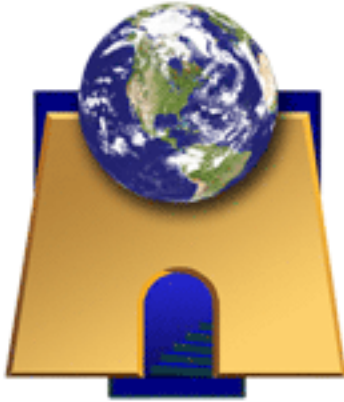


ITTIA DB User's Guide

ITTIA DB User's Guide



Copyright © 2005-2009 ITTIA LLC

Table of Contents

1. Introduction	1
1.1. Overview	1
1.2. Supported Platforms	2
1.3. ITTIA DB Product Family	2
1.3.1. Upgrading to Standard and Plus	3
1.4. About the Documentation	4
2. Database Concepts	5
2.1. Introduction to Databases	5
2.2. Mobile and Embedded Considerations	5
3. Using ITTIA DB	7
3.1. Selecting an API	7
3.2. C++ API Examples	7
3.3. Linking ITTIA DB into an Application	9
3.3.1. Microsoft Visual C++ 6	10
3.3.2. Microsoft Visual C++ 2003/2005	12
3.3.3. Microsoft eMbedded Visual C++ 4.0	13
3.3.4. GNU gcc	14
3.4. Included Example Programs	15
3.4.1. Microsoft Visual C++	15
3.4.2. Microsoft Visual Studio	15
3.4.3. Microsoft eMbedded Visual C++	16
3.4.4. GNU gcc	16
3.5. Multi-threading	16
3.6. Database Files	16
4. SQL	17
4.1. Introduction to SQL	17
4.1.1. Creating Tables and Inserting Values	17
4.1.2. Projection and Selection	18
4.1.3. Joining Related Tables	19
4.1.4. SQL Query Parameters	19
4.2. SQL Language Reference	20
4.2.1. Column Types	22
4.2.2. SELECT Queries	25
4.2.3. Natural Numbers Virtual Table	30
4.2.4. Built-in Functions	31
4.2.5. Data Manipulation	35
4.2.6. Transactions	36
4.2.7. Schema Definition	37
5. The C++ API	39
5.1. Getting started	39
5.1.1. Creating and Opening a Database	39
5.1.2. Error Handling	39
5.1.3. Run-time Configuration	40
5.2. Database Design	40
5.2.1. Overview	40
5.2.2. Column Types	41
5.2.3. Defining the Database Schema	42
5.3. Database Access	43
5.3.1. Transactions	43
5.3.2. Table Cursors	43
5.3.3. Inserting Data	44
5.3.4. Traversing a Table	44
5.3.5. Searching for a Row	44

5.3.6. Reading Data	45
5.3.7. Updating Data	45
5.3.8. Deleting the Row	45
5.3.9. Reading and Writing BLOBs	45
5.4. SQL Queries	46
6. The C API	49
6.1. Getting Started	49
6.1.1. Initializing the Library	49
6.1.2. Creating and Opening a Database	49
6.1.3. Error Handling	49
6.1.4. Data Structures	50
6.2. Database Design	50
6.2.1. Column Types	50
6.2.2. Tables, Fields and Indexes	52
6.2.3. Sequence Generators	53
6.3. Database Access	53
6.3.1. Transactions	53
6.3.2. Cursors	54
6.3.3. Rows	54
6.3.4. Inserting Data	59
6.3.5. Traversing a Cursor	60
6.3.6. Updating Data	60
6.3.7. Deleting the Row	60
6.3.8. Reading and Writing BLOBs	61
6.4. SQL Queries	62
7. Shared Database Access	65
7.1. Overview	65
7.2. Client/server Shared Access	65
7.3. Multi-threaded Shared Access	66
7.4. Isolation Levels	67
7.4.1. Serializable	67
7.4.2. Repeatable Read	68
7.4.3. Read Committed	68
7.4.4. Read Uncommitted	68
7.5. Selecting an Isolation Level	68
7.6. Change Notification	69
8. Advanced Topics	70
8.1. Introduction	70
8.1.1. Recommended Reading	70
8.2. ACID Properties	70
8.3. Logging	71
8.4. Locking	72
8.4.1. Types of Locks	72
8.4.2. Lockable Objects	72
8.4.3. Index Locking	72
8.4.4. Manual Locking	73
I. Utilities	
dbserver	75
dbserver GUI	77
ittiasql	78
Index	80

List of Figures

- 6.1. Managed Field Binding 55
- 6.2. Absolute Field Binding 56
- 6.3. Relative Field Bindings 58

List of Tables

1.1. Supported platforms	2
1.2. Features Available in Each Edition of ITTIA DB	3
3.1. ITTIA DB Alternative Kernel Libraries	10
4.1. SQL Reserved Keywords	22
4.2. SQL Column Types	23
4.3. Date and Time Literal Formats	25
4.4. SQL Comparison Operators	28
4.5. SQL Aggregate Functions	29
4.6. SQL Built-in Functions	31
5.1. C++ Column Types	41
6.1. C Column Types	51
7.1. Isolation Level Implementations	67
7.2. Phenomena prevented by Serializable Isolation	68
8.1. ITTIA DB Database Characteristics Quick Reference	70
8.2. ACID Properties	71
16. Utility program files by platform	74

List of Examples

3.1. C++ Hello World Program	7
3.2. C++ Hello World Program With SQL	8
4.1. CREATE TABLE	17
4.2. INSERT INTO	18
4.3. SELECT	18
4.4. SELECT ... JOIN	19
4.5. Positional SQL Query Parameters	19
4.6. Numbered SQL Query Parameters	20
4.7. coalesce vs. case	34
5.1. SQL Select Example in C++	47
5.2. SQL Insert Example in C++	48
6.1. Prepare to access BLOB field	61
6.2. Write data from a file to a BLOB field	61
6.3. Read data from a BLOB field to a file	62
6.4. Executing a SELECT statement	63

Chapter 1. Introduction

1.1. Overview

ITTIA DB is a database management system for environments with extremely limited memory and processor resources. A solid database kernel provides reliable data management at optimal speeds, protecting data from power failure and crashes.

ITTIA DB is:

Fast

Today's mobile device users demand instant data access. ITTIA DB was developed to help meet this requirement.

Flexible

ITTIA DB is designed with extensibility in mind. Add and remove database components to suite your application's exact requirements and achieve minimal footprint. The kernel is a tightly integrated set of components with clearly defined interfaces, which makes it easy to develop a customized version of any component using the included source code.

Efficient and Lightweight

Optimized for restricted environments, ITTIA DB is able to utilize scarce resources most effectively. Battery life, form-factor and cost requirements greatly restrict the resources available to mobile and embedded developers. ITTIA DB is built for minimum memory and processor time consumption.

Reliable

In mobile devices, the power can fail at any time. Thus, the database is fully ACID-compliant, which ensures data integrity through the use of transactions and recovery, and data consistency by means of isolation.

Features include:

- Cross-platform compatibility
- Shared database access
- Small footprint
- Support for multithreading
- Modular design: add new features with ease
- Configurable: flexibility at compile-time and run-time
- Embedded directly into applications for improved performance
- Portable database file format
- Support for solid state storage such as CompactFlash
- Data encryption

- Dynamic schema alteration
- Variable-sized records
- Multiple isolation levels

1.2. Supported Platforms

ITTIA DB is a cross-platform database management system, designed to work in a variety of environments. In addition to the platforms listed below, ITTIA DB can be easily ported to run on other operating systems and architectures.

Operating System	Architctures	Development Tools
Windows NT Windows 2000 Windows XP	x86	Visual C++ 6.0 Visual C++ .NET gcc (MingW)
Windows CE Windows Mobile	x86 ARM	Visual C++ eMbedded 4.0
Linux	x86 ARM	gcc
MontaVista Linux Mobilinux	x86 ARM	DevRocket gcc
VxWorks	x86	Wind River Workbench

Table 1.1. Supported platforms

1.3. ITTIA DB Product Family

ITTIA DB is a family of products for high-performance relational data storage on embedded systems and devices. Each edition of ITTIA DB is designed to meet the requirements of a certain class of devices with the best performance and minimum footprint. The same API and file format is used by all ITTIA DB products.

ITTIA DB Compact

For devices that only need table-based storage with high-performance indexing, this software library gives devices direct access to ITTIA DB database files.

ITTIA DB-SQL Standard

For devices storing mission-critical data, ITTIA DB-SQL Standard incorporates ACID-compliant crash recovery with optional run-time SQL queries.

ITTIA DB-SQL Plus

Share database files between threads, processes, and devices using ITTIA's lightweight data server, designed to support a high level of concurrency even in low-memory environments.

Table 1.2, “Features Available in Each Edition of ITTIA DB” explains each of the features in Compact, Standard,

and Plus. API functions that are not supported by a particular edition of ITTIA DB will produce the error DB_ENOTIMPL when called.

Feature	Compact	Standard	Plus
Transaction rollback	Yes	Yes	Yes
Strongly-typed tables	Yes	Yes	Yes
B+tree indexes	Yes	Yes	Yes
Platform-independent file format	Yes	Yes	Yes
SQL	No	Yes	Yes
Client/server shared access	No	No	Yes
Multi-threaded shared access	No ^a	No ^a	Yes
Row-level locking and isolation	No	No	Yes
Recovery logging	Optional ^b	Yes	Yes
Dynamic schema alteration	Yes ^b	Yes	Yes
Encryption callbacks	Optional ^b	Yes	Yes
Change notification	Optional ^b	Yes	Yes
Sequences	Optional ^b	Yes	Yes
BLOB data type	Optional ^b	Yes	Yes
Unicode data type	Yes ^b	Yes	Yes
Automatic type conversion	Optional ^b	Yes	Yes

Table 1.2. Features Available in Each Edition of ITTIA DB

^aWhile ITTIA DB API calls are not thread-safe without multi-threaded shared access, it is compiled against the multithreaded C runtime when available.

^bThis feature can be enabled or disabled in the source code version of ITTIA DB. Optional features are not included in the binary compact library distributed by ITTIA.

Also see Table 3.1, “ITTIA DB Alternative Kernel Libraries” for a list of libraries in ITTIA DB software development kits.

1.3.1. Upgrading to Standard and Plus

Applications developed using ITTIA DB Compact can be upgraded to ITTIA DB-SQL Standard or ITTIA DB-SQL Plus. Applications developed for ITTIA DB-SQL Standard can also be upgraded to ITTIA DB-SQL Plus.

To upgrade an existing ITTIA DB application to ITTIA DB-SQL Standard, modify the build configuration to link with one of the single-user libraries listed in Table 3.1, “ITTIA DB Alternative Kernel Libraries”. Automatic recovery will be automatically enabled without modifying the application code. The application will also be able to use SQL queries if the SQL library is selected.

To upgrade an existing ITTIA DB application to ITTIA DB-SQL Plus, modify the build configuration to link with one of the default, IPC client, or local storage libraries listed in Table 3.1, “ITTIA DB Alternative Kernel Libraries”. Row-level locking will be automatically enabled without modifying the application code. The application will also be able to share an open database between threads, or connect to **dbserver** to share databases between multiple applications running on one or more devices.

1.4. About the Documentation

Documentation for ITTIA DB consists of several documents:

- User's Guide
- C API Reference Manual
- C++ API Reference Manual

All products in the ITTIA DB family - Compand, Standard, and Plus - are covered by the documentation. To find out whether a feature described in the documentation is available in a particular edition of ITTIA DB, see Table 1.2, "Features Available in Each Edition of ITTIA DB".

The User's Guide provides an introduction to software development with ITTIA DB. Example code from the **phone-book** example application is used to demonstrate many key concepts and tasks.

Consult the API Reference Manuals for detailed information about specific ITTIA DB API functions.

Documentation is available in a variety of formats, including:

- HTML
- Windows Help
- PDF

Chapter 2. Database Concepts

2.1. Introduction to Databases

A database management system provides the most convenient means to store data in a reliable, organized format. This removes the burden of managing data from the application code and provides several key advantages that an application would otherwise not have access to.

Reliable storage is the primary motivation for using a database. Operations on a database are grouped into transactions, which will either succeed (commit) or fail (abort) as a single unit. Because transactions are persistent, data will not become lost or corrupted even after a power failure. After a crash, data is automatically recovered up to the last committed transaction.

Warning

A database cannot protect against a corrupted file system or damaged media hardware. Regular backups are the only way to provide full protection against data loss.

A database can also provide powerful tools to help organize and access data. The structure of a database is specified in the database schema, which is used as the basis for navigating the data. Data is stored in tables, which contain a list of typed columns. Keys are defined on certain columns of the table to quickly locate rows based on the data stored in the key column. This fast data access through indexed key fields is another primary advantage of using a database management system.

Because data is stored in an organized format, it is possible to perform generic queries on the data. This allows existing software tools to be used with the data stored in the database, regardless of what application it was created with. Data can also be stored in a platform-independent format so that database files can be moved between different operating systems and processor architectures.

To prevent data inconsistency, it is important to manage how multiple users modify the database at the same time. A database management system provides concurrency support to ensure that transactions committed by multiple users always appear to complete sequentially. This provides shared access to the database in a safe, efficient way.

In general databases support multiple levels of concurrency. Concurrency refers to the ability of the database to handle multiple simultaneous access to the data stored in the database. The most simple case is that the database supports no concurrency at all. This case is more common in embedded environment where a single process accesses the database sequentially. The databases that support concurrency can be divided into the following categories:

1. Multi-threaded: In this architecture single process accesses the data in the database using multiple threads and
2. Client server architecture: This model supports multiple clients accessing the database- remotely or locally.

Note

ITTIA DB supports all levels of concurrency mentioned above.

2.2. Mobile and Embedded Considerations

On mobile devices and embedded systems, resource restrictions provide an additional challenge to developers. Execution speed, storage space, and memory are often limited. In embedded environments, known bounds on the size of resources can allow those resources to be used more effectively.

Resources on such devices can also exhibit different behavior than similar devices on a desktop or server system. For example, compact flash memory is a popular form of persistent storage on mobile and embedded devices. Unlike disk-based storage, flash storage has no seek time but slower throughput, and if data is written to the same location too many times, it will wear out.

A database management system, or any embedded application, can take advantage of this knowledge to operate more efficiently.

Chapter 3. Using ITTIA DB

3.1. Selecting an API

An ITTIA DB database can be accessed through one of two APIs. The C API provides the lowest level interface to the database and thus the greatest flexibility. The C++ API is easy to use and helpful for introducing the concepts of database access. The C++ API is provided as source code that can be linked with the C API interface to the library.

Some ITTIA DB configurations also support SQL, a dynamic query language that allows complex queries to be planned and executed at run-time using a simple string-based syntax. SQL queries can be executed from either C or C++ API.

3.2. C++ API Examples

The following example program demonstrates how to create an empty database, insert data, and perform an indexed search with direct table access:

Example 3.1. C++ Hello World Program

```
// A Hello World console program for ITTIA DB

#include <ittia/db++.h>
#include <iostream>

int main(int argc, char* argv[])
{
    // Create an empty database.
    db::Database db;
    db::StorageMode mode;
    db.create("hello_world.db", mode);

    // Create a new table with two fields and one index.
    db::FieldDescSet fields;
    db::IndexDescSet indexes;
    fields.add_uint("id");
    fields.add_string("message", 50);
    indexes.add_index("by_id", db::DB_UNIQUE).add_field("id");
    db.create_table("hello_world", fields, indexes);

    // Open a table cursor.
    db::Table helloWorld;
    helloWorld.open(db, "hello_world");

    // Start a transaction before adding or accessing data.
    db.tx_begin();

    // Insert a row using the table cursor.
    helloWorld.insert();
    helloWorld["id"] = 0;
    helloWorld["message"] = "Hello World";
    helloWorld.post();

    // Search for the row that was inserted.
    helloWorld.set_sort_order("by_id");
    helloWorld.begin_seek(db::DB_SEEK_EQUAL);
    helloWorld["id"] = 0;
```

```

if (DB_SUCCESS(helloWorld.apply_seek())) {
    db::String message = helloWorld["message"].as_string();
    std::cout << message.c_str() << std::endl;
} else {
    std::cerr << "Could not find row." << std::endl;
}

// Commit the transaction to make changes persistent.
db.tx_commit();

// Close the table cursor and database.
helloWorld.close();
db.close();

return 0;
}

```

The following example shows how the previous example can be written using SQL.

Example 3.2. C++ Hello World Program With SQL

```

// A Hello World console program for ITTIA DB

#include <ittia/db++.h>
#include <iostream>

using namespace db;
using namespace std;

int main(int argc, char* argv[])
{
    //-----
    // Create a Database handle.
    //-----
    Database db;
    StorageMode mode;

    //-----
    // Create a Query object.
    //-----
    Query q;

    //-----
    // Create a new database file using the default mode.
    //-----
    db.create("hello_world.db", mode);
    int rc;

    //-----
    // Create hello_world table using the Query object.
    //-----
    if(!DB_SUCCESS(q.exec_direct(db,
        "create table hello_world (id integer, message varchar(20))")))
    {
        cout << "Unable to create table\n";
        return -1;
    }

    //-----
    // Start a transaction prior to modifying the hello_world table.

```

```

//-----
if(!DB_SUCCESS(q.exec_direct(db,
    "START TRANSACTION")))
{
    cout << "Unable to insert values into table\n";
    return -1;
}

//-----
// Insert 'hello world' message into the table.
//-----
if(!DB_SUCCESS(q.exec_direct(db,
    "insert into hello_world values(0, 'hello world')")))
{
    cout << "Unable to insert values into table\n";
    return -1;
}

//-----
// Select the message that was just inserted.
//-----
if(!DB_SUCCESS(q.exec_direct(db,
    "select message from hello_world where id = 0")))
{
    cout << "Unable to select from table\n";
    return -1;
}

//-----
// Print query result.
//-----
StringField message(q, "message");
for (rc = q.seek_first(); DB_SUCCESS(rc) && !q.is_eof(); rc = q.seek_next())
    cout << String(message).c_str() << endl;

q.exec_direct(db, "COMMIT");

//-----
// Clean up.
//-----
q.close();
db.close();

return 0;
}

```

3.3. Linking ITTIA DB into an Application

The ITTIA DB kernel is a software library that can be directly embedded in an application, or accessed through a separate server processes. The database kernel provides a transactional C API that the application can use directly, or through ITTIA DB's C++ API bindings.

To minimize an application's code footprint and optimize performance, ITTIA DB includes several alternative configurations of the ITTIA DB kernel that can be substituted for the default kernel library. These configurations omit specific features without changing the API interface. Available libraries are listed in Table 3.1, "ITTIA DB Alternative Kernel Libraries".

Configuration	Windows Library	Run-time DLL	gcc Argument
Default	ittiasql.lib	ittiasql.dll	-littiasql -loslib

Configuration	Windows Library	Run-time DLL	gcc Argument
IPC client only	it-tiasql_client.lib	it-tiasql_client.dll	-littiasql_client -loslib
Local storage only, with SQL	ittiasql_local.lib	ittiasql_local.dll	-littiasql_local -loslib
Local storage only, without SQL	ittiadb_local.lib	ittiadb_local.dll	-littiadb_local -loslib
Single-user local storage, with SQL	ittiasql_su.lib	ittiasql_su.dll	-littiasql_su -loslib
Single-user local storage, without SQL	ittiadb_su.lib	ittiadb_su.dll	-littiadb_su -loslib
Compact kernel	it-tiadb_compact_su.lib	it-tiadb_compact_su.dll	-lit-tiadb_compact_su -loslib

Table 3.1. ITTIA DB Alternative Kernel Libraries

Note

ITTIA DB-SQL Plus and the evaluation packaging include all available configurations. ITTIA DB-SQL Standard contains the *single-user local storage* configurations only. ITTIA DB Compact contains only the *Compact kernel* configuration.

The *IPC client only* configuration is a lightweight inter-processes communications (IPC) client library that can only connect to remote databases using **dbserver**. When this configuration is used, databases filenames must be of the form `xdbtp://server[:port]/database`, where *server* is the name or IP address of the server, *port* is the TCP/IP port number, and *database* is the name of a database file on the server.

Local storage only configurations can only open local database files and cannot connect to remote servers. However, multiple threads in a single process can share access to a local database file with automatic row level locking provided by the database kernel.

Tip

For more details on client/server and local multi-threaded sharef access, see Shared Database Access.

Single-user local storage configurations support only one transaction at a time. Each database file can only be opened by one process at a time.

Some configurations do not include support for SQL. Preparing an SQL statement using one of these libraries will produce an error.

To get started with ITTIA DB, first obtain the package appropriate for your operating system and architecture and extract all files. For more instruction on how to install ITTIA DB please refer to the README file for you platform.

3.3.1. Microsoft Visual C++ 6

Note

The following instructions assume that ITTIA DB has been installed to the directory \$DBDIR.

Note

\$DBDIR mentioned above is not an environment variable. In the instructions that follow, you must replace it with the directory in which ITTIA DB is installed.

The following explains how to add ITTIA DB to a Visual C++ Project. As mentioned in Linking ITTIA DB into an application the database comes in different flavors. Single and multi-threaded versions as well as client/server. In the following the main steps focus on linking with single and multi-threaded version and linking with the client library for client server is covered as sub-steps.

1. Open Visual Studio and create a new Win32 Console Application project.
2. Open the settings for your project from the menu:
Project -> Settings
3. Select All Configurations from the Settings For list.
4. In the Project Settings dialog, select the C/C++ tab and from the Category list, select Preprocessor.
5. Add the folder \$DBDIR\win32\include to the Additional include directories property.
6. In the Project Settings dialog, select the Link tab and from the Category list, select Input.
7. Add \$DBDIR\win32\lib to the Additional library path property.
8. Add `ittiasql.lib` to the Object/library modules property.
9. Click OK to close the project settings dialog.
10. For any project that uses the C++ API, follow these additional steps:
 - a. Add the project \$DBDIR\win32\dbcppapi\dbcppapi.dsp to the workspace, using the following menu item:
Project -> Insert Project into Workspace
 - b. Include the dbcppapi project as a dependency. Select your project, then go to the menu item:
Project -> Dependencies
Check dbcppapi in the list that is presented and click OK.
 - c. In the Project Settings dialog, select the C/C++ tab and from the Category list, select Code Generation.
 - d. Select Win32 Release from the Settings For list. Set the Use run-time library property to Multi-threaded DLL.

Note

This step is necessary because all projects must be compiled using the same runtime library. Alternatively, you can change the Use run-time library setting for the dbcppapi project to match the desired setting for your project.

- e. Select Win32 Debug from the Settings For list. Set the Use run-time library property to Debug Multi-threaded DLL.

11. Add application source code files to the project. You can use the C++ example code to get started.
12. Copy `ittiasql.dll` from `$DBDIR\win32\lib` to the project directory.
13. Build and run your application.

3.3.2. Microsoft Visual C++ 2003/2005

Add ITTIA DB to a Visual C++ Project:

1. Open Visual Studio and create a new Win32 Console Application project.
2. Open the settings for your project from the menu:
Project -> Properties
3. Select All Configurations from the Configuration list.
4. Select the property page:
Configuration Properties -> C/C++ -> General
5. Add `$DBDIR\win32\include` to the Additional Include Directories property.
6. Select the property page:
Configuration Properties -> Linker -> General
7. Add `$DBDIR\win32\lib` to the Additional Library Directories property.
8. Select the property page:
Configuration Properties -> Linker -> Input
9. Add `ittiasql.lib` to the Additional Dependencies property.
10. Click OK to close the project settings dialog.
11. For any project that uses the C++ API, follow these additional steps:
 - a. Add the project `$DBDIR\win32\dbcppapi\dbcppapi.dsp` to the workspace, using the following menu item:
File -> Add -> Existing Project
Answer Yes when asked whether you should convert and open this project.
 - b. Include the `dbcppapi` project as a dependency for your project. Select your project, then go to the menu item:
Project -> Project Dependencies
Check `dbcppapi` in the list that is presented and click OK.

- c. Open the settings for your project again and select the property page:
Configuration Properties -> C++ -> Code Generation
- d. Select Release from the Configuration list. Set the Runtime Library property to Multi-threaded DLL (/MD).

Note

This step is necessary because all projects must be compiled using the same runtime library. Alternatively, you can change the Runtime Library setting for the dbcppapi project to match the desired setting for your project.

- e. Select Debug from the Configuration list. Set the Runtime Library property to Multi-threaded Debug DLL (/MDd).
12. Add application source code files to the project. You can use the C++ example code to get started.
 13. Copy `ittiasql.dll` from `$DBDIR\win32\lib` to the project directory.
 14. Build and run your application.

3.3.3. Microsoft eMbedded Visual C++ 4.0

Add ITTIA DB to an eMbedded Visual C++ Project:

1. Open eMbedded Visual Studio and create a new Win32 Application project.
2. Open the settings for your project from the menu:
Project -> Settings
3. Select All Configurations from the Settings For: list.
4. Select the property page:
Project Settings -> C/C++ -> Category: Preprocessor
5. Add `$DBDIR\win32\include` to the Additional Include Directories property.
6. Select the property page:
Project Settings -> Linker -> Category: Input
7. Add `$DBDIR\win32\lib` to the Additional Library Directories property.
8. Add `ittiasql.lib` to the Object/library modules: property.
9. Click OK to close the project settings dialog.
10. For any project that uses the C++ API, follow these additional steps:

- a. Add the project `$DBDIR\win32\dbcppapi\dbcppapi.vcp` to the workspace, using the following menu item:

Project -> Insert Project into Workspace

Answer Yes when asked whether you should convert and open this project.

- b. Include the `dbcppapi` project as a dependency for your project. Select your project, then go to the menu item:

Project -> Dependencies

Check `dbcppapi` in the list that is presented and click OK.

11. Add application source code files to the project. You can use the C++ example code to get started.
12. Copy `ittiasql.dll` from `$DBDIR\win32\lib` to the project directory.
13. Build and run your application.

3.3.4. GNU gcc

On the command line:

1. When compiling source files, add ITTIA DB's `include` directory to the include path:

```
g++ -I$DBDIR/include main.cpp -o main.o
```

2. Include the ITTIA DB library and OS portability library when linking object files together into an executable program:

```
g++ -L$DBDIR -ldbcppapi -littiasql -loslib main.o -o program
```

If only the C API is needed, the C++ API library can be omitted from the above command by removing the `-ldbcppapi` option. While `gcc` can be used to compile source code files, `g++` should still be used for linking.

1. When compiling source files, add ITTIA DB's `include` directory to the include path:

```
gcc -I$DBDIR/include main.c -o main.o
```

2. Include the ITTIA DB library when linking object files together into an executable program:

```
g++ -L$DBDIR -littiadb -loslib main.o -o program
```

Tip

The `Makefile` included with the example program can also be used as a template for automating the build process.

3.4. Included Example Programs

ITTIA DB includes an example **phonebook** program to demonstrate some of its capabilities. The example program uses a simple console-based interface so as not to distract from the basic principles of using ITTIA DB. The phonebook example contains two C++ classes: `PhoneBook`, which handles database operations, and `PhoneBookConsoleApp`, which acts as a simple controller for the application.

A similar example program based on the C API named **phbook** is also included. This example shares the same schema as the C++ example program and can be used to access the same database files.

Both example programs demonstrate how to:

- Create and open a new database
- Define a database schema
- Insert, update, and delete records
- Search, scan, and join database tables
- Close a database

3.4.1. Microsoft Visual C++

To build and run the example program in Visual C++ 6:

1. Open the workspace `$DBDIR\examples\examples.dsw`. This workspace can also be accessed from the Start Menu icon for "Example project (VC++ 6)".
2. Use Set Active Configuration... from the Build menu to select an example.
3. Select Execute phonebook.exe from the Build menu.

3.4.2. Microsoft Visual Studio

To build and run the example program in Visual Studio 2005:

1. Open the workspace `$DBDIR\examples\examples.dsw`. This workspace can also be accessed from the Start Menu icon for "Example project (VS2005)".
2. Click on an example project in Solution Explorer. Go to the Project menu and select Set as StartUp Project.
3. Select Start Without Debugging from the Debug menu.

3.4.3. Microsoft eMbedded Visual C++

To build and run the example program in Microsoft eMbedded Visual C++:

1. Open the workspace `$DBDIR\examples\examples.vcw`.
2. Select Active WCE Configuration on the Workspace Configuration Tool-Bar.
3. Select Active Target Configuration on the Workspace Configuration Tool-Bar.
4. Select Default Device on the Workspace Configuration Tool-Bar.
5. Select Execute `phonebook.exe` from the Build menu.

Note

The required dll files for these examples must be copied to the device or emulator. If eMbedded Visual Studio does not copy these dlls to the target environment, you must copy them manually using the Remote File Viewer tool.

3.4.4. GNU gcc

To build and run the example program with gcc:

1. Change to the directory `$DBDIR/example`
2. Run **make**
3. Run **./phonebook** to start the C++ API example program.
4. Run **./phbook** to start the C API example program.

3.5. Multi-threading

ITTIA DB allows multiple threads within the same application process to access the database concurrently. See Multi-threaded shared access for more information on accessing the database from multiple threads.

Warning

Multithreading is only supported in the shared access version of ITTIA DB. The single user version is not thread safe and requires that all database operations occur in a single thread.

3.6. Database Files

ITTIA DB stores all data in a single file, including the system catalog, data, and keys. The transaction journal is stored in a separate log file that should not be deleted. A database file can be created at any time, either when an application is run for the first time, or when the application is packaged. The application can even treat database files as a custom document format, allowing the user to create as many databases as needed.

Chapter 4. SQL

4.1. Introduction to SQL

Structured Query Language, or SQL, is a standard database interface that uses query strings to access and modify data. A query string can be embedded in software code to simplify complex tasks or entered by a trusted user to perform advanced reporting. SQL is a powerful a development tool because it allows a developer to quickly inspect the contents of a database without writing any additional program code.

ITTIA DB includes the **ittiasql** utility to execute SQL statements through a command-line interface. Source code for this utility is included with the other examples. See **ittiasql** for usage information.

SQL is not available in all versions of ITTIA DB. See [Linking ITTIA DB into an application](#) for information on which libraries include support for SQL.

Tip

While SQL has many advantages, because SQL statements are parsed and executed at run-time, they require some additional overhead to use. All SQL operations can be performed directly with ITTIA DB's C and C++ APIs for best performance. Many operations, such as inserting data from variables and transaction control, require less code to perform directly than with SQL.

4.1.1. Creating Tables and Inserting Values

The **ittiasql** utility is an easy way to learn about ITTIA DB. The following session creates a database file and initializes the schema.

Example 4.1. CREATE TABLE

```
ittiasql version 2.5
Copyright (C) 2005-2007 ITTIA. All rights reserved.

SQL statements end with a semi-colon (;). Interactive commands start with
a period (.) and do not end with a semi-colon.

Type '.help' for a list of interactive commands. Type '.exit' to quit.
$ .create example.db
example.db$ create table contact (
    -$   id uint64, name nvarchar(50), ring_id uint64,
    -$   picture_name varchar(50), picture blob
    -$ );
example.db$ create unique index by_id on contact (id);
example.db$ create index by_name on contact (name);
example.db$ create table phone_number (
    -$   contact_id uint64, number ansistr(20),
    -$   type uint64, speed_dial sint64
    -$ );
example.db$ create index by_contact_id on phone_number (contact_id);
example.db$ create sequence contact_id start with 1;
```

After tables are created, sample data can be inserted as part of a transaction.

Example 4.2. INSERT INTO

```

example.db$ start transaction;
example.db$ insert into contact (id, name)
           -$ values (next value for contact_id, 'Bob');
example.db$ insert into phone_number (contact_id, number, type, speed_dial)
           -$ values (current value for contact_id, '555-5555', 0, 5);
example.db$ insert into phone_number (contact_id, number, type, speed_dial)
           -$ values (current value for contact_id, '555-6666', 1, null);
example.db$ commit;

```

Tip

The schema cannot be modified while a transaction is pending. Commit or rollback before creating additional tables.

4.1.2. Projection and Selection

Data is read from a table using the SELECT statement. SELECT can be used to read the entire contents of a table, or limit the results to only columns and rows of interest.

Projection limits the number of columns that are read, and is accomplished by giving SELECT a list of columns to include in the result. The list of columns can also contain expressions.

Selection limits the number of rows that are read, and is accomplished by including a WHERE clause with criteria that each row must satisfy. Rows that do not satisfy the criteria are not included in the results.

Example 4.3. SELECT

```
example.db$ select * from contact;
```

```

+---+-----+
| ID | NAME | RING_ID | PICTURE_NAME | PICTURE |
+---+-----+
| 1  | Bob  | NULL    | NULL          | NULL    |
+---+-----+

```

```
example.db$ select id, name from contact;
```

```

+---+-----+
| ID | NAME |
+---+-----+
| 1  | Bob  |
+---+-----+

```

```
example.db$ select * from phone_number where type = 0;
```

```

+-----+-----+-----+-----+
| CONTACT_ID | NUMBER | TYPE | SPEED_DIAL |
+-----+-----+-----+-----+
| 1          | 555-5555 | 0    | 5          |
+-----+-----+-----+-----+

```

4.1.3. Joining Related Tables

Tables are related when they each have columns that contain the same data. Related tables can be joined together by matching the related columns to create rows that contain data from both tables.

Example 4.4. SELECT ... JOIN

```
example.db$ select name, number, type, speed_dial
           -$ from contact
           -$ join phone_number on id = contact_id;
```

```
+-----+-----+-----+-----+
|NAME|NUMBER|TYPE|SPEED_DIAL|
+-----+-----+-----+-----+
|Bob |555-5555|0  |5          |
+-----+-----+-----+-----+
```

Joins are described in detail in Section 4.2.2.1, “FROM Clause”.

4.1.4. SQL Query Parameters

Literal values in an SQL statement can be replaced by parameters. This feature allows an application to separate variable data from the logic of the query, improving both performance and security.

ITTIA DB reads parameter values directly from application memory without parsing or unnecessary conversion. This avoids the overhead of converting numeric values to strings and then back when the query is executed. Frequently-used statements are automatically cached by the database and will execute more quickly on subsequent runs. This performance benefit is only available for identical statements executed multiple times, but parameter values for that statement can be changed.

When string literals are inserted directly into an SQL statement, the application is susceptible to SQL injection attacks. A carefully designed string can contain SQL code that modifies the meaning of the SQL statement, allowing the attacker to execute arbitrary SQL commands. Fortunately, a fully parameterized query is completely impervious to this kind of attack.

Important

Use of parameters is highly recommended in all application queries. Never construct an SQL statement string that contains values supplied by the user or from a source outside the program's control.

To use parameters, replace literal values with the ? placeholder, as shown in Example 4.5, “Positional SQL Query Parameters”. In the **ittiasql** utility, values for each parameter are then input one at a time. In application source code, parameter values can be set through the API before the query is executed.

Use the ? placeholder to order parameters by position.

Example 4.5. Positional SQL Query Parameters

```
example.db$ insert into contact (id, name) values (?, ?);
param[0] (integer): 2
param[1] (string): Charley

example.db$ insert into phone_number (contact_id, number, type) values (?, ?, ?);
```

```
param[0] (integer): 2
param[1] (string): 555-4444
param[2] (integer): 0
```

```
example.db$ select * from contact where name = ?;
param[0] (string): Charley
```

ID	NAME	RING_ID	PICTURE_NAME	PICTURE
2	Charley	NULL	NULL	NULL

Parameters are identified by the order they appear in the SQL statement, starting from zero. The first ? placeholder to appear in the statement corresponds to parameter 0, the second ? placeholder corresponds to parameter 1, and so on.

The ? placeholder can only appear in a WHERE clause or a VALUES list. The data type of the parameter is inferred from the expression it is used in.

An placeholder alternative syntax is also available in ITTIA DB that provides more parameter control. The ? placeholder can be replaced with the \$<type>n placeholder. Replace *type* with the name of the column data type to use for the parameter and *n* with the parameter number. See Example 4.6, “Numbered SQL Query Parameters” for an example using this syntax.

With numbered parameters, it is possible to use the same parameter multiple places in the query. And because they are explicitly typed, numbered parameters can also appear in the SELECT list where the data type of the parameter cannot be inferred from context.

Example 4.6. Numbered SQL Query Parameters

```
example.db$ select * from contact where name = $<nvarchar>0;
param[0] (string): Charley
```

ID	NAME	RING_ID	PICTURE_NAME	PICTURE
2	Charley	NULL	NULL	NULL

4.2. SQL Language Reference

The basic unit of work in SQL is the statement. A statement can span multiple lines and any extra whitespace between keywords and symbols is ignored. In the interactive **ittiasql** utility, statements are terminated with a semi-colon. When an SQL statement is executed through the API, a semi-colon should not be used.

The following conventions are used in this document to show SQL syntax:

In this document, SQL syntax is documented in a format similar to the following:

```
symbol ::=
KEYWORD
replaceable-symbol
replaceable-keyword
```

identifier
 [optional]
 choice 1 | choice 2 | choice 3
 { choice 1 | choice 2 | choice 3 }
 repeatable,...
 {repeatable},...

Each of these lines demonstrates a convention used to explain the grammar of SQL. Square brackets ([]), curly brackets ({}), single pipes (|) and ellipsis dots (...) are part of the notation and are not used in SQL syntax.

symbol	The grammar starts with the name of a symbol and is followed by a description of the syntax for that symbol.
KEYWORD	An SQL keyword is shown in all caps. In practice, SQL keywords are case insensitive and can be used with any capitalization.
replaceable-symbol	A replaceable symbol is replaced with syntax defined for that symbol elsewhere in the documentation.
replaceable-keyword	A replaceable keyword is replaced with an SQL keyword listed in a table, such as Table 4.2, “SQL Column Types”.
identifier	An identifier is used to identify a table, column, or other schema object. Identifiers must start with a letter and can contain letters, numbers, and the underscore character (_). Identifiers are limited to 32 characters.
optional	Optional syntax is enclosed in square brackets.
choice	Multiple choices are separated by the pipe () symbol. Only one choice should be used.
repeatable	In SQL, many items are repeatable, in which case items are separated by commas. If only one item is given, no comma is used.

Table 4.1, “SQL Reserved Keywords” lists all keywords that are reserved for the SQL language. Do not use these names for table, column or other identifiers.

add	all	alter
and	ansistr	as
asc	avg	between
bigint	blob	by
case	cast	coalesce
column	commit	committed
completion	count	create
currency	current	date
datetime	delete	desc
distinct	drop	else
end	except	float
float32	float64	for
forced	from	group

in	index	insert
int	integer	intersect
into	is	isolation
join	lazy	level
max	min	modify
next	not	null
on	only	or
order	read	rename
repeatable	rollback	select
sequence	serializable	set
sint16	sint32	sint64
sint8	smallint	start
sum	table	then
time	timestamp	tinyint
to	transaction	uint16
uint32	uint64	uint8
uncommitted	union	unsigned
update	utf16str	utf32str
utf8str	value	values
varchar	when	where
with	work	write

Table 4.1. SQL Reserved Keywords

4.2.1. Column Types

ITTIA DB can store data in a wide range of types. Every column has a data type, which limits the kind of data that can be stored in that column. This ensures that data is always in the expected format when retrieved from the database and imposes a hard limit on the amount of storage needed for each row.

Data types are listed in Table 4.2, “SQL Column Types”. Each data type has one or more SQL column type names, which can be used interchangeably. The column type name is used when a table is created or altered. See Section 4.2.7, “Schema Definition” for more information.

SQL Column Type	C++ Class	C Data Type	Maximum Storage Size
tinyint, sint8	IntegerField	int8_t	1 byte
smallint, sint16	IntegerField	int16_t	2 bytes
integer, int, sint32	IntegerField	int32_t	4 bytes

SQL Column Type	C++ Class	C Data Type	Maximum Storage Size
bigint, sint64	IntegerField	int64_t	8 bytes
uint8	IntegerField	uint8_t	1 byte
uint16	IntegerField	uint16_t	2 bytes
uint32	IntegerField	uint32_t	4 bytes
uint64	IntegerField	uint64_t	8 bytes
float32	FloatField	db_float32_t	4 bytes
float, float64	FloatField	db_float64_t	8 bytes
currency	CurrencyField	db_sint32_t	4 bytes
varchar(<i>n</i>), ansistr(<i>n</i>)	StringField	db_ansi_t, char *	1/2 page size
utf8str(<i>n</i>)	WStringField	db_utf8_t, char *	1/2 page size
nvarchar(<i>n</i>), utf16str(<i>n</i>)	WStringField	db_utf16_t	1/2 page size
utf32str(<i>n</i>)	WStringField	db_utf32_t	1/2 page size
date	DateTimeField	db_date_t	6 bytes
time	DateTimeField	db_time_t	6 bytes
datetime	DateTimeField	db_datetime_t	12 bytes
timestamp	DateTimeField	db_timestamp_t	16 bytes
blob	BlobField	db_blob_t	database size

Table 4.2. SQL Column Types

Tip

For each column type, the C API provides a platform-independent C variable type that best matches the size and format used in the database. For best performance, the corresponding C data type should be used, but some column types can be bound to other C data types as well. See Section 6.2.1, “Column Types” for more information.

Field classes in the C++ API can be cast to a variety of native types. See Section 5.2.2, “Column Types” for more information.

4.2.1.1. Numeric Types

Integer types are signed or unsigned, indicated by a "s" or "u" prefix to the type name. Numeric types also have a suffix that indicates the number of bits allocated to the data type. Type names are listed in Table 4.2, "SQL Column Types".

Integer types can be mixed in expressions and are always converted to the largest type.

ITTIA DB provides two floating-point types: float32, and float64. The float type is an alias for float64.

Floating-Point Type	Maximum Value
float32	3.4028234×10^{38}
float64	$1.7976931348623157 \times 10^{308}$

ITTIA DB also includes a 32-bit currency data type that has a fixed precision of 2 decimal digits. When a currency value is accessed through the C or C++ API, the value is stored in a sint32_t variable that must be divided by 100 to obtain the correct value stored in the database.

4.2.1.2. Charater String Types

ITTIA DB supports four character formats: ANSI, UTF-8, UTF-16, and UTF-32. When a character column is created, the maximum size must be provided as parameter *n*, though when data is stored, only as much space as is needed will be used.

Unicode is the best character format, and should be used whenever possible. The Unicode encodings supported by ITTIA DB # UTF-8, UTF-16, and UTF-32 # are each able to store the entire range of Unicode characters. ITTIA DB will automatically convert between Unicode encodings when a database column is bound to a variable, so always select the encoding that provides the most compact representation for the column type. UTF-8 is best for Latin text and many European languages, while UTF-16 is best for Asian languages.

To store a string in UTF-8 encoding, use the utf8str type. Similarly, use utf16str for UTF-16 encoding and utf32str for UTF-32 encoding. The type nvarchar is an alias for utf16str.

For Unicode encodings, the maximum size is given in code units. Each Unicode character requires one or more code units when encoded in UTF-8 or UTF-16.

Encoding	Code Unit
UTF-8	1 byte
UTF-16	2 bytes
UTF-32	4 bytes

As an alternative to Unicode, ITTIA DB also supports the ANSI character format with the ansistr type, which is also aliased as varchar. ITTIA DB assumes that ANSI character strings are encoded using the current locale, which is a platform-dependent setting. The most commonly encoding is ISO 8859-1.

String literals are formed by enclosing text in quotes and are of type varchar.

4.2.1.3. Date and Time Types

ITTIA DB uses a common string format for all date and time types.

Date and time literals are formed by following the name of the data type with a string literal in a specific format. The literal formats accepted by ITTIA DB are listed in Table 4.3, "Date and Time Literal Formats".

SQL Column Type	Literal Format	Literal Examples
date	date 'YYYY-mm-dd'	date '2008-05-23' date '2008-5-23'
time	time 'HH:MM:SS'	time '10:12:05' time '3:15:00' time '23:59:59'
datetime	datetime 'YYYY-mm-dd HH:MM:SS'	datetime '2008-05-23 10:12:05'
timestamp	timestamp 'YYYY-mm-dd HH:MM:SS'	timestamp '2008-05-23 10:12:05'
timestamp	timestamp 'YYYY-mm-dd HH:MM:SS.ffffff'	timestamp '2008-05-23 10:12:05.123456'

Table 4.3. Date and Time Literal Formats

4.2.1.4. NULL Values

ITTIA DB has a special value called NULL. When a value is NULL, it indicates that the value is unknown or has not been set. Every column type supports NULL. If a value is not specified for a column when a row is inserted, the default value is NULL.

Comparison operators handle NULL as a special case. When any value is compared with NULL, the result is false. To determine whether a value is NULL, use the expression "*value* IS NULL" or "*value* IS NOT NULL".

Tip

In C and C++, NULL is a pointer type equal to 0. In SQL, 0 and NULL are different values.

4.2.2. SELECT Queries

```

SELECT
[ ALL | DISTINCT ] field,...
[FROM table-expression]
[WHERE search-condition]
[GROUP BY field,... ]
[ORDER BY order-by]
[ { UNION | EXCEPT | INTERSECT } [ DISTINCT | ALL ] select-statement ...]
field ::=
[table-name.]column-name [[AS] alias-name]
| [table-name.]*
| term [[AS] alias-name]
order-by ::=
{ field [ ASC | DESC ] },...
column-list ::=
column-name,...
term ::=
term { + | - | * | / | % | || } term
| { + | - } term
| (term)
| function(arguments,...)
| { NEXT | CURRENT } VALUE FOR sequence-name

```

```
| CAST (term AS column-type)  
| CASE {WHEN search-condition THEN term}... [ELSE term] END  
| CASE term {WHEN term THEN term}... [ELSE term] END  
| literal-value  
| [table-name].column-name
```

SELECT is used to read data from the database. In its most basic form, SELECT will read the entire contents of a table. The following statement returns the entire contents of the contact table.

```
select *  
  from contact
```

The results can be restricted to only certain columns by naming them in the *field* list.

```
select id, name  
  from contact
```

Columns can be renamed when selected with the AS keyword.

```
select id as contact_id, name as contact_name  
  from contact
```

Tip

The AS keyword is optional and can be omitted, but is recommended for clarity.

The results of a query may contain multiple rows. To eliminate duplicates, use the DISTINCT keyword. The following query identifies all contacts with at least one phone number:

```
select distinct contact_id from phone_number
```

The CAST keyword is used to convert a term to a different data type.

```
select 'The ID is: ' || cast (id as varchar)  
  from contact
```

The CASE keyword provides conditional logic. There are two forms of CASE. One checks a series of search conditions, similar to an if statement in C. The other form compares two terms and uses the first match. The following example shows both forms:

```
select number,  
       case  
         when speed_dial >= 0 and speed_dial <= 9 then speed_dial  
         else null  
       end as speed_dial,  
       case type  
         when 1 then 'Mobile'  
         when 2 then 'Work'  
         when 3 then 'Fax'  
         when 4 then 'Pager'  
         else number  
       end as type  
  from phone_number
```

The sequence expressions NEXT VALUE FOR and CURRENT VALUE FOR are described in `nextval(str)` and `currval(str)`.

4.2.2.1. FROM Clause

```

FROM
table-expression
table-expression ::=
table-reference
| table-expression [join-type] JOIN table-reference [join-condition]
| table-expression NATURAL [join-type] JOIN table-reference [join-condition]
| table-expression,...
table-reference ::=
table-name [[AS] correlation-name]
join-type ::=
CROSS | INNER | LEFT OUTER | RIGHT OUTER | FULL OUTER
join-condition ::=
ON search-condition
| USING (column-list)

```

The FROM clause specifies which tables to include in the query. Related tables can be joined together by matching the related columns to create rows that contain data from both tables. The following statement selects all related rows from the `contact` and `phone_number` tables:

```

select name, number
  from contact
  join phone_number on id = contact_id

```

These two tables are related by the `id` column in `contact` and the `contact_id` column in `phone_number`. The search-condition following the ON keyword uses the same syntax as the WHERE clause discussed in Section 4.2.2.2, “WHERE Clause”. The join clause can be repeated to add more tables to the query.

Column names can be prefixed with a table name for clarity and to avoid ambiguity. The following query is equivalent to the previous query:

```

select contact.name, phone_number.number
  from contact
  join phone_number on contact.id = phone_number.contact_id

```

Tables can be given an alternate *correlation-name* to identify columns in the query, as in the following example:

```

select c.name, p.number
  from contact as c
  join phone_number as p on c.id = p.contact_id

```

The correlation name can also be used to join multiple instances of the same table. The following query lists alternate phone numbers for each phone number in the database by joining the `phone_number` table with itself:

```

select p1.number as number, p2.number as alternative_number
  from phone_number as p1
  join phone_number as p2 on p1.contact_id = p2.contact_id and p1.number <> p2.number

```

The FROM clause is optional. When the FROM clause is omitted, the select list must contain only literal values, which will be returned as a single row.

4.2.2.2. WHERE Clause

```

WHERE
search-condition

```

```

search-condition ::=
expression
expression ::=
expression { = | <> | > | < | >= | <= } expression
| expression OR expression
| expression AND expression
| NOT expression
| expression IS [NOT] NULL
| IN ( term,... )
| term BETWEEN term AND term
| term

```

The WHERE clause is optional, and if given will limit the rows in the result set to those satisfied by the *search-condition*. The *search-condition* can be a simple comparison or a complex expression.

The following query searches for a contact named Bob. Since the name column is indexed, this search will always be fast.

```

select *
  from contact
  where name = 'Bob'

```

=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Table 4.4. SQL Comparison Operators

4.2.2.3. GROUP BY Clause

```

GROUP BY
term,...

```

The GROUP BY clause is used to summarize the results of a query by grouping multiple rows into a single row in the result set. Grouping occurs after tables have been joined and filtered by the FROM and WHERE clauses.

The following statement groups phone numbers by contact_id, summarizing the number of phone numbers recorded for each contact.

```

select contact_id, count(number)
  from phone_number
  group by contact_id

```

This query can be extended to include the contact name by joining with the contact table and adding the name column to the grouping list.

```

select name, contact_id, count(number) as number_count
  from phone_number
  join contact on contact_id = id
  group by name, contact_id

```

When grouping is used, the SELECT field list can only contain terms that are listed in the GROUP BY clause, unless enclosed in an aggregate function.

Aggregate functions return a single value from an expression that is evaluated over multiple rows. Aggregate functions are listed in Table 4.5, “SQL Aggregate Functions”.

Name	Argument Type	Return Value
min	Any type	The lowest value
max	Any type	The highest value
sum	Number	Sum of values
avg	Number	Average of values
count	Any type or *	Number of rows

Table 4.5. SQL Aggregate Functions

Aggregate functions can also be used to group all rows into a single row in the result set when the GROUP BY clause is omitted. The SELECT field list can only contain aggregate and literal terms when any aggregate function is used without a GROUP BY clause.

4.2.2.4. ORDER BY Clause

```
ORDER BY
{ field [ ASC | DESC ] },...
```

By default, rows are returned in an arbitrary order. Use the ORDER BY clause to sort the results.

```
select *
  from contact
  order by name
```

If multiple columns are named, results are sorted by the first column and then by the following columns only when values in the first column are equal. By default, columns are sorted in ascending order, but the DESC modifier can be applied to any column to change its direction.

```
select *
  from phone_number
  order by contact_id asc, type desc
```

4.2.2.5. UNION, EXCEPT, and INTERSECT

```
select-statement { UNION | EXCEPT | INTERSECT } [ DISTINCT | ALL ] select-statement
```

SELECT statements can be combined through the set operations UNION, EXCEPT, and INTERSECT. Combined SELECT statements must all have the same types of fields, listed in the same order.

The UNION operation returns all rows from both statements. The EXCEPT operation returns only those rows from the first query that are also present in the second. The INTERSECT operation returns only rows that are present in the results of both statements. Two rows match only when all fields are identical.

An SQL statement can use multiple set operations to merge results from more than two SQL queries. UNION has the highest precedence. EXCEPT and INTERSECT have equal precedence and are evaluated from left to right.

```
select * from a
union
```

```

select * from b
union
select * from c
except
select * from d
union
select * from e

```

The result of this example is grouped as follows: (a union b) except (d union e).

Subqueries can be used to control group set operations.

```

select *
  from (
    select * from a
    intersect
    select * from b
  )
union
select *
  from c

```

The result of this example is grouped as follows: (a intersect b) union c.

Set operations can also include the `DISTINCT` or `ALL` keyword to control how duplicate values are handled in the result. By default, set operations are `DISTINCT`, meaning that each row in the result will be unique. The `ALL` keyword permits duplicate rows to appear in the result.

4.2.3. Natural Numbers Virtual Table

ITTIA DB SQL statements can use the natural numbers virtual table, `$nat(count)`, to select from a list of integers. The resulting table contains a single column named "n" listing the first `count` natural numbers.

```
$ select * from $nat(10);
```

```

+--+
|N|
+--+
|0|
|1|
|2|
|3|
|4|
|5|
|6|
|7|
|8|
|9|
+--+

```

```
$ select n + 1, 'Number ' || cast(n + 1 as varchar) from $nat(4);
```

```

+---+-----+
|$0|$1|
+---+-----+
|1| Number 1|
|2| Number 2|
|3| Number 3|
|4| Number 4|
+---+-----+

```

```
$ insert into some_table (n) select * from $nat(1000);
```

4.2.4. Built-in Functions

Several built-in functions are available to perform common mathematical and string operations. Available functions are listed in Table 4.6, “SQL Built-in Functions”. Functions can be used in the SELECT field list, JOIN ON condition, WHERE clause, GROUP BY clause, ORDER BY clause, and INSERT INTO ... VALUES list.

Name	Argument Types	Returns
<code>abs(<i>n</i>)</code>	Number	Absolute value of <i>n</i> .
<code>sign(<i>n</i>)</code>	Number	Sign of <i>n</i> : -1 for negative, 0 for zero, +1 for positive.
<code>mod(<i>n</i>, <i>m</i>)</code>	Number	Remainder of <i>n</i> divided by <i>m</i> .
<code>least(<i>n</i>, ...)</code>	Number	Lowest value listed. Result is NULL if any value is NULL.
<code>greatest(<i>n</i>, ...)</code>	Number	Greatest value listed. Result is NULL if any value is NULL.
<code>trim(<i>str</i>)</code>	String	<i>str</i> with leading and trailing spaces removed.
<code>ltrim(<i>str</i>)</code>	String	<i>str</i> with leading spaces removed.
<code>rtrim(<i>str</i>)</code>	String	<i>str</i> with trailing spaces removed.
<code>substr(<i>str</i>, <i>pos</i>, <i>len</i>)</code>	String, integer, integer	Substring of <i>str</i> of length <i>len</i> , starting from <i>pos</i> , where the first character is position 1.
<code>substr(<i>str</i>, <i>pos</i>)</code>	String, integer	Substring of <i>str</i> , starting from <i>pos</i> , where the first character is position 1, to the end of the string.
<code>concat(<i>str</i>, ...)</code>	String	Concatenation of listed strings.
<code>coalesce(<i>a</i>, ...)</code>	Any type	First value listed that is not NULL.
<code>nextval(<i>str</i>)</code>	String	Current value for the sequence identified by <i>str</i> .
<code>currval(<i>str</i>)</code>	String	Next value for the sequence identified by <i>str</i> .

Table 4.6. SQL Built-in Functions

`abs(n)` Absolute value of *n*.

```
$ select abs(1);
1
$ select abs(-10);
-10
$ select abs(-3.1415);
-3.1415
```

`sign(n)` Sign of *n*: -1 for negative, 0 for zero, +1 for positive. The result is the same type as the argu-

ment.

```
$ select sign(8);  
1
```

```
$ select sign(0);  
0
```

```
$ select sign(-16.9);  
-1.0
```

`mod(n, m)` Remainder of *n* divided by *m*.

```
$ select mod(23, 10);  
3
```

```
$ select mod(23, -10);  
3
```

```
$ select mod(-23, 10);  
3
```

```
$ select mod(17, 8);  
3
```

`least(n, ...)` Lowest value listed. Result is NULL if any value is NULL.

Each value can be a different numeric type. The return value will be the most general type listed.

```
$ select least(60, 22, 19, 37);  
19
```

```
$ select least(5, 10, null, -1);  
NULL
```

```
$ select least(-5.1, -10);  
-10.0
```

`greatest(n, ...)` Greatest value listed. Result is NULL if any value is NULL.

Each value can be a different numeric type. The return value will be the most general type listed.

```
$ select greatest(60, 22, 19, 37);  
60
```

```
$ select greatest(5, 10, null, -1);  
NULL
```

```
$ select greatest(-5.1, -10);  
-5.1
```

`trim(str)` *str* with leading and trailing spaces removed.

```
$ select trim(' Hello World ');  
'Hello World'
```

`ltrim(str)` *str* with leading spaces removed.

```
$ select ltrim(' Hello World ');
Hello World
```

`rtrim(str)` *str* with trailing spaces removed.

```
$ select rtrim(' Hello World ');
Hello World
```

`substr(str, pos, len)` Substring of *str* of length *len*, starting from *pos*, where the first character is position 1.

If *pos* is less than 1, it is treated as though it were 1.

```
$ select substr('Hello World', 1, 5);
Hello
$ select substr('Hello World', 3, 5);
llo W
```

`substr(str, pos)` Substring of *str*, starting from *pos*, where the first character is position 1, to the end of the string.

If *pos* is less than 1, it is treated as though it were 1.

```
$ select substr('Hello World', 7);
World
$ select substr('Hello World', 100);
''
```

`concat(str, ...)` Concatenate listed strings.

This function is equivalent to the string concatenation operator "||".

```
$ select concat('Hello', ' ', 'World');
Hello World
$ select 'Hello' || ' ' || 'World';
Hello World
```

`coalesce(a, ...)` Return the first value that is not NULL from a list of values.

All values must have the same data type.

```
$ select coalesce(null, 1, 2);
1
$ select coalesce(3, null, 4);
3
```

The `coalesce` function is a convenient shorthand that could otherwise be expressing using a CASE statement. Example 4.7, “`coalesce` vs. `case`” shows two equivalent queries, the first using `coalesce` and the second using a CASE statement.

Example 4.7. coalesce vs. case

```
select coalesce(a, b, c)
  from some_table

select
  case
    when a is not null then a
    when b is not null then b
    when c is not null then c
    else null
  end
  from some_table
```

`nextval(str)` Obtain the next value from the sequence generator identified by *str*. For the given sequence generator, the value returned is guaranteed to be unique and greater than any value previously returned.

Tip

`nextval` allows the name of the sequence generator to be computed by the SQL query. If the name of the sequence is already known, use the expression "NEXT VALUE FOR *sequence-name*" instead. This form can also be used in an INSERT statement's VALUES list.

```
$ create sequence myid start with 1;
$ select nextval('myid');
2
$ select next value for myid;
3
$ insert into some_table (n) values (next value for myid);
```

`currval(str)` Obtain the current value from the sequence generator identified by *str*.

Tip

`currval` allows the name of the sequence generator to be computed by the SQL query. If the name of the sequence is already known, use the expression "CURRENT VALUE FOR *sequence-name*" instead. This form can also be used in an INSERT statement's VALUES list.

```
$ create sequence myid start with 1;
$ select currval('myid');
1
$ select current value for myid;
1
$ select next value for myid;
2
$ select current value for myid;
```

2

```
$ insert into some_table (n) values (current value for myid);
```

4.2.5. Data Manipulation

Three kinds of modifications to database rows are available: insert new rows, update existing rows, and delete rows. Any modification of the database will start a transaction, which must be committed before changes will become persistent. For more information, see Section 4.2.6, “Transactions”.

4.2.5.1. INSERT

```
INSERT INTO
table-name [( column-name, ... )]
VALUES value, ...
| select-statement
```

INSERT INTO will insert zero, one or more rows into a single table. By default, values must be given for all columns in the order they were created.

Optionally, a list of column names can be used. Values should be given in the same order as the columns in the list. Columns not listed will be assigned the value NULL by default.

To insert literal values for a single row, use the VALUES keyword followed by a list of values. The following query adds a new phone number to the first contact:

```
insert into phone_number (contact_id, number, type, speed_dial)
values (1, '555-7777', 0, null)
```

Since speed_dial is left NULL, it can be omitted from column list:

```
insert into phone_number (contact_id, number, type)
values (1, '555-7777', 0)
```

To insert one or more rows, use a SELECT statement instead to obtain rows from another table. The following query copies all phone numbers from contact 1 to contact 2:

```
insert into phone_number (contact_id, number, type, speed_dial)
select 2, number, type, speed_dial
from phone_number
where contact_id = 1
```

4.2.5.2. UPDATE

```
UPDATE table-name [AS alias]
SET { column-name = value, ... }
[WHERE search-condition]
```

UPDATE modifies the value of zero, one or more rows in a single table. Columns and their new values are given in a list following the SET keyword, and by default all rows are modified. Values can be literal or expressions containing other columns from the table.

To limit changes to one or more rows, supply a search condition in the form of a WHERE clause with the same syntax as when used with SELECT. For more information, see Section 4.2.2.2, “WHERE Clause”.

The following statement updates a contact's name and changes their ring tone:

```
update contact
  set name = 'Bobby', ring_id = 8
  where name = 'Bob'
```

4.2.5.3. DELETE

```
DELETE FROM
 [AS alias]
[WHERE search-condition]
```

DELETE FROM removes zero, one or more rows from a single table. By default, all rows in the table are deleted.

If a WHERE clause is provided, only rows matching the search criteria will be deleted. For more information, see Section 4.2.2.2, “WHERE Clause”.

Tip

Before deleting rows, run a similar SELECT statement to test the WHERE condition.

4.2.6. Transactions

```
START TRANSACTION
[[ISOLATION LEVEL] READ COMMITTED | REPEATABLE READ | SERIALIZABLE ]
```

Database updates through SQL will automatically start a transaction so that related changes are grouped together into a single atomic operation. Changes will not become permanent or available to other users of the database until the transaction is committed.

A transaction can be started manually with the START TRANSACTION statement. When the database is shared, this can also be used to select an isolation level, as described in Section 7.4, “Isolation Levels”. If no isolation level is selected, the default isolation level set in the API will be used.

The schema cannot be modified while a transaction is active.

```
COMMIT
[ TRANSACTION | WORK
[ LAZY COMPLETION | GROUP COMPLETION | FORCED COMPLETION ]]
```

The COMMIT statement finishes a transaction.

A completion mode can be used to control when the changes are written to disk or other storage media immediately. FORCED COMPLETION ensures that all changes are persistent before the statement can return. GROUP COMPLETION and LAZY COMPLETION will return immediately and delay writing until later. GROUP COMPLETION waits for several transactions to be committed before writing changes to disk. LAZY COMPLETION will only write to disk when more memory is needed or when a disk flush is explicitly requested through the API. Changes are always written when the database is closed.

If no completion mode is given, the default completion mode selected in the API is used.

```
ROLLBACK
[ TRANSACTION | WORK
[ LAZY COMPLETION | GROUP COMPLETION | FORCED COMPLETION ]]
```

To cancel changes made during an active transaction, use the ROLLBACK statement instead of COMMIT. The transaction is aborted and the database is restored to its former state.

Tip

The C and C++ APIs also provide functions to control transactions without using SQL.

4.2.7. Schema Definition

Schema definition statements alter the layout of the database. These statements cannot be used when a transaction is active and cannot be rolled back.

4.2.7.1. CREATE TABLE

```
CREATE TABLE
table-name (
{ column-name column-type },...
)
```

Create an empty table with listed columns. Each column is assigned a name and a type from Table 4.2, “SQL Column Types”.

The following statement creates a contact table with columns for a unique identifier, a name encoded in Unicode, a ring tone identifier, a picture file name, and picture data.

```
create table contact (
  id uint64, name nvarchar(50), ring_id uint64,
  picture_name varchar(50), picture blob
)
```

4.2.7.2. CREATE INDEX

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

Create an index on one or more columns in a table. Indexed columns can be searched faster than non-indexed columns, especially when there are a large number of rows in the table. Multiple columns can be included in the same index to improve performance on queries that search multiple columns at once. Queries that only search on one column can only utilize the first field in each index, so more than one index is usually needed to support a variety of queries.

The following statement creates an index on contact names:

```
create index by_name on contact (name)
```

An index can also be set as UNIQUE, which prevents duplicate values for the indexed columns. The contact identifier uses a unique index to ensure that contact identifiers are not reused:

```
create unique index by_id on contact (id)
```

4.2.7.3. CREATE SEQUENCE

```
CREATE SEQUENCE
sequence-name START [WITH] number
```

A sequence generator provides unique numbers to use as identifiers in the database or in the application. A sequence generator can be assigned a non-negative starting value when created.

```
create sequence contact_id start with 1
```

4.2.7.4. ALTER TABLE

```
ALTER TABLE
table-name
ADD COLUMN column-name column-type
```

| DROP COLUMN *column-name*

Columns can be added and removed from a table using the ALTER TABLE statement. A column can only be dropped if it is not used by any index and is not the only column in the table.

4.2.7.5. DROP TABLE

DROP TABLE
table-name

DROP TABLE removes a table and all of its contents from the database, including indexes. This action cannot be undone.

4.2.7.6. DROP INDEX

DROP INDEX
index-name ON *table-name*

DROP INDEX removes an index from a table. Because the index does not contain any irreplaceable data, it can be recreated at a later time.

4.2.7.7. DROP SEQUENCE

DROP SEQUENCE
sequence-name

DROP SEQUENCE removes a sequence generator from the database.

Chapter 5. The C++ API

5.1. Getting started

5.1.1. Creating and Opening a Database

To use the ITTIA DB C++ API, include the header `ittia/db++.h`.

```
#include <ittia/db++.h>
```

The API is encapsulated in the `db` namespace.

A database is managed by a `Database` object. To create a database for the first time, use the object's `create()` function.

```
db::Database db;
db::StorageMode mode;
int rc;

rc = db.create("phone_book.db", mode);

if (DB_FAILED(rc)) {
    cerr << "Error creating database: " << rc << endl;
}
```

Caution

If the file already exists, `create()` will overwrite it.

To open an existing database, use the `open()` function.

```
rc = db.open("phone_book.db", mode);

if (DB_FAILED(rc)) {
    cerr << "Error opening database: " << rc << endl;
}
```

When done using the database, call `close()`.

```
db.close();
```

The database will also close automatically when the `Database` object is destroyed.

5.1.2. Error Handling

Most C++ API functions return an error code. To determine whether an operation was successful, check the return code with `DB_SUCCESS()` or `DB_FAILED()`.

```
if (DB_SUCCESS(rc)) {
    cout << "Operation successful" << endl;
}

if (DB_FAILED(rc)) {
```

```
    cout << "Operation failed with error code: " << rc << endl;  
}
```

Tip

Error codes are listed in the Error Handling section of the API Reference. The same error codes are used in both the C and C++ APIs.

5.1.3. Run-time Configuration

Various features of ITTIA DB can be configured dynamically at run-time. Changing these configurations in relation to available hardware and data access patterns can provide optimized performance.

The following options can be set by passing a `LibraryConfig` object to `Database::initialize()`:

<code>memory_mode</code>	Select a memory optimization strategy: <code>LibraryConfig::TIGHT</code> , <code>LibraryConfig::COMPACT</code> , or <code>LibraryConfig::LARGE</code> .
<code>transactions</code>	Approximate number of concurrent transactions expected.

The following options can be set in the `StorageMode` object before opening or creating a database:

<code>open_flags</code>	Select read-only access or disable logging to the journal.
<code>buffer_count</code>	Sets the number of pages to allocate for caching database pages. The total size of the buffers will be <code>buffer_count * page_size</code> .
<code>page_size</code>	Sets the page size for creating a new database. The page size of an existing database cannot be changed by setting this value.
<code>checkpoint_interval</code>	Specifies how many operations to perform between checkpoints in the journal.

See the API Reference manual for more details on these and other settings.

5.2. Database Design

5.2.1. Overview

Database design, like any software design, requires careful attention. When developing a database schema, a designer must consider how the data is going to be accessed to provide efficient access for the most common operations. Techniques for designing an optimal schema are beyond the scope of this document but ITTIA engineers will be happy to assist you.

An ITTIA DB database is a collection of tables which stores application data as rows of data values. A table is defined by its list of columns, which specifies the kind of data that is to be stored. Indexes can be defined to speed up searches for a specific row or to retrieve a range of values in sorted order.

A table can have many indexes defined, and an index can be created on more than one column. Relationships between tables can be expressed by defining a unique index on one table and including the same data values in another table.

5.2.2. Column Types

ITTIA DB can store data in a wide range of types. Every column has a data type, which limits the kind of data that can be stored in that column. This ensures that data is always in the expected format when retrieved from the database and imposes a hard limit on the amount of storage needed for each row.

Columns are created by calling a member function of the `FieldDescSet` class, as described in Section 5.2.3, “Defining the Database Schema”. Data stored in a column is accessed by binding a field class to a cursor, as described in Section 5.3, “Database Access”.

Column Type	FieldDescSet Function	C++ <code>db::Field</code> Class
sint8	<code>add_sint(name, 1)</code>	<code>IntegerField</code>
sint16	<code>add_sint(name, 2)</code>	<code>IntegerField</code>
sint32	<code>add_sint(name, 4)</code>	<code>IntegerField</code>
sint64	<code>add_sint(name)</code>	<code>IntegerField</code>
uint8	<code>add_uint(name, 1)</code>	<code>IntegerField</code>
uint16	<code>add_uint(name, 2)</code>	<code>IntegerField</code>
uint32	<code>add_uint(name, 4)</code>	<code>IntegerField</code>
uint64	<code>add_uint(name)</code>	<code>IntegerField</code>
float64	<code>add_float(name)</code>	<code>FloatField</code>
currency	<code>add_currency(name)</code>	<code>CurrencyField</code>
ansistr	<code>add_string(name, length)</code>	<code>StringField</code>
utf16str	<code>add_wstring(name, length)</code>	<code>WStringField</code>
date	<code>add_date(name)</code>	<code>DateTimeField</code>
time	<code>add_time(name)</code>	<code>DateTimeField</code>
datetime	<code>add_date(name, DB_DATETIME)</code>	<code>DateTimeField</code>
timestamp	<code>add_date(name, DB_TIMESTAMP)</code>	<code>DateTimeField</code>
blob	<code>add_blob(name)</code>	<code>BlobField</code>

Table 5.1. C++ Column Types

5.2.3. Defining the Database Schema

The database schema is defined programmatically by creating tables. Tables can be created at any time, and can be modified even after data has been entered into them.

Note

Schema updates cannot be performed inside a transaction. See Transactions.

5.2.3.1. Tables, Fields and Indexes

To create a table, first describe the table's fields and indexes using `FieldDescSet` and `IndexDescSet` objects. Then pass these objects to `Database::create_table()`.

```
db::Database db;
db::FieldDescSet fields;
db::IndexDescSet indexes;
int rc;

// ...
// Create the database
// ...

fields.add_uint("id");
fields.add_string("name");
fields.add_uint("ring_id");

indexes.add_index("id-index", db::DB_UNIQUE)
    .add_field("id");

indexes.add_index("name-index", db::DB_MULTISSET)
    .add_field("name");

rc = db.create_table("person", fields, indexes);
```

To modify the schema of an existing table, first use `describe_table()` to obtain a `FieldDescSet` and `IndexDescSet` for the current schema. Modify these objects and then call `update_table()` to update the schema. Fields are identified by name.

To remove a table from the database, including all data stored in the table, call `drop_table()`.

```
rc = db.drop_table("person");
```

Note

Calls to `create_table()`, `update_table()`, and `drop_table()` are atomic.

5.2.3.2. Sequence Generators

Sequence generators are used to generate unique, increasing integer values. A sequence generator is created with an initial starting value and will then retrieve successive numbers in the sequence. Each generated number is always greater than all previous numbers produced by the sequence generator. This provides a convenient source for unique identifiers.

Note

Sequences may contain gaps, and so should not be used in situations where numbers cannot be skipped,

such as matching with sequence numbers pre-printed on paper forms.

To create a sequence, call `create_sequence()` giving a name for the sequence and its initial value.

```
rc = db.create_sequence("person-id", 1);
```

If a sequence is no longer required, it can be dropped from the database.

```
rc = db.drop_sequence("person-id");
```

5.3. Database Access

5.3.1. Transactions

All data access operations must occur inside a transaction. A transaction is a logical group of operations that must all succeed or fail together. Changes made to the database do not become permanent until the transaction is committed. A transaction can also be aborted, which will undo any changes made since the transaction was started.

To start a transaction, call the `tx_begin()` function.

```
rc = db.tx_begin();
```

To commit the changes made during the transaction, call `tx_commit()`.

```
rc = db.tx_commit();
```

Note

When the commit function returns successfully, any modifications made within the transaction are guaranteed to be completely written to the storage media.

To cancel the changes made during the transaction, call `tx_abort()`.

```
rc = db.tx_abort();
```

5.3.2. Table Cursors

To access data in the database, ITTIA DB uses table cursors. A table cursor represents a single row in the database and is used to perform inserts, traversal, updates, and deletes.

To open a cursor, create a Table object and call the `open()` function.

```
db::Table person;  
person.open(db, "person");
```

When you are done with the cursor, call `close()` to free resources. The cursor will close itself automatically when it is destroyed, but it is best to release the cursor as soon as it is no longer needed.

```
person.close();
```

5.3.3. Inserting Data

Before inserting data, it is often useful to obtain a unique value from a sequence generator.

```
db::Sequence id_sequence;
db_uint id = 0;

id_sequence.open(db, "person-id");
id_sequence.get_next_value(id);
```

To insert data, first put the table in insert mode by calling `insert()`, then assign values to each field using the cursor's `[]` operator. These values are stored in a temporary buffer. Finally, call `post()` to actually insert the new row. `post()` copies the buffer into the table but does not commit the transaction.

```
person.insert();
person["id"] = id;
person["name"] = db::String("Bob");
person["ring_id"] = 7;
rc = person.post();
```

If, after calling `insert()` and possibly assigning values to the fields, you do not want to insert the row into the database, call the `cancel()` function instead of `post()` to clear the buffer.

Note

Data inserted into the database will not be persistent until the current transaction is committed. See Transactions.

5.3.4. Traversing a Table

To traverse a table, first select an index to define the order over which the rows will be iterated.

```
person.set_sort_order("name-index");
```

Then use `seek_first()`, `seek_next()`, and `is_eof()` to position on each row of data.

```
for (person.seek_first(); !person.is_eof(); person.seek_next()) {
    // Perform operation on current row
}
```

If order is not important, pass `NULL` to `set_sort_order` to traverse rows in table-scan order, which is arbitrary but fast.

5.3.5. Searching for a Row

To locate a specific row in the table, first choose an index to search by.

```
person.set_sort_order("name-index");
```

Then call `begin_seek()` to enter seek mode, assign search criteria to each field used by the index, and call `apply_seek()`.

```
person.begin_seek(db::DB_SEEK_EQUAL);
person["name"] = db::String("Bob");
```

```
rc = person.apply_seek();
```

The fields used as search criteria must be fields of the sort index. Furthermore, if the index contains multiple fields, the search criteria must be some or all of the first fields in the index. For example, if an index has three fields, the search criteria can be either the first field in the index, the first two fields, or all three fields.

To find additional rows that may match search criteria, call `seek_next()`.

5.3.6. Reading Data

Once the cursor has been used to select a row, the data can be read using the cursor. For each field that is needed, use the `[]` operator to get the field and call the appropriate `Field::as_*` function for the field's type.

```
db_uint id = person["id"].as_int();

db::String name = person["name"].as_string();
db_uint ring_id = person["ring_id"].as_int();

cout << "Id: " << (long) id << endl;
cout << "Name: " << name.c_str() << endl;
cout << "Ring tone id: " << (int) ring_id << endl;
```

5.3.7. Updating Data

In addition to accessing data, you can also update the current row. Call `edit()` to enter edit mode, assign new values to each field that should be changed, and call `post()` to update the row.

```
person.edit();
person["name"] = db::String("Sue");
person["ring_id"] = 3;
rc = person.post();
```

Note

As with inserts, modifications to the database will not be persistent until the current transaction is committed. See Transactions.

5.3.8. Deleting the Row

To delete the current row, call `remove()`.

```
rc = person.remove();
```

Note

The row will not be permanently removed from the database until the current transaction is committed. See Transactions.

5.3.9. Reading and Writing BLOBs

A BLOB (Binary Large Object) is a special type that can store data of arbitrarily large size. To do this, data in a BLOB field is accessed in small divisions called chunks. These chunks can be written to and read from using loops to access the entire BLOB or only a portion of the BLOB.

BLOB data is read and written directly from a `db::Table` object using the `read_blob()` and `write_blob()` functions. The table cursor must be positioned on an existing row and any pending updates to other columns must be posted with the `post()` function before the BLOB is accessed.

Here is an example of inserting a row with a BLOB field with data that is loaded from a file:

```
db::Table t;

... // Open t and position on an existing row. */

FILE *picture_file;
picture_file = fopen("read.png", "rb");

const db_len_t data_size = 1024;
char data[data_size];

int picture_field = t.find_field("picture");
int num_chunks = bytes_read = 0;

while((bytes_read = fread(data, 1, data_size, picture_file)) > 0)
{
    t.write_blob (picture_field, data_size * num_chunks, data, bytes_read);
    num_chunks++;
}

fclose(picture_file);
```

Here is an example of reading a BLOB from the database back into a file:

```
db::Table t;

... // Open t and position on an existing row. */

FILE *picture_file;
picture_file = fopen("write.png", "wb");

const db_len_t data_size = 1024;
char data[data_size];

int offset, bytes_read;
int picture_field = t.find_field("picture");
db_len_t blob_size = t.get_blob_size(picture_field);

for(offset = 0; offset < blob_size; offset += data_size)
{
    bytes_read = contact.read_blob(picture_field, offset, data, data_size);
    fwrite(data, bytes_read, 1, picture_file);
}

fclose(picture_file);
```

5.4. SQL Queries

Structured Query Language, or SQL, is a standard database interface that uses query strings to access and modify data. For the complete description of the SQL dialect used by ITTIA DB, see Chapter 4, *SQL*.

To execute an SQL statement from the ITTIA DB C++ API, create an instance of the `db::Query` class. Then use the `db::Query` object to perform the following steps:

1. Prepare an SQL statement.
2. Set query parameters (optional).
3. Execute the query.
4. Iterate through rows in the result set (optional).
5. Unexecute the query and repeat from step 2 (optional).

Steps 2 and 4 can be omitted depending on the nature of the SQL statement. For example, `INSERT` statements do not have a result set and thus do not use step 4. Schema definition statements, such as `CREATE TABLE`, can skip step 2 because they do not accept parameters.

Important

Use parameters whenever possible. Parameters are faster and more secure than embedding data directly in an SQL statement, for example using `sprintf`.

To execute a simple query with no parameters, call `db::Query::exec_direct()`, which combines the prepare and execute steps.

```
db::Query q;
int rc = q.exec_direct(db, "create table some_table (column_a integer, column_b integer)");
```

Alternately, these steps can be performed separately with the `prepare()` and `execute()` functions.

```
db::Query q;
int rc;
rc = q.prepare(db, "select column_a, column_b from some_table");
rc = q.execute();
```

If the query is a `SELECT` statement, a result set will be available after the query is executed. The result set is a read-only collection of rows with the same structure as a table. The result set is only sorted if the SQL statement contains an `ORDER BY` clause.

The `db::Query` class implements the `db::Cursor` interface, which is used to iterate over rows in a query result set or table. To iterate over the result set, use `seek_first()` to position on the first row, `seek_next()` to advance to the next row, and check `is_eof()` to determine when all rows have been processed. An SQL query can only be traversed forward and cannot be reset to the first row after `seek_next()` has been called.

Each row in the result set contains one or more fields that correspond to the columns and expressions in the `SELECT` list. Fields are identified by the order that they appear in the `SELECT` list, starting from zero.

To access the value of a field, first locate the `db::Field` class listed in Table 5.1, “C++ Column Types” that corresponds to the column’s data type. After the query has been executed, construct an instance of that class using the `db::Query` object and the position of the field in the select list. For each row, cast the `db::Field` to an appropriate C++ data type to access the value. Example 5.1, “SQL Select Example in C++” illustrates this process.

Example 5.1. SQL Select Example in C++

```
db::Database db = ...;
db::Query q;
```

```
char *cmd;

cmd = "select id, name "
      " from contact "
      " order by name ";

if (DB_SUCCESS(q.exec_direct(db, cmd))) {
    db::IntegerField id (q, 0);
    db::WStringField name(q, 1);

    for (q.seek_first(); !q.is_eof(); q.seek_next()) {
        std::wcout << (long) id << '\t';
        std::wcout << WString(name).c_str() << std::endl;
    }
}
```

If a query uses parameters, as described in Section 4.1.4, “SQL Query Parameters”, values can be assigned to each parameter after the query has been prepared but before it is executed. Use the `param()` function to assign a value to each parameter. The `param()` function returns a `db::Field` that can be assigned a value directly using the assignment operator, or set to `NULL` with the `set_null()` function. Each parameter is `NULL` until a value is assigned.

To execute the query again with different parameter values, first call `unexecute()` to return the query to a prepared state, then modify the parameters as before. Example 5.1, “SQL Select Example in C++” shows how to use parameters to execute a single query multiple times. Each parameter’s previous value will carry over to the next execution unless it is explicitly assigned a new value or set to `NULL`.

Example 5.2. SQL Insert Example in C++

```
db::Database db = ...;
const wchar_t *name = L"My Name";
db_uint ring_id = 6;

db::Query q;

q.prepare(db,
          "insert into contact (id, name, ring_id) "
          " values (next value for contact_id, ?, ?) ");
q.param(0) = name;
q.param(1) = ring_id;
q.execute();
q.unexecute();

q.param(0) = L>Your Name";
q.param(1) = 3;
q.execute();
q.unexecute();
```

Chapter 6. The C API

6.1. Getting Started

6.1.1. Initializing the Library

Before calling any other API functions, the library must be initialized. To initialize the library, call `db_init()` with the current API version:

```
db_init(DB_API_VER);
```

When the API is no longer needed, call `db_done()` to free all resources used by the library:

```
db_done();
```

6.1.2. Creating and Opening a Database

To use ITTIA DB, include the header `ittia/db.h`.

```
#include <ittia/db.h>
```

A database is managed by a `db_t` handle. To create a database for the first time, use the `db_create_file_storage()` function.

```
db_t hdb = db_create_file_storage(DATABASE_NAME, NULL);  
if (hdb == NULL)  
    fprintf(stderr, "Error creating database: %d\n", get_db_error());
```

Caution

If the file already exists, `db_create_file_storage()` will overwrite it.

To open an existing database, use the `db_open_file_storage()` function.

```
db_t hdb = db_open_file_storage(DATABASE_NAME, NULL);  
if (hdb == NULL)  
    fprintf(stderr, "Error opening database: %d\n", get_db_error());
```

When done using the database, call `db_shutdown`.

```
db_shutdown(hdb, DB_SOFT_SHUTDOWN, NULL);
```

6.1.3. Error Handling

Many errors can occur during a call to a C API function. Whether or not an error has occurred is indicated by the return value of the function. For functions that return a handle type, such as `db_t`, `db_cursor_t` or `db_row_t`, a value of `NULL` is used to indicate an error. For functions that return a value, such as a number or flags, a special value is reserved for errors, often `-1` (`DB_WTIME_FAIL`, `DB_LEN_FAIL`). Any function that would not otherwise return a

value returns a `db_result_t`, which can be either `DB_OK` (1) or `DB_FAIL` (0).

When an error occurs, an error code is set that indicates the kind of error. The error code can be retrieved by calling `get_db_error()`. Negative values indicate an error. Positive values indicate a warning.

Note

The value returned by `get_db_error()` is not cleared by any API calls except `clear_db_error()`. To check for an error in any of a series of calls, first call `clear_db_error()`, then call each API function, and finally use `get_db_error()` to check the result.

Tip

Error codes are listed in the Error Handling section of the API Reference. The same error codes are used in both the C and C++ APIs.

6.1.4. Data Structures

Several data structures in the C API can be instantiated statically at compile-time. These structures are `db_oid_t`, `db_indexdef_t` and `db_tabledef_t`. To initialize these data structures, use the `DB_ALLOC_INITIALIZER()` macro as the value for the `db_alloc` member. In the case of `db_oid_t`, use `DB_OID_INITIALIZER()` to initialize a new variable.

6.2. Database Design

For a detailed discussion of database design issues, see Section 5.2, “Database Design”.

Note

Schema updates cannot be performed inside a transaction. See Section 5.3.1, “Transactions” for more information.

6.2.1. Column Types

ITTIA DB can store data in a wide range of types. Every column has a data type, which limits the kind of data that can be stored in that column. This ensures that data is always in the expected format when retrieved from the database and imposes a hard limit on the amount of storage needed for each row.

To select the type of a column, set the `field_type` of a `db_fielddef_t` structure to the schema identifier listed in Table 6.1, “C Column Types” when a table is created as described in Section 6.2.2, “Tables, Fields and Indexes”. To access data stored in a column, use the data type identifier to bind the column to a variable of the listed C data type. See Section 6.3.3, “Rows” for more information about binding a column in a row to a variable.

Tip

Column types in the same category can be bound to the C data type of any other column type in the same category. For example, a column of type `DB_COLTYPE_SINT8` can be bound to a variable of type `int32_t` with the `DB_VARTYPE_SINT32` identifier. Or a column of type `DB_COLTYPE_UTF8STR` can be bound to a variable of type `db_utf16_t` with the `DB_VARTYPE_UTF16STR` identifier. Data in the variable will be automatically converted to the database format.

Data can be lost when converting to a smaller numeric format. Data is not lost when converting between Unicode string types, provided that the maximum string length is not exceeded.

Category	Schema Identifier	C Data Type	C Data Type Identifier
Integer	DB_COLTYPE_SINT8	int8_t	DB_VARTYPE_SINT8
Integer	DB_COLTYPE_SINT16	int16_t	DB_VARTYPE_SINT16
Integer	DB_COLTYPE_SINT32	int32_t	DB_VARTYPE_SINT32
Integer	DB_COLTYPE_SINT64	int64_t	DB_VARTYPE_SINT64
Integer	DB_COLTYPE_UINT8	uint8_t	DB_VARTYPE_UINT8
Integer	DB_COLTYPE_UINT16	uint16_t	DB_VARTYPE_UINT16
Integer	DB_COLTYPE_UINT32	uint32_t	DB_VARTYPE_UINT32
Integer	DB_COLTYPE_UINT64	uint64_t	DB_VARTYPE_UINT64
Float	DB_COLTYPE_FLOAT32	db_float32_t	DB_VARTYPE_FLOAT32
Float	DB_COLTYPE_FLOAT64	db_float64_t	DB_VARTYPE_FLOAT64
Currency	DB_COLTYPE_CURRENCY	db_sint32_t	DB_VARTYPE_SINT32
ANSI String	DB_COLTYPE_ANSISTR	db_ansi_t, char *	DB_VARTYPE_ANSISTR
Unicode String	DB_COLTYPE_UTF8STR	db_utf8_t, char *	DB_VARTYPE_UTF8STR
Unicode String	DB_COLTYPE_UTF16STR	db_utf16_t	DB_VARTYPE_UTF16STR
Unicode String	DB_COLTYPE_UTF32STR	db_utf32_t	DB_VARTYPE_UTF32STR
Date and Time	DB_COLTYPE_DATE	db_date_t	DB_VARTYPE_DATE
Date and Time	DB_COLTYPE_TIME	db_time_t	DB_VARTYPE_TIME
Date and Time	DB_COLTYPE_DATETIME	db_datetime_t	DB_VARTYPE_DATETIME
Date and Time	DB_COLTYPE_TIMESTAMP	db_timestamp_t	DB_VARTYPE_TIMESTAMP
BLOB	DB_COLTYPE_BLOB	db_blob_t	DB_VARTYPE_BLOB

Table 6.1. C Column Types

6.2.2. Tables, Fields and Indexes

To create a table, first describe the table's fields as arrays of `db_fielddef_t`.

first and indexes as arrays of `db_field_def_t` and `db_index_def_t`. Then call `db_create_table()` and `db_create_index()` to create the table and its indexes.

```
#define MAX_CONTACT_NAME 50
#define MAX_FILE_NAME 50

static db_fielddef_t contact_fields[] =
{
#define CONTACT_ID 0
#define CONTACT_NAME 1
#define CONTACT_RING_ID 2
#define CONTACT_PICTURE_NAME 3
#define CONTACT_PICTURE 4
    { CONTACT_ID, "id", DB_COLTYPE_UINT32 },
    { CONTACT_NAME, "name", DB_COLTYPE_UTF16STR, MAX_CONTACT_NAME },
    { CONTACT_RING_ID, "ring_id", DB_COLTYPE_UINT32 },
    { CONTACT_PICTURE_NAME, "picture_name", DB_COLTYPE_ANSISTR, MAX_FILE_NAME },
    { CONTACT_PICTURE, "picture", DB_COLTYPE_BLOB }
};
```

Next describe which fields will be indexed as arrays of `db_indexfield_t`.

```
static db_indexfield_t contact_by_id_fields[] =
{
    { CONTACT_ID }
};
```

Next define the attributes of the table's indexes as `db_indexdef_t`.

```
#define CONTACT_BY_ID "by_id"
static db_indexdef_t contact_index =
{
    DB_ALLOC_INITIALIZER(),
    DB_INDEXTYPE_DEFAULT,
    CONTACT_BY_ID,
    DB_UNIQUE_INDEX,
    1, /* Number of fields in the index. */
    contact_by_id_fields
};
```

Next define the table as `db_tabledef_t` using the field and index definitions created above.

```
#define CONTACT_TABLE "contact"
db_tabledef_t contact_table =
{
    DB_ALLOC_INITIALIZER(),
    DB_TABLETYPE_DEFAULT,
    CONTACT_TABLE,
    5, /* Number of fields in the table. */
    contact_fields,
    1, /* Number of indexes on the table. */
    contact_index
};
```

Finally, create the table by calling `db_create_table()`.

```
db_result_t rc;

rc = db_create_table(hdb, contact_table->table_name, contact_table, 0);

if (rc == DB_FAIL)
    fprintf(stderr, "Unable to create table\n");
```

And create the indexes by calling `db_create_index()`.

```
rc = db_create_index(hdb, contact_table->table_name, contact_index->index_name, contact_in

if (rc == DB_FAIL)
    fprintf(stderr, "Unable to add index\n");
```

6.2.3. Sequence Generators

Sequence generators are used to generate unique, increasing integer values. A sequence generator is created with an initial starting value and will then retrieve successive numbers in the sequence. Each generated number is always greater than all previous numbers produced by the sequence generator. This provides a convenient source for unique identifiers.

Note

Sequences may contain gaps, and so should not be used in situations where numbers cannot be skipped, such as matching with sequence numbers pre-printed on paper forms.

To create a sequence define a `db_seqdef_t` with a name and initial value then call `db_create_sequence()`.

```
#define CONTACT_ID_SEQUENCE    "contact_id"
db_seqdef_t contact_id_sequence = {
    CONTACT_ID_SEQUENCE,
    {{ 1, 0}}
};

rc = db_create_sequence(hdb, contact_id_sequence->name, contact_id_sequence);

if (rc == DB_FAIL)
    fprintf(stderr, "Unable to create sequence %s\n", contact_id_sequence->seq_name);
```

6.3. Database Access

6.3.1. Transactions

All data access operations must occur inside a transaction. A transaction is a logical group of operations that must all succeed or fail together. Changes made to the database do not become permanent until the transaction is committed. A transaction can also be aborted, which will undo any changes made since the transaction was started.

To start a transaction, call the `db_begin_tx()` function.

```
db_begin_tx(hdb, DB_DEFAULT_ISOLATION | DB_LOCK_SHARED);
```

Note

For Isolation mode and Lock mode parameters see [Isolation Levels and Locking](#).

To commit the changes made during the transaction, call `db_commit_tx()`. The completion parameter determines how quickly the changes made in this transaction will be flushed from the database cache to permanent storage.

```
db_commit_tx(hdb, DB_FORCED_COMPLETION);
```

Note

When the commit function returns successfully, any modifications made within the transaction are guaranteed to be completely written to the storage media.

To cancel the changes made during the transaction, call `db_abort_tx()`.

```
db_abort_tx(hdb, DB_FORCED_COMPLETION);
```

6.3.2. Cursors

A cursor is used to position on a specific row in a table and to search using an index. An index is selected when opening the cursor to determine which fields to sort by when traversing the table and which fields are available for searching.

```
db_cursor_t contact_cursor;
db_table_cursor_t p;

db_table_cursor_init(&p);
p.index = CONTACT_BY_ID;
p.flags = DB_CAN_MODIFY | DB_LOCK_EXCLUSIVE;
contact_cursor = db_open_table_cursor(hdb, CONTACT_TABLE, &p);
db_table_cursor_destroy(&p);
```

Alternately, the cursor options can be created statically:

```
db_cursor_t contact_cursor;
static const db_table_cursor_t contact_cursor_def =
{
    CONTACT_BY_NAME, /* index */
    DB_SCAN_FORWARD | DB_LOCK_SHARED /* flags */
};

contact_cursor = db_open_table_cursor(hdb, CONTACT_TABLE, &contact_cursor_def );
```

A cursor should be closed when it is no longer needed to free resources:

```
db_close_cursor(contact_cursor);
```

6.3.3. Rows

Data is stored and retrieved through rows. A row is a set of temporary buffers for storing the contents of database fields. Each field in a row can be stored in one of three locations: a managed internal buffer, a local variable referenced by memory address, or a member of a data structure. A row variable is independent of any specific row in the database and can be re-used for many insert, fetch and update operations.

A row is created by calling either `db_alloc_row()` or `db_alloc_cursor_row()`. A row can only be used to access fields that it is aware of and will not set or retrieve values for any fields that it is not. A row variable can be created that is initially bound to no database fields:

```
db_row_t contact_row;
contact_row = db_alloc_row(NULL, 2);
```

This row should then be dynamically bound to two fields by calling `db_bind_field()`.

The data in a row can be read and written by calling `db_get_field_*()` and `db_set_field_*()` functions. For safe access, use `db_get_field_data()` and `db_set_field_data()` to copy values to or from each field in the row. The size of the field can be determined by calling `db_get_field_size()`.

Tip

To obtain a direct pointer to the buffer that is used by the row, call `db_get_field_buffer()`. When writing to this pointer, be careful not to overrun the size of the buffer.

When a row is no longer needed, it must be freed:

```
db_free_row(contact_row);
```

6.3.3.1. Managed Fields

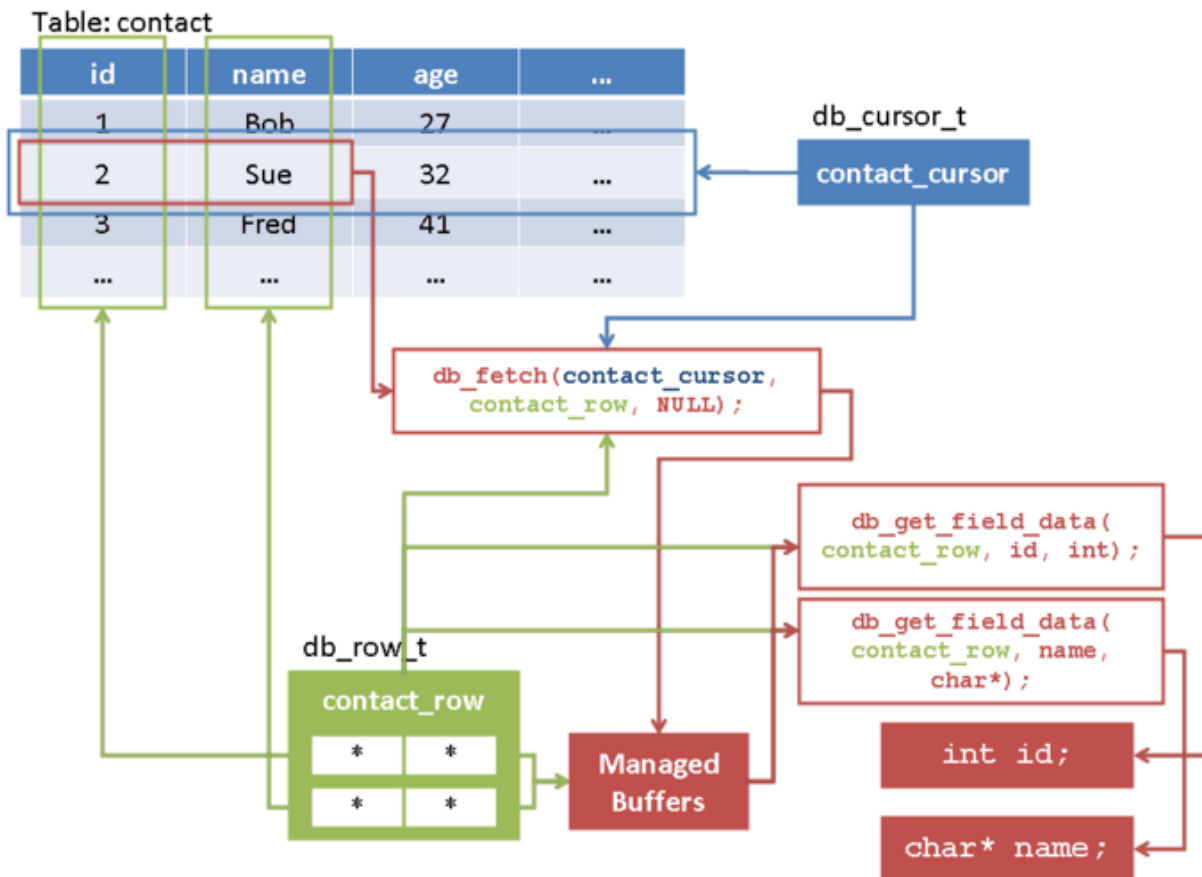


Figure 6.1. Managed Field Binding

The first method, a managed buffer, is most suitable when the exact data type of a field is not known at compile time. It allows the value to be extracted as a void pointer and cast to the appropriate type at run time. To create a row with managed buffers for all fields in a cursor, call `db_alloc_cursor_row()`:

```
db_row_t contact_row;
contact_row = db_alloc_cursor_row( contact_cursor );
```

6.3.3.2. Absolute Bound Fields

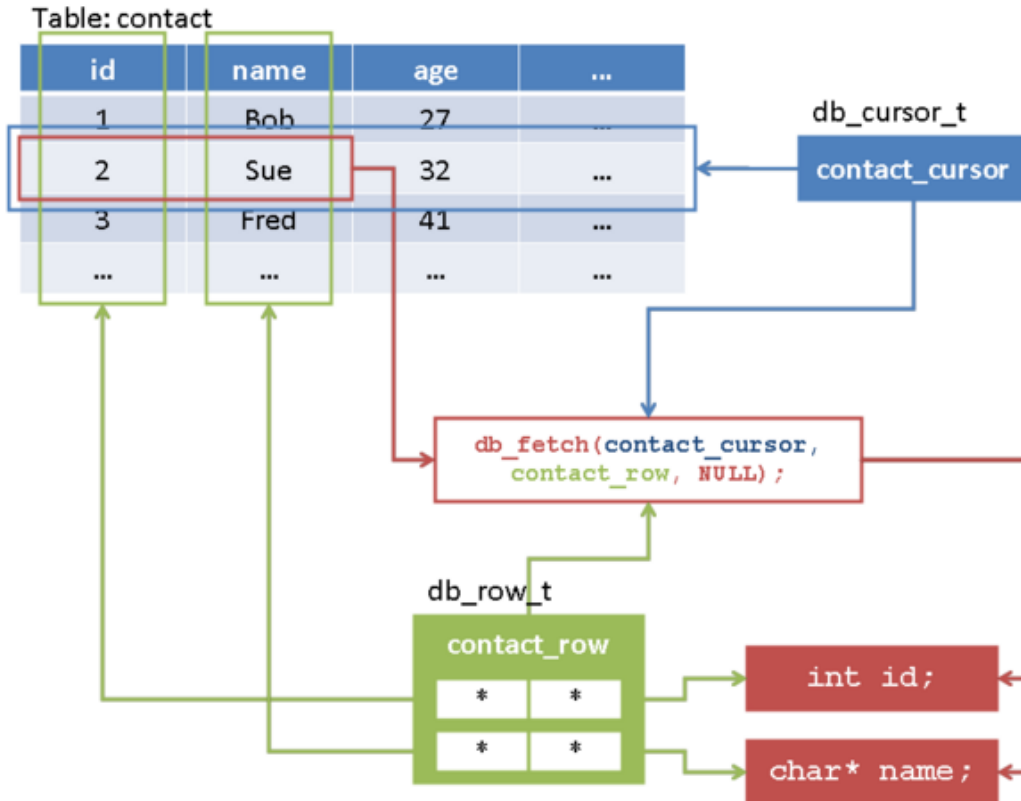


Figure 6.2. Absolute Field Binding

When the data type of a field is known at compile time, it can be bound directly to a local variable by providing the memory address and size of the variable in bytes. When data is retrieved from the database, it will be stored in the bound variables. When data is stored to the database, it will be read from these same variables. To create a row that has fields bound to memory addresses, call `db_alloc_row()` with an array of binding definitions:

```
uint32_t id;
db_ucs2_t name[50];

const db_bind_t row_def[] =
{
    {
```

```

        CONTACT_ID,                /* field_no */
        DB_VARTYPE_UINT32,         /* data_type */
        DB_BIND_ADDRESS(&id),     /* data_ptr */
        sizeof(id),               /* data_size */
        DB_BIND_ADDRESS(NULL),    /* data_ind */
        DB_BIND_ABSOLUTE,        /* data_flags */
    },
    CONTACT_NAME,                 /* field_no */
    DB_VARTYPE_UTF16STR,          /* data_type */
    DB_BIND_ADDRESS(name),       /* data_ptr */
    sizeof(name),                /* data_size */
    DB_BIND_ADDRESS(NULL),      /* data_ind */
    DB_BIND_ABSOLUTE,           /* data_flags */
}
};

contact_row = db_alloc_row(row_def, 2);

```

Instead of using internal buffers, the row will use the variables or memory addresses that have been provided for these fields. These fields can still be accessed using `db_get_field_*()` and `db_set_field_*()` functions if necessary, but it is generally more convenient to use the local variables to which they are bound directly.

Tip

The `data_ind` member of `db_bind_t` is used to bind a field length indicator for the given field. The field length indicator provides additional information about the value stored in the database, such as string length and NULL state. In an absolute binding, set `data_ind` to the address of a `db_len_t` variable to access the field length indicator.

Fields can be bound dynamically using `db_bind_field()`. In the following example, `db_bind_field()` is used to bind the variable `id` to two different row objects.

```

db_row_t simple_row, contact_row;
uint32_t id = 0;
db_len_t id_ind = DB_FIELD_NULL;

static db_bind_t bind_def =
{
    CONTACT_ID,                /* field_no */
    DB_VARTYPE_UINT32,         /* data_type */
    DB_BIND_ADDRESS(&id),     /* data_ptr */
    sizeof(id),               /* data_size */
    DB_BIND_ADDRESS(&id_ind), /* data_ind */
    DB_BIND_ABSOLUTE,        /* data_flags */
};

simple_row = db_alloc_row(NULL, 1);
db_bind_field(simple_row, &bind_def);

contact_row = db_alloc_cursor_row( contact_cursor );
db_bind_field(contact_row, &bind_def);

```

Tip

`db_bind_field()` will override any previous bindings for that field. In this example, managed bindings are first created for all cursor fields. The binding that was created for `CONTACT_ID` is then replaced by an absolute binding to the `id` variable. In this way, a row can contain a mixture of different bind methods.

Binding to the same variable in two different rows is an efficient way to copy fields between tables. Data can be fetched from one table using the first binding and then stored directly to another table using the second binding without making any additional copies.

6.3.3.3. Relative Bound Fields

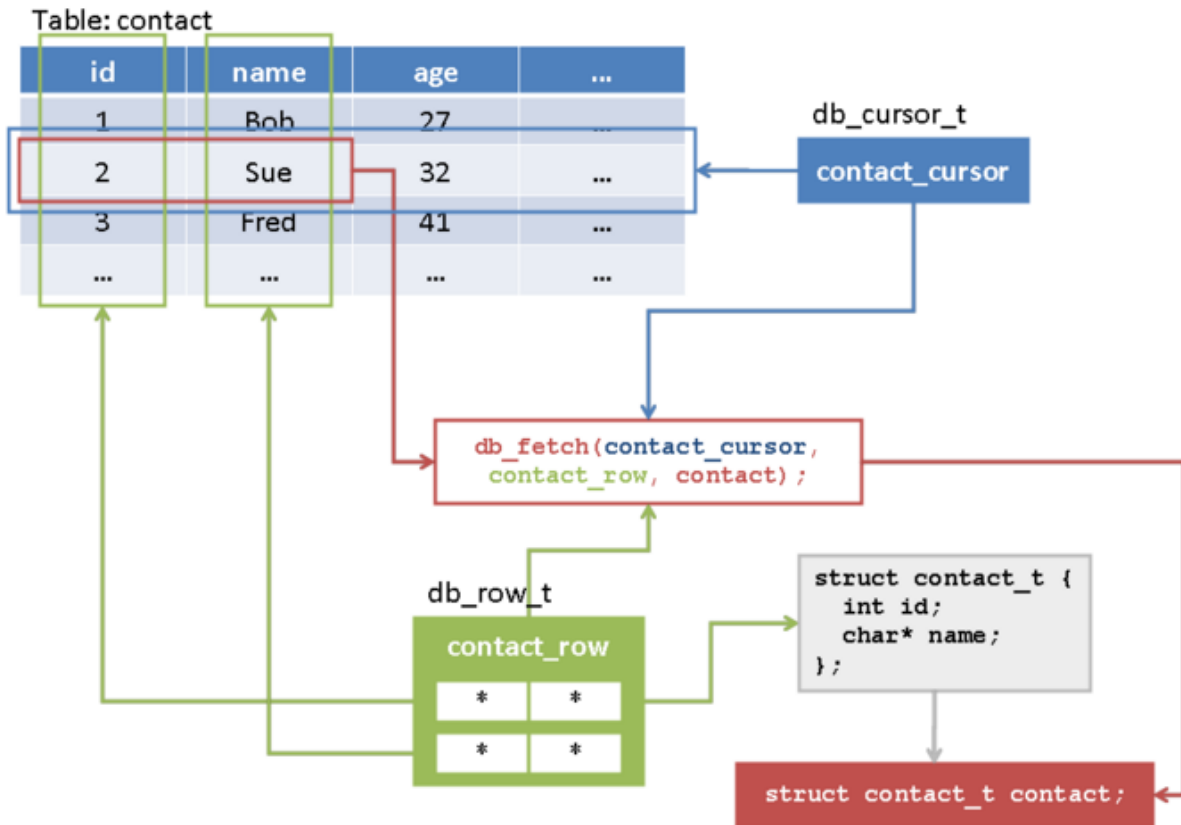


Figure 6.3. Relative Field Bindings

A row can also be bound to a data structure to provide convenient access using a C struct.

A relative binding is defined in much the same way as for address bindings, but instead of using an absolute address, an offset into the data structure is used. The `DB_BIND_OFFSET()` and `DB_BIND_SIZE()` macros can be used to determine the offset and size of a struct member.

```
#define MAX_PHONE_NUMBER 20
typedef struct phone
{
    contactid_t contact_id;
    db_ansi_t   number[MAX_PHONE_NUMBER + 1];
    uint32_t    type;
    int32_t     speed_dial;
} phone_t;
```

```

static const db_bind_t phone_number_binds[] =
{
    { PHONE_CONTACT_ID,
      DB_VARTYPE_UINT32,
      DB_BIND_OFFSET(phone_t, contact_id),
      DB_BIND_SIZE(phone_t, contact_id),
      -1,
      DB_BIND_RELATIVE},
    { PHONE_NUMBER,
      DB_VARTYPE_ANSISTR,
      DB_BIND_OFFSET(phone_t, number),
      DB_BIND_SIZE(phone_t, number),
      -1,
      DB_BIND_RELATIVE},
    { PHONE_TYPE,
      DB_VARTYPE_UINT32,
      DB_BIND_OFFSET(phone_t, type),
      DB_BIND_SIZE(phone_t, type),
      -1,
      DB_BIND_RELATIVE},
    { PHONE_SPEED_DIAL,
      DB_VARTYPE_SINT32,
      DB_BIND_OFFSET(phone_t, speed_dial),
      DB_BIND_SIZE(phone_t, speed_dial),
      -1,
      DB_BIND_RELATIVE},
};

phone_t phone;
db_row_t phone_row;

phone_row = db_alloc_row(phone_number_binds, 4);
db_fetch(phone_cursor, phone_row, &phone);

```

The functions `db_fetch()`, `db_insert()`, and `db_update()` each accept a *uobject* parameter that is only required for relative bound fields. This parameter is the address of a data structure instance that corresponds to the relative bindings.

Note

If relative binding is not used in the row, the parameter of type `db_object_t` should always be `NULL`.

Tip

Relative bindings can also be used with an untyped memory buffer instead of a C data structure by setting `data_ptr` to the byte offset of each field.

6.3.4. Inserting Data

To insert data, assign values to a row. Then call `db_insert()`:

```

uint32_t id = 4;
db_ucs2_t name[50] = "Bob";

const db_bind_t row_def[] =
{
    {
        CONTACT_ID,
        DB_VARTYPE_UINT32,
        /* field_no */
        /* data_type */

```

```
    DB_BIND_ADDRESS(&id),          /* data_ptr */
    sizeof(id),                   /* data_size */
    DB_BIND_ADDRESS(NULL),        /* data_ind */
    DB_BIND_ABSOLUTE,            /* data_flags */
},
{
    CONTACT_NAME,                 /* field_no */
    DB_VARTYPE_UTF16STR,          /* data_type */
    DB_BIND_ADDRESS(name),        /* data_ptr */
    sizeof(name),                 /* data_size */
    DB_BIND_ADDRESS(NULL),        /* data_ind */
    DB_BIND_ABSOLUTE,            /* data_flags */
}
};

contact_row = db_alloc_row(row_def, 2);

db_insert(contact_cursor, contact_row, NULL);

db_close_row(r);
```

Note

Data inserted into the database will not be persistent until the current transaction is committed. See Transactions.

6.3.5. Traversing a Cursor

To traverse a cursor, use `db_seek_first()`, `db_seek_next()`, and `db_eof()` to position on each row of data. The data for the current row can be accessed by fetching the data to a previously-created row.

```
for (db_seek_first(c); !db_eof(c); db_seek_next(c)) {
    db_fetch(contact_cursor, contact_row, NULL);
    /* Perform operation on current row */
}
```

6.3.6. Updating Data

In addition to accessing data, you can also update the current row. Update the values bound to the row and call `db_update()`.

```
db_update(contact_cursor, contact_row, NULL);
```

Note

As with inserts, modifications to the database will not be persistent until the current transaction is committed. See Transactions.

6.3.7. Deleting the Row

To delete the current row, call `db_delete()`.

```
db_delete(contact_cursor);
```

Note

The row will not be permanently removed from the database until the current transaction is committed. See Transactions.

6.3.8. Reading and Writing BLOBs

A BLOB (Binary Large Object) is a special type that can store data of arbitrarily large size. To do this, data in a BLOB field is accessed in small divisions called chunks. These chunks can be written to and read from using loops to access the entire BLOB or only a portion of the BLOB.

In the C API, a BLOB field is managed by binding to a variable of type `db_blob_t`. This structure is used to control what portion of the BLOB will be used in an update or fetch. When a BLOB field is fetched, `blob_size` is set to the total size of the BLOB field. If `chunk_data` is not NULL, it is filled with `chunk_size` bytes starting from position `offset` in the field. In case there is less data in the BLOB than `offset + chunk_size`, the actual number of bytes read is stored in `actual_size`.

Example 6.1, “Prepare to access BLOB field” shows how to bind a column in a table to a BLOB field. The resulting row is passed to `db_update()` for each chunk of data to be written into the database, as demonstrated in Example 6.2, “Write data from a file to a BLOB field”. Or the row can be passed to `db_fetch()` for each chunk of data to be read from the BLOB column, as in Example 6.3, “Read data from a BLOB field to a file”.

Example 6.1. Prepare to access BLOB field

```
#include <ittia/db.h>
#include "db_helper.h"

db_cursor_t t;

... /* Open t and position on an existing row. */

db_blob_t blob;

/* Bind "blob" to field number 3. */
const db_bind_t row_def[] =
{
    DB_BIND_VAR(3, DB_VARTYPE_BLOB, blob),
};

/* Allocate a row containing the BLOB field binding. */
db_row_t r = db_alloc_row(row_def, DIM(row_def));

/* Clear BLOB settings (optional). */
memset(&blob, 0, sizeof(blob));
```

Example 6.2. Write data from a file to a BLOB field

```
FILE *picture_file = fopen("read.png", "rb");

/* Allocate buffer */
#define DATA_SIZE 1024
char buffer[DATA_SIZE];

/* Prepare BLOB variables */
```

```
int bytes_read = 0;
picture.chunk_data = (void*)buffer;

/* Store picture into BLOB field */
while((bytes_read = fread(buffer, 1, DATA_SIZE, picture_file)) > 0)
{
    blob.offset = DATA_SIZE * num_chunks;
    blob.chunk_size = bytes_read;
    db_update(t, r, NULL);
}
fclose(picture_file);
```

Example 6.3. Read data from a BLOB field to a file

```
FILE *picture_file = fopen("write.png", "wb")

/* Allocate buffer */
#define DATA_SIZE 1024
char buffer[DATA_SIZE];

/* fetch file_name and blob.blob_size */
db_fetch(contact_cursor, contact_row, NULL);

/* Prepare BLOB variables */
blob.chunk_data = buffer;
blob.chunk_size = DATA_SIZE;

/* Export file from BLOB to disk */
for(blob.offset = 0; blob.offset < blob.blob_size; blob.offset += DATA_SIZE)
{
    db_fetch(t, r, NULL);
    fwrite(blob.chunk_data, blob.actual_size, 1, picture_file);
}

fclose(picture_file);
```

6.4. SQL Queries

Structured Query Language, or SQL, is a standard database interface that uses query strings to access and modify data. For the complete description of the SQL dialect used by ITTIA DB, see Chapter 4, *SQL*.

To execute an SQL statement from the ITTIA DB C API, create a query cursor by calling `db_prepare_sql_cursor()`. The cursor returned by this function can be used to execute the query and read the result set. An application should follow these steps to process an SQL statement:

1. Prepare an SQL statement.
2. Set query parameters (optional).
3. Execute the query.
4. Iterate through rows in the result set (optional).

5. Unexecute the query and repeat from step 2 (optional).

Steps 2 and 4 can be omitted depending on the nature of the SQL statement. For example, `INSERT` statements do not have a result set and thus do not use step 4. Schema definition statements, such as `CREATE TABLE`, can skip step 2 because they do not accept parameters.

Important

Use parameters whenever possible. Parameters are faster and more secure than embedding data directly in an SQL statement, for example using `sprintf`.

The function `db_prepare_sql_cursor()` parses an SQL statement and returns an SQL cursor. The last parameter is reserved for future use. Always use the value 0 for this parameter.

```
db_cursor_t sql_cursor;
sql_cursor = db_prepare_sql_cursor(hdb, "select * from some_table", 0);
```

After the cursor is prepared, the statement can be executed by a call to `db_execute()`. Executing a statement may modify the database or open a result set that can then be browsed through the cursor.

```
db_execute(sql_cursor, NULL, NULL);
```

If the query is a `SELECT` statement, a result set will be available after the query is executed. The result set is a read-only collection of rows with the same structure as a table. The result set is only sorted if the SQL statement contains an `ORDER BY` clause.

To iterate over the result set, use `db_seek_first()` to position on the first row, `db_seek_next()` to advance to the next row, and check `db_is_eof()` to determine when all rows have been processed. An SQL query can only be traversed forward and cannot be reset to the first row after `db_seek_next()` has been called.

Each row in the result set contains one or more fields that correspond to the columns and expressions in the `SELECT` list. Fields are identified by the order that they appear in the `SELECT` list, starting from zero.

To access field values in the result set, use `db_fetch()` to copy data into a C API row, as described in Section 6.3.3, “Rows”.

Because result set is read-only, `db_insert()`, `db_update()`, and `db_delete()` cannot be used on an SQL cursor.

Example 6.4, “Executing a `SELECT` statement” shows how to execute a `SELECT` statement and iterate over the results using a managed row.

Example 6.4. Executing a `SELECT` statement

```
db_cursor_t      sql_cursor;
int              nfield;
db_fielddef_t    field_def;
db_row_t         row;
int              field_count;

sql_cursor = db_prepare_sql_cursor(hdb, "select * from some_table", 0);
row = db_alloc_cursor_row(sql_cursor);
```

```
field_count = db_get_field_count(sql_cursor);

db_execute(sql_cursor);
db_seek_first(sql_cursor);

while ( !db_eof(sql_cursor) ) {
    //-----
    // RECORD LEVEL PROCESSING
    //-----
    db_fetch( sql_cursor, row, NULL );

    for ( nfield = 0; nfield < field_count; nfield++ ) {
        //-----
        // FIELD LEVEL PROCESSING
        //-----
        db_get_field ( sql_cursor, nfield, &field_def );
        if ( db_is_null(row, nfield) ) {
            //-----
            // NULL DATA CHECK
            //-----
            continue;
        }
        db_get_field_data(row, nfield, field_def.field_type, &var, sizeof(var));
        :
        .
    }
    db_seek_next(sql_cursor);
}

db_close_cursor(sql_cursor);
```

Some SQL statements accept one or more parameters. For these queries, a parameter row must be passed as the second parameter to `db_execute()`. If any fields in the row use relative bindings, an object address must also be passed as the third parameter, otherwise the third parameter can be NULL. The *fieldno* for each bound field must be set to the number of the parameter. See Section 4.1.4, “SQL Query Parameters” for more information.

An SQL query can be executed multiple times by repeatedly calling `db_execute()`. The values of the parameter row can be modified each time before the query is executed. If the query has a result set, the result set must be closed by calling `db_unexecute()` before the cursor can be executed again.

Tip

A prepared SQL cursor can be left open and reused later to avoid the overhead of opening the cursor. If the cursor is not closed immediately after use, it is a good practice to call `db_unexecute()` to free any resources held by the query.

Chapter 7. Shared Database Access

7.1. Overview

Shared database access allows ITTIA DB to simultaneously process multiple transactions. ITTIA DB uses fine-grained locking to prevent unnecessary waiting, which means that time-consuming operations, such as synchronization, and operations that continuously modify the database, such as sensor readings, will not block other transactions from accessing the database, unless a conflict would arise. Shared database access provides a great benefit to applications by allowing different data access tasks to be safely carried out at the same time. ITTIA DB supports both client/server shared access using TCP/IP communications, and multi-threaded shared access within a stand-alone application.

Each transaction is handled in isolation, which means that ITTIA DB prevents concurrent transactions from interfering with each other. Without isolation, a transaction may leave incorrect or inconsistent information in the database. If a transaction writes a value to a row that another transaction is in the process of modifying, then that change will be overwritten by the other transaction, resulting in incorrect information. For example, two transactions may attempt to update an account balance at the same time. If both transactions first read the current balance, then attempt to write the new balance, one will overwrite the other.

Because transactions are usually very short, such situations show up very rarely, which makes them almost impossible to catch. But given a long enough run time, they will eventually show up in any application that is not designed to prevent them. In many cases, correct data values are critical to the application's behavior. ITTIA DB provides isolation to automatically prevent these kinds of concurrency errors so that developers can be certain their data is valid.

ITTIA DB accomplishes isolation using two-phase locking (2PL), which locks resources as necessary to prevent conflicts. For example, when a transaction reads or writes a row of data, that row is automatically locked to prevent other transactions from modifying it. At the same time other transactions can lock and modify other rows. Each transaction's locks are released when it is committed or aborted. This creates the illusion that only one transaction is active at any time, without incurring the performance cost of actually waiting for each transaction to finish before starting a new one.

Tip

It is convenient to think of transactions as occurring in series, one after another. The effective order of transactions is determined by the order that they are committed in and not the order in which they begin.

7.. Client/server Shared Access

Database access can be shared between multiple processes using client/server shared access. Client applications connect to a server process that runs either on the same device, or on another machine on the network.

For details on how to run the data server, **dbserver**, see `dbserver`.

To access a database on a remote machine, use the following URL syntax for the file name:

```
xdbtp://server[:port]/database
```

Replace the following with appropriate values as described:

server Host name of the machine on which the data server is run. If **dbserver** is running on the same device, use *localhost* or *127.0.0.1*.

port Alternate port number if `--bind-port` server option was used. The default value is *16534*.

database Name of database file to open or create on the remote server.

Note

Square brackets indicate that the enclosed text is optional.

7.3. Multi-threaded Shared Access

Multithreading allows an application to handle more than one task at a time by breaking a single process into multiple threads of execution. If one thread takes too long to complete a task, for instance when performing a tedious calculation or waiting for input from a device, other threads can continue working. An application can safely interact with an ITTIA DB database from multiple threads by starting a separate transaction in each thread.

Because threads all use the same memory space, variables can be shared between threads. This usually requires special attention to ensure that threads do not interfere with each other. However, ITTIA DB automatically provides the isolation that is needed to accomplish this when accessing data in an ITTIA DB database.

To create a new thread on Windows, use the standard `CreateThread` function or the MFC `CWinThread` class. On POSIX operating systems, use the `pthread` library to create new threads. You can also use the platform-independent thread functions in `ittia/os/os_thread.h`.

The following example shows how to run two transactions simultaneously in different threads. The `parent` function spawns a thread that runs the `child` function. Each thread then starts its own transaction and performs database operations.

```
#include <ittia/db++.h>
#include <ittia/os/os_thread.h>

// Child thread
void child(void * db_ptr)
{
    Database db = *db_ptr;

    db.tx_begin();
    // ...
    // Database operations
    // ...
    db.tx_commit();
}

// Parent thread
void parent(Database db)
{
    os_thread_t * child_thread;

    // Spawn a child thread
    child_thread = os_thread_spawn(child, &db, DEFAULT_STACK_SIZE, OS_THREAD_JOINABLE);
    if (child_thread == NULL)
        return; // Error spawning child thread

    db.tx_begin();
    // ...
    // Database operations
    // ...
    db.tx_commit();

    // Wait for the child thread to return
    os_thread_join(child_thread);
}
```

7.4. Isolation Levels

Isolation of database transactions is achieved by blocking the read and write actions of other running transactions, which reduces concurrency and can hamper performance when many concurrent transactions are active. Fortunately, full isolation (executing each transaction to completion in a serialized way) is not always necessary for the correct operation of an application.

Isolation levels are used to measure how much a transaction depends on the behaviour of other transactions. If a transaction would not be affected by certain kinds of modifications that other transactions could make, a lower isolation level can be used to improve concurrency.

The following isolation levels are defined in ITTIA DB:

Isolation Level	Implementation Details
SERIALIZABLE	All the <i>Shared</i> and <i>Exclusive locks</i> on both table and index rows are held till the end of each transaction.
REPEATABLE READ	<i>Shared locks</i> on index rows are released as soon as an operation has completed. <i>Shared locks</i> and <i>Exclusive locks</i> on table rows are held till the end of each transaction.
READ COMMITTED	<i>Shared locks</i> on index rows are released as soon as an operation has completed. <i>Shared locks</i> on table rows are released as soon as the database record goes out of scope. All <i>Exclusive locks</i> on table and index rows are held till the end of each transaction.

Table 7.1. Isolation Level Implementations

7.4.1. Serializable

The serializable isolation level provides full isolation. The database behaves as though all transactions occur sequentially, one after another.

According to SQL standard, the serializable isolation level guarantees the absence of all phenomena specified below.

Phenomena	Description
P1 (“Dirty read”)	Transaction T1 modifies a row. Transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed and that may thus be considered to have never existed.
P2 (“Non-repeatable read”)	Transaction T1 reads a row. Transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.
P3 (“Phantom”)	Transaction T1 reads the set of rows N that satisfy some <search condition>. Transaction T2 then executes SQL-statements that generate one or more rows that satisfy the

Phenomena	Description
	<search condition> used by SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

Table 7.2. Phenomena prevented by Serializable Isolation

7.4.2. Repeatable Read

The repeatable read isolation level ensures that all data accessed by a transaction is not modified by other transactions until after the transaction is committed. This means that reads are repeatable: every time you read a row, it contains the same data. If you write data to a row, or create a new row within a transaction, it will contain the same data until that transaction writes something different there.

Unlike serializable isolation, repeatable read isolation does not prevent phantom reads. A phantom read occurs when a transaction reads a row that was inserted by a later transaction. This can cause the results of a query to change over the life of a transaction. For example, a transaction at the repeatable read isolation level may read all people in a table with age greater than 18. If later the same transaction attempts to read all people of age greater than 21, it may process new entries that were not present in the first query. If this is okay, then using repeatable read isolation is sufficient.

7.4.3. Read Committed

Allows non-repeatable reads. A non-repeatable read happens when a transaction reads changes to data that it has already read. One transaction sees the committed changes made by a later transaction.

7.4.4. Read Uncommitted

Allows dirty reads. A dirty read occurs when a transaction reads uncommitted changes made by a later transaction. The data read from the database may be inconsistent, representing only part of the changes made by the later transaction.

The database provides no isolation at the read uncommitted isolation level.

7.5. Selecting an Isolation Level

The isolation level can be selected in ITTIA DB when starting a transaction. Three isolation levels are available:

- `DB_SERIALIZABLE`
- `DB_REPEATABLE_READ`
- `DB_READ_COMMITTED`

The default isolation level is read committed. To select a different isolation level, specify it when starting a transaction.

```
db.tx_begin(DB_REPEATABLE_READ);
```

When opening a table cursor, it is possible to select a different isolation level than the isolation level used by the transaction. For example:

```
table.open(db, "table", DB_CURSOR_FOR_READ, DB_REPEATABLE_READ);
```

7.6. Change Notification

An advanced feature of ITTIA DB, data change notification allows an application to receive an automatic notification whenever another application makes a change in the database. This is a convenient way to send messages between applications that are sharing the database.

For more information about data change notification, see the functions `db_watch_table`, `db_unwatch_table`, and `db_wait` in the C API Reference, and the functions `Database::db_watch_table`, `Database::db_unwatch_table`, and `Database::db_wait` in the C++ API Reference.

Chapter 8. Advanced Topics

Concurrency control	Pessimistic locking
Locking method	Strict 2-phase locking
Locking mode	Multiple granularity locks
Locking granularity	Record
Isolation levels	SERIALIZABLE, REPEATABLE READ, READ COMMITTED
SERIALIZABLE isolation level implemented using	ARIES/IM next key locking
Lock owner	Application thread
Deadlock detection	Waits-For-Graph depth search
Deadlock resolution	The thread detected deadlock state refuses the operation.
Logging method	UNDO/REDO write-ahead logging
Indexing methods	B+ Tree
Buffer scheduling	Clock algorithm

Table 8.1. ITTIA DB Database Characteristics Quick Reference

8.1. Introduction

In this section, we explain in further detail some of the technologies employed by the ITTIA DB kernel. Understanding these topics will help developers to optimize and diagnose database-driven software. The reader is assumed to have some prior knowledge of general database concepts such as B+Tree indexes and locking.

8.1.1. Recommended Reading

Readers who are interested in a good introduction to database implementation techniques and related issues may enjoy the following books:

Bibliography

Jim Gray. Andreas Reuter. *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*.

Hector Garcia-Molina. Jeff Ullman. Jennifer Widom. *Database Systems: The Complete Book*.

C. Mohan. *Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM*.

C. Mohan. Donald J. Haderle. Bruce G. Lindsay. `Hamid Pirahesh. Peter M. Schwarz. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*.

8.2. ACID Properties

ITTIA DB fully obeys Atomicity, Consistency, Isolation, and Durability (ACID) properties. These properties are defined as follows:

Atomicity	The changes to a database performed within a transaction are either all applied when committed or none are applied when aborted. ^a
Consistency	Consistency refers to the legality of the database state at the time the transaction starts and commits. There are consistency constraints which the database must obey, if not the transaction cannot be performed.
Isolation	Isolation refers to the ability to perform an operation in a transaction independent of concurrently run transactions. Each transaction should behave as if is the only one modifying the database. Each time a conflict between transactions is about to occur, this conflict is detected and one transaction is delayed. ^b
Durability	Durability refers to the guarantee that as soon as a commit operation reports success the changes, made by that transaction, will persist and will not be undone. ^c

Table 8.2. ACID Properties

^aSequence number generators are an exception to atomicity. A sequence generator will never return the same value twice, regardless of whether the transaction commits or aborts.

^bThis property can be relaxed using multiple isolation levels to increase concurrent performance.

^cThis property can be relaxed using lazy transaction commits, which have no durability guarantee.

Atomicity and Durability properties are maintained using logging. Consistency and Isolation properties maintained using locking.

8.3. Logging

Logging is a way of tracking changes in database content in order to be able to rollback these changes on system crash or transaction abort. ITTIA DB uses UNDO/REDO WAL logging method, which is a well known as ARIES. The basis of operation of ARIES is that the physical changes (i.e. changes made at the database page level) are logged during forward operation, containing all the necessary information to repeat the same action again (redo) or rollback those changes (undo).

As soon as an operation completes a journal/log record is made. This can be done in two ways through a completion journal record or a completion log record (CLR). A CLR describes how to undo the operation logically, rather than physically so these records can be undone only. In the other words, while an operation performs changes to the database object, physical object consistency is maintained so that the operation can be undone physically, leaving the object in the exact state as it was before the operation started. On the contrary, operation completion records carry logical operations, thus when the operation is undone, the object can be in a different physical state.

This kind of logging makes fine-grained locking possible, allowing concurrency control on entities with no regard to database pages.

All operation completion records within a transaction are chained together, making it possible to selectively undo operations, enforcing the Atomicity property. Due to chaining, the undo operation is very efficient and requires almost the same amount of time as the original forward operation. This assessment assumes that most transactions commit rather than abort.

To ensure correctness the journal is flushed to the permanent storage before corresponding data changes are written to the storage. The journal is flushed to the storage at transaction commit time as well to ensure Durability property. The latter requirement greatly contributes to the commit time.

8.4. Locking

The database uses *Strict Two Phase Locking* (Strict 2PL) to ensure consistency and isolation. 2PL locking follows these rules:

1. If a transaction wants to read or write an object it must acquire a shared or exclusive lock on that object.
2. The transaction is not allowed to acquire any new locks after a lock has been released. The effect of this is transactions only release exclusive locks at commit time.

These two rules divide transactions into two phases. The first is the lock acquisition phase and the second is lock releasing phase. Locks are acquired automatically, during the transaction progress, or can be requested manually to avoid deadlock situations.

8.4.1. Types of Locks

There are two general types of locks used on database objects:

Shared locks	Shared locks allow read only access on a resource. Multiple shared locks can be held on one database object.
Exclusive locks	Exclusive locks allow both read and write access to a resource. Only one exclusive lock can be held on a single database object.

8.4.2. Lockable Objects

There are number of objects being locked during database operations. They are:

Storage	This kind of lock is acquired at transaction start. By default transactions run concurrently, acquiring shared locks, but sometimes an operations needs to guarantee it's performing a transaction alone.
Table schema	This kind of lock is acquired automatically in shared mode as soon as the first cursor on a table is opened and released when the last cursor is deleted. When a transaction wants to modify a table's schema – add or drop fields or indexes, the exclusive lock is acquired.
Table row (record)	This lock is acquired in the appropriate mode when a cursor observes a table row. When the lock is released depends on the isolation level, and will be described in more detail in Section 7.4, “Isolation Levels”.
Index row	This lock is acquired in the appropriate mode when performing an operation that involves indexes. The exact handling is described in Section 8.4.3, “Index Locking”.

8.4.3. Index Locking

Imagine we have an index which contains the sequence of values [1,2,3,4,5]. A transaction deletes the record with key [3], and the index is modified during the operation to exclude the deleted key (i.e. the index modification is not delayed till the commit). At the same time if another transaction scans the same index it would not know if a key was deleted from the index or not, thus causing the phantom of record absence in case the first transaction aborts. It

can try to insert a record with the same key as deleted, or update another record to have the same key, which would lead to collision if the first transaction rollbacks. To prevent a collision from occurring, we should lock the modified range, since it contains an uncertain state that is unknown to the other transaction.

To avoid this situation the database locks the next key, i.e. the lock is delegated to the next key in index order, even if the next record has not actually been modified. In such a case the concurrently scanning transaction would see the locked row and would wait for the first transaction to resolve the state by either committing or aborting.

The next key locking technique is actually a range lock, where one lock represents the whole locked range – in our example, after deleting the key [3], we lock the whole range or keys from [2] to [4].

Now we can describe how various operations are performed to cope with the problem:

During insert the database finds an appropriate place to insert and then requests a lock on the next record, assuming that there is a possibility of a key being deleted. As soon as the lock is granted the insert operation is performed, and the next row lock is released.

Key update is done in a way that is similar to the insert operation; it first checks the new key position just like the insert operation does. The only difference is that since an update operation deletes the old key value, it leaves the row that is next to the old key value locked.

Fetch is done as usual, requesting a single lock per record fetched.

Next key locking might cause ‘mysterious’ locks – since there is no difference between a row lock and a range lock this might cause unexpected operation waiting sometimes. In our example, if the concurrent transaction tries to update the row with key [4], which is used as a range lock, it would wait until the first transaction completes.¹

8.4.4. Manual Locking

The database performs automatic locking on every operation. Sometimes this locking is not enough, so the database allows locking tables and rows manually. Since these locks do not reflect the transaction state, but serve application synchronization, the manual locks are considered to be advisory and are not enforced to obey the rules of 2PL. This means that the application can request and release advisory locks at any time. Moreover the application can request locks which have a life-time that is longer than transaction time, making it possible to guarantee inter-transaction consistency.

For more information on manual locking, see the section on Locking in the C API Reference and the section on Object Ids and Locking in the C++ API Reference.

¹In practice these collisions are very rare. So rare that almost no database server using exactly this technique will ever mention this in the documentation.

Utilities

Platform	dbserver	dbserver GUI
Windows	bin\dbserver.exe	N/A
Windows CE	bin\arch\dbservercmd.exe	bin\arch\dbserver.exe
Linux	bin/dbserver	N/A

Table 16. Utility program files by platform

Name

`dbserver` -- Light-weight data server for multi-process shared access.

```
dbserver
dbserver [-t threadnum] [[-N] | [-R]]
[-ba address] [-bp port]
[-b bufnum] [-c chknum] [-p preallocnum] [-n name] [-r]
[-l locknum] [-w ownnum] [-o objnum] [-tr transnum]
[database...]
```

Description

dbserver provides shared database access to network clients. This allows processes running on one or more devices and computers to share access to the same database files.

An application will automatically connect to **dbserver** if it is linked with the inter-process communications (IPC) client library. For more details, see Section 7., “Client/server Shared Access”.

Tip

dbserver is a light-weight program and is designed to be run on devices. Devices that are often disconnected from the network can use **dbserver** to store data locally and provide remote access to data stored on the device.

Database files are created in the current working directory from which **dbserver** is run. Clients will have access to all database files in the current directory and all subdirectories. On Windows CE, the full path to the database file must always be given. The `--restrict` option can be used to restrict access to a list of database files.

By default, **dbserver** only accepts connections from clients running on the same device. To accept connections from other devices or computers on the network, use the `--bind-addr` option to specify which network adapter should be allowed to connect.

Important

In a production environment, always use `--restrict` if the `--bind-addr` option is used. Otherwise, any network client may be able to access any file on the system with the same privileges as the user who runs **dbserver**.

On Windows CE, the `dbserver` command is named `dbservercmd.exe`.

Options

Server Options:

- | | |
|---|---|
| <code>-t, --threads=<i>threadnum</i></code> | Maximum number of simultaneous connections. |
| <code>-N, --no-restrict</code> | Allow clients to open and create databases in any location (default). |
| <code>-R, --restrict</code> | Restrict access to the specified database files. |

TCP Options:

-ba, --bind-addr=*address* Bind to a network adapter to allow remote access. The default value is *127.0.0.1*, which only allows access from the same device. Use the value *0.0.0.0* to accept connections from all network adapters.

-bp, --bind-port=*port* Listen for a connection on the given TCP port. The default port number is *16534*.

Database Options:

-b, --buffers=*bufnum* The number of buffers in the database cache.

-c, --checkpoint=*chknum* Checkpoint interval.

-p, --preallocate=*preallocnum* Number of journal pages to preallocate.

-n, --name=*name* Database alias.

-r, --reset Reset database parameters to default values.

Environment Options:

-l, --locks=*locknum* Expected number of locks.

-w, --owners=*ownnum* Number of lock owners.

-o, --objects=*objnum* Number of objects locked.

-tr, --transactions=*transnum* Number of concurrent transactions expected.

Help Options:

-?, --help Displays complete command line option list.

-u, --usage Displays brief usage message.

Name

dbserver GUI -- Graphical interface to run the database server (Windows CE only).

dbserver GUI

Description

On Windows CE, **dbserver.exe** is a graphical user interface to run and manage the data server.

Important

dbservercmd.exe must be located in the same directory as **dbserver.exe**.

Name

ittiasql -- SQL interactive query prompt

ittiasql
ittiasql [*database...*]

Description

ittiasql is an interactive command-driven program that allows you to use common SQL syntax to **create**, **query**, **modify**, and **delete** ITTIA DB databases and tables. This utility is extremely useful for testing SQL commands and syntax before compiling them into your programs.

Options

ittiasql take the following command line parameters:

[*database*] The name of a database to open. A database can also be opened interactively with the **.open** command.

Interactive Commands

The following commands are available within an **ittiasql** session:

Session Commands

- .help** Displays a list of interactive commands.
- .exit** Closes any open database and exits from **ittiasql**.

Database Commands

- .open** *data-*
base Opens an existing database file.
Arguments: *database* - Name of database file to open.
- .create**
database Creates a new database file.
Arguments: *database* - Name of the database to be created. If there is a database currently opened with the same name, the user is prompted to overwrite the file.
- .close** Closes the current open database.

Table Commands

- .list** Displays a list of all tables in the current open database.

Arguments: none.

.describe *table_name*

Displays a table's schema.

Arguments: *table_name* - Name of the table to describe.

Example Session

Welcome to ittiasql.

All SQL Commands end with a semi-colon (;).

Non-sql userinputs all start with a period (.).

No ending semi-colon (;) is required for non-sql userinputs.

Type '.help' for help.

```
ittia_sql$ .open phone_book.db
phone_book.db$ .list
```

```
+-----+-----+
|TABLE_ID|TABLE_NAME|
+-----+-----+
| 13     |CONTACT   |
| 16     |PHONE_NUMBER|
| 22     |TEST_TYPES|
+-----+-----+
```

```
phone_book.db$ select A.name, B.number from contact A, phone_number B
                -$ where A.id = B.contact_id
                -$ order by A.name;
```

```
+-----+-----+
|NAME   |NUMBER   |
+-----+-----+
|Bob    |206-555-1000|
|Fred   |206-555-1308|
|Fred   |206-555-2335|
|Fred   |206-555-5361|
|Susan  |206-555-3890|
+-----+-----+
```

```
phone_book.db$ .close
ittia_sql$ .exit
c:\>
```

Index

Symbols

2PL, 72

A

ACID, 70
apply_seek, 44
ARIES, 71

B

begin_seek, 44
BLOB data type, 45, 61

C

cancel, 44
client/server,
close
 db::Table, 43
conventions, SQL grammar, 20
create
 db::Database, 39
CREATE INDEX, 17
CREATE SEQUENCE, 17
CREATE TABLE, 17
create_sequence, 42
create_table, 42

D

Database, 39
database concepts, 5
db namespace
 Database, 39
 Field, 41
 Query, 46
 Sequence, 44
db_abort_tx, 53
db_alloc_cursor_row, 55
DB_ALLOC_INITIALIZER, 50
db_alloc_param_row, 62
db_alloc_row, 54, 59
db_begin_tx, 53
DB_BIND_ABSOLUTE, 56
DB_BIND_ADDRESS, 56
db_bind_field, 57
DB_BIND_OFFSET, 58
DB_BIND_RELATIVE, 58
DB_BIND_SIZE, 58
db_bind_t, 56, 58, 59
db_blob_t, 61
DB_CAN_MODIFY, 54
db_close_cursor, 54
db_close_row, 59
db_commit_tx, 53

db_create_file_storage, 49
db_create_index, 52
db_create_sequence, 53
db_create_table, 52
db_delete, 60
db_done, 49
db_eof, 60
db_execute, 62
DB_FAIL, 49
DB_FAILED, 39
db_fielddef_t, 52
 field_type, 50
db_free_row, 54
db_get_field_*, 57
db_indexfield_t, 52
db_index_def_t, 52
db_init, 49
db_insert, 59
DB_LEN_FAIL, 49
DB_LOCK_EXCLUSIVE, 54
DB_LOCK_SHARED, 54
DB_OID_INITIALIZER, 50
DB_OK, 49
db_open_file_storage, 49
db_open_table_cursor, 54
db_prepare_sql_cursor, 62
db_result_t, 49
db_seek_first, 60
db_seek_next, 60
db_seqdef_t, 53
db_set_field_*, 57
db_shutdown, 49
DB_SUCCESS, 39
db_table_cursor_destroy, 54
db_table_cursor_init, 54
db_table_cursor_t, 54
db_unexecute, 62
db_update, 60
DB_VARTYPE_BLOB, 61
DB_WTIME_FAIL, 49
drop_sequence, 42
drop_table, 42

E

edit, 45
eMbedded Visual C++,
eVC,
example,
execute, 46
exec_direct, 46

F

features, 1
Field, 41

G

GCC,

get_db_error, 49
grammar conventions, 20

I

identifier, 21
index locking, 72, 72
insert, 44
INSERT INTO, 18
inter-process communications,
IPC,
isolation level, 67, 67
is_eof, 44
ITTIA
 platforms,
 reference for database characteristics, 70
ittia/db++.h, 39
ittia/db.h, 49

K

keywords, 21

L

locking, 72
 types of locks, 72
logging, 71

M

Manual locking, 73
multi-process,
multi-threading,

O

open
 db::Database, 39
 db::Table, 43

P

param, 46
phonebook,
platform,
platform
 MontaVista linux,
 VxWorks,
 Windows,
post, 44, 45
prepare, 46

Q

Query, 46

R

read_blob, 45
remove, 45
reserved keywords, 21
rollback, 71

row lock, 72

S

sample code,
seek_first, 44
seek_next, 44
Sequence, 44
set_db_error, 49
set_sort_order, 44, 44
shared access, 65
 multithreaded, 66
SQL keywords, 21
SQL syntax conventions, 20
syntax conventions, 20

T

table lock, 72
tx_abort, 43
tx_begin, 43
tx_commit, 43

U

unexecute, 46

V

Visual C++, ,
Visual Studio,

W

write_blob, 45