
S60 Platform: Using DBMS APIs

Version 2.0
July 3, 2006

S60 platform

Legal notice

Copyright © 2004–2006 Nokia Corporation. All rights reserved.

Nokia and Forum Nokia are registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1.	Introduction	6
2.	DBMS structure and elements	7
2.1	Permanent file stores and streams	7
2.2	Create a database.....	8
2.3	Define tables within a database	10
2.3.1	Columns.....	11
2.3.2	Column sets.....	11
2.3.3	Index keys	12
2.4	Query schema information	12
2.5	Open and close a database	14
2.6	Create data	16
2.6.1	Long column data in a database	17
3.	Using RowSets and cursors.....	19
3.1	RowSet definition	19
3.1.1	RowSet base	19
3.1.2	Table RowSets	19
3.1.3	SQL views.....	20
3.2	Cursors.....	20
3.2.1	Cursor states	21
3.3	Retrieving data by index.....	21
4.	Using SQL with DBMS	23
4.1	SQL and the native C++ API.....	23
4.2	Breakdown of a Select statement	23
4.3	Supported SQL subset.....	25
4.4	Using TDbQuery class	26
4.5	SQL schema and data updates	27
4.5.1	Database Definition Language (DDL)	27
4.5.2	SQL data update statements (DML).....	28
4.5.3	Examples	28
5.	Incremental database operations	30
5.1	Overview	30
5.2	Database incremental operations — RDbIncremental	30
5.3	DML statement incremental execution — RDbUpdate	31
5.3.1	DML synchronous.....	31

5.3.2	DML asynchronous	31
6.	Sharing databases	33
6.1	DBMS server session.....	33
6.2	Database change notifier	33
6.3	Transactions and locks	33
7.	Summary	35
8.	Terms and abbreviations	36
9.	References	37
10.	Evaluate this resource	38

Change history

March 15, 2004	Version 1.0	Document first released
July 3, 2006	Version 2.0	Updated for S60 3rd Edition

1. Introduction

The purpose of this document is to demonstrate how to use the relational database APIs available on Symbian OS. Collectively, this functionality is known as the Database Management System (DBMS) APIs.

Symbian OS DBMS provides features for creating and maintaining databases, and implements reliable and secure data access to these databases via both native and SQL calls. These calls are supported by a transaction/rollback mechanism that ensures that either all data is written or none at all.

The document provides code snippets from [S60 Platform: DBMS Example](#) [1] to demonstrate key techniques by creating and manipulating a simple database of books. The code snippets embedded throughout the document are predominantly gathered or adapted from this example.

Although the document focuses specifically on describing the support of DBMS APIs in the S60 platform, it is mostly applicable to other Symbian OS devices too.

2. DBMS structure and elements

Symbian OS DBMS provides a functional interface to a relational database. Any Symbian OS DBMS is represented by a hierarchy of elements — at the base level is the file store that contains the database, which itself is the container for tables. Tables are containers for rows and are defined by columns. Columns are constituents of rows defined by attributes of data type and length. DBMS supports a rich set of data types.

For developers used to working in a non-DBMS database environment such as ISAM, rows can be thought of as analogous to records, and columns as analogous to fields. However, it should be pointed out that within DBMS, rows and columns typically have other attributes that are not normally associated with records and fields.

Symbian OS DBMS is a powerful, if lightweight, implementation that supports normal add/search/retrieve/update/delete functionality as well as basic Structured Query Language (SQL), Data Definition Language (DDL), and Data Modeling Language (DML) statement handling.

More advanced Relational Database Management System (RDBMS) functions, such as SQL Joins and Trigger events, are not available.

Though lightweight, Symbian OS DBMS provides adequate capabilities for the mobile user and supports database sizes limited only by available resources.

2.1 Permanent file stores and streams

Symbian OS DBMS relies on features of the File Server, Permanent File Stores, and Streams implementations of Symbian OS to provide underlying storage functionality. In Symbian OS, file stores and streams are very closely related.

Note that this document will not dwell on these features except where they are relevant to DBMS. Detailed information on the File Server, File Stores, and Streams APIs can be found in the S60 SDK documentation and on the Symbian Web site.

In general terms, a file store is literally a file on “disk,” that is, a file store will have a name visible in a directory listing. However, depending on the implementation used to create it, neither its content nor its purpose may be immediately apparent in such a view.

Streams are object representations internalized or externalized to and from a store. A store may therefore be thought of as simply a collection of streams.

A permanent file store implements a store type where streams within can be changed individually (rather than all at once as occurs with other store types), and the object network comprising the application’s data is structured so that sections of data can be loaded from, and written to, the store if modified. RAM is used as the change media.

The permanent file store is provided by `CPermanentFileStore` and is a concrete type, derived from `CFileStore`. These classes are defined in the File Store API and are the building blocks for DBMS.

This form of store implements most of the operations defined by the store abstract framework. Most importantly, streams in a permanent file store can be:

- Created in advance of being written to
- Overwritten
- Replaced
- Deleted

2.2 Create a database

There are two APIs for creating a database:

- `RDbStoreDatabase` provides an interface to create and open a database exclusively, that is, database access is not shared. Database operations are performed directly to a file; thus this method can be called *client-side* access.
- `RDbNamedDatabase` provides an interface for creating and opening a database identified by name and format. The class allows both *client-side* (exclusive) and shared *client/server* database access.

Regardless of the API choice, the database is created into a file. In the creation phase, both of the APIs open the database in exclusive mode using RFS file server session. If the database needs to be shared, it must be created with `RDbNamedDatabase` and closed, and finally be opened with `RDbNamedDatabase` using database server session `RDBs`.

`RDbNamedDatabase` provides the preferred means to work with databases.

`RDbStoreDatabase` can be used if the database must be part of a file (for example, a general configuration file also having other streams) and the stream ID for the database has to be other than the file root. Databases created with `RDbStoreDatabase` can be opened with `RDbNamedDatabase` and vice versa, as long as the root stream ID of the file store points to the database. However, the client/server mechanism is not supported.

Given this, the APIs allow:

- Creation of the database structure (table creation)
- Creation of index(es)

When these steps are complete, the database is ready to accept data. Symbian OS DBMS ensures that once in operation:

- When an entry is created, it is added to the database.
- An internal copy of the indexes in the database file is maintained.
- When an entry is selected for displaying or editing, its data is retrieved from the database and all such entries' data is maintained in an internal, nonpersistent form.
- When an entry is edited and saved, that entry is replaced in the database.
- When the application exits, there will be nothing to do as there can be no data unsaved except, possibly, the current entry if it was being edited.

A database application rarely replaces the entire store. The database file itself is permanent and is not overwritten when an entry is saved. Because of this, care is needed when maintaining this type of store; all links within the database must be maintained without corruption; and no stream subnetworks in the database should become inaccessible, otherwise space is wasted.

Overwriting an existing stream does not change its length. However, attempting to write beyond the end of the stream fails and results in a leave. Replacing a stream can result in a stream of any size. The order in which streams are written to a permanent file store is unimportant because streams can be changed and rewritten. To avoid stream problems, `RDbNamedDatabase` should be used.

Creating a database using `RDbStoreDatabase`

The following code fragment illustrates the creation of a database using `RDbStoreDatabase`. It creates a File Store object and constructs a database within it.

```
class CBookDb : public CBase
{
...
private: // Member data

    RFs          iFsSession;
    RDbStoreDatabase iBookDb;
    CFileStore*   iFileStore;
...
};

TInt CBookDb::CreateDb(const TFileName& aNewBookFile)
{
    Close();

    // Create empty database file.
    TRAPD(error,
        iFileStore = CPermanentFileStore::ReplaceL(iFsSession,
            aNewBookFile, EFileRead|EFileWrite);
        iFileStore->SetTypeL(iFileStore->Layout()); // Set file store type
        TStreamId id = iBookDb.CreateL(iFileStore); // Create stream object
        iFileStore->SetRootL(id); // Keep database ID as root of store
        iFileStore->CommitL(); // Complete creation by committing
        // Create Book tables and indexes
        CreateBooksTableL();
        CreateBooksIndexL();
    );
}
```

`CreateDb` creates a new database file. It takes `TFileName`, which is the full file name (including path). The example overwrites any previous file of the same name (`ReplaceL()`). Finally, it creates the root stream object and commits the database structure.

Creating a database using RDbNamedDatabase

By using `RDbNamedDatabase`, creating a database is simpler because there is no need to work with streams:

```

...
RFs          iFsSession;
RDbNamedDatabase iBookDb;
...

TInt CBookDb::CreateDb(const TFileName& aNewBookFile)
{
    Close();
    TInt error=iBookDb.Replace(iFsSession, aNewBookFile);
    if(error!=KErrNone)
    {
        return error;
    }
    ...// Database is now open. Create tables etc.
}

```

The `Replace` method creates a new database file or replaces an existing one. It also formats the file as an empty database. The database is then open and ready for creating table structures, and so on. Note that by creating a database, the access mode is exclusive. `Replace` takes in a `TFileName`, which is a full file name (including path).

In Symbian OS DBMS, DBMS name length is limited to 64 characters. The database names correspond to file names. It is up to the programmer to decide whether to use a file extension. As an example, the following names are valid:

```

_LIT(KDbName, "C:\\system\\apps\\DBMS\\DBMS.dat");
_LIT(KDbName, "C:\\system\\apps\\DBMS\\DBMS");

```

The valid paths and names in S60 3rd Edition are:

Application private path:

```

_LIT(KDbName, "C:\\Private\\<SID>\\DBMS.dat");
_LIT(KDbName, "C:\\Private\\<SID>\\DBMS");

```

Public path (e.g.):

```

_LIT(KDbName, "C:\\Data\\<Some_path_element>\\DBMS.dat");
_LIT(KDbName, "C:\\Data\\<Some_path_element>\\DBMS");

```

2.3 Define tables within a database

To define tables in a database, the developer needs to be aware of the three key API concepts: column, column set, and index key.

Table names must be unique within DBMS, and column names must be unique within the table to which they belong. Table and column names are case-insensitive.

2.3.1 Columns

A table in a database is defined by a set of columns. Each column has attributes such as name, data type, and maximum length if the data type is text or binary.

A column definition is encapsulated in `TDbCol`.

2.3.2 Column sets

A set of columns that describe a table are encapsulated in `CDbColSet`.

`TDbColSetter()` can be used to iterate through the set.

The following code fragment illustrates creation of a database table in the permanent file store created in Section 2.1, "Permanent file stores and streams."

```
...
const int KTitleMaxLength = 60;
_LIT(KBooksTable, "Books");
_LIT(KBooksAuthorCol, "Author");
_LIT(KBooksTitleCol, "Title");
_LIT(KBooksDescriptionCol, "Description");
...
// Specify columns for Books table
TDbCol authorCol(KBooksAuthorCol, EDbColText); // Default length
TDbCol titleCol(KBooksTitleCol, EDbColText, KTitleMaxLength);
titleCol.iAttributes = TDbCol::ENotNull;
// Stream Data
TDbCol descriptionCol(KBooksDescriptionCol, EDbColLongText);

// Create columnset
CDbColSet* bookColSet = CDbColSet::NewLC();
bookColSet->AddL(authorCol);
bookColSet->AddL(titleCol);
bookColSet->AddL(descriptionCol);

// Create the Books table
User::LeaveIfError(iBookDb.CreateTable(KBooksTable,
    *bookColSet));
CleanupStack::PopAndDestroy(bookColSet);
...
```

First, column name literals are defined via the `_LIT` macro. Then columns are defined, the `bookColSet` object is initialized, and the columns are added to the set. The example provides both a default and an overridden size allocation for the text fields. The default length is 50.

For the `titleCol` column, the member variable `iAttributes` is assigned the enumeration value `ENotNull`, enforcing that a row cannot be created when this column value is unset.

Finally, because `bookColSet` was added to the cleanup stack when created, it must be removed before exiting this code section; this is accomplished by calling `PopAndDestroy(bookColSet)` on the cleanup stack.

2.3.3 Index keys

An index key orders one or more table columns. Each key has attributes such as being unique or primary, a comparison specification for text columns, and a list of columns that make up the key. If no index keys are defined, rows are retrieved in an arbitrary order.

The index key is encapsulated in `CDbKey`. A column for the key is encapsulated in `TDbKeyCol`.

The following code fragment illustrates creation of a two-column index on the table created in Section 2.3.2, "Column sets."

```
...
_LIT(KBooksTable, "Books");
_LIT(KBooksAuthorCol, "Author");
_LIT(KBooksTitleCol, "Title");
_LIT(KBooksIndexName, "BooksIndex");

...
// Create index consisting of two columns
TDbKeyCol authorCol(KBooksAuthorCol);
TDbKeyCol titleCol(KBooksTitleCol);

CDbKey* index = CDbKey::NewLC(); // create index key set
index->AddL(titleCol);
index->AddL(authorCol);
User::LeaveIfError(iBookDb.CreateIndex(
    KBooksIndexName, KBooksTable, *index));
CleanupStack::PopAndDestroy(index);
```

First, the columns for the index are defined. Then the `CDbKey` index is initialized and the columns are added to it. The index is created to the database by a call to `CreateIndex`.

Finally, because the `index` was added to the cleanup stack when created, it is removed by calling `PopAndDestroy(index)` on the cleanup stack.

2.4 Query schema information

There are APIs to query database structure (indexes, tables, and their structures). The database base class, `RdbDatabase`, provides the base means:

- `TableNamesL()`: List of table names encapsulated in `CDbTableNames`.
- `IndexNamesL()`: List of index names encapsulated in `CDbIndexNames`. The method takes in a table name.
- `ColSetL()`: Column definitions for a table encapsulated in `CDbColSet`. The method takes in a table name.
- `KeyL()`: Index definition for an index within a table. The method takes in a table name and an index name.

The column definitions `CDbColSet` for a table can also be queried from a table or an SQL view (see Section 3.1, “RowSet definition”) using their base class method `RdbRowSet::ColSetL()`. The `CdbColSet` can be iterated using its methods or utilizing the `TdbColSetIter` class.

The following example code retrieves column names and sizes for the Books table. The results are packaged to `CdesCArrayFlat`.

```

...
private: // Member data
    RDbStoreDatabase iBookDb;
...

// Items in the array are in format <column_name>: <column_size>
CDesCArrayFlat* CBookDb::ColumnNamesAndSizesL()
{
    RDbTable booksTable;
    TBuf<KDbMaxColName> columnNameAndSize;
    _LIT(KDelimiter, ": ");
    _LIT(KNoSize, "No size");

    // Open the Books table.
    User::LeaveIfError(
        booksTable.Open(iBookDb, KBooksTable,
            booksTable.EReadOnly));
    CleanupClosePushL(booksTable); // Remember to pop and close

    CDesCArrayFlat* resultArray =
        new (ELeave) CDesC16ArrayFlat(KArrayGranularity);
    CleanupStack::PushL(resultArray);

    // Iterate through the columns of Books table. Extract the
    // column name and column size (size only for text columns).
    CDbColSet* colSet = booksTable.ColSetL();
    CleanupStack::PushL(colSet);
    TDbColSetIter colIter(*colSet);
    while(colIter)
    {
        columnNameAndSize.Zero();
        columnNameAndSize.Append(colIter->iName);
        columnNameAndSize.Append(KDelimiter);
        if(colIter->iType == EDbColText)
            columnNameAndSize.AppendNum(colIter->iMaxLength);
        else
            columnNameAndSize.Append(KNoSize);
        resultArray->AppendL(columnNameAndSize);
        colIter++;
    }
    CleanupStack::PopAndDestroy(colSet);
    CleanupStack::Pop(resultArray);

    // Pop the booksTable from cleanup stack and close it.
    CleanupStack::PopAndDestroy();
    return resultArray;
}

```

2.5 Open and close a database

The database created in Section 2.2, “Create a database,” is open and ready for database operations. Also, it is often needed to open an existing database.

As discussed in Section 2.2, the database can be opened in an exclusive *client-side* mode and in a shared *client/server* mode. `RDbNamedDatabase` supports both modes, but the `RDbStoreDatabase` API provides the means to open a database in the *client-side* mode only.

It is recommended to use the `RDbNamedDatabase` API. The *client-side* access mode is slightly more efficient, but the database is not accessible to other applications because the file is locked. In a typical application, the *client/server* mode is recommended.

Client/server mode

The following code snippet illustrates opening and closing a database in *client/server* mode. The implementation requires `RDbNamedDatabase` and database server session `RDbs`.

```
...
private: // Member data

    RDbs iDbSession;
    RDbNamedDatabase iBookDb;
...

void CBookDb::OpenDbInClientServerModeL(const TFileName&
    aExistingBookFile)
{
    User::LeaveIfError(iDbSession.Connect());
    User::LeaveIfError(iBookDb.Open(iDbSession,
        aExistingBookFile));
}

... // Use the open database

void CBookDb::CloseDb()
{
    iBookDb.Close(); // note the order of closing.
    iDbSession.Close();
}
```

The `Open` method of `RDbNamedDatabase` opens the database. It takes in an open (connected) database server session `RDbs`. Both the database and the session are closed with the `Close()` method.

To ensure clean `CbookDb` destruction, the destructor should call `CloseDb`; typically the user uses `CloseDb`, but if, for example, `OpenDbInClientServerModeL` leaves while opening the database (`iBookDb.Open`), `RDbs` might not be closed.

Client-side mode

The following code snippet illustrates opening and closing a database in *client-side* mode using the `RDbNamedDatabase` API. The code is similar to that of *client/server*, except that a file server session `RFs` is used instead of the database server session `RDb`s.

```

...
private: // Member data

    RFs iFsSession;
    RDbNamedDatabase iBookDb;
...

void CBookDb::OpenDbInClientSideModeL(const TFileName&
    aExistingBookFile)
{
    User::LeaveIfError(iFsSession.Connect());
    User::LeaveIfError(iBookDb.Open(iFsSession,
        aExistingBookFile));
}

... // Use the open database

void CBookDb::CloseDb()
{
    iBookDb.Close(); // note the order of closing.
    iFsSession.Close();
}

```

If `RDbStoreDatabase` is used, the streams and sessions must be handled correctly:

```

...
private: // Member data

    RFs iFsSession;
    RDbStoreDatabase iBooksDb;
    CFileStore* iFileStore;
...

TInt CBookDb::OpenDb(const TFileName&
    aExistingBookFile)
{
    Close();

    if(!BaflUtils::FileExists(iFsSession, aExistingBookFile))
    {
        return KErrNotFound;
    }

    TRAPD(error,
        iFileStore = CPermanentFileStore::OpenL(iFsSession,
            aExistingBookFile, EFileRead|EFileWrite);
        // Set file store type
        iFileStore->SetTypeL(iFileStore->Layout());
        iBookDb.OpenL(iFileStore, iFileStore->Root())

```

```

        );
        if(error!=KErrNone)
        {
            return error;
        }

        iOpen = ETrue;
        return KErrNone;
    }

void CBookDb::Close()
{
    iBookDb.Close();
    if(iFileStore)
    {
        delete iFileStore; // Closes the file too
        iFileStore = NULL;
    }
    iFsSession.Close();
}

```

The `OpenL` method of `RDbStoreDatabase` opens a database. It takes in an open File Store object and a stream ID, which can be queried from the File Store by the `Root()` method. When the database is closed, the objects are cleaned in reverse order.

2.6 Create data

Once the database is open, data can be maintained in the database. Note that it is not necessary to define indexes to be able to add or retrieve data.

The following code fragment illustrates the table method (see Section 3.1.2, “Table RowSets”) for adding data to the table created in Section 2.3, “Define tables within a database.” The example assumes that the database is open.

```

...
_LIT(KBooksTable, "Books");
_LIT(KBooksAuthorCol, "Author");
_LIT(KBooksTitleCol, "Title");
_LIT(KBooksDescriptionCol, "Description");

...
private: // Member data

    RDbStoreDatabase iBookDb;
...

void CBookDb::AddBookWithCppApiL(const TDesC& aAuthor,
                                const TDesC& aTitle,
                                const TDesC& aDescription)
{
    // Create an updateable database table object.
    // Assume the database is open.
    RDbTable table;
    User::LeaveIfError(table.Open(iBookDb,
                                KBooksTable, table.EUpdateable));
}

```

```

CleanupClosePushL(table); // Remember to pop and close

// Construct CdbColSet. Use it to query column numbers.
CdbColSet* booksColSet = table.ColSetL();
CleanupStack::PushL(booksColSet);

table.Reset();
table.InsertL(); // Insert empty row
// Set the author and title for the new row
table.SetColL(booksColSet->ColNo(KBooksAuthorCol), aAuthor);
table.SetColL(booksColSet->ColNo(KBooksTitleCol), aTitle);

// Set the description. Use a stream for the long column
RDbColWriteStream writeStream;
writeStream.OpenLC(table,
    booksColSet->ColNo(KBooksDescriptionCol));
writeStream.WriteL(aDescription);
writeStream.Close();
CleanupStack::Pop(); // writeStream

CleanupStack::PopAndDestroy(booksColSet);

table.PutL(); // Complete insertion
CleanupStack::Pop() // table
table.Close();
}

```

The Books table is first opened in an updateable mode. `booksColSet` is initialized for use in finding column numbers for the Author, Title, and Description columns.

Adding a new row to the database involves first inserting an empty row, updating values for the row, and finally completing insertion to the database. `table.InsertL()` inserts an empty row and `table.SetCol(...)` sets the author and title. The description column is long and requires use of `RDbColWriteStream`. For more information on long textual columns, refer to Section 2.6.1, “Long column data in a database.” The creation is completed with `PutL()`.

Finally, the table is closed.

2.6.1 Long column data in a database

Accessing text data within an `EDbColLongText` DBMS column is achieved through the use of `RDbColReadStream`; similarly, `RDbColWriteStream` is used to set the contents of this column type within `RowSet`.

Symbian OS DBMS supports only one column at a time in a `RowSet` object open for reading as a stream. While it is open, no column in the same `RowSet` object may be set using `RDbColWriteStream`.

The following is an example of stream function code fragments to retrieve the contents of a long column in a database:

```

...
_LIT(KBooksTable, "Books");
_LIT(KBooksDescriptionCol, "Description");

```

```

...
private: // Member data

    RDbStoreDatabase iBookDb;
...

    TBuf<128> description; // read result placed here
    RDbTable table;
    User::LeaveIfError(
        table.Open(iBookDb, KBooksTable, table.EReadOnly));
    table.Reset();

    // Find the column number for 'Description' column
    CDbColSet* colSet = table.ColSetL();
    TDbColNo descrColNo = colSet->ColNo(KBooksDescriptionCol);
    delete colSet;
    table.FirstL(); // Set cursor to first row. Should check
                   // this succeeds.
    table.GetL(); // Get the first row for operation
    // Read all characters. Assume there are no more
    // than 128 characters.
    RDbColReadStream readStream;
    readStream.OpenLC(table, descrColNo);
    readStream.ReadL(description, table.ColLength(descrColNo));
    readStream.Close();
    CleanupStack::Pop(); //readStream

    CleanupStack::PopAndDestroy(colSet);
    table.Close();
...

```

Long column data is added or updated similarly using `RDbColWriteStream` (complete code for inserting a row with a long column is given in Section 2.6, "Create data"):

```

    RDbTable table;
    // Open table in updatable mode
    User::LeaveIfError(
        table.Open(iBookDb, KBooksTable, table.EUpdatable));

    CDbColSet* booksColSet = table.ColSetL();
    CleanupStack::PushL(booksColSet);

    ... // Find or create a row here and retrieve it for operation

    // Use a stream to read data from the long column
    RDbColWriteStream writeStream;
    writeStream.OpenLC(table,
        booksColSet->ColNo(KBooksDescriptionCol));
    writeStream.WriteL(aDescription);
    writeStream.Close();
    CleanupStack::Pop(); // writeStream
    ...

    CleanupStack::PopAndDestroy(booksColSet);
    table.Close();

```

3. Using RowSets and cursors

RowSets within Symbian OS DBMS allow database data to be searched, retrieved, and modified. It is important to remember that, being nonpersistent, the data contained by a RowSet is not the data itself — that is, an element of a RowSet will resolve an index to a database row that actually holds the data.

3.1 RowSet definition

The API has three key concepts:

- *RowSet base* (`RDbRowSet`) — Abstract base class that provides navigation, row retrieval, and updates to data. The data source is specified by the two concrete implementations:
- *Table RowSet* (`RDbTable`) — Provides full view of a table.
- *SQL view* (`RDbView`) — Provides a view to a table. The view depends on the SQL query used for building the view. It may contain only part of the data and only part of the columns of the table.

3.1.1 RowSet base

The following functionality is provided by RowSet implementation in Symbian OS DBMS:

- Row finding and matching
- Retrieving data from a database
- Updating or inserting rows
- Navigating the RowSet with a cursor
- Getting the schema of the rows within the RowSet
- Extracting and setting columns in a row

A base class is defined for all RowSet types by `RDbRowSet`. Concrete types of RowSet derive from this class.

To access data, database application programs use a RowSet that maintains a cursor on a current row within that set. The RowSet provides an abstract interface and two concrete types are provided: table RowSets and SQL views.

3.1.2 Table RowSets

Table RowSet provides all of the rows and columns of a table as a RowSet. Indexes can be used to order the RowSet and to provide fast, key-based row retrieval from the table. If indexes are not used, row ordering is undefined. Table RowSets are encapsulated in `RDbTable`.

Table RowSet is useful when the full table structure must be viewed or all the table data is handled in a certain order. It is preferred over SQL views if an individual, uniquely

identifiable row must be found quickly. (See Section 3.3, “Retrieving data by index,” for an explanation of how indexes are used to achieve this.) However, in most cases, SQL views are preferred.

3.1.3 SQL views

`RDbView` provides a `RowSet` based on an SQL query. An SQL query is encapsulated in `TDbQuery`.

An SQL view is a `RowSet` generated by an SQL query on a database and is currently restricted to a single table. The `Prepare` method of `RDbView` parses the SQL and specifies how the data is evaluated. By default, cursor navigation is implemented by evaluating the view as necessary.

Since evaluation may be time-consuming, a pre-evaluation window `TDbWindow` may be specified in the prepare phase, allowing the `RowSet` to be evaluated once and navigated quickly. Full evaluation may take a long time, and may be best done in steps.

For large `RowSets`, it can be useful to define the balance between memory usage (greatest if the view stores the complete `RowSet`) and speed (slowest if the `RowSet` is evaluated for each cursor navigation). This can be achieved by defining a limited (or partial) pre-evaluation window with a preferred size. A partial evaluation window can also be used to get a partial view to SQL result set for local navigation.

The following code constructs a view using an SQL select:

```
...
private: // Member data
    RDbStoreDatabase iBookDb;
...

_LIT(KViewSql, "SELECT Author, Title FROM Books")
RDbView view;
User::LeaveIfError(
    view.Prepare(iBookDb, TDbQuery(KViewSql),
        view.EReadOnly));
CleanupClosePushL(view);
User::LeaveIfError(view.EvaluateAll());
for (view.FirstL(); view.AtRow(); view.NextL())
{
    view.GetL(); // Fetch a cached copy of current row
    ... // perform operations for the row
}
CleanupStack::PopAndDestroy(); // This also closes the view
```

SQL use will be covered in more detail in Chapter 4, “Using SQL with DBMS.”

3.2 Cursors

A cursor is used to navigate all available rows in the `RowSet`. The available rows depend on how the `RowSet` is generated and, particularly, whether the `RowSet` is an SQL view using an evaluation window.

3.2.1 Cursor states

The most common state of a cursor is maintaining a value that yields a row from a RowSet. However, there are other states that a cursor may maintain.

A cursor can maintain "beginning" and "end" states. Initially, and also following a reset, the cursor is positioned at the beginning. The beginning lies immediately prior to the first row, and the end immediately follows the last row. Navigating to the first row is equivalent to navigating to the next row from the beginning, which, in an empty set, also takes the cursor to the end of the set. A similar process applies to navigating from the last row of a set.

When updating or inserting a row, the cursor has a state that prevents navigation until the update or insertion has been completed.

After deletion of a current row, the cursor is unchanged and maintains the now-invalid value referring to the "hole" left by the deletion. Attempting to access this deleted row with `GetL()` will cause the function to leave. Navigating to the next, previous, or a specific row will move the cursor to either the requested row or the beginning/end if there are insufficient rows, and the deleted RowSet value will be removed from the RowSet.

A cursor can become invalid after an error during navigation, due to failure in the store or file system, or due to loss of context when multiple RowSets are concurrently updating a table. Navigating to any fixed location — not next or previous — will restore the cursor.

3.3 Retrieving data by index

Retrieving data by an index can be accomplished implicitly through the use of an SQL view or through the use of an `RDbTable` RowSet object.

`RdbTable` provides a simple `SeekL` method to find a row quickly. It uses `TDbSeekKey` and can be used to read the first occurrence only. It is useful for retrieving a row quickly with a known unique index key. If more complex find operations are needed, the SQL view `RDbView` and the `RdbRowSet` base provide adequate means.

The following code uses `RDbTable` and an index to find a first match for a row. The index was defined in Section 2.3.3, "Index keys."

```
...
_LIT(KBooksTable, "Books");
_LIT(KBooksIndexName, "BooksIndex");
...
private: // Member data

    RDbStoreDatabase iBookDb;
    ...

void CBookDb::GetABookFastL(const TDesC& aTitle, TDes& aResult)
{
    RDbTable rowset;
    RDbTable table;
    TDbSeekKey seekKey(aTitle); // Initialize one-column seek key
```

```

// Open the 'Books' table. Specify what index is used.
User::LeaveIfError(
    table.Open(iBookDb, KBooksTable, rowset.EReadOnly));
CleanupClosePushL(table); // Remember to pop and close
User::LeaveIfError(table.SetIndex(KBooksIndexName));

if( rowset.SeekL(seekKey) ) // SeekL requires index is set
{
    // Cursor is set to the found row. Do something with it.
    rowset.GetL();
    ...
}
else
{
    // Result not found
}

...
// Pop the table from cleanup stack and close it.
CleanupStack::PopAndDestroy();

```

The index data type must match the target type, that is, TInt to ColIntnn, and so on. A TdbSeekKey object is created with the required index matching value (aTitle). In the example, there is one known index, but it is also possible to query indexes for a table as follows:

```
CDbNames* idxs = iBookDb.IndexNamesL(KBooksTable);
```

It is important to set the index for the table, before trying to seek it. Finally, SeekL() attempts to match the value. If the key value is matched, the cursor is set to that position. A subsequent GetL() will retrieve the data ready for further processing.

4. Using SQL with DBMS

4.1 SQL and the native C++ API

Symbian OS DBMS supports many operations through both a "native" C++ API and through API calls backed by SQL statements. In very early versions, the support for SQL was limited to Select statements; the proprietary native C++ API provided the main means to work with databases. Currently, there is much better support for SQL. From the developer point of view, this literally means that most operations can be accomplished in two ways.

Generally, the code using SQL is simpler and executes faster, which is why it should be favored. There are, however, some restrictions. For example, long transactions, compaction, recovery, and updating of binary columns require using the C++ API. Additionally, the SQL Insert statement for large amounts of data is slower than using the equivalent C++ API.

It should be noted from the beginning that the DBMS engine does not support a full implementation of SQL; it is designed around limited resources and therefore does not carry the power or capabilities of a server-based SQL implementation. Within this restriction, the SQL implementation is powerful enough for the day-to-day needs of a mobile device.



Note: Both native and SQL DBMS statement handling may have particular advantages or disadvantages in certain circumstances, and some operations can only be accomplished with one or the other. Developers should be aware of this fact and tailor their applications against expected usage. More information about this characteristic can be found on the Symbian Web site and in the SDK documentation.

In Symbian OS DBMS, SQL statements are passed unambiguously to the SQL engine, for example, the C++ code passes a descriptor of a string holding the entire SQL statement for interpretation.

4.2 Breakdown of a Select statement

As a frequently used SQL statement, the Select statement will be covered in this section in some detail. For the sake of clarity, and here only, the SQL syntax shown is presented largely without the Symbian OS DBMS member function code in which it embeds. Section 4.3, "Supported SQL subset," gives a full breakdown of the terms used in the syntax of the statement.

The Select statement will return a RowSet that can be used to retrieve actual row data. The Select statement has the following basic form:

```
SELECT select-list FROM table-name [ WHERE search-condition ] [ ORDER BY sort-order ]
```

select-list has two forms. Specifying * retrieves all columns in the table to the RowSet in an undefined order. Specifying a comma-separated list defines which columns to return and their order in the RowSet.

table-name refers to a Table that must exist in the referred-to database.

`search-condition` specifies condition(s) that a row must meet in order to be present in the resultant RowSet; it yields the result of Boolean operations such as:

boolean-term [OR *search-condition*] for example,

```
a=1 OR NOT b=2 AND c=3
```

A *search-condition* can be defined by a single *predicate*; more complex searches are constructed by combining predicates using the keywords AND, OR, and NOT, and parentheses to override the precedence of these operators. If used without brackets, the order of precedence is NOT, AND, and then OR. Thus,

```
a=1 OR NOT b=2 AND c=3
```

is equivalent to

```
(a=1 OR ((NOT b=2) AND c=3))
```



Note: In SQL, the equals (=) character is *not* an assignment.

A *search-condition* is also used by `RDbRowSet::FindL()` and `RDbRowConstraint::Open()`, to specify which rows of the RowSet will be returned by `RDbRowSet::FindL()` or matched by `RDbRowSet::MatchL()`.

Predicates are the building blocks of a search condition. There are three types: *comparison*, *like*, and *null*; each tests a condition of a column in the selected table.

A *comparison-predicate* compares a column value with a literal value. Column comparisons are type-dependent, so the literal supplied to the statement must be of the same type. Numeric columns are compared numerically, text columns are compared lexically, and date columns are compared historically. Binary columns cannot be compared.

The *like-predicate* is used to determine whether or not a text column matches a pattern string. See the description of the `TDesC::Match()` function.

The *null-predicate* tests whether a column is NULL. It can be applied to all column types.

The *sort-order* is specified with the `ORDER BY` clause in the Select statement. If the clause is not used, the order in which rows are presented is undefined. Columns specified in *sort-order* can be retrieved in ascending (default) or descending order, and should appear in the clause in decreasing order of precedence.

An example of an embedded select Select statement is:

```
_LIT(KSQLStatement, "SELECT Author, Title FROM Books WHERE Author = 'Brown,
Dale' ORDER BY Author, Title");
...
User::LeaveIfError(view.Prepare(iBookDb, TDbQuery(KSQLStatement,
    EDbCompareNormal)));
...

```

Here the `_LIT` macro loads the complete SQL statement into the `KSQLStatement` descriptor where it is then used as a parameter to a `TDbQuery()`.



Note: The `RDbView` class uses a `Select` statement to specify what data should be present in the `RowSet` and how to present it. It is passed as a parameter to `RDbView::Prepare()`, wrapped in a `TDbQuery` object.

4.3 Supported SQL subset

The following table details the SQL subset implemented in Symbian OS DBMS.

Term	Description
add-column-set	<i>add-column-spec</i> (<i>add-column-spec</i> ,...)
add-column-spec	<i>column-identifier data-type</i>
boolean-factor	[NOT <i>boolean-primary</i>]
boolean-primary	<i>predicate</i> (<i>search-condition</i>)
boolean-term	<i>boolean-factor</i> [AND <i>boolean-term</i>]
column-definition	<i>column-identifier data-type</i> [NOT NULL]
column-identifier	<i>user-defined-name</i>
column-value	<i>literal</i> NULL
comparison-operator	< > <= >= = <>
comparison-predicate	<i>column-identifier comparison-operator literal</i>
data-type	BIT [UNSIGNED] TINYINT [UNSIGNED] SMALLINT [UNSIGNED] INTEGER COUNTER BIGINT REAL FLOAT DOUBLE [PRECISION] DATE TIME TIMESTAMP CHAR [(n)] VARCHAR [(n)] LONG VARCHAR BINARY [(n)] VARBINARY [(n)] LONG VARBINARY
date-literal	#{date expression}# Note: Date and time expressions are those that can be parsed by <code>TTime::Parse()</code> .
digit	0 1 2 ... 8 9
drop-column-set	<i>column-identifier</i> (<i>column-identifier</i> ,...)
index-name	<i>user-defined-name</i>
letter	a b ... y z A B ... Y Z
like-predicate	<i>column-identifier</i> [NOT] LIKE <i>pattern-value</i>
literal	<i>string-literal</i> <i>numeric-literal</i> <i>date-literal</i>

Term	Description
null-predicate	<i>column-identifier</i> IS [NOT] NULL
numeric-literal	{numeric types} Note: Numeric forms are those that can be interpreted by <code>Tlex::Val(TInt64&)</code> and <code>Tlex::Val(TReal&)</code> .
pattern-value	<i>string-literal</i> Note: When wildcarding, DBMS uses File Server conventions rather than SQL-standard characters in the pattern mask: Question mark (?) represents any single character and asterisk (*) represents any number characters. Standard SQL uses the underscore (_) and percent (%) characters, respectively.
predicate	<i>comparison-predicate</i> <i>like-predicate</i> <i>null-predicate</i>
search-condition	<i>boolean-term</i> [OR <i>search-condition</i>]
select-list	* <i>column-identifier</i> ,...
sort-specification	<i>column-identifier</i> [ASC DESC]
string-literal	'{character}' Note: Character strings can contain any text character. Single quote characters can be embedded by using two consecutive single quote characters ("').
table-name	<i>user-defined-name</i>
update-column	<i>column-identifier</i> = <i>column-value</i>
user-defined-name	<i>letter</i> [<i>letter</i> <i>digit</i> _]...

4.4 Using TDbQuery class

The design of this class is such that the descriptor containing the SQL can be passed in place of the `TDbQuery` object, which will be implicitly constructed with normal text comparison as its default. For example, in the following code snippet:

```
_LIT(KSQLStatement ="SELECT Author, Title FROM Books WHERE Author LIKE  
'BR*');  
RDbView dbview;  
view.Prepare(iBookDb, KSQLStatement);
```

the `view.Prepare` statement is functionally equivalent to:

```
view.Prepare(iBookDb, TDbQuery(KSQLStatement, EDbCompareNormal));
```

To specify any other comparison type, the query object must be explicitly constructed, for example:

```
view.Prepare(iBookDb, TDbQuery(KSQLStatement, EDbCompareFolded));
```

4.5 SQL schema and data updates

Modification of SQL schema and data can be accomplished through direct use of SQL commands as well as through the functionally equivalent DBMS interfaces `RDbDatabase` and `RDbIncremental`.

Note that the use of SQL statements may not provide all of the functionality presented by the native DBMS API.

4.5.1 Database Definition Language (DDL)

DDL is used to alter high-level functionality in a database; this functionality encompasses the creation and dropping (deletion) of tables and indexes in the database, as well as the alteration of an existing table.

DDL statements cannot modify the data in a database; however, when a table is dropped, all data in the named table is lost. Dropping an index simply removes the ability to access particular table data through that index.

4.5.1.1 Create table statement

```
CREATE TABLE table-name (column-definition,...)
```

4.5.1.2 Drop table statement

```
DROP TABLE table-name
```

4.5.1.3 Alter table statement

```
ALTER TABLE table-name { ADD add-column-set [ DROP drop-column-set ] | DROP drop-column-set }
```

4.5.1.4 Create index statement

```
CREATE [ UNIQUE ] INDEX index-name ON table-name ( sort-specification,... )
```

4.5.1.5 Drop index statement

```
DROP INDEX index-name FROM table-name
```

4.5.2 SQL data update statements (DML)

DML is used to alter functionality in the database at the column level, and the usage is confined to row insertion, deletion, and update.

DML statements can modify the content of existing data in the database.

4.5.2.1 Insert statement

```
INSERT INTO table-name [ ( column-identifier,... ) ] VALUES ( column-value,... )
```

4.5.2.2 Delete statement

```
DELETE FROM table-name [ WHERE search-condition ]
```

4.5.2.3 Update statement

```
UPDATE table-name SET update-column,... [ WHERE search-condition ]
```

4.5.3 Examples

The simplest way to perform DDL or DML statements is to use `Execute` of `RdbDatabase`. If operations are long lasting, it is recommended to use incremental methods (see Chapter 5, “Incremental database operations”).

The following code snippets show simple database schema and data updates:

```
private: // Member data

    RdbStoreDatabase iBookDb; // Could be RDbNamedDatabase
    ...

TInt CBookDb::AddDateColumn()
{
    _LIT(KSqlAddDate, "ALTER TABLE Books ADD PublishDate DATE");
    return iBookDb.Execute(KSqlAddDate);
}

TInt CBookDb::DropBooksTable()
{
    _LIT(KSqlDropBooks, "DROP TABLE Books");
    return iBookDb.Execute(KSqlDropBooks);
}

TInt CBookDb::DeleteAllBooks()
{
    _LIT(KSqlDeleteAllBooks, "DELETE FROM Books");
    return iBookDb.Execute(KSqlDeleteAllBooks);
}

// Delete books, name of which starts with F-letter
```

```

TInt CbookDb::DeleteFBooks ()
{
    _LIT(KSqlDeleteFBooks, "DELETE FROM Books where
                           Title like 'F*'");
    return iBookDb.Execute(KSqlDeleteFBooks);
}

TInt CBookDb::UpdateBookTitle(const TDesC& aOldTitleKey,
                              const TDesC& aNewTitle)
{
    _LIT(KSQLUpdateStart, "UPDATE Books SET Title = '");
    _LIT(KSQLUpdateMiddle, "' WHERE Title = '");
    _LIT(KSQLUpdateEnd, "'");

    // SQL: UPDATE Books SET Title = 'aNewTitle'
    //      WHERE Title = 'aOldTitleKey'

    TBuf<KCustomSqlMaxLength> sqlStr;
    sqlStr.Append(KSQLUpdateStart);
    sqlStr.Append(aNewTitle);
    sqlStr.Append(KSQLUpdateMiddle);
    sqlStr.Append(aOldTitleKey);
    sqlStr.Append(KSQLUpdateEnd);

    return iBookDb.Execute(sqlStr);
}

```

5. Incremental database operations

5.1 Overview

Some database operations can take an extended time to complete, such as new index creation or a database recovery. This may be acceptable sometimes, but generally any such activity has the drawback of making the host device unresponsive.

To circumvent this scenario, Symbian OS DBMS implements the Incremental Database Operations API, which allows potentially long-running tasks to be performed in steps so that the device remains responsive to other events.

The API has two key concepts: DDL (SQL schema update) for database incremental operations and DML (SQL data update) for statement incremental execution. Two classes are provided to support this paradigm: `RdbIncremental` and `RdbUpdate`, respectively.

While any incremental operation is in progress, the database cannot be used for any other operations such as opening tables or preparing views. Beginning an incremental operation also requires that there are no RowSets or unaccomplished transactions open in the database.

If no explicit transaction has been started by the database, an automatic transaction is begun when any incremental operation is started and is committed when complete, or rolled back if the operation fails or is abandoned prior to completion.

Transactions are covered in more detail in Section 6.3, "Transactions and locks."

5.2 Database incremental operations — `RDbIncremental`

`RDbIncremental` allows long-running DDL statement executions such as table or index alteration/discard, or database compaction and recovery, to be performed in incremental steps.

The following code fragment illustrates the incremental execution of an SQL DDL statement that is no better than use of direct execution but still demonstrates the conditions for running `RDbIncremental`.

```

_LIT(KSqlTxt, "DROP TABLE Books");      // SQL for execution
RDbIncremental incOp;                    // Create Incremental Object
TInt incStep = 0xFFFF;                  // Step Variable
TInt incStat = incOp.Execute(iBookDb, KSqlTxt, incStep);
                                           // Initialize Execution
while (incStep>0 && incStat==KErrNone)
{
    incStat = incOp.Next(incStep);        // Do the work
}
...
incOp.Close();

```

5.3 DML statement incremental execution — RDbUpdate

The `RDbUpdate` class is provided to perform DML statements incrementally. The class can be used either synchronously or asynchronously; in the latter case, it may be wrapped in an active object if required. Based on the example, the following code fragments detail methods of both synchronous and asynchronous incremental execution of a DML statement.

5.3.1 DML synchronous

`RdbUpdate` provides the means to execute data manipulation operations synchronously. `Execute` initializes operation, while a call to `Next` performs it bit by bit.

```
TInt CBookDb::RemoveBooks(const TDesC& aTitle, TInt& aResultCount)
{
    RDbUpdate updOp;

    // Sql: DELETE FROM Books WHERE Title LIKE 'aTitle'
    TBuf<KCustomSqlMaxLength> sqlStr;
    sqlStr.Append(_L("DELETE FROM "));
    sqlStr.Append(KBooksTable);
    sqlStr.Append(_L(" WHERE "));
    sqlStr.Append(KBooksTitleCol);
    sqlStr.Append(_L(" LIKE '"));
    sqlStr.Append(aTitle);
    sqlStr.Append(_L("'"));

    // Initialize execution and perform the first step.
    // Note: Execute() returns 0 (=KErrNone), but it does not
    // affect database until Next() is called.
    TInt incStat = updOp.Execute(iBookDb, sqlStr, EDbCompareFolded);
    incStat = updOp.Next(); // This will leave, if Execute()
                          // failed.

    // Just in case, if the operation has more steps
    while( incStat == 1 )
    {
        incStat = updOp.Next();
    }
    aResultCount = updOp.RowCount();
    updOp.Close();
    return incStat; // KErrNone or system wide error code
}
```

In this example, and once the DML operation is initiated, any requirement for a large amount of processing to be performed may make the device unresponsive until completion. This is why the asynchronous approach should be considered.

5.3.2 DML asynchronous

`RdbUpdate` provides the means to execute data manipulation operations asynchronously.

In the interest of good design, the asynchronous code should be put within a custom active object. The `Next` method of `RdbUpdate` takes in a `TrequestStatus`, which should be `iStatus` of the active object. Completion of `Next` is handled in `RunL`.

For the sake of simplicity, in the following example the code waits for asynchronous operation to complete, that is, in practice it is synchronous.

```
TInt CBooksDb::RemoveAllBooks(TInt& aResultCount)
{
    _LIT(KSql, "DELETE FROM Books");

    RdbUpdate updOp;
    TRequestStatus incStat(1);
    TInt updStat = updOp.Execute(iBookDb, KSql, EDbCompareFolded);
    while (updStat==KErrNone && incStat ==1)
    {
        updOp.Next(incStat);    // Start async operation. Call returns
                               // immediately. Operation will continue
                               // in the background.

        // For simplicity wait completion here. If active objects were
        // used, active objects framework would handle completion.
        User::WaitForRequest(incStat);
    }
    aResultCount = updOp.RowCount();
    updOp.Close();

    if (updStat!=KErrNone)
        return updStat;        // System wide error code
    else
        return incStat.Int(); // KErrNone or system wide error code
}
```

The `Update` object is created and a `TRequestStatus` variable initialized on it. After the initial invocation to commence the operation (`Execute()`), update operations continue as long as `incStat` is greater than zero.

Because the call to `Next` is asynchronous, it is necessary to wait for the request to complete before calling `Next` again. In this simplified example, it is achieved manually by a call to `User::WaitForRequest(incStat)`.

When the incremental operation is complete, the `RowCount()` method is used to return the number of rows affected by the operation. Finally, the `RdbUpdate` object is closed.

6. Sharing databases

Symbian OS DBMS provides a server facility that allows multiple clients to access the same database simultaneously. A transaction mechanism ensures that only one client at a time can change data.

The API has two key concepts: DBMS server session and database change notifier.

Shared database access can be achieved with `RDbNamedDatabase` only.

6.1 DBMS server session

A session with the DBMS server allows suitable databases to be shared with read/write access by multiple clients. Note that a session can be used to open any number of databases, and that if write access is required by one client, all clients of that database must use the DBMS server.

The DBMS server session is provided by `RDBs`.

6.2 Database change notifier

Clients can be notified of changes (such as transactions committed or rolled back, database closed or recovered) to shared databases through `RDbNotifier`.

6.3 Transactions and locks

Transaction support is provided by the `RDbDatabase` abstract base class. A transaction state may be invoked explicitly via `Begin()`; otherwise all updates are made inside an automatic transaction. Explicitly invoked transactions are ended by either a `Commit()` or a `Rollback()` that either permanently changes the current database state or restores the previous respectively. If a transaction was started with `Begin()`, an enquiry as to whether a transaction state exists is possible with a call to `InTransaction()` that returns a `TBool` type.

The transaction mechanism has a restriction in that any transaction may contain either DML operations or DDL operations, but not both. To prevent a single transaction from containing both forms of statements, all affected cursors are invalidated when the definition of their underlying table is modified. A cursor yields a `KErrDisconnected` error in this state and can only be closed. The `RowSet` can be regenerated once the DDL statement has completed. Calling `Reset()` on such a cursor has no effect. Rows cannot be updated while changing the schema.

It is possible for a client to change the database schema while other clients are using that database, provided they have no locks on it. However, those other clients may find that their `RowSets` become invalidated asynchronously.

The transaction mechanism does not provide isolation between clients; within a transaction, while one client is updating a table, other clients can see the changes as they are made. For example, if a client retrieves two separate rows from a database, there is no automatic guarantee that the data being retrieved has not been changed between the reads; this can lead to an "inconsistent read." A client can prevent an update while retrieving related rows by enclosing the individual reads within a transaction. This transaction does not modify the

database but operates as a read-lock; the `Commit()` or `Rollback()` member functions of the abstract base class `RDbDatabase` release this type of lock and do not affect the database.

Calling `Begin()` on a shared database attempts to get a shared read-lock on the database; this fails if any other client already has an exclusive write-lock. Other clients with read-locks will not affect completion of this statement.

Any operation that modifies the database attempts to get an exclusive write-lock, which will fail if another client has any kind of lock on that database. If the current client already has a read-lock as a result of calling `Begin()`, the lock will be changed to an exclusive write-lock.

Calling `Commit()` or `Rollback()` after a read-lock has been acquired will release the client's lock. The database is only considered to be unlocked when all such locks are removed by all clients. It will then report an `RDbNotifier::EUnlock` database event to any change notifier.

Calling `Commit()` or `Rollback()` after a write-lock has been acquired releases the client's lock and reports an `RDbNotifier::ECommit` or an `RDbNotifier::ERollback` database event to any change notifier.

Automatic transactions are used if updates are made outside of explicit transactions; these updates can fail if an exclusive lock cannot be acquired.

Sharing read-locks enables greater concurrency while providing a safeguard against inconsistent reads. However, the method allows the possibility of a deadlock. If two clients attempt to update a database and both `Begin()` a transaction before either of them starts an update, then one client's read-lock will prevent the other from upgrading to a write-lock, and vice versa. Coding the clients to back out of such a deadlock situation, rather than retry forever without releasing the locks, will resolve this situation.

7. Summary

Symbian OS DBMS is a relatively straightforward implementation with extensive capabilities for its size and target platform. Smartphones based on Symbian OS will continue to acquire larger amounts of memory/storage as the platform evolves, and Symbian OS DBMS will be able to support increasingly larger databases on these devices for the foreseeable future.

The DBMS API has been coded specifically to allow alternative implementations to be written should the developer desire to do so. However, any such undertaking would be an extensive commitment.

The current implementation provides two alternative approaches to manipulating data. The C++ API is supported, but for most operations SQL is more efficient and cleaner, and thus preferred.

More information about Symbian OS DBMS and related technologies mentioned in this document can be found in the S60 SDK and on the Symbian Web site.

8. Terms and abbreviations

Term or Abbreviation	Meaning
API	Application Programming Interface
DBMS	Database Management System
DDL	Data Definition Language
DML	Data Modeling Language
ISAM	Indexed Sequential Access Method
RDBMS	Relational Database Management System
SQL	Structured Query Language

9. References

- [1] [S60 Platform: DBMS Example](#), available at www.forum.nokia.com

10. Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by [rating this resource](#).