Syntax

```
RENAME TRIGGER <Name> ON <TableName>
TO <Name>
```

Usage

Use this statement to rename a trigger on a table.

Examples

```
-- The following statement renames the UpdateNotes
-- trigger to UpdNotes.

RENAME TRIGGER "UpdateNotes" ON "Customer"
TO "UpdNotes"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.55 ENABLE TRIGGER

Enables an existing trigger on a given table.

Syntax

```
ENABLE TRIGGER <Name> ON <TableName>
```

Usage

Use this statement to enable a trigger on a table. If the trigger is already enabled, then this statement does nothing.

Note
Triggers are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement enables the UpdateNotes trigger

ENABLE TRIGGER "UpdateNotes" ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.56 DISABLE TRIGGER

Disables an existing trigger on a given table.

Syntax

```
DISABLE TRIGGER <Name> ON <TableName>
```

Usage

Use this statement to disable a trigger on a table. If the trigger is already disabled, then this statement does nothing.

Note
Triggers are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement disables the UpdateNotes trigger

DISABLE TRIGGER "UpdateNotes" ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.57 ENABLE TRIGGERS

Enables all existing triggers on a given table.

Syntax

```
ENABLE TRIGGERS ON <TableName>
```

Usage

Use this statement to enable all triggers on a table. If any of the triggers are already enabled, then this statement does nothing for those triggers.

Note
Triggers are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement enables all triggers on
-- the Customer table

ENABLE TRIGGERS ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.58 DISABLE TRIGGERS

Disables all existing triggers on a given table.

Syntax

```
DISABLE TRIGGERS ON <TableName>
```

Usage

Use this statement to disable all triggers on a table. If any of the triggers are already disabled, then this statement does nothing for those triggers.

Note
Triggers are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement disables all triggers on
-- the Customer table

DISABLE TRIGGERS ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.59 ENABLE DEFAULTS

Enables the use of default column values for inserts on a given table.

Syntax

```
ENABLE DEFAULTS ON <TableName>
```

Usage

Use this statement to enable the use of default column values on a table. By default, default column values are enabled for all tables.

You can find out if default column values are enabled or not by querying the Tables system information table.

> **Note**
> Default column values are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement enables all default column values on
-- the Customer table

ENABLE DEFAULTS ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.60 DISABLE DEFAULTS

Disables the use of default column values for inserts on a given table.

Syntax

```
DISABLE DEFAULTS ON <TableName>
```

Usage

Use this statement to disable the use of default column values on a table. By default, default column values are enabled for all tables.

You can find out if default column values are enabled or not by querying the Tables system information table.

Note
Default column values are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement disables all default column values on
-- the Customer table

DISABLE DEFAULTS ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.61 ENABLE GENERATED

Enables the evaluation of generated column values for inserts and updates on a given table.

Syntax

```
ENABLE GENERATED ON <TableName>
```

Usage

Use this statement to enable the evaluation of generated column values on a table. By default, generated column values are enabled for all tables.

You can find out if generated column values are enabled or not by querying the Tables system information table.

Note
Generated column values are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement enables all generated column values on
-- the Customer table

ENABLE GENERATED ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.62 DISABLE GENERATED

Disables the evaluation of generated column values for inserts and updates on a given table.

Syntax

```
DISABLE GENERATED ON <TableName>
```

Usage

Use this statement to disable the evaluation of generated column values on a table. By default, generated column values are enabled for all tables.

You can find out if generated column values are enabled or not by querying the Tables system information table.

> **Note**
> Generated column values are enabled or disabled on a per-session basis, so this statement only affects the current session.

Examples

```
-- The following statement disables all generated column values on
-- the Customer table

DISABLE GENERATED ON "Customer"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.63 CREATE INDEX

Creates a new index on a given table.

Syntax

```
CREATE INDEX <Name> ON <TableName>|<ViewName>
(<ColumnName> [COLLATE <CollationName>]
   [[ASC|ASCENDING]|[DESC|DESCENDING]] [,<ColumnName>])
[DESCRIPTION <Description>]
[NO BACKUP FILES]
```

Usage

Use this statement to create a new index on a table or non-updateable view. Multiple columns can be defined for an index, however it is recommended that you try to keep the number and size of the columns, and subsequently the size of the index keys in the index, to a minimum for performance purposes.

If a collation is specified for a CHAR, VARCHAR, or CLOB column, it overrides the default collation specified for the column being included in the index.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of any physical table files that were altered during the execution of the statement. Also, this clause does **not** apply to physical backup files created for the database catalog, which are always created and retained.

As of ElevateDB 2.21, you can now index non-updateable views. Non-updateable views are views that generate a static, insensitive result set.

> **Note**
> If you alter a view definition using the ALTER VIEW DDL statement, it will remove all defined indexes for the view and will require that you use this statement to recreate the indexes.

Examples

```
-- The following statement creates a Name index on the
-- Customer table consisting of the Name column in
-- ascending order.

CREATE INDEX "Name" ON "Customer" (Name ASC)

-- The following statement creates a State index on the
-- Customer table consisting of the State column in
-- ascending order.

CREATE INDEX "State" ON "Customer" (State ASC)
```

Required Privileges

The current user must be granted the CREATE privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.64 CREATE TEXT INDEX

Creates a new text index on a given table.

Syntax

```
CREATE TEXT INDEX <Name> ON <TableName>
(<ColumnName> [COLLATE <CollationName>])
[DESCRIPTION <Description>]
[INDEXED WORD LENGTH <WordLength>]
[FILTER TYPE COLUMN <ColumnName>]
[WORD GENERATOR <WordGeneratorName>]
[NO BACKUP FILES]
```

Usage

Use this statement to create a new text index on a table column. Please see the Text Indexing topic for more information.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of any physical table files that were altered during the execution of the statement. Also, this clause does **not** apply to physical backup files created for the database catalog, which are always created and retained.

Examples

```
-- The following statement creates a text index on the
-- Notes column in the Customer table.  Notice that the collation
-- for the Notes column is overridden with the case-insensitive
-- ANSI collation.

CREATE TEXT INDEX "Notes" ON "Customer"
(Notes COLLATE ANSI_CI)
INDEXED WORD LENGTH 20

-- The following statement creates a text index on the
-- Notes column in the Customer table.  In this example,
-- however, another column called TextType is used to
-- determine the type of text in the Notes column so that
-- it can be properly filtered using a text filter.  This
-- will allow us to store HTML, RTF, and other non-plain
-- text in the Notes column and have it be indexed properly.

CREATE TEXT INDEX "Notes" ON "Customer"
(Notes COLLATE ANSI_CI)
INDEXED WORD LENGTH 20
FILTER TYPE COLUMN "TextType"
```

Required Privileges

The current user must be granted the CREATE privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.65 ALTER INDEX

Alters an existing index on a given table.

Syntax

```
ALTER INDEX <Name> ON <TableName>|<ViewName>
(<ColumnName> [COLLATE <CollationName>]
   [[ASC|ASCENDING]|[DESC|DESCENDING]] [,<ColumnName>])
[DESCRIPTION <Description>]
[NO BACKUP FILES]
```

Usage

Use this statement to alter an existing index in a table or non-updateable view.

Note
All clauses after the column definitions are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER INDEX statement was executed.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of any physical table files that were altered during the execution of the statement. Also, this clause does **not** apply to physical backup files created for the database catalog, which are always created and retained.

As of ElevateDB 2.21, you can now index non-updateable views. Non-updateable views are views that generate a static, insensitive result set.

Examples

```
-- The following statement changes the Name index on the
-- Customer table so that the Name column is sorted case-insensitive

ALTER INDEX "Name" ON "Customer" (Name COLLATE "ANSI_CI" ASC)
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Extension	This SQL statement is an ElevateDB extension.

5.66 ALTER TEXT INDEX

Alters an existing text index on a given table.

Syntax

```
ALTER TEXT INDEX <Name> ON <TableName>
(<ColumnName> [COLLATE <CollationName>])
[DESCRIPTION <Description>]
[INDEXED WORD LENGTH <WordLength>]
[FILTER TYPE COLUMN <ColumnName>]
[WORD GENERATOR <WordGeneratorName>]
[NO BACKUP FILES]
```

Usage

Use this statement to alter an existing text index on a table column. Please see the Text Indexing topic for more information.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of any physical table files that were altered during the execution of the statement. Also, this clause does **not** apply to physical backup files created for the database catalog, which are always created and retained.

> **Note**
> All clauses after the column definitions are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER INDEX statement was executed.

Examples

```
-- The following statement changes the Notes text
-- index so that the indexed word length is 30 characters

ALTER TEXT INDEX "Notes" ON "Customer"
(Notes COLLATE ANSI_CI)
INDEXED WORD LENGTH 30
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.67 DROP INDEX

Drops an existing index from a given table.

Syntax

```
DROP INDEX <Name> FROM <TableName>|<ViewName>
[NO BACKUP FILES]
```

Usage

Use this statement to drop an index or text index from a table or non-updateable view.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of any physical table files that were altered during the execution of the statement. Also, this clause does **not** apply to physical backup files created for the database catalog, which are always created and retained.

As of ElevateDB 2.21, you can now index non-updateable views. Non-updateable views are views that generate a static, insensitive result set.

> **Warning**
> Dropping an index from a table can affect the performance of DML statements that refer to the table columns defined for the index in JOIN or WHERE clauses.

Examples

```
-- The following statement drops the Name index.

DROP INDEX "Name" FROM "Customer"
```

Required Privileges

The current user must be granted the DROP privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.68 RENAME INDEX

Renames an existing index on a given table.

Syntax

```
RENAME INDEX <Name> ON <TableName>|<ViewName>
TO <Name>
```

Usage

Use this statement to rename an index or text index on a table or non-updateable view.

As of ElevateDB 2.21, you can now index non-updateable views. Non-updateable views are views that generate a static, insensitive result set.

Examples

```
-- The following statement renames the Name
index to CustName.

RENAME INDEX "Name" FROM "Customer"
TO "CustName"
```

Required Privileges

The current user must be granted the ALTER privilege on the specified table in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.69 CREATE VIEW

Creates a new view.

Syntax

```
CREATE VIEW <Name> AS
<View Definition>
[WITH CHECK OPTION|WITHOUT CHECK OPTION]
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]
```

Usage

Use this statement to create a new view. A view is a query expression that can act like a virtual table, and is useful when you want restrict which columns in a table (or tables) are visible to users. This can be accomplished by giving users (or roles) SELECT privileges on a view that only references a few select columns while not giving users (or roles) any SELECT privileges on the base table(s) that are referenced by the view.

The WITH CHECK OPTION clause is used with updateable views to specify whether INSERTS or UPDATES that would violate the WHERE clause will be permitted or not. If WITH CHECK OPTION is specified, then INSERTS or UPDATES that would violate the WHERE clause are not permitted.

Note
Using the WITHOUT CHECK OPTION clause is the same as not specifying the WITH CHECK OPTION clause, and is present for compatibility with the ALTER VIEW syntax.

By default, ElevateDB always tries to make a view updateable if possible. The rules for updateability are the same as those for sensitive query result sets, and are discussed in detail in the Result Set Cursor Sensitivity topic.

Note
It is possible to have a view be considered as updateable and still be read-only. Such is the case in situations where the current view SQL does fullfill the requirements for a sensitive result set, but the view references other views or derived tables that are not updateable. In such a case, the current view will inherit the updateable state of the referenced views or derived tables.

Any time the columns in any referenced base table or view change, ElevateDB will automatically reflect these changes in the view columns. You can always query this information via the ViewColumns Information schema table.

Examples

```
-- The following view selects the employee Name and
```

```
-- HireDate column from the Employees table.

CREATE VIEW "EmployeesList" AS
SELECT Name, HireDate
FROM Employees

-- The following view uses a derived table to retrieve
-- data.  It will be considered updateable, but will not
-- be updateable at runtime.

CREATE VIEW "DerivedSum" AS
SELECT *
FROM (SELECT CustNo, SUM(Orders.ItemsTotal) AS Total
      FROM Customer INNER JOIN Orders ON Orders.CustNo=Customer.CustNo
      GROUP BY CustNo) AS CustomerTotals
WHERE Total > 80000
```

Required Privileges

The current user must be granted the CREATE privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Columns List	ElevateDB does not support specifying a list of column names for the view. Instead, it always uses the column correlation names from the SELECT columns in the query expression to determine the names of the columns in the view.
RECURSIVE	ElevateDB does not support the RECURSIVE clause and recursive views. This means that you cannot reference the view being created within the view definition.
LOCAL or CASCADED	ElevateDB does not support the LOCAL or CASCADED clauses in the WITH CHECK OPTION clause.
DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.

5.70 ALTER VIEW

Alters an existing view.

Syntax

```
ALTER VIEW <Name> AS
<View Definition>
[WITH CHECK OPTION|WITHOUT CHECK OPTION]
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]
```

Usage

Use this statement to alter an existing view.

> **Note**
> All clauses after the view definition are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER VIEW statement was executed.

> **Note**
> If you alter a view definition using this statement, it will remove any defined indexes for the view and will require that you use the CREATE INDEX statement to recreate the index(es).

Examples

```
-- The following statement changes the description of the
-- EmployeesList view.

ALTER VIEW "EmployeesList" AS
SELECT Name, HireDate
FROM Employees
DESCRIPTION 'List of all employees and hire dates'
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension

5.71 DROP VIEW

Drops an existing view.

Syntax

```
DROP VIEW <Name>
```

Usage

Use this statement to drop a view from a database.

Warning
Dropping a view can cause other jobs, functions, procedures, and triggers to generate an error if they refer to the view being dropped.

Examples

```
-- The following statement drops the EmployeesList view.

DROP VIEW "EmployeesList"
```

Required Privileges

The current user must be granted the DROP privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
RESTRICT and CASCADE	ElevateDB does not support the RESTRICT or CASCADE clauses.

5.72 RENAME VIEW

Renames an existing view.

Syntax

```
RENAME VIEW <Name> TO <Name>
```

Usage

Use this statement to rename a view in a database.

> **Warning**
> Renaming a view can cause other jobs, functions, procedures, and triggers to generate an error if they refer to the view being renamed.

Examples

```
-- The following statement renames the EmployeesList
-- view to Employees.

RENAME VIEW "EmployeesList" TO "Employees"
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.73 CREATE FUNCTION

Creates a new function.

Syntax

```
CREATE FUNCTION <Name>
([<ParamDefinition>[,ParamDefinition]])
RETURNS <DataType>
EXTERNAL NAME <ModuleName> | <BodyDefinition>
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]

<ParamDefinition> =

<Mode> <Name> <DataType> [<Description>]

<Mode> =

IN|OUT|INOUT

<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
   RETURN <Expression>
```

```
[EXCEPTION
   [<Statement>;]]
END
```

Usage

Use this statement to create a new function in a given database. Functions can be used in jobs, other functions and procedures, triggers, DML statements, and catalog-bound expressions such as table column default expressions and table constraint check expressions.

Examples

```
-- This function looks up the sales tax
-- rate for a given state and county

CREATE FUNCTION LookupSalesTaxRate(IN State CHAR(2), IN County VARCHAR)
RETURNS DECIMAL(19,2)
BEGIN
   DECLARE TempCursor CURSOR FOR stmt;
   DECLARE Result DECIMAL(19,2) DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM SalesTaxes WHERE State = ? AND County =
      ?';

   OPEN TempCursor USING State, County;

   IF (ROWCOUNT(TempCursor) > 0) THEN
      FETCH FIRST FROM TempCursor ('TaxRate') INTO Result;
   END IF;

   CLOSE TempCursor;

   RETURN Result;
END
```

Required Privileges

The current user must be granted the CREATE privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.

5.74 ALTER FUNCTION

Alters an existing function.

Syntax

```
ALTER FUNCTION <Name>
([<ParamDefinition>[,ParamDefinition]])
RETURNS <DataType>
EXTERNAL NAME <ModuleName> | <BodyDefinition>
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]

<ParamDefinition> =

<Mode> <Name> <DataType> [<Description>]

<Mode> =

IN|OUT|INOUT

<DataType> =

CHARACTER|CHAR [(<Length>)] [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>)] [<CollationName>]
GUID
BYTE [(<LengthInBytes>)]
BYTE VARYING|VARBYTE [(<LengthInBytes>)]
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
   RETURN <Expression>
```

```
[EXCEPTION
   [<Statement>;]]
END
```

Usage

Use this statement to alter an existing function.

Note
All clauses after the body definition are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER FUNCTION statement was executed.

Examples

```
-- The following statement changes the description of the
-- LookupSalesTaxRate function.

ALTER FUNCTION LookupSalesTaxRate(IN State CHAR(2), IN County VARCHAR)
RETURNS DECIMAL(19,2)
BEGIN
   DECLARE TempCursor CURSOR FOR stmt;
   DECLARE Result DECIMAL(19,2) DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM SalesTaxes WHERE State = ? AND County =
      ?';

   OPEN TempCursor USING State, County;

   IF (ROWCOUNT(TempCursor) > 0) THEN
      FETCH FIRST FROM TempCursor ('TaxRate') INTO Result;
   END IF;

   CLOSE TempCursor;

   RETURN Result;
END
DESCRIPTION 'Function for looking up sales tax rates'
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.

5.75 DROP FUNCTION

Drops an existing function.

Syntax

```
DROP FUNCTION <Name>
```

Usage

Use this statement to drop a function from a database.

Warning
Dropping a function can cause other jobs, functions, procedures, views, triggers, constraints, or column defaults to generate an error if they refer to the function being dropped.

Examples

```
-- The following statement drops the LookupTaxRate function.

DROP FUNCTION "LookupTaxRate"
```

Required Privileges

The current user must be granted the DROP privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
RESTRICT and CASCADE	ElevateDB does not support the RESTRICT or CASCADE clauses.

5.76 RENAME FUNCTION

Renames an existing function.

Syntax

```
RENAME FUNCTION <Name> TO <Name>
```

Usage

Use this statement to rename a function in a database.

Warning
Renaming a function can cause other jobs, functions, procedures, views, triggers, constraints, or column defaults to generate an error if they refer to the function being renamed.

Examples

```
-- The following statement renames the LookupTaxRate
-- function to LookupRate.

RENAME FUNCTION "LookupTaxRate" TO "LookupRate"
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

5.77 CREATE PROCEDURE

Creates a new procedure.

Syntax

```
CREATE PROCEDURE <Name>
([<ParamDefinition>[,ParamDefinition]])
EXTERNAL NAME <ModuleName> | <BodyDefinition>
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]

<ParamDefinition> =

<Mode> <Name> <DataType> [<Description>]

<Mode> =

IN|OUT|INOUT

<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION
   [<Statement>;]]
```

```
END
```

Usage

Use this statement to create a new procedure in a given database. Procedures can be used in jobs, other functions and procedures, and triggers.

Note
If you wish to return a result set from a procedure, declare the cursor in the procedure using the WITH RETURN clause and leave the cursor open when the procedure completes.

Examples

```
-- The following procedure updates any Customer row
-- with a State column value of 'FL' to 'NY' and returns a cursor
-- on the Customer table.

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor ('State') INTO State;

   WHILE NOT EOF(CustCursor) DO
      IF (State = 'FL') THEN
         UPDATE CustCursor SET 'State' = 'NY';
      END IF;
      FETCH NEXT FROM CustCursor ('State') INTO State;
   END WHILE;
END
```

Required Privileges

The current user must be granted the CREATE privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.

5.78 ALTER PROCEDURE

Alters an existing procedure.

Syntax

```
ALTER PROCEDURE <Name>
([<ParamDefinition>[,ParamDefinition]])
EXTERNAL NAME <ModuleName> | <BodyDefinition>
[DESCRIPTION <Description>]
[VERSION <VersionNumber>]
[ATTRIBUTES <CustomAttributes>]

<ParamDefinition> =

<Mode> <Name> <DataType> [<Description>]

<Mode> =

IN|OUT|INOUT

<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<BodyDefinition> =

BEGIN
   [<Declaration>;]
   [<Declaration>;]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION
   [<Statement>;]]
```

```
END
```

Usage

Use this statement to alter an existing procedure.

> **Note**
> All clauses after the body definition are optional. If they are not specified, then they will not be altered and will stay the same as before the ALTER PROCEDURE statement was executed.

Examples

```
-- The following statement changes the description of the
-- UpdateState procedure.

ALTER PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor ('State') INTO State;

   WHILE NOT EOF(CustCursor) DO
      IF (State = 'FL') THEN
         UPDATE CustCursor SET 'State' = 'NY';
      END IF;
      FETCH NEXT FROM CustCursor ('State') INTO State;
   END WHILE;
END
DESCRIPTION 'Procedure for updating FL states to NY'
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

DESCRIPTION	The DESCRIPTION clause is an ElevateDB extension.
VERSION	The VERSION clause is an ElevateDB extension.
ATTRIBUTES	The ATTRIBUTES clause is an ElevateDB extension.

5.79 DROP PROCEDURE

Drops an existing procedure.

Syntax

```
DROP PROCEDURE <Name>
```

Usage

Use this statement to drop a procedure from a database.

> **Warning**
> Dropping a procedure can cause other jobs, functions, procedures, and triggers to generate an error if they refer to the procedure being dropped.

Examples

```
-- The following statement drops the UpdateState procedure.

DROP PROCEDURE "UpdateState"
```

Required Privileges

The current user must be granted the DROP privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
RESTRICT and CASCADE	ElevateDB does not support the RESTRICT or CASCADE clauses.

5.80 RENAME PROCEDURE

Renames an existing procedure.

Syntax

```
RENAME PROCEDURE <Name> TO <Name>
```

Usage

Use this statement to rename a procedure in a database.

Warning
Renaming a procedure can cause other jobs, functions, procedures, and triggers to generate an error if they refer to the procedure being renamed.

Examples

```
-- The following statement renames the UpdateState
-- procedure to UpdState.

RENAME PROCEDURE "UpdateState" TO "UpdState"
```

Required Privileges

The current user must be granted the ALTER privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

Chapter 6
DML Statements

6.1 Introduction

DML (data manipulation language) statements are used to select, insert, update, or delete rows from tables in an ElevateDB database. This section of the manual details the available DML statements in ElevateDB.

Notation

The notation used in the syntax section for each DML statement is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

6.2 SELECT

Selects rows from one or more tables.

Syntax

```
SELECT [ALL|DISTINCT]
*|<SelectColumn> [,<SelectColumn>]
[INTO <OutputParameter>[,<OutputParameter>]]
[FROM <SelectTable> [<Join> [,<Join>]|,<SelectTable>]]
[WHERE <FilterCondition>]
[GROUP BY <GroupColumn> [,<GroupColumn>]]
[HAVING <GroupFilterCondition>]
[UNION|
UNION ALL|
INTERSECT|
INTERSECT ALL|
EXCEPT|
EXCEPT ALL <QueryExpression>]
[ORDER BY <OrderColumn> [,<OrderColumn>]]
[RANGE <Start> [TO <End>]]
[NOJOINOPTIMIZE]
[JOINOPTIMIZECOSTS]
[JOININDEXTHRESHHOLD <ThreshholdPercent>]

<SelectColumn> = <ColumnExpression> [AS <ColumnCorrelationName>]

<SelectTable> = <TableName>|<ViewName>|<DerivedTable> [AS
      <TableCorrelationName>]

<DerivedTable> =

(<SELECT Statement>)

<Join> = [INNER|[LEFT|RIGHT OUTER] JOIN <SelectTable> ON <JoinCondition>

<GroupColumn> = <ColumnExpression> [COLLATE <CollationName>]

<OrderColumn> = <ColumnExpression> [COLLATE <CollationName>]
[[ASC|ASCENDING]|[DESC|DESCENDING]]

<Start> = INTEGER
<End> = INTEGER

<ThreshholdPercent> = INTEGER
```

Usage

Use this statement to select rows from one or more tables in an ElevateDB database. The SELECT statement generates a result set that will contain the selected rows in the using the grouping specified by the GROUP BY statement, if present, and the ordering specified by the ORDER BY clause, if present.

ALL and DISTINCT Clauses

The ALL and DISTINCT clauses control whether the generated result set contains duplicate rows. The DISTINCT clause prevents duplicate rows while the ALL clause outputs all rows, including duplicates. The ALL clause is the default condition and does not need to be specified in order to allow for duplicate rows in the result set.

SELECT Columns

The SELECT column list specifies the list of columns to be output into the result set. The columns specified in the SELECT column list can contain any combination of columns and valid SQL expressions. The only requirement is that any column references be valid in the context of the tables being selected from via the FROM clause (see below) or via any sub-queries that are present as part of a valid SQL expression.

The special columns wildcard * can be used to specify that all columns from the first table in the FROM clause, or a specific table if prefaced with the table name using the <TableName>.* notation, be output into the result set.

Use the AS clause to output any SELECT column in the result set using a specific column name.

> **Note**
> Any duplicate SELECT column names will be output using a numbered suffix in order to make them unique. Furthermore, any SQL expressions without an associated AS clause will be output using a special column name of:

```
Expression
```

for general SQL expressions and a column name of:

```
<AggregateFunction> OF <ColumnName>|ALL
```

for any aggregate function expressions using the MIN, MAX, SUM, RUNSUM, AVG, STDDEV, or COUNT functions.

INTO Clause

The INTO clause allows you to specify one or more output parameters as the target of a SELECT statement. Such a statement doesn't return a result set at all, which is useful for situations where you only want one, or a few, values from a specific row in a table.

> **Note**
> The use of the INTO clause requires that the SELECT statement only return a single row. If the SELECT statement returns more than one row, then an exception will be raised.

FROM Clause

The FROM clause specifies the table or view, or tables or views, from which the rows in the result set should be selected. In addition, ElevateDB supports the use of derived tables in the FROM clause. A derived table is another SELECT statement enclosed in parentheses, and can be any valid SELECT

statement. ElevateDB uses temporary views to implement derived tables.

Each table or view can be assigned a correlation name that is used instead of the actual table or view name in column references. This is useful when you must specify the same table or view name more than once in the FROM clause, such as is the case when joining a table or view to itself. Each table or view's name or correlation name must be unique in the context of the FROM clause. Derived tables, however, must be assigned a correlation name so that the derived table can be indentified elsewhere in the SELECT statement.

Note
As of ElevateDB 2.08, the FROM clause is optional. If you do not include the FROM clause, then you cannot specify the WHERE, GROUP BY, HAVING, NOJOINOPTIMIZE, JOINOPTIMIZECOSTS, ORDER BY, or RANGE clauses. Executing a SELECT statement without a FROM clause is useful for retrieving information via system functions such as the CURRENT_USER or CURRENT_DATABASE function.

If more than one table is specified, then the JOIN clause can be used to specify the relationship(s) between the tables. ElevateDB supports three different JOIN clauses:

Join Clause	Description
INNER JOIN	An INNER join specifies that any rows output into the result set from the target table of the join must match the join expression specified in the join expression. If any row from the target table does not match the join expression, then it is discarded.
LEFT OUTER JOIN	A LEFT OUTER join specifies that any rows output into the result set from the target table of the join must match the join expression specified in the join expression. If any row from the target table does not match the join expression, then NULL values are generated for all column references to the target table in the SELECT column list.
RIGHT OUTER JOIN	A RIGHT OUTER join is the exact opposite of a LEFT OUTER JOIN and specifies that any rows output into the result set from the source table of the join must match the join expression specified in the join expression. If any row from the source table does not match the join expression, then NULL values are generated for all column references to the source table in the SELECT column list.

Note
If you specify multiple tables in the FROM clause without specifying JOIN clauses between all of them, then the tables without applicable JOIN clauses will be joined using a CROSS JOIN, which is a join that joins every row from the source table to every row in the target table. This produces a cartesian product of both tables, and even very small tables can result in very large result sets, so one should be careful to ensure that join conditions are always specified for all tables in the SELECT statement.

WHERE Clause

The WHERE clause is used to filter the rows output into the result set after the rows have been filtered

using any join expressions that may be present. The WHERE clause can contain any valid boolean SQL expression.

> **Note**
> Aggregate functions such as the MIN, MAX, or SUM functions cannot be used anywhere in the WHERE clause. Also, do not specify joins in the WHERE clause according to the outdated SQL-89 SQL standard. Use the SQL-92 or higher standard JOIN syntax mentioned above instead. ElevateDB will not optimize any joins that are specified in the WHERE clause.

GROUP BY Clause

The GROUP BY clause is used to group the rows output into the result set by one or more SQL columns or expressions. Each GROUP BY column or expression may optionally include a COLLATE clause that specifies the collation that should be used for the grouping.

> **Note**
> Any aggregate functions such as the MIN, MAX, or SUM functions in the SELECT column list will be aggregated based upon the columns specified in the GROUP BY clause. If aggregate functions are present in the SELECT column list, but no GROUP BY clause is specified, then the result set will contain a single row.

HAVING Clause

The HAVING clause is used to filter any rows after they have been grouped using the GROUP BY clause, but before they are output to the result set. The HAVING clause can contain any valid boolean SQL expression. Also, aggregate functions are allowed to be used in the HAVING clause.

UNION, INTERSECT, and EXCEPT Clauses

The UNION, INTERSECT, and EXCEPT clauses are used to perform set operations between two query expressions. The SELECT column list of the query expressions involved in a set operation must contain the same number of columns or expressions, and the columns or expressions must be type-compatible. The set operations work as follows:

Clause	Description
UNION	Outputs the rows of both query expressions into the result set.
INTERSECT	Outputs the rows of the first query expression that match the rows of the second query expression into the result set.
EXCEPT	Outputs the rows of the first query expression that do not match the rows of the seoncd query expressions into the result set.

By default, non-distinct rows are aggregated into single rows in a UNION, INTERSECT, or EXCEPT operation. Use the ALL clause to retain non-distinct rows.

ORDER BY Clause

The ORDER BY clause is used to order the rows output into the result set by one or more SQL columns or

expressions. Each ORDER BY column or expression may optionally include a COLLATE clause that specifies the collation that should be used for the ordering and an ASCENDING or DESCENDING clause that specifies the direction in which the ordering should be performed. The default direction is ASCENDING.

RANGE Clause

The RANGE clause is used to limit the rows generated in the result set to the sequential range specified, with the start value being the first row to return and the end value being the last. The TO clause and end value are optional, and can be left off if you wish to return all of the rows in the result set starting with the specified first row.

You may use dynamic parameter markers instead of constant values in the RANGE clause. This permits you to prepare the query once, and then execute it multiple times with different ranges in the result set without forcing ElevateDB to re-compile the query. See your product-specific manual for more information on preparing and executing parameterized queries.

Incremental Result Set Population

The RANGE clause can also be used with insensitive result sets. ElevateDB can use the ending row value for the range to perform incremental population of the result set, resulting in better performance when you only want to see a small set of rows at a time. Combined with dynamic parameters for the start and end values, this allows you to incrementally populate the result set as each set of rows is viewed. For example, consider the following SQL, set in the client application to return an insensitive result set:

```
SELECT * FROM Orders
RANGE ? TO ?
```

Once this query is prepared, you may then execute the query many times with different values. In this example, let's assume that the first execution uses 1 as the starting parameter value and 20 for the ending parameter value. This will cause ElevateDB to populate the first 20 rows in the result set, and return these rows as the insensitive result set. The second execution uses 21 as the starting parameter value and 40 as the ending parameter value. This will cause ElevateDB to populate the next 20 rows in the result set, and return rows 21 through 40 as the insensitive result set. The third execution uses the starting and ending values of 1 and 20 again. In this case, ElevateDB won't populate any more rows into the result and will quickly return the first 20 rows again as the result set.

Please see the Result Set Cursor Sensitivity topic for more information on sensitive and insensitive result sets.

Scalar Queries

Scalar queries are SELECT statements that result in a single row containing exactly one column. Such queries can be used almost anywhere that a normal scalar value would be used. However, it is important to note that if such a query returns more than a single row, or more than one column in the single row, then an exception will be raised.

NOJOINOPTIMIZE Clause

The NOJOINOPTIMIZE clause is used to force the query optimizer to stop re-ordering joins for a SELECT statement. In certain rare cases the query optimizer might not have enough information to know that re-ordering the joins will result in worse performance than if the joins were left in their original order, so in such cases you can include this clause to force the query optimizer to not perform the join re-ordering.

Note
Only INNER JOIN expressions can be re-ordered by the query optimizer. LEFT and RIGHT OUTER JOIN expressions cannot be re-ordered.

JOINOPTIMIZECOSTS Clause

The JOINOPTIMIZECOSTS clause is used to force the query optimizer to use I/O cost projections to determine the most efficient way to process a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

JOININDEXTHRESHHOLD Clause

As of ElevateDB 2.26, there is a new JOININDEXTHRESHHOLD keyword available for the SELECT statement. This keyword controls how ElevateDB handles optimized (indexed) WHERE conditions on tables that are the target of INNER JOINs. For more general information, please see the **How ElevateDB Selects the Rows** section in the Optimizer topic.

Previously, ElevateDB would simply use any available, usable index and build a bitmap that represented the set of rows, irrespective of how many rows were selected. This works fine when there are no joins, but can be problematic when the number of rows selected is large and the table is also the target of an INNER JOIN. In such cases, the INNER JOIN condition's bitmap must constantly be assigned/ANDed with the WHERE condition's bitmap and, because the join condition's bitmap typically represents a much smaller set of rows than the WHERE condition, this process of reconciling the bitmaps becomes computationally expensive and a drag on performance.

The value provided with the JOININDEXTHRESHHOLD clause is an integer value representing a percentage of rows that, when exceeded, causes ElevateDB to treat such WHERE conditions as un-optimized row scans instead of index scans. This eliminates the computationally expensive bitmap operations and drastically improves the performance of the SELECT statement. The default value for the JOININDEXTHRESHHOLD is 75. This means that a WHERE condition must select at least 75% of the rows in a table is also the target of an INNER JOIN condition in order to be converted into a row scan.

Examples

```
-- This SELECT statement selects several columns
-- from the OrderItems table along with an expression
-- for computing the extended price of an ordered item

SELECT OrderNo,
LineNo,
ItemNo,
QtyOrdered,
UnitPrice,
(QtyOrdered * UnitPrice) AS ExtendedPrice
FROM OrderItems

-- This SELECT statement selects all columns
-- from the Orders and OrderItems tables
-- joined on the OrderNo column.  Note that
-- this statement will not output any rows
-- into the result set for any rows in the
-- Orders table that do not have a corresponding
-- row in the OrderItems table
```

```
SELECT Orders.*,
OrderItems.*
FROM Orders INNER JOIN OrderItems ON
Orders.OrderNo = OrderItems.OrderNo

-- This SELECT statement solves the previous
-- issue with missing OrderItems rows by using
-- a LEFT OUTER JOIN instead.  If a corresponding
-- row does not exist in the OrderItems table
-- for a given Orders row, then the Orders row
-- will still be included and NULL values will
-- be output for all OrderItems columns

SELECT Orders.*,
OrderItems.*
FROM Orders LEFT OUTER JOIN OrderItems ON
Orders.OrderNo = OrderItems.OrderNo

-- This SELECT statement outputs all rows from
-- the Customers table where the customer has
-- not placed an order within the last year

SELECT *
FROM Customers
WHERE NOT EXISTS
  (SELECT *
   FROM Orders
   WHERE CustNo=Customers.CustNo AND
   OrderDate BETWEEN (CURRENT_DATE - INTERVAL '1' YEAR) AND CURRENT_DATE)

-- This SELECT statement outputs all customers
-- and their total orders for the last year in
-- descending order by the TotalOrdersAmount
-- SELECT column expression

SELECT Customer.CustNo,
Customer.Name,
COUNT(Orders.*) AS TotalOrders,
SUM(OrderItems.QtyOrdered * OrderItems.UnitPrice) AS TotalOrdersAmount
FROM Customers INNER JOIN Orders ON
Customer.CustNo = Orders.CustNo
INNER JOIN OrderItems ON
Orders.OrderNo = OrderItems.OrderNo
WHERE Orders.OrderDate BETWEEN (CURRENT_DATE - INTERVAL '1' YEAR) AND
      CURRENT_DATE)
GROUP BY Customer.CustNo, Customer.Name
ORDER BY TotalOrdersAmount DESC

-- This SELECT statement selects the total orders
-- from the Orders table for all rows where the
-- OrderDate is in January and uses the UNION
-- clause to append the total orders from the Orders
-- table where the OrderDate is in February

SELECT 'January' AS OrderMonth,
SUM(OrderItems.QtyOrdered * OrderItems.UnitPrice) AS TotalOrdersAmount
FROM Orders INNER JOIN OrderItems ON
Orders.OrderNo = OrderItems.OrderNo
WHERE Orders.OrderDate BETWEEN DATE '2006-01-01' AND DATE '2006-01-31'
```

```
UNION ALL
SELECT 'February' AS OrderMonth,
SUM(OrderItems.QtyOrdered * OrderItems.UnitPrice) AS TotalOrdersAmount
FROM Orders INNER JOIN OrderItems ON
Orders.OrderNo = OrderItems.OrderNo
WHERE Orders.OrderDate BETWEEN DATE '2006-02-01' AND DATE '2006-02-28'

-- This SELECT statement returns the
-- user-defined version for a given
-- table, or NULL if the table does
-- not exist.  It uses the INTO clause
-- to put the resultant value into an
-- output parameter.

SELECT Version INTO :Version
FROM Information.Tables WHERE Name=:Name
```

Required Privileges

The current user must be granted the SELECT privilege on all tables referenced in the FROM clause in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Recursive Queries	ElevateDB does not support recursive queries.
Sampling	ElevateDB does not support sampling in SELECT statements.
FULL OUTER JOINS	ElevateDB does not support FULL OUTER JOINs.
NATURAL JOINS	ElevateDB does not support NATURAL JOINs.
USING	ElevateDB does not support the USING clause in joins.
CUBE and ROLLUP	ElevateDB does not support the CUBE and ROLLUP clauses in the GROUP BY clause.
GROUPING SETS	ElevateDB does not support the GROUPING SETS clause in the GROUP BY clause.
GROUP BY DISTINCT	ElevateDB does not support the DISTINCT clause in the GROUP BY clause.
WINDOW	ElevateDB does not support the WINDOW clause.
CORRESPONDING BY	ElevateDB does not support the CORRESPONDING BY clause in the UNION, INTERSECT, and EXCEPT set operators.
RANGE	The RANGE clause is an ElevateDB extension.

6.3 INSERT

Inserts one or more rows into a table.

Syntax

```
INSERT INTO <TableName>
[(<ColumnName> [,<ColumnName>])]
VALUES (<Value> [,<Value>])|
<QueryExpression>

<QueryExpression> = query with the same number of
columns as the INSERT statement and columns that
are type-compatible with the INSERT columns
```

Usage

Use this statement to insert a row or rows into a table. If a list of columns to populate is not specified, then the number of values specified in the VALUES clause must match the number of columns in the table. All values specified in the VALUES clause must be type-compatible with the specified columns, or all of the columns in the table if the columns are not specified. If a query expression is used to insert multiple rows into a table, then the SELECT column list of the query expression must contain columns or expressions that are type-compatible with the specified columns, or all of the or all of the columns in the table if the columns are not specified.

> **Note**
> If a list of columns is specified, then any columns not specified will be populated with the default value defined for the column.

Examples

```
-- This INSERT statement inserts a new
-- row into the Orders table

INSERT INTO Orders
(OrderNo, ItemNo, QtyOrdered, UnitPrice)
VALUES (1200, 23478, 10, 30.00)

-- This INSERT statement inserts all of
-- the Orders rows for the year 2006 into
-- the ArchivedOrders table

INSERT INTO ArchivedOrders
SELECT * FROM Orders
WHERE OrderDate BETWEEN DATE '2006-01-01' AND DATE '2006-12-31'
```

Required Privileges

The current user must be granted the INSERT and SELECT privileges on the target table. In addition, the current user must be granted the SELECT privilege on any tables referenced in the FROM clause of a query expression, if one is used. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
DEFAULT VALUES	ElevateDB does not support the DEFAULT VALUES clause.

6.4 UPDATE

Updates one or more rows in a table.

Syntax

```
UPDATE <TableName>
SET <ColumnName> = <Value> [,<ColumnName> = <Value>])
[WHERE <FilterCondition>]
```

Usage

Use this statement to update one or more rows in a table. The SET clause is used to specify which columns you want to update and the values to assign to the columns. Each value can be any valid SQL expression.

WHERE Clause

Use the WHERE clause to limit the rows that are updated to those that satisfy a boolean SQL expression.

Examples

```
-- This UPDATE statement updates
-- the customer with the customer # of
-- 8354 and sets their LastOrdered column
-- to today

UPDATE Customers
SET LastOrdered = CURRENT_DATE
WHERE CustNo = 8354

-- This UPDATE statement updates all of
-- the rows in the Customers table and sets
-- their SentMailer column to False

UPDATE Customers
SET SentMailer = FALSE
```

Required Privileges

The current user must be granted the UPDATE and SELECT privileges on the target table. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Positioned Updates	ElevateDB does not support positioned updates. Instead ElevateDB supports using a cursor-based UPDATE statement directly on the current position of a cursor.

6.5 DELETE

Deletes one or more rows from a table.

Syntax

```
DELETE FROM <TableName>
[WHERE <FilterCondition>]
```

Usage

Use this statement to delete one or more rows from a table.

WHERE Clause

Use the WHERE clause to limit the rows that are deleted to those that satisfy a boolean SQL expression.

Examples

```
-- This DELETE statement deletes
-- all rows from the Orders table for the
-- year 2006

DELETE FROM Orders
WHERE OrderDate BETWEEN DATE '2006-01-01' AND DATE '2006-12-31'
```

Required Privileges

The current user must be granted the DELETE and SELECT privileges on the target table. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Positioned Deletes	ElevateDB does not support positioned deletes. Instead ElevateDB supports using a cursor-based DELETE statement directly on the current position of a cursor.

Chapter 7
SQL/PSM Statements

7.1 Introduction

SQL/PSM (persistent stored module) statements are used to define stored functions and procedures that are stored in an ElevateDB database and can be called from both client code and other stored functions and procedures. This section of the manual details the available SQL/PSM statements in ElevateDB.

Notation

The notation used in the syntax section for each SQL/PSM statement is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

7.2 BEGIN..END

Declares a block of statements.

Syntax

```
[Label:]
BEGIN
   [<Statement>;]
   [<Statement>;]
END [Label];
```

Usage

Use these statements to declare a block of statements for execution in a procedure or function.

> **Note**
> The outermost BEGIN..END block for any procedure or function does not require a line termination character (;) after the END, where any BEGIN..END block within the outermost block does require a line termination character after the END.

Examples

```
-- This procedure produces a summary
-- of the number of albums and total album
-- purchases by genre, label, or artist

CREATE PROCEDURE Summaries(IN "SummaryType" CHAR(1) COLLATE ANSI_CI)
BEGIN
    DECLARE Result CURSOR WITH RETURN FOR Stmt;

    CASE SummaryType
    -- Genres summary
    WHEN 'G' THEN
       BEGIN
       PREPARE Stmt FROM 'SELECT Genre AS Name, COUNT(Name) AS NumAlbums,
                          SUM(PurchasePrice) AS TotalPurchases
                          FROM Albums
                          GROUP BY Genre';
       OPEN Result;
       END;
    -- Labels summary
    WHEN 'L' THEN
       BEGIN
       PREPARE Stmt FROM 'SELECT Label AS Name, COUNT(Name) AS NumAlbums,
                          SUM(PurchasePrice) AS TotalPurchases
                          FROM Albums
                          GROUP BY Label';
       OPEN Result;
       END;
```

```
     -- Artists summary
     WHEN 'A' THEN
        BEGIN
        PREPARE Stmt FROM 'SELECT Artist AS Name, COUNT(Name) AS NumAlbums,
                           SUM(PurchasePrice) AS TotalPurchases
                           FROM Albums
                           GROUP BY Artist';
        OPEN Result;
        END;
     END CASE;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.3 EXCEPTION

Declares a block of statements for handling an exception.

Syntax

```
[Label:]
BEGIN
   [<Statement>;]
   [<Statement>;]
EXCEPTION
   [<Statement>;]
END [Label];
```

Usage

Use these statements to declare a block of statements for execution in a procedure or function with an associated exception block of statements for handling any exceptions that may occur in the block of statements.

You can use the ERRORCODE and ERRORMSG functions in an exception handling block to determine the current error code and message.

Examples

```
-- This procedure uses an EXCEPTION
-- block to handle any exceptions while
-- executing a CREATE TABLE statement

CREATE PROCEDURE CreateTestTable()
BEGIN
   DECLARE stmt STATEMENT;

   PREPARE stmt FROM 'CREATE TEMPORARY TABLE "TestTable"
                      (
                      "FirstColumn" INTEGER,
                      "SecondColumn" VARCHAR(30),
                      "ThirdColumn" CLOB,
                      PRIMARY KEY ("FirstColumn")
                      )

                      DESCRIPTION ''Test Table''';

   EXECUTE stmt;
EXCEPTION
   IF ERRORCODE()=700 THEN
      RAISE ERROR CODE 10000 MESSAGE 'Syntax error';
   ELSE
      RAISE ERROR CODE 10000 MESSAGE 'Unexpected error - ' +
           ERRORMSG();
   END IF;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.4 FINALLY

Declares a block of statements for always executing a block, regardless of whether an exception occurs or not.

Syntax

```
[Label:]
BEGIN
   [<Statement>;]
   [<Statement>;]
FINALLY
   [<Statement>;]
END [Label];
```

Usage

Use these statements to declare a block of statements for execution in a procedure or function with an associated block of statements that will be executed regardless of any exceptions that are raised, or whether the block of statements was exited using the LEAVE statement.

FINALLY blocks are useful for ensuring that any resources that are allocated before the block is executed, are released after the block is executed. For example, if an external function/procedure is called that opens a file on disk, you would want to use a FINALLY block to ensure that another external function/procedure is called to close the file.

Examples

```
-- This procedure uses a FINALLY
-- block to make sure that the file opened
-- using the OpenFile() external function
-- is closed using the CloseFile() external
-- function

CREATE FUNCTION ReadTextFile(IN TextFileName VARCHAR)
RETURNS VARCHAR
BEGIN
   DECLARE FileHandle INTEGER DEFAULT 0;
   DECLARE Result VARCHAR DEFAULT '';

   SET FileHandle=OpenFile(TextFileName);
   BEGIN
      SET Result=ReadFile(FileHandle);
   FINALLY
      CloseFile(FileHandle);
   END;

END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.5 DECLARE

Declares one or more variables, cursors, or statements.

Syntax

```
DECLARE <VariableDefinition>|<CursorDefinition>|
        <StatementDefinition>

<VariableDefinition> =

<VariableName> [,<VariableName>] <DataType>
[ARRAY [<MaximumCardinality>]]
[DEFAULT <DefaultExpression>]

<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<CursorDefinition> =

<CursorName> [SENSITIVE|INSENSITIVE|ASENSITIVE] CURSOR
[WITH RETURN|WITHOUT RETURN] FOR <StatementName>

<StatementDefinition> =

<StatementName> STATEMENT
```

Usage

Use this statement to declare one or more variables, cursors, or statements to be used later in the function or procedure.

Variables

Variables can be declared as any valid data type, and the DEFAULT clause can be used to specify an initial value for the variable. To declare an array, use the ARRAY clause along with the maximum cardinality specifier after the data type. The maximum cardinality sets the limit on the size of the array, and any attempt to reference any index (1-based) greater than the maximum cardinality of the array will result in an exception.

> **Note**
> If you specify a default value for an array using the DEFAULT clause, then every single element in the array will be initialized to the specified default value.

Cursors

Cursors can be declared as sensitive, insensitive, or asensitive.

Type	Description
SENSITIVE	Sensitive cursors are dynamic and change along with the table used to output the rows in the result set on which the cursor is operating.
INSENSITIVE	Insensitive cursors are static and do not change even if the tables used to output the rows in the result set on which the cursor is operating change.
ASENSITIVE	Asensitive cursors, the default, are esentially the same as a sensitive cursor because ElevateDB will always attempt to open a sensitive cursor if the cursor is declared as asensitive.

It is important to recognize that the cursor type in a cursor declaration is simply a request for a certain type of cursor, except in the case of the ASENSITIVE cursor declaration which is equivalent to declaring that the type of cursor is irrelevant. ElevateDB may or may not be able to create a declared cursor type. To determine the actual type of cursor that was created, use the SENSITIVE function on any opened cursor. See the Result Set Cursor Sensitivity topic for more information on what rules determine whether a cursor can be sensitive or not.

Cursor declarations also may specify whether the cursor should be returned to the calling program. Returnability only applies to procedures and does not apply to functions.

Type	Description
WITH RETURN	Specifies that the cursor should be returned to the calling program if it is left open in the procedure.
WITHOUT RETURN	Specifies that the cursor should be automatically closed if it is left open when the function or procedure exits. This is the default.

The statement name given in a cursor associates a statement (see below) with the cursor for use in the preparation of the SQL SELECT statement used to output the result set on which the cursor will be operating.

Statements

Statements are simply containers for executing dynamic SQL statements in a procedure or function. To actually use a statement you must first bind SQL statement text to the statement using the PREPARE statement. Then the statement may be executed any number of times using the EXECUTE statement.

Examples

```
-- This procedure changes all
-- rows with a State column value of 'FL'
-- to 'NY'and returns a sensitive cursor
-- on the Customers table

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor ('State') INTO State;

   WHILE NOT EOF(CustCursor) DO
      IF (State='FL') THEN
         UPDATE CustCursor SET 'State'='NY';
      END IF;
      FETCH NEXT FROM CustCursor ('State') INTO State;
   END WHILE;
END

-- This procedure simply returns an insensitive
-- cursor on the States table

CREATE PROCEDURE States()
BEGIN
   DECLARE Test INSENSITIVE CURSOR WITH RETURN FOR stmt;

   PREPARE stmt FROM 'SELECT * FROM States';

   OPEN Test;
END

-- This procedure uses a statement to
-- execute a CREATE TABLE statement

CREATE PROCEDURE CreateTestTable()
BEGIN
   DECLARE stmt STATEMENT;

   PREPARE stmt FROM 'CREATE TEMPORARY TABLE "TestTable"
                     (
                     "FirstColumn" INTEGER,
                     "SecondColumn" VARCHAR(30),
                     "ThirdColumn" CLOB,
                     PRIMARY KEY ("FirstColumn")
                     )
```

```
                      DESCRIPTION ''Test Table''';

   EXECUTE stmt;
END

-- This script loops through the Customer table and
-- populates an array with the CustNo column value
-- for each row

SCRIPT
BEGIN
   DECLARE Done BOOLEAN DEFAULT False;
   DECLARE TotalRows INTEGER DEFAULT 0;
   DECLARE CustCursor CURSOR FOR CustStmt;
   DECLARE CustArray INTEGER ARRAY [56];

   PREPARE CustStmt FROM 'SELECT CustNo,
                          Company
                          FROM Customer';

   OPEN CustCursor;

   WHILE (NOT EOF(CustCursor)) DO
      SET TotalRows=TotalRows+1;
      FETCH NEXT FROM CustCursor INTO CustArray[TotalRows];
      SET PROGRESS TO TRUNC((TotalRows/ROWCOUNT(CustCursor))*100);
   END WHILE;

   CLOSE CustCursor;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Dynamic SQL	The use of dynamic SQL for cursor declarations and statement declarations instead of static SQL in procedures and functions is an ElevateDB extension.
BEGIN..END	Declarations can only be made at the beginning of the outermost BEGIN..END block in an ElevateDB procedure or function. The standard dictates that declarations can be made anywhere inside of any BEGIN..END block.

7.6 RAISE

Re-raises an exception or creates a new user-defined exception.

Syntax

```
RAISE [ERROR CODE <ErrorCode> MESSAGE <ErrorMessage>]

<ErrorCode> = Any user-defined (10000-High(INTEGER)) error code
```

Usage

Use this statement to re-raise an existing exception or raise a new exception using the user-defined error code range of 10000 or higher.

This statement can only be used from within an EXCEPTION block or from within an error trigger when it is specified without an error code and message and is simply trying to re-raise an existing exception. See the CREATE TRIGGER topic for more information on error triggers.

Examples

```
-- This procedure uses an exception
-- block to handle any exceptions while
-- executing a CREATE TABLE statement

PROCEDURE CreateTestTable()
BEGIN
   DECLARE stmt STATEMENT;

   PREPARE stmt FROM 'CREATE TEMPORARY TABLE "TestTable"
                     (
                     "FirstColumn" INTEGER,
                     "SecondColumn" VARCHAR(30),
                     "ThirdColumn" CLOB,
                     PRIMARY KEY ("FirstColumn")
                     )

                     DESCRIPTION ''Test Table''';

   EXECUTE stmt;
EXCEPTION
   IF ERRORCODE()=700 THEN
      RAISE ERROR CODE 10000 MESSAGE 'Syntax error';
   ELSE
      RAISE ERROR CODE 10000 MESSAGE 'Unexpected error - ' +
           ERRORMSG();
   END IF;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.7 IF

Provides conditional branching.

Syntax

```
IF <BooleanExpression> THEN
   <StatementBlock>
[ELSEIF <BooleanExpression> THEN
   <StatementBlock>]
[ELSE
   <StatementBlock>]
END IF;

<StatementBlock> =

[[Label:]
BEGIN]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
[END [Label];]
```

Usage

Use this statement to provide conditional branching based upon a single or multiple boolean expressions.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor ('State') INTO State;

   WHILE NOT EOF(CustCursor) DO
      IF (State='FL') THEN
         UPDATE CustCursor SET 'State'='NY';
      END IF;
      FETCH NEXT FROM CustCursor ('State') INTO State;
   END WHILE;
```

```
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.8 CASE

Provides conditional branching.

Syntax

```
CASE
WHEN <BooleanExpression> THEN
   <StatementBlock>
[WHEN <BooleanExpression> THEN
   <StatementBlock>]
[ELSE
   <StatementBlock>]
END CASE;

Shorthand Value Syntax

CASE <Expression>
WHEN <Expression> THEN
   <StatementBlock>
[WHEN <Expression> THEN
   <StatementBlock>]
[ELSE
   <StatementBlock>]
END CASE;

<StatementBlock> =

[[Label:]
BEGIN]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
[END [Label];]
```

Usage

Use this statement to provide conditional branching based upon a single or multiple boolean expressions, or based upon multiple simple value comparisons.

Examples

```
-- This procedure produces a summary
-- of the number of albums and total album
-- purchases by genre, label, or artist

CREATE PROCEDURE Summaries(IN "SummaryType" CHAR(1) COLLATE ANSI_CI)
BEGIN
    DECLARE Result CURSOR WITH RETURN FOR Stmt;

    CASE SummaryType
```

```
    -- Genres summary
    WHEN 'G' THEN
       BEGIN
       PREPARE Stmt FROM 'SELECT Genre AS Name, COUNT(Name) AS NumAlbums,
                          SUM(PurchasePrice) AS TotalPurchases
                          FROM Albums
                          GROUP BY Genre';
       OPEN Result;
       END;
    -- Labels summary
    WHEN 'L' THEN
       BEGIN
       PREPARE Stmt FROM 'SELECT Label AS Name, COUNT(Name) AS NumAlbums,
                          SUM(PurchasePrice) AS TotalPurchases
                          FROM Albums
                          GROUP BY Label';
       OPEN Result;
       END;
    -- Artists summary
    WHEN 'A' THEN
       BEGIN
       PREPARE Stmt FROM 'SELECT Artist AS Name, COUNT(Name) AS NumAlbums,
                          SUM(PurchasePrice) AS TotalPurchases
                          FROM Albums
                          GROUP BY Artist';
       OPEN Result;
       END;
    END CASE;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.9 LOOP

Provides an unconditional looping construct.

Syntax

```
[Label:]
LOOP
   <StatementBlock>
END LOOP [Label];

<StatementBlock> =

<StatementBlock> =

[[Label:]
BEGIN]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
[END [Label];]
```

Usage

Use this statement to loop unconditionally on a single statement or multiple statements. You can use the LEAVE statement to exit the loop at any time, and the ITERATE statement to jump to the top of the loop at any time.

Examples

```
-- This procedure loops through
-- the Customers table and exits the loop
-- when the EOF is reached on the cursor

CREATE PROCEDURE LoopCustomers()
BEGIN
   DECLARE CustCursor SENSITIVE CURSOR FOR Stmt;

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor;

   LOOP
      IF EOF(CustCursor) THEN
         LEAVE;
      ELSE
         FETCH NEXT FROM CustCursor;
      END IF;
   END LOOP;
```

```
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.10 REPEAT

Conditionally repeats a single statement or multiple statements.

Syntax

```
[Label:]
REPEAT
   <StatementBlock>
UNTIL <BooleanExpression> END REPEAT [Label];

<StatementBlock> =

<StatementBlock> =

[[Label:]
BEGIN]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
[END [Label];]
```

Usage

Use this statement to repeat a single statement or multiple statements until a boolean expression evaluates to True. You can use the LEAVE statement to exit the loop at any time, and the ITERATE statement to jump to the top of the loop at any time.

Examples

```
-- This function simply repeats a
-- a string the specified number of times
-- and returns the resultant string

CREATE FUNCTION RepeatString(IN "StringToRepeat" VARCHAR, IN "RepeatCount"
      INTEGER)
RETURNS VARCHAR
BEGIN
   DECLARE I INTEGER DEFAULT 1;
   DECLARE Result VARCHAR DEFAULT '';

   IF RepeatCount = 0 THEN
      LEAVE;
   END IF;

   REPEAT
      SET Result = (Result + StringToRepeat);
      SET I = (I + 1);
   UNTIL (I > RepeatCount) END REPEAT;

   RETURN Result;
```

```
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.11 WHILE

Provides an conditional looping construct.

Syntax

```
[Label:]
WHILE <BooleanExpression> DO
   <StatementBlock>
END WHILE [Label];

<StatementBlock> =

<StatementBlock> =

[[Label:]
BEGIN]
   [<Statement>;]
   [<Statement>;]
[EXCEPTION]
   [<Statement>;]
[END [Label];]
```

Usage

Use this statement to conditionally loop on a single statement or multiple statements. You can use the LEAVE statement to exit the loop at any time, and the ITERATE statement to jump to the top of the loop at any time.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor ('State') INTO State;

   WHILE NOT EOF(CustCursor) DO
      IF (State='FL') THEN
         UPDATE CustCursor SET 'State'='NY';
      END IF;
      FETCH NEXT FROM CustCursor ('State') INTO State;
```

```
    END WHILE;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.12 ITERATE

Jumps to the top of a loop.

Syntax

```
ITERATE [<Label>]
```

Usage

Use this statement to jump to the top of the current loop or a specified loop using the name of a labeled loop. This statement is only valid within a LOOP, REPEAT, or WHILE loop.

Examples

```
-- This procedure deletes all rows
-- in the Customers table with a State
-- column of 'FL'

CREATE PROCEDURE DeleteFLCustomers()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   FETCH FIRST FROM CustCursor ('State') INTO State;

   WHILE NOT EOF(CustCursor) DO
      IF (State='FL') THEN
         DELETE FROM CustCursor;
         FETCH FROM CustCursor ('State') INTO State;
         ITERATE;
      END IF;
      FETCH NEXT FROM CustCursor ('State') INTO State;
   END WHILE;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Optional Label	The label is optional in ElevateDB, and if not provided, defaults to the current loop being executed.

7.13 LEAVE

Exits a block of statements.

Syntax

```
LEAVE [<Label>]
```

Usage

Use this statement to exit a block of statements contained within a BEGIN..END block or LOOP, REPEAT, or WHILE loop.

Examples

```
-- This function simply repeats a
-- a string the specified number of times
-- and returns the resultant string

CREATE FUNCTION RepeatString(IN "StringToRepeat" VARCHAR, IN "RepeatCount"
      INTEGER)
RETURNS VARCHAR
BEGIN
   DECLARE I INTEGER DEFAULT 1;
   DECLARE Result VARCHAR DEFAULT '';

   IF RepeatCount = 0 THEN
      LEAVE;
   END IF;

   REPEAT
      SET Result = (Result + StringToRepeat);
      SET I = (I + 1);
   UNTIL (I > RepeatCount) END REPEAT;

   RETURN Result;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Optional Label	The label is optional in ElevateDB, and if not provided, defaults to the current block being executed.

7.14 SET

Assigns a value to a variable, parameter, or trigger NEWROW value.

Syntax

```
SET <TargetName> = <Expression> [,<TargetName> = <Expression>]

<TargetName> =

<VariableName>|<ParameterName>|NEWROW.<ColumnName>
```

Usage

Use this statement to assign a value to a variable, parameter, or trigger NEWROW value.

Examples

```
-- This function uses the SET
-- statement to assign the count of the
-- rows in the Customers table to the
-- result variable returned from the function

CREATE FUNCTION CountCustomers()
RETURNS INTEGER
BEGIN
   DECLARE Test SENSITIVE CURSOR FOR stmt;
   DECLARE Result INTEGER DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM Customers';

   OPEN Test;

   FETCH FIRST FROM Test;

   WHILE NOT EOF(Test) DO
      SET Result = Result + 1;
      FETCH NEXT FROM Test;
   END WHILE;

   RETURN Result;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.15 CALL

Calls a procedure.

Syntax

```
CALL <FunctionName>|<ProcedureName>([<Value>[,<Value>]])
```

Usage

Use this statement to call a function/procedure, passing parameter values if the function/procedure being called has been defined with parameters.

> **Note**
> Any return values from functions are ignored when using the CALL statement.

Examples

```
-- This trigger calls the external
-- SendMail procedure with which group to
-- send the email to along with the new
-- value of the Notes column for the customer
-- being updated

CREATE TRIGGER "NotesUpdate" AFTER UPDATE OF "Notes"
ON "Customer"
BEGIN
   CALL SendEmail('CustomerReps',NEWROW.Notes);
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.16 USE

Opens a new database or closes the existing database.

Syntax

```
USE [<DatabaseName>]
```

Usage

Use this statement in a script or job definition to open a new database or close a database that was previously opened. To open a new database, specify the database name after the USE keyword. Opening a new database automatically closes the current database that is open. To close the current database and not open a new database, leave the database name blank. This reverts the current database back to the default database, which is always the system-defined Configuration database for jobs.

Examples

```
-- This job optimizes the Inventory
-- table in the ShopFloor database.

CREATE JOB OptimizeInventory
RUN AS "System"
FROM DATE '2000-01-01' TO DATE '2030-12-31'
WEEKLY
ON SUN
BETWEEN TIME '03:00' AND TIME '04:00'
CATEGORY ''
BEGIN
   USE ShopFloor;
   EXECUTE IMMEDIATE 'OPTIMIZE TABLE Inventory';
   USE;
END

-- Here's the same thing as a script instead

SCRIPT
BEGIN
   USE ShopFloor;
   EXECUTE IMMEDIATE 'OPTIMIZE TABLE Inventory';
   USE;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Extension	This SQL statement is an ElevateDB extension.

7.17 EXECUTE IMMEDIATE

Executes an SQL statement.

Syntax

```
EXECUTE IMMEDIATE <SQLStatement>
[USING <Value> [,<Value>]]
```

Usage

Use this statement to execute the specified DDL, DML, or administrative SQL statement. If the SQL statement is parameterized, then you can use the USING clause to specify the values to use for the parameters. The values are in left-to-right order, corresponding to how the parameters were declared in the SQL statement.

Examples

```
-- This procedure executes a
-- CREATE TABLE statement to create a
-- temporary table

CREATE PROCEDURE CreateTestTable()
BEGIN
   EXECUTE IMMEDIATE 'CREATE TEMPORARY TABLE "TestTable"
                     (
                     "FirstColumn" INTEGER,
                     "SecondColumn" VARCHAR(30),
                     "ThirdColumn" CLOB,
                     PRIMARY KEY ("FirstColumn")
                     )

                     DESCRIPTION ''Test Table''';
END

-- This function returns the user-defined
-- version for a given table, or NULL if
-- the table does not exist

FUNCTION "GetTableVersion" (IN TableName VARCHAR)
RETURNS DECIMAL(19,2)
BEGIN
   DECLARE Version DECIMAL(19,2) DEFAULT 0;
   EXECUTE IMMEDIATE 'SELECT Version INTO ?
        FROM Information.Tables WHERE Name=?'
        USING Version,TableName;
   RETURN Version;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Dynamic SQL	The use of dynamic SQL for DDL, DML, and administrative statement execution instead of static SQL in procedures and functions is both an ElevateDB extension and a deviation from the standard.

7.18 PREPARE

Prepares an SQL statement and binds it to a statement variable.

Syntax

```
PREPARE <StatementName> FROM <SQLStatement>
```

Usage

Use this statement to prepare a DDL, DML, or administrative SQL statement and bind it to a statement variable. The statement variable must have been previously declared in a DECLARE statement.

Using PREPARE pre-compiles the SQL statement so that it may be executed multiple times using the EXECUTE statement. This is especially useful with parameterized SQL statements.

For a SELECT statement, the PREPARE statement can be used to bind the statement to a cursor so that the cursor may then be opened using the OPEN statement.

Examples

```
-- This function looks up the sales tax
-- rate for a given state and county

CREATE FUNCTION LookupSalesTaxRate(IN State CHAR(2), IN County VARCHAR)
RETURNS DECIMAL(19,2)
BEGIN
   DECLARE TempCursor CURSOR FOR stmt;
   DECLARE Result DECIMAL(19,2) DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM SalesTaxes WHERE State = ? AND County =
      ?';

   OPEN TempCursor USING State, County;

   IF (ROWCOUNT(TempCursor) > 0) THEN
      FETCH FIRST FROM TempCursor ('TaxRate') INTO Result;
   END IF;

   CLOSE TempCursor;

   RETURN Result;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Dynamic SQL	The use of dynamic SQL for DDL, DML, and administrative statement execution instead of static SQL in procedures and functions is both an ElevateDB extension and a deviation from the standard.

7.19 UNPREPARE

Un-prepares an SQL statement, releasing all associated resources.

Syntax

```
UNPREPARE <StatementName>
```

Usage

Use this statement to un-prepare a DDL, DML, or administrative SQL statement and release all resources associated with the statement, including any compiled symbols, opened tables, or result set cursors. The statement variable must have been previously declared in a DECLARE statement.

For a SELECT statement, the UNPREPARE statement will cause any cursor created using the OPEN to be released, making it inaccessible until the OPEN statement is used again to create a new cursor.

Examples

```
-- This function looks up the sales tax
-- rate for a given state and county

CREATE FUNCTION LookupSalesTaxRate(IN State CHAR(2), IN County VARCHAR)
RETURNS DECIMAL(19,2)
BEGIN
   DECLARE TempCursor CURSOR FOR stmt;
   DECLARE Result DECIMAL(19,2) DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM SalesTaxes WHERE State = ? AND County =
      ?';

   OPEN TempCursor USING State, County;

   IF (ROWCOUNT(TempCursor) > 0) THEN
      FETCH FIRST FROM TempCursor ('TaxRate') INTO Result;
   END IF;

   CLOSE TempCursor;
   UNPREPARE stmt;

   RETURN Result;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Dynamic SQL	The use of dynamic SQL for DDL, DML, and administrative statement execution instead of static SQL in procedures and functions is both an ElevateDB extension and a deviation from the standard.

7.20 EXECUTE

Executes a previously-prepared SQL statement.

Syntax

```
EXECUTE <StatementName> [USING <Value> [,<Value>]]
```

Usage

Use this statement to execute a previously-prepared DDL, DML, or administrative SQL statement. SQL statements are prepared using the PREPARE statement. If the prepared SQL statement is parameterized, then you can use the USING clause to specify the values to use for the parameters. The values are in left-to-right order, corresponding to how the parameters were declared in the SQL statement.

After executing an SQL statement, you can use the ROWSAFFECTED function to determine the number of rows affected by the statement.

Examples

```
-- This procedure creates 100,000 test
-- rows in the Customers table using a
-- parameterized INSERT statement

CREATE PROCEDURE PopulateCustomers()
BEGIN
   DECLARE stmt STATEMENT;
   DECLARE I INTEGER DEFAULT 1;

   PREPARE stmt FROM 'INSERT INTO Customers
                      (CustNo, Name)
                      VALUES (?, ?)';

   WHILE I <= 10000 DO
      EXECUTE stmt USING I, 'Test Customer #' + CAST(I AS VARCHAR(10));
      SET I = I + 1;
   END WHILE;

   UNPREPARE stmt;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Dynamic SQL	The use of dynamic SQL for DDL, DML, and administrative statement execution instead of static SQL in procedures and functions is both an ElevateDB extension and a deviation from the standard.

7.21 OPEN

Opens a result set cursor.

Syntax

```
OPEN <CursorName> [USING <Value> [,<Value>]]
```

Usage

Use this statement to open a result set cursor on a previously-prepared SELECT statement. If the statement is parameterized, then you can use the USING clause to specify the values to use for the parameters in left-to-right order corresponding to how they were declared in the SELECT statement.

After opening a cursor, you can use the ROWCOUNT function to determine the number of rows in the cursor and the SENSITIVE function to determine if the cursor is a sensitive or asensitive cursor. See the Result Set Cursor Sensitivity topic for more information on cursor sensitivity.

Examples

```
-- This procedure looks up the sales tax
-- rate for a given state and county

CREATE FUNCTION LookupSalesTaxRate(IN State CHAR(2), IN County VARCHAR)
RETURNS DECIMAL
BEGIN
   DECLARE TempCursor CURSOR FOR stmt;
   DECLARE Result DECIMAL DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM SalesTaxes WHERE State = ? AND County =
      ?';

   OPEN TempCursor USING State, County;

   IF (ROWCOUNT(TempCursor) > 0) THEN
      FETCH FIRST FROM TempCursor ('TaxRate') INTO Result;
   END IF;

   CLOSE TempCursor;

   RETURN Result;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Dynamic SQL	The use of dynamic SQL for DDL, DML, and administrative statement execution instead of static SQL in procedures and functions is both an ElevateDB extension and a deviation from the standard.

7.22 CLOSE

Closes a result set cursor.

Syntax

```
CLOSE <CursorName>
```

Usage

Use this statement to close a previously-opened result set cursor. If the cursor specified is not open, then this statement does nothing.

Examples

```
-- This procedure looks up the sales tax
-- rate for a given state and county

CREATE FUNCTION LookupSalesTaxRate(IN State CHAR(2), IN County VARCHAR)
RETURNS DECIMAL
BEGIN
   DECLARE TempCursor CURSOR FOR stmt;
   DECLARE Result DECIMAL DEFAULT 0;

   PREPARE stmt FROM 'SELECT * FROM SalesTaxes WHERE State = ? AND County =
      ?';

   OPEN TempCursor USING State, County;

   IF (ROWCOUNT(TempCursor) > 0) THEN
      FETCH FIRST FROM TempCursor ('TaxRate') INTO Result;
   END IF;

   CLOSE TempCursor;

   RETURN Result;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Dynamic SQL	The use of dynamic SQL for DDL, DML, and administrative statement execution instead of static SQL in procedures and functions is both an ElevateDB extension and a deviation from the standard.

7.23 FETCH

Navigates a result set cursor and reads column values into variables, parameters, or trigger NEWROW values.

Syntax

```
FETCH [<Orientation>] [FROM] <CursorName>
[[(<ColumnName> [,<ColumnName>])]
INTO <TargetName> [,<TargetName>]]

<Orientation> =

NEXT|PRIOR|FIRST|LAST|{ABSOLUTE|RELATIVE} <IntegerValue>

<TargetName> =

<VariableName>|<ParameterName>|NEWROW.<ColumnName>
```

Usage

Use this statement to navigate a result set cursor and read column values into variables, parameters, or NEWROW values in a trigger.

The various orientations work as follows:

Orientation	Description
NEXT	Navigates to the next row in the cursor. If there are no more subsequent rows in the cursor, then the EOF flag is set for the cursor and the current row stays the same. If there are no rows in the cursor, then both the BOF and EOF flags are set for the cursor.
PRIOR	Navigates to the prior row in the cursor. If there are no more prior rows in the cursor, then the BOF flag is set for the cursor and the current row stays the same. If there are no rows in the cursor, then both the BOF and EOF flags are set for the cursor.
FIRST	Navigates to the first row in the cursor. If there are no rows in the cursor, then both the BOF and EOF flags are set for the cursor.
LAST	Navigates to the last row in the cursor. If there are no rows in the cursor, then both the BOF and EOF flags are set for the cursor.
ABSOLUTE	Navigates to the Nth row specified. If the Nth row is greater than the number of rows in the cursor, then the EOF flag is set for the cursor. If there are no rows in the cursor, then both the BOF and EOF flags are set for the cursor.
RELATIVE	Navigates to the Nth row specified relative to the current row

	position. If the specified relative row is negative and the current row position minus the Nth row is less than 0, then the BOF flag is set for the cursor. If the current row position plus the Nth row is greater than the number of rows in the cursor, then the EOF flag is set for the cursor. If there are no rows in the cursor, then both the BOF and EOF flags are set for the cursor.

Note
If you do not specify a fetch orientation, then the default orientation is to fetch from the current row position in the cursor.

You can use the BOF and EOF functions to determine if the BOF flag or EOF flag has been set on a cursor.

Specify a list of columns to only read the column values from a specific set of columns. If a list of columns is not specified and the INTO keyword is specified, then it is assumed that all column values should be read.

Use the INTO keyword to list one or more variables, parameters, or trigger NEWROW values into which the column values should be read.

Examples

```
-- The following job backs up all tables in all databases
-- defined in the current system at 11:00 PM every evening.

CREATE JOB Backup
RUN AS "System"
FROM DATE '2006-01-01' TO DATE '2010-12-31'
DAILY
BETWEEN TIME '11:00 PM' AND TIME '11:30 PM'
CATEGORY 'Backup'
BEGIN
   DECLARE DBCursor CURSOR FOR DBStmt;
   DECLARE DBName VARCHAR DEFAULT '';

   PREPARE DBStmt FROM 'SELECT * FROM Databases';

   OPEN DBCursor;

   FETCH FIRST FROM DBCursor ('Name') INTO DBName;

   WHILE NOT EOF(DBCursor) DO
      IF (DBName <> 'Configuration') THEN
         EXECUTE IMMEDIATE 'BACKUP DATABASE "' + DBName + '" AS "' +
                           CAST(CURRENT_DATE AS VARCHAR(10)) +
                           '-' + DBName + '" TO STORE "Backups" INCLUDE
      CATALOG';
      END IF;
      FETCH NEXT FROM DBCursor ('Name') INTO DBName;
   END WHILE;

   CLOSE DBCursor;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Columns list	The list of columns to fetch from is an ElevateDB extension.
Exceptions	The SQL standard dictates that exceptions are raised whenever a fetch operation cannot be completed due to a BOF or EOF condition or a row not being found. ElevateDB does not raise an exception in any of these cases and instead uses the BOF and EOF functions to indicate these conditions.

7.24 START TRANSACTION

Starts a transaction.

Syntax

```
START TRANSACTION
[ON TABLES <TableName> [,<TableName>]]
[TIMEOUT <Timeout>]

<Timeout> = Milliseconds to wait for lock
```

Usage

Use this statement to start a transaction on the current database or a specific set of tables in the current database. Use the ON TABLES clause to specify a specific table or set of tables to start the transaction on.

> **Note**
> Using the ON TABLES clause will help concurrency by only locking the tables that will be involved in the transaction instead of all of the tables in the current database. See the Transactions topic for more information on transaction locking.

Use the TIMEOUT clause to specify the amount of time, in milliseconds, to wait for the transaction lock to succeed before raising a lock exception.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;
```

```
      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
ON TABLES	The ON TABLES clause is an ElevateDB extension.
TIMEOUT	The TIMEOUT clause is an ElevateDB extension.

7.25 COMMIT

Commits an transaction.

Syntax

```
COMMIT [WORK] [NO FLUSH]
```

Usage

Use this statement to commit an active transaction. The WORK keyword is optional. Use the NO FLUSH keyword to specify that the COMMIT should not force a flush to disk through the operating system. See the Transactions topic for more information on commit processing.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
NO FLUSH	The NO FLUSH clause is an ElevateDB extension.

7.26 ROLLBACK

Rolls back a transaction.

Syntax

```
ROLLBACK [WORK]
```

Usage

Use this statement to roll back an active transaction. The WORK keyword is optional. See the Transactions topic for more information on rolling back transactions.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

7.27 INSERT

Inserts a new row into a result set cursor.

Syntax

```
INSERT INTO <CursorName>
[(<ColumnName> [,<ColumnName>])]
VALUES (<Value> [,<Value>])
```

Usage

Use this statement to insert a new row into a result set cursor. If a list of columns to populate is not specified, then the number of values specified in the VALUES clause must match the number of columns in the result set that the cursor is using. All values specified in the VALUES clause must be type-compatible with the specified columns, or all of the columns in the result set if the columns are not specified.

Examples

```
-- This procedure checks to see if the
-- specified State exists in the States lookup
-- table and inserts it if it isn't

CREATE PROCEDURE LookupState(IN StateParam CHAR(2) COLLATE ANSI_CI)
BEGIN
   DECLARE StateCursor SENSITIVE CURSOR FOR Stmt;

   PREPARE Stmt FROM 'SELECT * FROM States WHERE State = ?';

   OPEN StateCursor USING StateParam;

   IF (ROWCOUNT(StateCursor) = 0) THEN
      INSERT INTO StateCursor ('State') VALUES (StateParam);
   END IF;

   CLOSE StateCursor;

END
```

Required Privileges

If the result set cursor is a sensitive cursor, then the current user must be granted the INSERT and SELECT privileges on the table in the SELECT statement used to output the result set that the cursor is using. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.28 UPDATE

Updates the current row in a result set cursor.

Syntax

```
UPDATE <CursorName>
SET <ColumnName> = <Value> [,<ColumnName> = <Value>])
```

Usage

Use this statement to update the current row in a result set cursor. The SET clause is used to specify which columns you want to update and the values to assign to the columns. Each value can be any valid SQL expression.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

Required Privileges

If the result set cursor is a sensitive cursor, then the current user must be granted the UPDATE and SELECT privileges on the table in the SELECT statement used to output the result set that the cursor is using. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.29 DELETE

Deletes the current row in a result set cursor.

Syntax

```
DELETE FROM <CursorName>
```

Usage

Use this statement to delete the current row in a result set cursor.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to delete the row

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE DeleteFLCustomers()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            DELETE FROM CustCursor;
            FETCH FROM CustCursor ('State') INTO State;
         ELSE
            FETCH NEXT FROM CustCursor ('State') INTO State;
         END IF;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

Required Privileges

If the result set cursor is a sensitive cursor, then the current user must be granted the DELETE and SELECT privileges on the table in the SELECT statement used to output the result set that the cursor is using. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.30 REFRESH

Refreshes a result set cursor.

Syntax

```
REFRESH <CursorName>
```

Usage

Use this statement to refresh an open result set cursor. Calling the REFRESH statement will cause any row changes made by other sessions to appear. This statement is the only way to refresh an insensitive result set cursor, since such a cursor does not automatically reflect changes made by other sessions. See the Result Set Cursor Sensitivity topic for more information on cursor sensitivity.

Note
This statement will only refresh the cursor itself. You must use the FETCH statement to re-fetch the updated data into whatever variables you have declared to hold the fetched column values.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.31 SET LOG MESSAGE

Creates a new log message.

Syntax

```
SET LOG MESSAGE TO <Message>
```

Usage

Use this statement to create a new log message that can be retrieved and viewed by the application that is executing the current routine. The message can be any text that you wish.

Examples

```
-- This procedure uses a SET LOG MESSAGE
-- statement to log which rows are updated

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE CustNo INTEGER DEFAULT 0;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
            FETCH FROM CustCursor ('CustNo') INTO CustNo;
            SET LOG MESSAGE TO 'Customer # '+CAST(CustNo AS VARCHAR)+'
      updated';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.32 SET PROGRESS

Creates a new progress update.

Syntax

```
SET PROGRESS TO <CompletionPercent>
```

Usage

Use this statement to create a progress update that can be displayed by the application that is executing the current routine. The completion percentage should be be any valid percentage between 1 and 100.

You can use the ABORTED SQL/PSM function to determine if the application indicated that it wished to abort the current execution in response to the progress update.

Examples

```
-- This procedure uses a SET PROGRESS
-- statement to display progress during its
-- execution and uses the ABORTED function
-- to abort the execution if the application
-- requests it

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';
   DECLARE TotalRows INTEGER DEFAULT 0;
   DECLARE NumRows INTEGER DEFAULT 0;

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;
   SET TotalRows=ROWCOUNT(CustCursor);

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE (NOT (EOF(CustCursor) OR ABORTED)) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
         SET NumRows=NumRows+1;
         SET PROGRESS TO TRUNC(((NumRows/TotalRows)*100));
      END WHILE;

      IF (NOT ABORTED) THEN
         COMMIT;
```

```
      ELSE
         ROLLBACK;
      END IF;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.33 SET STATUS MESSAGE

Creates a new status message.

Syntax

```
SET STATUS MESSAGE TO <Message>
```

Usage

Use this statement to create a new status message that can be retrieved and viewed by the application that is executing the current routine. The message can be any text that you wish.

Examples

```
-- This procedure uses a SET STATUS MESSAGE
-- statement to display which rows are updated

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE CustNo INTEGER DEFAULT 0;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
            FETCH FROM CustCursor ('CustNo') INTO CustNo;
            SET STATUS MESSAGE TO 'Customer # '+
                                CAST(CustNo AS VARCHAR)+' updated';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.34 ABORT

Sets the aborted flag for the current execution.

Syntax

```
ABORT
```

Usage

Use this statement to set the aborted flag for the current execution. This statement can be used in conjunction with the ABORTED function to perform conditional execution.

Starting in 2.05, you can use the ABORT statement to abort an INSERT, UPDATE, DELETE, or LOAD UPDATE operation from within a trigger. Please see the CREATE TRIGGER topic for more information.

Examples

```
-- This procedure uses an ABORT statement
-- to exit the WHILE loop once the number of
-- rows visited reaches 10

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';
   DECLARE TotalRows INTEGER DEFAULT 0;
   DECLARE NumRows INTEGER DEFAULT 0;

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;
   SET TotalRows=ROWCOUNT(CustCursor);

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE (NOT (EOF(CustCursor) OR ABORTED)) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
         SET NumRows=NumRows+1;
         SET PROGRESS TO TRUNC(((NumRows/TotalRows)*100));
         IF (NumRows=10) THEN
            ABORT;
         END WHILE;

      IF (NOT ABORTED) THEN
         COMMIT;
```

```
      ELSE
         ROLLBACK;
      END IF;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.35 RETRY

Retries an insert, update, or delete operation.

Syntax

```
RETRY
```

Usage

Use this statement to retry an insert, update, or delete operation from within the body of an error trigger. Error triggers are called whenever there is an error in an insert, update, or delete operation. If the trigger body has taken action to rectify the original exception that caused the error trigger to be called, then the RETRY statement can retry the operation. The body of the trigger is exited immediately at the point where the RETRY statement is specified, and any code after that point is skipped. Please see the CREATE TRIGGER statement for more information.

> **Warning**
> Retrying an operation without making sure that the operation isn't being recursively triggered can result in the trigger locking up the application. You should always make sure to check the conditions under which a RETRY is occurring.

Examples

```
-- This error trigger will handle a constraint
-- error, update the value so that the constraint error
-- doesn't happen.  However, if the value has already
-- been changed, then it re-raises the constraint error

CREATE TRIGGER "ErrorTest" ERROR INSERT ON "customer"
BEGIN
   IF ERRORCODE()=1004 THEN
      IF NEWROW.CustNo <> 100 THEN
         SET NEWROW.CustNo=100;
         RETRY;
      ELSE
         RAISE;
      END IF;
   END IF;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Extension	This SQL statement is an ElevateDB extension.

7.36 LOG EVENT

Logs an information, warning, or error event in the logged events for the current configuration.

Syntax

```
LOG [INFORMATION|WARNING|ERROR] EVENT <Message>
```

Usage

Use this statement to log an event in the logged events for the current configuration. The message can be any text that you wish.

Any events logged using this statement will have a function name of "Logged Event" in the logged events for the current configuration.

Note
Do not use this statement to log ElevateDB errors or exceptions. ElevateDB already logs such events automatically. The only exception is if a routine traps an exception, preventing it from escaping the routine in which the exception occurred. In such a case, you can use this statement to log the event.

Examples

```
-- This procedure uses a LOG ERROR EVENT
-- statement to log an event when an error occurs

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE CustNo INTEGER DEFAULT 0;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
            FETCH FROM CustCursor ('CustNo') INTO CustNo;
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;
```

```
      COMMIT;

   EXCEPTION
      LOG ERROR EVENT 'Unexpected error occurred during the UpdateState
      procedure: '+
                      CAST(ERRORCODE() AS VARCHAR)+' '+ERRORMSG();
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.37 SET STATEMENT CACHE

Sets the statement cache size within a job.

Syntax

```
SET STATEMENT CACHE TO <NumStatements>
```

Usage

Use this statement to set the statement cache size within a job. The default statement cache size in a job is 0, which results in no statement caching at all. Please see the Buffering and Caching topic for more information on the statement caching functionality in ElevateDB.

> **Note**
> This statement will override any existing statement cache size set for the session via client-specific connection parameters, so it is normally recommended that you only use this statement within the body of jobs, which use unique sessions per job execution. Please see the CREATE JOB statement for more information on how to create a job.

Examples

```
-- The following job backs up all tables in all databases
-- defined in the current system at 11:00 PM every evening.

CREATE JOB Backup
RUN AS "System"
FROM DATE '2006-01-01' TO DATE '2010-12-31'
DAILY
BETWEEN TIME '11:00 PM' AND TIME '11:30 PM'
CATEGORY 'Backup'
BEGIN
   DECLARE DBCursor CURSOR FOR DBStmt;
   DECLARE DBName VARCHAR DEFAULT '';

   -- 8 statements is more than we need, but isn't wasteful
   SET STATEMENT CACHE TO 8;

   PREPARE DBStmt FROM 'SELECT * FROM Databases';

   OPEN DBCursor;

   FETCH FIRST FROM DBCursor ('Name') INTO DBName;

   WHILE NOT EOF(DBCursor) DO
      IF (DBName <> 'Configuration') THEN
         EXECUTE IMMEDIATE 'BACKUP DATABASE "' + DBName + '" AS "' +
                           CAST(CURRENT_DATE AS VARCHAR(10)) +
                           '-' + DBName + '" TO STORE "Backups" INCLUDE
```

```
        CATALOG';
           -- This next statement is the one that we're interested in caching
           EXECUTE IMMEDIATE 'INSERT INTO BackupLog(ExecTime) Values
        (CURRENT_TIMESTAMP)';
        END IF;
        FETCH NEXT FROM DBCursor ('Name') INTO DBName;
     END WHILE;

     CLOSE DBCursor;
  END
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

7.38 SET PROCEDURE CACHE

Sets the function/procedure cache size within a job.

Syntax

```
SET PROCEDURE CACHE TO <NumStatements>
```

Usage

Use this statement to set the function/procedure cache size within a job. The default function/procedure cache size in a job is 0, which results in no function/procedure caching at all. Please see the Buffering and Caching topic for more information on the function/procedure caching functionality in ElevateDB.

Note
This statement will override any existing function/procedure cache size set for the session via client-specific connection parameters, so it is normally recommended that you only use this statement within the body of jobs, which use unique sessions per job execution. Please see the CREATE JOB statement for more information on how to create a job.

Examples

```
-- The following job backs up all tables in all databases
-- defined in the current system at 11:00 PM every evening.

CREATE JOB Backup
RUN AS "System"
FROM DATE '2006-01-01' TO DATE '2010-12-31'
DAILY
BETWEEN TIME '11:00 PM' AND TIME '11:30 PM'
CATEGORY 'Backup'
BEGIN
   DECLARE DBCursor CURSOR FOR DBStmt;
   DECLARE DBName VARCHAR DEFAULT '';

   -- 8 procedures is more than we need, but isn't wasteful
   SET PROCEDURE CACHE TO 8;

   PREPARE DBStmt FROM 'SELECT * FROM Databases';

   OPEN DBCursor;

   FETCH FIRST FROM DBCursor ('Name') INTO DBName;

   WHILE NOT EOF(DBCursor) DO
      IF (DBName <> 'Configuration') THEN
         EXECUTE IMMEDIATE 'BACKUP DATABASE "' + DBName + '" AS "' +
                           CAST(CURRENT_DATE AS VARCHAR(10)) +
                           '-' + DBName + '" TO STORE "Backups" INCLUDE
```

```
      CATALOG';
         -- This next call is the one that we're interested in caching
         CALL LogBackup(CURRENT_TIMESTAMP);
      END IF;
      FETCH NEXT FROM DBCursor ('Name') INTO DBName;
   END WHILE;

   CLOSE DBCursor;
END
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

This page intentionally left blank

Chapter 8
Administrative Statements

8.1 Introduction

Administrative statements are used to execute tasks such as migration, backup, restore, repair, optimization, and import/export. This section of the manual details the available administrative statements in ElevateDB.

Notation

The notation used in the syntax section for each administrative statement is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

8.2 ENABLE STATEMENT LOGGING

Enables engine-wide, SQL statement performance logging for ElevateDB.

Syntax

```
ENABLE STATEMENT LOGGING
[MINIMUM EXECUTION <MinExecutionTime>]
[MAXIMUM LOGGED <MaxNumStatements>]

<MinExecutionTime> = The minimum execution time, in seconds, of an SQL
      statement

<MaxNumStatements> = The maximum number of logged SQL statements allowed
```

Usage

Use this statement to enable SQL statement performance logging for the ElevateDB engine (or ElevateDB Server).

The MINIMUM EXECUTION clause is optional and allows you to specify the minimum amount of time, in seconds, that an SQL statement must execute before it is logged. The specified minimum execution time must be greater than 0, and the default minimum execution time is 30 seconds.

The MAXIMUM LOGGED clause is optional and allows you to specify the maximum number of SQL statements that will be logged at one time. The specified maximum number of SQL statements must be greater than 0, and the default maximum number of SQL statements is 128.

Note
You can execute this statement multiple times with different clauses in order to adjust the statement logging parameters. Also, the logged SQL statements are ordered by their execution time, so adjusting either of these values can effect which SQL statements are present in the log (after the ENABLE STATEMENT LOGGING statement is executed).

After enabling SQL statement logging, you can query the LoggedStatements system information table in order to find out which SQL statements have performance issues.

Examples

```
ENABLE STATEMENT LOGGING
MINIMUM EXECUTION 10
MAXIMUM LOGGED 256
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.3 DISABLE STATEMENT LOGGING

Disables engine-wide, SQL statement performance logging for ElevateDB.

Syntax

```
DISABLE STATEMENT LOGGING
```

Usage

Use this statement to disable SQL statement performance logging for the ElevateDB engine (or ElevateDB Server).

Statement performance logging can be enabled via the ENABLE STATEMENT LOGGING statement.

Examples

```
DISABLE STATEMENT LOGGING
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.4 MIGRATE DATABASE

Migrates the current database from an external source using a migrator.

Syntax

```
MIGRATE DATABASE FROM <MigratorName>
[USING <ParamName> = <ParamValue> [,<ParamName> = <ParamValue>]]
[WITH DATA|WITH NO DATA]
[START AT TABLE <TableName>]

<ParamName> = The name of a migrator parameter

<ParamValue> = A literal value to be assigned to the paramter
```

Usage

Use this statement to migrate the current database from an external source.

The FROM clause specifies the migrator to use for the migration process. Migrators are defined using the CREATE MIGRATOR statement.

The USING clause is optional and allows you specify a comma-delimited list of parameters and the values to assign to each parameter. You can find out the parameters for a migrator along with their type and default value by querying the MigratorParams table in the Configuration Database.

The WITH DATA clause specifies that the table data for the external data source should be migrated also, while the WITH NO DATA clause specifies that only the metadata for the database objects should be migrated. The default is WITH NO DATA.

Starting in 2.29, the START AT TABLE clause can be used to specify the table from which to start the migration. The order of the tables that are being migrated is determined by the migrator being used, along with its associated database engine.

Note
ElevateDB allows for tables (including constraints), indexes, triggers, functions, and procedures to migrated from an external data source. However, not all migrators and/or data sources support all of these types of objects.

Examples

```
-- This example migrates a BDE alias called
-- DBDEMOS using the BDE migrator

MIGRATE DATABASE FROM "BDE"
USING DatabaseName = 'DBDEMOS', Transliterate = False
```

Required Privileges

The current user must be granted the proper CREATE privilege on the current database in order to execute this statement. Additionally, if the WITH DATA clause is specified, then the current user must be granted the INSERT privilege on any tables that will be migrated so that the rows from the external data source can be inserted into the new ElevateDB tables. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.5 SET MIGRATOR

Sets the current migrator used for configuration queries.

Syntax

```
SET MIGRATOR TO <MigratorName>
```

Usage

Use this statement to set the current migrator. The current migrator dictates where ElevateDB will retrieve the list of avalable migrator parameters when queries on the MigratorParams Table in the special Configuration Database are executed. The name provided must be a valid migrator name. You can find out which migrators exist by querying the Migrators Table in the Configuration database.

Note
This statement is not persistent and is reset once the current session is disconnected.

Examples

```
SET MIGRATOR TO "EDB"
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.6 BACKUP DATABASE

Backs up the specified database.

Syntax

```
BACKUP DATABASE <Name>
AS <BackupName>
TO STORE <StoreName>
[TABLES <TableName> [,<TableName>]]
[DESCRIPTION <Description>]
[COMPRESSION <Compression>]
[INCLUDE CATALOG [ONLY]]
[EXCLUDE PUBLISHED UPDATES]

<Compression> = 0..9
```

Usage

Use this statement to backup the specified database to a single backup file in the specified store. The store must have already been created using the CREATE STORE statement.

The TABLES clause is optional and allows you specify a subset of tables from the database for backup. The default behavior is to backup all tables in the database.

> **Note**
> The TABLES clause only limits the table data that is backed up. It doesn't prevent the table metadata from being backed up. That is controlled by the INCLUDE CATALOG clause.

The COMPRESSION clause allows you to specify the compression level (0..9). The default level of compression is 6.

The INCLUDE CATALOG clause specifies that the database catalog should be backed up in addition to the table data. It is recommended that you always include this clause. At the very least, you should backup the database with this clause whenever any of the database objects changes. This includes views, procedures, and functions in addition to tables. You can use the ONLY clause to extend the INCLUDE CATALOG clause so that only the catalog is backed up.

> **Note**
> The ONLY clause can only be used if the TABLES clause is not present.

Starting in 2.05, the EXCLUDE PUBLISHED UPDATES clause can be used to specify that you do not want published updates to be included in the backup, which can reduce the size of the backup significantly if there are a lot of published updates present for the tables being backed up.

Examples

```
-- This example backs up the entire
-- Accounting database with the catalog included

BACKUP DATABASE Accounting
AS "AccountingDB-DailyBackup-2007-03-12"
TO STORE "Backups"
INCLUDE CATALOG
```

Required Privileges

The current user must be granted the BACKUP privilege on the specified database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.7 RESTORE DATABASE

Restores the specified backup to a database.

Syntax

```
RESTORE DATABASE <Name>
FROM <BackupName>
IN STORE <StoreName>
[TABLES <TableName> [,<TableName>]]
[INCLUDE CATALOG [ONLY]]
[RESET PUBLISHED TABLES]
```

Usage

Use this statement to restore a backup from a specific store to a database.

The TABLES clause is optional and can be used to limit which tables will be restored from the backup. The default behavior is to restore all tables in the backup.

> **Note**
> The TABLES clause only limits the table data that is restored. It doesn't prevent the table metadata from being restored. That is controlled by the INCLUDE CATALOG clause.

The INCLUDE CATALOG clause specifies that the database catalog should be restored in addition to the table data. It is recommended that you always include this clause. At the very least, you should restore the database with this clause whenever the catalog metadata in the backup doesn't match the existing catalog metadata in the database.

> **Warning**
> Failure to include this clause in such a scenario would cause the restore to fail with an error.

You can use the ONLY clause to extend the INCLUDE CATALOG clause so that only the catalog is restored. This is dependent, of course, on the backup having the database catalog included during the BACKUP DATABASE statement that created the backup.

> **Note**
> The ONLY clause can only be used if the TABLES clause is not present.

Starting in 2.05, the RESET PUBLISHED TABLES clause can be used to reset the publisher IDs for all published tables in the backup being restored. This is useful when you have backed up a database that includes published tables and do not want the target database to have the same publisher IDs. This is a common scenario when setting up a replicated database for the first time.

Note
The RESET PUBLISHED TABLES clause can only be used in conjunction with the INCLUDE CATALOG clause.

Examples

```
-- This example restores the entire
-- Accounting database with the catalog included

RESTORE DATABASE Accounting
FROM "AccountingDB-DailyBackup-2006-03-12"
IN STORE "Backups"
INCLUDE CATALOG
```

Required Privileges

The current user must be granted the RESTORE privilege on the specified database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.8 SET BACKUPS STORE

Sets the current backups store used for configuration queries.

Syntax

```
SET BACKUPS STORE TO <StoreName>
```

Usage

Use this statement to set the current backups store. The current backups store dictates where ElevateDB will retrieve the list of avalable backups when queries on the Backups Table in the special Configuration Database are executed. The name provided must be a valid store name. You can find out which stores exist by querying the Stores Table in the Configuration database.

Note
This statement is not persistent and is reset once the current session is disconnected.

Examples

```
SET BACKUPS STORE TO "Backups"
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.9 PUBLISH DATABASE

Publishes the specified database so that updates are logged.

Syntax

```
PUBLISH DATABASE <DatabaseName>
[TABLES <TableName> [,<TableName>]]
[BY COLUMN|ROW]
```

Usage

Use this statement to publish the specified database so that any updates to the database tables are logged. Once a database table has been published using this statement, any logged updates to the table can be saved to an update file using the SAVE UPDATES statement and then loaded on a diffent copy of the database using the LOAD UPDATES statement.

The TABLES clause is optional and allows you specify a subset of tables from the database for publishing. The default behavior is to publish all tables in the database. If the specified tables, either via the TABLES clause, or all of the tables in the database, have been already published, then this statement is essentially ignored.

The BY clause is optional and allows you to specify how inserts and updates are logged and replicated after the table(s) has/have been published. If a table is published BY COLUMN, then only modified column values are logged when an insert or update takes place. If a table is published BY ROW, then all columns in a row are logged, even if they aren't modified in any way during the insert or update. The default logging method is BY COLUMN.

> **Note**
> For updates and deletes, the column values for the primary key of the table are always logged, independently of the logging that takes place for other columns.

You can unpublish a database using the UNPUBLISH DATABASE statement.

Please see the Replication topic for more information on loading/saving updates for a database.

Examples

```
-- This example publishes the Transaction
-- table in the Accounting database

PUBLISH DATABASE Accounting
TABLES Transaction
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.10 UNPUBLISH DATABASE

Unpublishes the specified database so that updates are no longer logged.

Syntax

```
UNPUBLISH DATABASE <DatabaseName>
[TABLES <TableName> [,<TableName>]]
```

Usage

Use this statement to unpublish the specified database so that any updates to the database tables are no longer logged. Once a database table has been unpublished using this statement, it will no longer be possible to save any logged updates to the table to an update file using the SAVE UPDATES statement.

The TABLES clause is optional and allows you specify a subset of tables from the database for unpublishing. The default behavior is to unpublish all tables in the database. If the specified tables, either via the TABLES clause, or all of the tables in the database, have been already unpublished, then this statement is essentially ignored.

You can publish a database using the PUBLISH DATABASE statement.

Please see the Replication topic for more information on loading/saving updates for a database.

Examples

```
-- This example unpublishes the Transaction
-- table in the Accounting database

UNPUBLISH DATABASE Accounting
TABLES Transaction
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.11 SET INFORMATION COLLATE

Sets the collation used for SQL generation in information queries.

Syntax

```
SET INFORMATION COLLATE TO <CollationName>
```

Usage

Use this statement to set the collation for information queries. The specified collation will be used in place of the defined collation for table columns, index columns, and procedure/function parameters in the CreateSQL column when such objects are queried in the Information schema for a database.

For example, the Tables information table includes a CreateSQL column that indicates the SQL that is used to create the tables. By default, the collations used for the table columns will be the defined collations for the table. By using the SET INFORMATION COLLATE statement, we can change the collation used in the SQL in these columns to something other than the defined collations. This is useful for situations where one wants to reverse-engineer a database and use a different target collation.

Note
This statement is not persistent and is reset once the current session is disconnected. To reset the collation to the defined collations again, just leave out the collation name. Also, you cannot use collation modifiers such as "CI" (case-insensitive) in this statement. The specified collation must be the root collation identifier only.

Examples

```
SET INFORMATION COLLATE TO "UNI"
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.12 COMPARE DATABASE

Compares a source database to a target database and generates the differences as SQL statements.

Syntax

```
COMPARE DATABASE <SourceDatabaseName>
TO <TargetDatabaseName>
[STATEMENT TERM CHAR <TermChar>]
```

Usage

Use this statement to compare a database to another database, generating the minimal difference between the two databases as CREATE, ALTER, and DROP SQL statements. These statements are generated into the SchemaDifference table in the system Information schema of the source database.

If not specified, the statement terminator character defaults to the exclamation character ('!'). If an object requires multiple statements for creating, altering, or dropping sub-objects (such as indexes/triggers for tables), then the statement terminator character will be used for separating the multiple statements in the AlterSQL CLOB column of the SchemaDifference table.

Note
Every time this statement is executed, any existing information in the SchemaDifference table for the source database will be deleted and replaced with the results of the current statement execution.

Examples

```
-- This example compares the Accounting2012
-- database to the Accounting2013 database

COMPARE DATABASE Accounting2012 TO Accounting2013
```

Required Privileges

The current user must be granted the SELECT privilege on the both the source and target databases in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.13 SAVE UPDATES

Saves all logged updates for the specified database.

Syntax

```
SAVE UPDATES FOR DATABASE <DatabaseName>
AS <UpdateName> TO STORE <StoreName>
[TABLES <TableName> [,<TableName>]]
[DESCRIPTION <Description>]
[COMPRESSION <Compression>]
[IF NOT EMPTY]

<Compression> = 0..9
```

Usage

Use this statement to save the updates that have been logged for the specified tables to a single update file in the specified store. The store must have already been created using the CREATE STORE statement, and must be a local store.

The TABLES clause is optional and allows you specify a subset of tables from the database for saving. The default behavior is to save the logged updates for all tables in the database. All specified tables, either via the TABLES clause, or all of the tables in the database, must have been published already using the PUBLISH DATABASE statement, or you will receive an error.

Only the updates that have occurred since the last SAVE UPDATES execution will be saved. Once the SAVE UPDATES execution completes successfully, the logged updates are cleared from the specified tables, or all tables, if the TABLES clause is not specified.

The save process automatically uses special locking to ensure that none of the tables specified are updated until the process completes. This ensures a consistent snapshot of the logged updates for the specified tables.

The COMPRESSION clause allows you to specify the compression level (0..9). The default level of compression is 6.

Starting in 2.05, the IF NOT EMPTY clause can be used to specify that an update file should not be created if there aren't any updates present in any of the tables whose logged updates are being saved in the current operation. This is especially useful for situations where updates are being saved frequently.

Please see the Replication topic for more information on saving updates for a database.

Examples

```
-- This example saves the updates for the
-- Transaction table in the Accounting database

SAVE UPDATES FOR DATABASE Accounting
AS "AccountingDB-Updates-2007-03-12"
TO STORE TransactionUpdatesOut
```

```
TABLES Transaction
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.14 LOAD UPDATES

Loads all logged updates from an update file into the specified database.

Syntax

```
LOAD UPDATES FOR DATABASE <Name>
FROM <UpdateName> IN STORE <StoreName>
[TABLES <TableName> [,<TableName>]]
[MERGE DUPLICATE INSERTS]
[IGNORE|INSERT MISSING UPDATES]
[DISABLE TRIGGERS]
```

Usage

Use this statement to load an update file that has been created from logged updates for a different copy of the database from the specified store. The store must have already been created using the CREATE STORE statement, and must be a local store.

The TABLES clause is optional and allows you specify a subset of tables from the database for loading. The default behavior is to load the logged updates for all tables in the database.

The MERGE DUPLICATE INSERTS clause is optional and allows you to specify that row inserts whose primary key already exists are treated as updates. By default, row inserts whose primary key already exists will cause an exception to be raised.

The IGNORE/INSERT MISSING UPDATES clause is optional and allows you to specify that either:

- Row updates whose primary key cannot be found in the target table be ignored and not raise an exception. (IGNORE)

- Row updates whose primary key cannot be found in the target table be converted into inserts. (INSERT)

By default, row updates whose primary key cannot be found will cause an exception to be raised. Also, row deletions whose primary key cannot be found are always ignored since such a situation is considered equivalent to a row deletion.

The DISABLE TRIGGERS clause is optional and allows you to specify that all tables being updated should have all of their triggers disabled before any updates are loaded. The current state of the triggers is saved before the triggers are disabled, and restored after the updates are finished loading.

If loading multiple update files from the same source database, it is important that you load the update files in the creation order dictated by the creation timestamp of the update files. You can use the SET UPDATES STORE statement with the Updates table in the Configuration database to retrieve a list of the updates in a given store and order them by their creation timestamp.

> **Note**
> The creation timestamp is stored inside of the update file and is different from the operating system's file creation timestamp.

The load process automatically uses a transaction to ensure that the load is done in an atomic fashion. If any errors occur during the load, then the entire process is rolled back.

You can use the CREATE TABLE FROM UPDATES clause to create a table that contains the contents of any given update file. This is useful for auditing purposes, or for debugging issues when loading a particular update file into a database.

Please see the Replication topic for more information on loading updates for a database.

Examples

```
-- This example loads the updates for the
-- Transaction table in the Accounting database

LOAD UPDATES FOR DATABASE Accounting
FROM "AccountingDB-Updates-2007-03-12"
IN STORE TransactionUpdatesIn
TABLES Transaction
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.15 SET UPDATES STORE

Sets the current updates store used for configuration queries.

Syntax

```
SET UPDATES STORE TO <StoreName>
```

Usage

Use this statement to set the current updates store. The current updates store dictates where ElevateDB will retrieve the list of avalable updates when queries on the Updates Table in the special Configuration Database are executed. The name provided must be a valid store name. You can find out which stores exist by querying the Stores Table in the Configuration database.

> **Note**
> This statement is not persistent and is reset once the current session is disconnected.

Please see the Replication topic for more information on loading/saving updates for a database.

Examples

```
SET UPDATES STORE TO "RemoteOffice"
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.16 COPY FILE

Copies the specified file to a new file name in the same store or in a different store.

Syntax

```
COPY FILE <FileName> IN STORE <StoreName>
TO <FileName> [IN STORE <StoreName>]
```

Usage

Use this statement to copy the specified file in the specified store to a new file in the same or a different store. The stores may be either local or remote, and copying from a local store to a remote store, or vice-versa, is permitted. Both stores must have already been created using the CREATE STORE statement.

Examples

```
-- This example copies an update file
-- from a local store called OutUpdates to
-- a remote store called RemoteUpdates

COPY FILE "AccountingDB-Updates-2007-03-12.EDBUpd" IN STORE "OutUpdates"
TO "AccountingDB-Updates-2007-03-12.EDBUpd" IN STORE "RemoteUpdates"
```

Required Privileges

The current user must be granted the SELECT privilege on the store from which the file is being copied, and the CREATE privilege on the store where the new file will be created. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.17 RENAME FILE

Renames the specified file to a new file name in the same store.

Syntax

```
RENAME FILE <FileName> IN STORE <StoreName>
TO <FileName>
```

Usage

Use this statement to rename the specified file in the specified store to a new file in the same store. The store may be either local or remote. The store must have already been created using the CREATE STORE statement.

Examples

```
-- This example renames an update file
-- in a local store called OutUpdates

RENAME FILE "AccountingDB-Updates-2007-03-12.EDBUpd" IN "STORE OutUpdates"
TO "AcctUpdatersdelphiuni20070312.EDBUpd"
```

Required Privileges

The current user must be granted the ALTER privilege on the store in which the file is being renamed. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.18 DELETE FILE

Deletes the specified file in the specified store.

Syntax

```
DELETE FILE <FileName> FROM STORE <StoreName>
```

Usage

Use this statement to delete the specified file in the specified store. The store may be either local or remote. The store must have already been created using the CREATE STORE statement.

Examples

```
-- This example deletes an update file
-- in a local store called OutUpdates

DELETE FILE "AccountingDB-Updates-2007-03-12.EDBUpd" FROM STORE "OutUpdates"
```

Required Privileges

The current user must be granted the DROP privilege on the store in which the file is being deleted. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.19 SET FILES STORE

Sets the current files store used for configuration queries.

Syntax

```
SET FILES STORE TO <StoreName>
```

Usage

Use this statement to set the current files store. The current files store dictates where ElevateDB will retrieve the list of avalable filess when queries on the Files Table in the special Configuration Database are executed. The name provided must be a valid store name. You can find out which stores exist by querying the Stores Table in the Configuration database.

> **Note**
> This statement is not persistent and is reset once the current session is disconnected.

Examples

```
SET FILES STORE TO "RemoteOffice"
```

Required Privileges

Any user can execute this statement.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.20 VERIFY TABLE

Verifies a table to see if any corruption exists in the table.

Syntax

```
VERIFY TABLE <TableName>
[STRUCTURE ONLY]
```

Usage

Use this statement to verify the specified table.

The STRUCTURE ONLY clause can be used to specify that ElevateDB should only do a structural scan, and not actually examine the contents of rows for more in-depth corruption checks. Using this clause can improve the performance of a table verification greatly, but does not guarantee that the row contents are not possibly still corrupted. Usually the best practice is to first execute the VERIFY TABLE statement on the specified table with the STRUCTURE ONLY clause. If the VERIFY TABLE execution indicates any corruption at all, then you should execute the REPAIR TABLE statement on the specified table without the STRUCTURE ONLY clause so that the table is exhaustively repaired.

You can use the STMTRESULT function to retrieve the result of a table verification in any procedure, function, script, or job. For retrieving the result of a table verification from a client application, please see your compiler-specific manual for retrieving the result of a statement exectuion.

The VERIFY TABLE statement requires a read lock on the specified table during its execution.

Examples

```
VERIFY TABLE Orders
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.21 REPAIR TABLE

Repairs a table.

Syntax

```
REPAIR TABLE <TableName>
[STRUCTURE ONLY]
[STATISTICS]
```

Usage

Use this statement to repair the specified table.

The STRUCTURE ONLY clause can be used to specify that ElevateDB should only do a structural scan, and not actually examine the contents of rows for more in-depth corruption checks. Using this clause can improve the performance of a repair greatly, but does not guarantee that the row contents are not possibly still corrupted. Usually the best practice is to first execute the VERIFY TABLE statement on the specified table with the STRUCTURE ONLY clause. If the VERIFY TABLE execution indicates any corruption at all, then you should execute the REPAIR TABLE statement on the specified table without the STRUCTURE ONLY clause so that the table is exhaustively repaired.

The STATISTICS clause was added in ElevateDB 2.03 Build 7 in order to correct an issue with prior releases and builds where the engine was incorrectly calculating the index statistics for a table. These statistics are used by the query optimizer to estimate the amount of I/O that a particular index scan will cause, and so it is very important that this information be correct. Running the REPAIR TABLE statement with the STATISTICS clause will cause the index statistics to be recalculated for all indexes present in the specified table.

> **Note**
> The STATISTICS clause causes the REPAIR TABLE to only recalculate the index statistics, and not actually perform a complete repair.

If you create a table with ElevateDB 2.03 Build 7 or higher, then you will never need to use the STATISTICS clause. The STATISTICS clause is deprecated in 2.04 or higher due to the new repair/verify functionality.

You can use the STMTRESULT function to retrieve the result of a table repair in any procedure, function, script, or job. For retrieving the result of a table repair from a client application, please see your compiler-specific manual for retrieving the result of a statement exectuion.

The REPAIR TABLE statement requires exclusive access to the specified table.

Examples

```
REPAIR TABLE Orders
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.22 OPTIMIZE TABLE

Optimizes a table and removes any unused space.

Syntax

```
OPTIMIZE TABLE <TableName>
[USING <IndexName>]
[NO BACKUP FILES]
```

Usage

Use this statement to optimize the specified table. Optimization accomplishes three things:

- It removes any unused space from a table and compacts the data.
- It rebuilds all indexes, both system and user-defined, for the table.
- It can optionally physically order the rows in the table according to the most-frequently-used index via the USING clause. This helps ElevateDB optimize the read-ahead of physical rows when executing SQL SELECT queries.

The USING clause is optional. The default behavior is for ElevateDB to physically re-order the rows in the table according to the primary key, or the natural order if there is no primary key defined for the table.

The NO BACKUP FILES clause is optional. Unless this clause is specified, ElevateDB will create backup files (*.old) of the physical table files when optimizing the table.

Examples

```
-- This example optimizes the Customer table
-- and physically orders the rows according to the
-- primary key

OPTIMIZE TABLE Customer
```

Required Privileges

The current user must be granted the MAINTAIN privilege on the current database in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.23 IMPORT TABLE

Imports the data from a delimited text file into a table or view.

Syntax

```
IMPORT TABLE <ImportTable>
FROM <FileName>
IN STORE <StoreName>
[(<ColumnName>[,<ColumnName>])]
[FORMAT DELIMITED|XML]
[ENCODING AUTO|ANSI|UNICODE]
[DELIMITER CHAR <DelimiterChar>]
[QUOTE CHAR <QuoteChar>]
[DATE FORMAT <DateFormat>]
[TIME FORMAT <TimeFormat> [AM LITERAL <AMLiteral> PM LITERAL <PMLiteral>]]
[DECIMAL CHAR <DecimalChar>]
[BOOLEAN TRUE LITERAL <TrueLiteral> FALSE LITERAL <FalseLiteral>]
[USE HEADERS]
[MAX ROWS <MaxRowCount>]

<ImportTable> = <TableName>|<ViewName>

<DateFormat> =

YYYY or YY = Year digits
MM or M = Month digits
DD or D = Day digits
Any other character = literal

<TimeFormat> =

HH or H = Hours digits
MM or M = Minutes digits
SS or S = Seconds digits
ZZZ or Z = Milliseconds digits
N = AM/PM literal
Any other character = literal
```

Usage

Use this statement to import data from a delimited or XML text file in the specified store into a table or view.

Use the optional FORMAT clause to specify the format of the incoming text file. The format can be specified as DELIMITED or XML, and defaults to DELIMITED if the FORMAT clause is not specified.

For delimited text files, ElevateDB expects each incoming row of data to be terminated with a carriage return (#13) and line feed (#10) character, or just a line feed character (#10). For XML text files, ElevateDB expects the incoming data to be in in the following format:

```
<row>
```

```
    <columnname>data</columnname>
    [<columnname>data</columnname>]
</row>
```

The store must have already been created using the CREATE STORE statement, and must be a local store.

Any existing data in the table or view is not overwritten, and the data in the text file is appended to the table or view. If you wish to replace the contents of a table or view with the contents of the text file, then you should execute the following DELETE statement before executing the IMPORT TABLE statement:

```
DELETE FROM <TableName>|<ViewName>
```

If importing into a view, the view must be updateable or an error will be raised.

You can specify the columns in the import file by including them in parentheses after the FROM clause.

Use the ENCODING, DELIMITER CHAR, QUOTE CHAR, DATE FORMAT, TIME FORMAT, DECIMAL CHAR, and BOOLEAN LITERAL clauses to control how ElevateDB reads the data from the import file.

The ENCODING clause is used to determine the character encoding of the import file. If this cause isn't included, then the default encoding is AUTO, meaning that ElevateDB will attempt to determine the character encoding of the import file by first looking for BOM (Byte Order Mark) bytes at the beginning of the file. If it doesn't find them, then it will attempt to determine the character encoding by examining the actual import file characters themselves. If the import file does not include BOM bytes, and you know that the character encoding is UNICODE, then you should specify the UNICODE encoding option.

If the DELIMITER CHAR clause is not included, then the default delimiter character is the comma ',' character.

> **Note**
> The DELIMITER CHAR clause is only valid when the FORMAT clause is DELIMITED (the default if the FORMAT clause is not specified).

If the QUOTE CHAR clause is not included, then the default quote character for character strings is the double quote '"' character.

If the DATE FORMAT clause is not specified, then the default date format is the ANSI SQL standard date format 'YYYY-MM-DD'.

If the TIME FORMAT clause is not specified, then the default date format is 'HH:MM:SS.ZZZ N'. The AM LITERAL and PM LITERAL clauses are only used if the N format specifier is included in the time format.

If the DECIMAL CHAR clause is not included, then the default decimal separator character for character strings is the period '.' character.

If the BOOLEAN TRUE LITERAL is not included then the default boolean True literal value is 'TRUE'. If the BOOLEAN FALSE LITERAL is not included then the default boolean False literal value is 'FALSE'.

For delimited text files, the USE HEADERS clause determines whether ElevateDB interprets the first line in the import file as a list of column names that are included in the import file. If any column specified in this

line is not a valid column for the table, then it is simply ignored.

For XML text files, the USE HEADERS clause determines whether ElevateDB looks for the following tags in the import file to control which columns are imported:

```
<columns>
   <column>columnname</column>
   [<column>columnname</column>]
<columns>
```

If any column specified using these tags is not a valid column for the table, then it is simply ignored. Also, due to the nature of XML, it is possible to have multiple <columns> sections in the same XML import file in order to allow for different columns for the incoming data.

Note
This clause overrides any columns specified after the FROM clause.

The MAX ROWS clause can be used to limit the number of rows that are imported. This is useful when you simply want to do a preview in order to determine whether the first few lines of the import file are importing correctly.

Examples

```
-- The following example imports tab-delimited
-- data from an import file into the Customer table,
-- using the first line in the import file to determine
-- which columns to import

IMPORT TABLE Customer
FROM "custdata.txt"
IN STORE "ImportFiles"
DELIMITER CHAR #9
USE HEADERS
```

Required Privileges

The current user must be granted the INSERT and SELECT privileges on the table or view into which the data is being imported, and the SELECT privilege on the store from which the file is being imported. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.24 EXPORT TABLE

Exports the data from a table or view into a delimited text file.

Syntax

```
EXPORT TABLE <ExportTable>
TO <FileName>
IN STORE <StoreName>
[(<ColumnName>[,<ColumnName>])]
[FORMAT DELIMITED|XML]
[ENCODING AUTO|ANSI|UNICODE]
[DELIMITER CHAR <DelimiterChar>]
[QUOTE CHAR <QuoteChar>]
[DATE FORMAT <DateFormat>]
[TIME FORMAT <TimeFormat> [AM LITERAL <AMLiteral> PM LITERAL <PMLiteral>]]
[DECIMAL CHAR <DecimalChar>]
[BOOLEAN TRUE LITERAL <TrueLiteral> FALSE LITERAL <FalseLiteral>]
[INCLUDE HEADERS]
[MAX ROWS <MaxRowCount>]

<ExportTable> = <TableName>|<ViewName>

<DateFormat> =

YYYY or YY = Year digits
MM or M = Month digits
DD or D = Day digits
Any other character = literal

<TimeFormat> =

HH or H = Hours digits
MM or M = Minutes digits
SS or S = Seconds digits
ZZZ or Z = Milliseconds digits
N = AM/PM literal
Any other character = literal
```

Usage

Use this statement to export data from a table or view into a delimited or XML text file in the specified store.

Use the optional FORMAT clause to specify the format of the created text file. The format can be specified as DELIMITED or XML, and defaults to DELIMITED if the FORMAT clause is not specified.

For delimited text files, ElevateDB terminates each row of data in the export file with a carriage return (#13) and line feed (#10) character.

For XML text files, ElevateDB outputs the data in the export file in the following format:

```
<row>
   <columnname>data</columnname>
   [<columnname>data</columnname>]
</row>
```

The store must have already been created using the CREATE STORE statement, and must be a local store.

Any existing data in the destination export file is overwritten.

You can specify the columns in the export file by including them in parentheses after the TO clause.

The ENCODING clause is used to determine the character encoding of the export file. If this cause isn't included, then the default encoding is AUTO, meaning that ElevateDB will create the export file using the current character encoding of the engine itself (ANSI/UNICODE). Whenever the character encoding of the export file is UNICODE (explicitly or by using AUTO), ElevateDB will always write out BOM (Byte Order Mark) bytes as the first two bytes of the export file.

Use the DELIMITER CHAR, QUOTE CHAR, DATE FORMAT, TIME FORMAT, DECIMAL CHAR, and BOOLEAN LITERAL clauses to control how ElevateDB writes the data to the export file.

If the DELIMITER CHAR clause is not included, then the default delimiter character is the comma ',' character.

If the QUOTE CHAR clause is not included, then the default quote character for character strings is the double quote '"' character.

If the DATE FORMAT clause is not specified, then the default date format is the ANSI SQL standard date format 'YYYY-MM-DD'.

If the TIME FORMAT clause is not specified, then the default date format is 'HH:MM:SS.ZZZ N'. The AM LITERAL and PM LITERAL clauses are only used if the N format specifier is included in the time format.

If the DECIMAL CHAR clause is not included, then the default decimal separator character for character strings is the period '.' character.

If the BOOLEAN TRUE LITERAL is not included then the default boolean True literal value is 'TRUE'. If the BOOLEAN FALSE LITERAL is not included then the default boolean False literal value is 'FALSE'.

For delimited text files, the INCLUDE HEADERS clause determines whether ElevateDB writes the first line in the export file as a list of column names that have been included in the export.

For XML text files, the INCLUDE HEADERS clause determines whether ElevateDB writes the list of column names that have been included in the export in the following format:

```
<columns>
   <column>columnname</column>
   [<column>columnname</column>]
<columns>
```

The MAX ROWS clause can be used to limit the number of rows that are exported. This is useful when you simply want to test an import of the file on another system in order to determine whether the export file is being generated correctly.

Examples

```
-- The following example exports the CustomerNo
-- and TotalOrders columns in the Customer table as
-- comma-delimited data into an export file

EXPORT TABLE Customer
TO "custordtotals.txt"
IN STORE "ExportFiles"
(CustomerNo, TotalOrders)
```

Required Privileges

The current user must be granted the SELECT privilege on the table or view from which the data is being exported, and the CREATE privilege on the store in which the export file is being created. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.25 EMPTY TABLE

Empties a base table by truncation.

Syntax

```
EMPTY TABLE <TableName>
[IGNORE CONSTRAINTS]
```

Usage

Use this statement to empty a base table so that all rows are physically removed from the table. When all rows are normally deleted from a table using the DELETE statement, the physical row space is marked for re-use, and the physical table files where the rows, indexes, and BLOBs are stored does not decrease in size. Emptying a table also physically truncates the table file space so that the table files appear as if they were newly-created.

The IGNORE CONSTRAINT clause is used to bypass any constraint checks, which makes the EMPTY TABLE operation faster, but at the risk of violating foreign-key constraints, if any are present. See below for more information on the elevated privileges required to use this clause.

Examples

```
-- The following example empties
-- the Customer table, ignoring any
-- defined constraints

EMPTY TABLE Customer
IGNORE CONSTRAINTS
```

Required Privileges

The current user must be granted the DELETE privilege on the table being emptied. If the IGNORE CONSTRAINTS clause is specified, then the current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.26 DISCONNECT SERVER SESSION

Disconnects a server session on an ElevateDB Server.

Syntax

```
DISCONNECT SERVER SESSION <SessionID>
```

Usage

Use this statement to disconnect the specified session on the ElevateDB Server that the current remote session is connnected to. If the current session is local, and not remote, then this statement is simply ignored. To get the session IDs (and other information) for the current sessions on the ElevateDB Server that the current remote session is connected to, you can query the ServerSessions Table in the special Configuration Database.

Examples

```
DISCONNECT SERVER SESSION 8034217
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

8.27 REMOVE SERVER SESSION

Removes a server session from an ElevateDB Server.

Syntax

```
REMOVE SERVER SESSION <SessionID>
```

Usage

Use this statement to remove the specified session from the ElevateDB Server that the current remote session is connnected to. If the current session is local, and not remote, then this statement is simply ignored. To get the session IDs (and other information) for the current sessions on the ElevateDB Server that the current remote session is connected to, you can query the ServerSessions Table in the special Configuration Database.

Examples

```
REMOVE SERVER SESSION 8034217
```

Required Privileges

The current user must be granted the system-defined Administrators role in order to execute this statement. Please see the User Security topic for more information.

SQL 2003 Standard Deviations

This statement deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This SQL statement is an ElevateDB extension.

This page intentionally left blank

Chapter 9
Numeric Functions

9.1 Introduction

Numeric functions are used to convert and manipulate exact and approximate numeric types in ElevateDB SQL expressions. This section of the manual details the available numeric functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

9.2 ABS

Converts a number to its absolute value.

Syntax

```
ABS(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
Same as Input
```

Usage

The ABS function converts a numeric value to its absolute, or non-negative value.

Examples

```
SELECT ABS(Difference) AS Difference
FROM Populations
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

9.3 ACOS

Returns the arc cosine of a number as an angle expressed in radians.

Syntax

```
ACOS(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The ACOS function returns the arc cosine of a number as an angle expressed in radians. Arc cosine is the inverse operation of cosine.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.4 ASIN

Returns the arc sine of a number as an angle expressed in radians.

Syntax

```
ASIN(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The ASIN function returns the arc sine of a number as an angle expressed in radians.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.5 ATAN

Returns the arc tangent of a number as an angle expressed in radians.

Syntax

```
ATAN(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The ATAN function returns the arc tangent of a number as an angle expressed in radians.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.6 ATAN2

Returns the arc tangent of x and y coordinates as an angle expressed in radians.

Syntax

```
ATAN2(<NumericExpression>,<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The ATAN2 function returns the arc tangent of x and y coordinates as an angle expressed in radians.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.7 CEILING

Returns the lowest integer greater than or equal to a number.

Syntax

```
CEILING(<NumericExpression>)
CEIL(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
INTEGER
```

Usage

The CEIL or CEILING function returns the lowest integer greater than or equal to a number.

Examples

```
SELECT SUM(CEIL(Distance)) AS ApproxDistance
FROM Destinations
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

9.8 COS

Returns the cosine of an angle.

Syntax

Returns

```
FLOAT
```

Usage

The COS function returns the cosine of an angle.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.9 COT

Returns the cotangent of an angle.

Syntax

```
COT(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The COT function returns the cotangent of an angle.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.10 CURRENT_SESSIONID

Returns the current session ID as an integer.

Syntax

```
CURRENT_SESSIONID()
```

Returns

```
INTEGER
```

Usage

The CURRENT_SESSIONID function returns the current session ID as an integer.

Examples

```
SELECT *
FROM Configuration.ServerSessionLocks
WHERE SessionID = CURRENT_SESSIONID()
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.11 DEGREES

Converts a number representing radians into degrees.

Syntax

```
DEGREES(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The DEGREES function converts a number representing radians into degrees.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.12 EXP

Returns the exponential value of a number.

Syntax

```
EXP(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The EXP function returns the exponential value of a number, which is E raised to the power of X, where E is the base of the natural logarithms.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

9.13 FLOOR

Returns the highest integer less than or equal to a number.

Syntax

```
FLOOR(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
INTEGER
```

Usage

The FLOOR function returns the highest integer less than or equal to a number.

Examples

```
SELECT SUM(FLOOR(Distance)) AS ApproxDistance
FROM Destinations
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

9.14 LASTIDENTITY

Returns the last identity value assigned to the specified column in the specified table.

Syntax

```
LASTIDENTITY(<StringExpression>, <StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
INTEGER
```

Usage

The LASTIDENTITY function returns the last identity value assigned to the specified column in the specified table. The table name is the first parameter to the function, and the column name is the second parameter to the function.

Warning
This function only returns the last identity value assigned for the specified column for the current session. It does **not** reflect any identity values assigned to any other session for the same table. Therefore, if the current session has not inserted any rows in the specified table since the session was first logged in, then the value returned will be 0.

Examples

```
SELECT LASTIDENTITY('Customer', 'CustNo') AS LastCustNo
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.15 LOG

Returns the natural logarithm of a number.

Syntax

```
LOG(<NumericExpression>)
LN(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The LOG or LN function returns the natural logarithm of a number.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
LOG	The LOG version of the function is an ElevateDB extension.

9.16 LOG10

Returns the base 10 logarithm of a number.

Syntax

```
LOG10(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The LOG10 function returns the base 10 logarithm of a number.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.17 PI

Returns the ratio of a circle's circumference to its diameter.

Syntax

```
PI()
```

Returns

```
FLOAT
```

Usage

The PI function returns the ratio of a circle's circumference to its diameter - approximated as 3.1415926535897932385.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.18 POWER

Returns the value of a base number raised to the specified power.

Syntax

```
POWER(<NumericExpression> TO <IntegerExpression>)
POWER(<NumericExpression>, <IntegerExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC

<IntegerExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
```

Returns

```
FLOAT
```

Usage

The POWER function returns value of a base number raised to the specified power.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
TO	The TO version of the function is an ElevateDB extension.

9.19 RADIANS

Converts a number representing degrees into radians.

Syntax

```
RADIANS(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The RADIANS function converts a number representing degrees into radians.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.20 RAND

Returns a random number.

Syntax

```
RAND([RANGE <IntegerExpression>])
RAND([<IntegerExpression>])

<IntegerExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
```

Returns

```
FLOAT if no range specified
INTEGER equivalent to range if range specified
```

Usage

The RAND function returns a random number. The RANGE value is optional used to limit the random numbers returned to between 0 and the RANGE value specified. If the range is not specified then any number within the range of positive FLOAT values may be returned.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.21 ROUND

Rounds a number to a specified number of decimal places.

Syntax

```
ROUND(<NumericExpression> [TO <IntegerExpression>])
ROUND(<NumericExpression> [, <IntegerExpression>])

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC

<IntegerExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
```

Returns

```
Same as input
```

Usage

The ROUND function rounds a numeric value to a specified number of decimal places. The number of decimal places is optional, and if not specified the value returned will be rounded to 0 decimal places.

> **Note**
> The ROUND function performs "normal" rounding where the number is rounded up if the fractional portion beyond the number of decimal places being rounded to is greater than or equal to 5 and down if the fractional portion is less than 5. Also, if using the ROUND function with DOUBLE PRECISION or FLOAT values, it is possible to encounter rounding errors due to the nature of floating-point values and their inability to accurately express certain fractional real numbers.

Examples

```
SELECT SUM(ROUND(Distance TO 2)) AS ApproxDistance
FROM Destinations
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.22 SIGN

Returns -1 if a number is less than 0, 0 if a number is 0, or 1 if a number is greater than 0.

Syntax

```
SIGN(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
INTEGER
```

Usage

The SIGN function returns -1 if a number is less than 0, 0 if a number is 0, or 1 if a number is greater than 0.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.23 SIN

Returns the sine of an angle.

Syntax

```
SIN(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The SIN function returns the sine of an angle.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.24 SQRT

Returns the square root of a number.

Syntax

```
SQRT(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The SQRT function returns the square root of a number.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

9.25 TAN

Returns the tangent of an angle.

Syntax

```
TAN(<NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The TAN function returns the tangent of an angle.

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

9.26 TRUNCATE

Truncates a numeric argument to the specified number of decimal places.

Syntax

```
TRUNCATE(<NumericExpression> [TO <IntegerExpression>])
TRUNCATE(<NumericExpression> [, <IntegerExpression>])
TRUNC(<NumericExpression> [TO <IntegerExpression>])
TRUNC(<NumericExpression> [, <IntegerExpression>])

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC

<IntegerExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
```

Returns

```
Same as input
```

Usage

The TRUNC or TRUNCATE function truncates a numeric value to a specified number of decimal places. The number of decimal places is optional, and if not specified the value returned will be truncated to 0 decimal places.

> **Note**
> If using the TRUNC or TRUNCATE function with FLOAT or DOUBLE PRECISION values and a number of decimal places greater than 0, it is possible to encounter truncation errors due to the nature of floating-point values and their inability to accurately express certain fractional real numbers.

Examples

```
SELECT SUM(TRUNCATE(Distance TO 2)) AS ApproxDistance
```

```
FROM Destinations
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

This page intentionally left blank

Chapter 10
String Functions

10.1 Introduction

String functions are used to convert and manipulate string types in ElevateDB SQL expressions. This section of the manual details the available string functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

10.2 CHARACTER_LENGTH

Returns the length of a string value.

Syntax

```
CHARACTER_LENGTH(<StringExpression>)
CHAR_LENGTH(<StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
INTEGER
```

Usage

The CHAR_LENGTH function returns the length of a string value as an integer value.

Examples

```
SELECT Notes, CHAR_LENGTH(Notes) AS NumChars
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

10.3 CONCAT

Concatenates two string values together.

Syntax

```
CONCAT(<StringExpression> WITH <StringExpression>)
CONCAT(<StringExpression>, <StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as first input
```

Usage

The CONCAT function concatenates two strings together and returns the concatenated result.

Examples

```
UPDATE Customers
SET notes = CONCAT(Notes WITH #13 + #10 + #13 + #10 + 'End of Notes')
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.4 CURRENT_GUID

Returns a new GUID value as a string.

Syntax

```
CURRENT_GUID()
```

Returns

```
GUID
```

Usage

The CURRENT_GUID function returns a new GUID value as a 38-character string.

Examples

```
INSERT INTO Transactions
VALUES (CURRENT_GUID(), CURRENT_TIMESTAMP(), CURRENT_USER(),
        'DEBIT', 200.00)
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.5 CURRENT_USER

Returns the current user's name as a string.

Syntax

```
CURRENT_USER()
```

Returns

```
VARCHAR
```

Usage

The CURRENT_USER function returns the current user's name as a string.

Examples

```
SELECT *
FROM TransactionHistory
WHERE User = CURRENT_USER()
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

10.6 CURRENT_DATABASE

Returns the current database's name as a string.

Syntax

```
CURRENT_DATABASE()
```

Returns

```
VARCHAR
```

Usage

The CURRENT_DATABASE function returns the current database's name as a string.

Examples

```
-- This script backs up the current database

SCRIPT
BEGIN
   EXECUTE IMMEDIATE 'BACKUP DATABASE "' + CURRENT_DATABASE() + '" AS "' +
                     CAST(CURRENT_DATE AS VARCHAR(10)) +
                     '-' + CURRENT_DATABASE() +
                     '" TO STORE "Backups" INCLUDE CATALOG';
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.7 CURRENT_COMPUTER

Returns the current computer's name as a string.

Syntax

```
CURRENT_COMPUTER()
```

Returns

```
VARCHAR
```

Usage

The CURRENT_COMPUTER function returns the current computer's name as a string.

> **Note**
> When this function is used with the ElevateDB Server, it will always return the computer name for the ElevateDB Server machine.

Examples

```
-- This trigger logs information about
-- inserts, updates, and deletes,
-- including the computer the executed the
-- operation

CREATE TRIGGER "LogValues" AFTER ALL ON "customer"
BEGIN
   IF OPERATION() IN ('Insert','Update') THEN
      EXECUTE IMMEDIATE 'INSERT INTO AuditLog (Operation, Computer, Value)
         VALUES (?,?,?)' USING OPERATION(), CURRENT_COMPUTER(),
      NEWROW.MyColumn;
   ELSE IF OPERATION()='Delete' THEN
      EXECUTE IMMEDIATE 'INSERT INTO AuditLog (Operation, Computer, Value)
         VALUES (?,?,?)' USING OPERATION(), CURRENT_COMPUTER(),
      OLDROW.MyColumn;
   END IF;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.8 LEFT

Extracts a certain number of characters from the leading portion of a string value.

Syntax

```
LEFT(<StringExpression> FOR <IntegerExpression>)
LEFT(<StringExpression>, <IntegerExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The LEFT function extracts a certain number of characters from the leading portion of a string. The FOR parameter specifies the length of the extracted substring. If the FOR parameter is greater than the length of the input string, then the result will be the input string.

Examples

```
SELECT LEFT(CustomerID, 6)
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.9 LENGTH

Returns the length of a string or binary value

Syntax

```
LENGTH(<StringExpression>|<BinaryExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB

<BinaryExpression> =

Type of:

BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
```

Returns

```
INTEGER
```

Usage

The LENGTH function returns the length of a string or binary value as an integer value.

Examples

```
SELECT Notes, LENGTH(Notes) AS NumChars
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.10 LOWER

Forces a string to lower-case.

Syntax

```
LOWER(<StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The LOWER function converts all characters in a string value to lower-case. The collation of the input value is used to determine how the lower-case operation is performed.

Examples

```
SELECT LOWER(CustomerID) AS CustomerID
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Argument Separator	The use of a comma separator for the function arguments is an ElevateDB extension.

10.11 LTRIM

Removes any leading spaces from a string.

Syntax

```
LTRIM(<StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The LTRIM function removes any leading spaces from a string.

Examples

```
SELECT LTRIM(Name) AS Name
AS Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.12 OCCURS

Returns the number of times one string value is present within another string value.

Syntax

```
OCCURS(<StringExpression> IN <StringExpression>)
OCCURS(<StringExpression>, <StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
INTEGER
```

Usage

The OCCURS function returns the number of occurrences of one string within another string. If the search string is not present, then 0 will be returned.

Examples

```
SELECT *
FROM Customers
WHERE (OCCURS('COMPLAINT' IN UPPER(Notes)) > 0)
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.13 POSITION

Returns the position of one string value within another string value.

Syntax

```
POSITION(<StringExpression> IN <StringExpression>)
POSITION(<StringExpression>, <StringExpression>)
POS(<StringExpression> IN <StringExpression>)
POS(<StringExpression>, <StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
INTEGER
```

Usage

The POSITION function returns the position of one string within another string. If the search string is not present, then 0 will be returned.

Examples

```
SELECT *
FROM Customers
WHERE (POSITION('COMPLAINT' IN UPPER(Notes)) > 0)
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Argument Separator	The use of a comma separator for the function arguments is an ElevateDB extension.
POS	The POS version of the function is an ElevateDB extension.

10.14 REPEAT

Repeats a string value a specified number of times.

Syntax

```
REPEAT(<StringExpression> FOR <IntegerExpression>)
REPEAT(<StringExpression>, <IntegerExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The REPEAT function repeats a given string a specified number of times and returns the concatenated result.

Examples

```
SELECT REPEAT( '=' FOR 60) + #13 + #10 +
       CustomerID + #13 + #10 +
       REPEAT( '=' FOR 60) + #13 + #10 +
Notes AS Notes
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.15 REPLACE

Replaces all occurrences of one string value with a new string value within another string value.

Syntax

```
REPLACE(<StringExpression> WITH <StringExpression>
        IN <StringExpression>)
REPLACE(<StringExpression>, <StringExpression>,
        <StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as last input
```

Usage

The REPLACE function replaces all occurrences of a given string with a new string within another string. If the search string is not present, then the result will be the input string.

Examples

```
UPDATE Customers
SET Notes = REPLACE( 'Complaint' WITH 'Suggestion' IN Notes)
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.16 RIGHT

Extracts a certain number of characters from the trailing portion of a string value.

Syntax

```
RIGHT(<StringExpression> FOR <IntegerExpression>)
RIGHT(<StringExpression>, <IntegerExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The RIGHT function extracts a certain number of characters from the trailing portion of a string. The FOR parameter specifies the length of the extracted substring. If the FOR parameter is greater than the length of the input string, then the result will be the input string.

Examples

```
SELECT RIGHT(CustomerID, 6)
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.17 RTRIM

Removes any trailing spaces from a string.

Syntax

```
RTRIM(<StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The RTRIM function removes any trailing spaces from a string.

Examples

```
SELECT RTRIM(Name) AS Name
AS Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

10.18 SUBSTRING

Extracts a portion of a string value.

Syntax

```
SUBSTRING(<StringExpression> FROM <IntegerExpression>
          [FOR <IntegerExpression>])
SUBSTRING(<StringExpression>, <IntegerExpression>
          [, <IntegerExpression>])
SUBSTR(<StringExpression> FROM <IntegerExpression>
       [FOR <IntegerExpression>])
SUBSTR(<StringExpression>, <IntegerExpression>
       [, <IntegerExpression>])

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The SUBSTRING function extracts a portion of a string value. The second FROM parameter is the character position at which the extracted string starts within the original string. The index for the FROM parameter is based on the first character in the source value being 1.

The FOR parameter is optional, and specifies the length of the extracted string. If the FOR parameter is omitted, the extracted string will be equal to the portion of the string starting at the position specified by the FROM parameter to the end of the string.

Examples

```
SELECT SUBSTRING(CustomerID, 8 FOR 2) AS Category
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Argument Separator	The use of a comma separator for the function arguments is an ElevateDB extension.
SIMILAR	ElevateDB does not support the SIMILAR syntax for regular expression matching.

10.19 TRIM

Removes repetitions of a specified character from the left, right, or both sides of a string.

Syntax

```
TRIM(LEADING|TRAILING|BOTH <CharacterExpression>
     FROM <StringExpression>)
TRIM(LEADING|TRAILING|BOTH <CharacterExpression>,
     <StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The TRIM function removes any repetitions of the specified trailing or leading character, or both, from a string. The first parameter indicates the position of the character to be deleted, and has one of the following values:

Keyword	Description
LEADING	Deletes the character in the leading portion of the string.
TRAILING	Deletes the character in the trailing portion of the string.
BOTH	Deletes the character at both ends of the string.

The character parameter specifies the character to be deleted.

The FROM parameter specifies the string value to trim.

Examples

```
SELECT TRIM(TRAILING ' ' FROM CustomerID) AS CustomerID
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Argument Separator	The use of a comma separator for the function arguments is an ElevateDB extension.

10.20 UPPER

Forces a string to upper-case.

Syntax

```
UPPER(<StringExpression>)

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The UPPER function converts all characters in a string value to upper-case. The collation of the input value is used to determine how the upper-case operation is performed.

Examples

```
SELECT UPPER(CustomerID) AS CustomerID
FROM Customers
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

10.21 QUOTEDSTR

Escapes quotes in a string.

Syntax

```
QUOTEDSTR(<StringExpression> [USING <CharacterExpression>])
QUOTEDSTR(<StringExpression>[, <CharacterExpression>])

<StringExpression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
Same as input
```

Usage

The QUOTEDSTR function escapes quotes in a string so that it can be used as a literal constant without causing parsing errors. The default character that is escaped is the single quote ('), but you may also specify a character like the double quote ("), if necessary.

Examples

```
-- The following script accepts a table
-- name and returns a result set that includes
-- all of the rows in the specified table.

SCRIPT (IN TableName VARCHAR)
BEGIN
   DECLARE ResultCursor CURSOR WITH RETURN FOR ResultStmt;

   IF (COALESCE(TableName,'') <> '') THEN
      PREPARE ResultStmt FROM 'SELECT * FROM '+QUOTEDSTR(TableName USING '"');

      OPEN ResultCursor;
   END IF;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

This page intentionally left blank

Chapter 11
Array Functions

11.1 Introduction

Array functions are used to manipulate array types in ElevateDB SQL expressions. This section of the manual details the available array functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

11.2 CARDINALITY

Returns the cardinality of an array.

Syntax

```
CARDINALITY(<ArrayExpression>)

<ArrayExpression> =

Any array of type:
CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
INTEGER
```

Usage

The CARDINALITY function returns the index (1-based) of the highest defined array element in an array. The highest defined array element is the highest array element that has been referenced in the array. For example, if you define an array with a maximum cardinality of 10, and reference the fifth (5) element in the array, then the CARDINALITY function will return 5 for the array.

> **Note**
> ElevateDB currently only supports the use of arrays in SQL/PSM routines and does not support arrays as column types.

Examples

```
-- This script loops through the Customer table and
-- populates an array with the CustNo column value
-- for each row

SCRIPT
BEGIN
   DECLARE Done BOOLEAN DEFAULT False;
   DECLARE TotalRows INTEGER DEFAULT 0;
   DECLARE CustCursor CURSOR FOR CustStmt;
   DECLARE CustArray INTEGER ARRAY [56];

   SET LOG MESSAGE TO CAST(CARDINALITY(CustArray) AS VARCHAR);

   PREPARE CustStmt FROM 'SELECT CustNo,
                         Company
                         FROM Customer';

   OPEN CustCursor;

   WHILE (NOT EOF(CustCursor)) DO
      SET TotalRows=TotalRows+1;
      FETCH NEXT FROM CustCursor INTO CustArray[TotalRows];
      SET PROGRESS TO TRUNC((TotalRows/ROWCOUNT(CustCursor))*100);
   END WHILE;

   SET LOG MESSAGE TO CAST(CARDINALITY(CustArray) AS VARCHAR);

   CLOSE CustCursor;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Referenced Elements	ElevateDB sets the cardinality of an array on any array access to a specific index, not just the assignment of a value to a specific index.

This page intentionally left blank

Chapter 12
Date and Time Functions

12.1 Introduction

Date and time functions are used to convert and manipulate date and time types in ElevateDB SQL expressions. This section of the manual details the available date and time functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

12.2 CURRENT_DATE

Returns the current date.

Syntax

```
CURRENT_DATE([UTC])
```

Returns

```
DATE
```

Usage

The CURRENT_DATE function returns the current date. Use the UTC designation to indicate that the date returned should be a UTC (Coordinated Universal Time) value.

Note
When this function is used with the ElevateDB Server, it will always return the current date for the ElevateDB Server machine.

Examples

```
SELECT *
FROM Transactions
WHERE CAST(TransTimeStamp AS DATE) = CURRENT_DATE
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
UTC Parameter	The UTC parameter is an ElevateDB extension.

12.3 CURRENT_TIME

Returns the current time.

Syntax

```
CURRENT_TIME([UTC])
```

Returns

```
TIME
```

Usage

The CURRENT_TIME function returns the current time. Use the UTC designation to indicate that the time returned should be a UTC (Coordinated Universal Time) value.

Note
When this function is used with the ElevateDB Server, it will always return the current time for the ElevateDB Server machine.

Examples

```
INSERT INTO Logins
VALUES (CURRENT_USER(), CURENT_DATE(), CURRENT_TIME())
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
UTC Parameter	The UTC parameter is an ElevateDB extension.

12.4 CURRENT_TIMESTAMP

Returns the current timestamp.

Syntax

```
CURRENT_TIMESTAMP([UTC])
```

Returns

```
TIMESTAMP
```

Usage

The CURRENT_TIMESTAMP function returns the current date and time. Use the UTC designation to indicate that the date and time returned should be a UTC (Coordinated Universal Time) value.

Note
When this function is used with the ElevateDB Server, it will always return the current date/time for the ElevateDB Server machine.

Examples

```
INSERT INTO Logins
VALUES (CURRENT_USER(), CURENT_TIMESTAMP())
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
UTC Parameter	The UTC parameter is an ElevateDB extension.

12.5 EXTRACT

Extracts the a portion of a date, time, or timestamp value.

Syntax

```
EXTRACT(<ValueToExract> FROM <DateTimeExpression>)
EXTRACT(<ValueToExract>, <DateTimeExpression>)

<ValueToExtract> =

YEAR
MONTH
WEEK
DAYOFWEEK
DAYOFYEAR
DAY
HOUR
MINUTE
SECOND
MSECOND

<DateTimeExpression> =

DATE
TIME
TIMESTAMP
```

Returns

```
INTEGER
```

Usage

The EXTRACT function extracts a designated portion of a date, time, or timestamp value and returns it. The following table details which portions can be extracted from which types:

Portion	Types

YEAR	DATE TIMESTAMP
MONTH	DATE TIMESTAMP
WEEK	DATE TIMESTAMP
DAYOFWEEK	DATE TIMESTAMP
DAYOFYEAR	DATE TIMESTAMP
DAY	DATE TIMESTAMP
HOUR	TIME TIMESTAMP
MINUTE	TIME TIMESTAMP
SECOND	TIME TIMESTAMP
MSECOND	TIME TIMESTAMP

Note
All day and week values returned from EXTRACT follow the ISO 8601 standard for day and week numbers.

Examples

```
SELECT *
FROM Transactions
WHERE EXTRACT(YEAR FROM TransDateTime) = 2006
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

Chapter 13
Interval Functions

13.1 Introduction

Interval functions are used to convert and manipulate interval types in ElevateDB SQL expressions. This section of the manual details the available interval functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

13.2 ABS

Converts an interval to its absolute value.

Syntax

```
ABS(<IntervalExpression>)

<IntervalExpression> =

Type of:

INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND
```

Returns

```
Same as Input
```

Usage

The ABS function converts an interval value to its absolute, or non-negative value.

Examples

```
SELECT ABS(StartDate - EndDate) AS NumDays
FROM Reservations
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

13.3 EXTRACT

Extracts the a portion of an interval value.

Syntax

```
EXTRACT(<ValueToExract> FROM <IntervalExpression>)
EXTRACT(<ValueToExract>, <IntervalExpression>)

<ValueToExtract> =

YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
MSECOND

<IntervalExpression> =

Type of:

INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND
```

Returns

```
INTEGER
```

Usage

The EXTRACT function extracts a designated portion of an interval value and returns it. The following table details which portions can be extracted from which types:

Portion	Types

YEAR	INTERVAL YEAR INTERVAL YEAR TO MONTH
MONTH	INTERVAL YEAR TO MONTH INTERVAL MONTH
DAY	INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND
HOUR	INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND
MINUTE	INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND
SECOND	INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND
MSECOND	INTERVAL DAY TO MSECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND

Examples

```
SELECT EXTRACT(HOUR FROM (EndTime - StartTime)) AS NumHours
FROM TimeCards
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

This page intentionally left blank

Chapter 14
Conversion Functions

14.1 Introduction

Conversion functions are used to convert values from one type to another or otherwise return a value based upon various conditions in ElevateDB SQL expressions. This section of the manual details the available conversion functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

14.2 CAST

Converts a given value to a different, but compatible, type.

Syntax

```
CAST(<Expression> AS <DataType>
[DATE FORMAT <DateFormat>]
[TIME FORMAT <TimeFormat> [AM LITERAL <AMLiteral> PM LITERAL <PMLiteral>]]
[DECIMAL CHAR <DecimalChar>]
[BOOLEAN TRUE LITERAL <TrueLiteral> FALSE LITERAL <FalseLiteral>]
)
CAST(<Expression>, <DataType>
[DATE FORMAT <DateFormat>]
[TIME FORMAT <TimeFormat> [AM LITERAL <AMLiteral> PM LITERAL <PMLiteral>]]
[DECIMAL CHAR <DecimalChar>]
[BOOLEAN TRUE LITERAL <TrueLiteral> FALSE LITERAL <FalseLiteral>])

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

```
<DataType> =

CHARACTER|CHAR [(<Length>]) [<CollationName>]
CHARACTER VARYING|VARCHAR [(<Length>]) [<CollationName>]
GUID
BYTE [(<LengthInBytes>])
BYTE VARYING|VARBYTE [(<LengthInBytes>])
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB [<CollationName>]
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT [(<Precision>,<Scale>)]
DECIMAL|NUMERIC [(<Precision>,<Scale>)]
DATE
TIME
TIMESTAMP
INTERVAL YEAR [TO MONTH]
INTERVAL MONTH
INTERVAL DAY [TO HOUR|MINUTE|SECOND|MSECOND]
INTERVAL HOUR [TO MINUTE|SECOND|MSECOND]
INTERVAL MINUTE [TO SECOND|MSECOND]
INTERVAL SECOND [TO MSECOND]
INTERVAL MSECOND

<DateFormat> =

YYYY or YY = Year digits
MM or M = Month digits
DD or D = Day digits
Any other character = literal

<TimeFormat> =

HH or H = Hours digits
MM or M = Minutes digits
SS or S = Seconds digits
ZZZ or Z = Milliseconds digits
N = AM/PM literal
Any other character = literal
```

Returns

```
Input converted to specified data type, provided
that there aren't any numeric overflow errors
```

Usage

The CAST function converts a given value to a different, but compatible, type. The following table details the various types and which types they can be converted to:

Source Type	Destination Types

CHARACTER|CHAR	CHARACTER|CHAR CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC DATE TIME TIMESTAMP INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
CHARACTER VARYING|VARCHAR	CHARACTER|CHAR CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC DATE TIME TIMESTAMP INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE

	INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
GUID	CHARACTER|CHAR CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB
BYTE	CHARACTER|CHAR CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC DATE TIME TIMESTAMP INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
BYTE VARYING|VARBYTE	CHARACTER|CHAR

	CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC DATE TIME TIMESTAMP INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
BINARY LARGE OBJECT|BLOB	CHARACTER|CHAR CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC DATE TIME TIMESTAMP INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND

	INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
CHARACTER LARGE OBJECT|CLOB	CHARACTER|CHAR CHARACTER VARYING|VARCHAR GUID BYTE BYTE VARYING|VARBYTE BINARY LARGE OBJECT|BLOB CHARACTER LARGE OBJECT|CLOB BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC DATE TIME TIMESTAMP INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
BOOLEAN|BOOL	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC

SMALLINT	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC INTERVAL YEAR INTERVAL MONTH INTERVAL DAY INTERVAL HOUR INTERVAL MINUTE INTERVAL SECOND INTERVAL MSECOND
INTEGER|INT	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC INTERVAL YEAR INTERVAL MONTH INTERVAL DAY INTERVAL HOUR INTERVAL MINUTE INTERVAL SECOND INTERVAL MSECOND
BIGINT	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC INTERVAL YEAR INTERVAL MONTH INTERVAL DAY INTERVAL HOUR INTERVAL MINUTE INTERVAL SECOND INTERVAL MSECOND
FLOAT	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE

	BYTE VARYING|VARBYTE BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC
DECIMAL|NUMERIC	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE BOOLEAN|BOOL SMALLINT INTEGER|INT BIGINT FLOAT DECIMAL|NUMERIC
DATE	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE DATE TIMESTAMP
TIME	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE TIME TIMESTAMP
TIMESTAMP	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE DATE TIME TIMESTAMP
INTERVAL YEAR	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE SMALLINT INTEGER|INT BIGINT INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH
INTERVAL YEAR TO MONTH	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH

INTERVAL MONTH	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE SMALLINT INTEGER|INT BIGINT INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH
INTERVAL DAY	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE SMALLINT INTEGER|INT BIGINT INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL DAY TO HOUR	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL DAY TO MINUTE	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE

	BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL DAY TO SECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL DAY TO MSECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND

INTERVAL HOUR	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE SMALLINT INTEGER|INT BIGINT INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL HOUR TO MINUTE	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL HOUR TO SECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE

	INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL HOUR TO MSECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL MINUTE	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE SMALLINT INTEGER|INT BIGINT INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL MINUTE TO SECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND

	INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL MINUTE TO MSECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL SECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR BYTE BYTE VARYING|VARBYTE SMALLINT INTEGER|INT BIGINT INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL DAY TO MSECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL HOUR TO MSECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL MINUTE TO MSECOND INTERVAL SECOND INTERVAL SECOND TO MSECOND INTERVAL MSECOND
INTERVAL SECOND TO MSECOND	CHARACTER|CHAR CHARACTER VARYING|VARCHAR

<table>
<tr><td></td><td>BYTE
BYTE VARYING|VARBYTE
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND</td></tr>
<tr><td>INTERVAL MSECOND</td><td>CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
BYTE
BYTE VARYING|VARBYTE
SMALLINT
INTEGER|INT
BIGINT
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND</td></tr>
</table>

The CAST function can also be used to convert a string value from one collation to the other without changing the actual base string type.

When casting a value to and from a string type such as CHAR or VARCHAR, you can use the formatting extensions to specify how the resultant string or value will look once the conversion takes place. The following tables shows which source types can use which formatting extensions:

Data Type	Formatting Extension

BOOLEAN	BOOLEAN TRUE LITERAL FALSE LITERAL
SMALLINT	BOOLEAN TRUE LITERAL FALSE LITERAL (0 is False, > 0 is True)
INTEGER	BOOLEAN TRUE LITERAL FALSE LITERAL (0 is False, > 0 is True)
BIGINT	BOOLEAN TRUE LITERAL FALSE LITERAL (0 is False, > 0 is True)
FLOAT	DECIMAL CHAR and BOOLEAN TRUE LITERAL FALSE LITERAL (0 is False, > 0 is True)
DECIMAL|NUMERIC	DECIMAL CHAR and BOOLEAN TRUE LITERAL FALSE LITERAL (0 is False, > 0 is True)
DATE	DATE FORMAT
TIME	TIME FORMAT
TIMESTAMP	DATE FORMAT and TIME FORMAT

Examples

```
SELECT 'Date/Time: '+CAST(TransDateTime AS VARCHAR(25)) +
'User: '+TransUser AS TransInfo
FROM Transactions

SELECT CAST(LastInvoiceDate AS VARCHAR(24)
DATE FORMAT 'm/dd/yyyy' TIME FORMAT 'h:mm:ss n')
FROM customer
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
DATE FORMAT	The DATE FORMAT clause is an ElevateDB extension.
TIME FORMAT	The TIME FORMAT clause is an ElevateDB extension.
DECIMAL CHAR	The DECIMAL CHAR clause is an ElevateDB extension.
BOOLEAN	The BOOLEAN clause is an ElevateDB extension.

14.3 COALESCE

Returns the first non-NULL value from a list of expressions.

Syntax

```
COALESCE(<Expression>, [<Expression>])

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The COALESCE function returns the first non-NULL value from a list of expressions. There is no limit to the number of expressions that can be passed as parameters, and the expressions can be of any type.

This function is useful for converting NULL numeric column values into zeros for display purposes.

Examples

```
SELECT TransDateTime,
COALESCE(Amount, 0.00) AS Amount
FROM Transactions
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

14.4 IF

Performs IF..ELSE type of inline expression handling.

Syntax

```
IF(<BooleanExpression> THEN <Expression> ELSE <Expression>)
IF(<BooleanExpression>, <Expression>, <Expression>)

<BooleanExpression> =

Type of:

BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The IF function performs inline IF..ELSE boolean expression handling. Both result expressions must be type-compatible. Use the CAST function to ensure that both expressions are type-compatible.

Examples

```
SELECT CAST(TransDateTime AS VARCHAR(25)) + ':' +
IF(Amount < 0 THEN
   '(' + CAST(Amount AS VARCHAR(20)) + ')'
ELSE
   CAST(Amount AS VARCHAR(20)))
AS Entry
FROM Transactions
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

14.5 IFNULL

Performs IF..ELSE type of inline expression handling specifically for NULL values.

Syntax

```
IFNULL(<Expression> THEN <Expression> ELSE <Expression>)
IFNULL(<Expression>, <Expression>, <Expression>)

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The IF function performs inline IF..ELSE boolean expression handling specifically for NULL values. Both result expressions must be type-compatible. Use the CAST function to ensure that both expressions are type-compatible.

Examples

```
SELECT TransDateTime,
IFNULL(Amount THEN
   0.00
ELSE
   Amount)
AS Amount
FROM Transactions
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

14.6 NULLIF

Returns a NULL if two expressions are equivalent.

Syntax

```
NULLIF(<Expression>, <Expression>)

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
NULL if both expressions are equal, otherwise same as input
```

Usage

The NULLIF function returns a NULL if two expressions are equal. Both expressions must be type-compatible. Use the CAST function to ensure that both expressions are type-compatible.

Examples

```
SELECT TransDateTime,
NULLIF(Amount, -1) AS Amount
FROM Transactions
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

14.7 CASE

Evaluates a series of boolean expressions and returns the matching result value for the first boolean expression that evaluates to True.

Syntax

```
CASE
WHEN <BooleanExpression> THEN <Expression>
[WHEN <BooleanExpression> THEN <Expression>]
[ELSE] <Expression>
END

Short-hand syntax:

CASE <Expression>
WHEN <Expression> THEN <Expression>
[WHEN <Expression> THEN <Expression>]
[ELSE] <Expression>
END

<BooleanExpression> =

Type of:

BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
```

```
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The CASE expression is not actually a function, but it behaves like one so it is included with the functions. The CASE expression can be used in with two different syntaxes, one being the normal syntax while the other being a short-hand syntax. The normal syntax evaluates a series of boolean expressions and returns the matching result expression associated with the first boolean expression that evaluates to True. The primary difference between the short-hand syntax and the normal syntax is the inclusion of the expression directly after the CASE keyword itself. It is used as the comparison value for every WHEN expression. All WHEN expressions must be type-compatible with this expression, unlike the normal syntax which requires boolean expressions. The rest of the short-hand syntax is the same as the normal syntax.

Examples

```
SELECT CAST(TransDateTime AS VARCHAR(25)) + ':' +
CASE
   WHEN Amount < 0 THEN
      '(' + CAST(Amount AS VARCHAR(20)) + ')'
   ELSE
      CAST(Amount AS VARCHAR(20))
END
AS Entry
FROM Transactions
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

Chapter 15
Aggregate Functions

15.1 Introduction

Aggregate functions are used to perform specific calculations across all selected rows or sets of selected rows in an ElevateDB SELECT statement. This section of the manual details the available aggregate functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

15.2 AVG

Returns the average of a given numeric, date and time, or interval expression for all selected rows.

Syntax

```
AVG([DISTINCT] <NumericExpression>|<DateTimeExpression>|
<IntervalExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC

<IntervalExpression> =

Type of:

INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The AVG function returns the average of a given numeric, date and time, or interval expression for all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the averaged expression is NULL, it is excluded from the average

calculation.

Use the DISTINCT clause to specify that the average calculation will only use distinct values when calculating the result.

Examples

```
SELECT AVG(Amount) AS AvgAmount
FROM Transactions

SELECT CAST(TransDateTime AS DATE) AS TransDate,
AVG(Amount) AS AvgAmount
FROM Transactions
GROUP BY TransDate
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

15.3 COUNT

Returns the count of a given expression for all selected rows, or the count of all selected rows.

Syntax

```
COUNT(*|[DISTINCT] <Expression>)

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
INTEGER
```

Usage

The COUNT function returns the count of a given expression for all selected rows, or the count of all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the counted expression is NULL, it is excluded from the count calculation. Use of the asterisk (*) instead of an expression indicates that you want the function to count all selected rows, irrespective of any given expression.

Use the DISTINCT clause to specify that the count calculation will only use distinct values when calculating the result.

Examples

```
SELECT CAST(TransDateTime AS DATE) AS TransDate,
COUNT(*) AS NumTransactions
FROM Transactions
GROUP BY TransDate
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

15.4 MAX

Returns the maximum value of a given expression for all selected rows.

Syntax

```
MAX(<Expression>)

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The MAX function returns the maximum value of a given expression for all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the expression is NULL, it is excluded from the maximum calculation.

Examples

```
SELECT MAX(CAST(TransDateTime AS DATE)) AS TransDate
FROM Transactions
WHERE Amount > 1000.00
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

15.5 MIN

Returns the minimum value of a given expression for all selected rows.

Syntax

```
MIN(<Expression>)

<Expression> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
BYTE
BYTE VARYING|VARBYTE
BINARY LARGE OBJECT|BLOB
CHARACTER LARGE OBJECT|CLOB
BOOLEAN|BOOL
SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
DATE
TIME
TIMESTAMP
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input
```

Usage

The MIN function returns the minimum value of a given expression for all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the expression is NULL, it is excluded from the minimum calculation.

Examples

```
SELECT MIN(CAST(TransDateTime AS DATE)) AS TransDate
FROM Transactions
WHERE Amount > 1000.00
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

15.6 RUNSUM

Returns the running sum of a given numeric or interval expression for all selected rows.

Syntax

```
RUNSUM([DISTINCT] <NumericExpression>|<IntervalExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC

<IntervalExpression> =

Type of:

INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input, except for the following types:

A SMALLINT expression is promoted to an INTEGER
An INTEGER expression is promoted to a BIGINT

This is done to prevent numeric overflows
```

Usage

The RUNSUM function returns the running sum of a given numeric or interval expression for all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the numeric or interval expression is NULL, it is excluded from the running sum calculation.

Use the DISTINCT clause to specify that the running sum calculation will only use distinct values when calculating the result.

Examples

```
SELECT Month,
RUNSUM(Amount) AS RunningTotal
FROM TransactionHistory
GROUP BY Month
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

15.7 STDDEV

Returns the standard deviation of a given numeric expression for all selected rows.

Syntax

```
STDDEV([DISTINCT] <NumericExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC
```

Returns

```
FLOAT
```

Usage

The STDDEV function returns the standard deviation of a given numeric expression for all selected rows. The standard deviation is the distance from the mean for a set of values. If all of the values are equal, then the standard deviation is zero. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the numeric expression is NULL, it is excluded from the standard deviation calculation.

Use the DISTINCT clause to specify that the standard deviation calculation will only use distinct values when calculating the result.

Examples

```
SELECT STDDEV(TestScore) AS Deviation
FROM Scores
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

15.8 SUM

Returns the sum of a given numeric or interval expression for all selected rows.

Syntax

```
SUM([DISTINCT] <NumericExpression>|<IntervalExpression>)

<NumericExpression> =

Type of:

SMALLINT
INTEGER|INT
BIGINT
FLOAT
DECIMAL|NUMERIC

<IntervalExpression> =

Type of:

INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MSECOND
INTERVAL HOUR
INTERVAL HOUR TO MINUTE
INTERVAL HOUR TO SECOND
INTERVAL HOUR TO MSECOND
INTERVAL MINUTE
INTERVAL MINUTE TO SECOND
INTERVAL MINUTE TO MSECOND
INTERVAL SECOND
INTERVAL SECOND TO MSECOND
INTERVAL MSECOND
```

Returns

```
Same as input, except for the following types:

A SMALLINT expression is promoted to an INTEGER
An INTEGER expression is promoted to a BIGINT

This is done to prevent numeric overflows
```

Usage

The SUM function returns the sum of a given numeric or interval expression for all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the numeric or interval expression is NULL, it is excluded from the sum calculation.

Use the DISTINCT clause to specify that the sum calculation will only use distinct values when calculating the result.

Examples

```
SELECT SUM(Amount) AS TotalAmount
FROM Transactions

SELECT CAST(TransDateTime AS DATE) AS TransDate,
SUM(Amount) AS TotalAmount
FROM Transactions
GROUP BY TransDate
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

15.9 LIST

Returns the concatenation of a given string expression for all selected rows.

Syntax

```
LIST([DISTINCT] [ORDERED] <StringExpression> [USING <Delimiter>])
LIST([DISTINCT] [ORDERED] <StringExpression> [, <Delimiter>])

<StringExpression> =

Type of:

CHAR
VARCHAR
CLOB

<Delimiter> =

Type of:

CHAR
VARCHAR
CLOB
```

Returns

```
CLOB
```

Usage

The LIST function returns the concatenation of a given string expression for all selected rows. The selected rows can be grouped into logical sub-sets by using the GROUP BY clause of the SELECT statement. Any time the string expression is NULL, it is excluded from the concatenation operation. If the delimiter expression is not specified, then it defaults to using a comma (,).

Use the DISTINCT clause to specify that the list concatenation will only use distinct values when creating the result.

The ORDERED clause indicates that you wish to have the listed values in the result sorted in ascending order.

Examples

```
SELECT LIST(Company) AS CompanyNames
FROM Customer
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

Chapter 16
Boolean Functions

16.1 Introduction

Boolean functions are used to test for a certain condition and return a TRUE or FALSE value. This section of the manual details the available boolean functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

16.2 EXISTS

Returns whether or not any rows exist in a given subquery.

Syntax

```
EXISTS(<QueryExpression>)

<QueryExpression> = SELECT statement
```

Returns

```
BOOLEAN
```

Usage

The EXISTS function returns the TRUE if a given sub-query returns any rows, or FALSE if the sub-query does not return any rows. EXISTS is useful in situations where you simply want to know if any rows are present for a given set of conditions, which would be expressed via the WHERE clause of the sub-query.

Examples

```
SELECT *
FROM Customers
WHERE EXISTS(SELECT * FROM Orders
             WHERE Orders.CustomerNo=Customers.CustomerNo)
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
None	

Chapter 17
SQL/PSM Functions

17.1 Introduction

SQL/PSM functions are used strictly within an SQL/PSM routine such as an ElevateDB function, procedure, trigger, or job. This section of the manual details the available SQL/PSM functions in ElevateDB.

Notation

The notation used in the syntax section for each function is as follows:

Notation	Description
<Element>	Specifies an element of the statement that may be expanded upon further on in the syntax section
<Element> =	Describes an element specified earlier in the syntax section
[Optional Element]	Describes an optional element by enclosing it in square brackets []
Element|Element	Describes multiple elements, of which one and only one may be used in the syntax

17.2 ABORTED

Returns whether or not a the current execution has been aborted as a response to a progress update.

Syntax

```
ABORTED()
```

Returns

```
BOOLEAN
```

Usage

The ABORTED function returns whether or not the current execution has been aborted as the result of a progress update executed via the SET PROGRESS statement.

Examples

```
-- This procedure uses a SET PROGRESS
-- statement to display progress during its
-- execution and uses the ABORTED function
-- to abort the execution if the application
-- requests it

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';
   DECLARE TotalRows INTEGER DEFAULT 0;
   DECLARE NumRows INTEGER DEFAULT 0;

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;
   SET TotalRows=ROWCOUNT(CustCursor);

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;

      WHILE (NOT (EOF(CustCursor) OR ABORTED)) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
         SET NumRows=NumRows+1;
         SET PROGRESS TO TRUNC(((NumRows/TotalRows)*100));
      END WHILE;
```

```
      IF (NOT ABORTED) THEN
         COMMIT;
      ELSE
         ROLLBACK;
      END IF;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.3 BOF

Returns whether or not a result set cursor is at the beginning of the result set.

Syntax

```
BOF(<CursorName>)

<CursorName> =

Previously-opened result set cursor
```

Returns

```
BOOLEAN
```

Usage

The BOF function returns whether or not a result set cursor is at the beginning of the result set. The BOF function only returns True once an attempt is made to navigate prior to the first row via the FETCH statement, or if the result set is empty and contains no rows.

When a result set cursor is first opened via the OPEN statement, the cursor is always positioned so that the BOF function will return True. If a result set is empty, then both the BOF and the EOF functions will return True.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH LAST FROM CustCursor ('State') INTO State;
```

```
      WHILE NOT BOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH PRIOR FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.4 EOF

Returns whether or not a result set cursor is at the end of the result set.

Syntax

```
EOF(<CursorName>)

<CursorName> =

Previously-opened result set cursor
```

Returns

```
BOOLEAN
```

Usage

The EOF function returns whether or not a result set cursor is at the end of the result set. The EOF function only returns True once an attempt is made to navigate past the last row via the FETCH statement, or if the result set is empty and contains no rows.

When a result set cursor is first opened via the OPEN statement, the cursor is always positioned so that the EOF function will return False unless the result set is empty, in which case both the EOF and the BOF functions will return True.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer';

   OPEN CustCursor;

   START TRANSACTION ON TABLES 'Customer';
   BEGIN

      FETCH FIRST FROM CustCursor ('State') INTO State;
```

```
      WHILE NOT EOF(CustCursor) DO
         IF (State='FL') THEN
            UPDATE CustCursor SET 'State'='NY';
         END IF;
         FETCH NEXT FROM CustCursor ('State') INTO State;
      END WHILE;

      COMMIT;

   EXCEPTION
      ROLLBACK;
   END;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.5 ERRORCODE

Returns the current error code.

Syntax

```
ERRORCODE()
```

Returns

```
INTEGER
```

Usage

The ERRORCODE function returns the current error code. This function can only be called from within an EXCEPTION block or from within an error trigger. See the CREATE TRIGGER topic for more information on error triggers.

Examples

```
-- This procedure uses an exception
-- block to handle any exceptions while
-- executing a CREATE TABLE statement

CREATE PROCEDURE CreateTestTable()
BEGIN
   DECLARE stmt STATEMENT;

   PREPARE stmt FROM 'CREATE TEMPORARY TABLE "TestTable"
                     (
                     "FirstColumn" INTEGER,
                     "SecondColumn" VARCHAR(30),
                     "ThirdColumn" CLOB,
                     PRIMARY KEY ("FirstColumn")
                     )

                     DESCRIPTION ''Test Table''';

   EXECUTE stmt;
EXCEPTION
   IF ERRORCODE()=700 THEN
      RAISE ERROR CODE 10000 MESSAGE 'Syntax error';
   ELSE
      RAISE ERROR CODE 10000 MESSAGE 'Unexpected error - ' +
           ERRORMSG();
   END IF;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.6 ERRORMSG

Returns the current error message.

Syntax

```
ERRORMSG()
```

Returns

```
VARCHAR
```

Usage

The ERRORMSG function returns the current error message. This function can only be called from within an EXCEPTION block or from within an error trigger. See the CREATE TRIGGER topic for more information on error triggers.

Examples

```
-- This procedure uses an exception
-- block to handle any exceptions while
-- executing a CREATE TABLE statement

CREATE PROCEDURE CreateTestTable()
BEGIN
   DECLARE stmt STATEMENT;

   PREPARE stmt FROM 'CREATE TEMPORARY TABLE "TestTable"
                     (
                     "FirstColumn" INTEGER,
                     "SecondColumn" VARCHAR(30),
                     "ThirdColumn" CLOB,
                     PRIMARY KEY ("FirstColumn")
                     )

                     DESCRIPTION ''Test Table''';

   EXECUTE stmt;
EXCEPTION
   IF ERRORCODE()=700 THEN
      RAISE ERROR CODE 10000 MESSAGE 'Syntax error';
   ELSE
      RAISE ERROR CODE 10000 MESSAGE 'Unexpected error - ' +
           ERRORMSG();
   END IF;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.7 ROWCOUNT

Returns the row count for a result set cursor.

Syntax

```
ROWCOUNT(<CursorName>)

<CursorName> =

Previously-opened result set cursor
```

Returns

```
INTEGER
```

Usage

The ROWCOUNT function returns the row count for a result set cursor. If the result set cursor has not been opened yet by using the OPEN statement, then calling this function will result in an error.

Examples

```
-- This procedure checks to see if the
-- specified State exists in the States lookup
-- table and inserts it if it isn't

CREATE PROCEDURE LookupState(IN State CHAR(2) COLLATE ANSI_CI)
BEGIN
   DECLARE StateCursor SENSITIVE CURSOR FOR Stmt;

   PREPARE Stmt FROM 'SELECT * FROM States WHERE State = ?';

   OPEN StateCursor USING State;

   IF (ROWCOUNT(StateCursor) = 0) THEN
      INSERT INTO StateCursor VALUES (State);
   END IF;

   CLOSE StateCursor;

END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.8 ROWSAFFECTED

Returns the number of rows affected by the execution of a statement.

Syntax

```
ROWSAFFECTED(<StatementName>)

<StatementName> =

Previously-executed statement
```

Returns

```
INTEGER
```

Usage

The ROWSAFFECTED function returns the number of rows affected by a given statement.

Examples

```
-- This procedure updates all Customers
-- who have purchased a product last year and
-- flags them to receive a mailer

CREATE PROCEDURE UpdateMailer(OUT NumCustomers INTEGER)
BEGIN
   DECLARE UpdateStmt STATEMENT;

   EXECUTE IMMEDIATE 'UPDATE Customers SET Mailer = False';

   PREPARE UpdateStmt FROM 'UPDATE Customers SET Mailer = True ' +
      'WHERE EXTRACT(YEAR FROM LastPurchased) = ' +
      'EXTRACT(YEAR FROM CURRENT_DATE()) - 1';

   EXECUTE UpdateStmt;

   SET NumCustomers = ROWSAFFECTED(UpdateStmt);

   UNPREPARE UpdateStmt;

END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.9 SENSITIVE

Returns whether or not a result set cursor is sensitive to changes by other sessions.

Syntax

```
SENSITIVE(<CursorName>)

<CursorName> =

Previously-opened result set cursor
```

Returns

```
BOOLEAN
```

Usage

The SENSITIVE function returns whether or not a result set cursor is sensitive to changes by other sessions. If the result set cursor has not been opened yet by using the OPEN statement, then calling this function will result in an error. Please see the Result Set Cursor Sensitivity topic for more information.

Examples

```
-- This procedure uses an IF statement
-- to conditionally test if the State column
-- is equal to 'FL', and if so, to change it
-- to 'NY'

-- The whole update process is wrapped inside
-- of a transaction start..commit/rollback block

-- An error is raised if the result cursor generated
-- is not sensitive to changes by other sessions

CREATE PROCEDURE UpdateState()
BEGIN
   DECLARE CustCursor CURSOR WITH RETURN FOR Stmt;
   DECLARE State CHAR(2) DEFAULT '';

   PREPARE Stmt FROM 'SELECT * FROM Customer ORDER BY CustomerID';

   OPEN CustCursor;

   IF (NOT SENSITIVE(CustCursor)) THEN
      CLOSE CustCursor;
      RAISE ERROR CODE 12000 MESSAGE 'Result set cursor is insensitive';
   END IF;
```

```
    START TRANSACTION ON TABLES 'Customer';
    BEGIN

       FETCH FIRST FROM CustCursor ('State') INTO State;

       WHILE NOT EOF(CustCursor) DO
          IF (State='FL') THEN
             UPDATE CustCursor SET 'State'='NY';
          END IF;
          FETCH NEXT FROM CustCursor ('State') INTO State;
       END WHILE;

       COMMIT;

    EXCEPTION
       ROLLBACK;
    END;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.10 LOADINGUPDATES

Returns whether or not a trigger is being executed during the execution of a LOAD UPDATES statement.

Syntax

```
LOADINGUPDATES()
```

Returns

```
BOOLEAN
```

Usage

The LOADINGUPDATES function returns whether or not the current trigger is being executed during the execution of a LOAD UPDATES statement. This is useful for situations where you only want to log errors for insert, update, and delete operations that occur during a LOAD UPDATES statement, and have defined error triggers to handle this. Please see the CREATE TRIGGER statement for more information on creating error triggers.

Examples

```
-- This trigger logs any insert errors that
-- occur during a LOAD UPDATES for
-- the Customer table into a table called
-- LoadErrors

CREATE TRIGGER "LogInsertError" ERROR INSERT ON "customer"
WHEN LOADINGUPDATES()
BEGIN
   DECLARE ErrorData VARCHAR DEFAULT '';

   SET ErrorData = 'Cust #: ' + CAST(NEWROW.CustNo AS VARCHAR);
   SET ErrorData = ErrorData + 'Name: ' + NEWROW.Company;
   SET ErrorData = ErrorData + 'Error #: ' + CAST(ERRORCODE() AS VARCHAR);
   SET ErrorData = ErrorData + 'Error Msg: ' + ERRORMSG();

   EXECUTE IMMEDIATE 'INSERT INTO LoadErrors
                      (''Customer'',''INSERT'',''' + ErrorData + '''';
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details

Extension	This function is an ElevateDB extension.

17.11 INTRANSACTION

Returns whether or not the current database, or a specific table in the current database, is currently in a transaction started by the START TRANSACTION statement.

Syntax

```
INTRANSACTION([<TableName>])

<TableName> =

Type of:

CHARACTER|CHAR
CHARACTER VARYING|VARCHAR
GUID
CHARACTER LARGE OBJECT|CLOB
```

Returns

```
BOOLEAN
```

Usage

The INTRANSACTION function returns the transaction status of the current database, or a table in the current database. This is useful in situations such as triggers where you may be updating tables that are already part of a transaction. Please see the CREATE TRIGGER statement for more information on creating triggers.

Examples

```
-- This trigger checks to see if the
-- current table is involved in a transaction
-- and starts a transaction, if necessary.

CREATE TRIGGER "CascadeChanges" AFTER UPDATE ON "customer"
BEGIN
   DECLARE LocalTrans BOOLEAN DEFAULT FALSE;

   IF (NEWROW.CustNo <> OLDROW.CustNo) THEN
      IF NOT INTRANSACTION('customer') THEN
         START TRANSACTION ON TABLES 'customer';
         SET LocalTrans=TRUE;
      END IF;
      BEGIN
         EXECUTE IMMEDIATE 'UPDATE Orders SET CustNo=?
                           WHERE CustNo=?' USING NEWROW.CustNo,OLDROW.CustNo;
         IF LocalTrans THEN
            COMMIT;
```

```
         END IF;
      EXCEPTION
         IF LocalTrans THEN
            ROLLBACK;
         END IF;
         RAISE;
      END;
   END IF;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.12 OPERATION

Returns the current operation (Insert, Update, or Delete) when called from within a trigger.

Syntax

```
OPERATION()
```

Returns

```
VARCHAR

Return values are case-sensitive and can be one of the following:

Insert
Update
Delete
```

Usage

The OPERATION function returns the operation that is currently being executed when a trigger is fired. This is useful for universal (ALL) triggers that are defined so that they are fired during any insert, update, or delete operation. Please see the CREATE TRIGGER statement for more information on creating triggers.

Examples

```
-- This trigger checks to see if the
-- State column is filled in, and if not fills
-- in the column with a default value.

CREATE TRIGGER "SetDefaultValues" BEFORE ALL ON "customer"
WHEN OPERATION() IN ('Insert','Update')
BEGIN
   IF NEWROW.State IS NULL OR TRIM(BOTH ' ' FROM NEWROW.State)='' THEN
      SET NEWROW.State='NY';
   END IF;
END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.13 COLUMNCOUNT

Returns the column count for a result set cursor.

Syntax

```
COLUMNCOUNT(<CursorName>)

<CursorName> =

Previously-opened result set cursor
```

Returns

```
INTEGER
```

Usage

The COLUMNCOUNT function returns the column count for a result set cursor. If the result set cursor has not been opened yet by using the OPEN statement, then calling this function will result in an error.

Combined with the COLUMNNAME function, this function is useful for dynamically iterating over the result set columns, optionally fetching, inserting, or updating them.

Examples

```
-- This procedure returns a semicolon-delimited
-- string containing the column names for a given table

CREATE FUNCTION ColumnNames(IN TableName VARCHAR COLLATE ANSI_CI)
RETURNS VARCHAR COLLATE ANSI_CI
BEGIN
   DECLARE ResultCursor SENSITIVE CURSOR FOR Stmt;
   DECLARE I INTEGER;
   DECLARE ResultColumnCount INTEGER;
   DECLARE Result VARCHAR DEFAULT '';

   PREPARE Stmt FROM 'TABLE '+QUOTEDSTR(TableName,'"');

   OPEN ResultCursor;

   SET I=1;
   SET ResultColumnCount=COLUMNCOUNT(ResultCursor);

   WHILE I <= ResultColumnCount  DO
      IF Result <> '' THEN
         SET Result=Result+';';
      END IF;
      SET Result=Result+COLUMNNAME(ResultCursor,I);
```

```
      SET I=I+1;
   END WHILE;

   CLOSE ResultCursor;

   RETURN Result;

END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.14 COLUMNNAME

Returns the column name of the column at a given position in a result set cursor.

Syntax

```
COLUMNNAME(<CursorName>,<Position>)

<CursorName> =

Previously-opened result set cursor

<Position> = 1-based column position
```

Returns

```
VARCHAR
```

Usage

The COLUMNNAME function returns the column name for the column at a given position in a result set cursor. If the result set cursor has not been opened yet by using the OPEN statement, then calling this function will result in an error.

Combined with the COLUMNCOUNT function, this function is useful for dynamically iterating over the result set columns, optionally fetching, inserting, or updating them.

Examples

```
-- This procedure returns a semicolon-delimited
-- string containing the column names for a given table

CREATE FUNCTION ColumnNames(IN TableName VARCHAR COLLATE ANSI_CI)
RETURNS VARCHAR COLLATE ANSI_CI
BEGIN
   DECLARE ResultCursor SENSITIVE CURSOR FOR Stmt;
   DECLARE I INTEGER;
   DECLARE ResultColumnCount INTEGER;
   DECLARE Result VARCHAR DEFAULT '';

   PREPARE Stmt FROM 'TABLE '+QUOTEDSTR(TableName,'"');

   OPEN ResultCursor;

   SET I=1;
   SET ResultColumnCount=COLUMNCOUNT(ResultCursor);

   WHILE I <= ResultColumnCount  DO
      IF Result <> '' THEN
```

```
         SET Result=Result+';';
      END IF;
      SET Result=Result+COLUMNNAME(ResultCursor,I);
      SET I=I+1;
   END WHILE;

   CLOSE ResultCursor;

   RETURN Result;

END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

17.15 STMTRESULT

Returns the result of the last statement execution (if available).

Syntax

```
STMTRESULT(<StatementName>)

<StatementName> =

Previously-executed statement
```

Returns

```
BOOLEAN
```

Usage

The STMTRESULT function returns the result of the execution of a given statement. Currently, only the VERIFY TABLE and REPAIR TABLE statements report a result, and each reports False if no errors were found during the verification or repair process.

Examples

```
-- This function returns whether any errors
-- were found when repairing the passed table name

CREATE FUNCTION RepairTable(IN TableName VARCHAR COLLATE ANSI_CI,
                            IN StructureOnly BOOLEAN)
RETURNS BOOLEAN
BEGIN
   DECLARE Stmt STATEMENT;

   PREPARE Stmt FROM 'REPAIR TABLE '+QUOTEDSTR(TableName,'"')+
     IF(StructureOnly,' STRUCTURE ONLY','');

   EXECUTE Stmt;

   RETURN STMTRESULT(Stmt);

END
```

SQL 2003 Standard Deviations

This function deviates from the SQL 2003 standard in the following ways:

Deviation	Details
Extension	This function is an ElevateDB extension.

Appendix A - Error Codes and Messages

The following is a table of the error codes and messages for ElevateDB. You can find out more information on how to handle ElevateDB exceptions in your product-specific manual.

Error Code	Message and Further Details
EDB_ERROR_VALIDATE (100)	There is an error in the metadata for the <ObjectType> <ObjectName> (<ErrorMessage>)This error is raised whenever an attempt is made to create a new catalog or configuration object, and there is an error in the specification of the object. The specific error message is indicated within the parentheses.
EDB_ERROR_UPDATE (101)	There was an error updating the <ObjectType> <ObjectName> (<ErrorMessage>)This error is raised whenever ElevateDB encounters an issue while trying to update the disk file used to store a catalog or configuration. The specific error message is indicated within the parentheses.
EDB_ERROR_SYSTEM (200)	This operation cannot be performed on the system <ObjectType> <ObjectName> or any privileges granted to itThis error is raised whenever an attempt is made to alter or drop any system-defined catalog or configuration objects. Please see the System Information topic for more information on the system-defined objects in ElevateDB.
EDB_ERROR_DEPENDENCY (201)	The <ObjectType> <ObjectName> cannot be dropped or moved because it is still referenced by the <ObjectType> <ObjectName>This error is raised whenever an attempt is made to drop any catalog or configuration object, and that catalog or configuration object is still being referenced by another catalog or configuration object. You must first remove the reference to the object that you wish to drop before you can drop the referenced object.
EDB_ERROR_MODULE (202)	An error occurred with the module <ModuleName> (<ErrorMessage>)This error is raised whenever ElevateDB encounters an issue with loading an external module. Please see the External Modules topic for more information.
EDB_ERROR_LOCK (300)	Cannot lock <ObjectType> <ObjectName> for <AccessType> accessThis error is raised whenever ElevateDB cannot obtain the desired lock access to a given catalog or configuration object. This is usually due to another session already having an incompatible lock on the object already. Please see the Locking and Concurrency topic for more information.
EDB_ERROR_UNLOCK (301)	Cannot unlock <ObjectType> <ObjectName> for <AccessType> accessThis error is raised whenever ElevateDB cannot unlock a given catalog or configuration object. If this error occurs during normal operation of

	ElevateDB, please contact Elevate Software for further instructions on how to correct the issue
EDB_ERROR_EXISTS (400)	The <ObjectType> <ObjectName> already existsThis error is raised whenever an attempt is made to create a new catalog or configuration object, and a catalog or configuration object already exists with that name.
EDB_ERROR_NOTFOUND (401)	The <ObjectType> <ObjectName> does not existThis error is raised when an attempt is made to open/execute, alter, or drop a catalog or configuration object that does not exist.
EDB_ERROR_NOTOPEN (402)	The database <DatabaseName> must be open in order to perform this operation (<OperationName>)This error is raised when an attempt is made to perform an operation on a given database before it has been opened.
EDB_ERROR_READONLY (403)	The <ObjectType> <ObjectName> is read-only and this operation cannot be performed (<OperationName>)This error is raised whenever a create, alter, or drop operation is attempted on an object that is read-only.
EDB_ERROR_TRANS (404)	This operation cannot be performed while the database <DatabaseName> has an active transaction (<OperationName>)This error is raised whenever ElevateDB encounters an invalid transaction operation. Some SQL statements cannot be executed within a transaction. For a list of transaction-compatible statements, please see the Transactions topic.
EDB_ERROR_MAXIMUM (405)	The maximum number of <ObjectType>s has been reached (<MaximumObjectsAllowed>)This error is raised when an attempt is made to create a new catalog or configuration object and doing so would exceed the maximum allowable number of objects. Please see the Appendix B - System Capacities topic for more information.
EDB_ERROR_IDENTIFIER (406)	Invalid <ObjectType> identifier '<ObjectName>'This error is raised when an attempt is made to create a new catalog or configuration object with an invalid name. Please see the Identifiers topic for more information on what constitutes a valid identifier.
EDB_ERROR_FULL (407)	The table <TableName> is full (<FileName>)This error occurs when a given table contains the maximum number of rows or the maximum file size is reached for one of the files that make up the table. The file name is indicated within the parentheses.
EDB_ERROR_CONFIG (409)	There is an error in the configuration (<ErrorMessage>)This error is raised whenever there is an error in the configuration. The specific error message is indicated within the parentheses.

EDB_ERROR_NOLOGIN (500)	A user must be logged in in order to perform this operation (<OperationName>)This error is raised whenever an attempt is made to perform an operation for a session that has not been logged in yet with a valid user name and password.
EDB_ERROR_LOGIN (501)	Login failed (<ErrorMessage>)This error is raised whenever a user login fails. ElevateDB allows for a maximum of 3 login attempts before raising a login exception.
EDB_ERROR_ADMIN (502)	Administrator privileges are required to perform this operation (<Operation>)This error is raised when an attempt is made to perform an operation that requires administrator privileges. Administrator privileges are granted to a given user by granting the system-defined "Administrators" role to that user. Please see the User Security topic for more information.
EDB_ERROR_PRIVILEGE (503)	The current user does not have the proper privileges to perform this operation (<OperationName>)This error is raised when a user attempts an operation when he/she does not have the proper privileges required to execute the operation. Please see the User Security topic for more information.
EDB_ERROR_MAXSESSIONS (504)	Maximum number of concurrent sessions reached for the configuration <ConfigurationName>This error is raised when the maximum number of licensed sessions for a given configuration is exceeded. The number of licensed sessions for a given configuration depends upon the ElevateDB product purchased along with the particular compilation of the application made by the developer using the ElevateDB product.
EDB_ERROR_SERVER (505)	The ElevateDB Server cannot be started (<ErrorMessage>) The ElevateDB Server cannot be stopped (<ErrorMessage>)This error is raised when the ElevateDB Server cannot be started or stopped for any reason. Normally, the error message will contain a native operating system error message that will reveal the reason for the issue.
EDB_ERROR_FILEMANAGER (600)	File manager error (<ErrorMessage>)This error is raised whenever ElevateDB encounters a file manager error while trying to create, open, close, delete, or rename a file. The specific error message, including operating system error code (if available), is indicated within the parentheses.
EDB_ERROR_CORRUPT (601)	The table <TableName> is corrupt (<ErrorMessage>)This error is raised when ElevateDB encounters an issue while reading, writing, or validating a table. If this error occurs during normal operation of ElevateDB, please contact Elevate Software for further instructions on how to correct the issue. The specific error message is indicated within the parentheses.

EDB_ERROR_COMPILE (700)	An error was found in the <ObjectType> at line <Line> and column <Column> (<ErrorMessage>)This error is raised whenever an error is encountered while compiling an SQL expression, statement, or routine. The specific error message is indicated within the parentheses.
EDB_ERROR_BINDING (800)	A row binding error occurredThis error is raised when ElevateDB encounters an issue while trying to bind the cursor row values in a cursor row. It is an internal error and will not occur unless there is a bug in ElevateDB.
EDB_ERROR_STATEMENT (900)	An error occurred with the statement <StatementName> (<ErrorMessage>)This error is raised whenever an issue is encountered while executing a statement. The specific error message is indicated within the parentheses.
EDB_ERROR_PROCEDURE (901)	An error occurred with the procedure <ProcedureName> (<ErrorMessage>)This error is raised whenever an issue is encountered while executing a procedure. The specific error message is indicated within the parentheses.
EDB_ERROR_VIEW (902)	An error occurred with the view <ViewName> (<ErrorMessage>)This error is raised whenever an issue is encountered while opening a view. The specific error message is indicated within the parentheses.
EDB_ERROR_JOB (903)	An error occurred with the job <JobName> (<ErrorMessage>)This error is raised whenever an issue is encountered while running a job. The specific error message is indicated within the parentheses.
EDB_ERROR_IMPORT (904)	Error importing the file <FileName> into the table <TableName> (<ErrorMessage>)This error is raised when an error occurs during the import process for a given table. The specific error message is indicated within the parentheses.
EDB_ERROR_EXPORT (905)	Error exporting the table <TableName> to the file <FileName> (<ErrorMessage>)This error is raised when an error occurs during the export process for a given table. The specific error message is indicated within the parentheses.
EDB_ERROR_CURSOR (1000)	An error occurred with the cursor <CursorName> (<ErrorMessage>)This error is raised whenever an issue is encountered while operating on a cursor. The specific error message is indicated within the parentheses.
EDB_ERROR_FILTER (1001)	A filter error occurred (<ErrorMessage>)This error is raised whenever ElevateDB encounters an issue while trying to set or clear a filter on a given cursor. The specific error message is indicated within the parentheses.
EDB_ERROR_LOCATE (1002)	A locate error occurred (<ErrorMessage>)This error is raised whenever ElevateDB encounters an issue while trying to locate a row in a given cursor. The specific error message is indicated within the parentheses.

EDB_ERROR_STREAM (1003)	An error occurred in the cursor stream (<ErrorMessage>)This error is raised whenever an issue is encountered while loading or saving a cursor to or from a stream. The specific error message is indicated within the parentheses.
EDB_ERROR_CONSTRAINT (1004)	The constraint <ConstrainName> has been violated (<ErrorMessage>)This error is raised when a constraint that has been defined for a table is violated. This includes primary key, unique key, foreign key, and check constraints. The specific error message is indicated within the parentheses.
EDB_ERROR_LOCKROW (1005)	Cannot lock the row in the table <TableName>This error is raised when a request is made to lock a given row and the request fails because another session has the row already locked. Please see the Locking and Concurrency topic for more information.
EDB_ERROR_UNLOCKROW (1006)	Cannot unlock the row in the table <TableName>This error is raised whenever ElevateDB cannot unlock a specific row because the row had not been previously locked, or had been locked and the lock has since been cleared. Please see the Locking and Concurrency topic for more information.
EDB_ERROR_ROWDELETED (1007)	The row has been deleted since last cached for the table <TableName>This error is raised whenever an attempt is made to update or delete a row, and the row no longer exists because it has been deleted by another session. Please see the Updating Rows topic for more information.
EDB_ERROR_ROWMODIFIED (1008)	The row has been modified since last cached for the table <TableName>This error is raised whenever an attempt is made to update or delete a row, and the row has been updated by another session since the last time it was cached by the current session. Please see the Updating Rows topic for more information.
EDB_ERROR_CONSTRAINED (1009)	The cursor is constrained and this row violates the current cursor constraint condition(s)This error is raised when an attempt is made to insert a new row into a constrained cursor that violates the filter constraints defined for the cursor. Both views defined in database catalogs and the result sets of dynamic queries can be defined as constrained, and the filter constraints in both cases are the WHERE conditions defined for the underlying SELECT query that the view or dynamic query is based upon.
EDB_ERROR_ROWVISIBILITY (1010)	The row is no longer visible in the table <TableName>This error is raised whenever an attempt is made to update or delete a row within the context of a cursor with an active filter or range condition, and the row has been updated by another session since the last time it was cached by the current session, thus causing it to fall out of the scope of the cursor's active filter or range condition. Please see the Updating Rows topic for

	more information.
EDB_ERROR_VALUE (1011)	An error occurred with the <ObjectType> <ObjectName> (<ErrorMessage>)This error is raised whenever an attempt is made to store a value in a column, parameter, or variable and the value is invalid because it is out of range or would be truncated. The specific error message is indicated within the parentheses.
EDB_ERROR_CLIENTCONN (1100)	A connection to the server at <ServerAddress> cannot be established (<ErrorMessage>)This error is raised when ElevateDB encounters an issue while trying to connect to a remote ElevateDB Server. The error message will indicate the reason why the connection cannot be completed.
EDB_ERROR_CLIENTLOST (1101)	A connection to the server at <ServerAddress> has been lost <ErrorMessage>)This error is raised when ElevateDB encounters an issue while connected to a remote ElevateDB Server. The error message will indicate the reason why the connection was lost.
EDB_ERROR_INVREQUEST (1103)	An invalid or unknown request was sent to the serverThis error is raised when an ElevateDB Server encounters an unknown request from a client session.
EDB_ERROR_ADDRBLOCK (1104)	The IP address <IPAddress> is blockedThis error is raised when a session tries to connect to an ElevateDB Server, and the originating IP address for the session matches one of the configured blocked IP addresses in the ElevateDB Server, or does not match one of the configured authorized IP addresses in the ElevateDB Server.
EDB_ERROR_ENCRYPTREQ (1105)	An encrypted connection is requiredThis error is raised when a non-encrypted session tries to connect to an ElevateDB Server that has been configured to only accept encrypted session connections.
EDB_ERROR_SESSIONNOTFOUND (1107)	The session ID <SessionID> is no longer present on the serverThis error is raised whenever a remote session attempts to reconnect to a session that has already been designated as a dead session and removed by the ElevateDB Server. This can occur when a session is inactive for a long period of time, or when the ElevateDB Server has been stopped and then restarted.
EDB_ERROR_SESSIONCURRENT (1108)	The current session ID <SessionID> cannot be disconnected or removedThis error is raised whenever a remote session attempts to disconnect or remove itself.
EDB_ERROR_COMPRESS (1200)	An error occurred while compressing data (<ErrorMessage>)This error is raised when ElevateDB encounters an issue while attempting to compress data. It is an internal error and will not occur unless there is a bug in ElevateDB. The specific error message is indicated within the parentheses.
EDB_ERROR_DECOMPRESS (1201)	An error occurred while uncompressing data

	(<ErrorMessage>)This error is raised when ElevateDB encounters an issue while attempting to decompress data. It is an internal error and will not occur unless there is a bug in ElevateDB. The specific error message is indicated within the parentheses.
EDB_ERROR_BACKUP (1300)	Error backing up the database <DatabaseName> (<ErrorMessage>)This error is raised when any error occurs during the backing up of a database. The specific error message is indicated within the parentheses.
EDB_ERROR_RESTORE (1301)	Error restoring the database <DatabaseName> (<ErrorMessage>)This error is raised when any error occurs during the restore of a database. The specific error message is indicated within the parentheses.
EDB_ERROR_PUBLISH (1302)	Error publishing the database <DatabaseName> (<ErrorMessage>)This error is raised when any error occurs during the publishing of a database. The specific error message is indicated within the parentheses.
EDB_ERROR_UNPUBLISH (1303)	Error unpublishing the database <DatabaseName> (<ErrorMessage>)This error is raised when any error occurs during the unpublishing of a database. The specific error message is indicated within the parentheses.
EDB_ERROR_SAVEUPDATES (1304)	Error saving updates for the database <DatabaseName> (<ErrorMessage>)This error is raised when any error occurs during the saving of the updates for a database. The specific error message is indicated within the parentheses.
EDB_ERROR_LOADUPDATES (1305)	Error loading updates for the database <DatabaseName> (<ErrorMessage>)This error is raised when any error occurs during the loading of the updates for a database. The specific error message is indicated within the parentheses.
EDB_ERROR_STORE (1306)	Error with the store <StoreName> (<ErrorMessage>)This error is raised when any error occurs while trying to access a store, such as a read or write error while working with files in the store. The specific error message is indicated within the parentheses.
EDB_ERROR_CACHEUPDATES (1307)	Error caching updates for the cursor <CursorName> (<ErrorMessage>)This error is raised when any error occurs during the caching of updates for a specific table, view, or query cursor. The specific error message is indicated within the parentheses.
EDB_ERROR_FORMAT (1400)	Error in the format string <FormatString> (<ErrorMessage>)This error is raised when ElevateDB encounters an issue with a format string used in a date, time, or timestamp format used in a table import or export. The specific error message is indicated within the parentheses.

This page intentionally left blank

Appendix B - System Capacities

The following is a list of the capacities for the different objects in ElevateDB. Any object that is not specifically mentioned here has an implicit capacity of 2147483647, or High(Integer). For example, there is no stated capacity for the maximum number of roles allowed in a configuration. Therefore, the implicit capacity is 2147483647 roles.

Capacity	Details
Max BLOB Column Size	The maximum size of a BLOB column is 2GB.
Max CHAR/VARCHAR Column Length	The maximum length of a VARCHAR/CHAR columns is 1024 characters.
Max Identifier Length	The maximum length of an identifier is 80 characters.
Max Number of Columns in a Table	The maximum number of columns in a table is 2048.
Max Number of Columns in an Index	The maximum number of columns in an index is limited by the table's defined index page size.
Max Number of Concurrent Sessions	The maximum number of concurrent sessions for an application or ElevateDB server is 4096.
Max Number of Indexes in a Table	The maximum number of indexes in a table is 512.
Max Number of Jobs in a Configuration	The maximum number of jobs in a configuration is 4096.
Max Number of Routines in a Database	The maximum number of routines (procedures and functions combined) in a database is 4096.
Max Number of Rows in a Table	The maximum number of rows in a table is determined by whether global file I/O buffering is enabled in ElevateDB. If global file I/O buffering is enabled, then the maximum number of rows is determined by the maximum file size permitted in the operating system. If global file I/O buffering is not enabled, then the approximate maximum number of rows can be determined by dividing 128GB by the row size.
Max Number of Rows in a Transaction	The maximum number of rows in a single transaction is only limited by the available memory constraints of the operating system and/or hardware.
Max Number of Tables in a Database	The maximum number of tables in a database is 4096.
Max Number of Users in a Configuration	The maximum number of users in a configuration is 4096.
Max Row Size for a Table	The maximum row size for a table is 2GB.
Max Scale for DECIMAL/NUMERIC Columns	The maximum scale for DECIMAL or NUMERIC columns is 4.
Max Size of an In-Memory Table	The maximum size of an in-memory table is only limited by the available memory constraints of the operating system or hardware.
Min/Max BLOB Block Size for a Table	The minimum BLOB block size is 64 bytes for ANSI databases and 128 bytes for Unicode databases. The maximum BLOB block size is 2GB.

Min/Max Index Page Size for a Table	The minimum index page size is 1 kilobyte for ANSI databases and 2 kilobytes for Unicode databases. The maximum index page size is 2GB.