QNX® Neutrino® Realtime Operating System

QNX Database
Developer's Guide

For QNX® Neutrino® 6.4.x

Published under license by:

QNX Software Systems International Corporation
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published April 28, 2009.

Technical support options

To obtain technical support for any QNX product, visit the **Support + Services** area on our website (`www.qnx.com`). You'll find a wide range of support options, including community forums.

Contents

About This Guide

The *QNX Database (QDB) Developer's Guide* accompanies the QDB database server and is intended for application developers.
This table may help you find what you need in this book:

For information about:	See:
QDB Overview	Introduction
QDB command-line options	Starting QDB
Executing SQL statements from the command-line	The QDB Client **qdbc**
Managing a database	QDB Administration
Sample application	QDB Example
Supported data types	QDB Datatypes
Op codes	QDB Op Codes
Writing your own SQL or collation functions	Writing User-Defined Functions
Client API	QDB API reference
SQL commands	SQL reference

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	`something you type`

continued...

Reference	Example
Keyboard keys	Enter
Program output	**`login:`**
Programming constants	NULL
Programming data types	**`unsigned short`**
Programming literals	**`0xFF`**, **`"message string"`**
Variable names	*stdin*
User-interface components	**Cancel**

We use an arrow (→) in directions for accessing menu items, like this:

> You'll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (**`/`**) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

If you have any questions, comments, or problems with a QNX product, please contact Technical Support. For more information, see the How to Get Help chapter of the *Welcome to QNX Momentics* guide or visit our website, **`www.qnx.com`**.

Chapter 1

Introduction

QDB is a small-footprint, embeddable SQL database server that supports most SQL-92 syntax. It is designed as an easy-to-configure Neutrino resource manager. QDB is based on the SQLite project (**`http://www.sqlite.org`**), and inherits many of SQLite's features.

QDB has these features:

- support for most ANSI SQL-92 syntax
- transactions
- concurrent access
- synchronous safe writes
- triggers, views, multiple attached databases
- small footprint
- leverages all benefits of the Neutrino resource manager framework including network access to databases using QNet
- simple API for accessing the database
- result storing for repeated use. Results can also be passed from one application to another.
- in-memory database support
- auto-attach support, to join disparate databases into a single, virtual database

Chapter 2

Starting QDB

Synopsis:

```
qdb [-A] [-c config_file] [-C policy] [-D]
    [-I test] [-n mountpoint] [-N control] [-o option[,option2...]]
    [-P permissions] [-R mode]
    [-r mode] [-s routine]
    [-t timeout] [-T max_timeout] [-vV] [-W time] [-X path]
```

Options:

-A
Turn off exclusive mode: allow other applications to use the database files.

-c *config_file*
Specify a configuration file of databases and policies. See the "Configuration File" section below for more information.

-C *policy*
Specify a database connection sharing policy. The *policy* can be one of:

- **unique**
- **private**
- **reuse**
- **share**

See the "Sharing connections between clients" section below for more information.

-D
Disable shared cache. You should only use this option if you need to debug shared caching.

-I *test*
Perform a database integrity test at startup. The *test* can be one of:

- **none**
- **basic**
- **partial**
- **full**

See the "Database integrity testing" section below for more information.

-n *mountpoint*
The QDB resource manager mountpoint. By default this is **/dev/qdb**.

-N *control*
Name of the database control entry. By default this is **.control**.

-o *option*
Configure miscellaneous options. The options are:

- **unblock**=0|1 — set whether or not to install an unblock handler (that is, to allow a signal to interrupt an SQL operation).
- **threadmax** — the maximum number of threads to allocate to **qdb**; default is 64.
- **threadhi** — the maximum number of threads that can be kept in a blocked state ready to work.
- **threadlo** — the minimum number of threads to be kept in a blocked state ready for work.
  See also *thread_pool_create()* in the *Neutrino Library reference*.
- **tempstore**=*directory* — set the directory name where **qdb** places certain temporary files. You can set this to a **tmpfs** RAM disk location to prevent excessive disk access.
- **bkcopy**=*buffer_size* — set the size of the buffer to use when making a backup or compressing. The default value is 64 KB, and is probably acceptable for most cases.
- **trace** — log SQL statements before QDB executes them. You must set verbosity (**-v**) to six for this feature to work.
- **profile** — log SQL statements after QDB executes them, as well as the time it took to execute them. You can additionally specify the **-W***time* option to log only SQL statements that take more than the specified time, in milliseconds. You must set verbosity (**-v**) to six for this feature to work.

-P *permissions* — Access permissions for the database and backup files. By default this is **0664**.

-R *mode* — Set the database creation and recovery mode. The *mode* can be one of:

- **manual**
- **auto**
- **set**

See the “Database recovery” section below for more information.

-r *mode* — Set the connection recovery mode. The *mode* specifies what happens when a database problem is discovered and corrected. It can be one of:

- **manual** — clients receive ESTALE errors until they disconnect and reconnect.
- **auto** — clients are automatically reconnected, and receive no notification that a problem was detected and repaired.

-s *routine*[*@data*]	Name special collation routines and data. This setting specifies a name (or wildcard pattern) of collation functions which expect the format of data that you will pass in via *qdb_collation()*. only those registered collation functions that match this pattern will get their setup function invoked (since the format of the data must be known to the function). By default, all functions have their startup functions invoked. You can also use the **-s** option to set the initial setup data. For example, **-s cldr@en_US** would set the magic function name to "cldr", and also invoke the setup function with the "en_US" string at startup.
-t *timeout*	Set the busy-wait timeout on database access, in milliseconds. By default, this is 5000 milliseconds. See the "Busy timeout" section below for more information.
-T *max_timeout*	Set the maximum busy-wait timeout on internal database access, in milliseconds. By default, this is 5000 milliseconds. See the "Busy timeout" section below for more information.
-v	Increase output verbosity. Messages are written to **sloginfo**.
-V	Replicate output messages to the console, as well as to **sloginfo**.
-W *time*	Used in conjunction with the **-o profile** option: log only SQL statements that take longer than *time* (specified in milliseconds). The default for *time* is 5000 milliseconds.
-X *path*	Set a script to run when the QDB encounters a corrupt database. See "Handling corrupt databases" below.

Database directory

The QDB database directory can be on any QNX or POSIX filesystem with read/write access (including memory-based filesystems, such as **tmp-fs**). QDB can run from QNX filesystems visible via Qnet, but can *not* run from an NFS filesystem.

Schema files

A schema file contains all the SQL commands to create the database schema the way you want. Here's an example:

```
CREATE TABLE customers(
  customerid INTEGER PRIMARY KEY AUTOINCREMENT,
  firstname  TEXT,
  lastname   TEXT
);
```

Save that content in **/db/customerdb.sql**.

Starting the QDB server

If you have any database schema files (for example, **/db/customerdb.sql**), you need to add them to the QDB configuration file before starting the QDB server. For more information, see "The configuration file" below.

The QDB server must be run as **root**.

For debugging purposes, you should start **qdb** with **-vvvvvvvV** options to get very verbose output. The **v** option is cumulative, with each additional **v** adding a level of verbosity, up to 7 levels. The **-V** option sends output to the console, as well as to **sloginfo**.

Once QDB is running, you can check to see that it sees your databases by running **ls /dev/qdb/**. Using the previous example, we should see a file called **/dev/qdb/customerdb**.

Temporary storage filesystem

The filesystem the QDB uses for temporary storage must support POSIX file locking. File locking is required for database vacuuming.

The QDB checks its temporary storage as follows:

- If the **tempstore** option (**-o tempstore**) is specified on the command line, the QDB checks to see if the specified location:
  - exists
  - is writable
  - is *not* **/dev/shmem**
  - is *not* a link to **/dev/shmem**

  If *all* the above conditions are met, the QDB sets the internal temporary storage to the location specified by the **tempstore** option. If any of the above conditions are not met, the QDB logs errors to the **slog** and fails to start up.

- If no **tempstore** option (**-o tempstore**) is specified on the command line, the QDB uses the environment variable **TMPDIR** to obtain the location it should use for temporary storage. The QBD checks if **TMPDIR** exists and the location specified by this variable:
  - exists
  - is writable
  - is *not* **/dev/shmem**
  - is *not* a link to **/dev/shmem**

  If *all* the above conditions are met, the QDB sets the internal temporary storage to the value of **TMPDIR**. If any of the above conditions are not met, the QDB logs errors to the **slog** and fails to start up.

Auto-attaching databases

You can create a list of databases that you'd like to be combined as if they formed a single database. This is called *auto-attaching* a database. This is useful for breaking up a database into separate pieces for performance reasons (each piece gets its own lock, which makes multi-user access more responsive). It's also useful for moving parts of a database to different storage mediums (such as a RAM filesystem).

The list of databases is read from a configuration file, specified by the **`Auto Attach=`** option. For more information, see "The configuration file" below.

When using the **`Auto Attach`** parameter to attach more than one database to another database (attaching multiple sections to one section) you must make sure that the order in which the sections are listed in the configuration file are the same as the order in which they are listed via the **`Auto Attach`** parameters. The examples below show incorrect and correct lists. To simplify the examples, only the section headings are shown; parameters are not shown.

Incorrect

Note that the section definition order does not match the attach order.

```
[mme_library]
[mme_temp]
[mme_custom]

[mme]
Auto Attach = mme_library
Auto Attach = mme_custom
Auto Attach = mme_temp
```

Correct

Note that the section definition order matches the attach order.

```
[mme_library]
[mme_custom]
[mme_temp]

[mme]
Auto Attach = mme_library
Auto Attach = mme_custom
Auto Attach = mme_temp
```

Correct

Note that the attach order matches the section definition order.

```
[mme_library]
[mme_temp]
[mme_custom]
```

```
[mme]
Auto Attach = mme_library
Auto Attach = mme_temp
Auto Attach = mme_custom
```

Database integrity testing

At startup, QDB tests the integrity of databases, according the **-I** option specified. It will execute statements based on this option, as follows:

- **none**= — don't perform a database integrity check.
- **basic**=';' — verify only that SQLite can parse a string.
- **partial**= 'PRAGMA database_list;' — validate the PRAGMA database list.
- **full**= 'PRAGMA integrity_check;' — validate the database integrity.

The more verification the QDB performs at startup, the greater the time needed for startup. For production environments, you will need to find the optimal balance between the amount of verification required and the time needed to start the QDB.

Testing SQL statements

You can execute SQL statements on your QDB databases from the command-line using the **qdbc** utility. See **qdbc** for more information.

The configuration file

QDB is configured with a single file, which is specified with the **-c** command-line option. If this file is in the same location as the database SQL files (by default this is **/db/**), you can use relative paths in the configuration file to point to schema files and database locations. Otherwise, you need to use absolute paths.

The configuration file is composed of lines of text. Blank lines are ignored, as is any leading or trailing white space. Lines beginning with a **#** character are comments. The configuration file consists of named sections, each defined by a name enclosed in square brackets (**[ ]**). Following each section are parameter lines in the form *key=value*. Parameters apply to the current section.

Each section is the name of a database. This is the name presented under **/dev/qdb**, and that clients use to establish a connection. The database is then configured using the following parameters:

Filename= Set the name of the actual database file. This is the raw SQLite file. It can be an absolute path to any file location, or can be a relative name (in which case it is relative to the directory which holds the configuration file). At startup either this file must exist, or the directory in which it will be created must exist (otherwise

qdb will exit with an appropriate error). If the database file does not exist, it is restored from the newest valid backup if possible, or a blank database file is created.

Schema=
Schema File=

These options describe the initial schema of the database (as SQL commands which are used to create the initial set of tables, indices, views, content, etc) of a new database (if it did not already exist). In the first form, the SQL commands are in the configuration file. The second form names a file (with either an absolute or relative path) containing the SQL commands.

An initial schema is optional; without an initial schema, a new database will just be empty.

Client Schema=

This entry specifies a client schema, which is executed every time a client calls *qdb_connect()*. YOu can use this mechanism to implement cross-database triggers.

Auto Attach=

This entry specifies another database to be attached to the current one (using the SQL **ATTACH DATABASE** statement whenever a database connection is established). The name is the section name of the other database, not a filename. You can specify multiple databases, each on its own **Auto Attach=** line.

Attached databases are a convenience to provide access to tables that are physically stored in a different database file. Facilities exist in QDB to also include attached databases in other maintenance operations, such as backup or vacuum.

See also “Auto-attaching databases” above.

Backup Dir=

This entry specifies a directory which is used to store a backup of the database. UPi can specify multiple directories, each on its own **Backup Dir=** line, and they will be used in rotation to store backup files. This feature ensures that should a backup be interrupted or aborted by a power-failure, another, older, backup is still available.

This directory must exist at startup (though it does not need to contain a valid backup); otherwise **qdb** exits with an appropriate error. If any existing backup files are located in these directories, they are sorted by date and overwritten oldest-to-newest when performing backup operations, and used in newest-to-oldest order to restore a missing or corrupt database.

Compression=

This entry specifies a compression algorithm to apply to backups. The supported options are **none** (for no compression), **lzo** (for LZO compression), or **bzip** (for BZIP2 compression). The *lzo* compression algorithm is fastest, but the *bzip* algorithm offers the highest compression. The compressed files are created

with appropriate extensions added to the original database filename. By default, backup files are *not* compressed.

Collation=
Function=

These entries install user-provided collation (sorting) routines and user functions (scalar or aggregate) routines respectively. The argument format is *tag@library.so*, where *tag* is the symbol name of the function description structure and *library.so* is the name of the shared library containing the code. For more information, see the Writing User-Defined Functions chapter.

QDB checks for the existence of the library and the specified symbol at startup, and exits with an appropriate error if they're not found.

Vacuum Attached=
Backup Attached=
Size Attached=

These entries control what maintenance operations should apply by default to attached databases when a command is issued to the main database. These options can have a value of **TRUE|FALSE**, **YES|NO** or **ON|OFF**. The default setting for each is **FALSE**. You can change the option multiple times within the database section to apply differently to attached databases.

Here's a sample configuration:

```
[db]
Vacuum Attached = TRUE
Auto Attach = db1
Vacuum Attached = FALSE
Auto Attach = db2
```

In this example, a *qdb_vacuum()* operation on **db** will also vacuum **db1** but not **db2**.

You can use the **Backup Attached=TRUE** setting to provide a facility similar to the old ***.bks** files. For more details on the scope of maintenance operations with respect to attached databases, refer to *qdb_vacuum()*, *qdb_backup()*, and *qdb_getdbsize()*.

To create RAM-based databases, point the **Filename=** option to the RAM-disk file.

You can also create temporary databases by defining a database with a **Filename=:memory:** entry. This action creates a private, temporary, in-memory database, visible only in the scope of the database connection. Each connection to such a database has its own temporary file, which is removed when the connection is closed.

Backup Via= This entry specifies an interim directory into which the database is copied as part of the backup. To make sure the database backup is consistent, **qdb** places a read lock on the database while it is copying and compressing it, so the database may be locked a long time if the destination is slow (for example, flash).

For example, you could specify **Backup Via=/dev/shmem**. When backing up, QDB locks the database, copies it to **/dev/shmem**, and then releases the lock. Then, in a second step, **qdb** performs the copy and compress operation into the location specified by **Backup Dir=**, without needing to lock the database.

Compress Via=TRUE|FALSE

This entry is used in conjunction with the **Backup Via=** entry and any **Compression=** setting specified for the backup. By default, the **Backup Via=** makes a raw/uncompressed copy of the database into the temporary directory, and then performs the compression at the second step. This works if you have space, and read-locks the database for the least amount of time, but you can use less space (at the expense of more time) by compressing during the first copy. FALSE is the default; if you make this setting TRUE, then compression is done in the first step.

Sharing connections between clients

You can allow multiple clients to share a database connection. This is controlled by the **-C** option. These modes are:

unique Each individual client request gets a new connection. This mode exists for pre-3.3.1 SQLite libraries, which were not thread-safe in any way.

private Each client has a private persistent connection for its session; this connection is created when the client attaches and destroyed when it detaches. This mode is the backwards-compatible mode; it is also the mode forced when not using the QDB_CONN_DFLT_SHARE flag to *qdb_connect()*.

reuse Like **private**, except that connections are returned to a pool rather than being destroyed, and can be assigned from there to a new client for use in its duration.

share Like **unique**, except a connection pool is also used. This mode multiplexes all clients over a small number of active database connections.

Connection sharing exists because a non-negligable amount of work must be done to establish a database connection (QDB must allocate memory, access files, attach

databases and callback functions, configure connection parameters, and so on), and if clients do not assume any state, then this processing work can be avoided. The QDB server detects if connection parameters have been changed by a client, and restores them when the connection moves in or out of the pool in **`unique`**, **`reuse`** or **`share`** modes.

This connection sharing should be safe (unless the client destructively modifies the environment via SQL, such as by executing a **`DETACH DATABASE`** statement). However, for full backwards compatibility, connection sharing can be overridden on each *qdb_connect()* call, and the default **`libqdb`** access mode is *private*.

If a client is leaving open transactions across multiple calls to *qdb_statement()*, then it needs a dedicated connection (**`private`** or **`reuse`** or should not set the QDB_CONN_DFLT_SHARE flag).

Shared caching

The default startup mode for QDB is with both shared caching and exclusive modes enabled:

- If you want to disable shared caching, you must use the new **`-D`** command-line option.
- For shared caching, the QDB reserves exclusive write privileges to the database. To allow other applications to use the database files, use the new **`-A`** option.

QDB will exit immediately if it is started with shared cache disabled and exclusive mode enabled. For example:

```
# qdb -c /db/qdb.cfg -v -D -Otempstore=/fs/tmpfs -Rset

qdb: Exclusive locking mode requires that shared cache be enabled
```

Advantages of shared caching

Shared caching can both improve performance times and reduce the total amount of memory cache required for multiple connections. Shared caching also reduces the total amount of memory required for multiple database connections, because multiple connections can share the same memory cache.

For example, without shared caching, if 1 MB of memory is required for each database connection, 40 connections require 40 MB of memory. However, with shared cache enabled, these 40 connections can share the same memory cache, allowing you to reduce the memory cache to 25 MB (or another size determined by your environment and performance requirements). Further, with shared cache, there is no duplication in memory, so in the 25 MB of memory you may have almost the entire database, virtually eliminating the need for disk I/O.

Database recovery

The **-R** options controls the recovery actions QDB performs when it encounters a missing or corrupt database file. The options are:

auto In this mode, file manipulation is fully automatic and a best-effort is always made to establish a valid database connection (both at startup and runtime). Files are backed up individually, and restored individually.

A corrupt or missing database file is restored from the most recent, valid backup that can be located. If there is no such backup, then a blank database is recreated from the original schema definition.

manual In this mode, the only action performed is to create a blank database from the original schema definition if the database file is missing at startup. Databases are not restored from backups. If the file is corrupt, the server will not start. If the file is detected, missing, or corrupt at runtime, no access to that database is permitted, and it will not be restored or re-created. This mode is intended to allow the creation of a new system, or to give manual control over error recovery (for example, to preserve the corrupt database for later analysis).

set In this mode, backups of attached databases are treated as a coherent set, so an error with any one of the component databases cause **qdb** to restore a complete and matching set of all database files. This is useful if attached databases refer to each other.

The *set master* is the database that attaches other databases (by using the **Auto Attach** option in the configuration file). The *backup set* contains the set master and all attached databases that have **Backup Attached** enabled. The set master can be backed up incrementally and still belong to the set.

QNX recommends the following in order to back up and restore your databases as a coherent set:

- For the master database (the database to which the other databases are attached), in the QDB configuration file:
  - Set the **`Backup Attached`** option to **`TRUE`**, as follows:

    ```
    Backup Attached = TRUE
    ```

  - List the databases you want to attach. For example:

    ```
    Auto Attach = mme_library
    Auto Attach = mme_custom
    ```

- Use the **`-R set`** option when starting QDB.
- When doing backups, call *qdb_backup()* on the master database with the *scope* argument set to QDB_ATTACH_DEFAULT.

Busy timeout

The two timeout settings are differentiated as follows:

- The **`-t`** option sets the default user-level timeout which applies to each *qdb_connect()* connection, and can be privately modified with *qdb_setbusytimeout()*.
- The **`-T`** option sets the global internal timeout which applies to database connections made without a client context. Examples include verifying existing databases or constructing new databases at startup, and auto-attaching databases.

Handling corrupt databases

The **`-X`** lets you provide **`qdb`** with a program or script to run when it encounters a corrupt database. If the program or script appears to run correctly, **`qdb`** will continue. The program or script is responsible for stopping and starting **`qdb`** if a start of stop is necessary.

Sample script

Below is a sample **`qdb`** startup command with the **`-X`**:

```
# qdb -c /etc/qdb.cfg -X /usr/bin/recover_db.sh
```

Below is a sample script that can be launched by **`qdb`** when it encounters a corrupt database:

```
recover_db.sh:
#!/bin/sh
#
# This script will kill qdb and mme,
```

```
# remove the database files
# on disk, and restart qdb and mme.

slay qdb mme-generic
rm -f /fs/tmpfs/*
rm -f /mnt/qdb_backup/*

# Call an external program
# to launch qdb and the MME.
# /usr/bin/mme-launch

# EOF
```

- To kill **qdb** without killing the script, send SIGTERM (the default for **slay**). With this method **qdb** keeps the thread used by *popen()* to start the script available and logs output until the script quits.
- If you send SIGKILL, **qdb** is killed immediately. The script continues to run but its output is lost.

Maintenance Commands

You can write some maintenance commands to the **/dev/qdb/.control** entry (and read back the result). The current commands supported are (where *DBNAME* is the name of the database):

- **backup** *DBNAME* — make a backup of the database (*qdb_backup()*)
- **vacuum** *DBNAME* — vacuum the database (*qdb_vacuum()*)
- **verify** *DBNAME* — verify database integrity (like the **-I full** command-line option)
- **cancel** *DBNAME* — cancel any in-progress backups (*qdb_bkcancel()*)

Chapter 3

The QDB Client qdbc

Synopsis:

```
qdbc [-a scope] [-B]
 [-d database] [-f format]
 [-q] [-S] [-t timeout] [-V] [-v[v...]] [sql]
```

Options:

-a *scope*
: Set the scope of operation for the **-B**, **-S** and **-V** options. This can be one of:

  - **default** — act on attached databases as specified in the configuration file (honoring the value of the **Vacuum Attached**, **Backup Attached**, and **Size Attached** parameters). This gives backwards-compatible behavior.
  - **all** — always act on any attached databases, regardless of configuration file settings.
  - **none** — act only on the connected database itself, and never on any attached databases.

-B
: Perform a backup (the equivalent of calling *qdb_backup()*). The scope of this operation is determined by the configuration file for the database specified by **-d** or **QDBC_DBNAME**, or by the **-a** option, if specified.

-d *database*
: The database you want to execute the SQL statement or other operation on. If this isn't specified, the value of the **QDBC_DBNAME** environment variable is used.

-f *format*
: Format for the output. If this option isn't specified, the simple format is used by default. Can be one of:

  - **simple** — plain text, including column names, with field data separated by a pipe character (**|**) (default)
  - **html** — HTML-encoded text
  - **sgml** — SGML-encoded text
  - **data** — plain text, without column names, with field data separated by a tab character

-q
: Reset verbosity to quiet mode.

-S
: Print the database size information (the equivalent to calling *qdb_getdbsize()*) for the database specified by **-d** or **QDBC_DBNAME**. The scope of this operation is determined by the database configuration file, or the **-a** option, if specified.

-t *timeout*
: Set the database connection timeout, in ms.

-V	Perform a vacuum operation (the equivalent to calling *qdb_vacuum()*). The scope of this operation is determined by the configuration file for the database specified by **-d** or **QDBC_DBNAME**, or by the **-a** option, if specified.
-v[v...]	Increase verbosity.
sql	An SQL statement you want to run on the specified database. This statement should be quoted, and end in a semicolon. If no SQL statement is specified, **qdbc** enters interactive mode and takes input from the command-line, giving you an **SQL** prompt. When you are finished entering SQL statements, press Ctrl-C to exit.

Description

The QDB Client utility allows you to execute SQL statements on a **qdb** database without having to write code. It also allows you to perform backup, vacuum, and size query operations. This can be useful when developing QDB applications.

The **-B**, **-S**, **-V** and *sql* options are mutually exclusive; you cannot specify more than one.

The result of each SQL statement is displayed on the standard output by **qdbc**, if the **-q** option isn't set. You can also redirect a file containing SQL statements to QDB, for example: **qdbc < sql.txt**. If you enter SQL from a command-line in a terminal, **qdbc** enters interactive mode. In this mode, you can enter as many consecutive SQL statements as you want. Statements entered in interactive mode don't need to be enclosed in quotation marks, but should end in a semicolon.

Chapter 4

QDB Administration

The QDB offers special commands that you can issue to the database to make it easier to administer. You can use these commands to add new databases, delete old ones, perform backups, etc.

Backing up and restoring databases

You can back up databases to permanent storage (or any POSIX filesystem that allows read/write access) in the following ways; by:

- calling *qdb_backup()* from a client application
- using the **-b** option to **qdbc**.
- using the resource manager interface:

```
echo backup dbname >/dev/qdb/.control
```

These methods are affected by options in the QDB configuration file. See the "Configuration File" section of the Starting QDB chapter for more information.

To restore a database, start **qdb** with the **-R** option set to **auto**. For more information about this option, see the "Database Recovery" section in the Starting QDB chapter.

You can cancel a database backup in client code by calling *qdb_bkcancel()*. You can also cancel a backup operation using the resource manager interface:

```
echo cancel >/dev/qdb/.control
```

Chapter 5

QDB Examples

Your QDB client application should perform these general steps:

1. Connect to a database by calling *qdb_connect()*
2. You can now query the database:
   - **2a** Execute a statement on the database by calling *qdb_statement()*.
   - **2b** Get the results of the statement (if any) by calling *qdb_getresult()*.
   - **2c** Use the results by calling *qdb_cell()*.
   - **2d** Free the result by calling *qdb_freeresult()*.
   - **2e** Repeat executing statements and use the results, as required.
3. Close the database connection with *qdb_disconnect()*

Connecting to the database

Connecting to the database requires that you know the name of the database you want to connect to, and you need a database handle that the **qdb** client library links against.

```
qdb_hdl_t *dbhandle;  // The QDB database handle
dbhandle = qdb_connect("/dev/qdb/customerdb", 0)
if (dbhandle == NULL) {
   fprintf(stderr, "Connect failed: %d\n", errno);
}
```

Two threads can share the same database connection, provided they coordinate between themselves. Alternatively, each thread can call *qdb_connect()* and have its own connection.

Executing a Statement

Executing statements against a QDB database requires that you know and follow the QDB-supported SQL syntax, as described in the QDB SQL reference chapter. You must, of course, connect to the database before attempting to execute statements against it. See “Connecting to the database” above.

One example is to run the following query:

```
int rc;
qdb_hdl_t    *dbhandle;
rc = qdb_statement(dbhandle, "SELECT * FROM customers;");
if (rc == -1) {
   char *errmsg;
   errmsg = qdb_geterrmsg(dbhandle);
   fprintf(stderr, "QDB Error: %s\n", errmsg);
}
```

It is important to escape any strings that you pass in to *qdb_statement()*. The reason is that if you pass in the string:

```
SELECT lastname FROM customerdb WHERE lastname='O'Neil';
```

you would get an error, because the string in the WHERE clause would be just 'O' with trailing garbage characters Neil'. The proper way to run that query is:

```
SELECT lastname FROM customerdb WHERE lastname='O''Neil';
```

The second single quotation mark (**'**) is escaped by the first single quotation mark.

Getting the result of a query

Some queries give results, and others don't. For example, the data results for **UPDATE**, **INSERT**, or **DELETE** statements always contain 0 rows. When running a **SELECT** statement, there may or may not be rows that matched your query, so it is always a good idea to make sure that you have data by checking the return value of *qdb_statement()*.

This does not mean that you can't call *qdb_getresult()* for statements with **0** rows in the data result. In fact, it may be the only way to retrieve the result. If the connection was opened with the QDB_CONN_STMT_ASYNC flag bit set, then *qdb_statement()* will return before the statement has been completed. With complex statements this may mean a delayed error.

To help you debug your application, you can use **qdb_printmsg(stdout, result, QDB_FORMAT_SIMPLE)** to print the fetched result to *stdout()* so that you can visualize your data.

Here's an example of getting the results of an operation:

```
qdb_result_t *result;
// requires a statement previously run
result = qdb_getresult(dbhandle);
```

Memory for the results is allocated when the statement is run on the database, so you must free the result structure or you will have memory leaks. Do this by calling *qdb_freeresult()*, as shown in the example later in this chapter. Never call *free()* yourself.

Using a result

A result is a block of memory containing a description of each cell and the cell's data. There are functions that give you easy access to this data:

Function Name	Use
qdb_columns()	Returns the number of columns
qdb_rows()	Returns the number of rows. An empty result will return 0.
qdb_cell_type()	Returns the type of data in a cell (QDB_INTEGER, QDB_REAL, QDB_TEXT, QDB_BLOB, QDB_NULL).

continued...

Function Name	Use
qdb_column_name()	Returns the column name from the database schema
qdb_cell()	Returns the cell data as a **void** pointer that can be cast to the correct type
qdb_column_index()	Gets the column number that matches the passed in name
qdb_cell_length()	Returns the length of a cell's data
qdb_printmsg()	Prints the contents of a result, which can be useful for debugging

Disconnecting from the Server

To disconnect from the server when you no longer need to use it:

```
qdb_disconnect(dbhandle);
```

Example

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

#include <qdb/qdb.h>

/**
 * This sample program connects to the database and does one INSERT and one
 * SELECT.
 *
 * The database name is assumed to be /dev/qdb/customerdb
 * with schema:
 *    CREATE TABLE customers(
 *       customerid INTEGER PRIMARY KEY AUTOINCREMENT,
 *       firstname  TEXT,
 *       lastname   TEXT
 *     );
 */
int main(int argc, char **argv) {
   int rc;
   qdb_handle_t hdl;
   qdb_result_t *res;
   char *errmsg;

   // Connect to the database
   hdl = qdb_connect("/dev/qdb/customerdb", 0);
   if (hdl == NULL){
      fprintf(stderr, "Error connecting to database: %s\n", strerror(errno));
      return EXIT_FAILURE;
```

```
    }

    // INSERT a row into the database.
    rc = qdb_statement(hdl,
       "INSERT INTO customers(firstname, lastname) VALUES('Dan', 'Cardamore');");
    if (rc == -1) {
       errmsg = qdb_geterrmsg(hdl);
       fprintf(stderr, "Error executing INSERT statement: %s\n", errmsg);
       return EXIT_FAILURE;
    }

    // SELECT one row from the database
    // This statement combines the first and last names together into their
    // full name.
    rc = qdb_statement(hdl,
          "SELECT firstname || ' ' || lastname AS fullname FROM customers
            LIMIT 1;");
    if (rc == -1) {
       errmsg = qdb_geterrmsg(hdl);
       fprintf(stderr, "Error executing SELECT statement: %s\n", errmsg);
       return EXIT_FAILURE;
    }
    res = qdb_getresult(hdl); // Get the result
    if (res == NULL) {
       fprintf(stderr, "Error getting result: %s\n", strerror(errno));
       return EXIT_FAILURE;
    }
    if (qdb_rows(res) == 1) {
       printf("Got a customer's full name: %s\n", (char *)qdb_cell(res, 0, 0));
    }
    else {
       printf("No customers in the database!\n");
    }
    // Free the result
    qdb_freeresult(res);

    // Disconnect from the sever
    qdb_disconnect(hdl);

    return EXIT_SUCCESS;
}
```

Chapter 6

Datatypes in QDB

Storage classes

Each value stored in a QDB database (or manipulated by the database engine) has one of the following storage classes:

- **NULL** — a NULL value.
- **INTEGER** — a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes, depending on the magnitude of the value.
- **REAL** — a floating-point value, stored as an 8-byte IEEE floating-point number.
- **TEXT** — a text string, stored using the database encoding (UTF-8).
- **BLOB** — a blob of data, stored exactly as it was input.

Any column in a database except an **INTEGER PRIMARY KEY** may be used to store any type of value. The exception to this rule is described below under "Other Affinity Modes" as strict affinity mode.

All values supplied to QDB, whether as literals embedded in SQL statements or values bound to pre-compiled SQL statements are assigned a storage class before the SQL statement is executed. Under the circumstances described below, the database engine may convert values between numeric storage classes (**INTEGER** and **REAL**) and **TEXT** during query execution.

Storage classes are initially assigned as follows:

- Values specified as literals as part of SQL statements are assigned storage class **TEXT** if they are enclosed by single or double quotes, **INTEGER** if the literal is specified as an unquoted number with no decimal point or exponent, **REAL** if the literal is an unquoted number with a decimal point or exponent, and **NULL** if the value is a **NULL**. Literals with storage class **BLOB** are specified using the **X'ABCD'** notation.

The storage class of a value that is the result of an SQL scalar operator depends on the outermost operator of the expression.

Column affinity

In QDB, the type of a value is associated with the value itself, not with the column or variable in which the value is stored. (This is sometimes called *manifest typing*.) All other SQL databases engines that we are aware of use the more restrictive system of static typing where the type is associated with the container, not the value.

In order to maximize compatibility between QDB and other database engines, QDB supports the concept of "type affinity" on columns. The type affinity of a column is the recommended type for data stored in that column. The key here is that the type is recommended, not required. Any column can still store any type of data, in theory. It is just that some columns, given the choice, will prefer to use one storage class over another. The preferred storage class for a column is called its *affinity*.

Each column in an QDB database is assigned one of the following type affinities:

- **TEXT**
- **NUMERIC**
- **INTEGER**
- **NONE**

A column with **TEXT** affinity stores all data using the storage classes **NULL**, **TEXT** or **BLOB**. If numerical data is inserted into a column with **TEXT** affinity, it is converted to text form before being stored.

A column with **NUMERIC** affinity may contain values using all five storage classes. When text data is inserted into a **NUMERIC** column, an attempt is made to convert it to an integer or real number before it is stored. If the conversion is successful, then the value is stored using the **INTEGER** or **REAL** storage class. If the conversion cannot be performed, the value is stored using the **TEXT** storage class. No attempt is made to convert **NULL** or blob values.

A column that uses **INTEGER** affinity behaves in the same way as a column with **NUMERIC** affinity, except that if a real value with no floating point component (or text value that converts to such) is inserted, it is converted to an integer and stored using the **INTEGER** storage class.

A column with affinity **NONE** does not prefer one storage class over another. It makes no attempt to coerce data before it is inserted.

Determination of column affinity

The type affinity of a column is determined by the declared type of the column, according to the following rules:

1. If the datatype contains the string “INT”, then it is assigned **INTEGER** affinity.
2. If the datatype of the column contains any of the strings “CHAR”, “BLOB”, or “TEXT”, then that column has **TEXT** affinity. Notice that the type **VARCHAR** contains the string “CHAR” and is thus assigned **TEXT** affinity.
3. If the datatype for a column contains the string “BLOB” or if no datatype is specified, then the column has affinity **NONE**.
4. Otherwise, the affinity is **NUMERIC**.

If you create a table using a **CREATE TABLE** *table* **AS SELECT...** statement, then all columns have no datatype specified, and they are given no affinity.

Column affinity example

```
CREATE TABLE t1(
    t  TEXT,
    nu NUMERIC,
    i  INTEGER,
    no BLOB
);
```

```
-- Storage classes for the following row:
-- TEXT, REAL, INTEGER, TEXT
INSERT INTO t1 VALUES('500.0', '500.0', '500.0', '500.0');

-- Storage classes for the following row:
-- TEXT, REAL, INTEGER, REAL
INSERT INTO t1 VALUES(500.0, 500.0, 500.0, 500.0);
```

Comparison expressions

QDB features the binary comparison operators **=**, **<**, **<=**, **>=** and **!=**; **IN**, an operation to test for set membership; and the ternary comparison operator, **BETWEEN**.

The results of a comparison depend on the storage classes of the two values being compared, according to the following rules:

- A value with storage class **NULL** is considered less than any other value (including another value with storage class **NULL**).
- An **INTEGER** or **REAL** value is less than any **TEXT** or **BLOB** value. When you compare an **INTEGER** or **REAL** to another **INTEGER** or **REAL**, a numerical comparison is performed.
- A **TEXT** value is less than a **BLOB** value. When you compare two **TEXT** values, the C library function *memcmp()* is used to determine the result.
- When you compare two **BLOB** values, the result is always determined using *memcmp()*.

QDB may attempt to convert values between the numeric storage classes (**INTEGER** and **REAL**) and **TEXT** before performing a comparison. For binary comparisons, this is done in the cases enumerated below. The term “expression” below refers to any SQL scalar expression or literal other than a column value.

- When a column value is compared to the result of an expression, the affinity of the column is applied to the result of the expression before the comparison takes place.
- When two column values are compared, if one column has INTEGER or NUMERIC affinity and the other does not, the NUMERIC affinity is applied to any values with storage class TEXT extracted from the non-NUMERIC column.
- When the results of two expressions are compared, no conversions occur. The results are compared as they are presented. If a string is compared to a number, the number will always be less than the string.

In QDB, the expression **a BETWEEN b AND c** is equivalent to **a >= b AND a <= c**, even if this means that different affinities are applied to **a** in each of the comparisons required to evaluate the expression.

Expressions of the type **a IN (SELECT b ....)** are handled by the rules enumerated above for binary comparisons (e.g. in a similar manner to **a = b**). For

example, if **b** is a column value and **a** is an expression, then the affinity of **b** is applied to **a** before any comparisons take place.

QDB treats the expression **a IN (x, y, z)** as equivalent to **a = z OR a = y OR a = z**.

A comparison example

```
CREATE TABLE t1(
    a TEXT,
    b NUMERIC,
    c BLOB
);

-- Storage classes for the following row:
-- TEXT, REAL, TEXT
INSERT INTO t1 VALUES('500', '500', '500');

-- 60 and 40 are converted to '60' and '40' and values are compared as TEXT.
SELECT a < 60, a < 40 FROM t1;
1|0

-- Comparisons are numeric. No conversions are required.
SELECT b < 60, b < 600 FROM t1;
0|1

-- Both 60 and 600 (storage class NUMERIC) are less than '500'
-- (storage class TEXT).
SELECT c < 60, c < 600 FROM t1;
0|0
```

Operators

All mathematical operators (which is to say, all operators other than the concatenation operator **||**) apply **NUMERIC** affinity to all operands prior to being carried out. If one or both operands cannot be converted to **NUMERIC**, then the result of the operation is **NULL**.

For the concatenation operator, **TEXT** affinity is applied to both operands. If either operand cannot be converted to **TEXT** (because it is **NULL** or a **BLOB**) then the result of the concatenation is **NULL**.

Sorting, grouping and compound SELECTs

When values are sorted by an **ORDER BY** clause, values with storage class **NULL** come first, followed by **INTEGER** and **REAL** values interspersed in numeric order, followed by **TEXT** values (usually in *memcmp()* order) and, finally, **BLOB** values in *memcmp()* order. No storage class conversions occur before the sort.

When grouping values with the **GROUP BY** clause, values with different storage classes are considered distinct, except for **INTEGER** and **REAL** values, which are considered equal if they are numerically equal. No affinities are applied to any values as the result of a **GROUP BY** clause.

The compound **SELECT** operators **UNION**, **INTERSECT** and **EXCEPT** perform implicit comparisons between values. Before these comparisons are performed, an affinity may be applied to each value. The same affinity, if any, is applied to all values that may be returned in a single column of the compound **SELECT** result set. The affinity applied is the affinity of the column returned by the left-most component **SELECT**s that has a column value (and not some other kind of expression) in that position. If for a given compound **SELECT** column, none of the component **SELECT**s return a column value, no affinity is applied to the values from that column before they are compared.

Other affinity modes

The above sections describe the operation of the database engine in normal affinity mode. QDB features two other affinity modes, as follows:

- *Strict affinity* mode — if a conversion between storage classes is ever required, the database engine returns an error and the current statement is rolled back.
- *No affinity* mode — no conversions between storage classes are ever performed. Comparisons between values of different storage classes (except for **INTEGER** and **REAL**) are always false.

User-defined collation sequences

By default, when QDB compares two text values, the result of the comparison is determined using *memcmp()*, regardless of the encoding of the string. QDB lets you supply arbitrary comparison functions, known as user-defined collation sequences, to be used instead of *memcmp()*. See the chapter Writing User-Defined Functions for more information.

Aside from the default collation sequence **BINARY**, implemented using *memcmp()*, QDB features two extra built-in collation sequences intended for testing purposes, **NOCASE** and **REVERSE**:

- **BINARY** — Compare string data using *memcmp()*, regardless of text encoding.
- **REVERSE** — Collate in the reverse order to **BINARY**.
- **NOCASE** — The same as **BINARY**, except the 26 upper-case characters used by the English language are converted to their lower-case equivalents before the comparison is performed.

Assigning Collation Sequences from SQL

Each column of each table has a default collation type. If a column requires a collation type other than **BINARY**, you can define the collation type by specifying a **COLLATE** clause as part of the **CREATE TABLE** column definition.

Whenever two text values are compared by QDB, a collation sequence is used to determine the results of the comparison according to the following rules.

For binary comparison operators (**=**, **<**, **>**, ≤, and ≥), if either operand is a column, then the default collation type of the column determines the collation sequence to use for

the comparison. If both operands are columns, then the collation type for the left operand determines the collation sequence used. If neither operand is a column, then the **BINARY** collation sequence is used.

The expression **x BETWEEN y and z** is equivalent to **x ≥ y AND x ≤ z**. The expression **x IN (SELECT y ...)** is handled in the same way as the expression **x = y** for the purposes of determining the collation sequence to use. The collation sequence used for expressions of the form **x IN (y, z ...)** is the default collation type of **x** if **x** is a column, or **BINARY** otherwise.

An **ORDER BY** clause that is part of a **SELECT** statement may be assigned a collation sequence to be used for the sort operation explicitly. In this case, the explicit collation sequence is always used. Otherwise, if the expression sorted by an **ORDER BY** clause is a column, then the default collation type of the column is used to determine sort order. If the expression is not a column, then the **BINARY** collation sequence is used.

Collation Sequences Example

The examples below identify the collation sequences that would be used to determine the results of text comparisons that may be performed by various SQL statements. Note that a text comparison may not be required, and no collation sequence used, in the case of numeric, BLOB or NULL values.

```
CREATE TABLE t1(
    a,                  -- default collation type BINARY
    b COLLATE BINARY,   -- default collation type BINARY
    c COLLATE REVERSE,  -- default collation type REVERSE
    d COLLATE NOCASE    -- default collation type NOCASE
);

-- Text comparison is performed using the BINARY collation sequence.
SELECT (a = b) FROM t1;

-- Text comparison is performed using the NOCASE collation sequence.
SELECT (a = d) FROM t1;

-- Text comparison is performed using the BINARY collation sequence.
SELECT (d = a) FROM t1;

-- Text comparison is performed using the REVERSE collation sequence.
SELECT ('abc' = c) FROM t1;

-- Text comparison is performed using the REVERSE collation sequence.
SELECT (c = 'abc') FROM t1;

-- Grouping is performed using the NOCASE collation sequence
-- (i.e. values 'abc' and 'ABC' are placed in the same group).
SELECT count(*) GROUP BY d FROM t1;

-- Grouping is performed using the BINARY collation sequence.
SELECT count(*) GROUP BY (d || '') FROM t1;
```

```
-- Sorting is performed using the REVERSE collation sequence.
SELECT * FROM t1 ORDER BY c;

-- Sorting is performed using the BINARY collation sequence.
SELECT * FROM t1 ORDER BY (c || '');

-- Sorting is performed using the NOCASE collation sequence.
SELECT * FROM t1 ORDER BY c COLLATE NOCASE;
```

Chapter 7

QDB Virtual Machine Opcodes

Each instruction in the virtual machine consists of an opcode and up to three operands named *P1*, *P2* and *P3*. *P1* may be an arbitrary integer. *P2* must be a non-negative integer. *P2* is always the jump destination in any operation that might cause a jump. *P3* is a null-terminated string or NULL. Some operators use all three operands, some use one or two, and some use none.

The virtual machine begins execution on instruction number 0. Execution continues until:

1. a **`Halt`** instruction is seen, or
2. the program counter becomes one greater than the address of last instruction, or
3. there is an execution error.

When the virtual machine halts, all memory that it allocated is released, and all database cursors it may have had open are closed. If the execution stopped due to an error, any pending transactions are terminated, and changes made to the database are rolled back.

The virtual machine also contains an operand stack of unlimited depth. Many of the opcodes use operands from the stack. See the individual opcode descriptions for details.

The virtual machine can have zero or more cursors. Each cursor is a pointer into a single table or index within the database. There can be multiple cursors pointing at the same index or table. All cursors operate independently, even cursors pointing to the same indexes or tables. The only way for the virtual machine to interact with a database file is through a cursor. Instructions in the virtual machine can create a new cursor (Open), read data from a cursor (Column), advance the cursor to the next entry in the table (Next) or index (NextIdx), and many other operations. All cursors are automatically closed when the virtual machine terminates.

The virtual machine contains an arbitrary number of fixed memory locations with addresses beginning at zero and growing upward. Each memory location can hold an arbitrary string. The memory cells are typically used to hold the result of a scalar **`SELECT`** that is part of a larger expression.

The virtual machine contains a single sorter. The sorter is able to accumulate records, sort those records, then play the records back in sorted order. The sorter is used to implement the **`ORDER BY`** clause of a **`SELECT`** statement.

The virtual machine contains a single *list*, which stores a list of integers. This list is used to hold the row IDs for records of a database table that needs to be modified. The **`WHERE`** clause of an **`UPDATE`** or **`DELETE`** statement scans through the table and writes the row ID of every record to be modified into the list. Then the list is played back and the table is modified in a separate step.

The virtual machine can contain an arbitrary number of *sets*. Each set holds an arbitrary number of strings. Sets are used to implement the **`IN`** operator with a constant right-hand side.

The virtual machine can open a single external file for reading. This external read file is used to implement the **COPY** command.

Finally, the virtual machine can have a single set of aggregators. An aggregator is a device used to implement the **GROUP BY** clause of a **SELECT**. An aggregator has one or more slots that can hold values being extracted by the select. The number of slots is the same for all aggregators and is defined by the AggReset operation. At any point in time, a single aggregator is current or "has focus". There are operations to read or write to memory slots of the aggregator in focus. There are also operations to change the focus aggregator and to scan through all aggregators.

Viewing programs generated by QDB

Every SQL statement that QDB interprets results in a program for the virtual machine. However, if you precede the SQL statement with the keyword **EXPLAIN**, the virtual machine doesn't execute the program. Instead, the instructions of the program are returned like a query result. This feature is useful for debugging and for learning how the virtual machine operates, and for profiling an SQL statement. The following is an example of the output from the statement **EXPLAIN DELETE FROM tbl1 WHERE two<20;**:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------
0     Transaction   0      0
1     VerifyCookie  219    0
2     ListOpen      0      0
3     Open          0      3      tbl1
4     Rewind        0      0
5     Next          0      12
6     Column        0      1
7     Integer       20     0
8     Ge            0      5
9     Recno         0      0
10    ListWrite     0      0
11    Goto          0      5
12    Close         0      0
13    ListRewind    0      0
14    OpenWrite     0      3
15    ListRead      0      19
16    MoveTo        0      0
17    Delete        0      0
18    Goto          0      15
19    ListClose     0      0
20    Commit        0      0
```

All you have to do is add the **EXPLAIN** keyword to the front of the SQL statement. But if you use the **.explain** command to **qdb** first, it will set up the output mode to make the program more easily viewable.

You can put the QDB virtual machine in a mode where it will trace its execution by writing messages to standard output; and you can use the non-standard SQL **PRAGMA,** comments to turn tracing on and off. To turn tracing on, enter:

PRAGMA vdbe_trace=on;

You can turn tracing back off by entering a similar statement but changing the value **on** to **off**.

The opcodes

There are currently 125 opcodes defined by the virtual machine. All currently defined opcodes are described in the list below.

AbsValue Treat the top of the stack as a numeric quantity. Replace it with its absolute value. If the top of the stack is NULL, its value is unchanged.

Add Pop the top two elements from the stack, add them together, and push the result back onto the stack. If either element is a string, then it is converted to a double using the *atof()* function before the addition. If either operand is NULL, the result is NULL.

AddImm Add the value *P1* to whatever is on top of the stack. The result is always an integer.

To force the top of the stack to be an integer, just add 0.

AggFinal Execute the finalizer function for an aggregate. *P1* is the memory location that is the accumulator for the aggregate.

P2 is the number of arguments that the step function takes and *P3* is a pointer to the FuncDef for this function. The *P2* argument is not used by this opcode. It is there only to disambiguate functions that can take varying numbers of arguments. The *P3* argument is needed only for the degenerate case where the step function was not previously called.

AggStep Execute the step function for an aggregate. The function has *P2* arguments. *P3* is a pointer to the FuncDef structure that specifies the function. Use memory location *P1* as the accumulator.

The *P2* arguments are popped from the stack.

And Pop two values off the stack. Take the logical AND of the two values and push the resulting boolean value back onto the stack.

AutoCommit Set the database auto-commit flag to *P1* (1 or 0). If *P2* is true, roll back any currently active btree transactions. If there are any active VMs (apart from this one), then the **COMMIT** or **ROLLBACK** statement fails.

This instruction causes the VM to halt.

BitAnd Pop the top two elements from the stack. Convert both elements to integers. Push back onto the stack the bitwise AND of the two elements. If either operand is NULL, the result is NULL.

BitNot Interpret the top of the stack as an value. Replace it with its ones-complement. If the top of the stack is NULL, its value is unchanged.

BitOr Pop the top two elements from the stack. Convert both elements to integers. Push back onto the stack the bitwise OR of the two elements. If either operand is NULL, the result is NULL.

Blob *P3* points to a blob of data *P1* bytes long. Push this value onto the stack. This instruction is not coded directly by the compiler. Instead, the compiler layer specifies an **OP_HexBlob** opcode, with the hexadecimal string representation of the blob as *P3*. This opcode is transformed to an **OP_Blob** the first time it is executed.

Callback Pop *P1* values off the stack and form them into an array. Then invoke the callback function using the newly formed array as the third parameter.

Clear Delete all contents of the database table or index whose root page in the database file is given by *P1*. But, unlike **Destroy**, do not remove the table or index from the database file.

The table being cleared is in the main database file if *P2* is 0. If *P2* is 1, then the table to be cleared is in the auxiliary database file that is used to store tables create using **CREATE TEMPORARY TABLE**.

See also: **Destroy**

Close Close a cursor previously opened as *P1*. If *P1* is not currently open, this instruction is a no-op.

CollSeq *P3* is a pointer to a **CollSeq** struct. If the next call to a user function or aggregate calls *sqlite3GetFuncCollSeq()*, this collation sequence will be returned. This is used by the built-in *min()*, *max()* and *nullif()* functions.

Column Interpret the data that cursor *P1* points to as a structure built using the **MakeRecord** instruction. (See the **MakeRecord** opcode for additional information about the format of the data.) Push onto the stack the value of the *P2*th column contained in the data. If there are fewer than *P2*+1 values in the record, push a NULL onto the stack.

If the **KeyAsData** opcode has previously executed on this cursor, then the field might be extracted from the key rather than the data.

If *P1* is negative, then the record is stored on the stack rather than in a table. If *P1* is -1, the top of the stack is used, if *P1* is -2, the next on the stack is used, and so forth. The value pushed

is always just a pointer into the record that is stored further down on the stack. The column value is not copied. The number of columns in the record is stored on the stack just above the record itself.

If the column contains fewer than *P2* fields, then push a NULL. Or if *P3* is of type *P3*_MEM, then push the *P3* value. The *P3* value will be the default value for a column that has been added using the **`ALTER TABLE ADD COLUMN`** command. If *P3* is an ordinary string, just push a NULL. When *P3* is a string, it is really just a comment describing the value to be pushed, not a default value.

`Concat`

Look at the first *P1*+2 elements of the stack. Append them all together with the lowest element first. The original *P1*+2 elements are popped from the stack if *P2* is 0 and retained if *P2* is 1. If any element of the stack is NULL, then the result is NULL.

When *P1* is 1, this routine makes a copy of the top stack element into memory obtained from *sqliteMalloc()*.

`ContextPop`

Restore the Vdbe context to the state it was in when **`ContextPush`** was last executed. The context stores the last insert row ID, the last statement change count, and the current statement change count.

`ContextPush`

Save the current Vdbe context, so that it can be restored by a **`ContextPop`** opcode. The context stores the last insert row ID, the last statement change count, and the current statement change count.

`CreateIndex`

Allocate a new index in the main database file if *P2* is 0 or in the auxiliary database file if *P2* is 1. Push the page number of the root page of the new index onto the stack.

`CreateTable`

Allocate a new table in the main database file if *P2* is 0 or in the auxiliary database file if *P2* is 1. Push the page number for the root page of the new table onto the stack.

The difference between a table and an index is this: A table must have a 4-byte integer key and can have arbitrary data. An index has an arbitrary key but no data.

See also: **`CreateIndex`**

`Delete`

Delete the record at which the *P1* cursor is currently pointing.

The cursor will be left pointing at either the next or the previous record in the table. If it is left pointing at the next record, then the next **`Next`** instruction will be a no-op. Hence it is OK to delete a record from within a **`Next`** loop.

If the OPFLAG_NCHANGE flag of *P2* is set, then the row change count is incremented (otherwise not).

If *P1* is a pseudo-table, then this instruction is a no-op.

`Destroy`

Delete an entire database table or index whose root page in the database file is given by *P1*.

The table being destroyed is in the main database file if *P2* is 0. If *P2* is 1 then the table to be cleared is in the auxiliary database file that is used to store tables create using **`CREATE TEMPORARY TABLE`**.

If AUTOVACUUM is enabled, then it is possible that another root page might be moved into the newly deleted root page in order to keep all root pages contiguous at the beginning of the database. The former value of the root page that moved — its value before the move occurred — is pushed onto the stack. If no page movement was required (because the table being dropped was already the last one in the database), then a zero is pushed onto the stack. If AUTOVACUUM is disabled, then a zero is pushed onto the stack.

See also: **`Clear`**

`Distinct`

Use the top of the stack as a record created using **`MakeRecord`**. *P1* is a cursor on a table that declared as an index. If that table contains an entry that matches the top of the stack, then fall through. If the top of the stack matches no entry in *P1*, then jump to *P2*.

The cursor is left pointing at the matching entry if it exists. The record on the top of the stack is not popped.

This instruction is similar to **`NotFound`** except that this operation does not pop the key from the stack.

The instruction is used to implement the **`DISTINCT`** operator on **`SELECT`** statements. The *P1* table is not a true index but rather a record of all results that have been produced so far.

See also: **`Found`**, **`NotFound`**, **`IsUnique`**, **`NotExists`**

`Divide`

Pop the top two elements from the stack, divide the first element (what was on top of the stack) from the second element (the next on stack), and push the result back onto the stack. If either element is a string, then it is converted to a double using the *atof()* function before the division. Division by zero returns NULL. If either operand is NULL, the result is NULL.

`DropIndex`

Remove the internal (in-memory) data structures that describe the index named *P3* in database *P1*. This is called after an index is dropped in order to keep the internal representation of the schema consistent with what is on disk.

DropTable Remove the internal (in-memory) data structures that describe the table named *P3* in database *P1*. This opcode is called after a table is dropped in order to keep the internal representation of the schema consistent with what is on disk.

DropTrigger Remove the internal (in-memory) data structures that describe the trigger named *P3* in database *P1*. This is called after a trigger is dropped in order to keep the internal representation of the schema consistent with what is on disk.

Dup Make a copy of the *P1*th element of the stack and push it to the top of the stack. The top of the stack is element 0, so the instruction **Dup 0 0 0** will make a copy of the top of the stack.

If the content of the *P1*th element is a dynamically allocated string, then a new copy of that string is made if *P2* is 0. If *P2* is *note* 0, then just a pointer to the string is copied.

Also see the **Pull** instruction.

Eq Pop the top two elements from the stack. If they are equal, then jump to instruction *P2*. Otherwise, continue to the next instruction.

If the **0x100** bit of *P1* is true and either operand is NULL, then take the jump. If the **0x100** bit of *P1* is clear, then fall through if either operand is NULL.

If the **0x200** bit of *P1* is set and either operand is NULL, then both operands are converted to integers prior to comparison. NULL operands are converted to zero and non-NULL operands are converted to 1. Thus, for example, with **0x200** set, **NULL==NULL** is true, whereas it would normally be NULL. Similarly, **NULL==123** is false when **0x200** is set, but is NULL when the **0x200** bit of *P1* is clear.

The least significant byte of *P1* (mask 0xff) must be an affinity character - 'n', 't', 'i' or 'o' - or 0x00. An attempt is made to coerce both values according to the affinity before the comparison is made. If the byte is 0x00, then numeric affinity is used.

Once any conversions have taken place, and neither value is NULL, the values are compared. If both values are blobs, or both are text, then *memcmp()* is used to determine the results of the comparison. If both values are numeric, then a numeric comparison is used. If the two values are of different types, then they are unequal.

If *P2* is zero, do not jump. Instead, push an integer 1 onto the stack if the jump would have been taken, or a 0 if not. Push a NULL if either operand was NULL.

If *P3* is not NULL, it is a pointer to a collating sequence (a **CollSeq** structure) that defines how to compare text.

Expire

Cause precompiled statements to expire. An expired statement fails with an error code of QDB_SCHEMA if it is ever executed (via *sqlite3_step()*).

If *P1* is 0, then all SQL statements expire. If *P1* is non-zero, then only the currently executing statement is affected.

FifoRead

Attempt to read a single integer from the FIFO and push it onto the stack. If the FIFO is empty push nothing but instead jump to *P2*.

FifoWrite

Write the integer on the top of the stack into the FIFO.

ForceInt

Convert the top of the stack into an integer. If the current top of the stack is not numeric (meaning that is a NULL or a string that does not look like an integer or floating-point number), then pop the stack and jump to *P2*. If the top of the stack is numeric, then convert it into the least integer that is greater than or equal to its current value if *P1* is 0, or to the least integer that is strictly greater than its current value if *P1* is 1.

Found

The top of the stack holds a blob constructed by **MakeRecord**. *P1* is an index. If an entry that matches the top of the stack exists in *P1*, then jump to *P2*. If the top of the stack does not match any entry in *P1* then fall through. The *P1* cursor is left pointing at the matching entry if it exists. The blob is popped off the top of the stack.

This instruction is used to implement the **IN** operator where the left-hand side is a **SELECT** statement. *P1* is not a true index but is instead a temporary index that holds the results of the **SELECT** statement. This instruction just checks to see if the left-hand side of the **IN** operator (stored on the top of the stack) exists in the result of the **SELECT** statement.

See also: **Distinct**, **NotFound**, **IsUnique**, **NotExists**

Function

Invoke a user function (*P3* is a pointer to a Function structure that defines the function) with *P2* arguments taken from the stack. Pop all arguments from the stack and push back the result.

P1 is a 32-bit bitmask indicating whether or not each argument to the function was determined to be constant at compile time. If the first argument was constant, then bit 0 of *P1* is set. This is used to determine whether metadata associated with a user function argument using the *sqlite3_set_auxdata()* API may be safely retained until the next invocation of this opcode.

See also: **AggStep** and **AggFinal**

Ge

This opcode works just like the **Eq** opcode except that the jump is taken if the second element down on the stack is greater than or equal to the top of the stack. See the **Eq** opcode for additional information.

Gosub

Push the current address plus 1 onto the return address stack, then jump to address *P2*.

The return address stack is of limited depth. If too many **OP_Gosub** operations occur without intervening **OP_Returns**, then the return address stack will fill up and processing will abort with a fatal error.

Goto

An unconditional jump to address *P2*. The next instruction executed will be the one at index *P2* from the beginning of the program.

Gt

This works just like the **Eq** opcode except that the jump is taken if the second element down on the stack is greater than the top of the stack. See the **Eq** opcode for additional information.

Halt

Exit immediately. All open cursors, FIFOs, etc. are closed automatically.

P1 is the result code returned by *sqlite3_exec()*, *sqlite3_reset()*, or *sqlite3_finalize()*. For a normal halt, this should be QDB_OK (0). For errors, it can be some other value. If *P1* is non-zero, then *P2* will determine whether or not to rollback the current transaction. Do not roll back if *P2* is OE_Fail. Do the rollback if *P2* is OE_Rollback. If *P2* is OE_Abort, then back out all changes that have occurred during this execution of the VDBE, but do not rollback the transaction.

If *P3* is not null, then it is an error message string.

There is an implied **Halt 0 0 0** instruction inserted at the very end of every program. So a jump past the last instruction of the program is the same as executing **Halt**.

HexBlob

P3 is an UTF-8 SQL hex encoding of a blob. The blob is pushed onto the VDBE stack.

The first time this instruction executes, in transforms itself into a **Blob** opcode with a binary blob as *P3*.

IdxDelete

The top of the stack is an index key built using the **MakeIdxKey** opcode. This opcode removes that entry from the index.

IdxGE

The top of the stack is an index entry that omits the row ID. Compare the top of stack against the index that *P1* is currently pointing to. Ignore the row ID on the *P1* index.

If the *P1* index entry is greater than or equal to the top of the stack then jump to *P2*. Otherwise fall through to the next instruction. In either case, the stack is popped once.

If *P3* is the **"+"** string (or any other non-NULL string), then the index taken from the top of the stack is temporarily increased by an epsilon prior to the comparison. This makes the opcode work like **IdxGT** except that if the key from the stack is a prefix of the key in the cursor, the result is false whereas it would be true with **IdxGT**.

IdxGT

The top of the stack is an index entry that omits the ROWID. Compare the top of stack against the index that *P1* is currently pointing to. Ignore the ROWID on the *P1* index.

The top of the stack might have fewer columns than *P1*.

If the *P1* index entry is greater than the top of the stack then jump to *P2*. Otherwise fall through to the next instruction. In either case, the stack is popped once.

IdxInsert

The top of the stack holds an SQL index key made using the **MakeIdxKey** instruction. This opcode writes that key into the index *P1*. Data for the entry is nil.

This instruction works only for indexes. The equivalent instruction for tables is **OP_Insert**.

IdxIsNull

The top of the stack contains an index entry such as might be generated by the **MakeIdxKey** opcode. This routine looks at the first *P1* fields of that key. If any of the first *P1* fields are NULL, then a jump is made to address *P2*. Otherwise it falls straight through.

The index entry is always popped from the stack.

IdxLT

The top of the stack is an index entry that omits the ROWID. Compare the top of stack against the index that *P1* is currently pointing to. Ignore the ROWID on the *P1* index.

If the *P1* index entry is less than the top of the stack then jump to *P2*. Otherwise fall through to the next instruction. In either case, the stack is popped once.

If *P3* is the **"+"** string (or any other non-NULL string), then the index taken from the top of the stack is temporarily increased by an epsilon prior to the comparison. This makes the opcode work like **IdxLE**.

IdxRowid

Push onto the stack an integer which is the last entry in the record at the end of the index key pointed to by cursor *P1*. This integer should be the row ID of the table entry to which this index entry points.

See also: **Rowid**.

If Pop a single boolean from the stack. If the boolean popped is true, then jump to p2. Otherwise continue to the next instruction. An integer is false if zero, and true otherwise. A string is false if it has zero length, and true otherwise.

If the value popped of the stack is NULL, then take the jump if *P1* is true, and fall through if *P1* is false.

IfMemPos If the value of memory cell *P1* is 1 or greater, jump to *P2*. This opcode assumes that memory cell *P1* holds an integer value.

IfNot Pop a single boolean from the stack. If the boolean popped is false, then jump to P2. Otherwise continue to the next instruction. An integer is false if zero, and true otherwise. A string is false if it has zero length, and true otherwise.

If the value popped of the stack is NULL, then take the jump if *P1* is true and fall through if *P1* is false.

Insert Write an entry into the table of cursor *P1*. A new entry is created if it doesn't already exist or the data for an existing entry is overwritten. The data is the value on the top of the stack. The key is the next value down on the stack. The key must be an integer. The stack is popped twice by this instruction.

If the OPFLAG_NCHANGE flag of *P2* is set, then the row change count is incremented (otherwise not). If the OPFLAG_LASTROWID flag of *P2* is set, then row ID is stored for subsequent return by the *sqlite3_last_insert_row ID()* function (otherwise it's unmodified).

This instruction works only on tables. The equivalent instruction for indexes is **OP_IdxInsert**.

Int64 *P3* is a string representation of an integer. Convert that integer to a 64-bit value and push it onto the stack.

Integer Push the 32-bit integer value *P1* onto the stack.

IntegrityCk Do an analysis of the currently open database. Push onto the stack the text of an error message describing any problems. If there are no errors, push a **ok** onto the stack.

The root page numbers of all tables in the database are integer values on the stack. This opcode pulls as many integers as it can off of the stack and uses those numbers as the root pages.

If *P2* is not zero, the check is done on the auxiliary database file, not the main database file.

This opcode is used for testing purposes only.

IsNull

If any of the top *abs*(*P1*) values on the stack are NULL, then jump to *P2*. Pop the stack *P1* times if *P1* is greater than 0. If *P1* is less than 0, leave the stack unchanged.

IsUnique

The top of the stack is an integer record number. Call this record number *R*. The next on the stack is an index key created using **MakeIdxKey**. Call it *K*. This instruction pops *R* from the stack but it leaves *K* unchanged.

P1 is an index. So it has no data and its key consists of a record generated by OP_MakeRecord where the last field is the row ID of the entry that the index refers to.

This instruction asks if there is an entry in *P1* where the field matches *K* but the row ID is different from *R*. If there is no such entry, then there is an immediate jump to *P2*. If any entry does exist where the index string matches *K* but the record number is not *R*, then the record number for that entry is pushed onto the stack and control falls through to the next instruction.

See also: **Distinct**, **NotFound**, **NotExists**, **Found**

Last

The next use of the **Rowid**, **Column**, or **Next** instruction for *P1* will refer to the last entry in the database table or index. If the table or index is empty and *P2* is greater than 0, then jump immediately to *P2*. If *P2* is 0 or if the table or index is not empty, fall through to the following instruction.

Le

This works just like the **Eq** opcode, except that the jump is taken if the second element down on the stack is less than or equal to the top of the stack. See the **Eq** opcode for additional information.

LoadAnalysis

Read the **sqlite_stat1** table for database *P1* and load the content of that table into the internal index hash table. This will cause the analysis to be used when preparing all subsequent queries.

Lt

This works just like the **Eq** opcode, except that the jump is taken if the second element down on the stack is less than the top of the stack. See the **Eq** opcode for additional information.

MakeRecord

Convert the top *abs*(*P1*) entries of the stack into a single entry suitable for use as a data record in a database table or as a key in an index. The details of the format are irrelevant as long as the **OP_Column** opcode can decode the record later and as long as the *sqlite3VdbeRecordCompare()* function correctly compares two encoded records. Refer to source code comments for the details of the record format.

The original stack entries are popped from the stack if *P1* is greater than 0 but remain on the stack if *P1* is less than 0.

If *P2* is not zero and one or more of the entries are NULL, then jump to the address given by *P2*. This feature can be used to skip a uniqueness test on indexes.

P3 may be a string that is *P1* characters long. The nth character of the string indicates the column affinity that should be used for the nth field of the index key (i.e. the first character of *P3* corresponds to the lowest element on the stack).

The mapping from character to affinity is as follows:

- **`n`** = NUMERIC
- **`i`** = INTEGER
- **`t`** = TEXT
- **`o`** = NONE

If *P3* is NULL, then all index fields have the affinity NONE.

`MakeRecordI` This opcode works just **`OP_MakeRecord`** except that it reads an extra integer from the stack (thus reading a total of *abs(P1+1)* entries) and appends that extra integer to the end of the record as a variant. This results in an index key.

`MemIncr` Increment the integer valued memory cell *P1* by 1. If *P2* is not zero and the result after the increment is exactly 1, then jump to *P2*.

This instruction throws an error if the memory cell is not initially an integer.

`MemInt` Store the integer value *P1* in memory cell *P2*.

`MemLoad` Push a copy of the value in memory location *P1* onto the stack.

If the value is a string, then the value pushed is a pointer to the string that is stored in the memory location. If the memory location is subsequently changed (using OP_MemStore), then the value pushed onto the stack will change too.

`MemMax` Set the value of memory cell *P1* to the maximum of its current value and the value on the top of the stack. The stack is unchanged.

This instruction throws an error if the memory cell is not initially an integer.

`MemMove` Move the content of memory cell *P2* to memory cell *P1*. Any prior content of *P1* is erased. Memory cell *P2* is left containing a NULL.

`MemNull` Store a NULL in memory cell *P1*.

MemStore Write the top of the stack into memory location *P1*. *P1* should be a small integer, since space is allocated for all memory locations between 0 and *P1* inclusive.

After the data is stored in the memory location, the stack is popped once if *P2* is 1. If *P2* is zero, then the original data remains on the stack.

MoveGe Pop the top of the stack and use its value as a key. Reposition cursor *P1* so that it points to the smallest entry that is greater than or equal to the key that was popped from the stack. If there are no records greater than or equal to the key, and *P2* is not zero, then jump to *P2*.

See also: **Found**, **NotFound**, **Distinct**, **MoveLt**, **MoveGt**, **MoveLe**.

MoveGt Pop the top of the stack and use its value as a key. Reposition cursor *P1* so that it points to the smallest entry that is greater than the key from the stack. If there are no records greater than the key, and *P2* is not zero, then jump to *P2*.

See also: **Found**, **NotFound**, **Distinct**, **MoveLt**, **MoveGe**, **MoveLe**.

MoveLe Pop the top of the stack and use its value as a key. Reposition cursor *P1* so that it points to the largest entry that is less than or equal to the key that was popped from the stack. If there are no records less than or equal to the key, and *P2* is not zero, then jump to *P2*.

See also: **Found**, **NotFound**, **Distinct**, **MoveGt**, **MoveGe**, **MoveLt**.

MoveLt Pop the top of the stack and use its value as a key. Reposition cursor *P1* so that it points to the largest entry that is less than the key from the stack. If there are no records less than the key, and *P2* is not zero, then jump to *P2*.

See also: **Found**, **NotFound**, **Distinct**, **MoveGt**, **MoveGe**, **MoveLe**.

Multiply Pop the top two elements from the stack, multiply them together, and push the result back onto the stack. If either element is a string, then it is converted to a double using the *atof()* function before the multiplication. If either operand is NULL, the result is NULL.

MustBeInt Force the top of the stack to be an integer. If the top of the stack is not an integer and cannot be converted into an integer with out data loss, then jump immediately to *P2*, or if *P2* is 0, raise a QDB_MISMATCH exception.

If the top of the stack is not an integer and *P2* is not zero and *P1* is 1, then the stack is popped. In all other cases, the depth of the stack is unchanged.

`Ne`

This works just like the **`Eq`** opcode, except that the jump is taken if the operands from the stack are not equal. See the **`Eq`** opcode for additional information.

`Negative`

Treat the top of the stack as a numeric quantity. Replace it with its additive inverse. If the top of the stack is NULL, its value is unchanged.

`NewRowid`

Get a new integer record number (*rowid*) used as the key to a table. The record number is not previously used as a key in the database table that cursor *P1* points to. The new record number is pushed onto the stack.

If *P2* is greater than 0, then *P2* is a memory cell that holds the largest previously generated record number. No new record numbers are allowed to be less than this value. When this value reaches its maximum, a QDB_FULL error is generated. The *P2* memory cell is updated with the generated record number. This *P2* mechanism is used to help implement the AUTOINCREMENT feature.

`Next`

Advance cursor *P1* so that it points to the next key/data pair in its table or index. If there are no more key/data pairs, then fall through to the following instruction; if the cursor advance was successful, jump immediately to *P2*.

See also: **`Prev`**

`Noop`

Do nothing. This instruction is often useful as a jump destination.

`Not`

Interpret the top of the stack as a boolean value, and replace it with its complement. If the top of the stack is NULL, its value is unchanged.

`NotExists`

Use the top of the stack as a integer key. If a record with that key does not exist in table of *P1*, then jump to *P2*. If the record does exist, then fall through. The cursor is left pointing to the record if it exists. The integer key is popped from the stack.

The difference between this operation and **`NotFound`** is that this operation assumes the key is an integer and that *P1* is a table whereas **`NotFound`** assumes key is a blob constructed from **`MakeRecord`** and *P1* is an index.

See also: **`Distinct`**, **`Found`**, **`NotFound`**, **`IsUnique`**.

NotFound The top of the stack holds a blob constructed by MakeRecord. *P1* is an index. If no entry exists in *P1* that matches the blob, then jump to *P1*. If an entry does existing, fall through. The cursor is left pointing to the entry that matches. The blob is popped from the stack.

The difference between this operation and **Distinct** is that **Distinct** does not pop the key from the stack.

See also: **Distinct**, **Found**, **NotExists**, **IsUnique**.

NotNull Jump to *P2* if the top *P1* values on the stack are all not NULL. Pop the stack if *P1* times if *P1* is greater than zero. If *P1* is less than zero, then leave the stack unchanged.

Null Push a NULL onto the stack.

NullRow Move the cursor *P1* to a null row. Any **OP_Column** operations that occur while the cursor is on the null row will always push a NULL onto the stack.

OpenPseudo Open a new cursor that points to a fake table that contains a single row of data. Any attempt to write a second row of data causes the first row to be deleted. All data is deleted when the cursor is closed.

A pseudo-table created by this opcode is useful for holding the NEW or OLD tables in a trigger.

OpenRead Open a read-only cursor for the database table whose root page is *P2* in a database file. The database file is determined by an integer from the top of the stack. A 0 means the main database and a 1 means the database used for temporary tables. Give the new cursor an identifier of *P1*. The *P1* values need not be contiguous, but all *P1* values should be small integers. It is an error for *P1* to be negative.

If *P2* is 0, then take the root page number from the next of the stack.

There will be a read lock on the database whenever there is an open cursor. If the database was unlocked prior to this instruction then a read lock is acquired as part of this instruction. A read lock allows other processes to read the database but prohibits any other process from modifying the database. The read lock is released when all cursors are closed. If this instruction attempts to get a read lock but fails, the script terminates with a EBUSY error code.

The *P3* value is a pointer to a **KeyInfo** structure that defines the content and collating sequence of indexes. *P3* is NULL for cursors that are not pointing to indexes.

See also **OpenWrite**.

OpenVirtual Open a new cursor *P1* to a transient or virtual table. The cursor is always opened for reading and writing, even if the main database is read-only. The transient or virtual table is deleted automatically when the cursor is closed.

P2 is the number of columns in the virtual table. The cursor points to a BTree table if *P3* is 0, and to a BTree index if *P3* is not 0. If *P3* is not NULL, it points to a **KeyInfo** structure that defines the format of keys in the index.

OpenWrite Open a read/write cursor named *P1* on the table or index whose root page is *P2*. If *P2* is 0, then take the root page number from the stack.

The *P3* value is a pointer to a **KeyInfo** structure that defines the content and collating sequence of indexes. *P3* is NULL for cursors that are not pointing to indexes.

This instruction works just like **OpenRead**, except that it opens the cursor in read/write mode. For a given table, there can be one or more read-only cursors or a single read/write cursor, but not both.

See also **OpenRead**.

Or Pop two values off the stack. Take the logical OR of the two values and push the resulting boolean value back onto the stack.

ParseSchema Read and parse all entries from the QDB_MASTER table of database *P1* that match the **WHERE** clause *P3*.

This opcode invokes the parser to create a new virtual machine, then runs the new virtual machine. It is thus a reentrant opcode.

Pop Pop *P1* elements off the top of the stack and discarded.

Prev Back up cursor *P1* so that it points to the previous key/data pair in its table or index. If there is no previous key/value pair, then fall through to the following instruction. If the cursor backup was successful, then jump immediately to *P2*.

Pull Remove the *P1*th element from its current location on the stack and push it back on top of the stack. The top of the stack is element 0, so **Pull 0 0 0** is a no-op. **Pull 1 0 0** swaps the top two elements of the stack.

See also the **Dup** instruction.

Push Overwrite the value of the *P1*th element down on the stack (*P1* is 0 is the top of the stack) with the value of the top of the stack. Then pop the top of the stack.

ReadCookie Read cookie number *P2* from database *P1* and push it onto the stack. A value of *P2*==0 is the schema version, while *P2*==1 is the database format. *P2*==2 is the recommended pager cache size, and so forth. *P1*==0 is the main database file and *P1*==1 is the database file used to store temporary tables.

There must be a read-lock on the database (either a transaction must be started or there must be an open cursor) before executing this instruction.

Real The string value *P3* is converted to a real and pushed on to the stack.

Remainder Pop the top two elements from the stack, divide the first (the element that was on top of the stack) from the second (the element that was next on the stack) and push the remainder after division onto the stack. If either element is a string, then it is converted to a double using the *atof()* function before the division. Division by zero returns NULL. If either operand is NULL, the result is NULL.

ResetCount This opcode resets the VM's internal change counter to 0. If *P1* is true, then the value of the change counter is copied to the database handle change counter (returned by subsequent calls to *sqlite3_changes()*) before it is reset. This is used by trigger programs.

Return Jump immediately to the next instruction after the last unreturned **OP_Gosub**. If an OP_Return has occurred for all **OP_Gosub**, then processing aborts with a fatal error.

Rewind The next use of the **Rowid**, **Column**, or **Next** instruction for *P1* will refer to the first entry in the database table or index. If the table or index is empty and *P2*>0, then jump immediately to *P2*. If *P2* is 0 or if the table or index is not empty, fall through to the following instruction.

RowData Push onto the stack the complete row data for cursor *P1*. There is no interpretation of the data. It is just copied onto the stack exactly as it is found in the database file.

If the cursor is not pointing to a valid row, a NULL is pushed onto the stack.

Rowid Push onto the stack an integer which is the key of the table entry that *P1* is currently pointing to.

RowKey Push onto the stack the complete row key for cursor *P1*. There is no interpretation of the key. It is just copied onto the stack exactly as it is found in the database file.

If the cursor is not pointing to a valid row, a NULL is pushed onto the stack.

Sequence Push onto the stack an integer which is the next available sequence number for cursor *P1*. The sequence number on the cursor is incremented after the push.

SetCookie Write the top of the stack into cookie number *P2* of database *P1*. A value of *P2*==0 indicates the schema version, while a value of *P2*==1 indicates the database format. *P2*==2 is the recommended pager cache size, and so forth. *P1*==0 is the main database file and *P1*==1 is the database file used to store temporary tables.

A transaction must be started before executing this opcode.

SetNumColumns Before the **OP_Column** opcode can be executed on a cursor, this opcode must be called to set the number of fields in the table.

This opcode sets the number of columns for cursor *P1* to *P2*.

If **OP_KeyAsData** is to be applied to cursor *P1*, it must be executed before this op-code.

ShiftLeft Pop the top two elements from the stack, convert both elements to integers, and push back onto the stack the second element shifted left by *N* bits, where *N* is the top element on the stack. If either operand is NULL, the result is NULL.

ShiftRight Pop the top two elements from the stack, convert both elements to integers, andush back onto the stack the second element shifted right by *N* bits, where *N* is the top element on the stack. If either operand is NULL, the result is NULL.

Sort This opcode does exactly the same thing as **OP_Rewind**, except that it increments an undocumented global variable used for testing.

Sorting is accomplished by writing records into a sorting index, then rewinding that index and playing it back from beginning to end. We use the **OP_Sort** opcode instead of **OP_Rewind** to do the rewinding so that the global variable will be incremented and regression tests can determine whether or not the optimizer is correctly optimizing out sorts.

Statement Begin an individual statement transaction which is part of a larger BEGIN..COMMIT transaction. This opcode is needed so that the statement can be rolled back after an error without having to roll back the entire transaction. The statement transaction will automatically commit when the VDBE halts.

The statement is begun on the database file with index *P1*. The main database file has an index of 0, and the file used for temporary tables has an index of 1.

`String` The string value *P3* is pushed onto the stack. If *P3* is 0, then a NULL is pushed onto the stack. *P3* is assumed to be a null-terminated string encoded with the database native encoding.

`String8` *P3* points to a null-terminated UTF-8 string. This opcode is transformed into an **`OP_String`** before it is executed for the first time.

`Subtract` Pop the top two elements from the stack, subtract the first (the element that was on top of the stack) from the second (the element that was next on the stack) and push the result back onto the stack. If either element is a string, then it is converted to a double using the *atof()* function before the subtraction. If either operand is NULL, the result is NULL.

`ToBlob` Force the value on the top of the stack to be a BLOB. If the value is numeric, convert it to a string first. Strings are simply reinterpreted as blobs with no change to the underlying data.

A NULL value is not changed by this routine; it remains NULL.

`ToInt` Force the value on the top of the stack to be an integer. If the value is currently a real number, drop its fractional part. If the value is text or blob, try to convert it to an integer using the equivalent of *atoi()* and store 0 if no such conversion is possible.

A NULL value is not changed by this routine. It remains NULL.

`ToNumeric` Force the value on the top of the stack to be numeric (either an integer or a floating-point number. If the value is text or blob, try to convert it to an using the equivalent of *atoi()* or *atof()* and store 0 if no such conversion is possible.

A NULL value is not changed by this routine. It remains NULL.

`ToText` Force the value on the top of the stack to be text. If the value is numeric, convert it to an using the equivalent of *printf()*. Blob values are unchanged and are afterwards simply interpreted as text.

A NULL value is not changed by this routine. It remains NULL.

`Transaction` Begin a transaction. The transaction ends when a **`Commit`** or **`Rollback`** opcode is encountered. Depending on the **`ON CONFLICT`** setting, the transaction might also be rolled back if an error is encountered.

P1 is the index of the database file on which the transaction is started. Index 0 is the main database file and index 1 is the file used for temporary tables.

If *P2* is non-zero, then a write transaction is started. A RESERVED lock is obtained on the database file when a write transaction is started. No other process can start another write transaction while this transaction is underway. Starting a write transaction also creates a rollback journal. A write transaction must be started before any changes can be made to the database. If *P2* is 2 or greater, then an EXCLUSIVE lock is also obtained on the file.

If *P2* is zero, then a read lock is obtained on the database file.

`Vacuum` Vacuum the entire database. This opcode will cause other virtual machines to be created and run. It may not be called from within a transaction.

`Variable` Push the value of variable *P1* onto the stack. A variable is an unknown in the original SQL string as handed to *sqlite3_compile()*. Any occurrence of the **`?`** character in the original SQL is considered a variable. Variables in the SQL string are number from left to right beginning with 1. The values of variables are set using the *sqlite3_bind()* API.

`VerifyCookie` Check the value of global database parameter number 0 (the schema version) and make sure it is equal to *P2*. *P1* is the database number, which is 0 for the main database file, 1 for the file holding temporary tables, and some higher number for auxiliary databases.

The cookie changes its value whenever the database schema changes. This operation is used to detect when the cookie has changed and the current process needs to reread the schema.

Either a transaction needs to have been started or an **`OP_Open`** needs to be executed (to establish a read lock) before this opcode is invoked.

Chapter 8

Writing User-Defined Functions

There are two types of user-defined functions you can write for QDB to use: functions that transform some data (called *scalar* or *aggregate* functions), and functions that order data (called *collation* functions). The first type is invoked using the **SELECT** SQL statement, while the second by using the **COLLATE** clause. An example of a built in scalar function is *ABS()*, while *BINARY()* is an example of a built in collation function.

To define functions that QDB can use, you need to compile them into a DLL. You then tell QDB to load the DLL by setting the **Collation** and **Function** options in the QDB configuration file for each required function.

User scalar/aggregate functions

These are specified in the configuration file with the **Function** = *tag@library.so* option, where *library.so* is the name of a DLL containing your code (this can be an absolute path or a filename within the **LD_LIBRARY_PATH** search) and *tag* is the name of the **struct qdb_function** entry describing the function. This is set up as follows:

```
static void myfunc(sqlite3_context *context, int narg, sqlite3_value **value)
    {
    }
struct qdb_function ftag = { "func", SQLITE_UTF8, 1, NULL, myfunc, NULL, NULL };
```

The tag value in this case is **ftag**, the function name as visible to SQL is **func**, and the function called is *myfunc()*, which can retrieve the 4th field (here NULL) as its *sqlite3_user_data()*.

The **ftag** was used to clarify the example. You would probably use the name **func** here so it was the same as the SQL name.

There can be multiple functions defined (in the same or different DLLs), but each must have a **Function=** entry in the configuration file for the database it is associated with, and each must have a **struct qdb_function** with a unique name describing it.

The **qdb_function** structure has these members:

```
struct qdb_function {
 char       *name;
 int        encoding;
 int        narg;
 void       *arg;
 void       (*func)(struct sqlite3_context *, int, struct Mem **);
 void       (*step)(struct sqlite3_context *, int, struct Mem **);
 void       (*final)(struct sqlite3_context *);
};
```

name — The name used for this function in SQL statements. This is limited to 255 bytes, exclusive of the zero-terminator, and it can't contain any special tokens, or start with a digit. Any attempt to

create a function with an invalid name will result in an SQLITE_ERROR error.

encoding — The character encoding of strings passed to your function. Can be one of:

- SQLITE_UTF8
- SQLITE_UTF16
- SQLITE_UTF16BE
- SQLITE_UTF16LE

narg — The number of arguments that the function or aggregate takes. If this argument is -1, then the function or aggregate may take any number of arguments. The maximum number of arguments to a new SQL function is 127. A number larger than 127 for the third argument results in an SQLITE_ERROR error.

arg — An arbitrary pointer. The function implementations can gain access to this pointer using the *sqlite_user_data()* API.

func, *step*, *final* — Pointers to your function or aggregate. A scalar function requires an implementation of the *func* callback only; NULL pointers should be passed as the *step* and *final* arguments. An aggregate function requires an implementation of *step* and *final*, and NULL should be passed for *func*. Specifying an inconsistent set of callback values, such as a *func* and a *final*, or an *step* but no *final*, results in an SQLITE_ERROR return.

User collation routines

Collation routines can be used to order results from a **`SELECT`** statement. You can provide your own routine, and tell **`qdb`** to use it by providing the **`COLLATE`** keyword to the **`ORDER BY`** clause.

These routines are specified in the configuration file with the **`Collation`** = *tag@library.so* option, where *library.so* is the name of a DLL object containing your code (this can be an absolute path or a filename within the **LD_LIBRARY_PATH** search) and *tag* is the name of the **`struct qdb_collation`** entry describing the collation. This is set up as follows:

```
static int mysort(void *arg, int l1, const void *s1, int l2, const void *s2)
    {
        return(0);
    }

struct qdb_collation ctag = { "nosort", SQLITE_UTF8, NULL, mysort, NULL };
```

The tag value in this case is **`ctag`**, the collation name as visible to SQL will be **`nosort`**, and the function called is *mysort()*, which is passed in the 3rd field (here

NULL) as its *arg* argument (refer to SQLite docs on *sqlite3_create_collation* for more detail).

The **ctag** was used to clarify the example. You would probably use the name **nosort** here so it was the same as the SQL name.

There can be multiple collation sequences defined (in the same or different DLLs), but each must have a **Collation=** entry in the configuration file for the database it is associated with, and each must have a **struct qdb_collation** of a unique name describing it. This replaces the old mechanism of an array of **qdb_collmodule_list_t** always named **init_coll_list**.

The **qdb_collation** structure has these members:

```
struct qdb_collation {
 char       *name;
 int        encoding;
 void       *arg;
 int        (*compare)(void *, int, const void *, int, const void *);
 int        (*setup)(void *, const void *, int, char **);
};
```

name — The name used for this function in SQL statements. This is limited to 255 bytes, exclusive of the zero-terminator, and it can't contain any special tokens, or start with a digit. Any attempt to create a function with an invalid name will result in an SQLITE_ERROR error.

encoding — The character encoding of strings passed to your function. Can be one of:

- SQLITE_UTF8
- SQLITE_UTF16
- SQLITE_UTF16BE
- SQLITE_UTF16LE

arg — An arbitrary pointer to user data that is passed as the first argument to your function each time it's invoked. The function implementations can gain access to this pointer using the *sqlite_user_data()* API.

compare — A pointer to your collation function.

setup — A pointer to a setup function to allow dynamic configuration of sort order at runtime. See below.

The *setup* function takes this form:

int (*setup)(void ****arg*, const void *** *data*, **int** *nbytes*, **char **** *errmsg*);

The parameters of the setup function are:

void **arg*
: The context pointer. This is the same as the *arg* to the *compare* function, and is passed in from the *arg* element of the **qdb_collation** structure.

const void **data*
int *nbytes*
: The configuration data, used to configure the sort. When invoked from startup, this is **NULL** and **0**. When invoked at runtime, it is the data provided to the *qdb_collation()* function. QDB does not interpret the format in any way; the DLL must cooperate with the caller of *qdb_collation()* to exchange data of a known format.

char ***errmsg*
: A pointer to an error message string that is available to *qdb_geterrmsg()* displayed on failure (actually, from startup QDB will fail it, from runtime *qdb_collation()* will fail and this string will be available to it as *qdb_geterrmsg()*)

The function should return a POSIX *errno*, or EOK if it succeeds.

If a collation entry has a non-NULL *setup* entry, then this is invoked at startup and passed NULL for *data* and 0 for *nbytes*, which it can use as a hint to go into some default configuration. Then, whenever you call *qdb_collation()*, the setup function is invoked with new data.

If a collation has no dynamic configuration, then it can use a NULL setup entry in the **struct qdb_collation**, and it can't be dynamically configured.

Example

Here is an example of a table-driven collation algorithm, which uses the data pointer *arg* to say what table to use. The DLL would have the following entries exported from it:

```
uca_t _en_US_ = { ... };
uca_t _fr_FR_ = { ... };

int UCAsort(void *arg, int l1, const void *s1, int l2, const void *s2) { }

struct qdb_collation en_US = {
     "en_US", SQLITE_UTF8, &_en_US_, UCAsort, NULL };

struct qdb_collation fr_FR = {
     "fr_FR", SQLITE_UTF8, &_fr_FR_, UCAsort, NULL };
```

Note that both collations call the *UCASort()* routine, but they pass in different data pointers (**&_en_US_** vs **&_fr_FR_**), where those are tables inside the DLL that tell it how to sort in English or French. This is passed as the first argument to the function, *arg*.

You would install these to QDB in the configuration file as:

```
[DB]
Collation = en_US@/usr/lib/libqdb_uca.so
Collation = fr_FR@/usr/lib/libqdb_uca.so
```

SQLite C/C++ API

This is an abridged version of the C/C++ API documentation for SQLite, which covers just the functions you might call in user-defined functions. For the full API documentation, see the SQLite website (**www.sqlite.org**).

When consulting SQLite documentation, ensure that it corresponds to the SQLite library version that QDB is using.

sqlite3_result_*

```
void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_error(sqlite3_context*, const char*, int);
void sqlite3_result_error16(sqlite3_context*, const void*, int);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_text16(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_text16be(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_text16le(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_value(sqlite3_context*, sqlite3_value*);
```

User-defined functions invoke these routines in order to set their return value. The *sqlite3_result_value()* routine returns an exact copy of one of the arguments to the function.

Your user-defined function should pass as the first argument the **sqlite3_context*** that was passed to it by QDB.

sqlite3_value_*

```
const void *sqlite3_value_blob(sqlite3_value*);
int sqlite3_value_bytes(sqlite3_value*);
int sqlite3_value_bytes16(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
int sqlite3_value_int(sqlite3_value*);
long long int sqlite3_value_int64(sqlite3_value*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
const void *sqlite3_value_text16(sqlite3_value*);
const void *sqlite3_value_text16be(sqlite3_value*);
const void *sqlite3_value_text16le(sqlite3_value*);
int sqlite3_value_type(sqlite3_value*);
```

This group of routines returns information about arguments to a user-defined function. User-defined function implementations use these routines to access their arguments.

The *sqlite3_value_type()* routine returns one of:

- SQLITE_INTEGER
- SQLITE_FLOAT

- SQLITE_TEXT
- SQLITE_BLOB
- SQLITE_NULL

If the result is a BLOB, then the *sqlite3_value_blob()* routine returns the number of bytes in that BLOB. No type conversions occur. If the result is a string (or a number since a number can be converted into a string), then *sqlite3_value_bytes()* converts the value into a UTF-8 string and returns the number of bytes in the resulting string. The value returned does not include the **\000** terminator at the end of the string. The *sqlite3_value_bytes16()* routine converts the value into a UTF-16 encoding and returns the number of bytes (not characters) in the resulting string. The **\u0000** terminator is not included in this count.

These routines attempt to convert the value where appropriate. For example, if the internal representation is FLOAT, and a text result is requested, *sprintf()* is used internally to do the conversion automatically. The following table details the conversions that are applied:

Internal Type	Requested Type	Conversion
NULL	**INTEGER**	Result is 0
NULL	**FLOAT**	Result is 0.0
NULL	**TEXT**	Result is NULL pointer
NULL	**BLOB**	Result is NULL pointer
INTEGER	**FLOAT**	Convert from integer to float
INTEGER	**TEXT**	ASCII rendering of the integer
INTEGER	**BLOB**	Same as for **INTEGER** to **TEXT**
FLOAT	**INTEGER**	Convert from float to integer
FLOAT	**TEXT**	ASCII rendering of the float
FLOAT	**BLOB**	Same as **FLOAT** to **TEXT**
TEXT	**INTEGER**	Use *atoi()*
TEXT	**FLOAT**	Use *atof()*
TEXT	**BLOB**	No change
BLOB	**INTEGER**	Convert to **TEXT**, then use *atoi()*
BLOB	**FLOAT**	Convert to **TEXT**, then use *atof()*
BLOB	**TEXT**	Add a **\000** terminator if needed

sqlite3_user_data

```
void *sqlite3_user_data(sqlite3_context*);
```

The *arg* member to the **qdb_function** struct used to register user functions is available to the implementation of the function using this call.

Appendix A

QDB Client API Reference

These functions handle operations that directly involve the QDB. Using these functions, your client application can:

- attach to a database session
- set database properties
- create and execute SQL statements
- inspect the results of **SELECT** queries

Synopsis:

```
#include <qdb/qdb.h>

int qdb_backup( qdb_hdt_t *db,
                int scope );
```

Arguments:

db A pointer to the database handle.

scope The scope of the backup. Possible values are:

- QDB_ATTACH_DEFAULT — Act on attached databases as specified in the configuration file (honouring the value of the **`Vacuum Attached`**, **`Backup Attached`**, and **`Size Attached`** parameters. This gives backwards-compatable behavior.
- QDB_ATTACH_ALWAYS — Always act on any attached databases, regardless of configuration file settings.
- QDB_ATTACH_NEVER — Act only on the connected database itself, and never on any attached databses.

Library:

`qdb`

Description:

This function performs a backup on the connected database *hdl*, and optionally any attached databases, depending on the *scope* argument. Backups are controlled in the configuration file, via the **`Backup Dir=`** and **`Compression=`** options. For more information about these options, see the Configuration File section of the chapter Starting QDB.

A client can cancel a backup operation by calling *qdb_bkcancel()*. If a backup is cancelled (either by a client or via the QDB resource manager interface), the call to *qdb_backup()* fails and returns **`-1`**, with *errno* set to EINTR.

Returns:

`>0` Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_bkcancel()

qdb_bkcancel()

Cancel a database backup

Synopsis:

```
#include <qdb/qdb.h>

int qdb_bkcancel( qdb_hdl_t *hdl,
                  int *nactive );
```

Arguments:

hdl A pointer to the database handle.

nactive A pointer to a location where the function stores the number of backup operations that were aborted. You can use this if you want to know if a backup was interrupted and needs to be rescheduled, or set it to NULL if you don't need this information.

Library:

qdb

Description:

This function cancels all active backup operations for any databases on the QDB server associated with the specified *hdl* handle.

Returns:

>0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_backup()

Synopsis:

```
#include <qdb/qdb.h>

void * qdb_cell( qdb_result_t *res,
                 int row,
                 int col );
```

Arguments:

res	A pointer to a result structure to check.
row	The row number of the cell, where the first row is 0.**>**
col	The column number of the cell, where the first column is 0.

Library:

qdb

Description:

This function returns the data from one cell from a database query result. The returned pointer points to the beginning of the data. You must cast the pointer to the appropriate data type. For example:

```
uint64_t storage_type = *(uint64_t*)qdb_cell(res, 0, 0);
```

Returns:

A pointer	A pointer to the beginning of the cell's data.
NULL	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_cell_length(), qdb_cell_type()

qdb_cell_length()

Return the length of a cell's data

Synopsis:

```
#include <qdb/qdb.h>

int qdb_cell_length( qdb_result_t *res,
                     int row,
                     int col );
```

Arguments:

res	A pointer to a result structure to check.
row	The row number of the cell.
col	The column number of the cell.

Library:

qdb

Description:

This function returns the length of a specified cell in a database query result. This is useful for datatypes that are variable-length, such as QDB_TEXT and QDB_BLOB.

For QDB_TEXT, this function does not count the terminating **\0** character.

Returns:

>-1	The length of the specified cell's data, in bytes.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_cell(), qdb_cell_type()

Synopsis:

```
#include <qdb/qdb.h>

int qdb_cell_type( qdb_result_t *res,
                   int row,
                   int col );
```

Arguments:

res A pointer to a result structure to check.

row The row number of the data cell.

col The column number of the data cell.

Library:

qdb

Description:

This function returns the type of the specified cell, which you can use to cast the cell data to the proper C datatype. The datatypes that can be returned are defined in **<qdb/qdb.h>**. They are:

Return Type	ANSI C Type	Variable Length
QDB_UNSUPPORTED	**NULL**	No
QDB_INTEGER	**int64_t**	No
QDB_REAL	**double**	No
QDB_TEXT	**char ***	Yes
QDB_BLOB	**void ***	Yes
QDB_NULL	**NULL**	No

If the data can have variable length, then you should check its length by calling *qdb_cell_length()*. The text type QDB_TEXT (**char ***) is always null-terminated.

Returns:

>-1 The datatype of the specified cell.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_cell(), qdb_cell_length()

qdb_collation()

Change the runtime configuration of user-defined collation sequences

Synopsis:

```
#include <qdb/qdb.h>

int qdb_collation( qdb_hdl_t *db,
                   void *data,
                   int nbytes,
                   int reindex );
```

Arguments:

db A pointer to the database handle.

data A pointer to arbitrary configuration data used by the user-defined collation library.

nbytes The length of *data*, in bytes.

reindex A flag to indicate if QDB should reindex any database indexes that would be affected by changing the collation. If any indexes exist that have a **COLLATE** component, then these must be regenerated to reflect the potentially new sorting order.

Library:

qdb

Description:

This function is used to configure special user-defined collation sequences attached to the database, defined by **Collation=** entries in the configuration file. The *setup()* function of each entry is invoked with the specified *data* and *nbytes*, and any error raised by that function is returned to the client. Otherwise, the collation routine is expected to use the data in a proprietary manner to configure itself to a new sort order. The collation routine and the client must both know what format this configuration data is in. You might consider use strings as a simple self-documenting extensible format (e.g. *getsubopt()* style).

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_query()

qdb_column_index()

Find a column by name

Synopsis:

```
#include <qdb/qdb.h>

int qdb_column_index( qdb_result_t *result,
                      const char *name );
```

Arguments:

result A pointer to a result structure to check.

name The name of the column to get the index number for.

Library:

qdb

Description:

This function returns the index for specified column name, *name*.

Returns:

>-1 The index of the specified column

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_column_name(), qdb_columns()

qdb_column_name()

Return a column's name

Synopsis:

```
#include <qdb/qdb.h>

char * qdb_column_name( qdb_result_t *res,
                        int col );
```

Arguments:

res A pointer to a result structure to check.

col The index of the column name to return.

Library:

qdb

Description:

This function returns the name of a specified column index *col*, as defined in a database schema when the table was created.

Returns:

A pointer — A pointer to the specified column's name.

NULL — An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_column_index(), qdb_columns()

qdb_columns()

Return the number of columns in a result

Synopsis:

```
#include <qdb/qdb.h>

int qdb_columns(qdb_result_t *res);
```

Arguments:

res A pointer a result structure to check.

Library:

qdb

Description:

This function returns the number of columns in the result structure *res*. If your query matches 0 rows, you can still have a value greater than 0 for the number of columns. You should use *qdb_rows()* to determine if the results are empty.

Returns:

>-1 The number of columns in the result set.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_column_index(), *qdb_column_name()*

Synopsis:

```
#include <qdb/qdb.h>

qdb_hdl_t   *qdb_connect( const char *dbname,
                          int flags );
```

Arguments:

dbname The database device name (for example, **/dev/qdb/customerdb**).

flags Flags which can be used to control attributes of the connection. This argument can be **0**, or a combination of:

- QDB_CONN_DFLT_SHARE — Use the default database connection share mode (as given to the **-C** command line option to **qdb**). Without this flag, a private connection is forced.
- QDB_CONN_NONBLOCKING — If this bit is set, *qdb_statement()* fails and returns immediately (setting *errno* to EBUSY) if the database file is locked. By default, *qdb_statement()* waits for at least the busy timeout period (set using *qdb_setbusytimeout()*) if the database is locked, before failing and returning.

  Setting this bit also makes subsequent calls to *qdb_connect()* non-blocking (as if the **-T** commandline option was **0**).
- QDB_CONN_STMT_ASYNC — Execute statements asynchronously. In this mode, *qdb_statement()* may return before the statement has completed execution against the database. See “Using asynchronous mode” below.

Library:

qdb

Description:

This function connects to the database specified by *dbname*, and returns a pointer to the database connection. You need to call this function for every database, or for concurrent access to one database.

Two threads can share the same database connection, provided they coordinate between themselves. Alternatively, each thread can call *qdb_connect()* and have its own connection.

You should disconnect all connections with a call to *qdb_disconnect()* when you’re finished using them.

Using asynchronous mode

By default, QDB completes execution of statements against a database before returning from *qdb_statement()*. However, you can connect to QDB using asynchronous mode by setting the QDB_CONN_STMT_ASYNC in *flags*. While some errors (such as syntax errors) can be caught before *qdb_statement()* returns in this mode, others, such as database constraint violations, may not be generated until the statement is completed. These errors are available only to a subsequent *qdb_getresult()* call.

The advantage of asynchronous operation is that it allows parallelism between the client application and the database engine, especially in cases where the client will later retrieve the statement result anyway (for example, **SELECT** statements). The danger of asynchronous operation is that the client must be aware that the statement may not necessarily have completed or indicated all errors, and must be coded to call *qdb_getresult()* to retrieve any errors.

The mode you should use depends on the type of operation you are doing. If it is primarily **SELECT** statements, then you can use asynchronous mode and let the database engine run, since you are calling back in anyway for the row/results. If you are primarily doing database maintenance (that is, **INSERT**, **UPDATE**, and **DELETE** statements), then you probably want synchronous statement execution so you can just use one API call.

Returns:

A valid pointer to an opaque database connection (**qdb_hdl_t**), NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_disconnect(), qdb_setbusytimeout(), qdb_statement()

qdb_data_source()

Extract the data source for a specific database

Synopsis:

```
#include <qdb/qdb.h>

int qdb_data_source( qdb_hdl_t *db,
                     char *buffer,
                     int buffer_length );
```

Arguments:

db	A pointer to the database handle.
buffer	A buffer to hold the resulting source path information.
buffer_length	The length of *buffer*.

Library:

qdb

Description:

This function provides a path to the source used to initialize the database. This source may be one of several paths, depending on the state of the specified database when **qdb** is started and the database initialized:

- If the database is empty, the string will be empty.
- If the database is created with a schema only, the string will be the path to the schema file used to create the database.
- If the database is created with a schema and initialized with a data schema, then the string will be a colon delimited list of *schema*:*data schema1*[:*data schema2*...]
- If the database is created from an existing database that is not corrupted (and not a backup database), then the string will be the path to that database which will be the same as the Filename entry.
- If the database is created from a backup database, then the string will be the path to the restoring database from one of the Backup Dir entries.

Returns:

>0	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_connect()

qdb_disconnect()

Disconnect from a database

Synopsis:

```
#include <qdb/qdb.h>

int qdb_disconnect( qdb_hdt_t *hdl );
```

Arguments:

hdl A pointer to the database handle to disconnect from.

Library:

qdb

Description:

This function disconnects from a database connected to previously by *qdb_connect()*.

You should disconnect from all connections when you have finished with them.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_connect()

qdb_freeresult()

Free the result of an SQL statement

Synopsis:

```
#include <qdb/qdb.h>

int qdb_freeresult(qdb_result_t *res);
```

Arguments:

res A pointer a result structure to free.

Library:

qdb

Description:

Results returned from *qdb_getresult()* need to be freed using this function when you're finished using them.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_getresult()

Synopsis:

```
#include <qdb/qdb.h>

int qdb_getdbsize( qdb_hdt_t    *hdl,
                   int           scope,
                   uint32_t     *page_size,
                   uint32_t     *total_pages,
                   uint32_t     *free_pages );
```

Arguments:

hdl	A pointer to the database handle.
scope	Describes the scope of the operation. See the description of the *scope* argument to *qdb_backup()* for more information.
page_size	A pointer to a location where the function stores the size (in bytes) of a page in the database file.
total_pages	A pointer to a location where the function stores the number of pages in the database file.
free_pages	A pointer to a location where the function stores the number of pages that aren't being used to store data.

Library:

qdb

Description:

This function fills in arguments with information about the size (in bytes) of the database file associated with the database handle *hdl*. The size of the database on the filesystem is *page_size* × *total_pages*.

If you vacuum the database, **qdb** gets rid of the free pages so that the total pages goes down, free pages goes to 0, and the database file size becomes smaller. For more information, see the **VACUUM** SQL command, *qdb_vacuum()* function, and the **auto_vacuum** section of the PRAGMA command.

For a database to be included in the size count for a database handle, the **Size Attached** option for that database file must be set to **TRUE** in the QDB configuration file.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_vacuum(), PRAGMA, **VACUUM**

Synopsis:

```
#include <qdb/qdb.h>

const char * qdb_geterrmsg(qdb_hdl_t *hdl);
```

Arguments:

hdl A pointer to the database handle.

Library:

qdb

Description:

This function returns a pointer to a string containing an error message from the server for the most recent call to:

- *qdb_statement()*
- *qdb_getresult()*
- *qdb_getoption()*
- *qdb_setoption()*
- *qdb_busytimeout()*
- *qdb_vacuum()*
- *qdb_backup()*
- *qdb_getdbsize()*

You typically call this function after one of the above functions fails. If the error occurred within the SQL library, the returned string is an SQLite error message. If the error occurred in the QDB system, the returned string is a POSIX *errno* message.

Returns:

A pointer to string if there is an error message, or a pointer to an empty string if there is no error.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_backup(), qdb_getdbsize(), qdb_getoption(), qdb_getresult(), qdb_setoption(), qdb_statement(), qdb_vacuum(),

qdb_getoption()

Return the value for a database session option

Synopsis:

```
#include <qdb/qdb.h>

int qdb_getoption( qdb_hdt_t *hdl,
                   int option );
```

Arguments:

hdl A pointer to the database handle.

option The option you'd like to query. See *qdb_setoption()* for a list of database options.

Library:

qdb

Description:

This function returns the value of the *option* for the database *hdl*.

Returns:

≥**0** The value of the option passed, either **0** ("off") or **1** ("on").

-1 The specified option isn't supported (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_setoption()

qdb_getresult()

Return the result of an SQL statement

Synopsis:

```
#include <qdb/qdb.h>

qdb_result_t* qdb_getresult(qdb_hdt_t *hdl);
```

Arguments:

hdl A pointer to the database handle

Library:

qdb

Description:

After running a **SELECT** statement on the database, you can retrieve its result using this function. All rows that matched the query are returned into one result, which is returned as a **qdb_result_t** structure. You can get further information about the result using these functions:

- *qdb_cell()*
- *qdb_cell_length()*
- *qdb_column_index()*
- *qdb_column_name()*
- *qdb_columns()*
- *qdb_printmsg()*
- *qdb_rows()*

The result needs to be freed by calling *qdb_freeresult()* once you've finished using it.

Returns:

A pointer to the query result, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_cell(), qdb_cell_length(), qdb_column_index(), qdb_column_name(), qdb_columns(), qdb_printmsg(), qdb_rows()

qdb_gettransstate()

Return the size of a database

Synopsis:

```
#include <qdb/qdb.h>

int qdb_gettransstate(qdb_hdl_t *hdl);
```

Arguments:

hdl A pointer to the database handle.

Library:

qdb

Description:

This function returns the connection state for the current QDB connection. If the connection is in an SQL transaction, this function returns 1, and 0 if it the connection is *not* an SQL transaction. It returns -1 if there's an SQL error (you can use *qdb_geterrmsg()* to get the error string).

You can use this function to determine how to clean up after an SQL error, for example when you execute several commands including a transaction and need to determine which statement casued the error.

Returns:

=**0** There is no SQL transaction in progress.

≥**1** An SQL transaction is in progress.

-1 An SQL error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_geterrmsg()

qdb_last_insert_rowid()

Return the last inserted row primary key

Synopsis:

```
#include <qdb/qdb.h>

uint64_t qdb_last_insert_rowid( qdb_hdt_t *hdl,
                                qdb_result_t *result );
```

Arguments:

hdl A pointer to the database handle. You can pass as NULL if you provide *result*, and QDB_OPTION_LAST_INSERT_ROWID option has been set by *qdb_setoption()* (it's on by default).

result A pointer to a result set you want to query. If you pass NULL, the function queries the **qdb** server connection *hdl* for the last executed *qdb_statement()*.

Library:

qdb

Description:

Each entry in a QDB table has a unique integer key called the row ID. The row ID is always available as an undeclared column named *ROWID*, *OID*, or *_ROWID_*. If the table has a column of type INTEGER PRIMARY KEY, then that column is another an alias for the *rowid*.

This function returns the row ID of the last **INSERT**. It first looks in *result* (if the QDB_OPTION_LAST_INSERT_ROWID option has been set by *qdb_setoption()*), returning the information for the statement that produced the result. If *result* is NULL, or QDB_OPTION_LAST_INSERT_ROWID is off, the function queries the database handle *hdl* and returns the information about the last executed statement.

If this function returns 0, check *errno* to make sure that it is EOK, indicating that no rows were inserted (you should set *errno* to **0** before calling this function if you want to distinguish between an error and **0** rows). If *errno* is set, then there was an error with the request.

If an **INSERT** occurs within a trigger, then the rowid of the inserted row is returned by this function as long as the trigger is running. But once the trigger terminates, the value returned by this routine reverts to the last value inserted before the trigger fired.

Returns:

> 0 The integer primary key of the last row inserted

0 An error occurred (*errno* is set), or no rows were inserted.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_setoption(), qdb_statement()

qdb_mprintf()

Print formatted output to a new string

Synopsis:

```
#include <qdb/qdb.h>

char * qdb_mprintf( const char* fmt,... );
```

Arguments:

fmt A pointer to a formatting string to process. The formatting string determines what additional arguments you need to provide. For more information, see *printf()* in the Neutrino *Library Reference*.

Library:

qdb

Description:

This function is a variant of *sprintf()* from the standard C library. The resulting string is written into memory obtained from *malloc()*, so there is never a possibility of buffer overflow. This function also implements some additional formatting options that are useful for constructing SQL statements.

The *qdb_statement()* function also allows you to format strings in this way, and doesn't require that you remember to free the resulting string. However, *qdb_mprintf()* may be useful for building queries from multiple strings.

You should call *free()* to free the strings returned by this function.

All the usual *printf()* formatting options apply. In addition, there is a **%q** option. The **%q** option works like **%s**: it substitutes a null-terminated string from the argument list. But **%q** also doubles every **\'** character (every escaped single quotation). **%q** is designed for use inside a string literal. By doubling every **\'** character, it escapes that character and allows it to be inserted into the string.

For example, suppose some string variable contains text as follows:

```
char *zText = "It's a happy day!";
```

You can use this text in an SQL statement as follows:

```
qdb_mprintf(db, "INSERT INTO table VALUES('%q')",
     zText);
```

Because the **%q** format string is used, the **\'** character in *zText* is escaped, and the SQL generated is as follows:

```
INSERT INTO table1 VALUES('It''s a happy day!')
```

This is correct. Had you used **%s** instead of **%q**, the generated SQL would have looked like this:

```
INSERT INTO table1 VALUES('It's a happy day!');
```

This second example is an SQL syntax error. As a general rule, you should always use **%q** instead of **%s** when inserting text into a string literal.

The **%Q** option works like **%q** except that it also adds single quotes around the outside of the total string. Or, if the parameter in the argument list is a NULL pointer, **%Q** substitutes the text **"**NULL**"** (without single quotes) in place of the **%Q** option. So, for example, one could say:

```
char *zSQL = sqlite3_mprintf("INSERT INTO table VALUES(%Q)", zText);
sqlite3_exec(db, zSQL, 0, 0, 0);
sqlite3_free(zSQL);
```

The code above will render a correct SQL statement in the *zSQL* variable even if the *zText* variable is a NULL pointer.

Returns:

An escaped string	Success.
NULL	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_snprintf(), qdb_vmprintf(), printf() in the Neutrino *Library Reference*

qdb_parameters()

Get or set database connection parameters

Synopsis:

```
#include <qdb/qdb.h>

int qdb_parameters( qdb_hdl_t *db,
                    int mask,
                    int bits );
```

Arguments:

db	A pointer to the database handle.
mask	A bitmask of the bits you want to set or unset.
bits	The bits you want to set. If a bit is in *mask* but not in *bits*, it's knocked down.

Library:

qdb

Description:

This function queries or modifies the database connection parameters. You can set or unset the QDB_CONN_BLOCK_FOREVER and QDB_CONN_STMT_ASYNC parameters (see the description of the *flags* argument passed to *qdb_connect()* for a description of these flags). You can't change the QDB_CONN_DFLT_SHARE flag. The function returns the value of the old flags, so they can be temporarily changed and restored.

Returns:

>-1	The value of the old flags.
-1	An error occurred.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_connect()

Synopsis:

```
#include <qdb/qdb.h>

int qdb_printmsg( FILE *file,
                  qdb_result_t *result,
                  int format );
```

Arguments:

file A file handle where the function can send the results.

result The query result you want to print.

format The format you want the results in. Can be one of:

- QDB_FORMAT_SIMPLE — minimal formatting.
- QDB_FORMAT_HTML — HTML formatting, suitable for viewing in a web browser.
- QDB_FORMAT_COLUMN — column formatting, so that results appear under column names.

Library:

qdb

Description:

This function prints the results of an SQL **SELECT** query on a QDB database. You must specify a standard file stream, such as **stdout**.

Returns:

>0 Success.

-1 An error occurred (*errno* is set).

Errors:

EINVAL Invalid format specified

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_mprintf()

qdb_query()

Perform a database query

Synopsis:

```
#include <qdb/qdb.h>

qdb_result_t *qdb_query(
                        qdb_hdl_t *db,
                        int size_hint,
                        const char *fmt, ... );
```

Arguments:

db	A pointer to the database handle.
size_hint	An estimate (in bytes) of how much memory to initially allocate to receive the database result. Specifying a value of 0 will use a default initial setting. If you know the rough order of magnitude of the result in advance (either very small or very large), then you can improve performance by specifying that value in the *size_hint*. In all cases, the full result will be received.
fmt	A string that controls the format of the output, as described in *qdb_statement()*.

Library:

qdb

Description:

This convenience function provides a single-interface alternative to calling *qdb_statement()* and *qdb_getresult()*, and offers a potential performance improvement if the statement and result communication can be made with a single context switch.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_statement(), *qdb_getresult()*

qdb_rowchanges()

Return the number of rows affected by a statement

Synopsis:

```
#include <qdb/qdb.h>

uint64_t qdb_rowchanges( qdb_hdt_t *hdl
                         qdb_result_t *result );
```

Arguments:

hdl A pointer to the database handle. You can pass as NULL if you provide *result*, and QDB_OPTION_ROW_CHANGES option has been set by *qdb_setoption()* (it's on by default).

result A pointer to a result set you want to query. If you pass NULL, the function queries the result from the last executed *qdb_statement()* on *hdl*.

Library:

qdb

Description:

This function returns the number of rows that were affected in a statement. It first looks in *result* (if the QDB_OPTION_ROW_CHANGES option has been set by *qdb_setoption()*), returning the number of rows for the statement that produced the result. If *result* is NULL, or QDB_OPTION_ROW_CHANGES is off, the function queries the database handle *hdl* and returns the information about the last executed statement.

If this function returns 0, check *errno* to make sure that it is EOK, indicating that no row was affected (you should set *errno* to **0** before calling this function if you want to distinguish between an error and **0** rows). If *errno* is not EOK then there was an error with the request.

Returns:

> 0 The number of rows affected

0 An error occurred (*errno* is set), or 0 rows were affected.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

qdb_setoption(), qdb_statement()

qdb_rows()

Return the number of rows in a result

Synopsis:

```
#include <qdb/qdb.h>

int qdb_rows(qdb_result_t *res);
```

Arguments:

res A pointer a result structure to check.

Library:

qdb

Description:

This function returns the number of rows in the result. If your query matched no rows in the database, then this function returns 0.

Returns:

>-1 The number of rows in the result set.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_rowchanges()

qdb_setbusytimeout()

Set the busy timeout delay for a database connection

Synopsis:

```
#include <qdb/qdb.h>

int qdb_setbusytimeout( qdb_hdt_t *hdl,
                        int timeout );
```

Arguments:

hdl A pointer to the database handle to set the timeout for.

timeout The timeout delay, in ms. This value may also be:

- QDB_TIMEOUT_NONBLOCK — the equivalent of a 0 timeout. This means that calls to *qdb_statement()* return immediately with failure if the database file is locked.
- QDB_TIMEOUT_BLOCK — the equivalent of an infinite timeout period. Calls to *qdb_statement()* will wait forever, or until the database is unlocked and the call succeeds.

Library:

qdb

Description:

This function sets the busy timeout delay for the database connection specified by *hdl*. The initial value is specified on the **qdb** commandline with the **-t** option, with a default of 5000 ms. Specifying a value of 0 is the same as QDB_TIMEOUT_NONBLOCK.

The timeout is the amount of time that a client will attempt to access a database before it returns EBUSY. If two clients attempt to write to the database, for example, the database is locked while the first client is writing, and the second client's attempt will fail if the busy timeout period expires.

The QDB_CONN_NONBLOCKING flag bit is affected by the timeout value. If you set or toggle QDB_CONN_NONBLOCKING, the busy timeout value itself is set to 0 or back to the **-t** default. Similarly, if you set the timeout to be QDB_TIMEOUT_NONBLOCK, the QDB_CONN_NONBLOCKING bit is set.

(The QDB_CONN_NONBLOCKING flag bit is set with *qdb_connect()* and toggled with *qdb_parameters()*.)

Returns:

≥`0`	Success. The previous busy timeout setting is returned.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_connect(), *qdb_parameters()*, *qdb_statement()*

qdb_setoption()

Set an option for a database connection

Synopsis:

```
#include <qdb/qdb.h>

int qdb_setoption( qdb_hdt_t *hdl,
                   int option,
                   int value );
```

Arguments:

hdl A pointer to the database handle to set the option for.

option The option to set; can be one of:

- QDB_OPTION_LAST_INSERT_ROWID — automatically put the last inserted ROWID into any result you fetch. If this option isn't set, that data isn't included in the result structure, and calling *qdb_last_insert_rowid()* will query the database connection for this information instead.

  By default, this option is on.
- QDB_OPTION_ROW_CHANGES — put the number of rows affected by a statement into any result you fetch. If this option isn't set, that data isn't included in the result structure, and calling *qdb_rowchanges()* will query the database connection for this information instead.

  By default, this option is on.
- QDB_OPTION_COLUMN_NAMES — populate the column names into the **`qdb_result_t`** that is returned from *qdb_getresult()*. If this option isn't set, that data isn't provided, and calling *qdb_column_index()* won't work.

  By default, this option is on.

value The value to set the option to: either **`0`** (**"**off**"**) or **`1`** (**"**on**"**).

Library:

`qdb`

Description:

This function sets options for the database connection *hdl*. By default, all of these options are on.

Returns:

≥**`0`** Success. The previous value for *option* is returned.

`-1` The option specified isn't supported (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_column_index(), *qdb_getresult()*, *qdb_last_insert_rowid()*, *qdb_rowchanges()*

Print formatted output to a string, up to a given maximum number of characters

Synopsis:

```
#include <qdb/qdb.h>

char * qdb_snprintf( int n,
                     char *buf,
                     const char *format, ...);
```

Arguments:

n	The maximum number of characters to store in the buffer, including a terminating null character. The function will always write a zero-terminator if *n* is positive.
buf	A pointer to the buffer where you want the function to store the formatted string.
format	A pointer to a formatting string to process. The formatting string determines what additional arguments you need to provide. For more information, see *printf()* in the Neutrino *Library Reference*.

Library:

qdb

Description:

This function is a variant of the *snprintf()* from the standard C library. However, it is different from *snprintf()* in these ways:

- *qdb_snprintf()* returns a pointer to the buffer rather than the number of characters written
- the order of the *n* and *buf* parameters is reversed
- *qdb_snprintf()* always writes a zero-terminator if *n* is positive

For more information about additional formatting options, see *qdb_mprintf()*.

CAUTION: You shouldn't use the return value of this function. In future versions, it may be changed to return the number of characters written rather than a pointer to the buffer.

Returns:

A pointer to *buf*	Success.
NULL	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_mprintf(), printf() in the Neutrino *Library Reference*

qdb_statement()

Execute an SQL statement

Synopsis:

```
#include <qdb/qdb.h>

int qdb_statement( qdb_hdt_t *hdl,
                   const char *format, ]
                   ... );
```

Arguments:

hdl A pointer to the database handle.

format A string that controls the format of the output, as described below. The formatting string determines what additional arguments you need to provide. The string that results from the combination of *format* and the additional arguments is executed as a statement on the database referred to by *hdl*.

Library:

qdb

Description:

This function executes against the database all statements in the string generated by the combination of *format* and any additional arguments. A statement must be completed with a semicolon. The string may contain multiple statements as long as they are separated by semicolons. There's no arbitrary restriction on the length of the command string.

The format string and additional arguments work in the same way as the arguments for *printf()*, and all the same conversion specifiers apply. There are additional conversion type specifiers, **%q** and **%Q**, which in general should be used instead of **%s** for inserting text into a literal string. The **%q** type specifier properly escapes special characters for SQL. For more information, see *qdb_mprintf()*.

If you are passing in multiple statements, the function returns the number of affected rows only for the last statement.

By default, the SQL statement is executed on the database before *qdb_statement()* returns. However, if the connection is in asynchronous mode, this function can return before the statement is executed, and it may not report errors. In this case, you need to call *qdb_getresult()* to retrieve any errors generated by the statement. For more information, see "Using asynchronous mode" in *qdb_connect()*.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_mprintf(), *qdb_vmprintf()*, *printf()* in the Neutrino *Library Reference*

qdb_stmt_exec()

Execute a precompiled statement

Synopsis:

```
#include <qdb/qdb.h>

int qdb_stmt_exec( qdb_hdl_t *hdl,
                   int stmtid,
                   qdb_binding_t *bindings,
                   uint8_t binding_count );
```

Arguments:

hdl	A pointer to the database handle.
stmtid	The ID of a pre-compiled statement, returned by *qdb_stmt_init()*.
bindings	An array of **qdb_binding_t** structures filled in with pointers to data that will be bound in to the variable parameters in the pre-compiled statement. See below.
binding_count	The number of items in *bindings*.

Library:

qdb

Description:

This function executes a precompiled statement that was previously prepared with *qdb_stmt_init()*.

The qdb_binding_t structure

The **qdb_binding_t** structure has at least these members:

int *index*	The index of the variable parameter in the precompiled statement that this data should be bound to. The placeholder is in the form of **?***n*, where *n* is a number between 1 and 999.
int *type*	The type of the data. Can be one of: QDB_NULL, QDB_BLOB, QDB_TEXT, QDB_INTEGER, or QDB_REAL.
int *len*	The length of the *data* argument. This number should exclude **'\0'** for QDB_TEXT, should be **sizeof(uint64_t)** for QDB_INTEGER and sizeof(double) for QDB_REAL.
void **data*	A pointer to the data to be bound in.

You can initialize an instance of **qdb_binding_t** using one of the convenience macros below. In the macro prototypes, *bind* is the address of the **qdb_binding_t** structure, *i* is the *index* member, *t* is the *type* member, *l* is the *len* member, and *d* is the *data*:

QDB_SETBIND(*bind*, *i*, *t*, *l*, *d*)

Bind in any type of data.

QDB_SETBIND_INT(*bind*, *i*, *d*)

Bind in an integer.

*QDB_SETBIND_NULL**bind*, *i*)

Bind in NULL.

QDB_SETBIND_TEXT(*bind*, *i*, *d*)

Bind in text.

There is a limit to the amount of data that can be sent to a database when using *qdb_stmt_exec()*. This limit is the *lesser* of the following values:

- the limits set by the database
- $x = 2^{31}$ - (*binding_count* + 1) × 12, where *x* is the data limit, in bytes

Returns:

>0 Success.

-1 An error occurred (*errno* is set).

Examples:

See *qdb_stmt_init()*.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_stmt_free(), *qdb_stmt_init()*

qdb_stmt_free()

Free a precompiled statement

Synopsis:

```
#include <qdb/qdb.h>

int qdb_stmt_free( qdb_hdl_t *hdl,
                   int stmtid )
```

Arguments:

hdl A pointer to the database handle.

stmtid The ID of a pre-compiled statement to free, returned by *qdb_stmt_init()*.

Library:

qdb

Description:

This function frees a statement previously compiled by *qdb_stmt_init()*. It's not strictly necessary to call this function, as all precompiled statements are freed when you call *qdb_disconnect()*.

Returns:

>0 Success.

-1 An error occurred (*errno* is set).

Examples:

See *qdb_stmt_init()*.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_stmt_exec(), qdb_stmt_init()

Synopsis:

```
#include <qdb/qdb.h>

int qdb_stmt_init( qdb_hdl_t *hdl,
    const char *sql,
    uint32_t len)
```

Arguments:

hdl A pointer to the database handle.

sql An SQL statement. This statement may contain variable parameters of the form **?***n*, where *n* is a number between 1 and 999. These placeholders can be filled in with data on a subsequent call to *qdb_stmt_exec()*. Parameters that aren't filled in are interpreted as NULL. For more informatin, see "Parameters" in the description of expressions, in the appendix: SQL Expressions Reference.

len The length of *sql*.

Library:

qdb

Description:

This function initializes a prepared (precompiled) SQL statement. A prepared statement is compiled once, and can be executed multiple times (with calls to *qdb_stmt_exec()*). This function returns a statement ID for the precompiled statement, which you need to pass in to *qdb_stmt_exec()*.

QDB executes precompiled statements faster than uncompiled statements, so this approach can optimize your application's performance when executing frequently used statements.

You can free precompiled statements using *qdb_stmt_free()*, although all precompiled statements are freed when you call *qdb_disconnect()*.

Returns:

>0 Success. The returned value is the prepared statement ID, which you pass to *qdb_stmt_exec()* and *qdb_stmt_free()*.

-1 An error occurred (*errno* is set).

Examples:

The following code snippet shows how you could compile and execute a simple statement:

```
int           stmtid;
qdb_binding_t qbind[2];
uint64_t      msid, limit;

const char   *sql = "SELECT fid FROM library WHERE msid=?1 LIMIT ?2;";

stmtid = qdb_stmt_init(db, sql, strlen(sql)+1);

if (stmtid == -1) {
     // Could not compile
     return -1;
}

msid = 1;
limit = 10;
QDB_SETBIND_INT(&qbind[0], 1, msid);
QDB_SETBIND_INT(&qbind[1], 2, limit);

if (qdb_stmt_exec(db, stmtid, qbind, 2) == -1) {
     // Could not execute
     return -1;
}

qdb_stmt_free(db, stmtid);
```

Note the **+1** added to the length of the string returned by *strlen()*; this sends QDB the final NULL character required of a valid string.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_stmt_exec(), *qdb_stmt_free()*

Synopsis:

```
#include <qdb/qdb.h>

int qdb_vacuum ( qdb_hdt_t *hdl,
                 int        scope );
```

Arguments:

hdl A pointer to the database handle.

scope Describes the scope of the operation. See the description of the *scope* argument to *qdb_backup()* for more information.

Library:

qdb

Description:

This function starts a vacuum operation on the specified database, as well as any auto-attached databases (databases listed in the specified datases's **.aa** file). This is an alternative to using the **VACUUM** command for each database.

You can call *qdb_getdbsize()* to determine whether a database should be vacuumed.

If the auto-vacuum mode is set (see the PRAGMA SQL command for details), databases are vacuumed whenever free space is created. By default, auto-vacuum mode is off.

Returns:

>0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

VACUUM

Synopsis:

```
#include <qdb/qdb.h>

char * qdb_vmprintf( const char* fmt,
                     va_list arg );
```

Arguments:

fmt A pointer to a formatting string to process. The formatting string determines what additional arguments you need to provide. For more information, see *printf()* in the Neutrino *Library Reference*.

arg A variable-argument list of the additional arguments, which you must have initialized with the *va_start()* macro.

Library:

qdb

Description:

This function is a variant of the *vsprintf()* from the standard C library. For more information about additional formatting options, see *qdb_mprintf()*.

Returns:

An formatted string, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

qdb_mprintf(), *qdb_snprintf()*, *printf()*, *va_start()* and *vsprint()* in the Neutrino *Library Reference*

Appendix B

QDB SQL Reference

QDB supports a sub-set of ANSI SQL-92. This appendix provides information about supported capabilities, organized into the following topis:

- General information
- Information about Statements

General

General information is organized into the following topics:

- Row ID and Autoincrement
- Comments
- Expressions
- Keywords

Statements

The statements described in this appendix are:

- **ALTER TABLE**
- **ANALYZE**
- **ATTACH DATABASE**
- **CREATE INDEX**
- **CREATE TABLE**
- **CREATE TRIGGER**
- **CREATE VIEW**
- **DELETE**
- **DETACH DATABASE**
- **DROP INDEX**
- **DROP TABLE**
- **DROP TRIGGER**
- **DROP VIEW**
- **EXPLAIN**
- **INSERT**
- **ON CONFLICT**
- **PRAGMA**

- **REINDEX**
- **REPLACE**
- **SELECT**
- **TRANSACTION**
- **UPDATE**
- **VACUUM**

Description:

In QDB, every row of every table has a 64-bit signed integer row ID. The row ID for each row is unique among all rows in the same table.

You can access the row ID of an QDB table using one the special column names **ROWID**, **_ROWID_**, or **OID**. However, if you declare an ordinary table column to use one of those special names, then the use of that name refers to the declared column, not to the internal row ID.

If a table contains a column of type INTEGER PRIMARY KEY, then that column becomes an alias for the row ID. You can then access the row ID using any of four different names: the original three names described above, or the name given to the INTEGER PRIMARY KEY column. All these names are aliases for one another and work equally well in any context.

When you insert a new row into a QDB table, you can either specify the row ID as part of the **INSERT** statement, or the database engine can assign it automatically. To specify a row ID manually, just include it in the list of values to be inserted. For example:

```
CREATE TABLE test1(a INT, b TEXT);
INSERT INTO test1(rowid, a, b) VALUES(123, 5, 'hello');
```

If no row ID is specified on the insert, an appropriate row ID is created automatically. By default, QDB gives the newly created row a row ID that is one larger than the largest row ID in the table prior to the insert. If the table is initially empty, then QDB uses a row ID of 1. If the largest row ID is equal to the largest possible integer (9223372036854775807), then the database engine starts picking candidate IDs at random until it finds one that isn't previously used.

The normal row ID selection algorithm described above will generate monotonically increasing unique row IDs as long as you never use the maximum row ID value and you never delete the entry in the table with the largest row ID. If you ever delete rows or if you ever create a row with the maximum possible row ID, then row IDs from previously deleted rows might be reused when creating new rows, and newly created row IDs might not be in strictly ascending order.

The AUTOINCREMENT Keyword

If a column has the type INTEGER PRIMARY KEY AUTOINCREMENT then a slightly different row ID selection algorithm is used. The row ID chosen for the new row is one larger than the largest row ID that has ever before existed in that same table. If the table has never before contained any data, then the database engine uses a row ID of 1. If the table has previously held a row with the largest possible row ID, then new INSERTs are not allowed, and any attempt to insert a new row fails with a QDB_FULL error.

QDB keeps track of the largest row ID that a table has ever held using the special QDB_SEQUENCE table. The QDB_SEQUENCE table is created and initialized automatically whenever a normal table that contains an AUTOINCREMENT column is

created. The content of the QDB_SEQUENCE table can be modified using ordinary **UPDATE**, **INSERT**, and **DELETE** statements. But make sure you know what you are doing before you undertake such changes — making modifications to this table will likely perturb the AUTOINCREMENT key generation algorithm.

The behavior implemented by the **AUTOINCREMENT** keyword is subtly different from the default behavior. With **AUTOINCREMENT**, rows with automatically selected row IDs are guaranteed to have row IDs that have never been used before by the same table in the same database. And the automatically generated row IDs are guaranteed to be monotonically increasing. These are important properties in certain applications. But if your application does not require this behavior, you should probably stay with the default behavior, since the use of **AUTOINCREMENT** requires QDB to perform additional work as each row is inserted and thus causes **INSERT**s to run a little more slowly.

Comment

SQL comment

Synopsis:

```
-- single-line

/* multiple-lines [*/]
```

Description:

Comments aren't SQL commands, but can occur in SQL queries. They are treated as whitespace by the parser. They can begin anywhere whitespace can be found, including inside expressions that span multiple lines.

SQL comments extend only to the end of the current line.

C comments can span any number of lines. If there is no terminating delimiter, they extend to the end of the input. This is not treated as an error. A new SQL statement can begin on a line after a multiline comment ends. C comments can be embedded anywhere whitespace can occur, including inside expressions, and in the middle of other SQL statements. C comments do not nest. SQL comments inside a C comment will be ignored.

Synopsis:

expr binary-op expr **|**
expr **[NOT] { LIKE | GLOB }** *expr* **[ESCAPE** *expr***] |**
unary-op expr **|**
(*expr* **) |**
[[*database-name* **.] [***table-name* **.]** *column-name* **|**
literal-value **|**
parameter **|**
function-name **(** *expr-list* **| *) |**
expr **ISNULL |**
expr **NOTNULL |**
expr **[NOT] BETWEEN** *expr* **AND** *expr* **|**
expr **[NOT] IN (** *value-list* **) |**
expr **[NOT] IN (** *select-statement* **) |**
expr **[NOT] IN [***database-name* **.]** *table-name* **|**
[EXISTS] (*select-statement* **) |**
CASE [*expr***] (WHEN** *expr* **THEN** *expr* **)+ [ELSE** *expr***] END |**
CAST (*expr* **AS** *type* **)**
expr **COLLATE** *collation-name*

Description:

SQL expressions are subcomponents of most other commands. QDB understands the following binary operators, in order from highest to lowest precedence:

```
||
*    /    %
+    -
<<   >>   &    |
<    <=   >    >=
=    ==   !=   <>   IN
AND
OR
```

The supported unary prefix operators are:

```
-    +    !    ~    NOT
```

The **COLLATE** operator can be thought of as a unary postfix operator. The **COLLATE** operator has the highest precedence. It always binds more tightly than any prefix unary operator or any binary operator.

The unary operator [Operator +] is a no-op. It can be applied to strings, numbers, or blobs and it always gives as its result the value of the operand.

Note that there are two variations of the equals and not equals operators. Equals can be either = or ==. The non-equals operator can be either != or **<>**. The **||** operator is "concatenate" — it joins together the two strings of its operands. The operator % outputs the remainder of its left operand modulo its right operand.

The result of any binary operator is a numeric value, except for the **||** concatenation operator, which gives a string result.

Literal values

A literal value is an integer number or a floating point number. Scientific notation is supported. The "." character is always used as the decimal point even if the locale setting specifies "," for this role — the use of "," for the decimal point would result in syntactic ambiguity. A string constant is formed by enclosing the string in single quotation marks ('). A single quotation mark within the string can be encoded by putting two single quotes in a row, as in Pascal. C-style escapes using the backslash character are not supported because they are not standard SQL. BLOB literals are string literals containing hexadecimal data and preceded by a single "x" or "X" character. For example:

```
X'53514697465'
```

A literal value can also be the token NULL.

Parameters

A parameter specifies a placeholder in the expression for a literal value that is filled in at runtime using *qdb_stmt_exec()*. Parameters can take several forms:

?*NNN*	A question mark followed by a number, *NNN*, holds a spot for the NNN-th parameter. *NNN* must be between 1 and 999.
?	A question mark that is not followed by a number holds a spot for the next unused parameter.
:*AAAA*	A colon followed by an identifier name holds a spot for a named parameter with the name *AAAA*. Named parameters are also numbered. The number assigned is the next unused number. To avoid confusion, it is best to avoid mixing named and numbered parameters.
@AAAA	An "at" sign works exactly like a colon.
$*AAAA*	A dollar-sign followed by an identifier name also holds a spot for a named parameter with the name *AAAA*. The identifier name in this case can include one or more occurrences of **"**::**"** and a suffix enclosed in "(...)" containing any text at all. This syntax is the form of a variable name in the Tcl programming language.

Parameters that are not assigned values using *qdb_stmt_exec()* are treated as NULL.

LIKE

The **LIKE** operator does a pattern-matching comparison. The operand to the right contains the pattern; the left-hand operand contains the string to match against the pattern.

A percent symbol(%) in the pattern matches any sequence of zero or more characters in the string. An underscore (_) in the pattern matches any single character in the string. Any other character matches itself or its lower/upper case equivalent (i.e. case-insensitive matching). (A bug: QDB understands only upper/lower case for 7-bit Latin characters. Hence the **LIKE** operator is case sensitive for 8-bit iso8859 characters or UTF-8 characters. For example, the expression **'a' LIKE 'A'** is TRUE but **'æ' LIKE 'Æ'** is FALSE.).

If the optional **ESCAPE** clause is present, then the expression following the **ESCAPE** keyword must evaluate to a string consisting of a single character. This character may be used in the **LIKE** pattern to include literal percent or underscore characters. The escape character followed by a percent symbol, underscore or itself matches a literal percent symbol, underscore or escape character in the string, respectively. The infix **LIKE** operator is implemented by calling the user function *like*(*X*,*Y*).

GLOB

The **GLOB** operator is similar to **LIKE**, but uses the UNIX file-globbing syntax for its wildcards. Also, **GLOB** is case sensitive, unlike **LIKE**. Both **GLOB** and **LIKE** may be preceded by the **NOT** keyword to invert the sense of the test. The infix **GLOB** operator is implemented by calling the user function *glob(X,Y)* and can be modified by overriding that function.

Column Names

A column name can be any of the names defined in the **CREATE TABLE** statement or one of the following special identifiers: **ROWID**, **OID**, or **_ROWID_**. These special identifiers all describe the unique random integer key (the *row key*) associated with every row of every table. The special identifiers only refer to the row key if the **CREATE TABLE** statement does not define a real column with the same name. Row keys act like read-only columns. A row key can be used anywhere a regular column can be used, except that you cannot change the value of a row key in an **UPDATE** or **INSERT** statement. **SELECT * ...** does not return the row key.

SELECT statements

SELECT statements can appear in expressions as either the right-hand operand of the **IN** operator, as a scalar quantity, or as the operand of an **EXISTS** operator. As a scalar quantity or the operand of an **IN** operator, the **SELECT** should have only a single column in its result. Compound **SELECT**s (connected with keywords like **UNION** or **EXCEPT**) are allowed. With the **EXISTS** operator, the columns in the result set of the **SELECT** are ignored and the expression returns TRUE if one or more rows exist and FALSE if the result set is empty. If no terms in the **SELECT** expression refer to value in the containing query, then the expression is evaluated once prior to any other processing and the result is reused as necessary. If the **SELECT** expression does contain variables from the outer query, then the **SELECT** is reevaluated every time it is needed.

When a **SELECT** is the right operand of the **IN** operator, the **IN** operator returns TRUE if the result of the left operand is any of the values generated by the select. The **IN** operator may be preceded by the **NOT** keyword to invert the sense of the test.

When a **SELECT** appears within an expression but is not the right operand of an **IN** operator, then the first row of the result of the **SELECT** becomes the value used in the expression. If the **SELECT** yields more than one result row, all rows after the first are ignored. If the **SELECT** yields no rows, then the value of the **SELECT** is NULL.

CAST

A **CAST** expression changes the datatype of the *expr* into the type specified by *type*, where *type* can be any nonempty type name that is valid for the type in a column definition of a **CREATE TABLE** statement.

Functions

Both simple and aggregate functions are supported. A simple function can be used in any expression. Simple functions return a result immediately based on their inputs. Aggregate functions may only be used in a **SELECT** statement. Aggregate functions compute their result across all rows of the result set.

Core Functions

The functions shown below are available by default.

abs(*X*)	Return the absolute value of argument *X*.
coalesce(*X*,*Y*,...)	Return a copy of the first non-NULL argument. If all arguments are NULL, then NULL is returned. There must be at least 2 arguments.
glob(*X*,*Y*)	This function is used to implement the **X GLOB Y** syntax of QDB.
hex(*X*)	The argument is interpreted as a BLOB. The result is a hexadecimal rendering of the content of that blob.
ifnull(*X*,*Y*)	Return a copy of the first non-NULL argument. If both arguments are NULL, then NULL is returned. This behaves the same as *coalesce()* above.
last_insert_rowid()	Return the row ID of the last row inserted from this connection to the database. This is the same value that would be returned from the *qdb_last_insert_rowid()*.
length(*X*)	Return the string length of *X* in characters.
like(*X*,*Y* [,*Z*])	This function is used to implement the **X LIKE Y [ESCAPE Z]"** syntax of SQL. If the optional **ESCAPE** clause is present, then the user-function is invoked with three arguments. Otherwise, it is invoked with two arguments only.
lower(*X*)	Return a copy of string *X* will all characters converted to lower case.

ltrim(*X* [,*Y*])	Return a string formed by removing any and all characters that appear in *Y* from the left side of *X*. If the *Y* argument is omitted, spaces are removed.
max(*X*,*Y*,...)	Return the argument with the maximum value. Arguments may be strings in addition to numbers. The maximum value is determined by the usual sort order. Note that *max()* is a simple function when it has 2 or more arguments but converts to an aggregate function if given only a single argument.
min(*X*,*Y*,...)	Return the argument with the minimum value. Arguments may be strings in addition to numbers. The minimum value is determined by the usual sort order. Note that *min()* is a simple function when it has 2 or more arguments but converts to an aggregate function if given only a single argument.
nullif(*X*,*Y*)	Return the first argument if the arguments are different, otherwise return NULL.
quote(*X*)	This routine returns a string which is the value of its argument suitable for inclusion into another SQL statement. Strings are surrounded by single-quotes with escapes on interior quotes as needed. BLOBs are encoded as hexadecimal literals. The current implementation of **`VACUUM`** uses this function. The function is also useful when writing triggers to implement undo/redo functionality.
random(*)	Return a random integer between **`-2147483648`** and **`+2147483647`**.
randomblob(*N*)	Return a *N*-byte blob containing pseudo-random bytes. *N* should be a postive integer.
replace(*X*,*Y*,*Z*)	Return a string formed by substituting string *Z* for every occurrance of string *Y* in string *X*. The **`BINARY`** collating sequence is used for comparisons.
round(*X*[, *Y*])	Round off the number *X* to *Y* digits to the right of the decimal point. If the *Y* argument is omitted, 0 is assumed.
rtrim(*X* [,*Y*])	Return a string formed by removing any and all characters that appear in *Y* from the right side of *X*. If the *Y* argument is omitted, spaces are removed.
soundex(*X*)	Compute the soundex encoding of the string *X*. The string **`"?000"`** is returned if the argument is NULL.
sqlite_version()	Return the version string for the SQLite library that is running. Example: **"**2.8.0**"**

substr(*X*,*Y*,*Z*) Return a substring of input string *X* that begins with the *Y*-th character and which is *Z* characters long. The left-most character of *X* is number 1. If *Y* is negative the first character of the substring is found by counting from the right rather than the left. QDB is configured to support UTF-8, so characters indices refer to actual UTF-8 characters, not bytes.

trim(*X* [,*Y*]) Return a string formed by removing any and all characters that appear in *Y* from both sides of *X*. If the *Y* argument is omitted, spaces are removed.

typeof(*X*) Return the type of the expression *X*. The possible return values are

- **`"null"`**
- **`"integer"`**
- **`"real"`**
- **`"text"`**
- **`"blob"`**

QDB's type handling is explained in the chapter Datatypes in QDB.

upper(X) Return a copy of input string *X* converted to all upper-case letters. The implementation of this function uses the C library routine **toupper()** which means it may not work correctly on UTF-8 strings.

Aggregate Functions

In any aggregate function that takes a single argument, that argument can be preceded by the keyword **`DISTINCT`**. In such cases, duplicate elements are filtered before being passed into the aggregate function. For example, the function **`count(distinct`** *X***`)`** will return the number of distinct values of column *X* instead of the total number of non-NULL values in column *X*.

avg(*X*) Return the average value of all non-NULL *X* within a group. String and BLOB values that don't look like numbers are interpreted as 0. The result of *avg()* is always a floating point value, even if all inputs are integers.

count(*X*) The first form returns the number of times that *X* is not NULL in a group. The second form (with no argument) returns the total number of rows in the group.

max(*X*) Return the maximum value of all values in the group. The usual sort order is used to determine the maximum.

min(*X*)

Return the minimum non-NULL value of all values in the group. The usual sort order is used to determine the minimum. NULL is returned only if all values in the group are NULL.

sum(*X*)
total(*X*)

Return the numeric sum of all non-NULL values in the group. If there are no non-NULL input rows or all values are NULL, then *sum()* returns NULL, and *total()* returns 0.0. NULL is not normally a helpful result for the sum of now rows, but the SQL standard requires it, and most other SQL database engines implement *sum()* that way, so QDB does it in the same way in order to be compatible. The *total()* function is provided as a convenient way to work around this design problem in the SQL language.

The result of *total()* is always a floating point value. The result of *sum()* is an integer value if all non-NULL inputs are integers. If any input to *sum()* is neither an integer or a NULL, then *sum()* returns a floating point value which might be an approximation to the true sum.

The *sum()* function throws an "integer overflow" exception if all inputs are integers or NULL and an integer overflow occurs at any point during the computation. The *total()* function never throws an exception.

Description:

The SQL standard specifies a huge number of keywords that you can *not* use as the names of tables, indexes, columns, databases, user-defined functions, collations, virtual table modules, or any other named object. The list of keywords is so long that few people can remember them all. For most SQL code, your safest bet is to never use any word in the English language as the name of a user-defined object.

If you want to use a keyword as a name, you need to quote it. There are three ways of quoting keywords in QDB:

`'keyword'` A keyword in single quotes is interpreted as a literal string if it occurs in a context where a string literal is allowed, otherwise it is understood as an identifier.

`"keyword"` A keyword in double-quotes is interpreted as an identifier if it matches a known identifier. Otherwise it is interpreted as a string literal.

`[keyword]` A keyword enclosed in square brackets is always understood as an identifier. This is not standard SQL. This quoting mechanism is used by MS Access and SQL Server and is included in QDB for compatibility.

Quoted keywords are unaesthetic. To help you avoid them, QDB allows many keywords to be used unquoted as the names of databases, tables, indices, triggers, views, columns, user-defined functions, collations, attached databases, and virtual function modules. In the list of keywords that follows, keywords that can be used as identifiers are shown in italics. Keywords that must be quoted in order to be used as identifiers are shown in bold.

QDB adds new keywords from time to time when it take on new features. So to prevent your code from being broken by future enhancements, you should normally quote any identifier that is a word in English, even if you do not have to.

The following are the keywords currently recognized by QDB:

ABORT	**AUTOINCREMENT**	**COMMIT**
ADD	*BEFORE*	*CONFLICT*
AFTER	*BEGIN*	**CONSTRAINT**
ALL	**BETWEEN**	**CREATE**
ALTER	**BY**	**CROSS**
ANALYZE	*CASCADE*	*CURRENT_DATE*
AND	**CASE**	*CURRENT_TIME*
AS	*CAST*	*CURRENT_TIMESTAMP*
ASC	**CHECK**	*DATABASE*
ATTACH	**COLLATE**	**DEFAULT**

DEFERRABLE	**INNER**	**REFERENCES**
DEFERRED	**INSERT**	*REINDEX*
DELETE	*INSTEAD*	*RENAME*
DESC	**INTERSECT**	*REPLACE*
DETACH	**INTO**	*RESTRICT*
DISTINCT	**IS**	**RIGHT**
DROP	**ISNULL**	**ROLLBACK**
EACH	**JOIN**	*ROW*
ELSE	*KEY*	**SELECT**
END	**LEFT**	**SET**
ESCAPE	*LIKE*	**TABLE**
EXCEPT	**LIMIT**	*TEMP*
EXCLUSIVE	*MATCH*	*TEMPORARY*
EXPLAIN	**NATURAL**	**THEN**
FAIL	**NOT**	**TO**
FOR	**NOTNULL**	**TRANSACTION**
FOREIGN	**NULL**	*TRIGGER*
FROM	*OF*	**UNION**
FULL	*OFFSET*	**UNIQUE**
GLOB	**ON**	**UPDATE**
GROUP	**OR**	**USING**
HAVING	**ORDER**	*VACUUM*
IF	**OUTER**	**VALUES**
IGNORE	*PLAN*	*VIEW*
IMMEDIATE	*PRAGMA*	*VIRTUAL*
IN	**PRIMARY**	**WHEN**
INDEX	*QUERY*	**WHERE**
INITIALLY	*RAISE*	

Special names

The following words are *not* keywords in QDB, but are used as names of system objects. They can be used as identifiers for a different type of object.

- **`_ROWID_`**
- **`MAIN`**
- **`OID`**
- **`ROWID`**
- **`SQLITE_MASTER`**
- **`SQLITE_SEQUENCE`**
- **`SQLITE_TEMP_MASTER`**
- **`TEMP`**

ALTER TABLE

Rename or add a new column to an existing table

Synopsis:

```
ALTER TABLE [database-name .] table-name {RENAME TO new-table-name} |
  {ADD [COLUMN] column-def}
```

Description:

QDB's version of the **ALTER TABLE** command lets you add a new column to or rename an existing table. It isn't possible to remove a column from a table.

The **RENAME TO** syntax is used to rename the table identified by [*database-name*.]*table-name* to *new-table-name*. This command cannot be used to move a table between attached databases, only to rename a table within the same database.

If the table being renamed has triggers or indexes, then these remain attached to the table after it has been renamed. However, if there are any view definitions or statements executed by triggers that refer to the table being renamed, these are not automatically modified to use the new table name. If this is required, the triggers or view definitions must be dropped and recreated to use the new table name by hand.

The **ADD [COLUMN]** syntax is used to add a new column to an existing table. The new column is always appended to the end of the list of existing columns. The *column-def* may take any of the forms permissible in a **CREATE TABLE** statement, with the following restrictions:

- The column may not have a **PRIMARY KEY** or **UNIQUE** constraint.
- The column may not have a default value of CURRENT_TIME, CURRENT_DATE or CURRENT_TIMESTAMP.
- If a NOT NULL constraint is specified, then the column must have a default value other than NULL.

The execution time of the **ALTER TABLE** command is independent of the amount of data in the table. The **ALTER TABLE** command runs as quickly on a table with 10 million rows as it does on a table with one row.

After **ADD COLUMN** has been run on a database, that database will not be readable by QDB until the database is **VACUUM**ed.

ANALYZE

Analyze indexes to optimize queries

Synopsis:

ANALYZE [*database-name* **.] [***table-name***]**

Description:

The **ANALYZE** command gathers statistics about indexes and stores them in a special tables in the database where the query optimizer can use them to help make better index choices. If no arguments are given, all indexes in all attached databases are analyzed. If a database name is given as the argument, all indexes in that database are analyzed. If the argument is a table name, then only indexes associated with that table are analyzed.

The *database-name* can be the name of any *attached* database. You don't have to supply the database name of non-attached database; if you do, use **main**.

The initial implementation stores all statistics in a single table named **sqlite_stat1**. Future enhancements may create additional tables with the same name pattern except with the 1 changed to a different digit. The **sqlite_stat1** table cannot be **DROP**ped, but all the content can be **DELETE**d, which has the same effect.

ATTACH DATABASE

Add a database to the current connection

Synopsis:

ATTACH [DATABASE] *database-filename* **AS** *database-name*

Description:

The **ATTACH DATABASE** statement adds another database file to the current database connection. If the filename contains punctuation characters, it must be placed inside quotation marks. The names **main** and **temp** refer to the main database and the database used for temporary tables. These cannot be detached. Attached databases are removed using the **DETACH DATABASE** statement.

You can read from and write to an attached database, and you can modify the schema of the attached database.

You cannot create a new table with the same name as a table in an attached database, but you can attach a database which contains tables whose names are duplicates of tables in the main database. It is also permissible to attach the same database file multiple times.

Tables in an attached database can be referred to using the syntax *database-name.table-name*. If an attached table doesn't have a duplicate table name in the main database, it doesn't require a database name prefix. When a database is attached, all of its tables which don't have duplicate names become the default table of that name. Any tables of that name attached afterwards require the table prefix. If the default table of a given name is detached, then the last table of that name attached becomes the new default.

Transactions involving multiple attached databases are atomic. There is a compile-time limit of 10 attached database files.

CREATE INDEX

Create an index

Synopsis:

CREATE [UNIQUE] INDEX [IF NOT EXISTS] [*database-name* **.]** *index-name*
ON *table-name* **(** *column-name* **[,** *column-name***]*)**
column-name **=**
name [COLLATE collation-name] [ASC | DESC]

Description:

The **CREATE INDEX** command consists of the keywords **CREATE INDEX** followed by the name of the new index, the keyword **ON**, the name of a previously created table that is to be indexed, and a parenthesized list of names of columns in the table that are used for the index key. Each column name can be followed by one of the **ASC** or **DESC** keywords to indicate sort order, but the sort order is ignored in the current implementation. Sorting is always done in ascending order.

The **COLLATE** clause following each column name defines a collating sequence used for text entires in that column. The default collating sequence is the collating sequence defined for that column in the **CREATE TABLE** statement. If no collating sequence is otherwise defined, the built-in **BINARY** collating sequence is used.

There are no arbitrary limits on the number of indexes that can be attached to a single table, nor on the number of columns in an index.

If the **UNIQUE** keyword appears between **CREATE** and **INDEX**, then duplicate index entries are not allowed. Any attempt to insert a duplicate entry will result in an error.

The exact text of each **CREATE INDEX** statement is stored in the **sqlite_master** or **sqlite_temp_master** table, depending on whether the table being indexed is temporary. Every time the database is opened, all **CREATE INDEX** statements are read from the **sqlite_master** table and used to regenerate QDB's internal representation of the index layout.

If the optional **IF NOT EXISTS** clause is present and another index with the same name aleady exists, then this command becomes a no-op.

Indexes are removed with the **DROP INDEX** command.

CREATE TABLE

Create a table

Synopsis:

```
CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS] [database-name.]
    table-name (
        column-def [, column-def]*
        [, constraint]*
)

CREATE [TEMP | TEMPORARY] TABLE [database-name.]
    table-name AS select-statement

column-def =
name [type] [[CONSTRAINT name] column-constraint]*

type =
typename |
typename ( number ) |
typename ( number , number )

column-constraint =
NOT NULL [ conflict-clause ] |
PRIMARY KEY [sort-order] [ conflict-clause ] [AUTOINCREMENT] |
UNIQUE [ conflict-clause ] |
CHECK ( expr ) |
DEFAULT value |
COLLATE collation-name

constraint =
PRIMARY KEY ( column-list ) [ conflict-clause ] |
UNIQUE ( column-list ) [ conflict-clause ] |
CHECK ( expr ) [ conflict-clause ]

conflict-clause =
ON CONFLICT conflict-algorithm
```

Description:

A **CREATE TABLE** statement is followed by the name of a new table and a parenthesized list of column definitions and constraints. The table name can be either an identifier or a string. Tables names that begin with **sqlite_** are reserved for use by the engine.

Each column definition is the name of the column followed by the datatype for that column, then one or more optional column constraints. The datatype for the column does not restrict what data may be put in that column. See the chapter Datatypes in QDB for additional information. The **UNIQUE** constraint causes an index to be created on the specified columns. This index must contain unique keys. The **COLLATE** clause specifies what text-collating function to use when comparing text entries for the column. The built-in **BINARY** collating function is used by default.

The **DEFAULT** constraint specifies a default value to use when doing an **INSERT**. The value may be NULL, a string constant or a number. The default value may also be one of the special case-independant keywords **CURRENT_TIME**, **CURRENT_DATE** or **CURRENT_TIMESTAMP**. If the value is NULL, a string constant or number, it is literally inserted into the column whenever an **INSERT** statement that does not specify a value for the column is executed.

If the value is **CURRENT_TIME**, **CURRENT_DATE** or **CURRENT_TIMESTAMP**, then the current UTC date and/or time is inserted into the columns. For **CURRENT_TIME**, the format is *HH:MM:SS*. For **CURRENT_DATE**, the format is *YYYY-MM-DD*. The format for **CURRENT_TIMESTAMP** is *YYYY-MM-DD HH:MM:SS*.

Specifying a **PRIMARY KEY** normally just creates a **UNIQUE** index on the corresponding columns. However, if primary key is on a single column that has datatype INTEGER, then that column is used internally as the actual key of the B-Tree for the table. This means that the column may only hold unique integer values. (Except for this one case, QDB ignores the datatype specification of columns and allows any kind of data to be put in a column regardless of its declared datatype.)

If a table does not have an INTEGER PRIMARY KEY column, then the B-Tree key will be a automatically generated integer. The B-Tree key for a row can always be accessed using one of the special names ROWID, OID, or _ROWID_. This is true regardless of whether or not there is an INTEGER PRIMARY KEY. An INTEGER PRIMARY KEY column can also include the keyword AUTOINCREMENT. The AUTOINCREMENT keyword modifies the way that B-Tree keys are automatically generated. Additional detail on automatic B-Tree key generation is available separately.

According to the SQL standard, PRIMARY KEY should imply NOT NULL. Unfortunately, due to a long-standing coding oversight, this is not the case in SQLite. SQLite allows NULL values in a PRIMARY KEY column. We could change SQLite to conform to the standard (and we might do so in the future), but by the time the oversight was discovered, SQLite was in such wide use that we feared breaking legacy code if we fixed the problem. So for now we have chosen to contain allowing NULLs in PRIMARY KEY columns. Developers should be aware, however, that we may change SQLite to conform to the SQL standard in future and should design new programs accordingly.

If the **TEMP** or **TEMPORARY** keyword is used, then the created table is visible only within that same database connection and is automatically deleted when the database connection is closed. Any indexes created on a temporary table are also temporary. Temporary tables and indexes are stored in a separate file distinct from the main database file.

If a *database-name* is specified, then the table is created in the named database. It is an error to specify both a *database-name* and the **TEMP** keyword, unless the *database-name* is **temp**. If no database name is specified, and the **TEMP** keyword is not present, the table is created in the main database.

The optional *conflict-clause* following each constraint allows the specification of an alternative default constraint conflict resolution algorithm for that constraint. The

default is **ABORT**. Different constraints within the same table may have different default conflict resolution algorithms. If a **COPY**, **INSERT**, or **UPDATE** command specifies a different conflict resolution algorithm, then that algorithm is used in place of the default algorithm specified in the **CREATE TABLE** statement. See the section **ON CONFLICT** for additional information.

CHECK constraints are now supported and enforced.

There are no arbitrary limits on the number of columns or on the number of constraints in a table. As well, there is no arbitrary limit on the amount of data in a row.

The **CREATE TABLE AS** form defines the table to be the result set of a query. The names of the table columns are the names of the columns in the result.

The exact text of each **CREATE TABLE** statement is stored in the **sqlite_master** table. Every time the database is opened, all **CREATE TABLE** statements are read from the **sqlite_master** table and used to regenerate QDB's internal representation of the table layout. If the original command was a **CREATE TABLE AS**, then an equivalent **CREATE TABLE** statement is synthesized and stored in **sqlite_master** in place of the original command. The text of **CREATE TEMPORARY TABLE** statements is stored in the **sqlite_temp_master** table.

If the optional **IF NOT EXISTS** clause is present and another table with the same name aleady exists, then this command becomes a no-op.

Tables are removed using the **DROP TABLE** statement.

CREATE TRIGGER

Create a trigger

Synopsis:

CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] *trigger-name*
[BEFORE | AFTER] *database-event* **ON [***database-name* **.]**
table-name trigger-action

CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] *trigger-name*
INSTEAD OF *database-event* **ON [***database-name* **.]**
view-name trigger-action

database-event **=**
DELETE |
INSERT |
UPDATE |
UPDATE OF *column-list*

trigger-action **=**
[FOR EACH ROW] [WHEN expression]
BEGIN
trigger-step **; [** *trigger-step* **;]***
END

trigger-step **=**
update-statement **|** *insert-statement* **|**
delete-statement **|** *select-statement*

Description:

The **CREATE TRIGGER** statement is used to add triggers to the database schema. Triggers are database operations (the *trigger-action*) that are automatically performed when a specified database event (the *database-event*) occurs.

A trigger may be specified to fire whenever a **DELETE**, **INSERT** or **UPDATE** of a particular database table occurs, or whenever an **UPDATE** of one or more specified columns of a table are updated.

At this time, QDB supports only **FOR EACH ROW** triggers, not **FOR EACH STATEMENT** triggers. Hence explicitly specifying **FOR EACH ROW** is optional. **FOR EACH ROW** implies that the SQL statements specified as *trigger-steps* may be executed (depending on the **WHEN** clause) for each database row being inserted, updated or deleted by the statement causing the trigger to fire.

Both the **WHEN** clause and the *trigger-steps* may access elements of the row being inserted, deleted or updated using references of the form **NEW.***column-name* and **OLD.***column-name*, where *column-name* is the name of a column from the table that the trigger is associated with. **OLD** and **NEW** references may only be used in triggers on *trigger-event*s for which they are relevant, as follows:

Command	Valid references
INSERT	**NEW** references are valid
UPDATE	**NEW** and **OLD** references are valid
DELETE	**OLD** references are valid

If a **WHEN** clause is supplied, the SQL statements specified as *trigger-steps* are executed only for rows for which the **WHEN** clause is true. If no **WHEN** clause is supplied, the SQL statements are executed for all rows.

The specified *trigger-time* determines when the *trigger-steps* will be executed relative to the insertion, modification or removal of the associated row.

An **ON CONFLICT** clause may be specified as part of an **UPDATE** or **INSERT** *trigger-step*. However if an **ON CONFLICT** clause is specified as part of the statement causing the trigger to fire, then this conflict handling policy is used instead.

Triggers are automatically dropped when the table that they are associated with is dropped.

Triggers may be created on views, as well as ordinary tables, by specifying **INSTEAD OF** in the **CREATE TRIGGER** statement. If one or more **ON INSERT**, **ON DELETE** or **ON UPDATE** triggers are defined on a view, then it is not an error to execute an **INSERT**, **DELETE** or **UPDATE** statement on the view, respectively. Thereafter, executing an **INSERT**, **DELETE** or **UPDATE** on the view causes the associated triggers to fire. The real tables underlying the view are not modified (except possibly explicitly, by a trigger program).

Example:

Assuming that customer records are stored in the *customers()* table, and that order records are stored in the *orders()* table, the following trigger ensures that all associated orders are redirected when a customer changes his or her address:

```
CREATE TRIGGER update_customer_address UPDATE OF address ON customers
  BEGIN
    UPDATE orders SET address = new.address
      WHERE customer_name = old.name;
  END;
```

With this trigger installed, executing the statement:

```
UPDATE customers SET address = '1 Main St.'
  WHERE name = 'Jack Jones';
```

causes the following to be automatically executed:

```
UPDATE orders SET address = '1 Main St.'
  WHERE customer_name = 'Jack Jones';
```

Note that triggers may behave oddly when created on tables with **INTEGER PRIMARY KEY** fields. If a **BEFORE** trigger program modifies the **INTEGER PRIMARY KEY** field

of a row that will be subsequently updated by the statement that causes the trigger to fire, then the update may not occur. The workaround is to declare the table with a PRIMARY KEY column instead of an **INTEGER PRIMARY KEY** column.

A special SQL function *RAISE()* may be used within a trigger-program, with the following syntax

RAISE (ABORT, *error-message* **) |**
RAISE (FAIL, *error-message* **) |**
RAISE (ROLLBACK, *error-message* **) |**
RAISE (IGNORE)

When one of the first three forms is called during trigger-program execution, the specified **ON CONFLICT** processing is performed (either **ABORT**, **FAIL** or **ROLLBACK**) and the current query terminates. An error code of **SQLITE_CONSTRAINT** is returned to the user, along with the specified error message.

When *RAISE*(**IGNORE**) is called, the remainder of the current trigger program, the statement that caused the trigger program to execute and any subsequent trigger programs that would of been executed are abandoned. No database changes are rolled back. If the statement that caused the trigger program to execute is itself part of a trigger program, then that trigger program resumes execution at the beginning of the next step.

Triggers are removed using the **DROP TRIGGER** statement.

CREATE VIEW

Create a view

Synopsis:

CREATE [TEMP | TEMPORARY] VIEW [IF NOT EXISTS] [*database-name***.]**
view-name **AS** *select-statement*

Description:

The **CREATE VIEW** command assigns a name to a prepackaged **SELECT** statement. Once the view is created, it can be used in the **FROM** clause of another **SELECT** in place of a table name.

The **TEMP** or **TEMPORARY** keyword means the view that is created is visible only to the process that opened the database and is automatically deleted when the database is closed.

If a *database-name* is specified, then the view is created in the named database. It is an error to specify both a *database-name* and the **TEMP** keyword, unless the *database-name* is **temp**. If no database name is specified, and the **TEMP** keyword is not present, the table is created in the main database.

You cannot **COPY**, **DELETE**, **INSERT** or **UPDATE** a view. Views are read-only in QDB. However, in many cases you can use a **TRIGGER** on the view to accomplish the same thing. Views are removed with the **DROP VIEW** command.

DELETE

Remove records from a table

Synopsis:

DELETE FROM [*database-name* **.]** *table-name* **[WHERE** *expr***]**

Description:

The **DELETE** command is used to remove records from a table. The command is followed by the name of the table from which records are to be removed.

Without a **WHERE** clause, all rows of the table are removed. If a **WHERE** clause is supplied, only those rows that match the expression are removed.

DETACH DATABASE

Detach from a database

Synopsis:

DETACH [DATABASE] *database-name*

Description:

This statement detaches an additional database connection previously attached using the **ATTACH DATABASE** statement. It is possible to have the same database file attached multiple times using different names, and detaching one connection to a file will leave the others intact.

This statement will fail if QDB is in the middle of a transaction.

DROP INDEX

Remove an index

Synopsis:

DROP INDEX [IF EXISTS] [*database-name* **.]** *index-name*

Description:

The **DROP INDEX** statement removes an index added with the **CREATE INDEX** statement. The index named is completely removed from the disk. The only way to recover the index is to reenter the appropriate **CREATE INDEX** command.

The **DROP INDEX** statement does not reduce the size of the database file in the default mode. Empty space in the database is retained for later **INSERT**s. To remove free space in the database, use the **VACUUM** command. If AUTOVACUUM mode is enabled for a database, then space will be freed automatically by **DROP INDEX**.

The optional **IF EXISTS** clause suppresses the error that would normally result if the index does not exist.

DROP TABLE

Remove a table

Synopsis:

DROP TABLE [IF EXISTS] [*database-name***.]** *table-name*

Description:

The **DROP TABLE** statement removes a table added with the **CREATE TABLE** statement. The name specified is the table name. It is completely removed from the database schema and the disk file. The table can not be recovered. All indexes associated with the table are also deleted.

The **DROP TABLE** statement does not reduce the size of the database file in the default mode. Empty space in the database is retained for later **INSERT**s. To remove free space in the database, use the **VACUUM** command. If AUTOVACUUM mode is enabled for a database, then space will be freed automatically by **DROP TABLE**.

The optional **IF EXISTS** clause suppresses the error that would normally result if the table does not exist.

DROP TRIGGER

Remove an index

Synopsis:

```
DROP TRIGGER [IF EXISTS] [database-name .] trigger-name
```

Description:

The **DROP TRIGGER** statement removes a trigger created by the **CREATE TRIGGER** statement. The trigger is deleted from the database schema.

Triggers are automatically dropped when the associated table is dropped.

Synopsis:

```
DROP VIEW [IF EXISTS] view-name
```

Description:

The **DROP VIEW** statement removes a view created by the **CREATE VIEW** statement. The name specified is the view name. It is removed from the database schema, but no actual data in the underlying base tables is modified.

EXPLAIN

Report VM instructions for a command

Synopsis:

EXPLAIN *sql-statement*

Description:

The **EXPLAIN** command modifier is a non-standard extension. The idea comes from a similar command found in PostgreSQL, but the operation is completely different.

If the **EXPLAIN** keyword appears before any other QDB SQL command then instead of actually executing the command, the QDB library will report back the sequence of virtual machine instructions it would have used to execute the command had the **EXPLAIN** keyword not been present. This is useful for performance analysis.

For additional information about virtual machine instructions see the documentation on QDB opcodes for the virtual machine.

INSERT

Insert data into a table

Synopsis:

```
INSERT [OR conflict-algorithm] INTO [database-name .]
  table-name [(column-list)] VALUES(value-list) |
INSERT [OR conflict-algorithm] INTO [database-name .]
  table-name [(column-list)] select-statement
```

Description:

The **INSERT** statement comes in two basic forms. The first form (with the **VALUES** keyword) creates a single new row in an existing table. If no *column-list* is specified, then the number of values must be the same as the number of columns in the table. If a *column-list* is specified, then the number of values must match the number of specified columns. Columns of the table that do not appear in the column list are filled with the default value, or with NULL if no default value is specified.

The second form of the **INSERT** statement takes its data from a **SELECT** statement. The number of columns in the result of the **SELECT** must exactly match the number of columns in the table if no column list is specified, or it must match the number of columns named in the column list. A new entry is made in the table for every row of the **SELECT** result. The **SELECT** may be simple or compound. If the **SELECT** statement has an **ORDER BY** clause, the **ORDER BY** is ignored.

The optional *conflict-clause* allows the specification of an alternative constraint-conflict resolution algorithm to use during this one command. See **ON CONFLICT** for additional information. For compatibility with MySQL, the parser allows the use of the single keyword **REPLACE** as an alias for **INSERT OR REPLACE**.

ON CONFLICT

Deal with a conflict

Synopsis:

```
ON CONFLICT { ROLLBACK | ABORT | FAIL | IGNORE | REPLACE }
```

Description:

The **ON CONFLICT** clause is not a separate SQL command. It is a non-standard clause that can appear in many other SQL commands. It is given its own section in this document because it is not part of standard SQL and therefore might not be familiar.

The syntax for the **ON CONFLICT** clause is as shown above for the **CREATE TABLE** command. For the **INSERT** and **UPDATE** commands, the keywords **ON CONFLICT** are replaced by **OR**, to make the syntax seem more natural. For example, instead of **INSERT ON CONFLICT IGNORE** we have **INSERT OR IGNORE**. The keywords change but but the meaning of the clause is the same either way.

The **ON CONFLICT** clause specifies an algorithm used to resolve constraint conflicts:

ROLLBACK When a constraint violation occurs, an immediate **ROLLBACK** occurs, thus ending the current transaction, and the command aborts with a return code of SQLITE_CONSTRAINT. If no transaction is active (other than the implied transaction that is created on every command) then this algorithm works the same as **ABORT**.

ABORT When a constraint violation occurs, the command backs out any prior changes it might have made and aborts with a return code of SQLITE_CONSTRAINT. But no **ROLLBACK** is executed, so changes from prior commands within the same transaction are preserved. This is the default behavior.

FAIL When a constraint violation occurs, the command aborts with a return code of SQLITE_CONSTRAINT. Any changes to the database that the command made prior to encountering the constraint violation are preserved and are not backed out. For example, if an **UPDATE** statement encountered a constraint violation on the 100th row that it attempts to update, then the first 99 row changes are preserved but changes to rows 100 and beyond never occur.

IGNORE When a constraint violation occurs, the one row that contains the constraint violation is not inserted or changed. But the command continues executing normally. Other rows before and after the row that contained the constraint violation continue to be inserted or updated normally. No error is returned.

REPLACE When a UNIQUE constraint violation occurs, the pre-existing rows that are causing the constraint violation are removed prior to inserting or

updating the current row. Thus, the insertion or update always occurs. The command continues executing normally. No error is returned. If a **NOT NULL** constraint violation occurs, the NULL value is replaced by the default value for that column. If the column has no default value, then the **ABORT** algorithm is used. If a **CHECK** constraint violation occurs, then the **IGNORE** algorithm is used.

When this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows. This may change in a future release.

The algorithm specified in the **OR** clause of a **INSERT** or **UPDATE** overrides any algorithm specified in a **CREATE TABLE**. If no algorithm is specified anywhere, the **ABORT** algorithm is used.

PRAGMA

Modify or query the library

Synopsis:

PRAGMA *name* **[=** *value***]** **|** *function***(** *arg***)**

Description:

The **PRAGMA** command is a special command used to modify the operation of the QDB process or to query the library for internal (non-table) data. The **PRAGMA** command is issued using the same interface as other QDB commands (e.g. **SELECT** or **INSERT**), but is different in the following important respects:

- Specific pragma statements may be removed and others added in future releases of QDB. Use with caution!
- No error messages are generated if an unknown pragma is issued. Unknown pragmas are simply ignored. This means if there is a typo in a pragma statement the library does not inform the user of the fact.
- Some pragmas take effect during the SQL compilation stage, not the execution stage. This means if using the C-language *sqlite3_prepare(), sqlite3_step(), sqlite3_finalize()* API (or similar in a wrapper interface), the pragma may be applied to the library during the *sqlite3_prepare()* call.
- The pragma command is unlikely to be compatible with any other SQL engine.

The pragmas that take an integer value also accept symbolic names. The strings **on**, **true**, and **yes** are equivalent to 1. The strings **off**, **false**, and **no** are equivalent to 0. These strings are case-insensitive, and do not require quotes. An unrecognized string will be treated as 1, and will not generate an error. When the value is returned, it is as an integer.

The available pragmas fall into four basic categories:

1 Pragmas used to modify the operation of the QDB process in some manner, or to query for the current mode of operation:

- Auto Vacuum
- Cache Size
- Case Sensitivity
- Count Changes
- Default Cache Size
- Full Column Names
- Full Column Names
- Legacy File Format
- Page Size

- Short Column Names
- Synchronous
- Temp Store

2 Pragmas used to query the schema of the current database:

- Foreign Key List
- Index Info
- Index List
- Table Info

3 Pragmas used to query or modify the databases two version values, the schema-version and the user-version:

- Schema and User Version

4 Pragmas used to debug the library and verify that database files are not corrupted:

- Integrity Check

Auto vacuum

```
PRAGMA auto_vacuum;
PRAGMA auto_vacuum = 0 | 1;
```

Query or set the auto-vacuum flag in the database.

Normally, when a transaction that deletes data from a database is committed, the database file remains the same size. Unused database file pages are marked as such and reused later on, when data is inserted into the database. In this mode the **VACUUM** command or *qdb_vacuum()* is used to reclaim unused space.

When the auto-vacuum flag is set, the database file shrinks when a transaction that deletes data is committed (The **VACUUM** command is not useful in a database with the auto-vacuum flag set). To support this functionality, the database stores extra information internally, resulting in slightly larger database files than would otherwise be possible.

It is possible to modify the value of the auto-vacuum flag only before any tables have been created in the database. No error message is returned if an attempt to modify the auto-vacuum flag is made after one or more tables have been created.

Auto vacuum mode is off by default. Frequent vacuum operations can be costly on storage media with slow write-access times (such as NOR flash memory); when databases are stored on such media, you should consider using *qdb_vacuum* (or the **VACUUM** SQL statement) rather than turning on auto-vacuum mode.

Cache size

```
PRAGMA cache_size;
PRAGMA cache_size = Number-of-pages;
```

Query or change the maximum number of database disk pages that QDB will hold in memory at once. Each page uses about 1.5 KB of memory. The default cache size is 2000 pages. If you are doing **UPDATE**s or **DELETE**s that change many rows of a database and you do not mind if QDB uses more memory, you can increase the cache size for a possible speed improvement.

When you change the cache size using the **cache_size** pragma, the change endures only for the current session. The cache size reverts to the default value when the database is closed and reopened. Use the **default_cache_size** pragma to permanently change the cache size.

Case sensitivity

```
PRAGMA case_sensitive_like;
PRAGMA case_sensitive_like = 0 | 1;
```

The default behavior of the **LIKE** operator is to ignore case for Latin1 characters. Hence, by default **'a' LIKE 'A'** is true. The **case_sensitive_like** pragma can be turned on to change this behavior. When **case_sensitive_like** is enabled, **'a' LIKE 'A'** is false, but **'a' LIKE 'a'** is still true.

Count changes

```
PRAGMA count_changes;
PRAGMA count_changes = 0 | 1;
```

Query or change the *count-changes* flag. Normally, when the *count-changes* flag is not set, **INSERT**, **UPDATE** and **DELETE** statements return no data. When *count-changes* is set, each of these commands returns a single row of data consisting of one integer value: the number of rows inserted, modified or deleted by the command. The returned change count does not include any insertions, modifications or deletions performed by triggers.

Default cache size

```
PRAGMA default_cache_size;
PRAGMA default_cache_size = Number-of-pages;
```

Query or change the maximum number of database disk pages that QDB will hold in memory at once. Each page uses 1 KB on disk and about 1.5 KB in memory. This pragma works like the **cache_size** pragma with the additional feature that it changes the cache size persistently. With this pragma, you can set the cache size once and that setting is retained and reused every time you reopen the database.

Full column names

```
PRAGMA full_column_names;
PRAGMA full_column_names = 0 | 1;
```

Query or change the *full-column-names* flag. This flag affects the way QDB names columns of data returned by **SELECT** statements when the expression for the column is a table-column name or the wildcard *****. Normally, such result columns are named

table-name|*alias column-name* if the **SELECT** statement joins two or more tables together, or simply *column-name* if the **SELECT** statement queries a single table. When the *full-column-names* flag is set, such columns are always named *table-name*|*alias column-name* regardless of whether or not a join is performed.

If both the *short-column-names* and *full-column-names* are set, then the behavior associated with the *full-column-names* flag is exhibited.

Legacy file format

```
PRAGMA legacy_file_format;
PRAGMA legacy_file_format = ON | OFF
```

This pragma sets or queries the value of the *legacy_file_format* flag. When this flag is on, new SQLite databases are created in a file format that is readable and writable by all versions of SQLite going back to 3.0.0. When the flag is off, new databases are created using the latest file format which might not be readable or writable by older versions of SQLite.

This flag affects only newly created databases. It has no effect on databases that already exist.

Page size

```
PRAGMA page_size;
PRAGMA page_size = bytes;
```

Query or set the page size of the database. The page size may be set only if the database has not yet been created. The page size must be a power of two greater than or equal to 512 and less than or equal to 8192.

Short column names

```
PRAGMA short_column_names;
PRAGMA short_column_names = 0 | 1;
```

Query or change the *short-column-names* flag. This flag affects the way QDB names columns of data returned by **SELECT** statements when the expression for the column is a table-column name or the wildcard *****. Normally, such result columns are named *table-name*|*alias column-name* if the **SELECT** statement joins two or more tables together, or simply *column-name* if the **SELECT** statement queries a single table. When the *short-column-names* flag is set, such columns are always named *column-name* regardless of whether or not a join is performed.

If both the *short-column-names* and *full-column-names* are set, then the behavior associated with the *full-column-names* flag is exhibited.

Synchronous

```
PRAGMA synchronous;
PRAGMA synchronous = FULL; (2)
PRAGMA synchronous = NORMAL; (1)
PRAGMA synchronous = OFF; (0)
```

Query or change the setting of the *synchronous* flag. The first (query) form will return the setting as an integer. When synchronous is FULL (2), the QDB database engine will pause at critical moments to make sure that data has actually been written to the

disk surface before continuing. This ensures that if the operating system crashes or if there is a power failure, the database will be uncorrupted after rebooting. FULL synchronous is very safe, but it is also slow. When synchronous is NORMAL, the QDB database engine will still pause at the most critical moments, but less often than in FULL mode. There is a very small (though non-zero) chance that a power failure at just the wrong time could corrupt the database in NORMAL mode. But in practice, you are more likely to suffer a catastrophic disk failure or some other unrecoverable hardware fault. With synchronous OFF (0), QDB continues without pausing as soon as it has handed data off to the operating system. If the application running QDB crashes, the data will be safe, but the database might become corrupted if the operating system crashes or the computer loses power before that data has been written to the disk surface. On the other hand, some operations are as much as 50 or more times faster with synchronous OFF.

The default seting is FULL.

Temp store

```
PRAGMA temp_store;
PRAGMA temp_store = DEFAULT; (0)
PRAGMA temp_store = FILE; (1)
PRAGMA temp_store = MEMORY; (2)
```

Query or change the setting of the *temp_store* parameter. When *temp_store* is DEFAULT (0), the compile-time C preprocessor macro *TEMP_STORE* is used to determine where temporary tables and indexes are stored. When *temp_store* is MEMORY (2), temporary tables and indexes are kept in memory. When *temp_store* is FILE (1), temporary tables and indexes are stored in a file. The **`temp_store_directory`** pragma can be used to specify the directory containing this file. When the *temp_store* setting is changed, all existing temporary tables, indexes, triggers and views are immediately deleted.

It is possible for the library compile-time C preprocessor symbol *TEMP_STORE* to override this pragma setting. The following table summarizes the interaction of the *TEMP_STORE* preprocessor macro and the **`temp_store`** pragma. It shows the storage used for TEMP tables and indexes:

TEMP_STORE	PRAGMA temp_store	Storage
0	*Any*	File
1	0	File
1	1	File
1	2	Memory
2	0	Memory
2	1	File

continued...

TEMP_STORE	PRAGMA temp_store	Storage
2	2	Memory
3	*Any*	Memory

Foreign key list

PRAGMA foreign_key_list(*table-name***);**

For each foreign key that references a column in the argument table, invoke the callback function with information about that foreign key. The callback function will be invoked once for each column in each foreign key.

Index info

PRAGMA index_info(*index-name***);**

For each column that the named index references, invoke the callback function once with information about that column, including the column name and the column number.

Index list

PRAGMA index_list(*table-name***);**

For each index on the named table, invoke the callback function once with information about that index. Arguments include the index name and a flag to indicate whether or not the index must be unique.

Table info

PRAGMA table_info(*table-name***);**

For each column in the named table, invoke the callback function once with information about that column, including the column name, data type, whether or not the column can be NULL, and the default value for the column.

Schema and user version

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = integer ;
PRAGMA [database.]user_version;
PRAGMA [database.]user_version = integer ;
```

The pragmas **schema_version** and **user_version** are used to set or get the value of the *schema-version* and *user-version*, respectively. Both the *schema-version* and the *user-version* are 32-bit signed integers stored in the database header.

The *schema-version* is usually manipulated only internally by QDB. It is incremented by QDB whenever the database schema is modified (by creating or dropping a table or index). The schema version is used by QDB each time a query is executed to ensure that the internal cache of the schema used when compiling the SQL query matches the schema of the database against which the compiled query is actually executed.

Subverting this mechanism by using **PRAGMA schema_version** to modify the schema-version is potentially dangerous and may lead to program crashes or database corruption. Use with caution!

The *user-version* is not used internally by QDB. It may be used by applications for any purpose.

Integrity check

PRAGMA integrity_check;
PRAGMA integrity_check(*integer***)**

The command does an integrity check of the entire database. It looks for out-of-order records, missing pages, malformed records, and corrupt indexes. If any problems are found, then strings are returned (as multiple rows with a single column per row) which describe the problems. At most *integer* errors will be reported before the analysis quits. The default value for *integer* is 100. If no errors are found, a single row with the value **ok** is returned.

REINDEX

Recreate indexes from scratch

Synopsis:

```
REINDEX collation name |
    ( [database-name .] table | index-name )
```

Description:

The **REINDEX** command is used to delete and recreate indexes from scratch. This is useful when the definition of a collation sequence has changed.

In the first form, all indexes in all attached databases that use the named collation sequence are recreated. In the second form, if **[***database-name* **.]** **{** *table-name* **|** *index-name* **}** identifies a table, then all indexes associated with the table are rebuilt. If an index is identified, then only this specific index is deleted and recreated.

If no *database-name* is specified and there exists both a table or index and a collation sequence of the specified name, then indexes associated with the collation sequence only are reconstructed. This ambiguity may be dispelled by always specifying a *database-name* when reindexing a specific table or index.

REPLACE

Alias for **INSERT OR REPLACE**

Synopsis:

```
REPLACE INTO [database-name .] table-name [( column-list )]
  VALUES ( value-list ) |
REPLACE INTO [database-name .] table-name [( column-list )]
  select-statement
```

Description:

The **REPLACE** command is an alias for the **INSERT OR REPLACE** variant of the **INSERT** command. This alias is provided for compatibility with MySQL.

Synopsis:

```
SELECT [ALL | DISTINCT] result [FROM table-list]
[WHERE expr]
[GROUP BY expr-list]
[HAVING expr]
[compound-op select]*
[ORDER BY sort-expr-list]
[LIMIT integer [( OFFSET | , ) integer]]

result =
result-column [, result-column]*

result-column =
* | table-name . * | expr [ [AS] string ]

table-list =
table [join-op table join-args]*

table =
table-name [AS alias] |
( select ) [AS alias]

join-op =
, | [NATURAL] [LEFT | RIGHT | FULL]
    [OUTER | INNER | CROSS] JOIN

join-args =
[ON expr] [USING ( id-list )]

sort-expr-list =
expr [sort-order] [, expr [sort-order]]*

sort-order =
[ COLLATE collation-name ] [ ASC | DESC ]

compound_op =
UNION | UNION ALL | INTERSECT | EXCEPT
```

Description:

The **SELECT** statement is used to query the database. The result of a **SELECT** is zero or more rows of data where each row has a fixed number of columns. The number of columns in the result is specified by the expression list in between the **SELECT** and **FROM** keywords. Any arbitrary expression can be used as a result. If a result expression is *****, then all columns of all tables are substituted for that one expression. If the expression is the name of a table followed by **.***, then the result is all columns in that one table.

DISTINCT keyword

The **DISTINCT** keyword causes a subset of result rows to be returned, in which each result row is different. NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword **ALL**.

The query is executed against one or more tables specified after the **FROM** keyword. If multiple tables names are separated by commas, then the query is against the cross join of the various tables. The full SQL-92 join syntax can also be used to specify joins. A sub-query in parentheses may be substituted for any table name in the **FROM** clause. The entire **FROM** clause may be omitted, in which case the result is a single row consisting of the values of the expression list.

WHERE clause

The **WHERE** clause can be used to limit the number of rows over which the query operates.

GROUP BY clauses

The **GROUP BY** clauses causes one or more rows of the result to be combined into a single row of output. This is especially useful when the result contains aggregate functions. The expressions in the **GROUP BY** clause do *not* have to be expressions that appear in the result. The **HAVING** clause is similar to **WHERE** except that **HAVING** applies after grouping has occurred. The **HAVING** expression may refer to values, even aggregate functions, that are not in the result.

ORDER BY clauses

The **ORDER BY** clause causes the output rows to be sorted. The argument to **ORDER BY** is a list of expressions that are used as the key for the sort. The expressions do not have to be part of the result for a simple **SELECT**, but in a compound **SELECT** each sorting expression must exactly match one of the result columns. Each sorting expression may be optionally followed by a **COLLATE** keyword and the name of a collating function used for ordering text and/or keywords **ASC** or **DESC** to specify the sort order.

LIMIT clauses

The **LIMIT** clause places an upper bound on the number of rows returned in the result. A negative **LIMIT** indicates no upper bound. The optional **OFFSET** following **LIMIT** specifies how many rows to skip at the beginning of the result set. In a compound query, the **LIMIT** clause may appear only on the final **SELECT** statement. The limit is applied to the entire query, not to the individual **SELECT** statement to which it is attached. Note that if the **OFFSET** keyword is used in the **LIMIT** clause, then the limit is the first number and the offset is the second number. If a comma is used instead of the **OFFSET** keyword, then the offset is the first number and the limit is the second number. This seeming contradiction is intentional — it maximizes compatibility with legacy SQL database systems.

Compund SELECT statements

A compound **SELECT** is formed from two or more simple **SELECT**s connected by one of the operators **UNION**, **UNION ALL**, **INTERSECT**, or **EXCEPT**. In a compound **SELECT**, all the constituent **SELECT**s must specify the same number of result columns. There may be only a single **ORDER BY** clause at the end of the compound **SELECT**. The **UNION** and **UNION ALL** operators combine the results of the **SELECT**s to the right and left into a single big table. The difference is that in **UNION** all result rows are distinct, whereas in **UNION ALL** there may be duplicates. The **INTERSECT** operator takes the intersection of the results of the left and right **SELECT**s. **EXCEPT** takes the result of left **SELECT** after removing the results of the right **SELECT**. When three or more **SELECT**s are connected into a compound, they group from left to right.

TRANSACTION

Manually start, end, commit, or rollback a transaction

Synopsis:

BEGIN [DEFERRED | IMMEDIATE | EXCLUSIVE] [TRANSACTION [*name***]]**

END [TRANSACTION [*name***]]**

COMMIT [TRANSACTION [*name***]]**

ROLLBACK [TRANSACTION [*name***]]**

Description:

QDB supports transactions with rollback and atomic commit. The optional transaction name is ignored. QDB currently doesn't allow nested transactions.

No changes can be made to the database except within a transaction. Any command that changes the database (basically, any SQL command other than **SELECT**) will automatically start a transaction if one is not already in effect. Automatically started transactions are committed at the conclusion of the command.

Transactions can be started manually using the **BEGIN** command. Such transactions usually persist until the next **COMMIT** or **ROLLBACK** command. But a transaction will also **ROLLBACK** if the database is closed or if an error occurs and the **ROLLBACK** conflict-resolution algorithm is specified. See the documentation on the **ON CONFLICT** clause for additional information about the **ROLLBACK** conflict-resolution algorithm.

In QDB, transactions can be deferred, immediate, or exclusive. Deferred means that no locks are acquired on the database until the database is first accessed. Thus with a deferred transaction, the **BEGIN** statement itself does nothing. Locks are not acquired until the first read or write operation. The first read operation against a database creates a SHARED lock and the first write operation creates a RESERVED lock. Because the acquisition of locks is deferred until they are needed, it is possible that another thread or process could create a separate transaction and write to the database after the **BEGIN** on the current thread has executed. If the transaction is immediate, then RESERVED locks are acquired on all databases as soon as the **BEGIN** command is executed, without waiting for the database to be used.

After a **BEGIN IMMEDIATE**, you are guaranteed that no other thread or process will be able to write to the database or do a **BEGIN IMMEDIATE** or **BEGIN EXCLUSIVE**. Other processes can continue to read from the database, however. An exclusive transaction causes EXCLUSIVE locks to be acquired on all databases. After a **BEGIN EXCLUSIVE**, you are guaranteed that no other thread or process will be able to read or write the database until the transaction is complete.

Locks

This is a description of the meaning of SHARED, RESERVED, and EXCLUSIVE locks:

SHARED	The database may be read but not written. Any number of processes can hold SHARED locks at the same time, hence there can be many simultaneous readers. But no other thread or process is allowed to write to the database file while one or more SHARED locks are active.
RESERVED	A RESERVED lock means that the process is planning on writing to the database file at some point in the future but that it is currently just reading from the file. Only a single RESERVED lock may be active at one time, though multiple SHARED locks can coexist with a single RESERVED lock.
EXCLUSIVE	An EXCLUSIVE lock is needed in order to write to the database file. Only one EXCLUSIVE lock is allowed on the file and no other locks of any kind are allowed to coexist with an EXCLUSIVE lock. In order to maximize concurrency, QDB works to minimize the amount of time that EXCLUSIVE locks are held.

The default behavior for QDB is a deferred transaction.

The **COMMIT** command does not actually perform a commit until all pending SQL commands finish. Thus if two or more **SELECT** statements are in the middle of processing and a **COMMIT** is executed, the commit will not actually occur until all **SELECT** statements finish.

Returns:

An attempt to execute **COMMIT** might result in an SQLITE_BUSY return code. This indicates that another thread or process has a read lock on the database that prevented the database from being updated. When **COMMIT** fails in this way, the transaction remains active and the **COMMIT** can be retried later after the reader has had a chance to clear.

UPDATE

Change the value of columns

Synopsis:

```
UPDATE [ OR conflict-algorithm ] [database-name.] table-name
SET column-name = expr [, column-name = expr]*
[WHERE expr]
```

Description:

The **UPDATE** statement is used to change the value of columns in selected rows of a table. Each assignment in an **UPDATE** specifies a column name to the left of the equals sign and an arbitrary expression to the right. The expressions may use the values of other columns. All expressions are evaluated before any assignments are made. A **WHERE** clause can be used to restrict which rows are updated.

The optional *conflict-clause* allows the specification of an alternative constraint conflict resolution algorithm to use during this one command. See **ON CONFLICT** for additional information.

VACUUM

Clean up a table or index

Synopsis:

VACUUM [*index-or-table-name***]**

Description:

The **VACUUM** command is a QDB extension modeled after a similar command found in PostgreSQL. If **VACUUM** is invoked with the name of a table or index, then it is supposed to clean up the named table or index. The index or table name argument is ignored.

When an object (table, index, or trigger) is dropped from the database, it leaves behind empty space. This makes the database file larger than it needs to be, but can speed up insertions. In time, insertions and deletions can leave the database file structure fragmented, which slows down disk access to the database contents.

The **VACUUM** command cleans the main database by copying its contents to a temporary database file and reloading the original database file from the copy. This eliminates free pages, aligns table data to be contiguous, and otherwise cleans up the database file structure. It is not possible to perform the same process on an attached database file.

This command will fail if there is an active transaction. This command has no effect on an in-memory database.

An alternative to using the **VACUUM** command is the auto-vacuum mode. You can set the auto-vacuum mode using the **PRAGMA** SQL extension:

```
qdb_statement(&db, "PRAGMA auto_vacuum = 1;"); // on
qdb_statement(&db, "PRAGMA auto_vacuum = 0;"); // off
```

See also:

qdb_vacuum(), PRAGMA

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W