ElevateDB Version 2 Manual

Table Of Contents

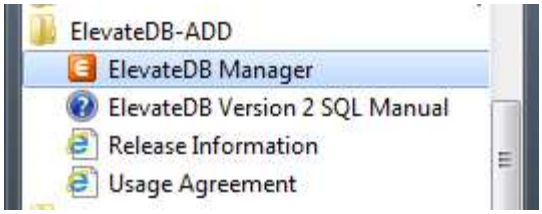
Chapter 1
Local Application Tutorial

1.1 Creating the Tutorial Database

Before creating the actual tutorial application, you must first create the Tutorial database that will be used in the application. The following steps will guide you through creating the Tutorial database using the ElevateDB Manager.

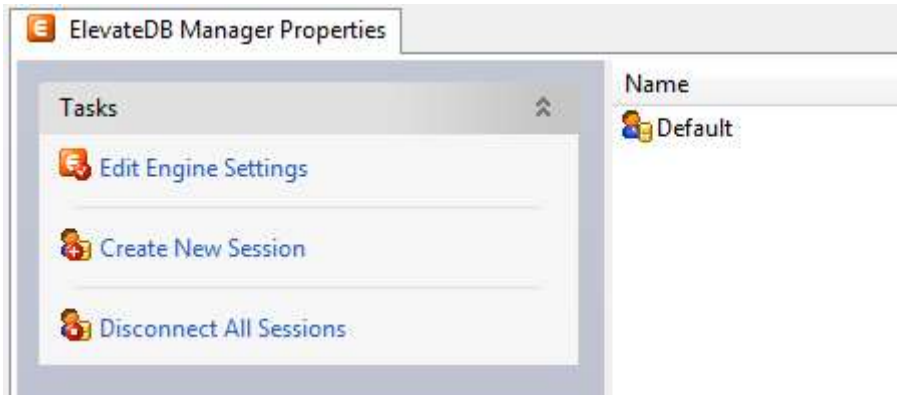
1. Start the ElevateDB Manager (edbmgr.exe) by clicking on the ElevateDB Manager link in the Start menu.

> **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.

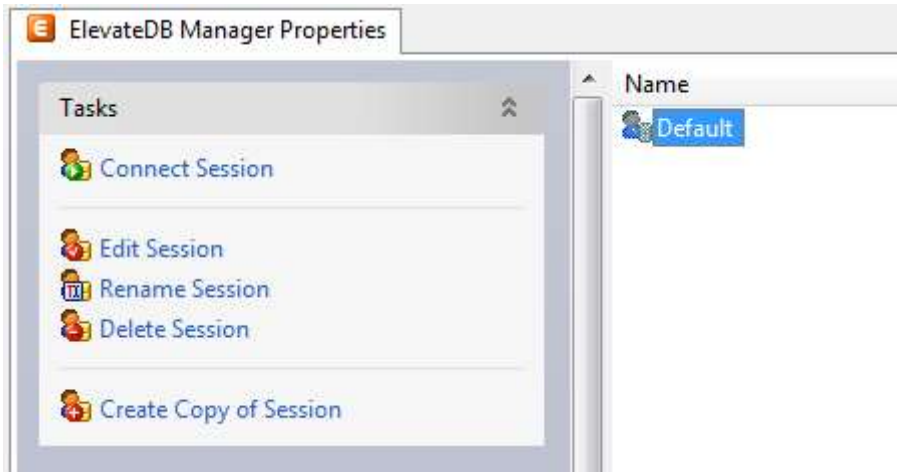


2. Make sure that the session is using the desired character set and configuration file folder (**C:\Tutorial**).

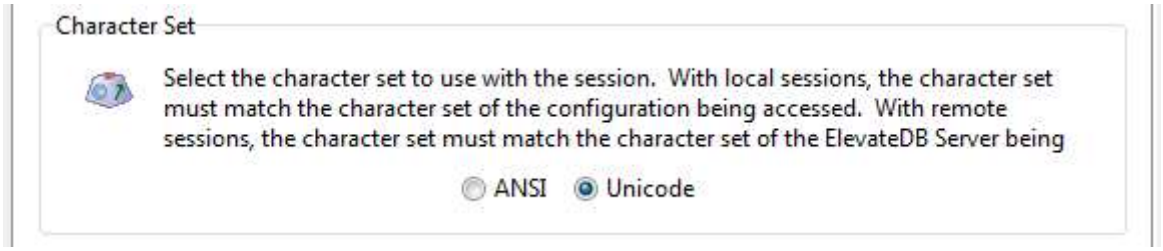
a. Select the **Default** session from the list of available sessions.



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

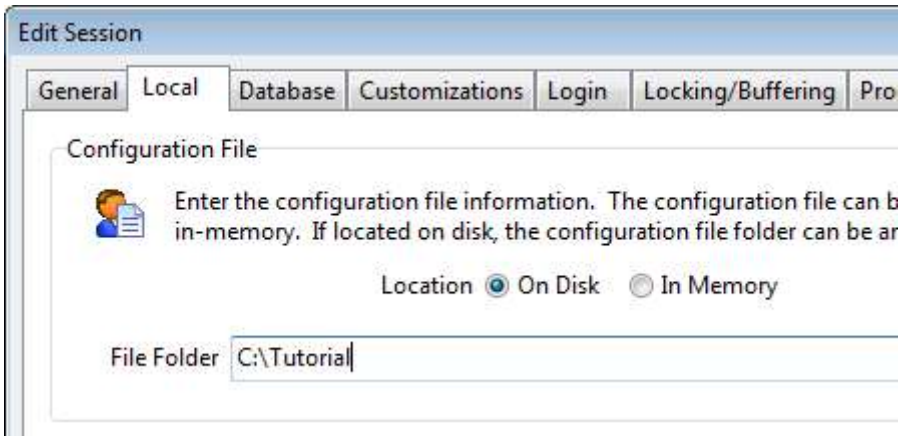


c. On the **General** page of the Edit Session dialog, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



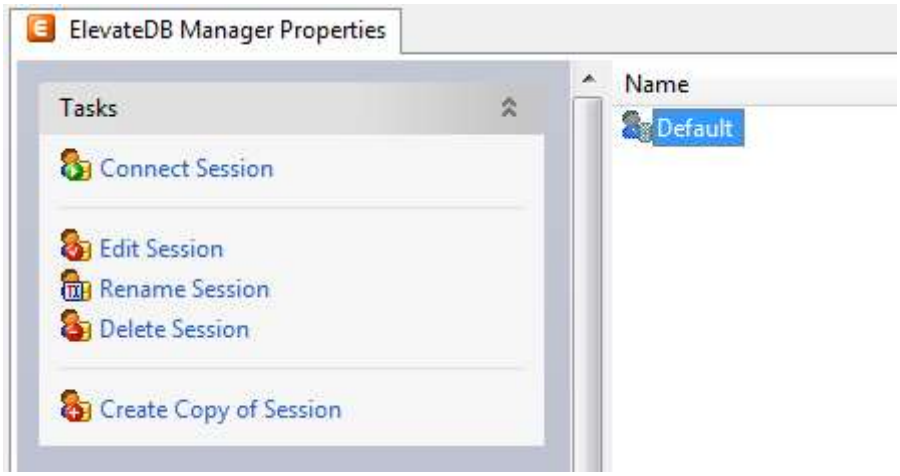
Note
If you're not sure which character set to select and this is the first time using the ElevateDB Manager, then leave the character set at the default of Unicode. The only exception to this rule is if you are using Borland Developer Studio 2005 or lower (including Delphi 5, 6, and 7, as well as C++Builder 5 and 6). You should use the ANSI character set with those older compilers, due to a lack of proper Unicode support for fixed-character and memo field types.

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

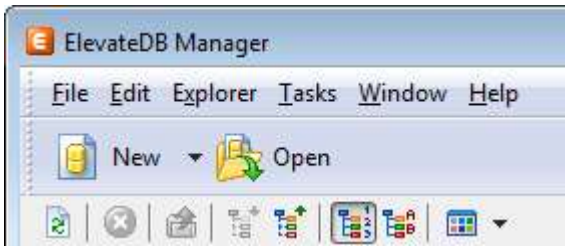


e. Click on the **OK** button.

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



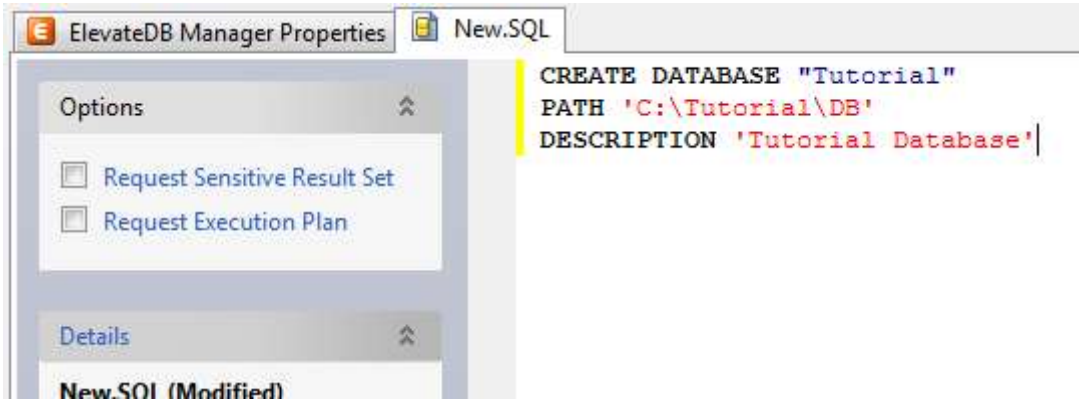
4. Click on the **New** button on the main toolbar.



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

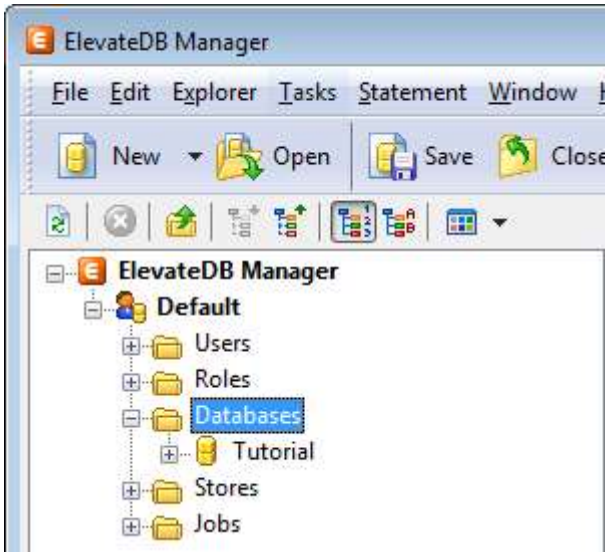
```
CREATE DATABASE "Tutorial"
PATH 'C:\Tutorial\DB'
DESCRIPTION 'Tutorial Database'
```

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

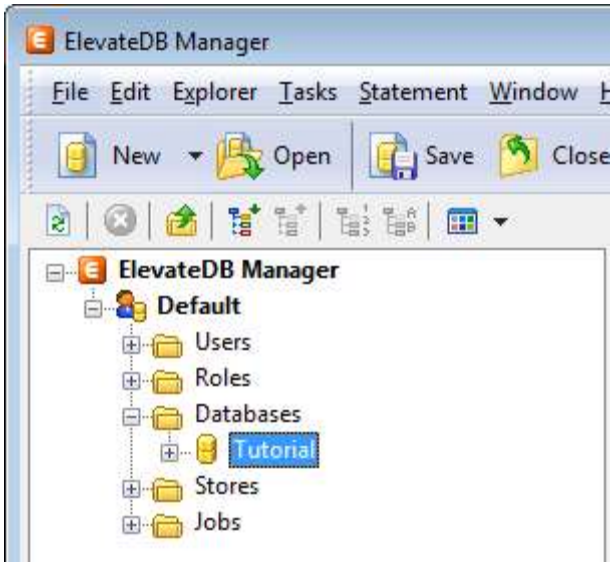


7. Press the **F5** key to refresh the explorer contents for the session.

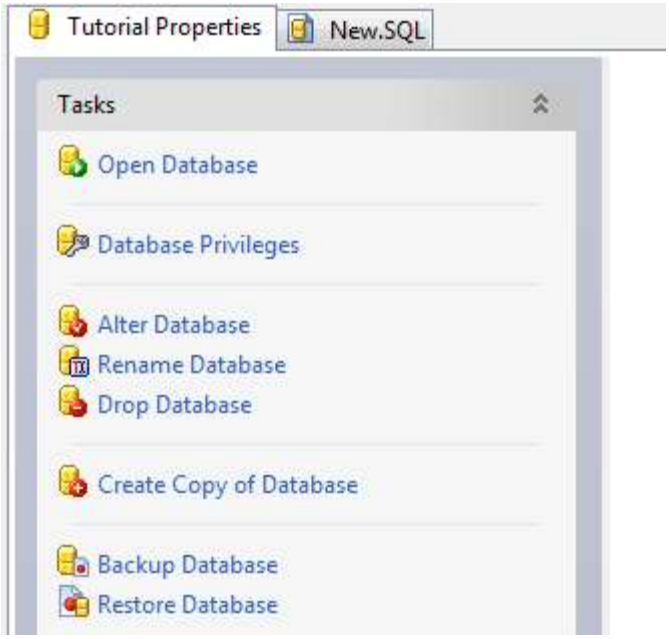
8. Click on the **+** sign next to the **Databases** node in the treeview.



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

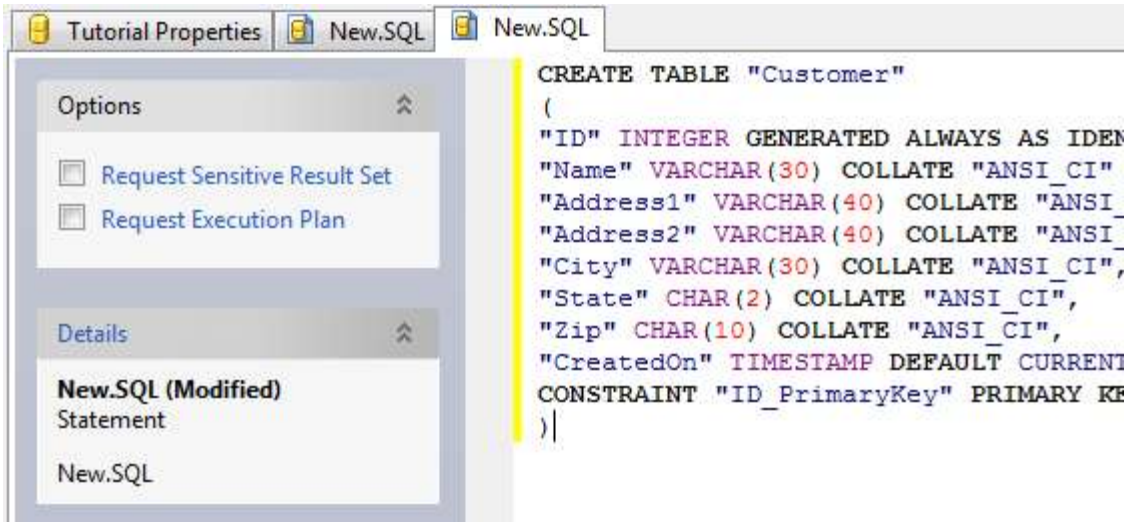
```
CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "ANSI_CI" NOT NULL,
"Address1" VARCHAR(40) COLLATE "ANSI_CI",
"Address2" VARCHAR(40) COLLATE "ANSI_CI",
"City" VARCHAR(30) COLLATE "ANSI_CI",
"State" CHAR(2) COLLATE "ANSI_CI",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

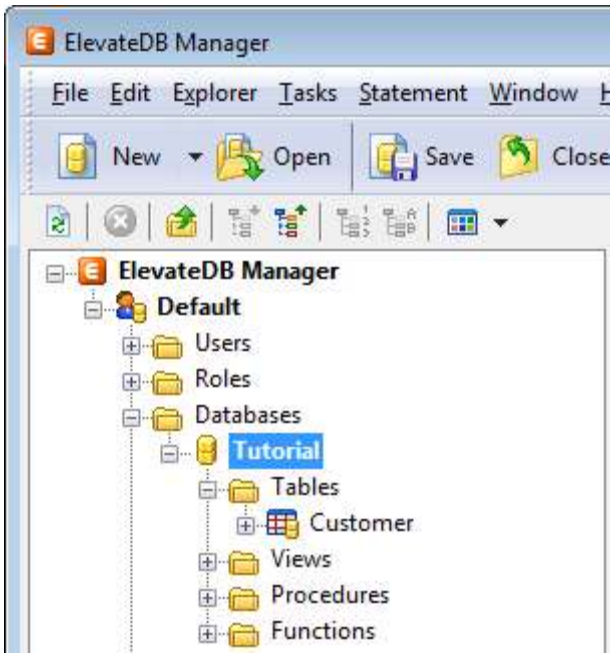
```
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID")
)
```

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.



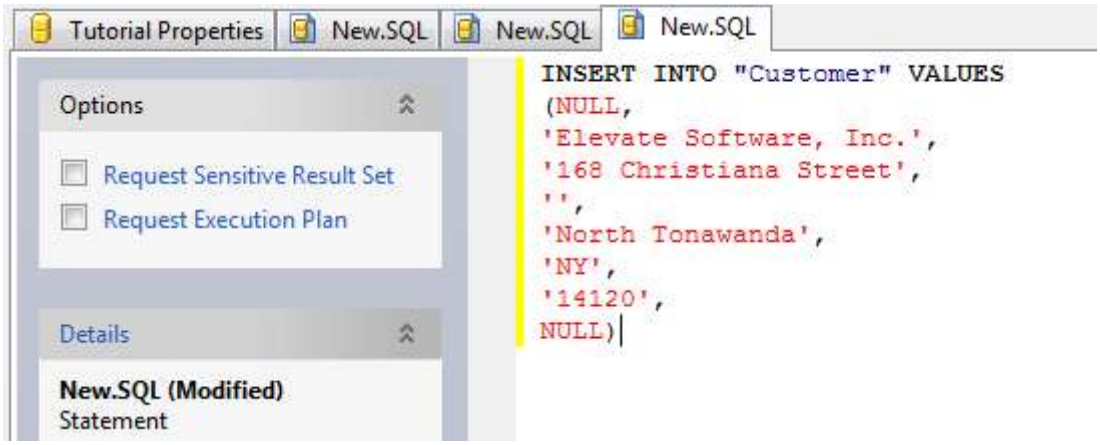
16. Click on the **New.SQL** tab to bring forward the SQL window.

17. Paste in the following INSERT SQL statement:

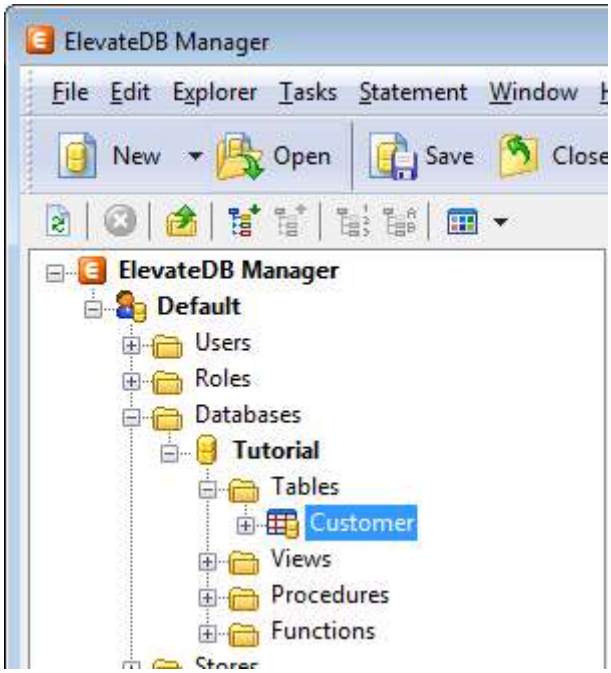
```
INSERT INTO "Customer" VALUES
(NULL,
```

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

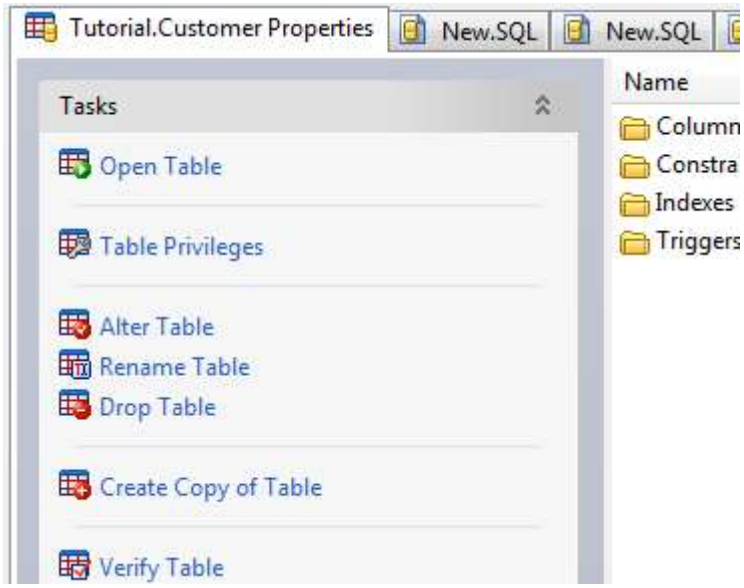
18. Press the **F9** key to execute the SQL statement.



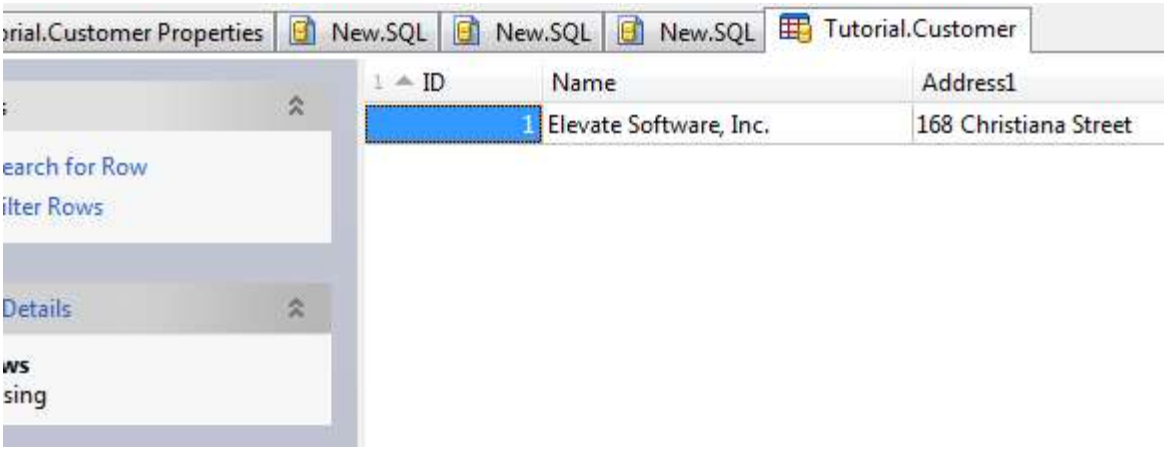
19. Click on the **Customer** table that you just created.



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.

1.2 Creating the Application

The following steps will guide you through creating a basic local application using ElevateDB.

> **Note**
> It is assumed that you have already created the required database using the steps outlined in the Creating the Tutorial Database topic.

You have now successfully created a basic local application for ElevateDB.

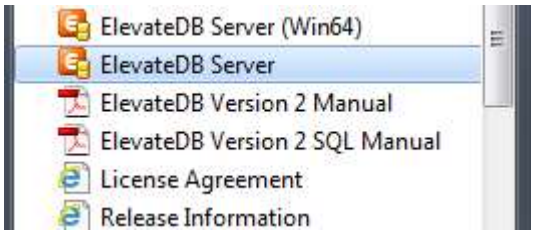
This page intentionally left blank

Chapter 2
Client-Server Application Tutorial

2.1 Configuring and Starting the ElevateDB Server

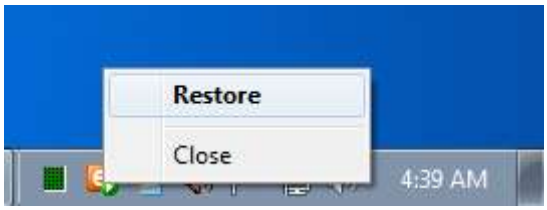
Before creating the tutorial database and application, you must first configure and start the ElevateDB Server.

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

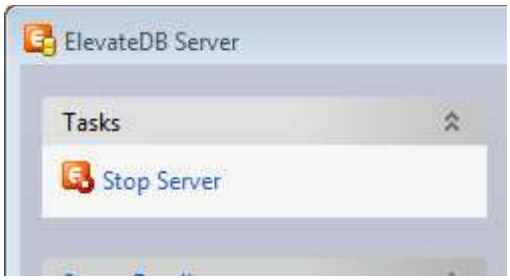


2. Make sure that the server is using the desired character set and configuration file folder (**C:\Tutorial**).

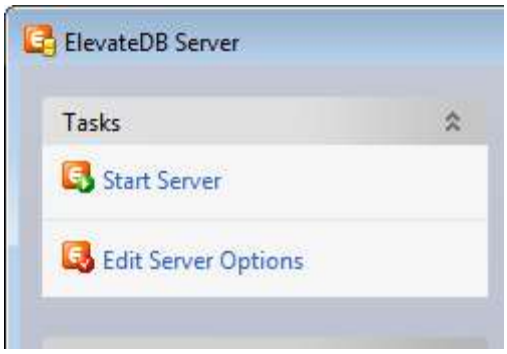
a. In the system tray, right-click on the ElevateDB Server icon to bring up the server menu, and click on the **Restore** option on the server menu.



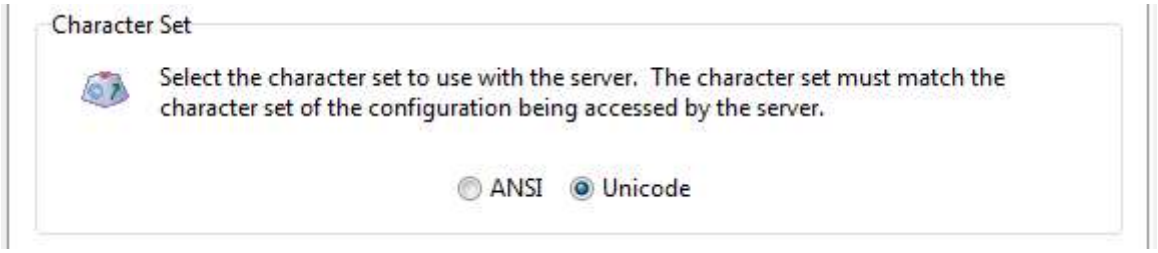
b. In the Tasks pane, click on the **Stop Server** link.



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

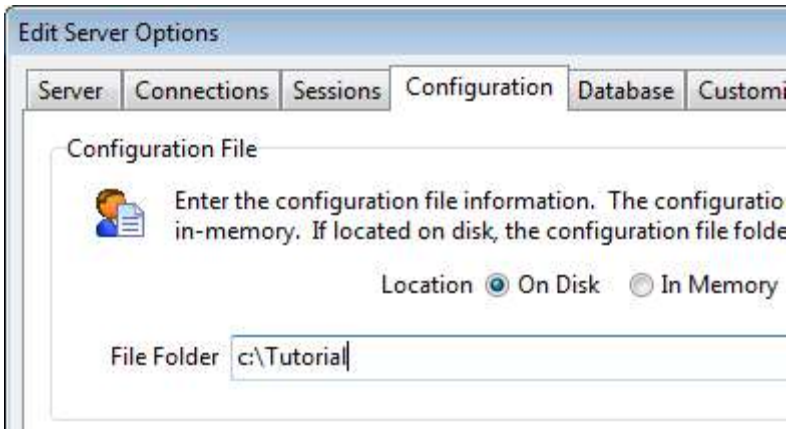


d. On the **Server** page, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



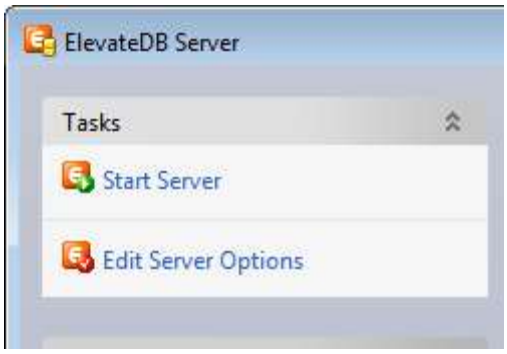
Note
If you're not sure which character set to select and this is the first time using the ElevateDB Server, then leave the character set at the default of Unicode.

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).

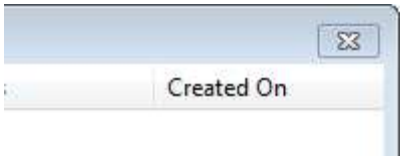


f. Click on the **OK** button.

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.



You have now successfully configured and started the ElevateDB Server.

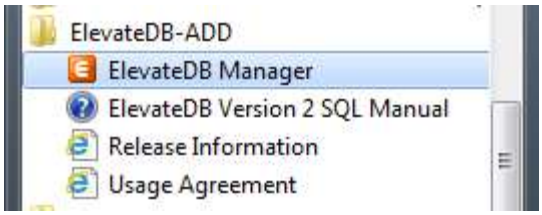
2.2 Creating the Tutorial Database

Before creating the actual tutorial application, you must first create the Tutorial database that will be used in the application. The following steps will guide you through creating the Tutorial database using the ElevateDB Manager.

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

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

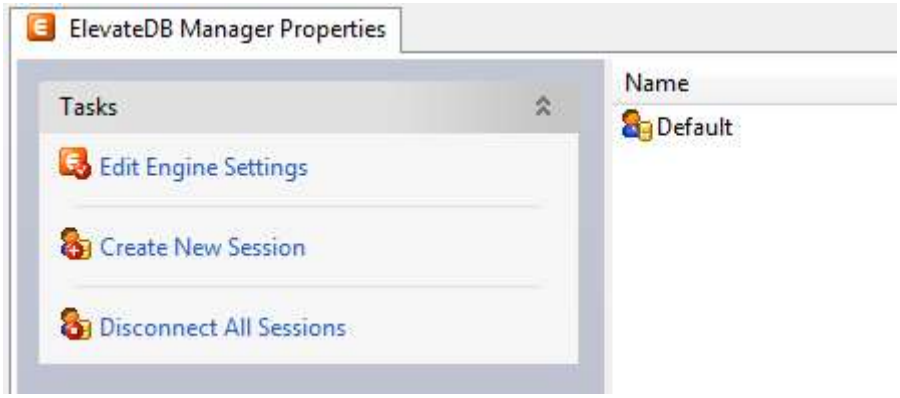
> **Note**
> The ElevateDB Manager is installed with the ElevateDB Additional Software and Utilities (EDB-ADD) installation available from the Downloads page of the web site.



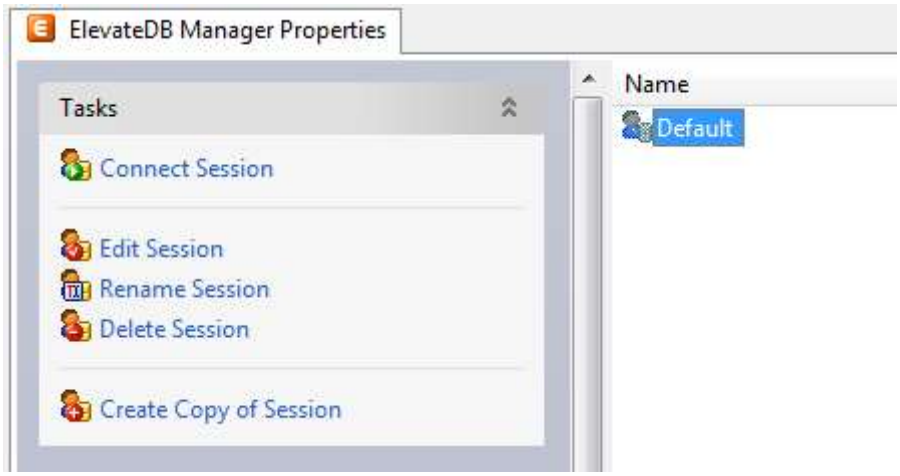
2. Make sure that the session is using the correct session type (**Remote**) and desired character set.

> **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.

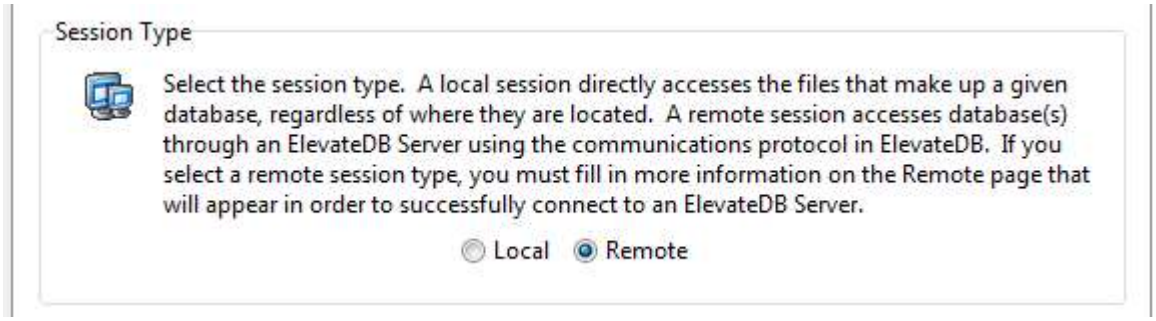
a. Select the **Default** session from the list of available sessions.



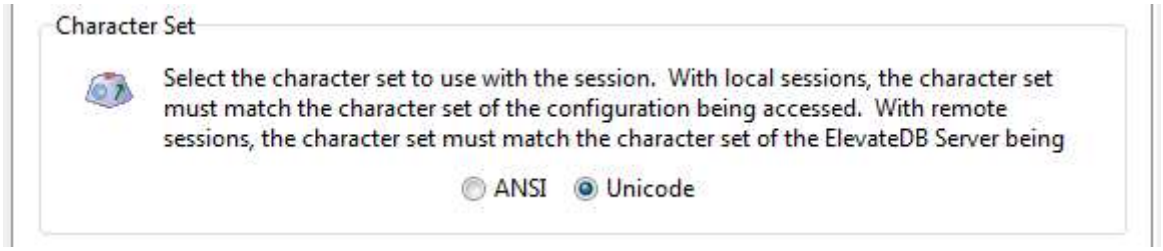
b. In the Tasks pane, click on the **Edit Session** link.



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



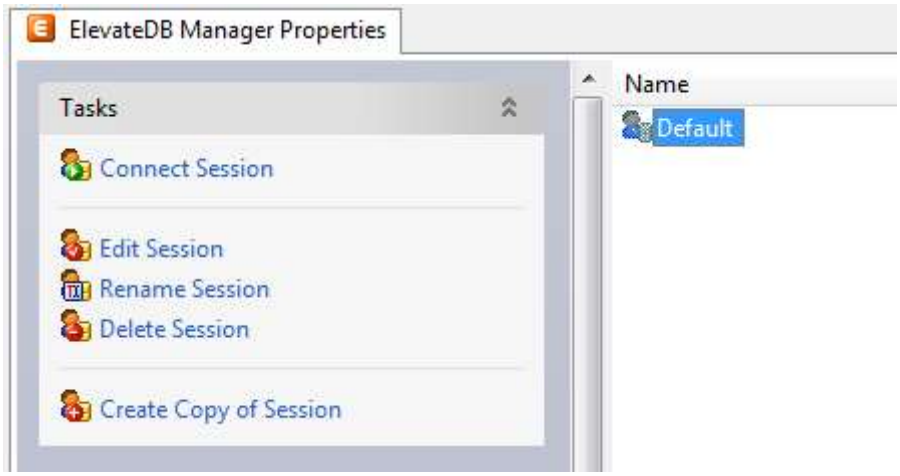
d. On the **General** page of the Edit Session dialog, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



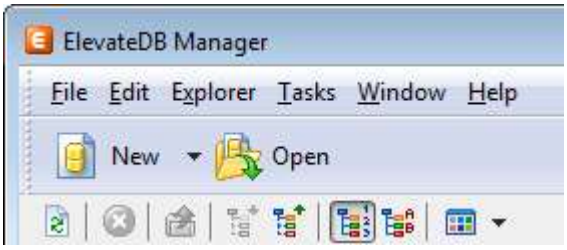
> **Note**
> If you're not sure which character set to select and this is the first time using the ElevateDB Manager, then leave the character set at the default of Unicode.

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

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



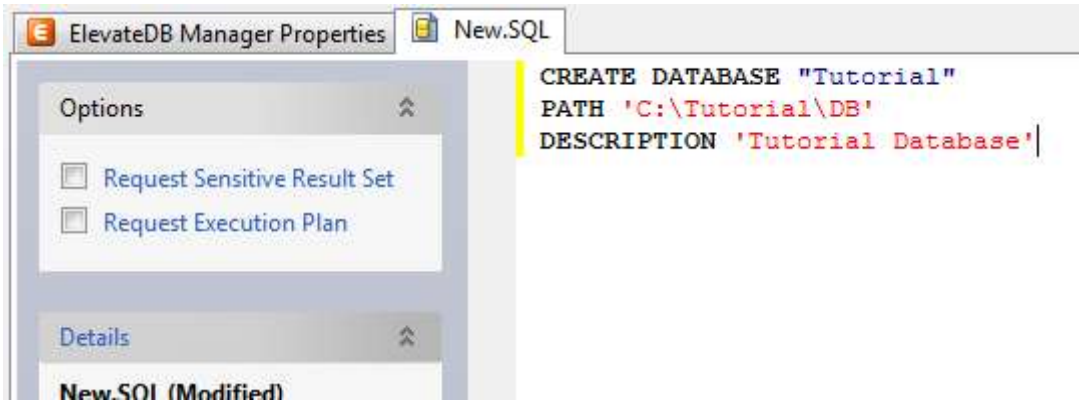
4. Click on the **New** button on the main toolbar.



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

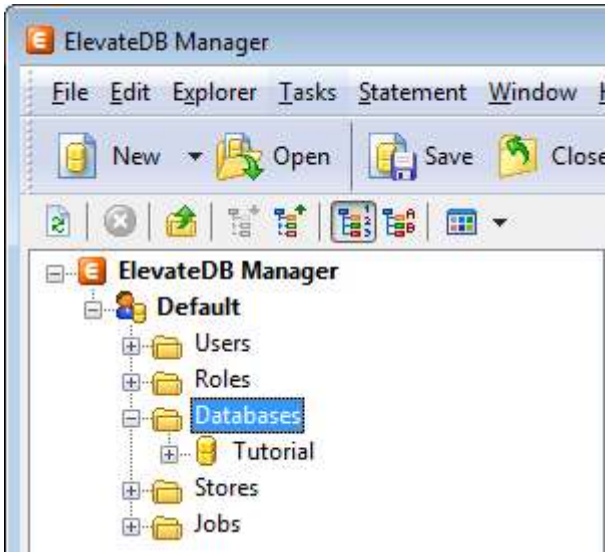
```
CREATE DATABASE "Tutorial"
PATH 'C:\Tutorial\DB'
DESCRIPTION 'Tutorial Database'
```

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

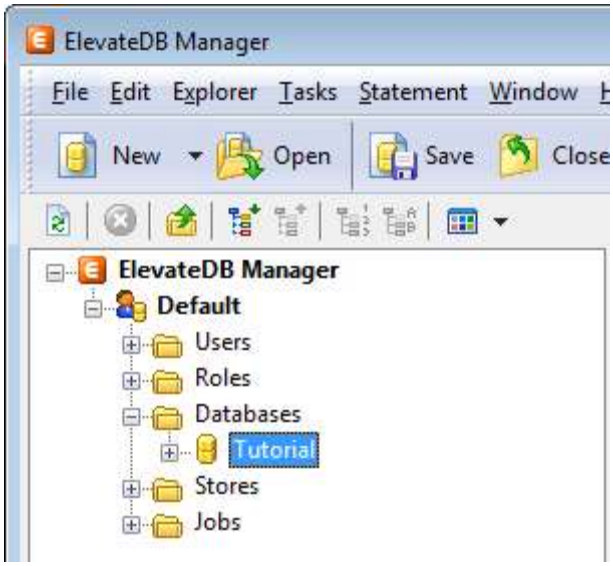


7. Press the **F5** key to refresh the explorer contents for the session.

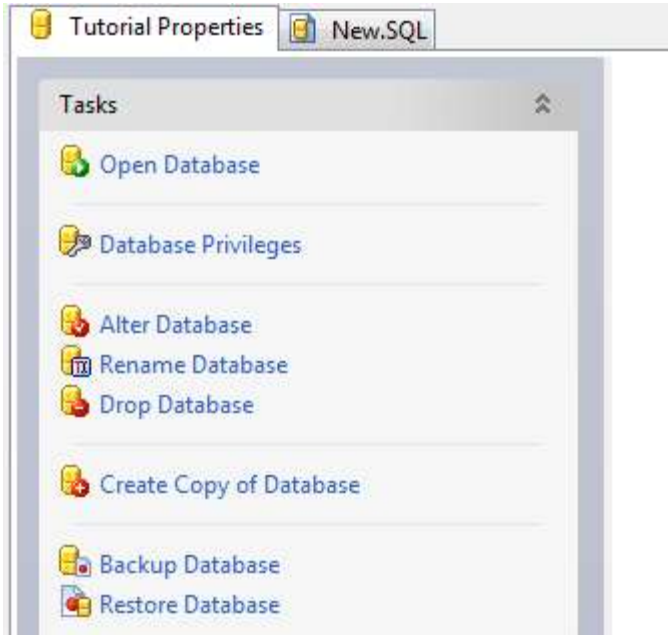
8. Click on the **+** sign next to the **Databases** node in the treeview.



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

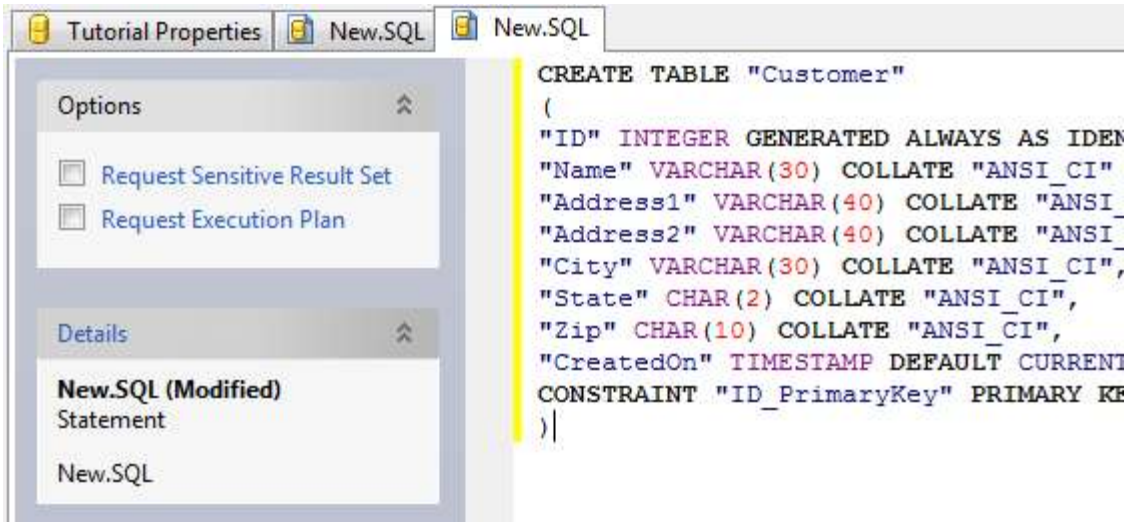
```
CREATE TABLE "Customer"
(
"ID" INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1),
"Name" VARCHAR(30) COLLATE "ANSI_CI" NOT NULL,
"Address1" VARCHAR(40) COLLATE "ANSI_CI",
"Address2" VARCHAR(40) COLLATE "ANSI_CI",
"City" VARCHAR(30) COLLATE "ANSI_CI",
"State" CHAR(2) COLLATE "ANSI_CI",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

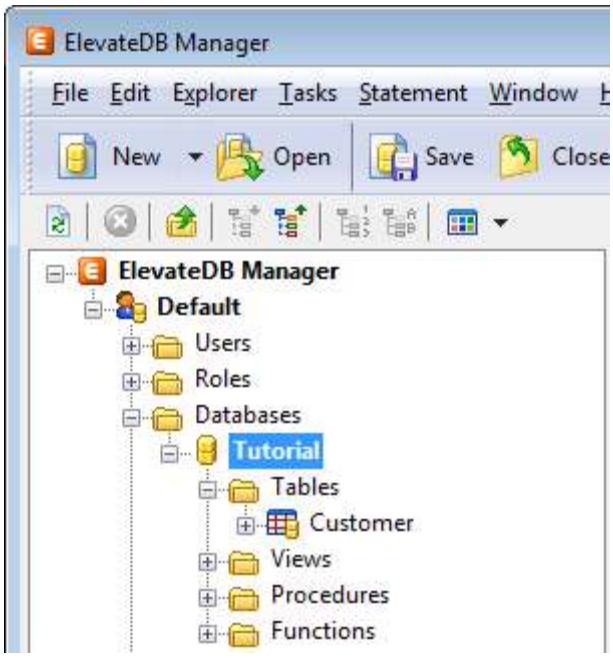
```
CONSTRAINT "ID_PrimaryKey" PRIMARY KEY ("ID")
)
```

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.



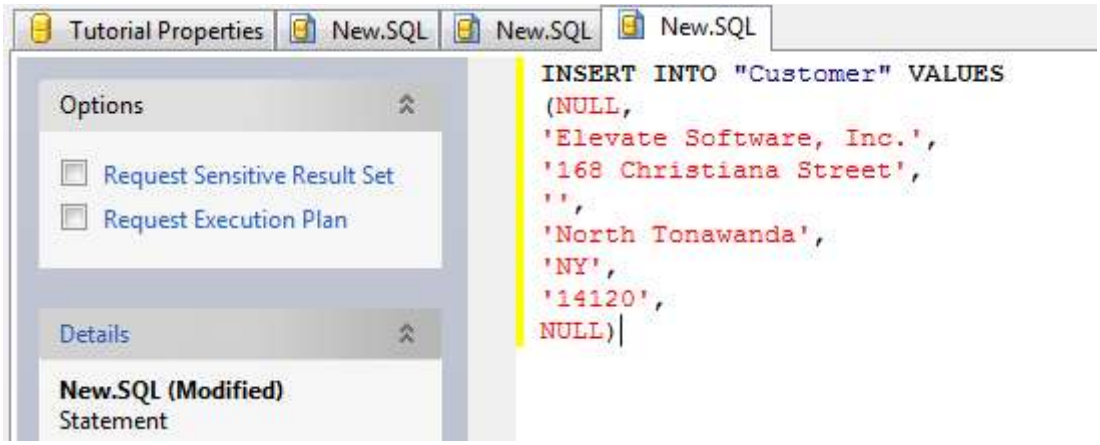
16. Click on the **New.SQL** tab to bring forward the SQL window.

17. Paste in the following INSERT SQL statement:

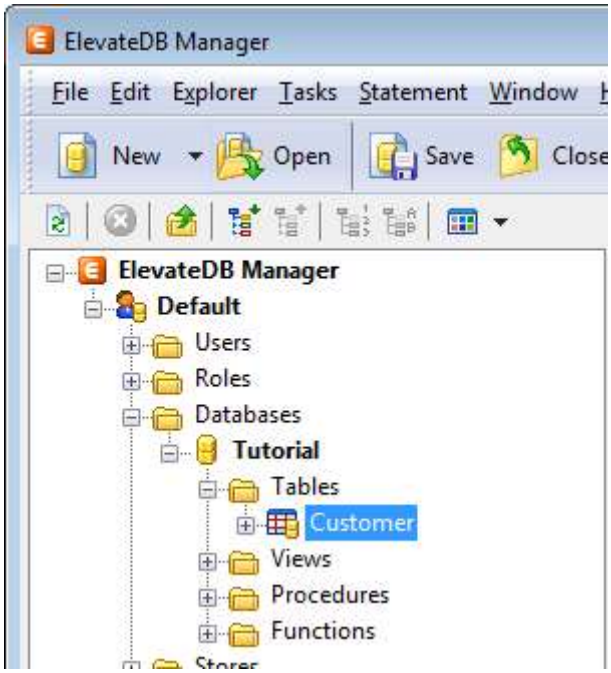
```
INSERT INTO "Customer" VALUES
(NULL,
```

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

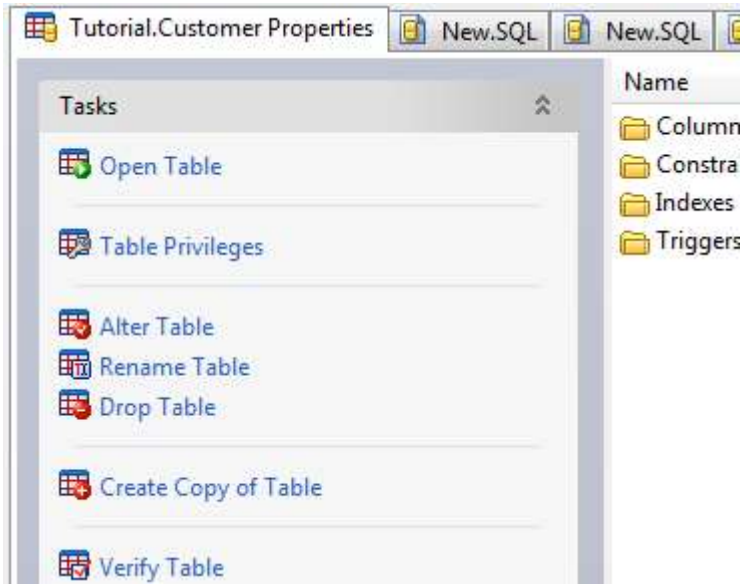
18. Press the **F9** key to execute the SQL statement.



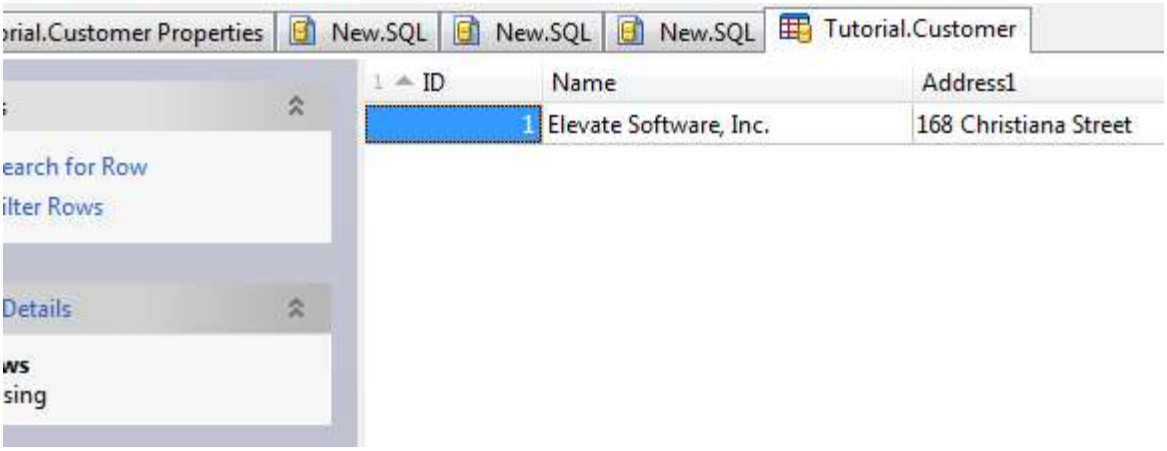
19. Click on the **Customer** table that you just created.



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.

2.3 Creating the Application

The following steps will guide you through creating a basic client-server application using ElevateDB.

> **Note**
> It is assumed that you have already created the required database using the steps outlined in the Creating the Tutorial Database topic, and have configured and started the ElevateDB Server using the steps outlined in the Starting and Configuring the ElevateDB Server topic.

You have now successfully created a basic client-server application for ElevateDB.

Chapter 3
DBISAM Migration

3.1 Introduction

Migrating an existing DBISAM application to ElevateDB is a 3-step process that is outlined below:

Step 1 - Migrating a DBISAM Database Using the ElevateDB Manager or Migrating a DBISAM Database Using Code

The first step is to migrate the existing DBISAM database (or databases) to ElevateDB format. This can be accomplished interactively via the ElevateDB Manager or via the MIGRATE DATABASE statement.

Step 2 - Renaming the DBISAM Components

The second step is to rename any existing DBISAM components in the application to their ElevateDB counterparts. This can be accomplished manually in the Delphi, C++Builder, Borland Developer Studio, CodeGear RAD Studio, Embarcadero RAD Studio, or Lazarus IDE.

Step 3 - Updating the Source Code

The third step is to update the application source code so that it uses the new ElevateDB components. This the most involved step of the migration process.

3.2 Migrating a DBISAM Database Using the ElevateDB Manager

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:

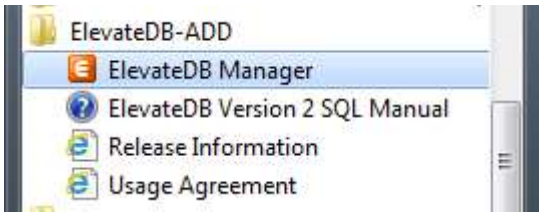
Module	Description
edbmigrate	ElevateDB migrator module
edbmigratedbisam1	DBISAM Version 1.x migrator module
edbmigratedbisam2	DBISAM Version 2.x migrator module
edbmigratedbisam3	DBISAM Version 3.x migrator module
edbmigratedbisam4	DBISAM Version 4.x migrator module
edbmigratebde	BDE (Borland Database 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.

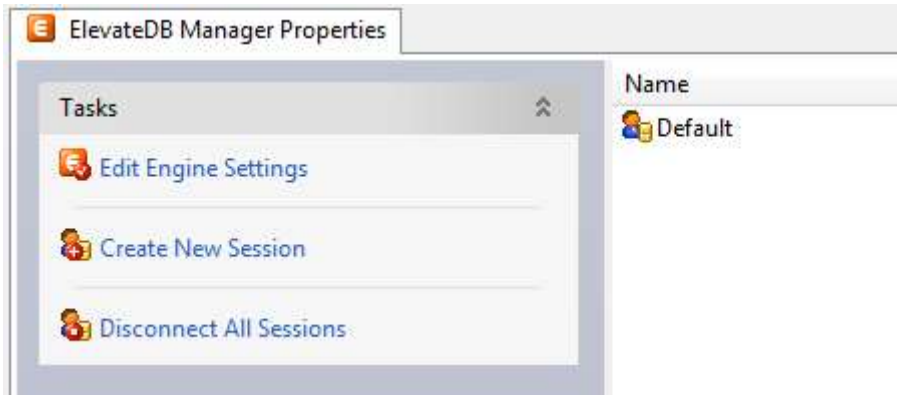
2. Start the ElevateDB Manager (edbmgr.exe).

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

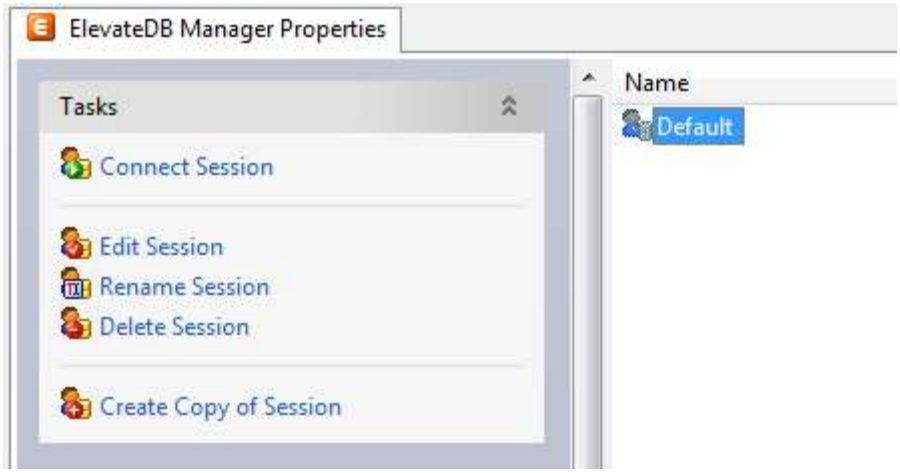


3. Make sure that the session is using the desired character set and configuration file folder (**C:\Tutorial**).

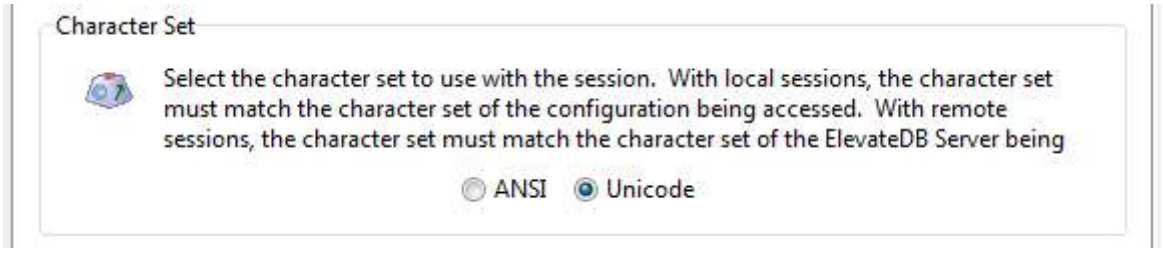
a. Select the **Default** session from the list of available sessions.



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

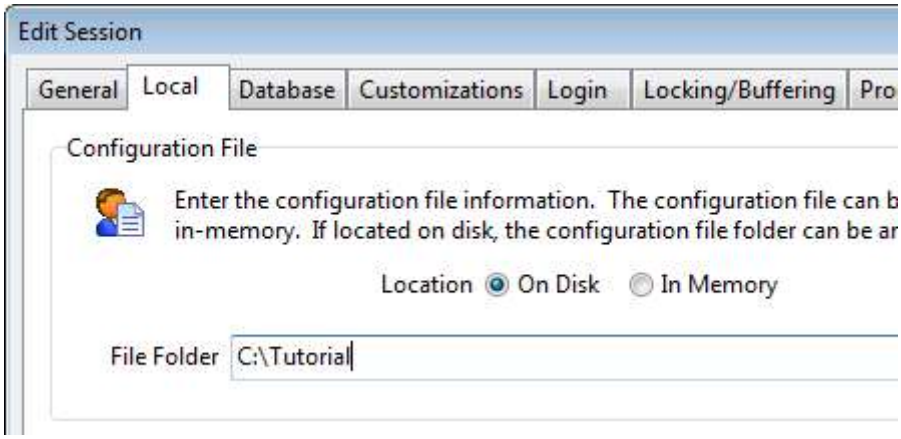


c. On the **General** page of the Edit Session dialog, make sure that the Character Set is set to the desired value - either **ANSI** or **Unicode**.



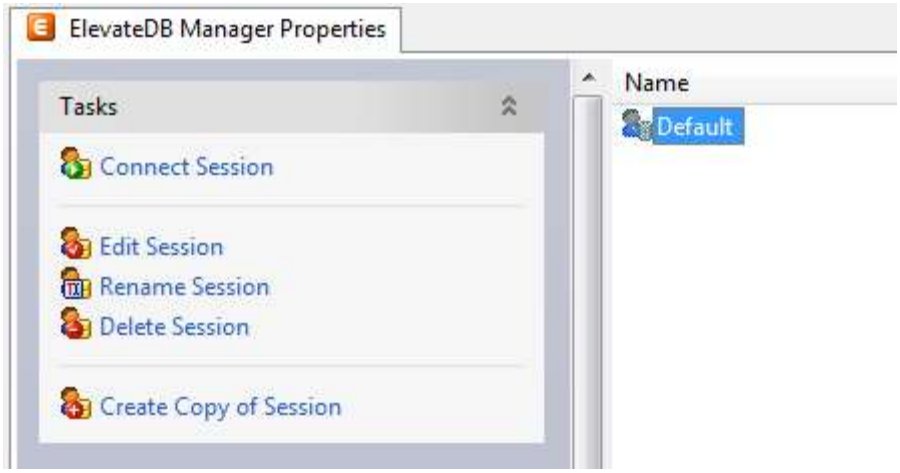
> **Note**
> If you're not sure which character set to select and this is the first time using the ElevateDB Manager, then leave the character set at the default of Unicode.

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

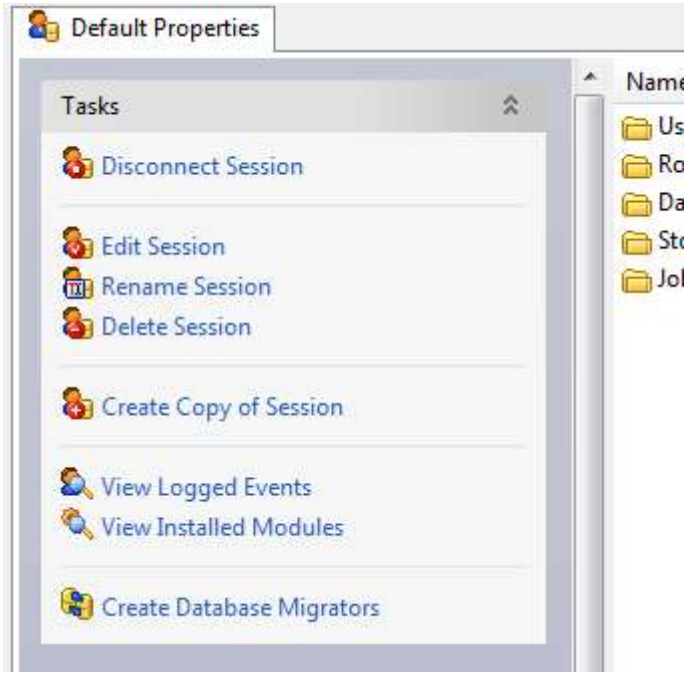


e. Click on the **OK** button.

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

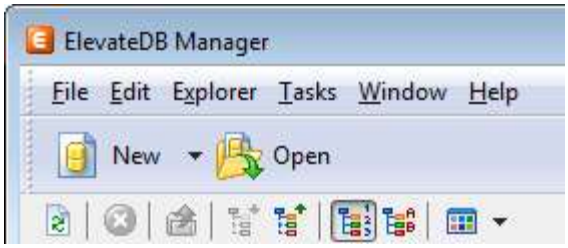


5. In the Tasks pane, click on the **Create Database Migrators** link. This will automatically create all of the database migrators that are shipped with the ElevateDB Manager.



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.

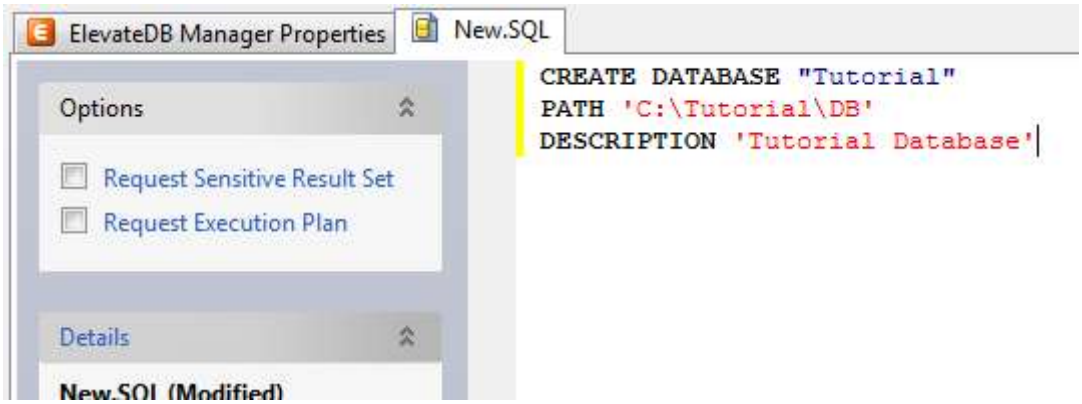
6. Click on the **New** button on the main toolbar.



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

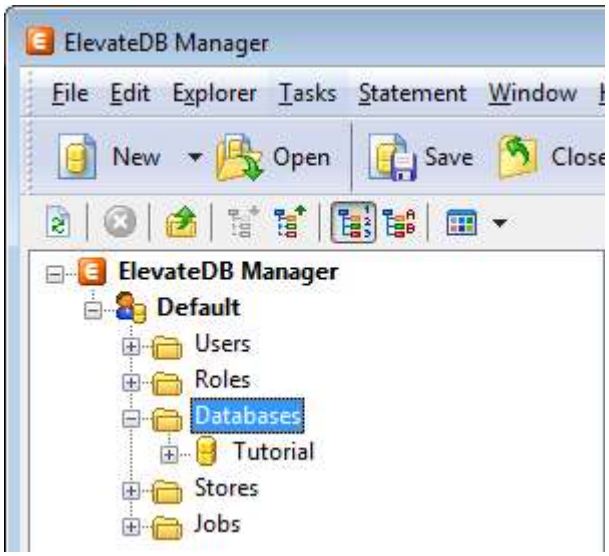
```
CREATE DATABASE "Tutorial"
PATH 'C:\Tutorial\DB'
DESCRIPTION 'Tutorial Database'
```

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

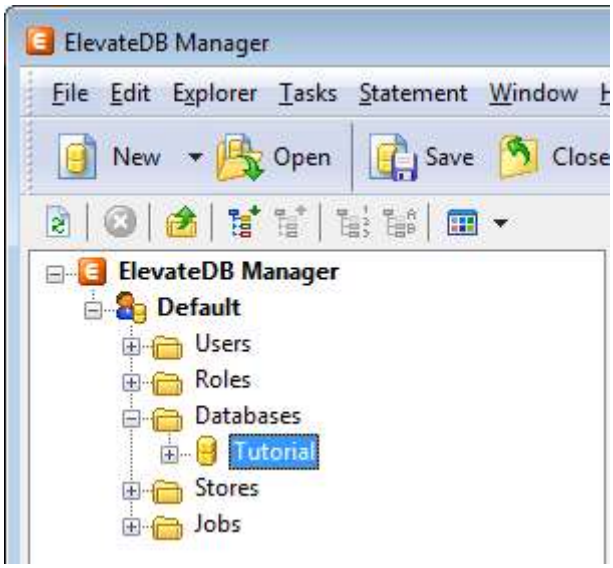


9. Press the **F5** key to refresh the explorer contents for the session.

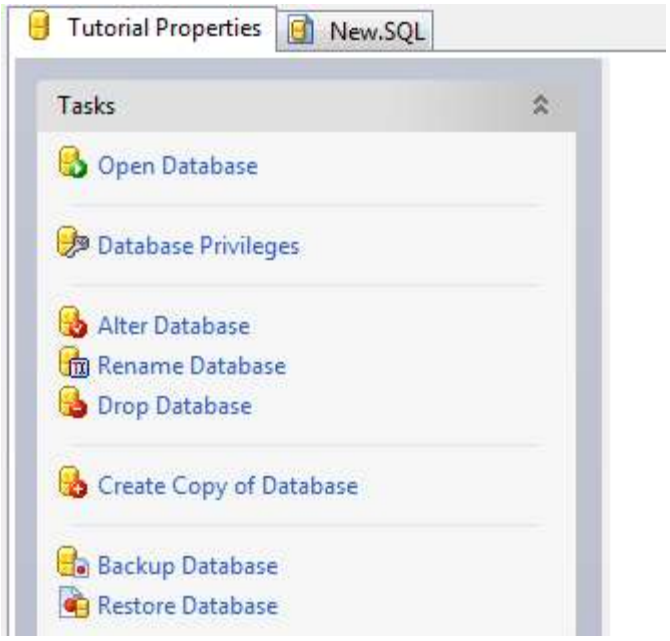
10. Click on the **+** sign next to the **Databases** node in the treeview.



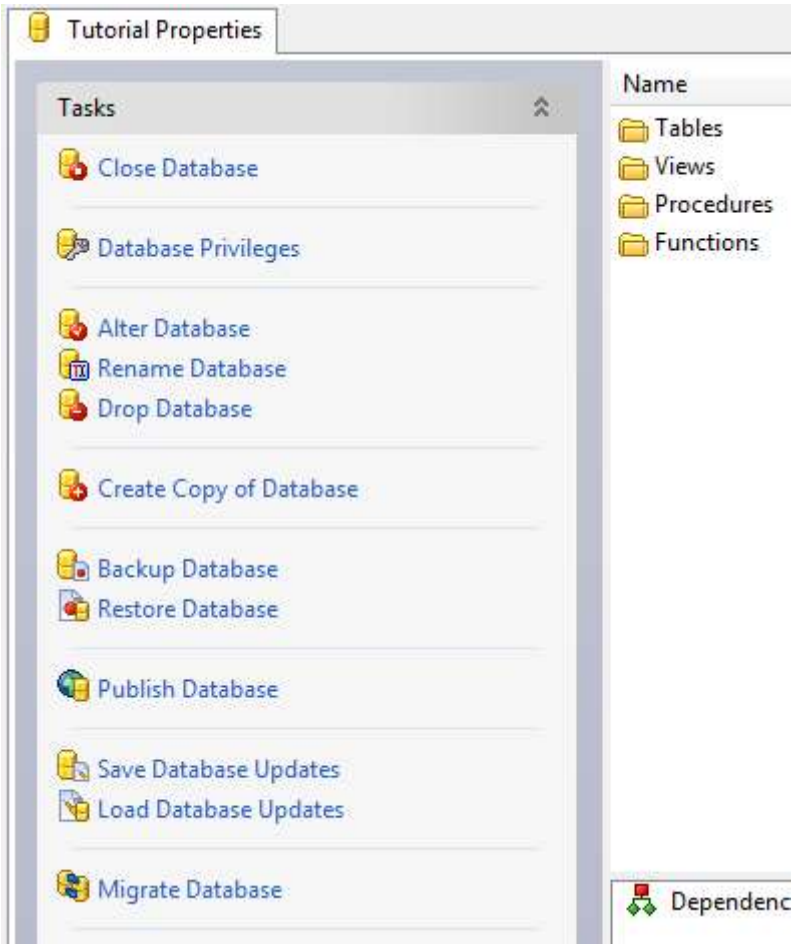
11. Click on the new **Tutorial** database that you just created.



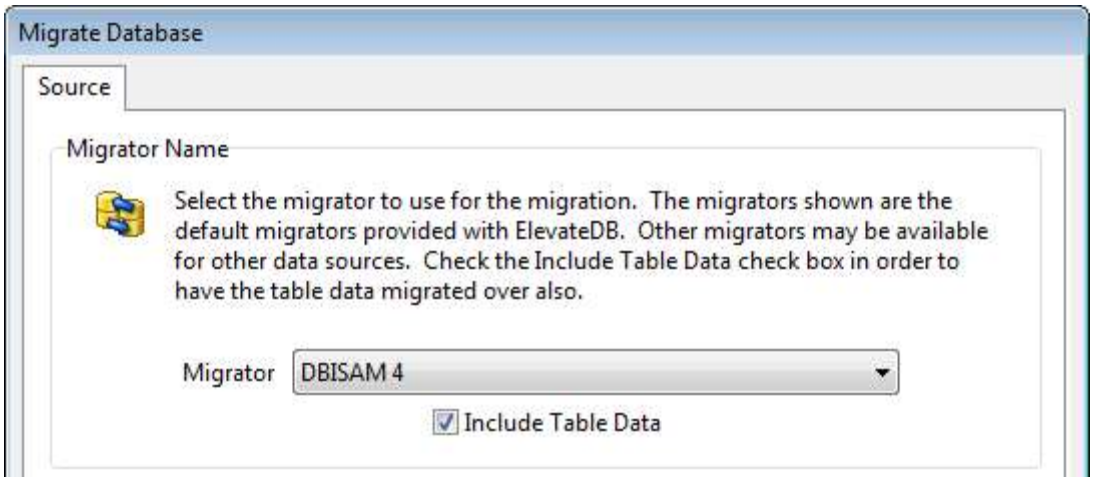
12. Press the **F6** key to make the Properties window the active window, and then click on the **Open Database** link in the Tasks pane.



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



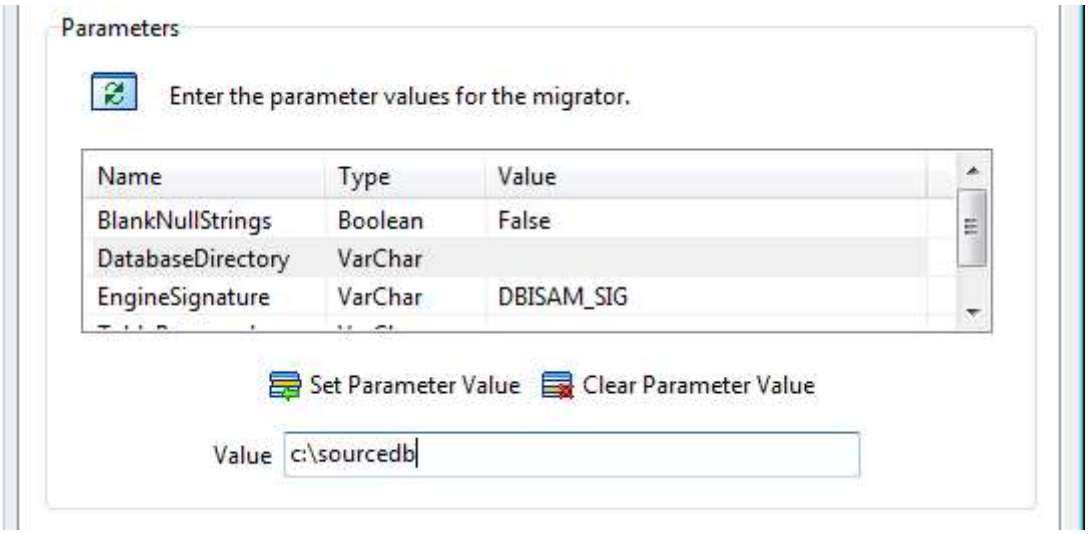
14. Select the desired migrator from the list of migrators.



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.

3.3 Migrating a DBISAM Database Using Code

The following steps will guide you through migrating a database from DBISAM format to ElevateDB format using the DBISAM migrators provided with ElevateDB.

1. Make sure that the DBISAM migrator modules (DLLs) are registered in the configuration file. The DBISAM migrator modules provided with ElevateDB are:

Module	Description
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

You can find these migrator modules as part of the ElevateDB additional software (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 register the required DBISAM migrator module(s), use the CREATE MODULE statement. You can use the TEDBSession Execute method to execute the statement:

```
// This example uses the default Session
// component to register the migrator module using the
// Execute method

with Session do
   Execute('CREATE MODULE "DBISAM4" '+
           'PATH ''C:\Program Files\ElevateDB 2
      ADD\libs\edbmigratedbisam4\unicode\win32\edbmigratedbisam4.dll'''+
           'DESCRIPTION ''DBISAM 4 Migrator''');
```

2. Create a migrator for the desired migrator module using the CREATE MIGRATOR statement. You can use the TEDBSession Execute method to execute the statement:

```
// This example uses the default Session
// component to create the migrator using the
// Execute method

with Session do
   Execute('CREATE MIGRATOR "DBISAM4" '+
           'MODULE "DBISAM4" '+
           'DESCRIPTION ''DBISAM 4 Migrator''');
```

Note
It's important that the MODULE referenced in the CREATE MIGRATOR statement matches the module that you registered first with the CREATE MODULE statement. You'll need to execute both statements for each migrator that you want to use with ElevateDB.

3. If necessary, create the ElevateDB database to use as the target database for the migration using the CREATE DATABASE statement. If you have already created the database or the database already exists, then you can skip this step. You can use the TEDBSession Execute method to execute the statement:

```
// This example uses the default Session
// component to create the database using the
// Execute method

with Session do
   Execute('CREATE DATABASE MyDatabase '+
           'PATH ''c:\mydatabase''');
```

4. Execute the MIGRATE DATABASE statement from the ElevateDB database that you just created, or that already existed. You can use the TEDBDatabase Execute method to execute the statement:

```
// This example uses an existing TEDBDatabase
// component to migrate the database using the
// Execute method

with MyDatabase do
   begin
   DatabaseName:='MyDatabase';
   Database:='MyDatabase';
   Execute('MIGRATE DATABASE FROM "DBISAM4" '+
           'USING DatabaseDirectory = ''c:\dbisamdata'''+
           'WITH DATA');
   end;
```

When the MIGRATE DATABASE statement is executed, the source DBISAM database directory should migrate to the current ElevateDB database. If you would like to display status and progress information during the migration, you can attach event handlers to the TEDBDatabase OnStatusMessage and OnProgress events.

3.4 Renaming the DBISAM Components

The ElevateDB VCL component set for the Delphi, C++Builder, Borland Developer Studio, CodeGear RAD Studio, Embarcadero RAD Studio, and Lazarus products are very similar to their DBISAM counterparts. Therefore, it is possible to simply edit the form files in the IDE and modify the names of the components and their published properties and events so that they will use the ElevateDB components instead.

Updating Components on a Form or Data Module

The following steps will allow you to modify the DBISAM components on a form or data module so that they are compatible with ElevateDB:

> **Warning**
> It is possible to corrupt a form file or otherwise cause the loss of components by not properly completing the following steps. Please be very careful when editing a form file as text and make sure that all defined objects are structured properly.

1. With the form or data module open in the IDE, press the Alt-F12 keys. This will open the form in text mode.

2. Modify the components and their published properties and events as required. Objects are always structured as:

```
object <ObjectName>
<Property or Event Definition>
[<Property or Event Definition>]
[<Property or Event Definition>]
end

<Property or Event Definition> =

<Property or Event Name> = <Value>
```

Collection properties are defined as:

```
<Property or Event Name> = <
<ItemDefinition>
[<ItemDefinition>]
[<ItemDefinition>]
>

<ItemDefinition> =

item
<Property or Event Definition>
[<Property or Event Definition>]
[<Property or Event Definition>]
end
```

3. Press the Alt-F12 keys to return the form to design mode. If there are any properties or events still defined that don't belong to any of the new ElevateDB components, then you will receive a warning and be prompted to remove them from the form definition. Ideally, if you have edited the form entirely so that all published properties and events reflect the new ElevateDB components, you will not receive any errors or warnings.

Component Changes

Detailed information regarding the changes in the existing DBISAM components can be found in the Component Changes topic.

3.5 Updating the Source Code

Updating the source code for an existing DBISAM application so that it works with ElevateDB is a 3-step process that is outlined below:

Step 1 - Rename All Component References

The first step is to rename all component references so that they are using the new ElevateDB component names. You can find information on the component name changes in the Component Changes topic.

Step 2 - Modify All Property, Method, and Event References

The second step is to modify all property, method, and event references so that they are using the new ElevateDB properties, methods, and events. You can find information on the changes to the properties, methods, and events in the Component Changes topic. In many cases you will find that ElevateDB requires an SQL statement to be executed in place of what used to be a method call in DBISAM.

Step 3 - Modify All SQL Statements

The third and final step is to modify all existing DBISAM SQL statements so that they use the new ElevateDB syntax. You can find information on the differences in the SQL implementations of DBISAM and ElevateDB in the SQL Changes topic.

3.6 Component Changes

The following is the list of components in DBISAM and their counterpart in ElevateDB. Click on each component name to find out the changes to the properties, methods, and events for the component.

DBISAM Component	ElevateDB Component
TDBISAMEngine	TEDBEngine
TDBISAMSession	TEDBSession
TDBISAMDatabase	TEDBDatabase
TDBISAMDataSet	TEDBDataSet
TDBISAMDBDataSet	TEDBDBDataSet
TDBISAMTable	TEDBTable
TDBISAMQuery	TEDBQuery
None	TEDBStoredProc
TDBISAMUpdateSQL	TEDBUpdateSQL
EDBISAMEngineError	EEDBError

3.7 TDBISAMEngine Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
CreateTempTablesInDatabase	This property is no longer necessary. ElevateDB always creates temporary tables used in optimizing, repairing, or altering tables in the same location as the tables themselves.
FilterRecordCounts	This property is no longer necessary. ElevateDB does not provide logical record numbers (sequence numbers).
Functions	This property is no longer necessary. ElevateDB uses SQL to create and drop functions, and a special Information Schema for storing the available functions in a given database. Please see the CREATE FUNCTION, DROP FUNCTION, and Functions Table topics for more information.
MaxTableBlobBufferCount MaxTableBlobBufferSize MaxTableDataBufferCount MaxTableDataBufferSize MaxTableIndexBufferCount MaxTableIndexBufferSize	These properties are no longer necessary. ElevateDB allows the buffering settings to be set on a per-table basis for each table when the table is created or altered. Please see the CREATE TABLE, ALTER TABLE, and Tables Table topics for more information.
ServerAdminAddress ServerAdminPort ServerAdminThreadCacheSize	These properties are no longer necessary. ElevateDB uses one port for both normal connections and administrative connections, and both types of operations can be performed using only one connection.
ServerConfigPassword	This property is no longer necessary. ElevateDB uses one encryption password per application for all encryption, and it is represented by the EncryptionPassword property.
TableBlobBackupExtension TableBlobTempExtension TableBlobUpgradeExtension TableDataBackupExtension TableDataTempExtension TableDataUpgradeExtension TableIndexBackupExtension TableIndexTempExtension TableIndexUpgradeExtension	These properties have been removed and replaced with the hard-coded value of ".Old". ElevateDB simply appends the ".Old" to the existing file when creating backup copies during the optimization, alteration, or repair of tables.
TableFilterIndexThreshhold	This property is no longer required under ElevateDB and has been removed.
TableMaxReadLockCount	This property is no longer necessary. For performance reasons, ElevateDB does not relinquish read locks when performing table scans in order to satisfy a query or filter condition.

TableReadLockTimeout TableTransLockTimeout TableWriteLockTimeout	These properties are no longer required under ElevateDB and have been removed

Methods

Removed	Description
AddServerDatabase ModifyServerDatabase DeleteServerDatabase GetServerDatabase GetServerDatabaseNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop databases, and a special Configuration database for storing the available databases in a given configuration. Please see the CREATE DATABASE, DROP DATABASE, and Databases Table topics for more information.
AddServerDatabaseUser ModifyServerDatabaseUser DeleteServerDatabaseUser GetServerDatabaseUser GetServerDatabaseUserNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop users and roles, and a special Configuration database for storing the available users and roles in a given configuration. ElevateDB also uses SQL for granting and revoking privileges on databases and other objects for existing users and roles. Please see the CREATE USER, DROP USER, CREATE ROLE, DROP ROLE, GRANT ROLES, GRANT PRIVILEGES, Users Table, Roles Table, UserRoles Table, and DatabasePrivileges Table topics for more information.
AddServerEvent ModifyServerEvent DeleteServerEvent GetServerEvent GetServerEventNames	These methods are no longer necessary. ElevateDB offers jobs, which are the same thing as scheduled events in DBISAM. ElevateDB uses SQL to create and drop jobs, and a special Configuration database for storing the available jobs in a given configuration. Please see the CREATE JOB, DROP JOB, and Jobs Table topics for more information.
AddServerProcedure ModifyServerProcedure DeleteServerProcedure GetServerProcedure GetServerProcedureNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop procedures, and a special Information Schema for storing the available procedures in a given database. Please see the CREATE PROCEDURE, DROP PROCEDURE, and Procedures Table topics for more information.
AddServerProcedureUser ModifyServerProcedureUser DeleteServerProcedureUser GetServerProcedureUser GetServerProcedureUserNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop users and roles, and a special Configuration database for storing the available users and roles in a given configuration. ElevateDB also uses SQL for granting and revoking privileges on procedures and other objects for existing users and roles. Please see the CREATE USER, DROP USER, CREATE ROLE, DROP ROLE, GRANT ROLES, GRANT PRIVILEGES, Users Table, Roles Table, UserRoles Table, and ProcedurePrivileges Table topics for more information.
AddServerUser ModifyServerUser DeleteServerUser GetServerUser GetServerUserNames ModifyServerUserPassword	These methods are no longer necessary. ElevateDB uses SQL to create and drop users, and a special Configuration database for storing the available users in a given configuration. Please see the CREATE USER, ALTER USER, DROP USER, and Users Table topics for more information.

BuildWordList GetDefaultTextIndexParams	These methods are no longer supported. Word generation and text filtering for text indexes is directly tied to the defined text indexes in ElevateDB, so these methods are no longer possible. Please see the Text Indexing topic for more information.
ConvertIDToLocaleConstant ConvertLocaleConstantToID GetLocaleNames IsValidLocale IsValidLocaleConstant	These methods are no longer necessary. ElevateDB uses a special Configuration database for storing the available collations (locales) in a given configuration. Please see the Collations Table topic for more information.
GetServerConfig ModifyServerConfig	These methods are no longer necessary. ElevateDB stores all server startup and operational information in the TEDBEngine component itself, and all additional configuration information, such as the defined databases, users, roles, and jobs, is stored in the server configuration file. The information in the server configuration file can be accessed via the special Configuration database available for each configuration. Please see the Configuration Database topic for more information.
GetServerLogCount GetServerLogRecord	These methods are no longer necessary. ElevateDB logs all error, warning, and information events in a special binary log file available for each configuraton. The information in the log file can be accessed via the special Configuration database available for each configuration. Please see the LogEvents Table topic for more information.
GetServerMemoryUsage	This method is no longer supported, and was deprecated in the latest DBISAM versions.
GetServerSessionInfo	This method is no longer supported. Use the OnServerSessionEvent event along to track session information as sessions are created, connected, etc.
StartAdminServer StopAdminServer StartMainServer StopMainServer	These methods are no longer necessary. ElevateDB uses one port for both normal connections and administrative connections, and both types of operations can be performed using only one connection. In addition, the ElevateDB server is automatically stopped and started when the TEDBEngine Active property is assigned a new value.

Events

Removed	Description

AfterDeleteTrigger AfterInsertTrigger AfterUpdateTrigger BeforeDeleteTrigger BeforeInsertTrigger BeforeUpdateTrigger	These methods are no longer necessary. ElevateDB uses SQL to create and drop triggers, and a special Information Schema for storing the available triggers defined for the tables in a given database. Please see the CREATE TRIGGER, DROP TRIGGER, and Triggers Table topics for more information.
OnDeleteError OnInsertError OnUpdateError	These events are no longer supported.
OnCompress OnDecompress	These events are no longer supported. ElevateDB does not allow for custom compression due to the need for it to run as managed code under .NET.
OnCryptoInit OnCryptoReset OnDecryptBlock OnEncryptBlock	These events are no longer supported. ElevateDB does not allow for custom encryption due to the need for it to run as managed code under .NET.
OnCustomFunction	This event is no longer necessary. ElevateDB uses SQL to create and drop functions, and a special Information Schema for storing the available functions in a given database. Please see the CREATE FUNCTION, DROP FUNCTION, and Functions Table topics for more information.
OnServerConnect OnServerDisconnect OnServerLogin OnServerLogout OnServerReconnect	These events have been removed and replaced with the single OnServerSessionEvent event in ElevateDB. See below for more information on the new OnServerSessionEvent event.
OnServerLogCount OnServerLogEvent OnServerLogRecord	These events are no longer necessary. ElevateDB logs all error, warning, and information events in a special binary log file available for each configuraton. The information in the log file can be accessed via the special Configuration database available for each configuration. Please see the LogEvents Table topic for more information.
OnServerProcedure	This event is no longer necessary. ElevateDB uses SQL to create and drop procedures, and a special Information Schema for storing the available procedures in a given database. Please see the CREATE PROCEDURE, DROP PROCEDURE, and Procedures Table topics for more information.
OnServerScheduledEvent	This event is no longer necessary. ElevateDB offers jobs, which are the same thing as scheduled events in DBISAM. ElevateDB uses SQL to create and drop jobs, and a special Configuration database for storing the available jobs in a given configuration. Please see the CREATE JOB, DROP JOB, and Jobs Table topics for more information.
OnTextIndexFilter OnTextIndexTokenFilter	These events are no longer supported. Word generation and text filtering for text indexes is directly tied to the defined text indexes in ElevateDB, so these methods are no longer possible. Please see the Text Indexing topic for more information.

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description
EngineSignature	This property has been renamed to the Signature property.
LockFileName	This property has been split into two properties. In ElevateDB, the ConfigName property or CatalogName property is combined with the LockExtension property to name the lock file for either the configuration or a given database catalog.
ServerConfigFileName	This property has been split into two properties. In ElevateDB, the ConfigName property is combined with the ConfigExtension property to name the configuration file. The ConfigPath property is used to determine where the configuration file is created. ElevateDB uses a configuration file for local applications as well as the ElevateDB Server, whereas DBISAM only used a configuration file for the DBISAM Database Server.
ServerEncryptionPassword	This property has been renamed to the EncryptionPassword property. ElevateDB uses the EncryptionPassword property for all encryption in the application.
ServerLicensedConnections	This property has been renamed to the LicensedSessions property. ElevateDB supports session count restrictions based upon the LicensedSessions property for both local applications and the ElevateDB server.
ServerMainAddress ServerMainPort ServerMainThreadCacheSize	These properties have been renamed with the "Main" portion stripped out. ElevateDB uses one port for both normal connections and administrative connections, and both types of operations can be performed using only one connection.
TableDataExtension TableIndexExtension TableBlobExtension	These proeprties have renamed to the TableExtension property, the TableIndexExtension property, and the TableBlobExtension property, respectively.

Methods

Changed	Description

AnsiStrToBoolean AnsiStrToCurr AnsiStrToDate AnsiStrToDateTime AnsiStrToFloat AnsiStrToTime BooleanToAnsiStr CurrToAnsiStr DateToAnsiStr DateTimeToAnsiStr FloatToAnsiStr TimeToAnsiStr	These methods have been renamed with the "Ansi" portion replaced with "SQL". This was done to reflect that these methods now work with both ANSI strings and Unicode (wide) strings.

Events

Changed	Description
OnServerStart OnServerStop OnShutdown OnStartup	These events have been replaced with the BeforeStart, AfterStart, BeforeStop, and AfterStop events. Also, the new events apply regardless of whether the engine component is configured to run as a client engine or a server engine via the EngineType property.

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
BackupExtension	This property is used to specify the extension used for ElevateDB backup files. Please see the BACKUP DATABASE, RESTORE DATABASE, and Backups Table topics for more information.
UpdateExtension	This property is used to specify the extension used for ElevateDB update files. Please see the SAVE UPDATES, LOAD UPDATES, and Updates Table topics for more information.
TablePublishExtension	This property is used to specify the extension used for the publish files associated with published ElevateDB tables. Please see the PUBLISH DATABASE, UNPUBLISH DATABASE, and Tables Table topics for more information.
CatalogName CatalogExtension	These two properties are combined together to specify the file name used by ElevateDB for all database catalogs.
LogExtension LogCategories MaxLogFileSize	These properties are used in ElevateDB to control the naming of the log file, what types of events are logged in the log file, and the maximum log file size. ElevateDB combines the ConfigName property with the LogExtension property to name the log file, and the log file is always created in the path specified by the ConfigPath property. The log file in ElevateDB is a ciruclar log file, and the MaximumLogFileSize determines at which file size ElevateDB starts to re-use the log file space

	of the oldest log entries with the newer log entries.
ServerAuthorizedAddresses ServerBlockedAddresses ServerDeadSessionExpiration ServerDeadSessionInterval ServerMaxDeadSessions ServerSessionTimeout	These properties were added to replace the same server configuration file settings that were available in the DBISAM Database Server.
ServerRunJobs ServerJobCategory	These properties determine whether the ElevateDB Server can run jobs, and if so, what category of jobs it should run.
TempTablesPath	This property specifies where any temporary tables created by the engine will be stored.

Methods

New	Description
GetTempTablesPath	This method returns the operating system-defined temporary files path.
DayTimeIntervalToSQLStr YearMonthIntervalToSQLStr SQLStrToDayTimeInterval SQLStrToYearMonthInterval	These four methods are used to convert SQL intervals, either day-time intervals or year-month intervals, to and from strings. Please see the Interval Types topic for more information.

Events

New	Description
None	

3.8 TDBISAMSession Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
CurrentServerUser	This property is no longer necessary. ElevateDB uses SQL for procedures and functions.
PrivateDir	This property is no longer necessary. ElevateDB uses one temporary tables property setting, the TempTablesPath property, for all sessions.
RemoteEncryptionPassword	This property is no longer necessary. ElevateDB uses one encryption password per application for all encryption, and it is represented by the EncryptionPassword property.
RemoteParams	This property is no longer necessary. ElevateDB uses SQL for procedures and the TEDBStoredProc component for executing the procedures.
StrictChangeDetection	This property is no longer supported. ElevateDB does not support strict change detection.

Methods

Removed	Description
AddPassword GetPassword RemoveAllPasswords RemotePassword	These methods are no longer supported. ElevateDB offers a complete user security architecture that surpasses simple password access to individual tables. Please see the User Security topic for more information.
AddRemoteDatabase ModifyRemoteDatabase DeleteRemoteDatabase GetRemoteDatabase GetRemoteDatabaseNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop databases, and a special Configuration database for storing the available databases in a given configuration. Please see the CREATE DATABASE, DROP DATABASE, and Databases Table topics for more information.
AddRemoteDatabaseUser ModifyRemoteDatabaseUser DeleteRemoteDatabaseUser GetRemoteDatabaseUser GetRemoteDatabaseUserNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop users and roles, and a special Configuration database for storing the available users and roles in a given configuration. ElevateDB also uses SQL for granting and revoking privileges on databases and other objects for existing users and roles. Please see the CREATE USER, DROP USER, CREATE ROLE, DROP ROLE, GRANT ROLES, GRANT PRIVILEGES, Users Table, Roles Table, UserRoles Table, and DatabasePrivileges Table topics for more information.
AddRemoteEvent ModifyRemoteEvent	These methods are no longer necessary. ElevateDB offers jobs, which are the same thing as scheduled events in

DeleteRemoteEvent GetRemoteEvent GetRemoteEventNames	DBISAM. ElevateDB uses SQL to create and drop jobs, and a special Configuration database for storing the available jobs in a given configuration. Please see the CREATE JOB, DROP JOB, and Jobs Table topics for more information.
AddRemoteProcedure ModifyRemoteProcedure DeleteRemoteProcedure GetRemoteProcedure GetRemoteProcedureNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop procedures, and a special Information Schema for storing the available functions in a given database. Please see the CREATE PROCEDURE, DROP PROCEDURE, and Procedures Table topics for more information.
AddRemoteProcedureUser ModifyRemoteProcedureUser DeleteRemoteProcedureUser GetRemoteProcedureUser GetRemoteProcedureUserNames	These methods are no longer necessary. ElevateDB uses SQL to create and drop users and roles, and a special Configuration database for storing the available users and roles in a given configuration. ElevateDB also uses SQL for granting and revoking privileges on procedures and other objects for existing users and roles. Please see the CREATE USER, DROP USER, CREATE ROLE, DROP ROLE, GRANT ROLES, GRANT PRIVILEGES, Users Table, Roles Table, UserRoles Table, and ProcedurePrivileges Table topics for more information.
AddRemoteUser ModifyRemoteUser ModifyRemoteUserPassword DeleteRemoteUser GetRemoteUser GetRemoteUserNames ModifyRemoteUserPassword	These methods are no longer necessary. ElevateDB uses SQL to create and drop users, and a special Configuration database for storing the available users in a given configuration. Please see the CREATE USER, ALTER USER, DROP USER, and Users Table topics for more information.
CallRemoteProcedure RemoteParamByName SendProcedureProgress	These methods are no longer necessary. ElevateDB uses SQL for procedures and the TEDBStoredProc component for executing the procedures.
DisconnectRemoteSession RemoveRemoteSession	These methods are no longer necessary. ElevateDB uses the DISCONNECT SERVER SESSION and REMOVE SERVER SESSION statements to disconnect and remove server sessions on an ElevateDB Server. You can issue these statements via the new Execute method.
GetRemoteAdminAddress GetRemoteAdminPort GetRemoteAdminThreadCacheSize GetMainAdminAddress GetMainAdminPort GetMainAdminThreadCacheSize	These methods are no longer necessary. ElevateDB uses one port for both normal connections and administrative connections, and both types of operations can be performed using only one connection. In addition, the address, port, and thread cache size parameters for an ElevateDB server are not configurable remotely and must be configured prior to starting an ElevateDB server.
GetRemoteConfig ModifyRemoteConfig	These methods are no longer necessary. ElevateDB stores all server startup and operational information in the TEDBEngine component itself, and all additional configuration information, such as the defined databases, users, roles, and jobs, is stored in the server configuration file. The information in the server configuration file can be accessed via the special Configuration database available for each configuration. Please see the Configuration Database topic for more information.

GetRemoteConnectedSessionCount GetRemoteSessionCount GetRemoteSessionInfo	These methods are no longer necessary. ElevateDB uses SQL to query any ElevateDB server sessions, and a special Configuration database for storing the server sessions on a given ElevateDB server. Please see the ServerSessions Table topic for more information.
GetRemoteLogCount GetRemoteLogRecord	These methods are no longer necessary. ElevateDB logs all error, warning, and information events in a special binary log file available for each configuraton. The information in the log file can be accessed via the special Configuration database available for each configuration. Please see the LogEvents Table topic for more information.
GetRemoteMemoryUsage	This method is no longer supported, and was deprecated in the latest DBISAM versions.
GetRemoteUpTime	This method is no longer supported.
RemoveAllRemoteMemoryTables	This method is no longer supported.
StartRemoteServer StopRemoteServer	These methods are no longer supported. The ElevateDB server cannot be remotely stopped and started.

Events

Removed	Description
OnPassword	This event is no longer supported. ElevateDB offers a complete user security architecture that surpasses simple password access to individual tables. Please see the User Security topic for more information.

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description

Active	This property has been renamed to the Connected property.
CurrentRemoteUser	This property has been renamed to the CurrentUser property. ElevateDB requires a user login for both local and remote sessions.
LockProtocol LockRetryCount LockWaitTime	These properties have been renamed and prefixed with "Record" in ElevateDB in order to make clear that these properties deal with row locking exclusively.
ProgressSteps	This property has been changed to the ProgressTimeInterval property, which uses a time interval instead of a fixed number of progress steps to ensure that progress updates still take place in a reasonable span of time irrespective of the length or scope of a given operation.
RemoteUser RemotePassword	These properties have been renamed to the LoginUser and LoginPassword properties, respectively. ElevateDB requires a user login for both local and remote sessions.

Methods

Changed	Description
GetRemoteEngineVersion	This method has been renamed to the GetRemoteServerVersion method.

Events

Changed	Description
OnRemoteLogin	This event has been renamed to the OnLogin event. ElevateDB requires a user login for both local and remote sessions.
OnRemoteTrace	This event uses a different record type for the trace record that is passed as a parameter to the event handler.
OnShutdown OnStartup	These events have been replaced with the BeforeConnect, AfterConnect, BeforeDisconnect, and AfterDisconnect events. Also, the new events apply regardless of whether the session component is configured to run as a remote session or a local session via the SessionType property.

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description

KeepTablesOpen	This property has been moved from the database level to the session level in ElevateDB. This gives the developer the ability to control whether tables should be kept open even in SQL procedures or functions in addition to controlling whether tables should be kept open during normal table and query processing.
RecordChangeDetection	This property was added to allow the developer to specify whether changes to a row will issue a warning exception when the row is updated or deleted. In DBISAM this behavior was not configurable and any changes to a row would cause an #8708 (DBISAM_KEYORRECDELETED) exception to be raised.
SessionDescription	This property allows the developer to specify a description for the session.
SQLStmtCacheSize	This property allows the developer to specify an SQL statement cache size all open databases in the session.
FuncProcCacheSize	This property allows the developer to specify a function/procedure cache size all open databases in the session.
ExcludeFromLicensedSessions	This property specifies whether the current session should be included in the session license count controlled by the TEDBEngine LicensedSessions property for local sessions, or by the ElevateDB Server for remote sessions.

Methods

New	Description
CalculateCRC32ForStream	This method calculates a CRC32 checksum for a stream.
Execute	This method allows you to execute an SQL statement against the special Configuration database. This is useful for performing configuration-level queries or operations.
GetStoredProcNames	This method populates a list with the names of all stored procedures and functions defined within the specified database.
SaveStoreFileToStream	This method loads a store file into a stream.
SaveStreamToStoreFile	This method saves a stream to a store file.
FreeCachedSQLStmts	This method allows you to free all cached SQL statements for a specific open database, or for all open databases.
FreeCachedFuncProcs	This method allows you to free all cached functions/procedures for a specific open database, or for all open databases.

Events

New	Description
None	

3.9 TDBISAMDatabase Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
KeepTablesOpen	This property has been moved to the session level and the TEDBSession component.

Methods

Removed	Description
Backup BackupInfo Restore	These methods are no longer necessary. ElevateDB uses SQL for backing up and restoring databases, as well as retrieving information about backups from disk, and a special Configuration database for storing the available backups in a given configuration. Please see the BACKUP DATABASE, RESTORE DATABASE, SET BACKUPS STORE, and Backups Table topics for more information.

Events

Removed	Description
None	

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description
Directory RemoteDatabase	These properties have been replaced by the single Database property. ElevateDB uses SQL to create and drop databases, and a special Configuration database for storing the available databases in a given configuration. Please see the CREATE DATABASE, DROP DATABASE, and Databases Table topics for more information.

Methods

Changed	Description

StartTransaction	The StartTransaction method accepts a list of tables as a string array instead of a TStrings object, and there is one additional parameter for specifying the transaction lock timeout in milliseconds.

Events

Changed	Description
OnBackupLog OnBackupProgress OnRestoreLog OnRestoreProgress	These events have been replaced with the OnLogMessage, OnProgress, and OnStatusMessage events.

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
None	

Methods

New	Description
TableInTransaction	The TableInTransaction method is used to determine if a specific table is involved in the current transaction.

Events

New	Description
None	

3.10 TDBISAMDataSet Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
AutoDisplayLabels	This property is no longer supported.
FilterOptimizeLevel	This property is no longer supported. Eventually it will be replaced by a FilterPlan property instead.
FilterRecordCount	This property is no longer necessary. ElevateDB does not provide logical record numbers (sequence numbers).
KeySize	This property has been moved to the TEDBTable component.
RecordHash RecordID	These properties are no longer necessary. ElevateDB does not use record hashes or IDs.

Methods

Removed	Description
ExportTable ImportTable	These methods are no longer necessary. ElevateDB uses SQL for importing and exporting tables to and from delimited text. Please see the EXPORT TABLE and IMPORT TABLE topics for more information.

Events

Removed	Description
OnCachedUpdateError	This event is not used anymore because ElevateDB uses ERROR triggers for handling update errors. Please see the CREATE TRIGGER topic in the ElevateDB SQL Manual for more information.
OnLoadFromStreamProgress OnSaveToStreamProgress	These events are no longer supported. ElevateDB streams should be kept fairly small since they are stored in memory. Any stream that is large enough to require progress updates is probably too large and should be handled differently.

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description

RecNo	This property no longer returns a logical record number as it did in DBISAM. It returns zero (0) at all times under ElevateDB. However, you can still assign a value to the property in order to navigate to a specific logical row in the dataset.

Methods

Changed	Description
IsSequenced	This method always returns False under ElevateDB. ElevateDB does not provide logical record numbers (sequence numbers).
LoadFromStream SaveToStream	ElevateDB uses a completely different stream format than DBISAM. Do not attempt to load a stream created by DBISAM into ElevateDB, or vice-versa.

Events

Changed	Description
None	

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
None	

Methods

New	Description
LockCurrentRecord UnlockCurrentRecord UnlockAllRecords	These methods allow you to manually lock and unlock rows in the current cursor.

Events

New	Description
None	

3.11 TDBISAMDBDataSet Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
None	

Methods

Removed	Description
None	

Events

Removed	Description
None	

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description
None	

Methods

Changed	Description
None	

Events

Changed	Description
None	

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
None	

Methods

New	Description
None	

Events

New	Description
None	

3.12 TDBISAMTable Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
LocaleID Description Encrypted Password IndexPageSize BlobBlockSize LastAutoIncValue TextIndexFields TextIndexIncludeChars TextIndexSpaceChars TextIndexStopWords UserMajorVersion UserMinorVersion	These properties are no longer necessary. ElevateDB maintains all database metadata in the special Information Schema for each database. The Information schema tables can be queried like any normal tables for information on the structure of tables, columns, indexes, etc.
Exists	This property is no longer necessary. To determine if a table or view exists in a database, query the special Information Schema for the database.
FullTableName LastUpdated TableSize	These properties are no longer supported. The TEDBTable component supports opening both tables and views. Therefore, returning the physical characteristics of a table is not feasible in all cases.
VersionNum	This property is no longer necessary.

Methods

Removed	Description

CreateTable AlterTable CopyTable RenameTable DeleteTable AddIndex DeleteIndex DeleteAllIndexes	These methods are no longer necessary. ElevateDB uses SQL for all table and index creation, alteration, or drops. Please see the CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, CREATE TEXT INDEX, and DROP INDEX topics for more information.
LockSemaphore UnlockSemaphore	These methods are no longer supported. ElevateDB does not support semaphore locks.
LockTable UnlockTable TableIsLocked	These methods are no longer supported. ElevateDB does not support table locks. Instead, it supports manual row locking via the LockCurrentRecord, UnlockCurrentRecord, and UnlockAllRecords methods.
OptimizeTable RepairTable VerifyTable UpgradeTable	These methods are no longer necessary. ElevateDB uses SQL for all administrative functionality. Please see the OPTIMIZE TABLE and REPAIR TABLE topics for more information.

Events

Removed	Description
OnAlterProgress OnDataLost OnCopyProgress OnIndexProgress	These events are no longer necessary. ElevateDB uses SQL for all table and index creation, alteration, or drops, and the OnLogMessage, OnProgress, and OnStatusMessage events provide the same functionality.
OnExportProgress OnImportProgress	These events are no longer necessary. ElevateDB uses SQL for importing and exporting tables, and the OnLogMessage, OnProgress, and OnStatusMessage events provide the same functionality.
OnOptimizeProgress OnRepairProgress OnRepairLog OnVerifyProgress OnVerifyLog OnUpgradeProgress OnUpgradeLog	These events are no longer necessary. ElevateDB uses SQL for all administrative functionality, and the OnLogMessage, OnProgress, and OnStatusMessage events provide the same functionality.

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description

FieldDefs	This property no longer uses a custom TDBISAMFieldDefs type for the field definitions collection. In ElevateDB this property uses the standard TFieldDefs collection type.
IndexDefs	This property no longer uses a custom TDBISAMIndexDefs type for the index definitions collection. In ElevateDB this property uses the standard TIndexDefs collection type.
TableName	This property now accepts a view name in addition to a table name. Furthermore, the drop-down combo box for this property in the Object Inspector will contain all tables and views defined for the database.

Methods

Changed	Description
None	

Events

Changed	Description
None	

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
None	

Methods

New	Description
None	

Events

New	Description
None	

3.13 TDBISAMQuery Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
TableName	This property is no longer supported.

Methods

Removed	Description
SaveToTable	This method is no longer supported. In ElevateDB, use the AS clause of the CREATE TABLE to create a table that is based upon a query expression.

Events

Removed	Description
BeforeExecute AfterExecute OnGetParams OnQueryError	These events are no longer supported. ElevateDB does not support multi-statement scripts in the TEDBQuery component.
OnAlterProgress OnDataLost OnCopyProgress	These events are no longer necessary. ElevateDB uses SQL for all table and index creation, alteration, or drops, and the OnLogMessage, OnProgress, and OnStatusMessage events provide the same functionality.
OnExportProgress OnImportProgress	These events are no longer necessary. ElevateDB uses SQL for importing and exporting tables, and the OnLogMessage, OnProgress, and OnStatusMessage events provide the same functionality.
OnOptimizeProgress OnRepairProgress OnRepairLog OnVerifyProgress OnVerifyLog OnUpgradeProgress OnUpgradeLog	These events are no longer necessary. ElevateDB uses SQL for all administrative functionality, and the OnLogMessage, OnProgress, and OnStatusMessage events provide the same functionality.
OnQueryProgress	This event is no longer necessary. The OnProgress event provides the same functionality.
OnSaveProgress	This event is no longer supported since the SaveToTable method is no longer supported. In ElevateDB, use the AS clause of the CREATE TABLE to create a table that is based upon a query expression.

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description
GeneratePlan	This property has been renamed to the RequestPlan property.
Params	This property no longer uses a custom TDBISAMParams type for the parameter definitions collection. In ElevateDB this property uses the standard TParams collection type.
RequestLive	This property has been renamed to the RequestSensitive property.
ResultIsLive	This property has been renamed to the Sensitive property.
SQL	This property only accepts a single SQL statement in ElevateDB. DBISAM allow for multi-statement scripts.
SQLStatementType	This property has been renamed to the StatementType property.
StmtHandle	This property has been renamed to the StatementHandle property.

Methods

Changed	Description
None	

Events

Changed	Description
None	

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
Constrained	This property allows you to specify that any inserts or updates made to a sensitive result set be subject to the WHERE clause used in the current SELECT statement.

Methods

New	Description

None	

Events

New	Description
None	

3.14 TDBISAMUpdateSQL Component

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
None	

Methods

Removed	Description
None	

Events

Removed	Description
None	

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description
None	

Methods

Changed	Description
None	

Events

Changed	Description
None	

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
None	

Methods

New	Description
None	

Events

New	Description
None	

3.15 EDBISAMEngineError Object

Removed Properties, Methods and Events

The following are the properties, methods, and events that have been removed for the component:

Properties

Removed	Description
ErrorDatabaseName ErrorEventName ErrorFieldName ErrorIndexName ErrorProcedureName ErrorRemoteName ErrorTableName ErrorUserName OSErrorCode SocketErrorCode	These properties are no longer necessary. ElevateDB provides logging facilities that negates the need for custom logging of the properties of an exception.

Methods

Removed	Description
None	

Events

Removed	Description
None	

Property, Method, and Event Changes

The following are the changes to the properties, methods, and events for the component:

Properties

Changed	Description
ErrorMessage	This property has been renamed to the ErrorMsg property.

Methods

Changed	Description
None	

Events

Changed	Description
None	

New Properties, Methods, and Events

The following are the new properties, methods, and events added in the new ElevateDB component:

Properties

New	Description
None	

Methods

New	Description
None	

Events

New	Description
None	

3.16 SQL Changes

The following is the list of the areas that describe the DBISAM SQL implementation. Click on each area to find out the changes to the SQL implementation.

Naming Conventions
Types
Operators
Functions
Statements

3.17 Naming Conventions

Removed Features

The following are the features that have been removed:

Removed	Description
Brackets []	The use of brackets [] for identifiers is no longer supported. Use double-quotes "" instead to specify an identifier in an SQL statement.

Feature Changes

The following are the changes to the features:

Changed	Description
Path Names	Path names are no longer supported for databases in ElevateDB. Use the database name with a period separator in order to specify a table from a specific database. Please see the Identifiers topic for more information.

New Features

The following are the new features:

New	Description
Line Feeds in String Constants	ElevateDB allows for carriage returns (character 13) and line feeds (character 10) in string constants.

3.18 Types

Removed Types

The following are the types that have been removed:

Removed	Description
AUTOINC	This type is no longer supported. Use the INTEGER type instead to store integer values, and use the GENERATED clause in a column definition to dictate that a column should be generated as an IDENTITY column. Please see the CREATE TABLE topic for more information.
MONEY	This type is no longer supported. Use the FLOAT type instead to store double-precision floating-point values. Please see the Approximate Numeric Types topic for more information.
GRAPHIC	This type is no longer supported. Use the BLOB type instead to store graphics or any other large binary objects. Please see the Binary Types topic for more information.
WORD	This type is no longer supported. Use the INTEGER type instead to store word values. Please see the Exact Numeric Types topic for more information.

Type Changes

The following are the changes to the types:

Changed	Description
CHAR	The CHAR (or CHARACTER) type now uses a fixed-length representation according to the SQL standard. Any strings that are shorter than the defined length of the column are padded with blanks.
VARCHAR	The alternate CHARACTER VARYING syntax is now acceptable. Also, VARCHAR columns no longer right-trim any spaces from strings that are stored in them. The string values are stored as-is.
BYTES or BINARY VARBYTES or VARBINARY	These types have been renamed to BYTE and VARBYTE (or BYTE VARYING), respectively.
LONGVARBINARY	This type has been renamed to BINARY LARGE OBJECT. The shorthand BLOB type notation is still retained also.
MEMO LONGVARCHAR	These types have been renamed to CLOB and CHARACTER LARGE OBJECT, respectively.
BIT	This shorthand notation for the BOOLEAN type is no longer permitted.
LARGEINT	This type has been renamed to BIGINT.
FLOAT	The alternate DOUBLE PRECISION syntax is now acceptable.

DATE TIME TIMESTAMP	Date, time, and timestamp literals must now be preceded with the DATE, TIME, and TIMESTAMP keywords, respectively.

New Types

The following are the new types:

New	Description
INTERVAL	ElevateDB now supports all day-time and year-month interval types. Please see the Interval Types topic for more information.

3.19 Operators

Removed Operators

The following are the operators that have been removed:

Removed	Description
None	

Operator Changes

The following are the changes to the operators:

Changed	Description
NULL Values	NULL constants can no longer be compared using the =, <>, >=, <=, >, <, BETWEEN, or IN operators. You must use the IS NULL and IS NOT NULL operators instead. Furthermore, none of the operators will result in a TRUE value if either side of the operator contains a NULL value. Please see the NULLs topic for more information.
Case-Insensitive Comparisons	DBISAM supported using the UPPER() or LOWER() function around a column reference and a string constant involved in a binary operator in order to force a case-insensitive comparison, and to allow the query optimizer to use a case-insensitive index to optimize the operation. This is no longer necessary in ElevateDB. Instead, you can simply use the COLLATE clause after the column reference to force the column to use a case-insensitive collation. Please see the Internationalization and Optimizer topics for more information.
Date, Time, and Timestamp Values	Subracting date, time, and timestamp values now results in an interval type, depending upon the type of the values being subtracted. Please see the Interval Types topic for more information.

New Operators

The following are the new operators:

New	Description
CONTAINS DOES NOT CONTAIN	These operators are used to implement a text search using a text index. If no text index exists on the column being searched, then these operators will always result in a FALSE value.

3.20 Functions

Removed Functions

The following are the functions that have been removed:

Removed	Description
MOD	This function is no longer necessary. You may use the MOD operator instead with ElevateDB.
LASTAUTOINC IDENT_CURRENT	These functions are no longer necessary. ElevateDB procedures and functions can retrieve the assigned IDENTITY value for a column using the FETCH statement on a cursor.
TEXTOCCURS	This function is no longer supported.
YEARSFROMMSECS DAYSFROMMSECS HOURSFROMMSECS MINSFROMMSECS SECSFROMMSECS MSECSFROMMSECS	These functions are no longer necessary. ElevateDB supports the standard SQL date and time interval types. Please see the Interval Types topic for more information.

Function Changes

The following are the changes to the functions:

Changed	Description
SUBSTRING	The alternate SUBSTR syntax is now acceptable.
TEXTSEARCH	This function has been changed to the CONTAINS and DOES NOT CONTAIN operators.

New Functions

The following are the new functions:

New	Description
None	

3.21 Statements

Removed Statements

The following are the statements that have been removed:

Removed	Description
EMPTY TABLE	This statement is no longer supported. ElevateDB requires that you use the DELETE statement to remove all rows from a table.
VERIFY TABLE	This statement is no longer supported. ElevateDB currently only offers repair facilities by using the REPAIR TABLE statement.
UPGRADE TABLE	This statement is no longer necessary.
START TRANSACTION COMMIT ROLLBACK	These statements are now considered part of the ElevateDB SQL/PSM support and are only allowed in jobs, procedures, functions, and triggers. Outside of SQL/PSM, use the TEDBDatabase StartTransaction, Commit, and Rollback

Statement Changes

The following are the changes to the statements:

Changed	Description
SELECT	ElevateDB supports single-row query expressions as values in the list of selected columns. The INTO clause is no longer supported. ElevateDB uses the standard SQL CREATE TABLE AS clause to create a table using a query expression. The EXCLUSIVE clause is no longer necessary. With ElevateDB you can use the actual table name or the table correlation name in column references anywhere in the SELECT statement. ElevateDB supports single-row query expressions as values in the JOIN clauses. ElevateDB does not optimize join expressions in the WHERE clause, otherwise known as SQL-89 style joins. You must use the JOIN clause in order to have ElevateDB optimize the joins. ElevateDB supports correlated sub-queries in the WHERE clause. ElevateDB supports single-row query expressions as values in the WHERE clause.

	The GROUP BY, HAVING, and ORDER BY clauses in ElevateDB support any type of expression, and may refer to columns that aren't in the SELECT list. The GROUP BY and ORDER BY clauses no longer support ordinal values as a way to specify a SELECT column position in the list of SELECT column expressions. You must specify the actual column reference or expression. The NOCASE clause is no longer necessary in the ORDER BY clause. ElevateDB uses the COLLATE clause to specify the collation for an ORDER BY expression. Please see the Internationalization topic for more information. The TOP clause is no longer supported. ElevateDB will introduce standard WINDOW clause support for selecting ranges of rows in a later release. The LOCALE clause is no longer necessary. ElevateDB supports column-level collations. Please see the Internationalization topic for more information. The ENCRYPTED WITH clause is no longer supported.
INSERT	The EXCLUSIVE clause is no longer necessary. The COMMIT clause is no longer supported. ElevateDB internally determines the optimal commit interval for lengthy INSERT statements.
UPDATE	The EXCLUSIVE clause is no longer necessary. The FROM clause is no longer supported. ElevateDB can use correlated sub-queries in the UPDATE values and/or WHERE clause. The COMMIT clause is no longer supported. ElevateDB internally determines the optimal commit interval for lengthy UPDATE statements. The NOJOINOPTIMIZE clause is no longer supported. The JOINOPTIMIZECOSTS clause is no longer supported.
DELETE	The EXCLUSIVE clause is no longer necessary. The FROM clause is no longer supported. ElevateDB can use correlated sub-queries in the WHERE clause. The COMMIT clause is no longer supported. ElevateDB internally determines the optimal commit interval for lengthy DELETE statements. The NOJOINOPTIMIZE clause is no longer supported. The JOINOPTIMIZECOSTS clause is no longer supported.

CREATE TABLE	The IF NOT EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information. The column definition NULLABLE clause is no longer supported. To make a column nullable in ElevateDB, don't include the NOT NULL clause. The column definition DEFAULT clause accepts any basic expression in ElevateDB. A column definition may now include a GENERATED clause to specify that the column is a generated column. Generated columns can be generated as sequence numbers or expressions. The column definition MIN and MAX clauses are no longer necessary. ElevateDB supports column constraints via the CHECK clause. ElevateDB allows for specifying primary key, unique key, and foreign key constraints in a column definition. The CHARCASE clause is no longer supported. The COMPRESS clause has been renamed to COMPRESSION and moved so that it is next to the data type definition. The NOCASE clause is no longer necessary in a primary key, unique key, or foreign key (new) constraint definition. ElevateDB uses the collation defined for the column in the column definition for determining the collation of these types of constraints. Please see the Internationalization topic for more information. The DESC and ASC clauses are no longer supported in a primary key, unique key, or foreign key (new) constraint definition. Use the CREATE INDEX statement in ElevateDB to create an index with custom column sorting. The COMPRESS clause is no longer supported in a primary key, unique key, or foreign key (new) constraint definition. ElevateDB performs automatic index compression as necessary. The TEXT INDEX, STOP WORDS, SPACE CHARS, and INCLUDE CHARS clauses are no longer necessary. Use the CREATE TEXT INDEX statement in ElevateDB to create a new text index. The LOCALE clause is no longer necessary. ElevateDB supports column-level collations. Please see the Internationalization topic for more information. The WITH clause of the ENCRYPTED clause is no longer necessary. ElevateDB uses one encryption password per

	application for all encryption, and it is represented by the EncryptionPassword property. Also, the ENCRYPTED clause now resides after the VERSION clause (see next item). The USER MAJOR VERSION and USER MINOR VERSION clauses have been combined into one VERSION clause that accepts a NUMERIC value with a scale of 2. Also, the VERSION clause now resides after the DESCRIPTION clause. The LAST AUTOINC clause is no longer necessary. The seed and increment values for IDENTITY columns can be specified in the column definitions.
CREATE INDEX	The IF NOT EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if an index exists. Please see the System Information topic for more information. The UNIQUE clause is no longer supported. ElevateDB requires that unique keys constraints be defined using a constraint definition in a CREATE TABLE or ALTER TABLE statement. The NOCASE clause is no longer necessary in an index definition. ElevateDB uses the collation defined for the column in the column definition for determining the default collation for the indexed columns, and also allows for the COLLATE clause to be used in the index definition in order to override the default column collation. Please see the Internationalization topic for more information. The COMPRESS clause is no longer supported in an index definition. ElevateDB performs automatic index compression as necessary.
ALTER TABLE	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information. The IF EXISTS and IF NOT EXISTS clauses are no longer supported for column definitions. ElevateDB uses catalog queries to determine if a table column exists. Please see the System Information topic for more information. The REDEFINE clause is no longer supported for column definitions. In order to redefine a column using the same column name, use the ALTER AS clause (see next). In order to rename a column, use the RENAME clause. The ALTER clause is new for column definitions. This clause allows you to alter the DEFAULT expression, drop the default expression, change the DESCRIPTION of the column, move the column to a new position in the table using the MOVE TO clause, or alter the entire column definition using the AS clause. The column definition AT clause has been moved to the end of the column definition.

The column definition NULLABLE clause is no longer supported. To make a column nullable in ElevateDB, don't include the NOT NULL clause.

The column definition DEFAULT clause accepts any basic expression in ElevateDB.

A column definition may now include a GENERATED clause to specify that the column is a generated column. Generated columns can be generated as sequence numbers or expressions.

The column definition MIN and MAX clauses are no longer necessary. ElevateDB supports column constraints via the CHECK clause.

ElevateDB allows for specifying primary key, unique key, and foreign key constraints in a column definition.

The CHARCASE clause is no longer supported.

The COMPRESS clause has been renamed to COMPRESSION and moved so that it is next to the data type definition.

The REDEFINE clause is no longer supported for constraint definitions. Use the RENAME clause to rename a constraint.

The NOCASE clause is no longer necessary in a primary key, unique key, or foreign key (new) constraint definition. ElevateDB uses the collation defined for the column in the column definition for determining the collation of these types of constraints. Please see the Internationalization topic for more information.

The DESC and ASC clauses are no longer supported in a primary key, unique key, or foreign key (new) constraint definition. Use the CREATE INDEX statement in ElevateDB to create an index with custom column sorting.

The COMPRESS clause is no longer supported in a primary key, unique key, or foreign key (new) constraint definition. ElevateDB performs automatic index compression as necessary.

The TEXT INDEX, STOP WORDS, SPACE CHARS, and INCLUDE CHARS clauses are no longer necessary. Use the CREATE TEXT INDEX statement in ElevateDB to create a new text index.

The LOCALE clause is no longer necessary. ElevateDB supports column-level collations. Please see the Internationalization topic for more information.

The WITH clause of the ENCRYPTED clause is no longer necessary. ElevateDB uses one encryption password per

	application for all encryption, and it is represented by the EncryptionPassword property. Also, the ENCRYPTED clause now resides after the VERSION clause (see next item). The USER MAJOR VERSION and USER MINOR VERSION clauses have been combined into one VERSION clause that accepts a NUMERIC value with a scale of 2. Also, the VERSION clause now resides after the DESCRIPTION clause. The LAST AUTOINC clause is no longer necessary. The seed and increment values for IDENTITY columns can be specified in the column definitions. The NOBACKUP clause has been renamed to the NO BACKUP FILES clause.
DROP TABLE	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information.
DROP INDEX	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if an index exists. Please see the System Information topic for more information. The PRIMARY clause is no longer supported. ElevateDB does not allow a primary key to be dropped using the DROP INDEX statement. Instead, you must use the ALTER TABLE statement to add or drop constraints for a table.
IMPORT TABLE	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information. The COLUMNS clause has been renamed and the COLUMN portion has been dropped, retaining only the columns list in parentheses. Also, the clause has been moved so that it is right after the import file name. The DELIMITER clause has been renamed to DELIMITER CHAR. The QUOTE CHAR clause has been added to allow you to specify the quote character to be used for string values. The DATE clause has been renamed to the DATE FORMAT clause. The TIME clause has been renamed to the TIME FORMAT clause. The DECIMAL clause has been renamed to the DECIMAL CHAR clause. The BOOLEAN clause has been added to allow you to specify the literals used for True and False, respectively. The WITH HEADERS clause has been renamed to the USE

	HEADERS clause and has been moved to right after the BOOLEAN clause. The MAX ROWS clause has been added to allow you to specify the maximum number of rows that should be imported from the file.
EXPORT TABLE	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information. The COLUMNS clause has been renamed and the COLUMN portion has been dropped, retaining only the columns list in parentheses. Also, the clause has been moved so that it is right after the export file name. The DELIMITER clause has been renamed to DELIMITER CHAR. The QUOTE CHAR clause has been added to allow you to specify the quote character to be used for string values. The DATE clause has been renamed to the DATE FORMAT clause. The TIME clause has been renamed to the TIME FORMAT clause. The DECIMAL clause has been renamed to the DECIMAL CHAR clause. The BOOLEAN clause has been added to allow you to specify the literals used for True and False, respectively. The WITH HEADERS clause has been renamed to the INCLUDE HEADERS clause and has been moved to right after the BOOLEAN clause. The MAX ROWS clause has been added to allow you to specify the maximum number of rows that should be exported to the file.
OPTIMIZE TABLE	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information. The ON clause has been renamed to the USING INDEX clause. The NOBACKUP clause has been renamed to the NO BACKUP FILES clause.
REPAIR TABLE	The IF EXISTS clause is no longer supported. ElevateDB uses catalog queries to determine if a table exists. Please see the System Information topic for more information. The FORCEINDEXREBUILD clause is no longer supported.

New Statements

The following are the new statements:

New	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.
CREATE STORE	Creates a new file store.
ALTER STORE	Alters an existing file store.
DROP STORE	Drops an existing file store.
RENAME STORE	Renames an existing file store.
CREATE USER	Creates a new user.
ALTER USER	Alters an existing user.
DROP USER	Drops an existing user.
RENAME USER	Renames an existing user.
CREATE ROLE	Creates a new role.
ALTER ROLE	Alters an existing role.
DROP ROLE	Drops an existing role.
RENAME ROLE	Renames an existing role.
GRANT PRIVILEGES	Grants privileges to an existing user or role on a specified object.
REVOKE PRIVILEGES	Revokes privileges for an existing user or role from an existing object.
GRANT ROLES	Grants roles to an existing user.
REVOKE ROLES	Revokes roles from an existing user.
CREATE JOB	Creates a new job.
ALTER JOB	Alters an existing job.
DROP JOB	Drops an existing job.
RENAME JOB	Renames an existing job.
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.
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.
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.
CREATE MIGRATOR	Creates a new database migrator.
ALTER MIGRATOR	Alters an existing database migrator.
DROP MIGRATOR	Drops an existing database migrator.
RENAME MIGRATOR	Renames an existing database migrator.
CREATE TRIGGER	Creates a new trigger on an existing table.
ALTER TRIGGER	Alters an existing trigger.
DROP TRIGGER	Drops an existing trigger from a table.
RENAME TRIGGER	Renames an existing trigger on a table.
CREATE TEXT INDEX	Creates a new text index on an existing table.
ALTER INDEX	Alters an existing index.
CREATE VIEW	Creates a new view.
ALTER VIEW	Alters an existing view.
DROP VIEW	Drops an existing view.
RENAME VIEW	Renames an existing view.
CREATE FUNCTION	Creates a new function.
ALTER FUNCTION	Alters an existing function.
DROP FUNCTION	Drops an existing function.
RENAME FUNCTION	Renames an existing function.
CREATE PROCEDURE	Creates a new procedure.
ALTER PROCEDURE	Alters an existing procedure.
DROP PROCEDURE	Drops an existing procedure.
RENAME PROCEDURE	Renames an existing procedure.
SET BACKUPS STORE	Sets the current backups store for ElevateDB.
BACKUP DATABASE	Backs up an existing database.
RESTORE DATABASE	Restores a database from a backup.
PUBLISH DATABASE	Publishes an existing database.
UNPUBLISH DATABASE	Unpublishes a database.
SET UPDATES STORE	Sets the current updates store for ElevateDB.
SAVE UPDATES	Saves all logged updates to published tables in an existing database.

LOAD UPDATES	Loads logged updates from an update file into an existing database.
COPY FILE	Copies a file in a store to a new file name, and optionally, store.
RENAME FILE	Renames a file in a store to a new file name.
DELETE FILE	Deletes a file in a store.
SET FILES STORE	Sets the current files store for ElevateDB.
DISCONNECT SERVER SESSION	Disconnects a server session on an ElevateDB Server.
REMOVE SERVER SESSION	Removes a server session from an ElevateDB Server.

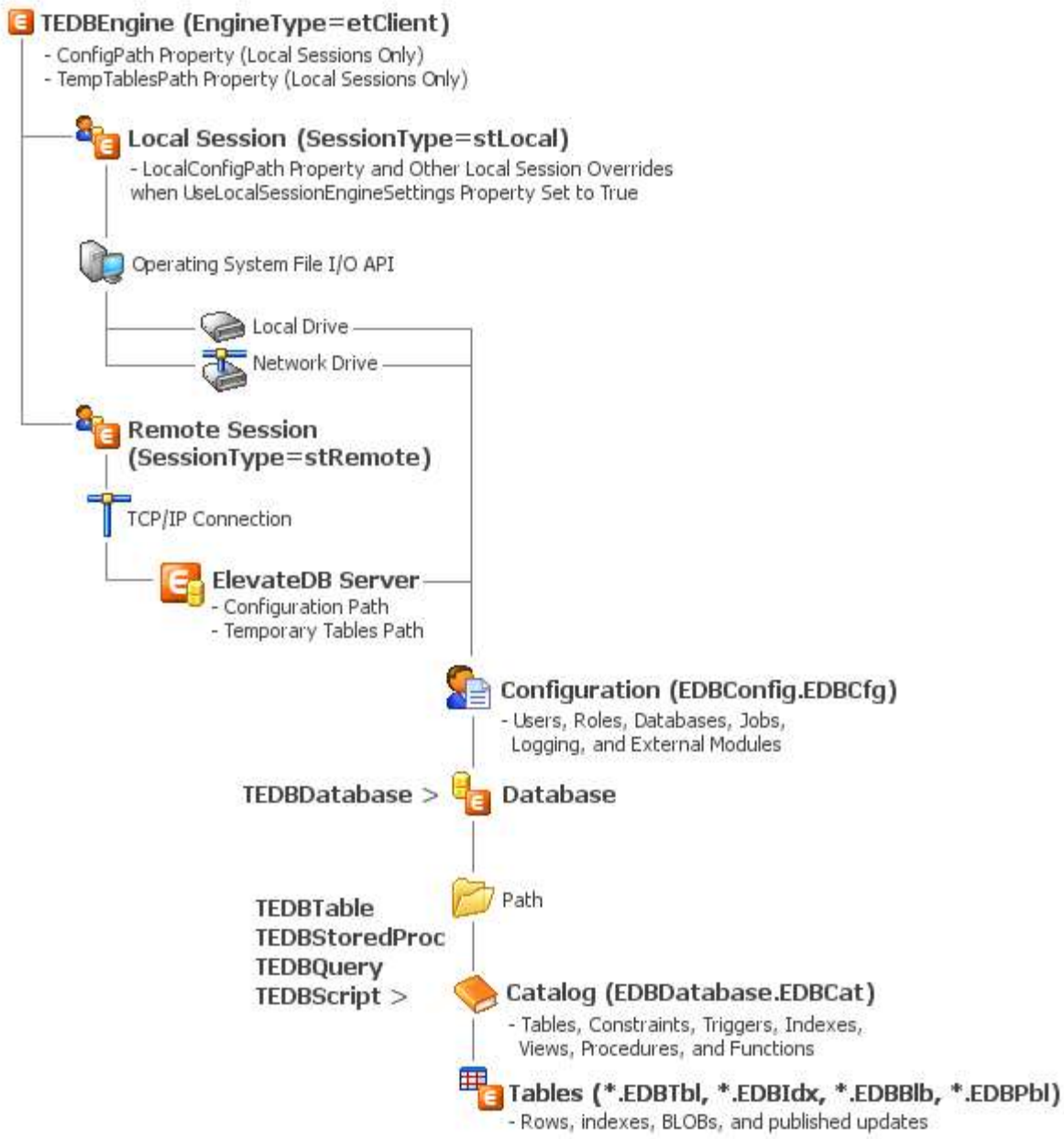
This page intentionally left blank

Chapter 4
Getting Started

4.1 Architecture

ElevateDB is a database engine that can be compiled directly into your Embarcadero Delphi, Embarcadero C++, or Lazarus application, be it a program or library, or it can be distributed as a runtime package (equivalent to a library) as part of your application. ElevateDB is available for Delphi 5 and later, as well as Lazarus 0.924 and later. ElevateDB was written in Delphi's Object Pascal language and can be used with the Delphi VCL (Win32, Win64, MacOS32, MacOS64, and Linux64) or Lazarus LCL (Win32, Win64, and Linux64) runtime libraries.

The following image illustrates the general architecture of ElevateDB:



The various components that make up this architecture are detailed next.

Component Architecture

ElevateDB uses a component architecture that includes the following components:

 TEDBEngine

The TEDBEngine component encapsulates the ElevateDB engine itself. A TEDBEngine component is created automatically when the application is started and can be referenced via the global Engine function in the edbcomps unit. You can also drop a TEDBEngine component on a form or data-module to change its properties at design-time. However, only one instance of the TEDBEngine component can exist in a given application, and both the global Engine function and any TEDBEngine component on a form or data module point to the same instance of the component (singleton model). The TEDBEngine component can be configured so that it acts like a local or client engine (etClient) or a server engine (etServer) via the EngineType property. The engine can be started by setting the Active property to True.

> **Note**
> Once the engine has been started, most of the properties that configure the engine cannot be modified.

By default, ElevateDB allows you to configure all local sessions via the TEDBEngine component and its ConfigMemory, ConfigPath, ConfigName, and TempTablesPath properties, as well as several other properties that can customize the local session access for a particular application. However, you can also set the UseLocalSessionEngineSettings property to True in order to tell ElevateDB to use the Local* versions of these same properties from the TEDBSession component to override the engine configuration. This is useful for applications that require access to multiple configuration files for multiple local sessions, such as the ElevateDB Manager that is provided with ElevateDB. Please see the Configuring and Starting the Engine topic for more information on the various engine properties that can be modified when configuring local sessions via the TEDBEngine component.

 TEDBSession

The TEDBSession component encapsulates a session in ElevateDB. A default TEDBSession component is created automatically when the application is started and can be referenced via the global Session function in the edbcomps unit. The TEDBSession component can be configured so that it acts like a local (stLocal) or a remote session (stRemote) via the SessionType property. A local session is single-tier in nature, meaning that all TEDBDatabase components connected to the session reference databases in a local or network file system and all TEDBTable, TEDBQuery, or TEDBStoredProc components access the physical tables directly from these directories using operating system calls. A remote session is two-tier in nature, meaning that all access is done through the remote session to an ElevateDB Server using a messaging protocol over a TCP/IP connection. A remote session is configured using the following properties:

RemoteHost or RemoteAddress
RemotePort or RemoteService

In a remote session, all TEDBDatabase components reference databases that are defined on the ElevateDB Server and all TEDBTable or TEDBQuery components access the physical tables through the messaging protocol rather than directly through the operating system.

> **Note**
> You cannot connect remote sessions in an application whose TEDBEngine component is configured as a server via the EngineType property.

As mentioned above, a local session is usually configured via the TEDBEngine component. However, if the UseLocalSessionEngineSettings property is set to True, then the Local* versions of the TEDBEngine configuration properties that are found in the TEDBSession component will be used to override the TEDBEngine configuration settings.

A session can be connected by setting the Connected property to True or by calling the Open method. The TEDBSession component contains a SessionName property that is used to give a session a name and a SessionDescription property that is used to assign a description to the session. All session components must have a name before they can be connected. The default TEDBSession component is called "Default". The TEDBDatabase, TEDBTable, TEDBQuery, and TEDBStoredProc components also have a SessionName property and these properties are used to specify which session these components belong to. Setting their SessionName property to "Default" or blank ("") indicates that they will use the default TEDBSession component. Please see the Connecting Sessions topic for more information.

TEDBDatabase

The TEDBDatabase component encapsulates a database in ElevateDB, and is used as an container for all access to a specific database. A database can be opened by setting the Connected property to True or by calling the Open method. A TEDBDatabase component contains a DatabaseName property that is used to give a database a name within the application. All database components must have a name before they can be opened. The TEDBTable, TEDBQuery, and TEDBStoredProc components also have a DatabaseName property and these properties are used to specify which database these components belong to. Please see the Opening Tables and Views topic for more information.

The TEDBDatabase Database property specifies the name of a database that you would like to connect to.

The TEDBDatabase component is used for transaction processing via the StartTransaction, Commit, and Rollback methods. Please see the Transactions topic for more information.

You can execute dynamic SQL on a specific database by using the Execute method. Please see the Executing Queries topic for more information.

TEDBTable

The TEDBTable component encapsulates a cursor on a table or view in ElevateDB. It is used to search,insert, update, or delete rows within the table or view specified by the TableName property. A table or view cursor can be opened by setting the Active property to True or by calling the Open method. The DatabaseName property specifies the name of the database component that references the database where the table or view resides. Please see the Opening Tables and Views topic for more information.

Because the TEDBTable component represents a table or view cursor, you can have multiple TEDBTable components referencing the same table or view. Each TEDBTable component maintains its own active order, filter and range conditions, current row position, row count statistics, etc.

Note
The TEDBTable component descends from the TEDBDBDataSet component, which descends from the TEDBDataSet component, which descends from the common TDataSet component that is the basis for all data access in VCL or CLX applications. None of these lower-level components should be used directly and are only for internal structuring purposes in the class hierarchy.

TEDBQuery

The TEDBQuery component encapsulates a single SQL statement in ElevateDB. This SQL statement may or may not return a result set, but if it does return a result set, then the TEDBQuery component will act as a cursor on the result set in the same way that the TEDBTable component acts as a cursor on a table or view. The SQL statement to execute is specified in the SQL property, and the statement can be executed by setting the Active property to True, by calling the Open method (for SQL statements that definitely return a result set), or by calling the ExecSQL method (for SQL statements that may or may not return a result set). The DatabaseName property specifies the name of the database component that references the database to be used when executing the SQL statement. Please see the Executing Queries topic for more information.

Note
The TEDBQuery component descends from the TEDBDBDataSet component, which descends from the TEDBDataSet component, which descends from the common TDataSet component that is the basis for all data access in Delphi, C++Builder, Borland Developer Studio, CodeGear RAD Studio, and Lazarus. None of these lower-level components should be used directly and are only for internal structuring purposes in the class hierarchy.

TEDBScript

The TEDBScript component encapsulates a single SQL script in ElevateDB. This script may or may not return a result set, but if it does return a result set, then the TEDBScript component will act as a cursor on the result set in the same way that the TEDBTable component acts as a cursor on a table or view. The script to execute is specified in the SQL property, and the script can be executed by setting the Active property to True, by calling the Open method (for scripts that definitely return a result set), or by calling the ExecScript method (for scripts that may or may not return a result set). The DatabaseName property specifies the name of the database component that references the database to be used when executing the script. Please see the Executing Scripts topic for more information.

Note
The TEDBScript component descends from the TEDBDBDataSet component, which descends from the TEDBDataSet component, which descends from the common TDataSet component that is the basis for all data access in Delphi, C++Builder, Borland Developer Studio, CodeGear RAD Studio, and Lazarus. None of these lower-level components should be used directly and are only for internal structuring purposes in the class hierarchy.

TEDBStoredProc

The TEDBStoredProc component encapsulates a single stored procedure in ElevateDB. This stored procedure may or may not return a result set, but if it does return a result set, then the TEDBStoredProc component will act as a cursor on the result set in the same way that the TEDBTable component acts as a

cursor on a table or view. The stored procedure to execute is specified in the StoredProcName property, and the stored procedure can be executed by setting the Active property to True, by calling the Open method (for stored procedures that definitely return a result set), or by calling the ExecProc method (for stored procedures that may or may not return a result set). The DatabaseName property specifies the name of the database component that references the database to be used when executing the stored procedure. Please see the Executing Stored Procedures topic for more information.

Note
The TEDBStoredProc component descends from the TEDBDBDataSet component, which descends from the TEDBDataSet component, which descends from the common TDataSet component that is the basis for all data access in Delphi, C++Builder, Borland Developer Studio, CodeGear RAD Studio, and Lazarus. None of these lower-level components should be used directly and are only for internal structuring purposes in the class hierarchy.

Opening a TEDBTable, TEDBQuery, TEDBScript, or TEDBStoredProc component will automatically cause its corresponding TEDBDatabase component to open, which will also automatically cause its corresponding TEDBSession component to connect, which will finally cause the TEDBEngine to start. This design ensures that the necessary connections for a session, database, etc. are completed before the opening of the table, query, or stored procedure is attempted.

4.2 Exception Handling and Errors

One of the first items to address in any application, and especially a database application, is how to anticipate and gracefully handle exceptions. This is true as well with ElevateDB.

ElevateDB Exception Types

ElevateDB uses the EEDBError object as its exception object for all errors. This object descends from the EDatabaseError exception object defined in the common DB unit, which itself descends from the common Exception object. This hierarchy is important since it allows you to isolate the type of error that is occurring according to the type of exception object that has been raised, as you will see below when we demonstrate some exception handling.

> **Note**
> ElevateDB also raises certain component-level exceptions as an EDatabaseError to maintain consistency with the way the common DB unit and TDataSet component behaves. These mainly pertain to design-time property modifications, but a few can be raised at runtime also.

The EEDBError object contains several important properties that can be accessed to discover specific information on the nature of the exception. The ErrorCode property is always populated with a value which indicates the error code for the current exception. Other properties may or may not be populated according to the error code being raised, and a list of all of the error codes raised by the ElevateDB engine along with this information can be found in Appendix A - Error Codes and Messages.

Exception Handling

The most basic form of exception handling is to use the try..except block (Delphi and Lazarus) or try..catch (C++) to locally trap for specific error conditions. The error code that is returned when an open fails due to an exclusive lock problem is 300, which is defined as EDB_ERROR_LOCK in the edberror unit. The following example shows how to trap for such an exception on open and display an appropriate error message to the user:

```
begin
   with MyEDBTable do
      begin
      DatabaseName:='Tutorial';
      TableName:='customer';
      Exclusive:=True;
      ReadOnly:=False;
      try
         Open;
      except
         on E: Exception do
            begin
            if (E is EDatabaseError) and (E is EEDBError) then
               begin
               if (EEDBError(E).ErrorCode=EDB_ERROR_LOCK) then
                  ShowMessage('Cannot open table '+TableName+
                    ', another user has the table open already')
               else
                  ShowMessage('Unknown or unexpected '+
```

```
                    'database engine error # '+
                    IntToStr(EEDBError(E).ErrorCode));
                end
             else
                ShowMessage('Unknown or unexpected '+
                  'error has occurred');
             end;
       end;
       end;
end;
```

Exception Events

Besides trapping exceptions with a try..except or try..catch block, you may also use a global TApplication.OnException event handler to trap database exceptions. However, doing so will eliminate the ability to locally recover from the exception and possibly retry the operation or take some other course of action. There are several events in ElevateDB components that allow you to code event handlers that remove the necessity of coding try..except or try..catch blocks while still providing for local recovery. These events are as follows:

Event	Description
OnEditError	This event is triggered when an error occurs during a call to the TEDBTable, TEDBQuery , or TEDBStoredProc Edit method . The usual cause of an error is a row lock failure if the current session is using the pessimistic row locking protocol (the default). Please see the Inserting, Updating, and Deleting Rows topic for more information on using this event, and the Locking and Concurrency topic for more information on the ElevateDB row locking protocols.
OnDeleteError	This event is triggered when an error occurs during a call to the TEDBTable, TEDBQuery , or TEDBStoredProc Delete method. The usual cause of an error is a row lock failure (a row lock is always obtained before a delete regardless of the locking protocol in use for the current session). Please see the Inserting, Updating, and Deleting Rows topic for more information on using this event, and the Locking and Concurrency topic for more information on the ElevateDB row locking protocols.
OnPostError	This event is triggered when an error occurs during a call to the TEDBTable, TEDBQuery , or TEDBStoredProc Post method. The usual cause of an error is a constraint violation, however it can also be triggered by a row lock failure if the locking protocol for the current session is set to optimistic. Please see the Inserting, Updating, and Deleting Rows topic for more information on using this event, and the Locking and Concurrency topic for more information on the ElevateDB row locking protocols.

4.3 Multi-Threaded Applications

ElevateDB is internally structured to be thread-safe and usable within a multi-threaded application provided that you follow the rules that are outlined below.

Unique Sessions

ElevateDB requires that you use a unique TEDBSession component for every thread that must perform any database access at all. Each of these TEDBSession components must also be assigned a SessionName property value that is unique among all TEDBSession components in the application. Doing this allows ElevateDB to treat each thread as a separate and distinct session and will isolate transactions and other internal structures accordingly. You may use the AutoSessionName property of the TEDBSession component to allow ElevateDB to automatically name each session so that is unique or you may use code similar to the following:

```
var
   LastSessionValue: Integer;
   SessionNameSection: TRTLCriticalSection;

{ Assume that the following code is being executed
  within a thread }

function UpdateAccounts: Boolean;
var
   LocalSession: TEDBSession;
   LocalDatabase: TEDBDatabase;
   LocalQuery:   TEDBQuery;
begin
   Result:=False;
   LocalSession:=GetNewSession;
   try
      LocalDatabase:=TEDBDatabase.Create(nil);
      try
         with LocalDatabase do
            begin
            { Be sure to assign the same session name
              as the TEDBSession component }
            SessionName:=LocalSession.SessionName;
            DatabaseName:='AccountsDB';
            Database:='Accounting';
            Connected:=True;
            end;
         LocalQuery:=TEDBQuery.Create(nil);
         try
            with LocalQuery do
               begin
               { Be sure to assign the same session and
                 database name as the TEDBDatabase
                 component }
               SessionName:=LocalSession.SessionName;
               DatabaseName:=LocalDatabase.DatabaseName;
               SQL.Clear;
               SQL.Add('UPDATE accounts SET PastDue=True');
               SQL.Add('WHERE DueDate < CURRENT_DATE'));
               Prepare;
```

```
               try
                  { Start the transaction and execute the query }
                  LocalDatabase.StartTransaction;
                  try
                     ExecSQL;
                     LocalDatabase.Commit;
                     Result:=True;
                  except
                     LocalDatabase.Rollback;
                  end;
               finally
                  UnPrepare;
               end;
               end;
         finally
            LocalQuery.Free;
         end;
      finally
         LocalDatabase.Free;
      end;
   finally
      LocalSession.Free;
   end;
end;

function GetNewSession: TEDBSession;
begin
   EnterCriticalSection(SessionNameSection);
   try
      LastSessionValue:=LastSessionValue+1;
      Result:=TEDBSession.Create(nil);
      with Result do
         SessionName:='AccountSession'+IntToStr(LastSessionValue);
   finally
      LeaveCriticalSection(SessionNameSection);
   end;
end;

{ initialization in application }
   LastSessionValue:=0;
   InitializeCriticalSection(SessionNameSection);
{ finalization in application }
   DeleteCriticalSection(SessionNameSection);
```

The AutoSessionName property is, by default, set to False so you must specifically turn it on if you want this functionality. You may also use the thread ID of the currently thread to uniquely name a session since the thread ID is guaranteed to be unique within the context of a process.

Unique Databases

Another requirement is that all TEDBDatabase components must also be unique and have values assigned to their SessionName properties that refer to the unique SessionName property of the TEDBSession component defined in the manner discussed above.

Unique Tables, Queries, and Stored Procedures

The final requirement is that all TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components

must also be unique and have values assigned to their SessionName properties that refer to the unique SessionName property of the TEDBSession component defined in the manner discussed above. Also, if a TEDBTable or TEDBQuery component refers to a TEDBDatabase component's DatabaseName property via its own DatabaseName property, then the TEDBDatabase referred to must be defined in the manner discussed above.

ISAPI Applications

ISAPI applications created using the WebBroker components or a similar technology are implicitly multi-threaded. Because of this, you should ensure that your ISAPI application is thread-safe according to these rules for multi-threading when using ElevateDB. Also, if you have simply dropped a TEDBSession component on the WebModule of a WebBroker ISAPI application, you must set its AutoSessionName property to True before dropping any other ElevateDB components on the form so that ElevateDB will automatically give the TEDBSession component a unique SessionName property and propogate this name to all of the other ElevateDB components.

Further Considerations

There are some other things to keep in mind when writing a multi-threaded database application with ElevateDB, especially if the activity will be heavy and there will be many threads actively running. Be prepared to handle any errors in a manner that allows the thread to terminate gracefully and properly free any TEDBSsion, TEDBDatabase, TEDBTable, TEDBQuery, and TEDBStoredProc components that it has created. Otherwise you may run into a situation where memory is being consumed at an alarming rate. Finally, writing multi-threaded applications, especially with database access, is not a task for the beginning developer so please be sure that you are well-versed in using threads and how they work before jumping into writing a multi-threaded application with ElevateDB.

4.4 Recompiling the ElevateDB Source Code

In some cases you may want to change the ElevateDB source code and recompile it to incorporate these changes into your application. However, you must first have purchased the ElevateDB client and/or server source code in order to make changes to the source code.

Setting Search Paths

The first thing that you must do is make sure that any search paths, either global to ElevateDB such as the Library Search Path or local to your project, are pointing to the directory or path where the ElevateDB source code was installed. By default this directory or path is:

```
\<base directory>\<product>\<compiler> <n>\code\source
```

The <product> component of the path can be one of the following values:

Value	Description
ElevateDB <type> STD-SRC	This indicates the standard version of ElevateDB with source code
ElevateDB <type> CS-SRC	This indicates the client-server version of ElevateDB with source code

The <type> component of the product name will be either VCL or DAC.

The <compiler> <n> component of the path indicates the development environment in use and the version number of the development environment. For example, for Delphi 6 this component would look like this:

```
Delphi 6
```

Setting Compiler Switches

The second thing that must be done is to make sure that the compiler switches that you are using are set properly for ElevateDB. The build system used to compile ElevateDB here at Elevate Software uses the dcc32.exe and dcc64.exe command-line compilers provided with Delphi, C++Builder, Borland Developer Studio, CodeGear RAD Studio, and Embarcadero RAD studio to compile ElevateDB. The following switches are set during compilation and any other switches are assumed to be at their default state for the compiler:

```
$D-   Debug information off
$L-   Local symbols off
```

Note
These same switches are used to compile all ElevateDB utilities and the ElevateDB Server project also.

A Word of Caution

Making changes to the ElevateDB source code is not an easy task. A mistake in such changes could result in the loss of critical data and Elevate Software cannot be held responsible for any losses incurred from such changes. Occasionally our support staff may send a fix to a customer that owns the source code in order to facilitate a quicker turnaround on a bug report, but it is the responsibility of the customer to weigh the risks of implementing such a change with the possible problems that such a change could bring about. Elevate Software tries very hard to also assist any customers that do want to make changes to the ElevateDB source code for custom purposes and will always make an attempt to guide the customer to a solution that fits their needs and is reliable in operation. In general, however, it is usually recommended that you use the generic configuration facilities provided with ElevateDB as opposed to making direct changes to the source code. Please see the Configuring and Starting the Engine topic for more information.

This page intentionally left blank

Chapter 5
Using ElevateDB

5.1 Configuring and Starting the Engine

Configuring the Engine

As already discussed in the Architecture topic, the TEDBEngine component represents the engine in ElevateDB. The following information will show how to configure the engine for use as a client engine in an application or as an server engine. The TEDBEngine EngineType property controls whether the engine is behaving as a local engine or a server engine.

> **Note**
> The TEDBEngine component must be inactive (Active=False) when modifying any of the configuration properties.

Character Set

The TEDBEngine CharacterSet property specifies which character set, ANSI or Unicode, to use for reading and writing the configuration file and all databases and tables. This property defaults to a value that matches the default string type used by the current compiler. For example, with Delphi XE the default string type is a Unicode string, so this property will default to csUnicode when used with Delphi XE. This setting can be overridden on a per-session basis by modifying the TEDBSession CharacterSet property.

Configuration Path

The TEDBEngine ConfigMemory and ConfigPath properties specify where the engine should look for the configuration file to use for the application, if running as a client engine, or the server, if running as a server engine. The configuration file is used to store the information in the Configuration database in ElevateDB. If the ConfigMemory property is set to True, then the configuration file will be "virtual" and stored in the process memory. If the ConfigMemory property is False and the path specified for the ConfigPath property does not exist, then an error will be raised when the engine is started (Active=True). If the path exists, but the configuration file does not exist in the path, then the ElevateDB engine will create the configuration file as necessary.

> **Note**
> It is very important that you do not have more than one instance of the ElevateDB engine using different configuration files (including mixing virtual and non-virtual configuration files) and accessing the same database(s). Doing so will cause locking errors. All instances of the ElevateDB engine must use the same type of configuration file (virtual or disk-based) and, if disk-based, the same configuration file if they will be accessing the same database(s).

Temporary Tables Path

The TEDBEngine TempTablesPath property controls where ElevateDB creates any temporary tables that

are required for storing query result sets. By default, the TempTablesPath property is set to the user-specific temporary tables path for the operating system.

Engine Signature

The TEDBEngine Signature property controls the engine signature for the engine. The default engine signature is "edb_signature". The engine signature in ElevateDB is used to "stamp" all configuration files, catalog files, table files, backup files, update files, and streams created by the engine so that only an engine with the same signature can open them or access them afterwards. If an engine does attempt to access an existing table, backup file, update file, or stream with a different signature than that of the table, backup file, update file, or stream, an EEDBError exception will be raised. The error that is raised when the access fails due to an invalid engine signature is 100 (EDB_ERROR_VALIDATE).

Also, if the EngineType property is set to etClient, the engine signature is used to stamp all requests sent from a remote session to an ElevateDB Server. If the ElevateDB Server is not using the same engine signature, then the requests will be treated as invalid and rejected by the ElevateDB Server. If the EngineType property is set to etServer, the engine signature is used to stamp all responses sent from the ElevateDB Server to any remote session. If the remote session is not using the same engine signature then the requests will be treated as invalid and rejected by the ElevateDB Server. In summary, both the remote sessions and the ElevateDB Server must be using the same engine signature or else communications between the two will be impossible.

Note
It is important to note that ElevateDB can always open any file that is stamped with the default signature, as well as communicate with any ElevateDB Server using the default signature, even if the engine signature has been changed to use a custom signature. Therefore, it is important that one make sure that the engine signature is changed *before* any files are created that one wants to be stamped with the custom engine signature.

Encryption Password

You can use the EncryptionPassword property to modify the encryption password used by ElevateDB for all file encryption purposes. ElevateDB uses this password for all configuration, database catalog (for encrypted catalogs), and table files (for encrypted tables) encryption. The default encryption password 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.

Licensed Sessions

You can specify that a certain maximum number of concurrent licensed sessions be allowed by modifying the TEDBEngine LicensedSessions property. The default value for this property is 4096 sessions. Setting this property to a lower figure will allow no more than the specified number of sessions to concurrently access the same configuration.

Buffered File I/O

You can specify whether to enable buffered file I/O in ElevateDB by modifying the TEDBEngine BufferedFileIO property. The default value for this property is False. If you enable buffered file I/O, you can use the BufferedFileIOSettings and BufferedFileIOFlushInterval properties to control how the buffered file I/O behaves.

The following is an example of how the buffered file I/O could be configured:

```
Engine.BufferedFileIO:=True;
with Engine.BufferedFileIOSettings do
   begin
   { Lock files don't use buffering }
   Add('"*EDBConfig.EDBLck",1,1,0,False');
   Add('"*EDBDatabase.EDBLck",1,1,0,False');
   { Configuration and catalog files:
     64KB block size
     4MB buffer size
     0-second flush age (always write any dirty buffers during flush checks)
     Always force flush to disk call in OS }
   Add('"*EDBDatabase.EDBCfg",64,4,0,True');
   Add('"*EDBDatabase.EDBCat",64,4,0,True');
   { Smaller database table files:
     64KB block size
     32MB buffer size
     120-second flush age
     Don't force flush to disk call in OS }
   Add('"*Customer.EDBTbl",64,32,120,False');
   Add('"*Customer.EDBIdx",64,32,120,False');
   Add('"*Customer.EDBBlb",64,32,120,False');
   { Larger database table files:
     64KB block size
     128MB-256MB buffer sizes
     120-second flush age
     Don't force flush to disk call in OS }
   Add('"*Orders.EDBTbl",64,128,120,False');
   Add('"*Orders.EDBIdx",64,256,120,False');
   Add('"*Orders.EDBBlb",64,256,120,False');
   end;
```

Please see the Buffering and Caching topic in the SQL manual for more information on buffered file I/O in ElevateDB.

File Names and Extensions

The following file customziations can be made for the engine:

File	Description
Configuration File	The ConfigName property determines the root name (without extension) used by the engine for the configuration file. The extension used for the configuration file is determined by the ConfigExtension property. The location of the configuration file is determined by the ConfigPath property.
Configuration Lock File	The ConfigName property determines the root name (without extension) used by the engine for the configuration lock file. The extension used for the configuration lock file is determined by the LockExtension property. The location of the configuration lock file is determined by the ConfigPath property, and the configuration lock file is hidden, by default.

Configuration Log File	The ConfigName property determines the root name (without extension) used by the engine for the configuration log file. The extension used for the configuration log file is determined by the LogExtension property. The location of the configuration log file is determined by the ConfigPath property. The maximum size of the log file can be controlled via the MaxLogFileSize property. Log entries are added to the log in a circular fashion, meaning that once the maximum log file size ia reached, ElevateDB will start re-using the oldest log entries for new log entries. The default value is 1048576 bytes. Which types of logged events are recorded in the log can be controlled by the LogCategories property. By default, all categories of events are logged (Information, Warning, or Error). **Warning** It is very important that all applications accessing the same configuration file use the same maximum log file size for the configuration log file. Using different values can result in log entries being prematurely overwritten or appearing "out-of-order" when viewing the log entries via the LogEvents Table.
Catalog File	The CatalogName property determines the root name (without extension) used by the engine for all database catalog files. The extension used for the catalog files is determined by the CatalogExtension property. The location of the catalog file is determined by the path designated for the applicable database when the database was created. Please see the Creating, Altering, or Dropping Configuration Objects topic for more information.
Catalog Lock File	The CatalogName property determines the root name (without extension) used by the engine for the database catalog lock files. The extension used for a catalog lock file is determined by the LockExtension property. The location of a catalog lock file is determined by the path designated for the applicable database when the database was created, and a catalog lock file is hidden, by default. Please see the Creating, Altering, or Dropping Configuration Objects topic for more information.
Backup File	The BackupExtension property determines the extension used for all backup files created by ElevateDB. Please see the Backing Up and Restoring Databases topic for more information.
Update File	The UpdateExtension property determines the extension used for all update files created by ElevateDB. Please see the Saving Updates To and Loading Updates From Databases topic for more information.
Table Files	The TableExtension determines the extension used for all table files used by ElevateDB, the TableIndexExtension determines the extension used for all table index files, the TableBlobExtension determines the extension used for all

	table BLOB files, and the TablePublishExtension determines the extension used for all table publish files. Every table in an ElevateDB database has at least a table file and a table index file. If the table contains BLOB columns, then it will also have a table BLOB file. If the table is published, then it will also have a table publish file. The location of the table files is determined by the path designated for the applicable database when the database was created. Please see the Creating, Altering, or Dropping Configuration Objects topic for more information.

Server Configuration

There are no extra steps required in order to use the TEDBEngine component in ElevateDB as a client engine since the default value of the EngineType property is etClient. However, in order to use the TEDBEngine component in ElevateDB as an ElevateDB Server you will need to make some property changes before starting the engine.

The TEDBEngine component has several key properties that are used to configure the ElevateDB Server, which are described below in order of importance:

Property	Description
EngineType	In order to start the TEDBEngine component as an ElevateDB Server, you must set this property to etServer.
ServerName	This property is used to identify the ElevateDB Server to external clients once they have connected to the ElevateDB Server. The default value is "EDBSrvr".
ServerDescription	This property is used in conjunction with the ServerName property 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".
ServerAddress	This property 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.
ServerPort	This property specifies the port that the ElevateDB Server should bind to when listening for incoming connections from remote sessions. The default value is 12010.
ServerThreadCacheSize	This property 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 property 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.
ServerEncryptionPassword	This property specifies the encryption password used by the ElevateDB Server for encrypting all communications with

	remote sessions. The default encryption password 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.
ServerEncryptedOnly	This property specifies whether all incoming connections from remote sessions should be encrypted or not. If this property is set to True, then all incoming connections to the ElevateDB Server that are not encrypted will be rejected with the error code 1105 (EDB_ERROR_ENCRYPTREQ). The default value is False. **Note** If you intend to use encrypted connections to an ElevateDB Server over a public network then you should always use a different ServerEncryptionPassword from the default password.
ServerSessionTimeout	This property specifies how long the server engine 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 server engine, 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 ServerDeadSessionInterval,ServerDeadSessionExpiration, or ServerMaxDeadSessions properties (see below). A remote session may enable pinging via the TEDBSession RemotePing property in order to prevent the server engine from disconnecting the remote session due to the ServerSessionTimeout property. The default value for this property is 180 seconds, or 3 minutes.
ServerDeadSessionInterval	This property controls how often the server engine will poll the disconnected sessions to see if any need to be removed according to the ServerDeadSessionExpiration, or ServerMaxDeadSessions properties (see below). The default value is 30 seconds.
ServerDeadSessionExpiration	This property controls how long a session can exist in the server in a disconnected, or "dead", state before the server engine 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 server are using pinging via the TEDBSession RemotePing property, then you should set this property to the minimum value of 10 seconds so that sessions are removed as soon as they stop pinging the server. The default value for this property is 300 seconds, or 5 minutes.
ServerMaxDeadSessions	This property controls how many "dead" sessions can accumulate in the server before the server engine begins to remove them immediately, irrespective of the ServerDeadSessionExpiration property above. If the ServerMaxDeadSessions property is exceeded, then the server engine removes the "dead" sessions in oldest-to-youngest order until the number of "dead" sessions is at or under the ServerMaxDeadSessions property setting. The default value for this property is 64.
ServerAuthorizedAddresses	This property controls which IP addresses are authorized to access the 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.
ServerBlockedAddresses	This property controls which IP addresses are not allowed to access the 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.
ServerRunJobs	This property controls whether the server engine is allowed to schedule and run jobs that are defined in the Configuration database. If this property is set to True (the default), then the ServerJobCategory property below determines which category of jobs that the server will schedule and run.
ServerJobCategory	This property controls which job category the server will schedule and run if the ServerRunJobs property is set to True. This property can contain any value, and the default value is blank (''), which indicates that the server engine can run all job categories. A job category is assigned to each job when it is created via the CREATE JOB DDL statement.
OnServerSessionEvent	Attach an event handler for this event in order to take certain actions when a remote session connects, reconnects, logs in, logs out, or disconnects from the server.
ServerTrace	This property controls whether the server will trigger the OnServerTrace event for every request and response to/from the server.

	Warning Do not enable this property in production without being aware of the consequences. Enabling this property can result in a significant amount of overhead, depending upon how the OnServerTrace event is handled. In the ElevateDB Server project that is provided with ElevateDB (see below), enabling this property will generate a large number of trace files that can easily consume large amounts of disk space on a busy server.

ElevateDB comes with a default GUI ElevateDB Server project for Delphi called edbsrvr.dpr (Windows only). You can examine the source code of these projects to see how you would go about setting up a TEDBEngine component as an ElevateDB Server in a project. Both of these projects are also provided in compiled form with ElevateDB. You can find these servers in the \servers\edbsrvr subdirectories under the main ElevateDB installation directory, and you can find the source code to these servers in the \source subdirectory under each server's directory.

Starting the Engine

Once you have configured the engine using the above information, starting the engine is quite simple. All you need to do is set the Active property to True. The following shows an example of how one might configure and start an ElevateDB Server using the default global Engine function in the edbcomps unit (Delphi and Lazarus) or edbcomps header file (C++):

```
with Engine do
   begin
   ConfigPath:='\MyApplication';
   ServerName:='MyTestServer';
   ServerDescription:='My Test Server';
   { Only listen on this IP address }
   ServerAddress:='192.168.0.1';
   Active:=True;
   end;
```

Note
You can use the TEDBEngine BeforeStart event to configure the TEDBEngine component before it is started. Likewise, you can use the AfterStart, BeforeStop, and AfterStop events to respond to the engine being started or stopped.

5.2 Connecting Sessions

As already discussed in the Architecture topic, the TEDBSession component represents a session in ElevateDB. The following information will show how to connect a session in an application.

Preparing a Local Session for Connection

If a TEDBSession component has its SessionType property set to stLocal, then it is considered a local session as opposed to a remote session. A local session must have values assigned to the LoginUser and LoginPassword properties if you do not wish to have ElevateDB display a login dialog when the session is connected.

The default Administrator user and password for an ElevateDB configuration is:

```
User: Administrator (case-insensitive)
Password: EDBDefault (case-sensitive)
```

Preparing a Remote Session for Connection

If a TEDBSession component has its SessionType property set to stRemote, then it is considered a remote session as opposed to a local session. In addition to the Login* properties detailed above that are required for a local or remote session, there are some additional properties for remote sessions that must be specified.

Connecting a remote session will cause ElevateDB to attempt a connection to the ElevateDB Server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties, and the RemoteConnectionTimeout property will indicate how long the remote session will wait for a successful connection attempt. In addition, the RemoteSignature property indicates the signature that the session's connection to the ElevateDB Server will be signed with, and the RemoteEncryption property indicates whether the session's connection to the ElevateDB Server will be encrypted using the RemoteEncryptionPassword property. You must set these properties properly before trying to connect the remote session or an exception will be raised.

Note
Even if a session is not encrypted by setting the RemoteEncryption property to True, any login information is encrypted using the RemoteEncryptionPassword property during session login, so the RemoteEncryptionPassword must always match the corresponding server encryption password for session communciations or logins to the ElevateDB Server will fail. Please see the Configuring and Starting the Engine topic for more information on how to configure the server encryption passowrd used with session communications.

The RemoteAddress and RemoteHost properties are normally mutually exclusive. They can both be specified, but the RemoteHost property will take precedence. The host name used for the server can be specified via the "hosts" text file available from the operating system. In Windows, for example, it is located in the Windows\System32\Drivers\Etc directory. Adding an entry in this file for the ElevateDB Server will allow you to refer to the ElevateDB Server by host name instead of IP address. The following is an example of an entry for an ElevateDB Server running on a LAN:

```
192.168.0.1     ElevateDBServer
```

This is sometimes more convenient than remembering several IP addresses for different ElevateDB Servers. It also allows the IP address to change without having to modify your application.

The RemotePort and RemoteService properties are also normally mutually exclusive. They can both be specified, but the RemoteService property will take precedence. By default the port that ElevateDB Servers use is 12010. This port can be changed, however, so check with your administrator or person in charge of the ElevateDB Server configuration to verify that this is the port being used.

The service name used for the ElevateDB Server can be specified via the "services" text file available from the operating system. In Windows, for example, it's located in the \Windows\System32\Drivers\Etc directory. Adding an entry to this file for the ElevateDB Server's port will allow you to refer to the server's port by service name instead of port number. The following is an example of an entry for the server:

```
ElevateDBServer     12010/tcp
```

This is sometimes more convenient than remembering the port numbers for different ElevateDB Servers. It also allows the port number to change without having to modify your application.

The RemoteEncryption property can be set to either True or False and determines whether the session's connection to the server will be encrypted or not. If this property is set to True, the RemoteEncryptionPassword property is used to encrypt and decrypt all data transmitted to and from the ElevateDB Server. This property must match the same encryption password that the ElevateDB Server is using for communications with remote sessions (TEDBEngine ServerEncryptionPassword property) or else an exception will be raised when a request is attempted on the server.

If for any reason the remote session cannot connect to an ElevateDB Server, an exception will be raised. The error that is raised when a connection fails is 1100 (EDB_ERROR_CLIENTCONN). It's also possible for ElevateDB to be able to connect to the server, but the connection will be rejected due to the ElevateDB Server blocking the client workstation's IP address from accessing the server (1104 and defined as EDB_ERROR_ADDRBLOCK), or an encrypted connection being required by the ElevateDB Server (1105 and defined as EDB_ERROR_ENCRYPTREQ).

Connecting a Session

To connect a session you must set the TEDBSession Active property to True or call its Open method. For a local session (SessionType property is set to stLocal), the session will be opened immediately. As discussed above, for a remote session (SessionType property is set to stRemote), performing this operation will cause the session to attempt a connection to the ElevateDB Server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties. The connection attempt will wait the number of seconds specified by theRemoteConnectionTimeout property.

For both local and remote sessions, if the LoginUser and LoginPassword properties are specified and are valid, then neither the OnLogin event nor the interactive login dialog will be triggered. If these properties are not specified or are not valid, the OnLogin event will be triggered if there is an event handler assigned to it. If an event handler is not assigned to the OnLogin event, ElevateDB will display an interactive login dialog that will prompt for a user ID and password. All ElevateDB configurations require a user ID and password in order to connect and login. ElevateDB will allow for up to 3 login attempts before issuing an exception. The error that is raised when a connection fails due to invalid login attempts is 501 (EDB_ERROR_LOGIN).

Note
Any version of ElevateDB for Delphi 6 or higher (including C++Builder 6 and higher) requires that you include the DBLogDlg unit in your uses clause in order to enable the display of a default login dialog. This is done to allow for ElevateDB to be included in applications without linking in the Forms unit, which can add a lot of unnecessary overhead and also cause unwanted references to user interface libraries. This is not required for Delphi 5 or C++Builder 5 since these versions always included the Forms unit.

The BeforeConnect event is useful for handling the setting of any properties for the session before the session is connected. This event is called right before the session is connected, so it is useful for situations where you need to change the session properties from values that were used at design-time to values that are valid for the environment in which the application is now running. The following is an example of using an BeforeConnect event handler to set the remote connection properties for a session:

```
procedure TMyForm.MySessionBeforeConnect(Sender: TObject);
var
   Registry: TRegistry;
begin
   Registry:=TRegistry.Create;
   try
      Registry.RootKey:=HKEY_LOCAL_MACHINE;
      if Registry.OpenKey('SOFTWARE/My Application',False) then
         begin
         if Registry.ReadBool('IsRemote') then
            begin
            with MySession do
               begin
               SessionType:=stRemote;
               RemoteAddress:=Registry.ReadString('RemoteAddress');
               RemotePort:=Registry.ReadString('RemotePort');
               end;
            end
         else
            MySession.SessionType:=stLocal;
         end
      else
         ShowMessage('Error reading connection information '+
                     'from the registry');
   finally
      Registry.Free;
   end;
end;
```

Note
You should not call the session's Open method or toggle the Active property from within this event handler. Doing so can cause infinite recursion.

The AfterDisconnect event can be used for taking specific actions after a session has been disconnected. As is the case with the BeforeConnect event, the above warning regarding the Open method or Active property also applies for the AfterDisconnect event.

More Session Properties

A session can also be configured to control several global settings for all TEDBDatabase, TEDBTable, TEDBQuery, TEDBStoredProc, and TEDBScript components that link to the session via their SessionName properties. The properties that represent these global settings are detailed below:

Property	Description
ForceBufferFlush	Controls whether the session will automatically force the operating system to flush data to disk after every write operation completed by ElevateDB. Please see the Buffering and Caching topic for more information. The default value is False.
RecordLockProtocol	Controls whether the session will use a pessimistic or optimistic locking model when editing rows via navigational or SQL methods. Please see the Locking and Concurrency topic for more information. The default value is lpPessimistic.
RecordLockRetryCount	Controls the number of times that the engine will retry a row lock before raising an exception. This property is used in conjunction with the RecordLockWaitTime property. The default value is 15 retries.
RecordLockWaitTime	Controls the amount of time, in milliseconds, that the engine will wait in-between row lock attempts. This property is used in conjuction with the RecordLockRetryCount property. The default value is 100 milliseconds.
RecordChangeDetection	Controls whether the session will detect changes to a row during editing or deletion and issue an error if the row has changed since it was last cached. Please see the Change Detection topic for more information. The default value is False.
KeepConnections	Controls whether temporary TEDBDatabase components are kept connected even after they are no longer needed. This property has no obvious effect upon a local session, but can result in tremendous performance improvements for a remote session, therefore it defaults to True and should be left as such in most cases.
KeepTablesOpen	Controls whether the physical tables opened within the session are kept open even after they are no longer being used by the application. Setting this property to True can dramatically improve the performance of SQL statements and any other operations that involve constantly opening and closing the same tables over and over.
SQLStmtCacheSize	Controls how many SQL statements can be cached in memory for each open database in the session. Caching SQL statements improves the performance of ElevateDB by avoiding very expensive preparation/un-preparation cycles. The default value is 0, which means that SQL statements will not be cached for the session. If a session needs to free any cached SQL statements, it can do so at any time by calling the TEDBSession FreeCachedSQLStmts method.
FuncProcCacheSize	Controls how many functions/procedures can be cached in

	memory for each open database in the session. Caching functions/procedures improves the performance of ElevateDB by avoiding very expensive preparation/un-preparation cycles. The default value is 0, which means that functions/procedures will not be cached for the session. If a session needs to free any cached functions/procedures, it can do so at any time by calling the TEDBSession FreeCachedFuncProcs method.
ProgressTimeInterval	Controls the amount of time, in milliseconds, that must elapse between progress updates before ElevateDB will generate a progress event. The default value is 1000 milliseconds, or 1 second.
ExcludeFromLicensedSessions	Specifies whether the current session should be included in the session license count controlled by the TEDBEngine LicensedSessions property for local sessions, or by the ElevateDB Server for remote sessions. This is useful for situations where you have a utility session that you want to exclude from your own licensing restrictions, such as when a session is used in a thread for performance reasons. **Note** This property does not cause the session to be excluded from the ElevateDB licensed session count and only affects the user-defined licensed session count.

Note
You cannot modify any of the above properties unless the session is disconnected. Attempting to modify these properties while the session is connected will result in an exception being raised.

5.3 Creating, Altering, or Dropping Configuration Objects

Configuration objects are objects that are stored in the ElevateDB configuration file, which is represented by the special system-created Configuration database. Creating, altering, or dropping configuration objects can be accomplished by using the TEDBSession Execute method to execute the desired DDL (Data Definition Language) statement against the Configuration database. This method is always set to execute any passed SQL statement from the context of the Configuration database, which makes it ideal for use in creating, altering, or dropping configuration objects such as databases, users, roles, and jobs with a minimal amount of work.

The following example shows how to create a database called "Test" using the CREATE DATABASE DDL statement:

```
// This example uses a session component that
// has already been created and connected
// called MySession

with MySession do
   Execute('CREATE DATABASE "Test" PATH ''C:\Test'''+
           'DESCRIPTION ''Test Database''');
```

Configuration Object DDL Statements

The following DDL statements can be used to manipulate the various configuration objects available in the Configuration database:

- 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 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

Please see the User Security topic for more information on the required privileges to execute the above DDL statements.

Note
Keep in mind that Linux has a case-sensitive file system when specifying path names in any SQL.

5.4 Opening Databases

As already discussed in the ElevateDB Architecture topic, the TEDBDatabase component represents a database in ElevateDB. The following information will show how to open a database in an application.

Preparing a Database for Opening

Before you can open a database using the TEDBDatabase component, you must set a couple of properties. The TEDBDatabase DatabaseName property is the name given to the database within the application and is required for naming purposes only. The Database property should contain the name of an existing database that has already been created using a CREATE DATABASE DDL statement.

Opening a Database

To open a database you must set the TEDBDatabase Connected property to True or call its Open method. For a local TEDBDatabase component whose SessionName property is linked to a local TEDBSession component, the database will cause the local TEDBSession to be opened if it is not already, and then the database will be opened. For a remote database whose SessionName property is linked to a remote TEDBSession component, performing this operation will cause the remote session to attempt a connection to the ElevateDB Server if it is not already connected. If the connection is successful, the database will then be opened.

The BeforeConnect event is useful for handling the setting of any pertinent properties for the TEDBDatabase component before it is opened. This event is triggered right before the database is opened, so it's useful for situations where you need to change the database information from that which was used at design-time to something that is valid for the environment in which the application is now running.

> **Note**
> You should not call the TEDBDatabase Open method or modify the Connected property from within the BeforeConnect event handler. Doing so can cause infinite recursion.

More Database Properties

A TEDBDatabase component has one other property of importance that is detailed below:

Property	Description
KeepConnection	Controls whether the database connection is kept active even after it is no longer needed. This property has no effect upon a local session, but can result in tremendous performance improvements for a remote session, therefore it defaults to True and should be left as such in most cases.

5.5 Creating, Altering, or Dropping Database Objects

Database objects are objects that are stored in an ElevateDB database catalog, which is represented by the special system-created Information schema in every ElevateDB database. Creating, altering, or dropping database objects can be accomplished by using the TEDBDatabase Execute method to execute the desired DDL statement against the target database. This method is always set to execute any passed SQL statement from the context of the target database, which makes it ideal for use in creating, altering, or dropping database objects such as tables, indexes, triggers, views, functions, and procedures with a minimal amount of work.

The following example shows how to create a table called "Customer" using the CREATE TABLE DDL (Data Definition Language) statement:

```
// This example uses a database component that
// has already been created and opened
// called MyDatabase

with MyDatabase do
   Execute('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)'+
           ')');
```

Database Object DDL Statements

The following DDL statements can be used to manipulate the various database objects available in a database catalog:

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- RENAME TABLE
- CREATE INDEX
- CREATE TEXT INDEX
- ALTER INDEX
- DROP INDEX
- RENAME INDEX
- CREATE TRIGGER
- ALTER TRIGGER
- DROP TRIGGER
- RENAME TRIGGER
- CREATE VIEW

- ALTER VIEW
- DROP VIEW
- RENAME VIEW
- CREATE FUNCTION
- ALTER FUNCTION
- DROP FUNCTION
- RENAME FUNCTION
- CREATE PROCEDURE
- ALTER PROCEDURE
- DROP PROCEDURE
- RENAME PROCEDURE

Please see the User Security topic for more information on the required privileges to execute the above DDL statements.

> **Note**
> Keep in mind that Linux has a case-sensitive file system when specifying path names in any SQL.

5.6 Executing Queries

Executing SQL queries is accomplished through the ExecSQL and Open methods of the TEDBQuery component, or by setting the Active property to True. Before executing a query you must first specify the source database for the query. The source database is specified via the DatabaseName property of the TEDBQuery component. The actual SQL for the query is specified in the SQL property. You may select whether you want a sensitive or insensitive query result cursor set via the RequestSensitive property. Please see the Result Set Cursor Sensitivity topic for more information.

Setting the DatabaseName Property

You may specify the DatabaseName property using two different methods:

1) The first method is to set the DatabaseName property of the TEDBQuery component to the DatabaseName property of an existing TEDBDatabase component within the application. In this case the actual source database being used will come from the Database property. The following example shows how to use the DatabaseName property to point to an existing TEDBDatabase component for the source database:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyQuery do
      begin
      DatabaseName:='AccountingDB';
      SQL.Clear;
      SQL.Add('SELECT * FROM ledger');
      Active:=True;
      end;
end;
```

Note
The above example does not assign a value to the SessionName property of either the TEDBDatabase or TEDBQuery component because leaving this property blank for both components means that they will use the default session that is automatically created by ElevateDB when the engine is initialized. This session is, by default, a local, not remote, session named "Default" or "". Please see the Connecting Sessions topic for more information.

Another useful feature is using the BeforeConnect event of the TEDBDatabase component to dynamically set the Database property before the TEDBDatabase component attempts to connect to the database. This is especially important when you have the Connected property for the TEDBDatabase component set to True at design-time during application development and wish to change the Database property before the connection is attempted when the application is run.

2) The second method is to enter the name of an existing database directly into the DatabaseName property. In this case a temporary database component will be automatically created, if needed, for the

database specified and automatically destroyed when no longer needed. The following example shows how to use the DatabaseName property to point directly to the desired database without referring to a TEDBDatabase component:

```
begin
   with MySession do
      begin
      SessionName:='Remote';
      SessionType:=stRemote;
      RemoteAddress:='192.168.0.2';
      Active:=True;
      end;
   with MyQuery do
      begin
      SessionName:='Remote';
      DatabaseName:='Accounting';
      SQL.Clear;
      SQL.Add('SELECT * FROM ledger');
      Active:=True;
      end;
end;
```

Setting the SQL Property

The SQL statement is specified via the SQL property of the TEDBQuery component. The SQL property is a TEDBStrings object. You may enter an SQL statement by using the Add method of the SQL property to specify the SQL statement line-by-line. You can also assign the entire SQL to the Text property of the SQL property.

When dynamically building SQL statements that contain literal string constants, you can use the TEDBEngine QuotedSQLStr method to properly format and escape any embedded single quotes in the string. For example, suppose you have a TEdit component that contains the following string:

```
Pete's Garage
```

The string contains an embedded single quote, so it cannot be specified directly without causing an error in the SQL statement.

To build an SQL INSERT statement that inserts the above string into a VARCHAR column, you should use the following code:

```
MyEDBQuery.SQL.Text:='INSERT INTO MyTable '+
   '(MyVarCharColumn) VALUES ('+
   Engine.QuotedSQLStr(MyEdit.Text)+')';
```

Note
If re-using the same TEDBQuery component for multiple query executions, please be sure to call the SQL property's Clear method to clear the SQL from the previous query before calling the Add method to add more SQL statement lines.

Preparing the Query

By default ElevateDB will automatically prepare a query before it is executed. However, you may also manually prepare a query using the TEDBQuery Prepare method. Once a query has been prepared, the Prepared property will be True. Preparing a query parses the SQL, opens all referenced tables, and prepares all internal structures for the execution of the query. You should only need to manually prepare a query when executing a parameterized query. Please see the Parameterized Queries topic for more information.

Executing the Query

To execute the query you should call the TEDBQuery ExecSQL or Open methods, or you should set the Active property to True. Setting the Active property to True is the same as calling the Open method. The difference between using the ExecSQL and Open methods is as follows:

Method	Usage
ExecSQL	Use this method when the SQL statement specified in the SQL property may or may not return a result set. The ExecSQL method can handle both situations.
Open	Use this method only when you know that the SQL statement specified in the SQL property will return a result set. Using the Open method with an SQL statement that does not return a result set will result in an EDatabaseError exception being raised with an error message "Error creating table handle".

Note
The SQL SELECT statement is the only statement that returns a result set. All other types of SQL statements do not.

The following example shows how to use the ExecSQL method to execute an UPDATE statement:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyQuery do
      begin
      DatabaseName:='AccountingDB';
      SQL.Clear;
      SQL.Add('UPDATE ledger SET AccountNo=100');
```

```
      SQL.Add('WHERE AccountNo=300');
      ExecSQL;
      end;
end;
```

Query Execution Plans

If you wish to retrieve a query execution plan for the current execution via the Plan property, then set the RequestPlan property to True before executing the query.

Sensitive Result Set Cursors

If you wish to have a sensitive result set generated from the executed query, then set the RequestSensitive property to True before executing the query. This only requests a sensitive result set cursor, and the query may still generate an insensitive result set cursor based upon the query being executed. Please see the Result Set Cursor Sensitivity topic for more information.

Retrieving Query Information

You can retrieve information about a query both after the query has been prepared and after the query has been executed. The following properties can be interrogated after a query has been prepared or executed:

Property	Description
SQLStatementType	Indicates the type of SQL statement currently ready for execution.

The following properties can only be interrogated after a query has been executed:

Property	Description
Plan	Contains information about how the current query was executed, including any optimizations performed by ElevateDB. This information is very useful in determining how to optimize a query further or to simply figure out what ElevateDB is doing behind the scenes. The Plan property is automatically cleared before each execution of an SQL statement. **Note** Query plans are only generated for SQL SELECT, INSERT, UPDATE, or DELETE statements.
RowsAffected	Indicates the number of rows affected by the current query.
ExecutionTime	Indicates the amount of execution time in seconds consumed by the current query.
ExecutionResult	Indicates the Boolean result of the current SQL execution.

Sensitive	Indicates the whether the result set cursor for the query is sensitive or insensitive. Please see the Result Set Cursor Sensitivity topic for more information.

The following example shows how to use the ExecSQL method to execute an UPDATE SQL statement and report the number of rows affected as well as how long it took to execute the statement:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyQuery do
      begin
      DatabaseName:='AccountingDB';
      SQL.Clear;
      SQL.Add('UPDATE ledger SET AccountNo=100');
      SQL.Add('WHERE AccountNo=300');
      ExecSQL;
      ShowMessage(IntToStr(RowsAffected)+
                  ' rows updated in '+
                  FloatToStr(ExecutionTime)+' seconds');
      end;
end;
```

Tracking the Progress of a Query

To take care of tracking the progress of the query execution, we have provided the TEDBQuery OnProgress event. You may set the Continue parameter of this event to False in your event handler to indicate to ElevateDB that you wish to abort the execution of the current SQL statement.

5.7 Parameterized Queries

Parameters allow the same SQL statement to be used with different data values, and are placeholders for those data values. At runtime, the application prepares the query with the parameters and fills the parameter with a value before the query is executed. When the query is executed, the data values assigned to the parameters are substituted for the parameter placeholder and the SQL statement is executed.

Specifying Parameters in SQL

Parameter markers can be used in SQL SELECT, INSERT, UPDATE, and DELETE statements in place of constants. Parameters are identified by a preceding colon (:). For example:

```
SELECT Last_Name, First_Name
FROM Customer
WHERE (Last_Name=:LName) AND (First_Name=:FName)
```

Parameters are used to pass data values to be used in WHERE clause comparisons and as update values in updating SQL statements such as UPDATE or INSERT. Parameters cannot be used to pass values for Identifiers. The following example uses the TotalParam parameter to pass the data value that needs to be assigned to the ItemsTotal column for the row with the OrderNo column equal to 1014:

```
UPDATE Orders
SET ItemsTotal = :TotalParam
WHERE (OrderNo = 1014)
```

Populating Parameters with the TEDBQuery Component

You can use the TEDBQuery Params property to populate the parameters in an SQL statement with data values. You may use two different methods of populating parameters using the Params property:

- By referencing each parameter by its index position in the available list of parameters
- By reference each parameter by name using the ParamByName method

The following is an example of using the index positions of the parameters to populate the data values for an INSERT SQL statement:

```
begin
   with MyQuery do
      begin
      SQL.Clear;
      SQL.Add('INSERT INTO Country (Name, Capital, Population)');
      SQL.Add('VALUES (:Name, :Capital, :Population)');
      Params[0].AsString := 'Lichtenstein';
      Params[1].AsString := 'Vaduz';
      Params[2].AsInteger := 420000;
      ExecSQL;
      end;
```

```
end;
```

The next block of code is an example of using the TEDBQuery ParamByName method in order to populate the data values for a SELECT SQL statement:

```
begin
   with MyQuery do
      begin
      SQL.Clear;
      SQL.Add('SELECT *');
      SQL.Add('FROM Orders');
      SQL.Add('WHERE CustID = :CustID');
      ParamByName('CustID').AsFloat:=1221;
      Open;
      end;
end;
```

Preparing Parameterized Queries

It is usually recommended that you manually prepare parameterized queries that you intend to execute many times with different parameter values. This can result in significant performance improvements since the process of preparing a query can be time-consuming. The following is an example of inserting 3 rows with different values using a manually-prepared, parameterized query:

```
begin
   with MyQuery do
      begin
      SQL.Clear;
      SQL.Add('INSERT INTO Customer (CustNo, Company');
      SQL.Add('VALUES (:CustNo, :Company)');
      { Manually prepare the query }
      Prepare;
      ParamByName('CustNo').AsInteger:=1000;
      ParamByName('Company').AsString:='Chocolates, Inc.';
      ExecSQL;
      ParamByName('CustNo').AsInteger:=2000;
      ParamByName('Company').AsString:='Flowers, Inc.';
      ExecSQL;
      ParamByName('CustNo').AsInteger:=3000;
      ParamByName('Company').AsString:='Candies, Inc.';
      ExecSQL;
      end;
end;
```

5.8 Querying Configuration Objects

Configuration objects are objects that are stored in the ElevateDB configuration file, which is represented by the special system-created Configuration database. Querying configuration objects can be accomplished by using the TEDBQuery component to execute queries against the Configuration database. This allows you to determine which configuration objects exist in the configuration along with specific information about the configuration objects.

The following example shows how to use a TEDBQuery component containing a SELECT statement to query the Databases Table in the Configuration database in order to see if the "Sales" database exists:

```
// This example uses a query component that
// has already been created and opened
// called MyQuery

with MyQuery do
   begin
   DatabaseName:='Configuration';
   SQL:='SELECT * FROM Databases '+
        'WHERE Name='+Engine.QuotedSQLStr('Sales');
   Open;
   if (RecordCount=1) then
      ShowMessage('The Sales database exists')
   else
      ShowMessage('The Sales database does not exist');
   end;
```

You can also use the TEDBSession Execute method as a quicker method to determine if a configuration object or objects exist. The Execute method returns the number of rows affected or returned by a particular SQL statement, so you can use the return value of an indication of whether any rows exist for the SELECT statement on the Configuration database:

```
// This example uses a session component that
// has already been created and opened
// called MySession

with MySession do
   begin
   if (Execute('SELECT * FROM Databases '+
               'WHERE Name='+Engine.QuotedSQLStr('Sales'))=1) then
      ShowMessage('The Sales database exists')
   else
      ShowMessage('The Sales database does not exist');
   end;
```

5.9 Querying Database Objects

Database objects are objects that are stored in an ElevateDB database catalog, which is represented by the special system-created Information schema in every ElevateDB database. Querying database objects can be accomplished by using the TEDBQuery component to execute queries against the Information Schema for a given database. This allows you to determine which database objects exist in the database along with specific information about the database objects.

The following example shows how to use a TEDBQuery component containing a SELECT statement to query the Tables Table in the Information Schema in order to see if the "Customer" table exists:

```
// This example uses a query component that
// has already been created and opened
// called MyQuery

with MyQuery do
   begin
   DatabaseName:='SalesDB';
   SQL:='SELECT * FROM Information.Tables '+
        'WHERE Name='+Engine.QuotedSQLStr('Customer');
   Open;
   if (RecordCount=1) then
      ShowMessage('The Customer table exists')
   else
      ShowMessage('The Customer table does not exist');
   end;
```

You can also use the TEDBDatabase Execute method as a quicker method to determine if a database object or objects exist. The Execute method returns the number of rows affected or returned by a particular SQL statement, so you can use the return value of an indication of whether any rows exist for the SELECT statement on the Information schema:

```
// This example uses a database component that
// has already been created and opened
// called MyDatabase

with MyDatabase do
   begin
   if (Execute('SELECT * FROM Information.Tables '+
               'WHERE Name='+Engine.QuotedSQLStr('Customer'))=1) then
      ShowMessage('The Customer table exists')
   else
      ShowMessage('The Customer table does not exist');
   end;
```

5.10 Executing Scripts

Executing scripts is accomplished through the ExecScript and Open methods of the TEDBScript component, or by setting the Active property to True. Before executing a script you must first specify the source database for the script. The source database is specified via the DatabaseName property of the TEDBScript component. The actual script is specified in the SQL property.

Setting the DatabaseName Property

You may specify the DatabaseName property using two different methods:

1) The first method is to set the DatabaseName property of the TEDBScript component to the DatabaseName property of an existing TEDBDatabase component within the application. In this case the actual source database being used will come from the Database property. The following example shows how to use the DatabaseName property to point to an existing TEDBDatabase component for the source database:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyScript do
      begin
      DatabaseName:='AccountingDB';
      SQL.LoadFromFile('c:\scripts\GetLedgerEntries.sql');
      Active:=True;
      end;
end;
```

Note
The above example does not assign a value to the SessionName property of either the TEDBDatabase or TEDBScript component because leaving this property blank for both components means that they will use the default session that is automatically created by ElevateDB when the engine is initialized. This session is, by default, a local, not remote, session named "Default" or "". Please see the Connecting Sessions topic for more information.

Another useful feature is using the BeforeConnect event of the TEDBDatabase component to dynamically set the Database property before the TEDBDatabase component attempts to connect to the database. This is especially important when you have the Connected property for the TEDBDatabase component set to True at design-time during application development and wish to change the Database property before the connection is attempted when the application is run.

2) The second method is to enter the name of an existing database directly into the DatabaseName property. In this case a temporary database component will be automatically created, if needed, for the database specified and automatically destroyed when no longer needed. The following example shows how to use the DatabaseName property to point directly to the desired database without referring to a TEDBDatabase component:

```
begin
   with MySession do
      begin
      SessionName:='Remote';
      SessionType:=stRemote;
      RemoteAddress:='192.168.0.2';
      Active:=True;
      end;
   with MyScript do
      begin
      SessionName:='Remote';
      DatabaseName:='Accounting';
      SQL.Clear;
      SQL.Add('SCRIPT ()');
      SQL.Add('BEGIN');
      SQL.Add('   EXECUTE IMMEDIATE ''BACKUP DATABASE Test ');
      SQL.Add('       AS TestBackup TO STORE "Backups" ');
      SQL.Add('       INCLUDE CATALOG'';');
      SQL.Add('END');
      ExecScript;
      end;
end;
```

Setting the SQL Property

The script is specified via the SQL property of the TEDBScript component. You can use the ConvertSQL method to convert a script that consists of a series of SQL statements (INSERT, UPDATE, DELETE, or SELECT) separated by semicolons (;) into a proper ElevateDB script that can be executed by the TEDBScript component.

Preparing the script

By default ElevateDB will automatically prepare a script before it is executed. However, you may also manually prepare a script using the TEDBScript Prepare method. Once a script has been prepared, the Prepared property will be True. Preparing a script compiles the script, opens all referenced tables, and prepares all internal structures for the execution of the script. You should only need to manually prepare a script when executing a script that requires parameters.

Executing the Script

To execute the script you should call the TEDBScript ExecScript or Open methods, or you should set the Active property to True. Setting the Active property to True is the same as calling the Open method. The difference between using the ExecScript and Open methods is as follows:

Method	Usage

ExecScript	Use this method when the script specified in the SQL property may or may not return a result set. The ExecScript method can handle both situations.
Open	Use this method only when you know that the script specified in the SQL property will return a result set. Using the Open method with a script that does not return a result set will result in an EDatabaseError exception being raised with an error message "Error creating table handle".

The following example shows how to use the ExecScript method to execute a script:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyScript do
      begin
      DatabaseName:='AccountingDB';
      SQL.LoadFromFile('UpdateLedgerEntries.SQL');
      Prepare;
      ParamByName('AccountNo').AsString:='00100';
      ExecScript;
      end;
end;
```

Tracking the Progress of a Script

To take care of tracking the progress of the script execution, we have provided the TEDBScript OnProgress event. This event will only be fired if the script contains manual progress update calls specifically included by the script creator.

5.11 Executing Stored Procedures

Executing stored procedures is accomplished through the ExecProc and Open methods of the TEDBStoredProc component, or by setting the Active property to True. Before executing a stored procedure you must first specify the source database for the procedure. The source database is specified via the DatabaseName property of the TEDBStoredProc component. The actual procedure name is specified in the StoredProcName property.

Setting the DatabaseName Property

You may specify the DatabaseName property using two different methods:

1) The first method is to set the DatabaseName property of the TEDBStoredProc component to the DatabaseName property of an existing TEDBDatabase component within the application. In this case the actual source database being used will come from the Database property. The following example shows how to use the DatabaseName property to point to an existing TEDBDatabase component for the source database:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyStoredProc do
      begin
      DatabaseName:='AccountingDB';
      StoredProcName:='GetLedgerEntries';
      Active:=True;
      end;
end;
```

Note
The above example does not assign a value to the SessionName property of either the TEDBDatabase or TEDBStoredProc component because leaving this property blank for both components means that they will use the default session that is automatically created by ElevateDB when the engine is initialized. This session is, by default, a local, not remote, session named "Default" or "". Please see the Connecting Sessions topic for more information.

Another useful feature is using the BeforeConnect event of the TEDBDatabase component to dynamically set the Database property before the TEDBDatabase component attempts to connect to the database. This is especially important when you have the Connected property for the TEDBDatabase component set to True at design-time during application development and wish to change the Database property before the connection is attempted when the application is run.

2) The second method is to enter the name of an existing database directly into the DatabaseName property. In this case a temporary database component will be automatically created, if needed, for the database specified and automatically destroyed when no longer needed. The following example shows how to use the DatabaseName property to point directly to the desired database without referring to a

TEDBDatabase component:

```
begin
   with MySession do
      begin
      SessionName:='Remote';
      SessionType:=stRemote;
      RemoteAddress:='192.168.0.2';
      Active:=True;
      end;
   with MyStoredProc do
      begin
      SessionName:='Remote';
      DatabaseName:='Accounting';
      StoredProcName:='GetLedgerEntries';
      Active:=True;
      end;
end;
```

Setting the StoredProcName Property

The procedure is specified via the StoredProcName property of the TEDBStoredProc component.

Preparing the Stored Procedure

By default ElevateDB will automatically prepare a procedure before it is executed. However, you may also manually prepare a procedure using the TEDBStoredProc Prepare method. Once a procedure has been prepared, the Prepared property will be True. Preparing a procedure compiles the procedure, opens all referenced tables, and prepares all internal structures for the execution of the procedure. You should only need to manually prepare a procedure when executing a procedure that requires parameters.

Executing the Procedure

To execute the procedure you should call the TEDBStoredProc ExecProc or Open methods, or you should set the Active property to True. Setting the Active property to True is the same as calling the Open method. The difference between using the ExecProc and Open methods is as follows:

Method	Usage
ExecProc	Use this method when the procedure specified in the StoredProcName property may or may not return a result set. The ExecProc method can handle both situations.
Open	Use this method only when you know that the procedure specified in the StoredProcName property will return a result set. Using the Open method with a procedure that does not return a result set will result in an EDatabaseError exception being raised with an error message "Error creating table handle".

The following example shows how to use the ExecProc method to execute a procedure:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyStoredProc do
      begin
      DatabaseName:='AccountingDB';
      StoredProcName='UpdateLedgerEntries';
      Prepare;
      ParamByName('AccountNo').AsString:='00100';
      ExecProc;
      end;
end;
```

Tracking the Progress of a Procedure

To take care of tracking the progress of the procedure execution, we have provided the TEDBStoredProc OnProgress event. This event will only be fired if the procedure contains manual progress update calls specifically included by the procedure creator.

5.12 Executing Transactions

A transaction is executed entirely by using the StartTransaction, Commit, and Rollback methods of the TEDBDatabase component. A typical transaction block of code looks like this:

```
begin
   with MyDatabase do
      begin
      StartTransaction(EmptyEDBStringsArray);
      try
         { Perform some updates to the table(s) in this database }
         Commit;
      except
         Rollback;
      end;
      end;
end;
```

The EmptyEDBStringsArray variable is defined in the edbtype unit (Delphi or Lazarus) or edbtype header file (C++) in ElevateDB.

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 locks held by the transaction are released and other sessions can continue to update data while the exception is dealt with. Also, if you roll back a transaction it is always a good idea to refresh any open TEDBTable or TEDBQuery components linked to the TEDBDatabase component involved in the transaction so that they reflect the current data and not any data from the transaction that was just rolled back. Along with refreshing, you should make sure that any pending inserts or edits for the TEDBTable or TEDBQuery components are cancelled using the Cancel method before the transaction is rolled back to ensure that the inserts or edits are not accidentally posted using the Post method after the transaction is rolled back (unless that is specifically what you wish to do).

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 StartTransaction method accepts an optional array 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 nil (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:

```
var
   Tables: TEDBStringsArray;
begin
   with MyDatabase do
      begin
```

```
      SetLength(Tables,2);
      Tables[0]:='Customer';
      Tables[1]:='Orders';
      StartTransaction(Tables);
      try
         { Perform some updates to the table(s) in the transaction }
         Commit;
      except
         Rollback;
         raise;
      end;
      end;
end;
```

For more information on transactions in ElevateDB, please see the Transactions topic.

5.13 Creating and Using Stores

A store is simply a named storage area that holds files and includes user security privileges so that you can prevent any accidental destruction or viewing of sensitive files. Creating, altering, and dropping stores, and working with the files contained within them, is accomplished by using the TEDBSession Execute method to execute the CREATE STORE, ALTER STORE, DROP STORE, RENAME STORE,SET FILES STORE, COPY FILE, RENAME FILE, and DELETE FILE statements. You can also attach event handlers to the TEDBSession OnStatusMessage and OnProgress events in order to track any status messages and progress during a file copy operation.

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.

Creating a Store

To create a store, you can use the CREATE STORE statement. If, at a later time, you wish to change the store from a local store to a remote store, or vice-versa, you can do so by using the ALTER STORE statement.

Adding Files to a Store

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.

```
begin
   MySession.Execute('COPY FILE "MyBackup.EDBkp" IN STORE "LocalStore" '+
                     'TO "MyBackup.EDBBkp" IN STORE "RemoteStore"');
end;
```

Tracking the Copy File Progress

To take care of tracking the progress of copying files we have provided the OnProgress and OnStatusMessage events within the TEDBSession component. The OnProgress event will report the progress of the file copy operation and the OnStatusMessage event will report any status messages regarding the file copy operation.

Retrieving Information About Files

To retrieve information about the files in a specific store, you can use the SET FILES STORE statement to specify the store where the files are located, and then use a SELECT statement to query the Files Table in the Configuration Database. The Files table contains information about all of the files in the store specified by the SET BACKUPS STORE statement, with one row per file. Please see the Executing SQL Statements for more information on executing a query.

5.14 Publishing and Unpublishing Databases

Publishing and unpublishing databases is accomplished by using the TEDBSession Execute method to execute the PUBLISH DATABASE and UNPUBLISH DATABASE statements. You can also attach event handlers to the TEDBSession OnStatusMessage event in order to track any status messages during a publish or unpublish operation.

Publishing a database causes ElevateDB to mark all tables that are included in the publishing as published and begin to log all insert, update, or delete operations on the published tables. ElevateDB then will continue to log all such operations until a SAVE UPDATES statement is executed for the published tables, at which time an update file will be created that contains these logged updates, and then remove the logged updates from the log associated with each published table.

The logging of the updates for a published table works as follows for each type of operation:

Operation	Description
Inserts	All modified columns are logged.
Updates	The primary key columns for the pre-update version of the row are logged, and all new modified columns are logged also.
Deletes	The primary key columns for the pre-delete version of the row are logged.

Unpublishing a database causes ElevateDB to mark all tables that are included in the unpublishing as unpublished, and to drop all logged updates for the table, making a backup of the logged updates in the process. The unpublish process effectively undoes the publishing process.

Publishing a Database

When the publish executes, it has to obtain an exclusive lock on all tables that are being published in the specified database. This is due to the fact that publishing a table alters its metadata in the database catalog.

The following example shows how to publish a database called "MyDatabase" using the PUBLISH DATABASE statement and the TEDBSession Execute method:

```
begin
   MySession.Execute('PUBLISH DATABASE "MyDatabase"');
end;
```

You can also, optionally, use the TABLES clause of the PUBLISH DATABASE statement to specify a subset of tables in the DATABASE to publish.

Tracking the Publish Progress

To take care of tracking the status of the publishing we have provided the OnStatusMessage event within the TEDBSession component. The OnStatusMessage event will report any status messages regarding the publishing operation.

Unpublishing a Database

When the unpublish executes, it has to obtain an exclusive lock on all tables that are being unpublished in the specified database. This is due to the fact that unpublishing a table alters its metadata in the database catalog.

The following example shows how to unpublish a database called "MyDatabase" using the UNPUBLISH DATABASE statement and the TEDBSession Execute method:

```
begin
   MySession.Execute('UNPUBLISH DATABASE "MyDatabase"');
end;
```

You can also, optionally, use the TABLES clause of the UNPUBLISH DATABASE statement to specify a subset of tables in the DATABASE to unpublish.

Retrieving Publishing Information

To retrieve information about which tables are published, an when they were published, you can use a SELECT statement to query the Tables Table in the Information schema in the published database. The Tables table contains information about all of the tables in the published database, with one row per table. Please see the Executing SQL Statements for more information on executing a query.

5.15 Saving Updates To and Loading Updates From Databases

Saving updates to databases and loading updates from databases is accomplished by using the TEDBSession Execute method to execute the SAVE UPDATES, SET UPDATES STORE, and LOAD UPDATES statements. You can also attach event handlers to the TEDBSession OnStatusMessage and OnProgress events in order to track any status messages and progress during a save or load operation.

Saving the updates to a database copies the updates to all or some of the tables within the database to a compressed or uncompressed update file in a local store. Loading the updates from a database applies the updates from all or some of the tables in a compressed or uncompressed update file in a local store into the database.

In order to save the updates for a given table or tables in a database, the database table(s) must be published first using the PUBLISH DATABASE statement. Please see the Publishing and Unpublishing Databases topic for more information.

Saving the Updates for a Database

When the updates are saved, a read lock is obtained for all tables whose updates are being saved that prevents any sessions from performing any writes to any of the involved tables in the database until the save completes. However, since the saving of the updates is quite fast, the time during which the tables cannot be changed is usually pretty small. To ensure that the database is available as much as possible for updating, it is recommended that you save the database updates to a file in a local store on a fast hard drive and then copy the file to a store that references a CD, DVD, or other slower device outside of the scope of the database being locked instead of creating the update file directly in the store on the slower device.

The following example shows how to save the updates for a database called "MyDatabase" using the SAVE UPDATES statement and the TEDBSession Execute method:

```
begin
   MySession.Execute('SAVE UPDATES FOR DATABASE "MyDatabase" '+
                     'AS "MyDatabase-Updates-'+
                        Engine.DateToSQLStr(Date)+'" '+
                     'TO STORE "Updates"');
end;
```

Note
You cannot specify a remote store as the location for the update file. It must be a local store. Please see the Creating and Using Stores for more information on stores.

Tracking the Progress of the Saving

To take care of tracking the progress of the saving we have provided the OnProgress and OnStatusMessage events within the TEDBSession component. The OnProgress event will report the progress of the saving operation and the OnStatusMessage event will report any status messages regarding the saving operation.

Retrieving Information from an Update File

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. Please see the Executing SQL Statements for more information on executing a query.

Loading the Updates for a Database

When the updates are loaded, a write lock is obtained for all of the tables specified for the load that prevents any sessions from performing any reads or writes to any of the specified tables until the load completes. However, since the execution of a load is quite fast, the time during which the tables cannot be accessed is usually pretty small.

Note
Update files from the same source database should always be loaded in their creation order. For example, if you have 3 update files that have come from two different copies of the database, then the 2 update files from one of the source databases should be loaded in their creation order. The other update file doesn't matter because updates from different source databases can be loaded in any order. You can find out the creation order by querying the Updates table in the Configuration database, as described above in the Retrieving Information from an Update File section.

The following example shows how to load the updates for a database called "MyDatabase" using the LOAD UPDATES statement and the TEDBSession Execute method:

```
begin
   MySession.Execute('LOAD UPDATES FOR DATABASE "MyDatabase" '+
                     'FROM "MyDatabase-Updates-'+
                        Engine.DateToSQLStr(Date)+'" '+
                     'IN STORE "Updates"');
end;
```

Note
You cannot specify a remote store as the location for the update file. It must be a local store. Please see the Creating and Using Stores for more information on stores.

Tracking the Progress of the Loading

To take care of tracking the progress of the loading we have provided the OnProgress and OnStatusMessage events within the TEDBSession component. The OnProgress event will report the progress of the load operation and the OnStatusMessage event will report any status messages regarding the load operation.

5.16 Backing Up and Restoring Databases

Backing up and restoring databases is accomplished by using the TEDBSession Execute method to execute the BACKUP DATABASE, SET BACKUPS STORE, and RESTORE DATABASE statements. You can also attach event handlers to the TEDBSession OnStatusMessage and OnProgress events in order to track any status messages and progress during a backup or restore operation.

Backing up a database copies all or some of the tables within the database, along with (optionally) the database catalog, to a compressed or uncompressed backup file in a local store. Restoring a database copies all or some of the tables in a compressed or uncompressed backup file in a local store into the database, overwriting any tables with the same names that already exist in the database. You can also choose to restore the database catalog during a restore operation, if the database catalog was backed up originally with the tables.

Backing Up a Database

When the backup executes, it obtains a read lock for the entire database that prevents any sessions from performing any writes to any of the tables in the database until the backup completes. However, since the execution of a backup is quite fast, the time during which the tables cannot be changed is usually pretty small. To ensure that the database is available as much as possible for updating, it is recommended that you backup the database to a file in a local store on a fast hard drive and then copy the file to a store that references a CD, DVD, or other slower backup device outside of the scope of the database being locked instead of creating the backup file directly in the store on the slower backup device.

The following example shows how to backup a database called "MyDatabase" using the BACKUP DATABASE statement and the TEDBSession Execute method:

```
begin
   MySession.Execute('BACKUP DATABASE "MyDatabase" '+
                     'AS "MyDatabase-Backup-'+
                        Engine.DateToSQLStr(Date)+'" '+
                     'TO STORE "Backups" '+
                     'INCLUDE CATALOG');
end;
```

Note
You cannot specify a remote store as the location for the backup file. It must be a local store. Please see the Creating and Using Stores for more information on stores.

Tracking the Backup Progress

To take care of tracking the progress of the backup we have provided the OnProgress and OnStatusMessage events within the TEDBSession component. The OnProgress event will report the progress of the backup operation and the OnStatusMessage event will report any status messages regarding the backup operation.

Retrieving Information from a Backup File

To retrieve information about the backup files in a specific store, you can use the SET BACKUPS STORE statement to specify the store where the backup files are located, and then use a SELECT statement to query the Backups Table in the Configuration Database. The Backups table contains information about all of the backup files in the store specified by the SET BACKUPS STORE statement, with one row per backup file. Please see the Executing SQL Statements for more information on executing a query.

Restoring a Database

When the restore executes, it obtains an exclusive lock for the entire database that prevents any sessions from opening the database until the restore completes. However, since the execution of a restore is quite fast, the time during which the database cannot be accessed is usually pretty small.

> **Note**
> The Restore method overwrites any existing database catalogs and tables. You should be very careful when restoring to an existing database to prevent loss of data.

The following example shows how to restore a database called "MyDatabase" using the RESTORE DATABASE statement and the TEDBSession Execute method:

```
begin
   MySession.Execute('RESTORE DATABASE "MyDatabase" '+
                     'FROM "MyDatabase-Backup-'+
                        Engine.DateToSQLStr(Date)+'" '+
                     'IN STORE "Backups" '+
                     'INCLUDE CATALOG');
end;
```

> **Note**
> You cannot specify a remote store as the location for the backup file. It must be a local store. Please see the Creating and Using Stores for more information on stores.

Tracking the Restore Progress

To take care of tracking the progress of the restore we have provided the OnProgress and OnStatusMessage events within the TEDBSession component. The OnProgress event will report the progress of the restore operation and the OnStatusMessage event will report any status messages regarding the restore operation.

5.17 Opening Tables and Views

Opening tables and views can be accomplished through the Open method of the TEDBTable component, or by setting the Active property to True. Before opening a table or view, however, you must first specify the source database of the table or view and the table or view name. The source database of the table or view is specified in the DatabaseName property of the TEDBTable component, and the table or view name is specified in the TableName property.

Setting the DatabaseName Property

You may specify the DatabaseName property using two different methods:

1) The first method is to set the DatabaseName property of the TEDBTable component to the DatabaseName property of an existing TEDBDatabase component within the application. In this case the actual source database being used will come from the Database property. The following example shows how to use the DatabaseName property to point to an existing TEDBDatabase component for the source database:

```
begin
   with MyDatabase do
      begin
      DatabaseName:='AccountingDB';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyTable do
      begin
      DatabaseName:='AccountingDB';
      TableName:='ledger';
      Active:=True;
      end;
end;
```

Note
The above example does not assign a value to the SessionName property of either the TEDBDatabase or TEDBTable component because leaving this property blank for both components means that they will use the default session that is automatically created by ElevateDB when the engine is initialized. This session is, by default, a local, not remote, session named "Default" or "". Please see the Starting Sessions topic for more information.

Another useful feature is using the BeforeConnect event of the TEDBDatabase component to dynamically set the Directory or RemoteDatabase property before the TEDBDatabase component attempts to connect to the database. This is especially important when you have the Connected property for the TEDBDatabase component set to True at design-time during application development and wish to change the Directory or RemoteDatabase property before the connection is attempted when the application is run.

2) The second method is to enter the name of an existing database directly into the DatabaseName property. In this case a temporary database component will be automatically created, if needed, for the database specified and automatically destroyed when no longer needed. The following example shows how to use the DatabaseName property to point directly to the desired database without referring to a

TEDBDatabase component:

```
begin
   with MySession do
      begin
      SessionName:='Remote';
      SessionType:=stRemote;
      RemoteAddress:='192.168.0.2';
      Active:=True;
      end;
   with MyTable do
      begin
      SessionName:='Remote';
      DatabaseName:='Accounting';
      TableName:='ledger';
      Active:=True;
      end;
end;
```

Exclusive and ReadOnly Open Modes

In the above two examples we have left the Exclusive and ReadOnly properties of the TEDBTable component at their default value of False. However, you can use these two properties to control how the table or view is opened and how that open affects the ability of other sessions and users to open the same table or view.

When the Exclusive property is set to True, the table or view specified in the TableName property will be opened exclusively when the Open method is called or the Active property is set to True. This means that neither the current session nor any other session or user may open this table or view again without causing an EEDBError exception. It also means that the table or view open will fail if anyone else has the table or view opened either shared (Exclusive=False) or exclusively (Exclusive=True). The error code raised when a table open fails due to access problems is 300 (EDB_ERROR_LOCK). The following example shows how to trap for such an exception using a try..except block (Delphi and Lazarus) or try..catch block (C++) and display an appropriate error message to the user:

```
begin
   with MySession do
      begin
      SessionName:='Remote';
      SessionType:=stRemote;
      RemoteAddress:='192.168.0.2';
      Active:=True;
      end;
   with MyDatabase do
      begin
      SessionName:='Remote';
      DatabaseName:='AccountingData';
      Database:='Accounting';
      Connected:=True;
      end;
   with MyTable do
      begin
      SessionName:='Remote';
      { We're using a database component for the source
```

```
        database, so we use the same value as the DatabaseName
        property for the TEDBDatabase component above, not
        the same value as the Database property, which
        is the name of the actual database }
      DatabaseName:='AccountingData';
      TableName:='ledger';
      Exclusive:=True;
      ReadOnly:=False;
      try
         Open;
      except
         on E: Exception do
            begin
            if (E is EDatabaseError) and
               (E is EEDBError) then
               begin
               if (EEDBError(E).ErrorCode=EDB_ERROR_LOCK) then
                  ShowMessage('Cannot open table '+TableName+
                              ', another user has the table '+
                              'open already')
               else
                  ShowMessage('Unknown or unexpected database '+
                              'engine error # '+
                              IntToStr(EEDBError(E).ErrorCode));
               end
            else
               ShowMessage('Unknown or unexpected error has occurred');
            end;
      end;
      end;
end;
```

Note
Regardless of whether you are trying to open a table or view exclusively, you can still receive this exception if another user or application has opened the table or view exclusively.

When the ReadOnly property is set to True, the table or view specified in the TableName property will be opened read-only when the Open method is called or the Active property is set to True. This means that the TEDBTable component will not be able to modify the contents of the table or view until the table is closed and re-opened with write access (ReadOnly=False). If any of the physical files that make up a table are marked read-only at the operating system level (such as is the case with CD-ROMs) then ElevateDB automatically detects this condition and sets the ReadOnly property to True. ElevateDB is also able to do extensive read buffering on any table that is marked read-only at the operating system level, so if your application is only requiring read-only access then it would provide a big performance boost to mark the tables as read-only at the operating system level. Finally, if security permissions for any of the physical files that make up the table prevent ElevateDB from opening the table with write access, then ElevateDB will also automatically detect this condition and set the ReadOnly property to True.

Updateable Views

Views behave just like tables in most cases. However, views can only be updated if they are actually flagged as updateable by ElevateDB when they are created. You can find out if a view is updateable by querying the Views Table in the Information Schema for the current database. For a view to be flagged as updateable, it must adhere to the requirements of a query that can generate a sensitive result set cursor.

Please see the Result Set Cursor Sensitivity topic for more information. If a view is not updateable, then it will always have its ReadOnly property set to True when it is opened.

5.18 Closing Tables and Views

Closing tables and views can be accomplished through the Close method of the TEDBTable component, or by setting the Active property to False.

The following example shows how to use the Close method to close a table:

```
begin
   MyTable.Close;
end;
```

Note
Once a table or view is closed you cannot perform any operations on the table or view until the table or view is opened again.

5.19 Navigating Tables, Views, and Query Result Sets

Navigation of tables, views, and query result sets is accomplished through several methods of the TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components. The basic navigational methods include the First, Next, Prior, Last, and MoveBy methods. The Bof and Eof properties indicate whether the row pointer is at the beginning or at the end of the table, view, or query result set, respectively. These methods and properties are used together to navigate a table, view, or query result set.

Moving to the First or Last Row

The First method moves to the first row in the table, view, or query result set based upon the current index order. The Last method moves to the last row in the table, view, or query result set based upon the current index order. The following example shows how to move to the first and last rows in a table:

```
begin
   with MyTable do
      begin
      First;
      {  do something to the first row }
      Last;
      { do something to the last row }
      end;
end;
```

Skipping Rows

The Next method moves to the next row in the table, view, or query result set based upon the current index order. If the current row pointer is at the last row in the table, view, or query result set, then calling the Next method will set the Eof property to True and the row pointer will stay on the last row. The Prior method moves to the previous row in the table, view, or query result set based upon the current index order. If the current row pointer is at the first row in the table, view, or query result set, then calling the Prior method will set the Bof property to True and the row pointer will stay on the first row. The following example shows how to use the First and Next methods along with the Eof property to loop through an entire table:

```
begin
   with MyTable do
      begin
      First;
      while not Eof do
         Next;
      end;
end;
```

The following example shows how to use the Last and Prior methods along with the Bof property to loop backwards through an entire table:

```
begin
   with MyTable do
```

```
      begin
      Last;
      while not Bof do
         Prior;
      end;
end;
```

Skipping Multiple Rows

The MoveBy method accepts a positive or negative integer that represents the number of rows to move by within the table, view, or query result set. A positive integer indicates that the movement will be forward while a negative integer indicates that the movement will be backward. The return value of the MoveBy method is the number of rows actually visited during the execution of the MoveBy method. If the row pointer hits the beginning of file or hits the end of file then the return value of the MoveBy method will be less than the desired number of rows. The following example shows how to use the MoveBy method to loop through an entire table 10 rows at a time:

```
begin
   with MyTable do
      begin
      First;
      while not Eof do
         MoveBy(10);
      end;
end;
```

5.20 Inserting, Updating, and Deleting Rows

Updating of tables, views, and query result sets is accomplished through several methods of the TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components. The basic update methods include the Append, Insert, Edit, Delete, FieldByName, Post, and Cancel methods. The State property indicates whether the current table, view, or query result set is in Append/Insert mode (dsInsert), Edit mode (dsEdit), or Browse mode (dsBrowse). These methods and properties are used together in order to update a table, view, or query result set. Depending upon your needs, you may require additional methods to update BLOB columns within a given table, view, or query result set, and information on how to use these methods are discussed at the end of this topic.

Note
For the rest of this topic, a table, view, or query result set will be referred to as a dataset to reduce the amount of references to both. Also, it is important to note here that a query result set can be either sensitive or insensitive, which affects whether an update to a query result set is permitted or not. Please see the Result Set Cursor Sensitivity topic for more information. Likewise, a view may or may not be updateable depending upon the view definition. Please see the Opening Tables and Views topic for more information on updateable views.

Adding a New Row

The Append and Insert methods allow you to begin the process of adding a row to the dataset. The only difference between these two methods is the Insert method will insert a blank row buffer at the current position in the dataset, and the Append method will add a blank row buffer at the end of the dataset. This row buffer does not exist in the physical datset until the row buffer is posted to the actual dataset using the Post method. If the Cancel method is called, then the row buffer and any updates to it will be discarded. Also, once the row buffer is posted using the Post method it will be positioned in the dataset according to the active index order, not according to where it was positioned due to the Insert or Append methods.

The FieldByName method can be used to reference a specific column for updating and accepts one parameter, the name of the column to reference. This method returns a TField object if the column name exists or an error if the column name does not exists. This TField object can be used to update the data for that column in the row buffer via properties such as AsString, AsInteger, etc.

The following example shows how to use the Append method to add a row to a table with the following structure:

```
Column #    Name              DataType     Size
-------------------------------------------
1          CustomerID        ftString     10
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
7          LastSaleDate      ftDate       0
8          Notes             ftMemo       0

Index Name      Columns In Index      Options
-------------------------------------------
```

```
Primary_Key     CustomerID          ixPrimary
```

```
begin
   with MyEDBDataSet do
      begin
      Append;  { State property will now reflect dsInsert }
      FieldByName('CustomerID').AsString:='100';
      FieldByName('CustomerName').AsString:='The Hardware Store';
      FieldByName('ContactName').AsString:='Bob Smith';
      FieldByName('Phone').AsString:='5551212';
      FieldByName('Fax').AsString:='5551616';
      FieldByName('Email').AsString:='bobs@thehardwarestore.com';
      Post;  { State property will now return to dsBrowse }
      end;
end;
```

If the row that is being posted violates a table constraint for the dataset then an EEDBError exception will be raised with the error code 1004 (EDB_ERROR_CONSTRAINT). Please see the Exception Handling and Errors and Appendix A - Error Codes and Messages topics for general information on exception handling in ElevateDB.

You may use the OnPostError event to trap for any of these error conditions and display a message to the user. You can also use a try..except block to do the same, and the approach is very similar. The following shows how to use an OnPostError event handler to trap for a constraint error:

```
procedure TMyForm.MyTablePostError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
begin
   Action:=daAbort;
   if (E is EEDBError) then
      begin
      if (EEDBError(E).ErrorCode=EDB_ERROR_CONSTRAINT) then
         ShowMessage('This row violates a table or column constraint ('+
                     E.Message+')')
      else
         ShowMessage(E.Message);
      end
   else
      ShowMessage(E.Message);
end;
```

Note
You will notice that the OnPostError event handler uses the more general EDatabaseError exception object for it's exception (E) parameter. Because of this, you must always first determine whether the exception object being passed is actually an EEDBError before casting the exception object and trying to access specific properties such as the ErrorCode property. The EEDBError object descends from the EDatabaseError object.

The following shows how to use a try..except block to trap for a constraint error:

```
begin
   try
      with MyEDBDataSet do
         begin
         Append;  { State property will now reflect dsInsert }
         FieldByName('CustomerID').AsString:='100';
         FieldByName('CustomerName').AsString:='The Hardware Store';
         FieldByName('ContactName').AsString:='Bob Smith';
         FieldByName('Phone').AsString:='5551212';
         FieldByName('Fax').AsString:='5551616';
         FieldByName('Email').AsString:='bobs@thehardwarestore.com';
         Post;  { State property will now return to dsBrowse }
         end;
   except
      on E: Exception do
         begin
         if (E is EEDBError) then
            begin
            if (EEDBError(E).ErrorCode=EDB_ERROR_CONSTRAINT) then
               ShowMessage('This row violates a table or column constraint
      ('+
                           E.Message+')')
            else
               ShowMessage(E.Message);
            end
         else
            ShowMessage(E.Message);
         end;
   end;
end;
```

Editing an Existing Row

The Edit method allows you to begin the process of editing an existing row in the dataset. ElevateDB offers the choice of a pessimistic or optimistic locking protocol, which is configurable via the RecordLockProtocol property for the TEDBSession assigned to the current dataset (see the SessionName property for more information on setting the session for a dataset). With the pessimistic locking protocol a row lock is obtained when the Edit method is called. As long as the row is being edited ElevateDB will hold a row lock on that row, and will not release this lock until either the Post or Cancel methods is called. With the optimistic locking protocol a row lock is not obtained until the Post method is called, and never obtained if the Cancel method is called. This means that another user or session is capable of editing the row and posting the changes to the row before the Post method is called, thus potentially causing an EEDBError exception to be raised with the error code 1007 (EDB_ERROR_ROWDELETED), or even error code 1008 (EDB_ERROR_ROWMODIFIED) if row change detection is turned on for the current session via the TEDBSession RecordChangeDetection property. In such cases you must discard the edited row by calling the Cancel method and begin again with a fresh copy of the row using the Edit method.

Note
Any updates to the row are done via a row buffer and do not actually exist in the actual dataset until the row is posted using the Post method. If the Cancel method is called, then any updates to the row will be discarded. Also, once the row is posted using the Post method it will be positioned in the dataset according to the active index order based upon any changes made to the row. What this means is that if any column that is part of the current active index is changed, then it is possible for the row to re-position itself in a completely different place in the dataset after the Post method is called.

The following example shows how to use the Edit method to update a row in a dataset:

```
begin
   with MyEDBDataSet do
      begin
      Edit;  { State property will now reflect dsEdit }
      { Set LastSaleDate column to today's date }
      FieldByName('LastSaleDate').AsDateTime:=Date;
      Post;  { State property will now return to dsBrowse }
      end;
end;
```

If the row that you are attempting to edit (or post, if using the optimistic locking protocol) is already locked by another session, then an EEDBError exception will be raised with the error code 1005 (EDB_ERROR_LOCKROW).

It is also possible that the row that you are attempting to edit (or post) has been deleted by another session since it was last cached by ElevateDB. If this is the case then a ElevateDB exception will be raised with the error code 1007 (EDB_ERROR_ROWDELETED). If row change detection is enabled, then it is also possible that the row that you are attempting to edit (or post) has been changed by another session since it was last cached by ElevateDB. If this is the case then a ElevateDB exception will be raised with the error code 1008 (EDB_ERROR_ROWMODIFIED).

You may use the OnEditError (or OnPostError, depending upon the locking protocol) event to trap for these error conditions and display a message to the user. You can also use a try..except block to do the same, and the approach is very similar. The following shows how to use an OnEditError event handler to trap for several errors:

```
procedure TMyForm.MyTableEditError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
begin
   Action:=daAbort;
   if (E is EEDBError) then
      begin
      if (EEDBError(E).ErrorCode=EDB_ERROR_LOCKROW) then
         begin
         if MessageDlg('The row you are trying to edit '+
                       'is currently locked, do you want to '+
                       'try to edit this row again?',
                        mtWarning,[mbYes,mbNo],0)=mrYes then
            Action:=daRetry;
         end
      else if (EEDBError(E).ErrorCode=EDB_ERROR_ROWDELETED) then
```

```
         begin
         MessageDlg('The row you are trying to edit '+
                    'has been deleted since it was last '+
                    'retrieved',mtError,[mbOk],0);
         DataSet.Refresh;
         end
      else if (EEDBError(E).ErrorCode=EDB_ERROR_ROWMODIFIED) then
         begin
         MessageDlg('The row you are trying to edit '+
                    'has been modified since it was last '+
                    'retrieved, the row will now be '+
                    'refreshed',mtWarning,[mbOk],0);
         DataSet.Refresh;
         Action:=daRetry;
         end
      else
         MessageDlg(E.Message,mtError,[mbOK],0);
      end
   else
      MessageDlg(E.Message,mtError,[mbOK],0);
end;
```

The following shows how to use a try..except block to trap for several errors:

```
begin
   while True do
      begin
      try
         with MyEDBDataSet do
            begin
            Edit;  { State property will now reflect dsEdit }
            { Set LastSaleDate column to today's date }
            FieldByName('LastSaleDate').AsDateTime:=Date;
            Post;  { State property will now return to dsBrowse }
            end;
         Break; { Break out of retry loop }
      except
         on E: Exception do
            begin
            if (E is EEDBError) then
               begin
               if (EEDBError(E).ErrorCode=EDB_ERROR_LOCKROW) then
                  begin
                  if MessageDlg('The row you are trying '+
                                'to edit is currently locked, '+
                                'do you want to try to edit '+
                                'this row again?,mtWarning,
                                [mbYes,mbNo],0)=mrYes then
                     Continue;
                  end
               else if (EEDBError(E).ErrorCode=EDB_ERROR_ROWDELETED) then
                  begin
                  MessageDlg('The row you are trying '+
                             'to edit has been deleted '+
                             'since it was last retrieved',
                             mtError,[mbOk],0);
                  MyTable.Refresh;
```

```
                  Break;
                  end
               else if (EEDBError(E).ErrorCode=EDB_ERROR_ROWMODIFIED) then
                  begin
                  MessageDlg('The row you are trying '+
                             'to edit has been modified '+
                             'since it was last retrieved, '+
                             'the row will now be '+
                             'refreshed',mtWarning,[mbOk],0);
                  MyTable.Refresh;
                  Continue;
                  end
               else
                  begin
                  MessageDlg(E.Message,mtError,[mbOK],0);
                  Break;
                  end;
                end
            else
               begin
               MessageDlg(E.Message,mtError,[mbOK],0);
               Break;
               end;
            end;
      end;
      end;
end;
```

Deleting an Existing Row

The Delete method allows you to delete an existing row in a dataset. Unlike the Append, Insert, and Edit methods, the Delete method is a one-step process and does not require a call to the Post method to complete its operation. A row lock is obtained when the Delete method is called and is released as soon as the method completes. After the row is deleted the current position in the dataset will be the next closest row based upon the active index order.

The following example shows how to use the Delete method to delete a row in a dataset:

```
begin
   with MyEDBDataSet do
      Delete;
end;
```

If the row that you are attempting to delete is already locked by another user or session, then an EEDBError exception will be raised with the error code 1005 (EDB_ERROR_LOCKROW).

It is also possible that the row that you are attempting to delete has been deleted by another session since it was last cached by ElevateDB. If this is the case then a ElevateDB exception will be raised with the error code 1007 (EDB_ERROR_ROWDELETED). If row change detection is enabled, then it is also possible that the row that you are attempting to delete has been changed by another session since it was last cached by ElevateDB. If this is the case then a ElevateDB exception will be raised with the error code 1008 (EDB_ERROR_ROWMODIFIED).

You may use the OnDeleteError event to trap for these error conditions and display a message to the user.

You can also use a try..except block to do the same, and the approach is very similar. The code for an handling Delete errors is the same as that of an Edit, so please refer to the above code samples for handling Edit errors.

Cancelling an Insert/Append or Edit Operation

You may cancel an existing Insert/Append or Edit operation by calling the Cancel method. Doing this will discard any updates to an existing row if you are editing, or will completely discard a new row if you are inserting or appending. The following example shows how to cancel an edit operation on an existing row:

```
begin
   with MyEDBDataSet do
      begin
      Edit;  { State property will now reflect dsEdit }
      { Set LastSaleDate column to today's date }
      FieldByName('LastSaleDate').AsDateTime:=Date;
      Cancel;  { State property will now return to dsBrowse }
      end;
end;
```

Additional Events

There are several additional events that can be used to hook into the updating process for a dataset. They include the BeforeInsert, AfterInsert, OnNewRow, BeforeEdit, AfterEdit, BeforeDelete, AfterDelete, BeforePost, AfterPost, BeforeCancel, and AfterCancel events. All of these events are fairly self-explanatory, however the OnNewRow is special in that it can be used to assign values to columns in a newly-inserted or appended row without having the dataset mark the row as modified. If a row has not been modified in any manner, then the dataset will not perform an implicit Post operation when navigating off of the row. Instead, the Cancel method will be called and the row discarded.

Updating BLOB and CLOB Columns

Most of the time you can simply use the general TField AsString and AsVariant properties to update a BLOB or CLOB column in the same fashion as you would any other column. Both of these properties allow very large strings or binary data to be stored in a BLOB or CLOB column. However, in certain cases you may want to take advantage of additional methods and functionality that are available through the TBlobField object that descends from TField or the TEDBBlobStream object that provides a stream interface to a BLOB or CLOB column. The most interesting methods of the TBlobField object are the LoadFromFile, LoadFromStream, SaveToFile, and SaveToStream methods. These methods allow you to very easily load and save the data to and from BLOB and CLOB columns.

Note
You must make sure that the dataset's State property is either dsInsert or dsEdit before using the LoadFromFile or LoadFromStream methods.

The following is an example of using the LoadFromFile method of the TBlobField object to load the contents of a text file into a CLOB column:

```
begin
```

```
   with MyEDBDataSet do
      begin
      Edit;  { State property will now reflect dsEdit }
      { Load a text file from disk }
      TBlobField(FieldByName('Notes')).LoadFromFile('c:\temp\test.txt');
      Post;  { State property will now return to dsBrowse }
      end;
end;
```

Note
You'll notice that we must cast the result of the FieldByName method, which returns a TField object reference, to a TBlobField type in order to allow us to call the LoadFromFile method. This is okay since a CLOB column uses a TMemoField object, which descends directly from TBlobField, which itself descends directly from TField.

In addition to these very useful methods, you can also directly manipulate a BLOB or CLOB column like any other stream by using the TEDBBlobStream object. The following is an example of using a TEDBBlobStream component along with the TEDBTable or TEDBQuery SaveToStream method for storing ElevateDB tables themselves in the BLOB column of another table:

```
var
   BlobStream: TEDBBlobStream;
begin
   { First create the BLOB stream - be sure to make sure that
     we put the table into dsEdit or dsInsert mode first since
     we're writing to the BLOB stream }
   FirstEDBDataSet.Append;
   try
      BlobStream:=TEDBBlobStream.Create(TBlobField(
          FirstEDBDataSet.FieldByName('TableStream')),bmWrite);
      try
         { Now save the table to the BLOB stream }
         SecondEDBDataSet.SaveToStream(BlobStream);
      finally
         { Be sure to free the BLOB stream *before* the Post }
         BlobStream.Free;
      end;
      FirstEDBDataSet.Post;
   except
      { Cancel on an exception }
      FirstEDBDataSet.Cancel;
   end;
end;
```

Note
For proper results when updating a BLOB or CLOB column using a TEDBBlobStream object, you must create the TEDBBlobStream object after calling the Append/Insert or Edit methods for the dataset containing the BLOB or CLOB column. Also, you must free the TEDBBlobStream object before calling the Post method to post the changes to the dataset. Finally, be sure to use the proper open mode when creating a TEDBBlobStream object for updating (either bmReadWrite or bmWrite).

5.21 Searching and Sorting Tables, Views, and Query Result Sets

Searching and sorting tables, views, and query result sets is accomplished through several methods of the TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components. The basic searching methods for tables (not views or query result sets) include the FindKey, FindNearest, SetKey, EditKey, GotoKey, and GotoNearest methods. The KeyColumnCount property is used with the SetKey and EditKey methods to control searching using the GotoKey and GotoNearest methods. The extended searching methods that do not necessarily rely upon an index and can be used with both tables and query result sets include the Locate, FindFirst, FindLast, FindNext, and FindPrior methods. The basic sorting methods for tables include the IndexName and IndexFieldNames properties.

Changing the Sort Order

You may use the TEDBTable IndexName and IndexFieldNames properties to set the current index order, and in effect, sort the current table based upon the index definition for the selected index order.

The IndexName property is used to set the name of the current index. This property should be set to the name of the index that you wish to use as the current index order. Setting the IndexName property to blank (') will cause the index order to reset to the default order for the table, which is usually the order defined by the primary key of the table, or the natural insertion order of the table if the table does not have a primary key defined. The following example shows how you would set the current index order for a table to an index called "CustomerName":

```
begin
   with MyTable do
      begin
      IndexName:='CustomerName';
      { do something }
      end;
end;
```

Note
Changing the index order can cause the current row pointer to move to a different position in the table (but not necessarily move off of the current row unless the row has been changed or deleted by another session). Call the First method after setting the IndexName property if you want to have the row pointer set to the beginning of the table based upon the next index order. Changing the index order will also remove any ranges that are active.

If you attempt to set the IndexName property to a non-existent index an EEDBError exception will be raised with the error code 401 (EDB_ERROR_NOTFOUND).

The IndexFieldNames property is used to set the current index order by specifying the column names of the desired index instead of the index name. Multiple column names should be separated with a semicolon. Using the IndexFieldNames property is desirable in cases where you are trying to set the current index order based upon a known set of columns and do not have any knowledge of the index names available. The IndexFieldNames property will attempt to match the given number of columns with the same number of beginning columns, in left-to-right order, in any of the available indexes for the table. The following example shows how you would set the current index order to an index called "CustomerName" that consists of the CustomerName column and the CustomerNo column:

```
begin
   with MyTable do
      begin
      IndexFieldNames:='CustomerName;CustomerNo';
      { do something }
      end;
end;
```

Note
Setting the IndexFieldNames will not work on indexes that contain descending columns or contain columns using case-insensitive collations, so you must use the IndexName property instead. Please see the Internationalization topic for information on collations and index columns.

If ElevateDB cannot find any indexes that match the desired column names an EDatabaseError exception will be raised instead of an EEDBError exception. If you are using this method of setting the current index order you should also be prepared to trap for this exception and deal with it appropriately.

Searching Using an Index

The TEDBTable FindKey method accepts an array of search values to use in order to perform an exact search for a given row using the active index. The return value of the FindKey method indicates whether the search was successful. If the search was successful then the row pointer is moved to the desired row, whereas if the search was not successful then the row pointer stays at its current position. The search values must correspond to the columns that make up the active index or the search will not work properly. However, FindKey does not require that you fill in all of the column values for all of the columns in the active index, rather only that you fill in the column values from left to right. The following example shows how to perform a search on the index used to enforce the primary key and comprised of the CustomerNo column:

```
begin
   with MyTable do
      begin
      { Set to the natural order, which in this case
        is the primary key }
      IndexName:='';
      { Search for customer 100 }
      if FindKey([100]) then
         { Row was found, now do something }
      else
         ShowMessage('Row was not found');
      end;
end;
```

The FindNearest method accepts an array of search values to use in order to perform a near search for a given row using the active index. If the search was successful then the row pointer is moved to the desired row, whereas if the search was not successful then the row pointer is moved to the next row that most closely matches the current search values. If there are no rows that are greater than the search values then the row pointer will be positioned at the end of the table. The search values must correspond to the columns that make up the active index or the search will not work properly. However, FindNearest

does not require that you fill in all of the column values for all of the columns in the active index, rather only that you fill in the column values from left to right. The following example shows how to perform a near search on the index used to enforce the primary key and comprised of the CustomerNo column:

```
begin
   with MyTable do
      begin
      { Set to the natural order, which in this case
        is the primary key }
      IndexName:='';
      { Search for customer 100 or closest }
      FindNearest([100]);
      end;
end;
```

The SetKey and EditKey methods are used in conjunction with the GotoKey and GotoNearest methods to perform searching using column assignments instead of an array of column values. The SetKey method begins the search process by putting the TEDBTable component into the dsSetKey state and clearing all column values. You can examine the state of the table using the State property. The application must then assign values to the desired columns and call the GotoKey or GotoNearest method to perform the actual search. The GotoNearest method may be used if you wish to perform a near search instead of an exact search. The EditKey method extends or continues the current search process by putting the TEDBTable component into the dsSetKey state but not clearing any column values. This allows you to change only one column without being forced to re-enter all column values needed for the search. The KeyColumnCount property controls how many columns, based upon the current index, are to be used in the actual search. By default the KeyColumnCount property is set to the number of columns for the active index. The following example shows how to perform an exact search using the SetKey and GotoKey methods and KeyColumnCount property. The active index is an index called "CustomerName" comprised of the CustomerName column and the CustomerNo column:

```
begin
   with MyTable do
      begin
      { Set to the CustomerName index }
      IndexName:='CustomerName';
      { Search for the customer with the
        name 'The Hardware Store' }
      SetKey;
      ColumnByName('CustomerName').AsString:='The Hardware Store';
      { This causes the search to only look at the first column
        in the current index when searching }
      KeyColumnCount:=1;
      if GotoKey then
         { Row was found, now do something }
      else
         ShowMessage('Row was not found');
      end;
end;
```

Note
In the previous example we executed a partial-column search. What this means is that we did not include all of the columns in the active index. ElevateDB does not require that you use all of the columns in the active index for searching.

The following example shows how to perform a near search using the SetKey and GotoNearest methods, and KeyColumnCount property. The active index is an index called "CustomerName" comprised of the CustomerName column and the CustomerNo column:

```
begin
   with MyTable do
      begin
      { Set to the CustomerName index }
      IndexName:='CustomerName';
      { Search for the customer with the
        name 'The Hardware Store' }
      SetKey;
      ColumnByName('CustomerName').AsString:='The Hardware Store';
      { This causes the search to only look at the first column
        in the current index when searching }
      KeyColumnCount:=1;
      GotoNearest;
      end;
end;
```

Searching Without a Specific Index Order Set

The Locate method of the TEDBTable, TEDBQuery, and TEDBStoredProc components is used to locate a row independent of the active index order or of any indexes at all. This is why it can be used with query result sets in addition to tables. The Locate method will attempt to use the active index for searching, but if the current search columns do not match the active index then the Locate method will attempt to use another available index. Indexes are selected based upon the options passed to the Locate method in conjunction with the column names that you wish to search upon. The index columns are checked from left to right, and if an index is found that matches the search columns from left to right and satisfies the options desired for the search it will be used to perform the search. Finally, if no indexes can be found that can be used for the search, a table scan will be used to execute the search instead. This is usually a sub-optimal solution and can take a bit of time since the table scan will read every row in the table in order to examine the desired column values. The Locate method uses the following criteria when determining whether to use an index or not for the search:

1) ElevateDB matches the index columns to the search columns in left-to-right order.

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

3) ElevateDB can only use an index for the search if the first column(s) in the index in left-to-right order match(es) both the column(s) being searched upon and the setting of the loCaseInsensitive flag in the Locate options. If the loCaseInsensitive flag is not specified, then the index column in the index (being examined for possible use in the search) must be assigned a case-sensitive collation. If the loCaseInsensitive flag is specified, then the index column in the index must be assigned a case-insensitive collation.

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 code:

```
begin
   with MyTable do
      begin
      { Search for the customer with the
        state "FL" }
      if Locate('State',['FL'],[loCaseInsensitive]) then
         { Row was found, now do something }
      else
         ShowMessage('Row was not found');
      end;
end;
```

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

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

In order to allow ElevateDB to use this index to optimize any searches on the State column, you must now not include the loCaseInsensitive flag:

```
begin
   with MyTable do
      begin
      { Search for the customer with the
        state "FL" }
      if Locate('State',['FL'],[]) then
         { Row was found, now do something }
      else
         ShowMessage('Row was not found');
      end;
end;
```

Please see the Internationalization topic for more information on collations.

The FindFirst, FindLast, FindNext, and FindPrior methods all rely on the Filter and FilterOptions properties to do their work. These methods are the most flexible for searching and can be used with both tables and query result sets, but there are some important caveats. To get acceptable performance from these methods you must make sure that the filter expression being used for the Filter property is optimized or at

least partially-optimized. If the filter expression is un-optimized it will take a significantly greater amount of time to complete every call to any of the FindFirst, FindLast, FindNext, or FindPrior methods unless the table or query result set being searched only has a small number of rows. Please see the Setting Filters on Tables and Query Result Sets topic for more information. Also, because the Filter property is being used for these methods, you cannot use a different filter expression in combination with these methods. However, you can set the Filtered property to True and show only the filtered rows if you so desire. Finally, the FilterOptions property controls how the filtering is performed during the searching, so you should make sure that these options are set properly. The following example shows how to use the Filter property and FindFirst and FindNext methods to find matching rows and navigate through them in a table:

```
begin
   with MyTable do
      begin
      { Search for the first customer with the
        name "The Hardware Store" }
      Filter:='CustomerName='+QuotedStr('The Hardware Store');
      { We want the search to be case-insensitive }
      FilterOptions:=[foCaseInsensitive];
      if FindFirst then
         begin
         { Row was found, now search through
           the rest of the matching rows }
         while FindNext do
            { Do something here }
         end
      else
         ShowMessage('Row was not found');
      end;
end;
```

5.22 Setting Ranges on Tables

Setting ranges on tables is accomplished through several methods of the TEDBTable component. The basic range methods include the SetRange, SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, and ApplyRange methods. The KeyColumnCount property is used with the SetRangeStart, SetRangeEnd, EditRangeStart and EditRangeEnd methods to control searching using the ApplyRange method. All range operations are dependent upon the active index order set using the IndexName or IndexFieldNames properties. Ranges may be combined with expression filters set using the Filter and Filtered propertes and/or code-based filters set using the OnFilterRow event to further filter the rows in the table.

Setting a Range

The SetRange method accepts two arrays of values to use in order to set a range on a given table. If the current row pointer does not fall into the range values specified, then the current row pointer will be moved to the nearest row that falls within the range. These value arrays must contain the column values in the same order as the column names in the active index or the range will not return the desired results. However, SetRange does not require that you fill in all of the column values for all of the columns in the active index, rather only that you fill in the column values from left to right. The following example shows how to perform a range on the index used to enforce the primary key and comprised of the CustomerNo column:

```
begin
   with MyTable do
      begin
      { Set to the natural order, which in this case
        is the primary key }
      IndexName:='';
      { Set a range from customer 100 to customer 300 }
      SetRange([100],[300]);
      end;
end;
```

The SetRangeStart, SetRangeEnd, EditRangeStart, and EditRangeEnd methods are used in conjunction with the ApplyRange method to perform a range using column assignments instead of arrays of column values. The SetRangeStart method begins the range process by putting the TEDBTable component into the dsSetKey state and clearing all column values. You can examine the state of the table using the State property. The application must then assign values to the desired columns for the start of the range and then proceed to call SetRangeEnd to assign values to the desired columns for the end of the range. After this is done the application can call the ApplyRange method to perform the actual range operation. The EditRangeStart and EditRangeEnd methods extend or continue the current range process by putting the TEDBTable component into the dsSetKey state but not clearing any column values. You can examine the state of the table using the State property. This allows you to change only one column without being forced to re-enter all column values needed for the beginning or ending values of the range. The KeyColumnCount property controls how many columns, based upon the active index, are to be used in the actual range and can be set independently for both the starting and ending column values of the range. By default the KeyColumnCount property is set to the number of columns in the active index. The following example shows how to perform a range using the SetRangeStart, SetRangeEnd, and ApplyRange methods and KeyColumnCount property. The active index is an index called "CustomerName" that consists of the CustomerName column and the CustomerNo column:

```
begin
```

```
   with MyTable do
      begin
      { Set to the CustomerName index }
      IndexName:='CustomerName';
      { Set a range to find all customers with
        a name beginning with 'A' }
      SetRangeStart;
      ColumnByName('CustomerName').AsString:='A';
      { This causes the range to only look at
        the first column in the current index }
      KeyColumnCount:=1;
      SetRangeEnd;
      { Note the padding of the ending range
        values with lowercase z's
        to the length of the CustomerName
        column, which is 20 characters }
      ColumnByName('CustomerName').AsString:='Azzzzzzzzzzzzzzzzzz';
      { This causes the range to only look at
        the first column in the current index }
      KeyColumnCount:=1;
      ApplyRange;
      end;
end;
```

Note
In the previous example we executed a partial-column range. What this means is that we did not include all of the columns in the active index in the range. ElevateDB does not require that you use all of the columns in the active index for the range.

5.23 Setting Master-Detail Links on Tables

A master-detail link is a property-based linkage between a master TDataSource component and a detail TEDBTable component. Once a master-detail link is established, any changes to the master TDataSource component will cause the detail TEDBTable component to automatically reflect the change and show only the detail rows that match the current master row based upon the link criteria. Master-detail links use ranges for their functionality, and therefore are dependent upon the active index in the detail table. Like ranges, master-detail links may be combined with expression filters set using the Filter and Filtered propertes and/or code-based filters set using the OnFilterRow event to further filter the rows in the detail table.

Defining the Link Properties

Setting master-detail links on tables is accomplished through four properties in the detail TEDBTable component. These properties are the MasterSource, MasterColumns, IndexName, and IndexFieldNames properties.

The first step in setting a master-detail link is to assign the MasterSource property. The MasterSource property refers to a TDataSource component. This makes master-detail links very flexible, because the TDataSource component can provide data from any TDataSet-descendant component such as a TEDBTable or TEDBQuery component as well as many other non-ElevateDB dataset components.

> **Note**
> For the link to be valid, the TDataSource DataSet property must refer to a valid TDataSet-descendant component.

The next step is to assign the IndexName property, or IndexFieldNames property, so that the active index, and the columns that make up that index, will match the columns that you wish to use for the link. The only difference between specifying the IndexName property versus the IndexFieldNames property is that the IndexName property expects the name of an index, whereas the IndexFieldNames only expects the names of columns in the table that match the columns found in an index in the table from left-to-right. The IndexFieldNames property also does not require that all of the columns in an existing index be specified in order to match with that existing index, only enough to be able to select the index so that it will satisfy the needs of the master-detail link.

Finally, the MasterColumns property must be assigned a value. This property requires a column or list of columns separated by semicolons from the master data source that match the columns in the active index for the detail table.

To illustrate all of this we'll use an example. Let's suppose that we have two tables with the following structure and we wish to link them via a master-detail link:

```
Customer Table

Column #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
```

```
6          EMail             ftString     30
```

Note
Indexes in this case are not important since this will be the master table

```
Orders Table

Column #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          OrderNumber       ftString     10
3          OrderDate         ftDate       0
4          OrderAmount       ftBCD        2

Index Name      Columns In Index             Options
-----------------------------------------------------
Primary_Key     CustomerID;OrderNumber      ixPrimary
```

We would use the following example code to establish a master-detail link between the two tables. In this example it is assumed that a TDataSource component called CustomerSource exists and points to a TEDBTable component for the "customer" table:

```
begin
   with OrdersTable do
      begin
      { Set to the natural order, which in this case
        is the primary key }
      IndexName:='';
      { Assign the MasterSource property }
      MasterSource:=CustomerSource;
      { Set the MasterColumns property to point to the
        CustomerID column from the Customer table }
      MasterColumns:='CustomerID';
      end;
end;
```

Now any time the current row in the CustomerSource data source changes in any way, the OrdersTable will automatically reflect that change and only show rows that match the master row's CustomerID column. Below is the same example, but changed to use the IndexFieldNames property instead:

```
begin
   with OrdersTable do
      begin
      { Set to the CustomerID column }
      IndexFieldNames:='CustomerID';
      { Assign the MasterSource property }
      MasterSource:=CustomerSource;
      { Set the MasterColumns property to point to the
        CustomerID column from the Customer table }
```

```
      MasterColumns:='CustomerID';
      end;
end;
```

Note
Because a master-detail link uses data-event notification in the TDataSource component for maintaining the link, if the TDataSet component referred to by the TDataSource component's DataSet property calls its DisableControls method, it will not only disable the updating of any data-aware controls that refer to it, but it will also disable any master-detail links that refer to it also. This is the way the TDataSet and TDataSource components have been designed, so this is an expected behavior that you should keep in mind when designing your application.

5.24 Setting Filters on Tables, Views, and Query Result Sets

Setting filters on tables, views, and query result sets is accomplished through several properties of the TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components. These properties include the Filter, FilterOptions, and Filtered properties. The OnFilterRow event is used to assign a code-based filter event handler that can be used to filter rows using Delphi, C++Builder, or Lazarus code. All filter operations are completely independent of any active index order.

Setting an Expression Filter

The Filter, FilterOptions, Filtered, and FilterOptimizeLevel properties are used to set an expression filter. The steps to set an expression filter include setting the filter expression using the Filter property, specifying any filter options using the FilterOptions property, and then making the expression filter active by setting the Filtered property to True. You can turn off or disable an expression filter by setting the Filtered property to False. If the current row pointer does not fall into the conditions specified by an expression filter, then the current row pointer will be moved to the nearest row that falls within the filtered set of rows. Expression filters may be combined with ranges, master-detail links, and/or code-based filters to further filter the rows in the table or query result set.

ElevateDB's expression filters use the same naming conventions, operators, and functions as its SQL implementation of WHERE conditions. The only differences are as follows:

Difference	Description
Correlation Names	You cannot use table or column correlation names in filter expressions.
Query expressions	You cannot use query expressions in filter expressions.
Wildcards	You can additionally use the asterisk (*) wildcard character with the equality operator (=) or inequality operator (<>) in order to perform partial-length comparisons. However, this only works when the foNoPartialCompare element is not included in the FilterOptions property.

Please see the Identifiers, Types and Operators, Numeric Functions, String Functions, Date/Time Functions, Interval Functions, and Conversion Functions topics for more information.

The following example shows how to set an expression filter where the LastSaleDate column is between January 1, 1998 and December 31, 1998 and the TotalSales column is greater than 10,000 dollars:

```
begin
   with MyTable do
      begin
      { Set the filter expression  }
      Filter:='(LastSaleDate >= DATE '+Engine.QuotedSQLStr('1998-01-01')+')
      '+
              'and (LastSaleDate <= DATE
      '+Engine.QuotedSQLStr('1998-12-31')+') '+
              'and (TotalSales > 10000)';
      FilterOptions:=[];
      Filtered:=True;
      end;
end;
```

ElevateDB attempts to optimize all expression filters, and the filter optimization process is the same as that used for optimizing SQL WHERE conditions. Please see the Optimizer topic for more information.

Setting a Code-Based Filter

The OnFilterRow event and the Filtered property are used together to set a code-based filter. The steps to set a code-based filter include assigning an event handler to the OnFilterRow event and then making the code-based filter active by setting the Filtered property to True. You can turn off or disable a code-based filter by setting the Filtered property to False. If the current row pointer does not fall into the conditions specified within the code-based filter, then the current row pointer will be moved to the nearest row that falls within the filtered set of rows.

The following example shows how to write a code-based filter event handler where the CustomerName column contains the word "Hardware" (case-sensitive):

```
procedure TMyForm.TableFilterRow(DataSet: TDataSet;
   var Accept: Boolean);
begin
   Accept:=False;
   if Pos('Hardware',
          DataSet.ColumnByName('CustomerName').AsString) > 0) then
      Accept:=True;
end;
```

Code-based filters implemented via an OnFilterRow event handler are always completely un-optimized. However, ElevateDB only incrementally calls the OnFilterRow event handler for the row or rows necessary for any data-aware controls or for positioning on a desired row (if data-aware controls are not being used). For example, if you positioned a table with an active code-based filter on a new row using the Locate method, then ElevateDB will call the OnFilterRow event handler for the current row and any subsequent rows using the active index order until it has found a row that satisfies the event handler (Accept=True). ElevateDB then stops and does not attempt to filter any further rows. The OnFilterRow event handler can, therefore, be used to filter large numbers of rows incrementally without a large amount of overhead.

5.25 Using Streams with Tables, Views and Query Result Sets

Loading and saving tables, views, and query result sets to and from streams is accomplished through the LoadFromStream and SaveToStream methods of the TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components. A stream is any TStream-descendant object such as TFileStream, TMemoryStream, or even the ElevateDB TEDBBlobStream object used for reading and writing to BLOB columns. Loading a stream copies the entire contents of a stream to an existing table, view, or query result set. When loading a stream, the contents of the stream must have been created using the SaveToStream method or else an EEDBError exception will be raised. The error code given when a load from a stream fails because of an invalid stream is 1003 (EDB_ERROR_STREAM). Saving to a stream copies the contents of a table, view, or query result set to the stream, overwriting the entire contents of the stream. The rows that are copied can be controlled by setting a range or filter on the source table or query result set prior to calling the SaveToStream method. Please see the Setting Ranges on Tables and Setting Filters on Tables and Query Result Sets topics for more information.

Loading Data from a Stream

To load data from a stream into an existing table, view, or query result set, you must open the TEDBTable, TEDBQuery, or TEDBStoredProc component and then call the LoadFromStream method.

The following example shows how to load data from a memory stream (assumed to already be created) into a table using the LoadFromStream method:

```
begin
   with MyTable do
      begin
      DatabaseName:='SalesDB';
      TableName:='customer';
      Open;
      LoadFromStream(MyMemoryStream);
      end;
end;
```

Note
Tables, views, or query result sets in remote sessions can load data from a local (client-side) stream. However, since the stream contents are sent as one buffer to the ElevateDB Server as part of the load request, it is recommended that you do not load particularly large streams since you will run the risk of exceeding the available memory on the local workstation or ElevateDB Server.

Saving Data to a Stream

To save the data from a table, view, or query result set to a stream, you must open the TEDBTable, TEDBQuery, or TEDBStoredProc component and then call the SaveToStream method.

The following example shows how to save the data from a table to a memory stream (assumed to already be created) using the SaveToStream method of the TEDBTable component:

```
begin
```

```
   with MyTable do
      begin
      DatabaseName:='SalesDB';
      TableName:='customer';
      Open;
      SaveToStream(MyMemoryStream);
      end;
end;
```

Note
When the SaveToStream method is called, the existing position of the stream pointer in the destination stream is not moved, and the size of the destination stream is not changed except in the case where the size must be expanded to accomodate the new stream data being saved from the table, view, or query result set. Therefore, if you wish to overwrite any existing data in the destination stream during the SaveToStream method call, you should use the following code on the stream before calling the SaveToStream method:

```
begin
   with MyStream do
      begin
      Size:=0;
      Position:=0;
      end;
end;
```

The reason for this behavior is that it allows the developer the possibility of combining multiple streams from multiple tables, views, or query result sets into one stream.

5.26 Cached Updates

Using cached updates with tables, views, and query result sets is accomplished through the BeginCachedUpdates, and ApplyCachedUpdates, and CancelCachedUpdates methods of the TEDBTable, TEDBQuery, TEDBScript, and TEDBStoredProc components. In addition, the CachingUpdates property can be used to find out when cached updates are in effect for a dataset.

Using cached updates permits an application to copy all existing rows in a given table, view, or query result set to a temporary table that is then used for any inserts, updates, or deletes. Once all updates are complete, the application may then call the ApplyCachedUpdates method to apply all updates to the source table or query result set, or the CancelCachedUpdates method to cancel all updates and revert the table or query result set to its original state prior to the cached updates. The rows that are included in the cached updates can be controlled by setting a range or filter on the source table or query result set prior to calling the BeginCachedUpdates method. Please see the Setting Ranges on Tables and Setting Filters on Tables,Views, and Query Result Sets topics for more information.

Warning
Do not use cached updates on very large tables or query result sets with large number of rows in the active set according to any active ranges and/or filters. Doing so can result in some serious performance problems as the entire set of rows will need to be copied when cached updates are begun.

Beginning Cached Updates

To begin cached updates, you must call the BeginCachedUpdates method. When using either a TEDBTable, TEDBQuery, TEDBStoredProc, or TEDBScript component, the table, view, or query result set must be opened (Active property is set to True) or an exception will be raised.

Note
Cached updates require that a primary key be defined for the underlying table that is being updated or else an EEDBError exception will be raised. The error code given when a BeginCachedUpdates call fails due to a missing primary key is 1307 (EDB_ERROR_CACHEUPDATES).

Applying Cached Updates

To apply any cached updates to the source table, view, or query result set, you must call the ApplyCachedUpdates method. This method will apply any updates that were made to the temporary table used for the cached updates to the source table, view, or query result set. Only rows that were inserted, updated, or deleted are processed, so the result is the same as calling the CancelCachedUpdates method if no rows were inserted, updated, or deleted while cached updates were enabled. You can examine the CachingUpdates property to determine whether cached udpdates are in effect before trying to apply any cached updates.

A transaction is not required around the ApplyCachedUpdates method call in order to make it atomic. The ApplyCachedUpdates method is always executed as an atomic unit of work.

Reconciling Errors

Cached updates are handled in an optimistic manner, which means that ElevateDB does not hold any locks on the rows that are held in the cache while the cached updates are in effect. Subsequently, it is possible that another session has changed some or all of the rows that were cached and updated or deleted in the cache. When the cached updates are then applied using the ApplyCachedUpdates method, an error message will be raised and it is possible that only a portion of the cached updates will be applied to the source table, view, or query result set. To avoid this, you can define an ERROR trigger on the underlying table being updated. For more information on ERROR triggers, please see the CREATE TRIGGER topic in the ElevateDB SQL Manual.

Note
Calling the LOADINGUPDATES function during an ERROR trigger will return TRUE during the execution of the ApplyCachedUpdates call. This is because the cached updates functionality uses the ElevateDB replication manager for their implementation.

Filters, Ranges, and Master-Detail Links

Most of the operations that can be performed on a TEDBTable, TEDBQuery, TEDBScript, or TEDBStoredProc component behave the same regardless of whether cached updates are active or not. This includes the following operations:

Navigating Tables, Views, and Query Result Sets
Searching and Sorting Tables, Views, and Query Result Sets
Inserting, Updating, and Deleting Rows

However, certain states of the table, view, or query result set are not carried over to the cached updates temporary table. These include:

Filters
Ranges
Master-Detail Links

All of these states are reset for the cached updates temporary table. You may apply new filters, ranges, and/or master-detail links on the cached updates temporary table if you wish, but they will not apply to the base table nor will they affect the base table's state with respect to filters, ranges, or master-detail links. After the cached updates are applied or cancelled, all of these states are set back to what they were prior to the cached updates being active.

Refreshing During Cached Updates

If you call the TEDBTable, TEDBQuery, TEDBStoredProc, or TEDBScript Refresh method while cached updates are active, then the current contents of the cached updates temporary table will be discarded and replaced with the latest data from the base table. Cached updates will remain in effect after the Refresh is complete.

Appendix A - Error Codes and Messages

The following is a table of the error codes and messages for ElevateDB. ElevateDB uses the exception object to raise exceptions when an error occurs.

> **Note**
> This list only covers the exceptions raised by ElevateDB itself and does not cover the general exceptions raised by the component units.

If you wish to use the error constants defined by ElevateDB in your applications you need to make sure:

For Delphi applications, that the edberror unit file is included in your uses clause for the source unit in question

For C++Builder applications, that the edberror header file is included in your .h header file for the source file in question

If you wish to change the following error messages or translate them into a different language, you may do so by altering the contents of the edbconsts unit that can be found in the same directory where the other ElevateDB units were installed.

For more information on exception handling in your application, please see the Exception Handling and Errors topic in this 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 object. This is usually due to another session already having an incompatible lock on the object already. Please see the Locking topic for more information.
EDB_ERROR_UNLOCK (301)	Cannot unlock <ObjectType> <ObjectName> for <AccessType> accessThis error is raised whenever ElevateDB cannot unlock a given 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)	Transaction error (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 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 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_ROWCORRUPTED (1012)	The row has been corrupted 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 buffer being used for the operation has been corrupted. This is typically due to improper multi-threaded access to the ElevateDB client engine.
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 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_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.

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 or 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.