QDB Developer's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Thursday, July 9, 2015

Table of 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:
Launching QDB and managing databases	*Getting Started with QDB* (p. 11)
Using the QDB command utility	*QDB Command Line* (p. 27)
Executing SQL statements from the command line	*QDB Client* (p. 41)
Writing a client application to use QDB	*QDB Example* (p. 45)
Supported data types	*Datatypes in QDB* (p. 53)
Opcodes used to execute SQL statements	*QDB Virtual Machine Opcodes* (p. 65)
Writing your own SQL or collation functions	*Writing User-Defined Functions* (p. 95)
Client API functions	*QDB API Reference* (p. 107)
SQL commands	*QDB SQL Reference* (p. 187)

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`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

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.

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

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 typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

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

Chapter 1
Getting Started with QDB

The QNX Database (QDB) server is a small-footprint, embeddable SQL database server. It is designed as an easy-to-configure QNX 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
- network access to databases using QNet
- simple API for accessing the database
- result storing for repeated use
- result passing from one application to another
- in-memory database support
- auto-attach support, to join disparate databases into a single, virtual database

Starting the QDB server

QDB uses the Persistent Publish/Subscribe (PPS) service to dynamically configure databases. Before starting QDB, you must set up PPS. You can then use QDB to load and unload databases.

To set up and use QDB:

The first step is optional and is possible only if you've installed the QNX SDK for Apps and Media. The action described herein improves performance but isn't necessary to successfully use QDB.

1. In a QNX Neutrino terminal, enter `io-fs-media` to start the IO service for reading and writing to RAM-based locations.

   Although it's not required, we recommend running your QDB databases in RAM; for example, from a `tmpfs` filesystem. You can also run databases from locations in QNX filesystems and flash filesystems but performance may suffer with these two filesystem types due to the inherent slowness in writing to the storage media.

2. Enter `pps` to start PPS as a background process.

   PPS creates a root directory (`/pps` by default) to store the PPS configuration objects, which are text files that describe the configuration of QDB databases.

3. Enter `mkdir -p /pps/qnx/qdb` to create the directory structure used in the PPS configuration path.

4. Enter `qdb` followed by any desired options to start the QDB server.

   For debugging purposes, you should start `qdb` with `-v` and `-V` options to get verbose output. The `-v` option is cumulative, with each additional `v` adding a level of verbosity, up to seven levels. The `-V` option sends output to the console and to the `sloginfo` log file.

The PPS and QDB server processes are running. You can now add and remove database configuration objects to and from the directory named in the PPS configuration path to dynamically load and unload databases.

Loading QDB databases

To load a database into QDB:

- Copy an existing configuration object or, from a client application using the *open()* and *write()* system calls, output the list of configuration attributes and values into the `config` subdirectory under the PPS configuration path (`/pps/qnx/qdb/`).

  The configuration object is a text file, accessed through PPS, that lists database parameters as attribute-value pairs. This file must name the database storage file and can optionally name SQL command files that define the schema of the database and populate it with initial data, as explained in "*Database configuration objects* (p. 17)".

  QDB parses the configuration object's contents and tries to load the database with the same name as the object. QDB then creates a status object that indicates the database state after the loading attempt. If the storage file named in the configuration object doesn't exist, QDB creates the storage file and if directed, populates the database with initial data.

The directory that will contain the new storage file must exist before you start loading the database. Otherwise, the loading fails and QDB sets the status to **Error**.

Unloading QDB databases

To unload a database from QDB:

- Delete the configuration object with the same name as the database to unload, from the `config` subdirectory under the PPS configuration path (`/pps/qnx/qdb/`).

  QDB removes the database from its control. Any databases that are attached go into the **AttachWait** state and can't be accessed until the missing database is reloaded (see the "*Database configuration objects* (p. 17)" section for details on the `AutoAttach` parameter). QDB also deletes the status file in the `status` subdirectory because the database is no longer visible.

No files referred to in the configuration object, including the raw storage file, are deleted when the database is unloaded. To delete a database, the storage file must be deleted in the filesystem.

PPS configuration path

QDB requires a dedicated location for storing the database configuration and status objects. This location is called the *PPS configuration path*. The database configuration objects are text files that control database setup. The status objects are text files that report the database states.

The `config` subdirectory

The contents of the `config` subdirectory under the PPS configuration path determine which databases are currently visible to QDB. QDB monitors this location for file additions and removals. When a file is added to `config`, QDB parses the file's contents and tries to load the database with the same name as the file. When a file is removed from the subdirectory, QDB unloads the database, meaning it removes the database from visibility but doesn't delete it.

The `status` subdirectory

The files in the `status` subdirectory report the states of databases that QDB attempted to load. Each status file contains a **Status** attribute, whose value can be one of:

Initializing

The database configuration object has been seen and the database is initializing.

Error

There was an error with the configuration of the database.

Valid

The database has been configured and can be accessed.

AttachWait

The database is waiting for an auto-attached database to become available before configuration can continue.

Overriding the PPS configuration path

The default PPS configuration path for QDB is `/pps/qnx/qdb`. You can override this default path to run multiple instances of QDB. This is useful if you want to run databases with different connection settings.

The PPS configuration path is based on the PPS mountpoint. Before starting QDB, you can start PPS with either the default mountpoint (`/pps`) by using no command-line options, or with an overriden mountpoint by using the `-m` option, as in the following example:

```
pps -m /temp
```

You can then override the PPS configuration path by running the `qdb` utility with the `-c` option, as follows:

```
qdb -c /temp/qdb
```

The directory named in the PPS configuration path must exist before you start QDB. You can launch multiple instances of QDB with different configuration paths so long as each path is based on the PPS mountpoint. So in the above example, you could also run an instance of QDB with the PPS configuration path `/temp/customerdbs`.

We recommend you use separate QDB device mountpoints for separate QDB instances; this is done with the `-n` option on the `qdb` command line. For details, see the *QDB Command Line* (p. 27) chapter.

Database configuration objects

QDB databases are managed with configuration objects, each of which configures one database. The configuration objects are text files that specify the paths to the database schema and storage files as well as policy settings such as backups.

The files are named after the databases that they configure. For example, suppose you have two media content databases named `hdd` and `cdrom`. You would then create two configuration objects named `hdd` and `cdrom`. For an overview of all database-related files, see "*Summary of database files* (p. 25)".

By using a separate SQLite storage file for each database, QDB allows you to dynamically load and unload individual databases so you can keep in memory only the data needed for the active client application.

QDB doesn't support "on-the-fly" configuration changes after a database is loaded. To modify the configuration, you must unload the database, update its configuration object, and reload the database. You must also ensure that the changes are compatible with the previous configuration.

Configuration object contents

The database configuration object gets parsed by QDB to set up the database. Lines specifying parameters are in the form *key*`::`*value*. Unknown parameter types are ignored and so they can be made into comments, but you must still use the `::` delimiter on the comment line. For example, you can enter `MyComment::` on a line and QDB will treat it as an unsupported parameter and ignore that line when parsing the file. For an example of a database configuration, see "*Sample configuration object* (p. 21)".

The configuration object must contain this parameter:

`Filename`

The name of the database storage file, which is the raw SQLite file. It must have an absolute path but it can refer to any file location. At database loading time, either this file or the directory in which it will be created must exist; otherwise, the loading attempt fails and QDB sets the status to **Error**. If the database file doesn't exist but its parent directory does, the file is restored from the newest valid backup (if possible) or a blank database file is created.

All other database configuration parameters are optional. You can define these parameters:

`AutoAttach`

A comma-separated list of other databases to attach to the current one (using the SQL `ATTACH DATABASE` statement) whenever a database connection is established.

Attached databases are a convenience to provide access to tables that are physically stored in different database files. 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 transfering parts of a database to different storage medias such as RAM filesystems.

QDB allows you to include attached databases in other maintenance operations, such as backup or vacuum.

If any attached database is unavailable at loading time, QDB sets the current database's status to **AttachWait** and makes the database inaccessible.

`BackupAttached`, `SizeAttached`, `VacuumAttached`

These entries control what maintenance operations should apply to attached databases when a command is issued to the main database. Each parameter lists the attached databases affected by the operation.

Suppose you have a main database named `mp3_tunes_0` with two attached databases named `mp3_tunes_1` and `mp3_tunes_2` and you define these paramaters:

```
AutoAttach::mp3_tunes_1,mp3_tunes_2
VacuumAttached::mp3_tunes_1
```

In this case, a *qdb_vacuum()* (p. 183) operation on `mp3_tunes_0` will also vacuum `mp3_tunes_1` but not `mp3_tunes_2`.

Any database named in an operation-based attachment parameter such as `VacuumAttached` must also be named in `AutoAttach`, or that database won't be processed during the operation.

For more details on the scope of maintenance operations for attached databases, refer to *qdb_backup()* (p. 108), *qdb_getdbsize()* (p. 137), and *qdb_vacuum()* (p. 183).

`BackupDir`

A comma-separated list of directories that store database backups. When you specify multiple directories, they're used in rotation to store the backup files. This feature ensures that should a backup be interrupted or aborted by a power failure, another older backup is still available.

These directories must exist at loading time (though they don't need to contain valid backups); otherwise, the loading attempt fails and QDB sets the database status to **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 when restoring a missing or corrupt database.

`BackupVia`

An interim directory that the database is copied into as part of the backup. To make sure the backup is consistent, QDB places a read lock on the database while copying and compressing it, so the database may be locked a long time if the destination is slow (e.g., flash).

Suppose you specify:

```
BackupVia::/dev/shmem
```

When backing up, QDB locks the database, copies it to `/dev/shmem`, and releases the lock. Then, in a second step, QDB copies and compresses the database into the location specified by `BackupDir`, without needing to lock the database.

`ClientSchemaFile`

The name of the file (with an absolute path) that contains the SQL commands to execute every time a client calls *qdb_connect()* (p. 131).

Use this feature for changing database settings that can't be premanently modified. An example would be the PRAGMA commands, which modify non-table data such as journaling mode or case-sensitive-like status. Don't use client schema files to do regular database work because doing so will slow down new connections.

You can also use this mechanism to implement cross-database triggers.

`Collation`, `Function`

These entries install user-provided collation (sorting) routines and scalar/aggregate functions. The argument format is a comma-separated list of library symbols in the form *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.

Unlike the paths to other key files, the library file paths can be relative or absolute. Relative paths are looked up in the library search directories (refer to *dlopen()* in the QNX Neutrino *C Library Reference* for more detail). In this release, you can install only one collation function but many scalar functions per database. For the latter type of functions, you can specify symbols from as many shared libraries as you want. For example, you could write:

```
Collation::UTF-8_Sort@libsort.so
Function::sampleData@libstats.so,implToMtrc@libunits.so
```

For information about the function description structures and the setup and sorting functions that you must define, see the *Writing User-Defined Functions* (p. 95) chapter.

QDB checks for the existence of the libraries and the specified symbols at loading time. If any are not found, the loading attempt fails and QDB sets the database status to **Error**.

`Compression`

The compression algorithm to apply to backups. The supported options are:

- `none` (for no compression, which is the default)
- `lzo` (for LZO compression)
- `bzip` (for BZIP2 compression)
- `diocopy` (for direct I/O copy)

The `lzo` compression algorithm is the fastest but the `bzip` algorithm offers the highest compression. Direct I/O doesn't perform any compression; instead, QDB uses the *`fileset`* (p. 235) utility to copy the database using direct memory access (DMA). Direct I/O is a fast way to back up data if the persistent storage supports DMA.

The compressed files are named by adding the appropriate extensions to the original database filenames. By default, backup files aren't compressed.

`CompressionVia`

This entry is used with the `BackupVia` entry and any `Compression` setting specified for the backup. By default, the `BackupVia` feature first makes a raw/uncompressed copy of the database in the temporary directory and then performs the compression. This works if you have enough space and it 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 copy. This

option is false by default; if you set it to true, the compression is done in the first step.

`DataSchemaFile`

The name of the file (with an absolute path) that creates the initial data in the database. This text file contains the SQL commands to populate the database when it is created.

This option is processed only if the `SchemaFile` option is set.

`SchemaFile`

The name of the file (with an absolute path) that contains the SQL commands to create the initial schema of tables, indexes, and views for the database. The schema file is used only to set up the database if it didn't already exist.

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

Sample configuration object

This basic database configuration names the files for storing the database, defining the schema, and populating the database with initial data:

```
Filename::/root/tmpfs/cdrom0_db
SchemaFile::/etc/mm/sql/mmsync.sql
DataSchemaFile::/etc/mm/sql/mmsync_data.sql
```

You would give the configuration object the same name as the database storage file (`cdrom0_db`).

Database storage

QDB uses raw SQLite files to store databases individually. Each database configuration object must provide a path to the raw storage file for the database being defined. Storage files for different databases can be kept in different areas of the filesystem or in different filesystems altogether.

Database storage files can be stored on any QNX or POSIX filesystem with read/write access (including memory-based filesystems, such as `tmpfs`). QDB can run from QNX filesystems visible via Qnet, but can't run from a CIFS or NFS filesystem or a non-POSIX system such as `/dev/shmem/`.

When loading a database, QDB creates the storage file in the location specified in the configuration object if the storage file doesn't exist.

When unloading a database, QDB leaves the storage file intact; it's up to the application to take appropriate action by either copying or deleting the storage file.

Schema files

Schema files contain all the SQL commands to create the database schema the way you want, populate the database after creation, or run whenever a client connects to the database.

Schema creation file

This file defines the tables, views, indexes, and triggers that make up the database. Here's an example of a schema creation file:

```
CREATE TABLE library (
  fid             INTEGER PRIMARY KEY AUTOINCREMENT,
  ftype           INTEGER DEFAULT 0 NOT NULL,
  last_sync       INTEGER DEFAULT 0 NOT NULL,
  last_played     INTEGER DEFAULT 0 NOT NULL,
  filename        TEXT DEFAULT '' NOT NULL,
  offset          TEXT DEFAULT '' NOT NULL
);
CREATE TABLE library_genres (
  genre_id        INTEGER PRIMARY KEY AUTOINCREMENT,
  genre           TEXT
);
CREATE INDEX library_genres_index_1 on library_genres(genre);

CREATE TABLE library_artists (
  artist_id       INTEGER PRIMARY KEY AUTOINCREMENT,
  artist          TEXT
);
CREATE INDEX library_artists_index_1 on library_artists(artist);
```

Data schema file

This file contains SQL commands to populate the database with initial data just after creation. Here's an example of a data schema file:

```
INSERT INTO library_genres(genre_id, genre)
  VALUES(1, NULL);
INSERT INTO library_artists(artist_id, artist)
  VALUES(1, NULL);
INSERT INTO library_artists(artist_id, artist)
  VALUES(2, "The Beatles");
INSERT INTO library_artists(artist_id, artist)
  VALUES(3, "The Rolling Stones");
INSERT INTO library_artists(artist_id, artist)
  VALUES(4, "The Doors");
```

Client schema file

This file contains SQL commands to execute whenever a client connects to the database. Here's an example of a client schema file:

```
PRAGMA journal_mode = PERSIST;
PRAGMA case_sensitive_like = true;
PRAGMA locking_mode = EXCLUSIVE;
```

Summary of database files

QDB database configuration uses many files. Suppose you want to create a database named `customerdb` and store its data in `/usr/local/db/` and its schema definition files in your home directory of `/home/user1/`, and that you're using the default PPS configuration path (`/pps/qnx/qdb/`). You would then need the following files to set up this database:

File	Purpose
`/pps/qnx/qdb/config/customerdb`	Text file acting as configuration object, which provides paths to other setup files and specifies policy settings such as auto-attached databases
`/pps/qnx/qdb/status/customerdb`	Status file created by QDB to indicate database state after attempted loading
`/usr/local/db/customerdb`	Raw SQLite file (created by QDB if necessary) that stores database content
`/home/user1/customerdb.sql`	Schema definition file that defines tables, views, indexes, and triggers that make up database
`/home/user1/customerdb_data.sql`	Data schema file that specifies initial data for populating database
`/home/user1/customerdb_connect.sql`	Client schema file that defines commands to run whenever a client connects
`/dev/qdb/customerdb`	QDB device file used for PPS object publishing

Chapter 2
QDB Command Line

Configure and maintain QDB databases

Synopsis:

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

Options:

-A

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

-c *config_path*

Specify an overridden PPS configuration path. See the "*PPS configuration path* (p. 15)" section 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* (p. 34)" section for more information.

-D

Disable the shared cache. You should use this option only if you need to debug *shared caching* (p. 35).

-I *test*

Perform a database integrity test at loading time. The *test* can be one of:

- `none`
- `basic`

- `partial`
- `full`

See the "*Database integrity testing* (p. 33)" section for more information.

`-n` *mountpoint*

QDB resource manager mountpoint. The default is `/dev/qdb`.

`-N` *control*

Name of the database control entry. The default is `.control`.

`-o` *option* [,*option2*...]

Configure miscellaneous options. The options are:

- `unblock`=0 | 1 — set whether or not to install an unblock handler (i.e., to allow a signal to interrupt an SQL operation).
- `threadmax` — set the maximum number of threads to allocate to `qdb`; the default is 64.
- `threadhi` — set the maximum number of threads that can be kept in a blocked state ready to work.
- `threadlo` — set the minimum number of threads to keep in a blocked state ready for work.
- `rwbias=r|w` — set preferential access for readers (with 'r') or writers (with 'w') when multiple threads are contending for a database lock. By default, no preferential access is granted; you can override this setting to improve performance.
- `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 and the time it took to execute them, after `qdb` finishes executing them. You can additionally specify the `-W` *time* option to log only SQL statements that take longer than the time specified in milliseconds. You must set verbosity (`-v`) to six for this feature to work.

- `verchk=none|major|minor|revision|strict` — check the compatibility between the SQLite version from which QDB was built and that of the installed SQLite library.

  The default value is `strict`, which means the exact revision, including any patches, must match between two the SQLite versions. For example, if you're running a QDB service built from version 3.7.11-patch17 but your installed SQLite library is of version 3.7.11 (but no patch), the version check fails and QDB doesn't start.

  By relaxing the strictness of the version checking, you can use a SQLite library with a version different from the one that QDB was built from.

See also *thread_pool_create()* in the QNX Neutrino *C Library reference*.

-P ***permissions***

Define access permissions for the database and backup files. By default this is `0664`.

-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.

-R ***mode***

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

- `manual`
- `auto`
- `set`

See the "*Database recovery* (p. 36)" section for more information.

-s [*data***@]***routine***

Change the configuration of a user-defined collation sequence. The collation setup function expects data in the same format as you would specify it through *qdb_collation()* (p. 122). For example, `-s en_US@cldr` would name the `cldr` collation routine and invoke its setup function at startup, passing in the `en_US` string.

Note that the data portion of the argument is optional; specifying a `-s` argument without data lets you set the configuration of a collation to its default setting.

This release supports only one collation per database and hence, you can provide only one `-s` argument. Any user-defined collation has its setup function invoked at startup, regardless of whether you provided any data using this option. However, to pass nondefault data to the collation setup function, you must use this option and include data in its value.

`-t` *timeout* [`block` | `nonblock`]

Set the busy-wait timeout on database access, in milliseconds. By default, this is 5000 milliseconds. See the "*Busy timeout* (p. 38)" section for more information.

`-T` *timeout* [`block` | `nonblock`]

Set the busy-wait timeout on database connection, in milliseconds. By default, this is 5000 milliseconds. See the "*Busy timeout* (p. 38)" section for more information.

`-v`

Increase output verbosity. The more `-v` options you specify, the more verbose the output. You can specify up to seven `-v` levels. Messages are written to the `sloginfo` log.

`-V`

Replicate output messages to the console and the `sloginfo` log.

`-W` *time*

Used with the `-o profile` option to log only those SQL statements that take longer than *time* (which is specified in milliseconds). The default for *time* is 5000 milliseconds.

`-X` *path*

Set a script to run when `qdb` encounters a corrupt database. See "*Handling corrupt databases* (p. 39)".

Description:

The `qdb` utility lets you set the properties of a database connection. You can override default paths to the configuration object or the device file, set the creation and recovery modes, and configure miscellaneous options such as the number of threads used by `qdb`.

The `-v` option causes `qdb` to output the results of database operations to `sloginfo`; this verbose mode is useful for troubleshooting your setup and database usage. The verbose option can be used with the trace and profile settings under the `-o` option to log the results of SQL statement execution.

Temporary storage filesystem

The filesystem that `qdb` uses for temporary storage must support POSIX file locking. File locking is required for database vacuuming.

The `qdb` utility checks its temporary storage as follows:

- If the `tempstore` option (`-o tempstore`) is specified on the command line, `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, `qdb` sets the internal temporary storage to the location specified by the `tempstore` option. If any of the above conditions are not met, `qdb` logs errors to `slog` and fails to start up.
- If no `tempstore` option is specified on the command line, `qdb` uses the environment variable ***TMPDIR*** to obtain the location to use for temporary storage. The `qdb` utility then 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, `qdb` sets the internal temporary storage to the value of ***TMPDIR***. If any of the above conditions are not met, `qdb` logs errors to `slog` and fails to start up.

Database integrity testing

At startup, the `qdb` utility tests the integrity of databases, according to the `-I` option specified. It executes statements based on this option's setting, as follows:

- `none` — don't perform a database integrity check.
- `basic` — verify only that `qdb` can parse a string.
- `partial` — validate the PRAGMA database list. This is equivalent to running the `PRAGMA database_list;` command.
- `full` — validate the database integrity. This is equivalent to running the `PRAGMA integrity_check;` command.

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

You can execute SQL statements on your QDB databases from the command line using the *qdbc* (p. 41) utility.

Sharing connections between clients

You can allow multiple clients to share a database connection. This is controlled by the `-C` option. The connection 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 backward-compatible mode; it is also the mode that must be used when you don't pass `QDB_CONN_DFLT_SHARE` flag to *qdb_connect()* (p. 131).

`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-negligible 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 more. 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 backward 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 it shouldn't set the `QDB_CONN_DFLT_SHARE` flag).

Shared caching

The default startup mode for qdb is with both shared caching and exclusive mode enabled. You can change this as follows:

- To disable shared caching, use the -D command-line option.
- When shared caching is enabled, qdb reserves exclusive privileges for writing to the database. To allow other applications to use the database files, use the -A option.

The qdb utility exits immediately if it is started with shared cache disabled but exclusive mode enabled. For example:

```
# qdb -v -D -otempstore=/fs/tmpfs -Rset

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

Advantages of shared caching

Shared caching both improves performance times and reduces the total amount of memory required for multiple database connections by having multiple connections 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 caching, these 40 connections could share a common memory cache of, say, 25 MB (or another size determined by your environment and performance requirements). Furthermore, there is no duplication in memory, so you may be able to hold all or most of the database, greatly reducing the need for disk I/O.

Database recovery

The `-R` option 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 at startup. 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's 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 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 causes 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 `AutoAttach` option in the configuration object). The *backup set* contains the set master and all attached databases that have `BackupAttached` enabled. Note that the set master can be backed up incrementally and still belong to the set.

We recommend the following actions to back up and restore your databases as a coherent set:

- For the set master database, in the database configuration object:
  - In the `AutoAttach` option, list all the databases you want to attach. For example:

```
AutoAttach::mp3_tunes_1,mp3_tunes2
```

- In the `BackupAttached` option, list all dependant databases. For example:

  ```
  BackupAttached::mp3_tunes_1,mp3_tunes_2
  ```

- Use the `-R set` option when starting `qdb`.
- When doing backups, call *qdb_backup()* (p. 108) on the set master 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 that applies to each *qdb_connect()* (p. 131) connection. You can privately modify this setting with *qdb_setbusytimeout()* (p. 165).
- The `-T` option sets the global internal timeout that applies to database connections made without a client context. Examples include connections for verifying existing databases, constructing new databases at startup, and auto-attaching databases.

A value of `block` is equivalent to an infinite timeout period, and `nonblock` is equivalent to a timeout period of 0.

Handling corrupt databases

The `-X` option lets you provide the `qdb` utility with a program or script to run when it encounters a corrupt database. If the script appears to run correctly, the database server will continue to run; the recovery script is responsible for stopping and restarting the `qdb` service if necessary.

The following is a sample `qdb` startup command with the `-X` option:

```
# qdb -X /usr/bin/recover_db.sh
```

Below is a sample script that can be launched by `qdb` when it encounters a corrupt database. You can copy the code and save it as `recover_db.sh` to use in the above command.

```
#!/bin/ksh
# Corrupt database recovery script

# Set up some variables.
# Database (DB) name comes from argv[1], or $1.
# There is no way to automatically get the PPS object name from the
# DB name so the database PPS object name is made up of a path
# set by the system integrator and the DB name.

QDBPPSPATH=/pps/qnx/qdb/

DBNAME=$1
DBPPSCOBJ=${QDBPPSPATH}/config/${DBNAME}
DBPPSSOBJ=${QDBPPSPATH}/status/${DBNAME}
DBFILENAME=$(grep "^Filename\:\:" "${DBPPSCOBJ}" | cut -f3- -d \:)
BACKUPDIRS_STR=$(grep "^BackupDir\:\:" "${DBPPSCOBJ}" | cut -f3- -d \:)
IFS=","
set -A BACKUPDIRS_ARR -- ${BACKUPDIRS_STR}
PPSCOBJCONTENT=$(cat "${DBPPSCOBJ}" | grep -v "^@${DBNAME}\$")

# Delete the PPS object, then wait for the status object to go.
# The script waits for 2.5 seconds but continues even if the
# status object is still there; the script logs an error in this case.
DBPPSSOBJ_DELETED=0
rm ${DBPPSCOBJ}
for i in 1 2 3 4 5; do
  if [ ! -e ${DBPPSSOBJ} ]; then
    DBPPSSOBJ_DELETED=1
    break;
  fi
  sleep 0.5
done
if [ "${DBPPSSOBJ_DELETED}" -eq "0" ]; then
  echo "Status object \"${DBPPSSOBJ}\" still exists; continuing anyway."
fi
```

```
# Delete the database file and any backups. The backup deletion should
# be customized by the integrator to delete only the appropriate backup
# filenames (for example DBNAME.gz).
# NOTE: This example doesn't consider auto-attached databases.
#       If you are using auto-attached databases or any other advanced
#       configuration, you may need to do some special handling.
rm ${DBFILENAME}
for dir in ${BACKUPDIRS_ARR[@]}; do
  rm -f ${dir}/${DBNAME} ${dir}/${DBNAME}.*
done

# Re-create the PPS configuration object.
echo "${PPSCOBJCONTENT}" > ${DBPPSCOBJ}

# EOF
```

- To kill `qdb` without killing the script, send `SIGTERM` (the default for `slay`). This way, `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.

Chapter 3
QDB Client

Execute SQL statements on QDB databases

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 object (honoring the value of the `VacuumAttached`, `BackupAttached`, and `SizeAttached` parameters). This gives backward-compatible behavior.
- `all` — always act on any attached databases, regardless of configuration object settings.
- `none` — act only on the connected database itself, never on any attached databases.

`-B`

Perform a backup (the equivalent of calling *qdb_backup()* (p. 108)). The scope of this operation is determined by the configuration object 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 (|); this is the default setting

- `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 of calling *qdb_getdbsize()* (p. 137)) for the database specified by `-d` or ***QDBC_DBNAME***. The scope of this operation is determined by the database configuration object or the `-a` option, if specified.

`-t` ***timeout***

Set the database connection timeout, in milliseconds.

`-v[v...]`

Increase verbosity. You can set up to seven `-v` options.

`-V`

Perform a vacuum operation (the equivalent of calling *qdb_vacuum()* (p. 183)). The scope of this operation is determined by the configuration object for the database specified by `-d` or ***QDBC_DBNAME***, or by the `-a` option, if specified.

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 can't 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 as input to qdbc, for example:

```
qdbc < sql.txt
```

If you don't provide SQL code on the qdbc command-line, 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 must end in semicolons.

Backing up and restoring databases

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

- calling *qdb_backup()* (p. 108) from a client application
- passing the `-b` option to *qdbc* (p. 41).
- using the resource manager interface as follows:

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

These methods are affected by options in the *database configuration object* (p. 17).

To restore a database, start `qdb` with the `-R` option set to `auto`. For more information about this option, see the "*Database Recovery* (p. 36)" section in the QDB Command Line chapter.

You can cancel a database backup in client code by calling *qdb_bkcancel()* (p. 114). You can also cancel a backup operation using the QDB resource manager interface:

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

Chapter 4
QDB Example

QDB provides methods for client applications to query a database and step through the query results. The sample program shown here provides a walkthrough of the sequence of QDB method calls necessary to connect to, query, and disconnect from a QDB database.

Your client application should perform these general steps:

1. Connect to a database by calling *qdb_connect()* (p. 131).
2. Query the database and examine the result:
   a. Execute an SQL statement on the database by calling *qdb_statement()* (p. 171).
   b. Get the result set for the statement by calling *qdb_getresult()* (p. 145).
   c. Read a data cell from the result set by calling *qdb_cell()* (p. 116) and then use the data as needed.
   d. Free the result set by calling *qdb_freeresult()* (p. 136).
   e. Repeat these steps to execute statements and use their results as many times as required.
3. Close the database connection with *qdb_disconnect()* (p. 135).

Connecting to a database

Connecting to a database requires that you know its name and use the name to obtain a handle from the QDB client library that your application links with.

The name is used in the device path provided to the connection function:

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

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

Using asynchronous mode

By default, QDB completes execution of SQL statements against a database before returning from *qdb_statement()* (p. 171). However, you can connect to QDB using asynchronous mode by setting the `QDB_CONN_STMT_ASYNC` in *flags* in the call to *qdb_connect()* (p. 131).

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()* (p. 145) 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 (e.g., `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, because you are calling back in anyway for the rows/results. If you are primarily doing database maintenance (i.e., `INSERT`, `UPDATE`, and `DELETE` statements), then you probably want synchronous statement execution so you can use just one API call.

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* (p. 187) chapter. You must, of course, connect to the database before attempting to execute statements against it. See "*Connecting to the database* (p. 46)".

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()*. For example, 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 interpreted as just `'O'`, because the second single quotation mark signals the end of the string, and the remaining characters produce an error. To correctly run the query, escape the single quotation mark in the middle of the string, as follows:

```
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 but 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 check whether you have data by examining the return value of *qdb_statement()*.

You can still call *qdb_getresult()* for statements with 0 rows in the result set. In fact, it may be the only way to retrieve the result. If the connection was opened with the QDB_CONN_STMT_ASYNC flag set, then *qdb_statement()* returns before the statement has been completed (see "*Using asynchronous mode* (p. 46)"). With complex statements, this may mean a delayed error.

To help debug your application, you can print the fetched result to *stdout()* to visualize your data, using this call:

```
qdb_printmsg(stdout, result, QDB_FORMAT_SIMPLE);
```

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

```
qdb_result_t *result;
// requires a statement previously run
result = qdb_getresult(dbhandle);
if (result == NULL) {
   char *errmsg;
   errmsg = qdb_geterrmsg(dbhandle);
   fprintf(stderr, "Error getting result: %s\n", errmsg);
}
```

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()* (p. 136), 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() (p. 130)	Returns the number of columns
qdb_rows() (p. 164)	Returns the number of rows. An empty result will return 0.
qdb_cell_type() (p. 120)	Returns the type of data in a cell (`QDB_INTEGER`, `QDB_REAL`, `QDB_TEXT`, `QDB_BLOB`, `QDB_NULL`).
qdb_column_name() (p. 128)	Returns the column name from the database schema
qdb_cell() (p. 116)	Returns the cell data as a `void` pointer that can be cast to the correct type
qdb_column_index() (p. 126)	Gets the column number that matches the passed in name
qdb_cell_length() (p. 118)	Returns the length of a cell's data
qdb_printmsg() (p. 158)	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:

```
int rc;
rc = qdb_disconnect(dbhandle);
if (rc == -1) {
     fprintf(stderr, "Disconnect failed: %s\n", strerror(errno));
}
```

Sample program

```
#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 this schema:
 *    CREATE TABLE customers(
 *       customerid INTEGER PRIMARY KEY AUTOINCREMENT,
 *       firstname  TEXT,
 *       lastname   TEXT
 *     );
 */
int main(int argc, char **argv) {
   int rc;
   qdb_hdl_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 into full names.
   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;
      }

      // Get the result
      res = qdb_getresult(hdl);
      if (res == NULL) {
         errmsg = qdb_geterrmsg(hdl);
         fprintf(stderr, "Error getting result: %s\n",
                 errmsg);
         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
      rc = qdb_freeresult(res);
      if (rc == -1) {
         fprintf(stderr, "Error freeing SQL statement results: %s\n",
                 strerror(errno));
         return EXIT_FAILURE;
      }

      // Disconnect from the server
      rc = qdb_disconnect(hdl);
      if (rc == -1) {
         fprintf(stderr, "Error disconnecting from database: %s\n",
                 strerror(errno));
         return EXIT_FAILURE;
      }

      return EXIT_SUCCESS;
   }
```

Chapter 5
Datatypes in QDB

QDB databases, which are stored in SQLite files, allow any value stored in any database field to have one of five standard storage classes. The SQLite design doesn't enforce data typing on database columns but instead maps each input value to a storage class based on the column's type affinity (preference).

QDB uses these storage classes to format data and to apply comparison or mathematical operators when evaluating queries and ordering and grouping the results.

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 Binary Large OBject, 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 under "*Other Affinity Modes* (p. 61)" as strict affinity mode.

All values supplied to QDB, whether as literals embedded in SQL statements or values bound to precompiled 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 enclosed by single or double quotes in SQL statements are assigned the storage class `TEXT`
- literals specified as unquoted numbers with no decimal point or exponent are assigned the `INTEGER` storage class
- literals specified as unquoted numbers with a decimal point or exponent are assigned the `REAL` storage class
- `NULL` values are assigned the `NULL` storage class
- literals specified using the `X'ABCD'` notation are assigned the `BLOB` storage class

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's 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's stored. If the conversion is successful, then the value is stored using the `INTEGER` or `REAL` storage class. If the conversion can't be performed, the value is stored using the `TEXT` storage class. No attempt is made to convert `NULL` or BLOB storage classes.

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` doesn't prefer one storage class over another. It makes no attempt to coerce data before it's 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 following comparison operators:

- `=`, `<`, `<=`, `>`, `>=` and `!=`, for arithmetic comparisons
- `IN`, an operation to test for set membership
- `IS`, which works similar to `=` except when at least one operand is `NULL`. If both operands are `NULL`, `IS` evaluates to 1 (true). If one operand is `NULL` but the other isn't, the operator evaluates to 0 (false).
- `IS NOT`, which works similar to `!=` except when at least one operand is `NULL`. If both operands are `NULL`, `IS NOT` evaluates to 0 (false). If one operand is `NULL` but the other isn't, the operator evaluates to 1 (true).
- `BETWEEN`, the ternary comparison operator, which tests if a value lies within a range

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 doesn't, 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`.

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 SELECT statements

When query results 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 in collating sequence 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. No affinity is applied to comparison operands for the implicit comparisons associated with `UNION`, `INTERSECT`, or `EXCEPT`—the values are compared as is. For more details, see *Compound SELECT statements* (p. 230).

Other affinity modes

The earlier sections describe the operation of the database engine in normal affinity mode. QDB features two other affinity modes:

Strict-affinity mode

If a conversion between storage classes is required, the database engine returns an error and the current statement is rolled back.

No-affinity mode

No conversions between storage classes are 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 text encoding. QDB lets you supply arbitrary comparison functions, known as *user-defined collation sequences*, to use instead of *memcmp()*. See *Writing User-Defined Functions* (p. 95) for more information.

QDB features three built-in collation sequences:

BINARY

Compare string data using *memcmp()*, regardless of text encoding. This is the default comparison method.

NOCASE

The same as `BINARY`, except that the 26 uppercase characters used by the English language are converted to their lowercase equivalents before the comparison is performed.

RTRIM

The same as `BINARY`, except that trailing spaces are ignored.

The last two sequences are intended for testing purposes.

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 preferred collation type by specifying a `COLLATE` clause as part of the *CREATE TABLE* (p. 208) column definition.

Whenever QDB compares two text values, it uses a collation sequence to determine the results of the comparison according to the following rules:

- If either operand has a collation sequence explicitly named by the `COLLATE` clause in the SQL expression, the named collation type takes precedence over that in the column definition. If both operands have named collation sequences, then the collation type for the left operand determines the collation sequence used.
- 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's part of a *SELECT* (p. 228) statement may be assigned a collation sequence to be used for the sort operation explicitly. In this case, the collation sequence named by the `COLLATE` clause is 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(
    x INTEGER PRIMARY KEY,
    a,                 /* collating sequence BINARY */
    b COLLATE BINARY,  /* collating sequence BINARY */
    c COLLATE RTRIM,   /* collating sequence RTRIM  */
    d COLLATE NOCASE   /* collating sequence NOCASE */
);

/* Text comparison a=b is performed using the BINARY sequence. */
SELECT x FROM t1 WHERE a = b ORDER BY x;
--result 1 2 3

/* Text comparison a=b is performed using the RTRIM sequence. */
SELECT x FROM t1 WHERE a = b COLLATE RTRIM ORDER BY x;
--result 1 2 3 4

/* Text comparison d=a is performed using the NOCASE sequence. */
SELECT x FROM t1 WHERE d = a ORDER BY x;
--result 1 2 3 4

/* Text comparison a=d is performed using the BINARY sequence. */
SELECT x FROM t1 WHERE a = d ORDER BY x;
--result 1 4

/* Text comparison 'abc'=c is performed using the RTRIM sequence. */
SELECT x FROM t1 WHERE 'abc' = c ORDER BY x;
--result 1 2 3

/* Text comparison c='abc' is performed using the RTRIM sequence. */
SELECT x FROM t1 WHERE c = 'abc' ORDER BY x;
--result 1 2 3

/* Grouping is performed using the NOCASE sequence (Values
```

```
** 'abc', 'ABC', and 'Abc' are placed in the same group). */
SELECT count(*) FROM t1 GROUP BY d ORDER BY 1;
--result 4

/* Grouping is performed using the BINARY sequence.  'abc' and
** 'ABC' and 'Abc' form different groups */
SELECT count(*) FROM t1 GROUP BY (d || '') ORDER BY 1;
--result 1 1 2

/* Sorting or column c is performed using the RTRIM sequence. */
SELECT x FROM t1 ORDER BY c, x;
--result 4 1 2 3

/* Sorting of (c||'') is performed using the BINARY sequence. */
SELECT x FROM t1 ORDER BY (c||''), x;
--result 4 2 3 1

/* Sorting of column c is performed using the NOCASE sequence. */
SELECT x FROM t1 ORDER BY c COLLATE NOCASE, x;
--result 2 4 3 1
```

Chapter 6
QDB Virtual Machine Opcodes

QDB uses a virtual machine to execute SQL statements. You can configure QDB to list the opcodes of the programs generated to execute the SQL statements and you can trace program execution.

Before you can easily interpret virtual machine programs, you must understand the instruction format and virtual machine features.

Instruction format and execution

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. In any operation that might cause a jump, *P2* is always the jump destination. *P3* may point to a function, a data structure, or a string, or it may be `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.

Virtual machine features

The virtual machine 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 so on. 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 more than 120 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 Binary Large OBject (BLOB) that is *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.

This opcode ignores *P2*.

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* (p. 73)

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.

This opcode ignores *P1* and *P2*.

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.

This opcode ignores *P1*.

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`* (p. 73)

This opcode ignores *P1*.

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 the auto-vacuum mode 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 (i.e., 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 auto-vacuum is disabled, then a zero is pushed onto the stack.

See also: *Clear* (p. 71)

Distinct

Use the top of the stack as a record created using MakeRecord. *P1* is a cursor on a table that declared 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 doesn't 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* (p. 76), *NotFound* (p. 86), *IsUnique* (p. 81), *NotExists* (p. 86)

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`* (p. 88) 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 make 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`, 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, then push nothing but instead jump to *P2*.

This opcode ignores *P1*.

`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 it's a `NULL` or a string that doesn't 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 doesn't 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`* (p. 74), *`NotFound`* (p. 86), *`IsUnique`* (p. 81), *`NotExists`* (p. 86)

`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`* (p. 70) and *`AggFinal`* (p. 70)

`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`* (p. 75) 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.

This opcode ignores *P1*.

`Goto`

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

This opcode ignores *P1*.

`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`* (p. 75) opcode for additional information.

Halt

Exit immediately. All open cursors and FIFOs 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 Binary Large OBject (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*.

This opcode ignores *P1* and *P2*.

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 row ID. Compare the top of stack against the index that *P1* is currently pointing to. Ignore the row ID 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 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 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* (p. 89).

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.

This opcode ignores *P1* and *P2*.

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 an `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. *P1* is ignored.

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`* (p. 74), *`NotFound`* (p. 86), *`NotExists`* (p. 86), *`Found`* (p. 76)

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* (p. 75) 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* (p. 75) 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 like `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* (p. 76), *NotFound* (p. 86), *Distinct* (p. 74), *MoveLt* (p. 84), *MoveGt* (p. 84), *MoveLe* (p. 84).

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* (p. 76), *NotFound* (p. 86), *Distinct* (p. 74), *MoveLt* (p. 84), *MoveGe* (p. 84), *MoveLe* (p. 84).

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* (p. 76), *NotFound* (p. 86), *Distinct* (p. 74), *MoveGt* (p. 84), *MoveGe* (p. 84), *MoveLt* (p. 84).

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* (p. 76), *NotFound* (p. 86), *Distinct* (p. 74), *MoveGt* (p. 84), *MoveGe* (p. 84), *MoveLe* (p. 84).

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 without 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* (p. 75) 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 a number not already being 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* (p. 88)

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 doesn't 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`* (p. 74), *`Found`* (p. 76), *`NotFound`* (p. 86), *`IsUnique`* (p. 81).

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 *P2*. If an entry does exist, 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` doesn't pop the key from the stack.

See also: *`Distinct`* (p. 74), *`Found`* (p. 76), *`NotExists`* (p. 86), *`IsUnique`* (p. 81).

NotNull

Jump to *P2* if the top *P1* values on the stack are all not `NULL`. If *P1* is greater than 0, the stack is popped *P1* times. If *P1* is less than or equal to 0, the stack is left 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 element on 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 an 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* (p. 87).

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* (p. 87).

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* (p. 75) 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.

This opcode ignores *P1* and *P2*.

Remainder

Pop the top two elements from the stack, divide the first (the element that was on top of the stack) by 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 that's 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 that's 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 opcode.

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, and push 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 is similar to `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 that's 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.

This opcode ignores *P1* and *P2*.

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.

This opcode ignores *P1* and *P2*.

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()*; 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 a number using the equivalent of *atoi()* or *atof()*; 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 a character sequence 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 7
Writing User-Defined Functions

QDB allows you to write your own functions for manipulating data. The QDB library includes several data structures and functions useful for defining custom functionality, while the SQLite API provides methods your custom code can call to access arguments or set results.

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 (see "*Database configuration objects* (p. 17)").

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 database configuration object for each required function.

User scalar/aggregate functions

User scalar/aggregate functions are specified in the configuration object with the `Function::`*tag@library.so* option, where *tag* is the name of the `struct qdb_function` entry describing the function, and *library.so* is the name of a DLL containing your code (this can be an absolute path or a relative path within the library search path). 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 fourth 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 object 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 null-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 to user data that is passed to your function each time it's invoked. The function can gain access to this pointer by using the *sqlite_user_data()* function.

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* (p. 228) statement. You can define your own routine and tell QDB to use it by providing the COLLATE keyword in the ORDER BY clause.

These routines are specified in the database configuration object with the `Collation::`*tag@library.so* option, where *tag* is the name of the `qdb_collation` entry describing the collation, and *library.so* is the name of a DLL object containing your code (this can be an absolute path or a relative path within the library search path).

The code 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
};
```

In this case, the tag value for the structure is `ctag`, the collation name as visible to SQL is `nosort`, and the C function that implements your collation routine is called *mysort()* and has a NULL value for its *arg* argument. For more information on defining SQLite collation sequences, refer to the SQLite docs on *sqlite3_create_collation()*. Full details on the meaning of each `qdb_collation` field are given in the subsection that follows.

The `ctag` tag was used to clarify the example. You would probably use the name `nosort` here so the tag name matched the SQL name.

In this release, you can define only one collation per database. This collation must be loaded in the configuration object, by specifying a `Collation::` entry that lists the name of the `struct qdb_collation` describing the collation, followed by the library filename.

The `qdb_collation` struct

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 null-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 either the comparison or setup function, each time it's invoked. The function can gain access to this pointer by using the *sqlite_user_data()* function.

compare

A pointer to your comparison function.

setup

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

The *setup* function

The *setup* function takes this form:

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

The function parameters are:

`void` ****arg***

The context pointer. This is the same as the *arg* to the *compare* function, and is copied from the *arg* field in the `qdb_collation` structure.

`const void *`*`data`*, **`int` *`nbytes`*

The data used to configure the sort. When the routine is invoked at startup, these values are NULL and 0. At runtime, they refer to the data provided to the *qdb_collation()* (p. 122) function. QDB doesn't 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()* and is displayed on failure (actually, at startup, QDB will fail it; at runtime, *qdb_collation()* will fail and this string will be available through *qdb_geterrmsg()*).

The function returns either a POSIX *errno* value or `EOK` (if it succeeds).

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

If a collation has no dynamic configuration, it can specify `NULL` for the *setup* entry. Note that this entry can't be changed at runtime.

Collation algorithm example

Here is an example of a collation algorithm that uses the data pointer *arg* to compare two items. The DLL would export the following entries:

```
my_sort_ctx_t my_sort_ctx = { ... };

static int my_compare_func(void *arg,
                           int l1, const void *s1,
                           int l2, const void *s2)
{
    /* Custom code for comparing two items */
}

static int my_setup_func(void *arg, const void *data, char **errmsg)
{
    /* Custom code for initializing collation based on setup data
       in my_sort_ctx (which is referred to by arg) */
    return errno;
}

struct qdb_collation my_sort = {
    .name="my_sort", .encoding=SQLITE_UTF8, .arg=&my_sort_ctx,
    .compare=my_compare_func, .setup=my_setup_func };
```

Here, the collation routine is described by a structure named `my_sort` and calls the *my_compare_func()* function to do the actual sorting. This function is given data that is stored in the *my_sort_ctx* structure and passed in the *arg* argument. The same data is also passed in to the setup function, *my_setup_func()*.

You would install this routine to QDB in the configuration object with this setting:

```
Collation::my_sort@/usr/lib/libqdb_mysort.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. At the time of this writing, the latest version of SQLite is listed on the SQLite homepage, and the official documentation is updated for major releases (e.g., going from version 3.6.X to 3.7.X). You can find out the library version QDB is using by looking in the `sloginfo` log just after you start the QDB service, or by calling the *sqlite_version()* function.

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 doesn't 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`

Internal Type	Requested Type	Conversion
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.

Chapter 8
QDB API Reference

QDB provides a comprehensive API for managing databases and accessing data. Using the API, your client application can:

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

qdb_backup()

Start a database backup

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 object, honoring the value of the `VacuumAttached`, `BackupAttached`, and `SizeAttached` parameters. This gives backward-compatible behavior.

QDB_ATTACH_ALWAYS

Always act on any attached databases, regardless of configuration object settings.

QDB_ATTACH_NEVER

Act only on the connected database itself, never on any attached databases.

Library:

`qdb`

Description:

This function performs a backup on the connected database *db* and optionally any attached databases, depending on the *scope* argument. Backups are controlled in the

configuration object, with the `BackupDir` and `Compression` options. For more information about these options, see the *Database configuration objects* (p. 17) section.

A client can cancel a backup operation by calling *qdb_bkcancel()* (p. 114). If a backup is canceled (either by a client or through 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

qdb_binding_t

Information to bind data to an SQL statement variable

Synopsis:

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

typedef struct {
    int index;
    int type;
    int len;
    const void *data;
    unsigned long long intcopy;
} qdb_binding_t;
```

Data:

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.

type

The data type. Can be one of: `QDB_NULL`, `QDB_BLOB`, `QDB_TEXT`, `QDB_INTEGER`, or `QDB_REAL`.

len

The length of the *data* argument. This number excludes the null-terminator for `QDB_TEXT`, and is set to `sizeof(double)` for `QDB_REAL` and `sizeof(int64_t)` for `QDB_INTEGER`.

data

The data to bind. The data must be the same size as what's set in *len*, because *len* bytes are read during the binding. For example, for the `QDB_REAL` type, whose length is `sizeof(double)`, if you assign a `float` to *data*, only half the bytes will be read, resulting in unknown behavior. To correct this, use a `double` for *data*.

intcopy

A 64-bit field for holding a copy of integer values. This field may or may not be used.

Library:

qdb

Description:

The `qdb_binding_t` structure stores the information needed to bind a data value to a variable parameter in a prepared SQL statement. This information includes fields describing the data type and length as well as a reference to the data value.

You should always initialize an instance of `qdb_binding_t` with one of the convenience macros. It is not recommended to manually set the fields in the binding structure one-by-one.

The variables in the macro prototypes have these meanings:

- *bind* is the address of the `qdb_binding_t` structure
- *i* is the *index* member (which references the statement variable you're binding the data to)
- *t* is the *type* member
- *l* is the *len* member
- *d* is the *data* member

The following macros can be used to define a single data-binding structure:

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

Bind in any specified data type.

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

Bind in a 64-bit integer; subsequent changes to the same integer variable alter the bound data.

***QDB_SETBIND_NULL*(*bind*, *i*)**

Bind in `NULL`.

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

Bind in text.

***QDB_SETBIND_INTCOPY*(*bind*, *i*, *d*)**

Bind in a copy of an integer; subsequent changes to the same integer variable do *not* alter the bound data. Also, with this macro only, you can bind data of varying sizes because the *intcopy* field is used to store a copy of the integer value, and this field can accept assignments from narrower variable types; for example, 32- or even 16-bit integers.

***QDB_SETBIND_BLOB*(*bind*, *i*, *d*)**

Bind in a blob.

***QDB_SETBIND_REAL*(*bind*, *i*, *d*)**

Bind in a real number.

The single structure-based macros are useful only when your statement has one variable. In this case, you can just pass in the address of the structure when calling *qdb_stmt_exec()* (p. 176) (and set *binding_count* to 1 to indicate there's only one binding item). If you need to define multiple variables, you must declare an array of `qdb_binding_t` structures and then fill in the individual array entries. You can do this by using the array-based data-binding macros.

These macros have names of the form ***QDB_SETARRAYBIND_*****. There is a matching array macro for each single-structure macro, with identical parameters. For example, ***QDB_SETARRAYBIND*** accepts the same five parameters as ***QDB_SETBIND***, and binds any specified data type. The only difference is that for array macros, the *i* argument acts as an index not only for the parameter variable being bound but also for the array entry being written.

Suppose you're using an array of binding structures to assign data to multiple variable parameters in an SQL statement. To bind the first parameter, call one of the macro arrays with the index *i* set to 1 to fill in the first structure in the array. For the second parameter, use an index of 2, and so on.

After you've defined the data for the statement variables, you can execute the SQL statement by calling *qdb_stmt_exec()*, passing in the statement ID and a reference to the array of `qdb_binding_t` structures. Any variables that aren't defined are interpreted as `NULL`.

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

- the limit set by the database
- $x = 2^{31} - (binding\_count + 1) \times 12$, where x is the data limit, in bytes

---

Examples:

The following code sample shows the difference between the `QDB_SETBIND_INT` and `QDB_SETBIND_INTCOPY` macros:

```
qdb_binding_t qbind[2];
int64_t i = 17;
QDB_SETARRAYBIND_INT(qbind, 1, i);
```

```
QDB_SETARRAYBIND_INTCOPY(qbind, 2, i);

int stmtid = qdb_stmt_init(
           "INSERT INTO testtable (val1, val2) VALUES (?1, ?2);");

for (i=0; i<10; i++) {
    qdb_stmt_exec(stmtid, qbind, 2);
}
```

Both bound parameters refer to the local variable `i` initially; however, the loop uses this variable as the index, causing a different value to be inserted in the first column each time *qdb_stmt_exec()* runs. The first parameter changes because *QDB_SETBIND_INT* only stores the variable address (and not the value itself) in the binding structure, so modifying the variable modifies the bound value as well. The second parameter remains unchanged because *QDB_SETBIND_INTCOPY* makes a copy of the passed-in value. So, the resulting table values are:

```
val1 | val2
===========
  0  |  17
  1  |  17
  2  |  17
...
  9  |  17
```

See *qdb_stmt_init()* (p. 180) for an example on how to compile, execute, and free an SQL statement.

qdb_bkcancel()

Cancel a database backup

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

nactive

A pointer to a location for storing the number of aborted backup operations. You can use this to see if a backup was interrupted and needs to be rescheduled. If don't need this information, set this parameter to `NULL`.

Library:

`qdb`

Description:

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

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_cell()

Return a cell's data

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

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

For QDB_TEXT, this function doesn't count the terminating null character.

Returns:

>=0

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

qdb_cell_type()

Return a cell's datatype

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, where the first row is 0.

col

The column number of the data cell, where the first column is 0.

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>`:

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()* (p. 118). The text type QDB_TEXT (`char *`) is always null-terminated.

Returns:

>=0

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

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 configures user-defined collation sequences attached to the database. These collation sequences are listed under the `Collation` option in the database configuration object. 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 can consider 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

qdb_column_decltype()

Retrieve the declared type for a column in a result set

Synopsis:

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

const char *qdb_column_decltype( qdb_result_t *result,
                                 int col );
```

Arguments:

result

A pointer to a result structure.

col

The index of the column type to return, where 0 (zero) indicates the first column.

Library:

qdb

Description:

The function *qdb_column_decltype()* returns the declared type of the specified column of a result set. The result set is specified by the *result* argument. The returned string is valid until *qdb_freeresult()* is called.

If the specified column is the result of an expression or subquery, an empty string is returned.

To use *qdb_column_decltype()*, you must set the option `QDB_OPTION_COLUMN_DECLTYPES` by calling *qdb_setoption()*. By default, this option is off.

Returns:

A pointer

A pointer to the specified column's declared type.

NULL

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_column_index()

Return a column's index

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:

>=0

The index of the specified column.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

qdb_column_name()

Return a column's name

Synopsis:

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

const 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, where the leftmost column is 0.

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 to the specified column's name, or NULL if an error occurred (*errno* is set). The string containing the column name is part of the results set, so the string memory is freed (along with the rest of the results set memory) by *qdb_freeresult()*. If you want to keep the column name longer, you must create a copy of the string (and manage that copy's memory).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

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 to 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()* (p. 164) to determine if the results are empty.

Returns:

>=0

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

qdb_connect()

Connect to a database

Synopsis:

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

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

Arguments:

dbname

The database device name (e.g., `/dev/qdb/customerdb`).

flags

Flags for determining parameters of the connection. This argument can be 0 (no flags) or a combination of:

QDB_CONN_DFLT_SHARE

Use the default database connection share mode (as given in the `-C` command-line option to QDB). Without this flag, a private connection is forced.

QDB_CONN_NONBLOCKING

If this flag is set, *qdb_statement()* (p. 171) 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()* (p. 165)) if the database is locked, before returning with a failure result.

Setting this flag also makes subsequent calls to *qdb_connect()* nonblocking (as if the `-T` command-line option were 0).

QDB_CONN_STMT_ASYNC

Execute statements asynchronously. In this mode, *qdb_statement()* (p. 171) may return before the statement has finished executing against the database (see "*Using asynchronous mode* (p. 46)").

Library:

qdb

Description:

This function connects to the database specified by *dbname* and sets parameters for the database connection. When successful, the function returns a pointer to a handle for the new 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 that they coordinate between themselves. Alternatively, each thread can call *qdb_connect()* and have its own connection.

You should terminate all connections by calling *qdb_disconnect()* (p. 135) when you're finished using them.

Returns:

A valid pointer to an opaque database connection (`qdb_hdl_t`)

Success.

`NULL`

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_data_source()

Extract the data source for a database

Synopsis:

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

int qdb_data_source( qdb_hdl_t *hdl,
                     char *buffer,
                     int buflen );
```

Arguments:

hdl

A pointer to the database handle.

buffer

A buffer to hold the resulting source path information. You can set this parameter to NULL to make the function return the required buffer size. The buffer is managed by the client application, not by QDB.

buflen

The length of *buffer*; this argument is relevant only when the buffer is not NULL.

Library:

qdb

Description:

This function provides a description of the source used to initialize the database. This source may be one of several paths, or a list of the schema creation and data population files, depending on the state of the specified database when the qdb utility is started and on how the database is 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, 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 neither corrupted nor a backup database, 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, the string will be the path to the restoring database from one of the `BackupDir` entries.

If you don't know the buffer size required to hold the data source string, call *qdb_data_source()* with *buffer* set to `NULL` and use the function's return value for the amount of memory to allocate for the buffer. Then, call *qdb_data_source()* again, passing in the address of the created buffer in *buffer*, to get the data source.

Returns:

>=0

Depending on the arguments, either the required size of the buffer to store the data source or the length of the string stored in *buffer* (both size values include the null-terminator).

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_disconnect()

Disconnect from a database

Synopsis:

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

int qdb_disconnect( qdb_hdt_t *db );
```

Arguments:

db

A pointer to the handle of the database to disconnect from.

Library:

qdb

Description:

This function terminates the connection with a database previously connected to with *qdb_connect()* (p. 131).

You should disconnect from all databases 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

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 to a result structure to free.

Library:

qdb

Description:

This function frees a result previously returned from *qdb_getresult()* (p. 145). You must call *qdb_freeresult()* when you're finished using a result set, to reduce your application's memory footprint.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_getdbsize()

Return the size of a database

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

scope

The scope of the operation. See the description of the *scope* argument in *qdb_backup()* (p. 108) for more information.

page_size

A pointer to a location for storing the size (in bytes) of a page in the database file.

total_pages

A pointer to a location for storing the number of pages in the database file.

free_pages

A pointer to a location for storing the number of pages not being used to store data.

Library:

qdb

Description:

This function provides information about the size of the database file associated with the database handle *db*. The database's size on the filesystem is *page_size* × *total_pages*.

If you vacuum the database, QDB deletes the free pages so the total pages goes down, free pages goes to 0, and the database file becomes smaller. For more information, see the *VACUUM* (p. 232) SQL command, *qdb_vacuum()* (p. 183) function, and the *Auto-vacuum section* (p. 221) of the PRAGMA command.

For an attached database to be included in the size calculation of the main database (i.e., the one with handle *db*), the attached database must be listed under the `SizeAttached` option in the configuration object for the main database.

Returns:

>=0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_geterrcode()

Return the last error code

Synopsis:

```
#include <qdb/qdb.h>
int qdb_geterrcode( qdb_hdl_t *db );
```

Arguments:

db

A pointer to the database handle.

Library:

qdb

Description:

This function returns the SQL error code for the most recent call to one of:

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

You typically call this function after one of the above functions fails.

Returns:

>0

The SQL error code from the last failed QDB operation.

0

There is no error, or the database handle is invalid.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_geterrmsg()

Return the last error string

Synopsis:

```
#include <qdb/qdb.h>
const char *qdb_geterrmsg( qdb_hdl_t *db );
```

Arguments:

db

A pointer to the database handle.

Library:

qdb

Description:

This function returns a string containing an error message for the most recent call to one of:

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

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 an unmodifiable string containing the error message from the last QDB operation, or an empty string if there is no error. If the database handle is invalid, the function returns a pointer to a string containing the POSIX "not connected" *errno* message.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_getoption()

Return the value for a database session option

Synopsis:

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

int qdb_getoption( qdb_hdl_t *db,
                   int option );
```

Arguments:

db

A pointer to the database handle.

option

The option you want to query. See *qdb_setoption()* (p. 167) for a list of database options.

Library:

qdb

Description:

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

Returns:

1

The value of the option is 1 (on).

0

The value of the option is 0 (off).

-1

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

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_getresult()

Return the result of an SQL statement

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

Library:

qdb

Description:

After running a SELECT statement on the database, you can retrieve its result using *qdb_getresult()*. All rows that match the query are returned in one result set, represented as a `qdb_result_t` structure, which is an opaque data type. You can get further information about the result using these functions:

- *qdb_cell()* (p. 116)
- *qdb_cell_length()* (p. 118)
- *qdb_column_index()* (p. 126)
- *qdb_column_name()* (p. 128)
- *qdb_columns()* (p. 130)
- *qdb_printmsg()* (p. 158)
- *qdb_rows()* (p. 164)

When you're finished using the result, you must free it by calling *qdb_freeresult()* (p. 136) .

Returns:

A pointer to the query result

Success.

NULL

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_gettransstate()

Return the transaction state for a QDB connection

Synopsis:

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

int qdb_gettransstate( qdb_hdl_t *db );
```

Arguments:

db

A pointer to the database handle.

Library:

qdb

Description:

This function returns the transaction state for the specified QDB connection. If an SQL transaction is in progress over the connection, the function returns 1. If no SQL transaction is happening, 0 is returned. If there's an SQL error, -1 is returned (you can use *qdb_geterrmsg()* (p. 141) to get the error string).

You can use this function to determine how to clean up after an SQL error; for example, if you execute several commands in a transaction and need to determine which statement is causing the error.

Returns:

1

An SQL transaction is in progress.

0

No 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

qdb_interrupt()

Interrupt long-running queries

Synopsis:

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

int qdb_interrupt( qdb_hdl_t *db );
```

Arguments:

db

A pointer to the database handle.

Library:

qdb

Description:

This function interrupts all currently running queries that are using the specified connection handle (*db*) and rolls back any uncommitted transactions. If *qdb_interrupt()* is called on a connection when no query is executing, it doesn't do anything.

If the QDB connection is synchronous, you must call this function in a thread other than the one that initiated the query. In *asynchronous mode* (p. 46), this function may be called from the same thread.

If a query is nearly finished when you call this function, the query might not get interrupted and instead finish executing.

The *qdb_interrupt()* call is active until all currently running queries associated with *db* either finish executing or are successfully interrupted. For queries started after the *qdb_interrupt()* call was issued but before the count of active queries reaches 0, they're interrupted as if they had been running prior to the call. Any queries started after the active query count reaches 0 aren't affected by this function.

You must not disconnect from the database while this call is still active; otherwise, the database may end up in an inconsistent state.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_last_insert_rowid()

Return the primary key of the last inserted row

Synopsis:

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

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

Arguments:

db

A pointer to the database handle. You can pass `NULL` for this argument if you provide a value for *result* and you set the `QDB_OPTION_LAST_INSERT_ROWID` option when calling *qdb_setoption()* (p. 167). Note that this option is enabled by default.

result

A pointer to the result set you want to query. If you pass in `NULL`, the function queries the server connection *db* for the last *qdb_statement()* (p. 171) call.

Library:

`qdb`

Description:

This function returns the row ID of the last `INSERT` statement. Each entry in a QDB table has a unique integer key called the row ID. This key 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 an alias for the *rowid*.

The *qdb_last_insert_rowid()* function first looks in *result* (if the `QDB_OPTION_LAST_INSERT_ROWID` option is enabled), returning the information for the statement that produced the result. If *result* is `NULL` or `QDB_OPTION_LAST_INSERT_ROWID` is disabled, the function queries the database handle *db* 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, there was an error with the request.

If an `INSERT` occurs within a trigger, then the row ID of the inserted row is returned by this function as long as the trigger is running. When the trigger terminates, the function will return the last value inserted before the trigger fired.

Returns:

>0

The integer primary key of the last inserted row.

0

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

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_mprintf()

Print formatted output to a new string

Synopsis:

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

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

Arguments:

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 QNX Neutrino *C 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's no possibility of buffer overflow. The function implements some additional formatting options that are useful for constructing SQL statements.

The *qdb_statement()* (p. 171) function also allows you to format strings in the same way and doesn't require you to free the resulting string. However, *qdb_mprintf()* is 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. The *qdb_mprintf()* function adds these options:

%q

This option works like `%s`: it substitutes a null-terminated string from the argument list; however, `%q` also doubles each single-quote character (`'`). This is useful inside a string literal. By doubling each single-quote character, `%q` escapes that character and allows it to be inserted into the SQL statement.

%Q

This option works like %q except that it adds single quotes around the contents of the entire string. Or, if the parameter in the argument list is a NULL pointer, %Q substitutes the text NULL in place of the %Q option.

%z

This option works like %s except that after the source string (which is the argument referred to by %s) has been copied into the formatted string, the source string memory doesn't have to be freed.

Returns:

A pointer to an escaped string

Success.

NULL

An error occurred (*errno* is set).

Examples:

Suppose some string variable contains the following text:

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

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

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

Because the %q formatting option is used, the single-quote character in *zText* is escaped, and the generated SQL is:

```
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.

Suppose you're unsure if your text reference is NULL. You can use this reference as follows:

```
char *zSQL = qdb_mprintf("INSERT INTO table VALUES(%Q)", zText);
```

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

The `%z` option is handy for nested strings:

```
char id[] = "12345678";
char *nested = qdb_mprintf(
            "SELECT msid FROM mediastores WHERE id = %Q", id);
char *sql = qdb_mprintf(
            "DELETE FROM library WHERE msid = (%z);", nested);
qdb_exec(sql);
free(sql);
```

The *nested* string doesn't have to be freed after it gets copied into the formatted string and the SQL code within the formatted string is executed.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

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 representing the connection parameters you want to update.

bits

The bits corresponding to the parameters you want to enable. If a bit is in *mask* but not in *bits*, the parameter is unset (i.e., disabled).

Library:

qdb

Description:

This function queries or modifies the database connection parameters. You can set or unset the QDB_CONN_NONBLOCKING and QDB_CONN_STMT_ASYNC bits (see the *flags* argument for *qdb_connect()* (p. 131) for a description of these settings). You can't change the QDB_CONN_DFLT_SHARE bit.

The function returns the previous bitmask (i.e., parameter settings), so the parameter settings can be temporarily changed and restored.

Returns:

>=0

The previous value of the bitmask.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_printmsg()

Print data from a query result

Synopsis:

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

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

Arguments:

fp

A handle for a file to print the results to.

result

The query result you want to print.

format

The desired format of the results. Can be one of:

QDB_FORMAT_SIMPLE

Minimal formatting.

QDB_FORMAT_HTML

HTML formatting, suitable for viewing in a browser.

QDB_FORMAT_COLUMN

Column formatting, so the results appear under column names.

Library:

`qdb`

Description:

This function prints the results of a SELECT query to a file. You must specify a standard file stream in *file*, such as `stdout`.

Returns:

>=0

The number of rows in the results.

-1

An error occurred (*errno* is set).

Errors:

EINVAL

An invalid format was specified.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

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()* (p. 171).

Library:

qdb

Description:

This convenience function provides a single-interface alternative to calling *qdb_statement()* (p. 171) and *qdb_getresult()* (p. 145), 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

qdb_rowchanges()

Return the number of rows affected by an SQL statement

Synopsis:

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

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

Arguments:

db

A pointer to the database handle. You can pass in `NULL` for this argument if you provide a value for *result* and the `QDB_OPTION_ROW_CHANGES` option has been set by *qdb_setoption()* (p. 167) (it's set by default).

result

A pointer to the result set you want to query. If you pass in `NULL`, the function queries the result from the last *qdb_statement()* (p. 171) call on *db*.

Library:

`qdb`

Description:

This function returns the number of rows affected by an SQL statement. It first looks in *result* (if the `QDB_OPTION_ROW_CHANGES` option has been set by *qdb_setoption()* (p. 167)), returning the number of rows for the statement that produced the result. If *result* is `NULL` or `QDB_OPTION_ROW_CHANGES` is not set, the function queries the database handle *db* 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`, 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
Signal handler	No
Thread	Yes

qdb_rows()

Return the number of rows in a result set

Synopsis:

```
#include <qdb/qdb.h>
int qdb_rows( qdb_result_t *res );
```

Arguments:

res

A pointer to a result structure to check.

Library:

`qdb`

Description:

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

Returns:

>=0

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

qdb_setbusytimeout()

Set the busy timeout for a database connection

Synopsis:

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

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

Arguments:

db

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

timeout

The timeout period, in milliseconds. This value may also be:

QDB_TIMEOUT_NONBLOCK

The equivalent of setting a timeout of 0. Calls to *qdb_statement()* (p. 171) return immediately with failure if the database file is locked.

QDB_TIMEOUT_BLOCK

The equivalent of an infinite timeout period. Calls to *qdb_statement()* (p. 171) will wait indefinitely or until the database is unlocked, at which point the call will succeed.

Library:

qdb

Description:

This function sets the busy timeout for the database connection specified by *db*. The initial value can be specified on the qdb command line with the -t option; by default, it's 5000 ms. Specifying a value of 0 is the same as setting QDB_TIMEOUT_NONBLOCK.

The timeout is the amount of time that a client can attempt to access a database before it returns EBUSY. If two clients attempt to write to the database, 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 is affected by the timeout value. If you set or toggle `QDB_CONN_NONBLOCKING`, the busy timeout value is set to 0 or back to the `-t` value. Similarly, if you set the timeout to be `QDB_TIMEOUT_NONBLOCK`, the `QDB_CONN_NONBLOCKING` flag is set.

The `QDB_CONN_NONBLOCKING` flag bit can be set with *qdb_connect()* (p. 131) and toggled with *qdb_parameters()* (p. 156).

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

qdb_setoption()

Set a database connection option

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

option

The option to set. It 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()* (p. 151) 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()* (p. 162) will query the database connection for this information instead.

By default, this option is on.

QDB_OPTION_COLUMN_NAMES

Write the column names into the `qdb_result_t` structure returned by *qdb_getresult()* (p. 145). If this option isn't set, that data isn't provided and calling *qdb_column_index()* (p. 126) won't work.

By default, this option is on.

QDB_OPTION_COLUMN_DECLTYPES

Write the declared column types in the `qdb_result_t` structure returned by *qdb_getresult()* (p. 145). If this option isn't set, that data isn't provided and calling *qdb_column_decltype()* (p. 124) won't work.

By default, this option is off.

value

The new value for the option, either 0 (off) or 1 (on).

Library:

qdb

Description:

This function sets options for the database connection *db*. By default, all supported options are enabled (on) except for `QDB_OPTION_COLUMN_DECLTYPES`.

Returns:

>=0

Success. The previous value for *option* is returned.

-1

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

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_snprintf()

Print formatted output to a length-bounded string

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 *buf* string, including the null-terminator. The function will always write a null-terminator if *n* is positive.

buf

A pointer to the buffer for storing 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 QNX Neutrino *C Library Reference*.

Library:

qdb

Description:

This function is a variant of *snprintf()* in 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 null-terminator if *n* is positive

For more information about additional formatting options, see *qdb_mprintf()* (p. 153).

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 the formatted output string (*buf*)

Success.

NULL

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_statement()

Execute an SQL statement

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

format

A pointer to a formatting string to process. The formatting string determines what additional arguments you need to provide. The string that results from combining *format* and the additional arguments is executed as SQL code on the database.

Library:

`qdb`

Description:

This function combines the formatting string in *format* with the values of the additional arguments to construct an SQL command string and then executes that string on the database referred to by *db*.

Individual statements within the command string must be completed with and separated by semicolons. There's no length restriction for the command string.

The formatting string and additional arguments work in the same way as with *printf()* (all the same conversion specifiers apply). There are additional conversion specifiers, `%q` and `%Q`, which in general should be used instead of `%s` for inserting text into a literal string. The `%q` specifier properly escapes special characters for SQL. For more information, see *qdb_mprintf()* (p. 153).

To determine how many rows were affected by the SQL command string, you can call *qdb_rowchanges()* (p. 162) after executing the command string.

Because *qdb_rowchanges()* returns the number of rows affected by only the last SQL statement executed by *qdb_statement()*, we recommend defining only one statement in each *qdb_statement()* call; otherwise, if there's an issue, you won't be able to determine which statement failed.

By default, the SQL code is executed on the database before *qdb_statement()* returns. However, if the connection is in *asynchronous mode* (p. 46), this function may return before the SQL code completes execution and may not report errors. In this case, you need to call *qdb_getresult()* (p. 145) to retrieve any errors.

Returns:

>=0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_stmt_decltypes()

Get the declared column types for a prepared statement

Synopsis:

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

int qdb_stmt_decltypes( qdb_hdl_t *db,
                        int stmtid,
                        char **buf,
                        ssize_t bufsize,
                        ssize_t *required_size );
```

Arguments:

db

A pointer to the database handle.

stmtid

The prepared statement ID returned by *qdb_stmt_init()*.

buf

A pointer to a buffer for storing the column types.

bufsize

The actual buffer size, in bytes.

required_size

The required buffer size, in bytes, for storing all declared column types. The function always fills in this field, even if you provide a sufficiently large buffer.

Library:

qdb

Description:

This function gets the declared column types for the prepared statement referred to by *stmtid*. The behavior of this function depends on the argument settings:

- If *buf* is undefined (i.e., `NULL`) and *bufsize* is 0, the number of bytes needed for the buffer is written in *required_size* and the total number of columns in the statement is returned.
- If *buf* is defined and *bufsize* is less than *required_size*, only partial data is written in the buffer and the number of valid declared column types is returned.
- If *buf* is defined and *bufsize* is greater than or equal to *required_size*, the declared column types are written in the buffer and the total number of columns is returned.

When this function returns, the beginning of *buf* is an array of pointers to strings, which are also stored in the buffer. These buffer strings contain the individual declared column types. Note that if a column is the result of an expression or subquery, an empty string is written in the corresponding buffer position.

You must allocate (and manage) the memory in *buf*. If you need to know the buffer size required to store the results, call this function with *bufsize* set to 0. The function will write the necessary number of bytes in *required_size*. You can then use this value to allocate the required amount of memory and call the function again, passing in a pointer to the newly allocated buffer.

Returns:

>=0

Success. The returned value is either the number of columns in the statement or the number of valid declared column types, depending on the arguments.

-1

An error occurred (*errno* is set).

Examples:

The following code sample demonstrates how you can call *qdb_stmt_decltypes()* once to determine the required buffer size and again to retrieve the declared column types:

```
char **pp;
ssize_t required_size, bufsize = 0;
int cols, i;

if ((cols = qdb_stmt_decltypes(db, stmtid, NULL,
                0, &required_size)) > 0) {
   pp = malloc(required_size);
   if (pp) {
      bufsize = required_size;
      cols = qdb_stmt_decltypes(db, stmtid, pp,
                bufsize, &required_size);
      for (i=0; i<cols; i++)
         printf("column %d: %s\n", i, pp[i]);
      free(pp);
```

```
    }
}
```

You can optimize the use of *qdb_stmt_decltypes()* by providing a buffer that you estimate is large enough before you call this function for the first time. On return, if *bufsize* is greater than or equal to *required_size*, then all the data has been returned and you don't need to call the function again. This also lets you re-use a single, sufficiently large buffer.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_stmt_exec()

Execute a precompiled statement

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

stmtid

The ID of a precompiled statement to execute.

bindings

An array of `qdb_binding_t` (p. 110) structures containing pointers to data that will be bound to the variable parameters in the precompiled statement.

binding_count

The number of items in *bindings*.

Library:

`qdb`

Description:

This function executes a precompiled statement prepared with *qdb_stmt_init()* (p. 180). If the SQL string previously passed to *qdb_stmt_init()* contains variable parameters, you can bind data to these parameters by placing the data values in one or more `qdb_binding_t` structures and then passing in these structures through the *bindings* argument. Parameters that aren't filled in are interpreted as `NULL`.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Examples:

See *qdb_stmt_init()* (p. 180) for an example on how to compile, execute, and free an SQL statement.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_stmt_free()

Free a precompiled statement

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

stmtid

The ID of a precompiled statement to free.

Library:

qdb

Description:

This function frees a statement previously compiled by *qdb_stmt_init()* (p. 180). It's not strictly necessary to call this function, as all precompiled statements are freed when you call *qdb_disconnect()* (p. 135).

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Examples:

See *qdb_stmt_init()* (p. 180) for an example on how to compile, execute, and free an SQL statement.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_stmt_init()

Initialize a precompiled statement

Synopsis:

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

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

Arguments:

db

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()* (p. 176).

len

The length of *sql*.

Library:

qdb

Description:

This function initializes a prepared (precompiled) SQL statement. A prepared statement is compiled once but can be executed multiple times. This function returns a statement ID for the precompiled statement, which you need to pass in to *qdb_stmt_exec()* (p. 176).

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()* (p. 178), although all precompiled statements are freed when you call *qdb_disconnect()* (p. 135).

Returns:

>=0

Success. The returned value is the prepared statement's ID. Note that 0 is a valid statement ID.

-1

An error occurred (*errno* is set).

Examples:

The following code sample shows how to compile, execute, and free an SQL 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

Safety:	
Thread	Yes

qdb_vacuum()

Vacuum a database

Synopsis:

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

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

Arguments:

db

A pointer to the database handle.

scope

The scope of the operation (see the *scope* argument in *qdb_backup()* (p. 108) for more information).

Library:

qdb

Description:

This function starts a vacuum operation on the specified database and any auto-attached databases (which are listed in the main databases' `.aa` file). This is an alternative to using the *VACUUM* (p. 232) command for each database.

You can call *qdb_getdbsize()* (p. 137) to determine whether a database should be vacuumed.

If the auto-vacuum mode is enabled (see the *PRAGMA* (p. 232) SQL command for details), databases are vacuumed whenever free space is created. By default, auto-vacuum mode is disabled.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

qdb_vmprintf()

Print formatted output to a new string

Synopsis:

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

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

Arguments:

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 QNX Neutrino *C Library Reference*.

arg

A variable-argument list of the additional arguments. You must have initialized this list 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()* (p. 153).

Returns:

A pointer to a formatted string

Success.

NULL

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 9
QDB SQL Reference

QDB supports a subset of ANSI SQL-92. The supported capabilities are presented in the following sections:

- Details on the *Row ID and Autoincrement* (p. 188) features
- Syntax for SQL *Comments* (p. 190)
- Syntax and semantics of SQL *Expressions* (p. 191)
- SQL *Keywords* (p. 199) recognized by QDB
- Information about *Statements* (p. 204)

Row ID and Autoincrement

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 prevent row IDs from ever being reused in a table by using the `AUTOINCREMENT` keyword.

Description:

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 number that can be stored in a signed 64-bit integer (9223372036854775807), then the database engine starts picking candidate IDs at random until it finds one that isn't already 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 you create 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 doesn't 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 INSERTs to run a little more slowly.

Comments

Comments make your SQL queries easier to read and understand.

Syntax:

```
-- 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.

Expressions

SQL expressions are subcomponents of most other commands. Expressions combine one or more values, operators, and SQL functions to produce a result that can used in the enclosing command.

Syntax:

```
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 made up of several smaller components, including literals for specifying exact values, operators for comparing values and performing pattern matching, and functions for calculating and modifying values. These smaller components evaluate to a single result that's used in a broader SQL command.

Operators

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()* (p. 176). 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)* (p. 195).

`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)* (p. 195) 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 refer to the row key only if the CREATE TABLE statement doesn't 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 * ... doesn't 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 SELECTs (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()* (p. 151).

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* with 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 two 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 two 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 you're 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 occurrence 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: "3.7.9"

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 leftmost 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 indexes 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* (p. 53).

upper(X)

Return a copy of input string *X* converted to all uppercase 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, which takes the argument *X*, returns the number of times that *X* is not NULL in a group. The second form, which takes no arguments, 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 no 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 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.

Keywords

QDB recognizes more than 100 SQL keywords related to command names, operators, sorting directives, and table, column, and query constraints. You must use special syntax to use keywords as names of program objects.

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*

- **ADD**
- *AFTER*
- **ALL**
- **ALTER**
- *ANALYZE*
- **AND**
- **AS**
- *ASC*
- *ATTACH*
- **AUTOINCREMENT**
- *BEFORE*
- *BEGIN*
- **BETWEEN**
- **BY**
- *CASCADE*
- **CASE**
- *CAST*
- **CHECK**
- **COLLATE**
- **COMMIT**
- *CONFLICT*
- **CONSTRAINT**
- **CREATE**
- **CROSS**
- *CURRENT_DATE*
- *CURRENT_TIME*
- *CURRENT_TIMESTAMP*
- *DATABASE*
- **DEFAULT**
- **DEFERRABLE**
- *DEFERRED*
- **DELETE**
- *DESC*
- *DETACH*
- **DISTINCT**
- **DROP**

- *EACH*
- **ELSE**
- *END*
- **ESCAPE**
- **EXCEPT**
- *EXCLUSIVE*
- *EXPLAIN*
- *FAIL*
- *FOR*
- **FOREIGN**
- **FROM**
- **FULL**
- *GLOB*
- **GROUP**
- **HAVING**
- *IF*
- *IGNORE*
- *IMMEDIATE*
- **IN**
- **INDEX**
- *INITIALLY*
- **INNER**
- **INSERT**
- *INSTEAD*
- **INTERSECT**
- **INTO**
- **IS**
- **ISNULL**
- **JOIN**
- *KEY*
- **LEFT**
- *LIKE*
- **LIMIT**
- *MATCH*
- **NATURAL**
- **NOT**

- **NOTNULL**
- **NULL**
- *OF*
- *OFFSET*
- **ON**
- **OR**
- **ORDER**
- **OUTER**
- *PLAN*
- *PRAGMA*
- **PRIMARY**
- *QUERY*
- *RAISE*
- **REFERENCES**
- *REINDEX*
- *RENAME*
- *REPLACE*
- *RESTRICT*
- **RIGHT**
- **ROLLBACK**
- *ROW*
- **SELECT**
- **SET**
- **TABLE**
- *TEMP*
- *TEMPORARY*
- **THEN**
- **TO**
- **TRANSACTION**
- *TRIGGER*
- **UNION**
- **UNIQUE**
- **UPDATE**
- **USING**
- *VACUUM*
- **VALUES**

- *VIEW*
- *VIRTUAL*
- **WHEN**
- **WHERE**

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`

Statements

QDB recognizes more than 20 types of SQL statements that create, modify, and delete tables, indexes, and triggers, and perform specialized tasks such as database cleanup.

The statements described in this appendix are:

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 by hand to use the new table name.

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 cleaned up with the *VACUUM* (p. 232) statement.

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 can't be erased by the *DROP* (p. 216) command, but all its content can be deleted with the *DELETE* (p. 214) statement, 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* (p. 215) 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 that 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 that 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 entries 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* (p. 215) 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 consists of the column name followed by the datatype for that column, then one or more optional column constraints. The datatype for the column doesn't restrict what data may be put in that column. See the chapter *Datatypes in QDB* (p. 53) for additional information.

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.

Tables are removed using the *DROP TABLE* (p. 216) statement.

Temp keyword

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.

Default constraint

The DEFAULT constraint specifies a default value to use for a column 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 doesn't 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 column. 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*.

Primary Key constraint

Specifying a PRIMARY KEY normally just creates a UNIQUE index on the corresponding columns. However, if the primary key is on a single column that has the INTEGER datatype, then that column is used internally as the actual key of the B-Tree for the table. This means that the column may hold only 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 doesn't have an INTEGER PRIMARY KEY column, then the B-Tree key will be an 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 given in the "*Row ID and Autoincrement* (p. 188)" section.

The SQL standard specifies that PRIMARY KEY should always imply NOT NULL. Unfortunately, due to a long-standing coding oversight, SQLite has allowed NULL values in PRIMARY KEY columns. At the time of this writing, SQLite continues to allow NULL values in PRIMARY KEY columns to support legacy code. However, SQLite may eventually be changed to support the SQL standard. Therefore, developers should design new programs to conform to the SQL standard.

Unique constraint

The UNIQUE constraint is similar to the PRIMARY KEY constraint, except that a table may have any number of UNIQUE constraints. Each UNIQUE constraint creates an index on the specified columns. This index must contain unique keys.

Collate constraint

The COLLATE constraint specifies what *text-collating function* (p. 62) to use when comparing text entries for the column. The built-in BINARY collating function is used by default.

Not Null constraint

A NOT NULL constraint dictates that the associated column may not contain a NULL value. Attempting to set the column value to NULL when inserting a new row or updating an existing one causes a constraint violation.

Check constraint

CHECK constraints are now supported and enforced. Each time a new row is inserted into the table or an existing row is updated, the expression associated with each CHECK constraint is evaluated and cast to a NUMERIC value. If the result is zero (integer value 0 or real value 0.0), then a constraint violation has occurred. If the CHECK expression evaluates to NULL, or any other non-zero value, it is not a constraint violation.

Constraint conflict resolution

Constraint violations are handled by constraint conflict resolution algorithms. Each PRIMARY KEY, UNIQUE, NOT NULL, and CHECK constraint has a default algorithm for resolving constraint conflicts. The optional *conflict-clause* following a constraint lets you specify an alternative algorithm for resolving constraint conflicts. The default algorithm is ABORT. Different constraints within the same table may have different 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* (p. 218) for additional information.

Statement storage and interpretation

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.

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 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 be used only 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.

You may create triggers 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).

Examples:

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* (p. 216) 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* (p. 228) 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* (p. 211) on the view to accomplish the same thing. Views are removed with the *DROP VIEW* (p. 217) 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* (p. 206) 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* (p. 207) 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 doesn't reduce the size of the database file in the default mode. Empty space in the database is retained for later INSERTs. To remove free space in the database, use the *VACUUM* (p. 232) command. If the auto-vacuum 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 doesn't 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* (p. 208) 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 doesn't reduce the size of the database file in the default mode. Empty space in the database is retained for later INSERTs. To remove free space in the database, use the *VACUUM* (p. 232) command. If the auto-vacuum 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 doesn't exist.

DROP TRIGGER

Remove a trigger

Synopsis:

```
DROP TRIGGER [IF EXISTS] [database-name .] trigger-name
```

Description:

The DROP TRIGGER statement removes a trigger created by the *CREATE TRIGGER* (p. 211) statement. The trigger is deleted from the database schema.

Triggers are automatically dropped when the associated table is dropped.

DROP VIEW

Remove a view

Synopsis:

```
DROP VIEW [IF EXISTS] view-name
```

Description:

The `DROP VIEW` statement removes a view created by the *`CREATE VIEW`* (p. 214) 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* (p. 65) 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* (p. 218) for additional information. For compatibility with MySQL, the parser allows the use of the single keyword *REPLACE* (p. 227) 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 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 doesn't 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 doesn't 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 the following 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* (p. 221)
   - *Cache Size* (p. 222)
   - *Case Sensitivity* (p. 222)
   - *Count Changes* (p. 222)
   - *Default Cache Size* (p. 223)
   - *Full Column Names* (p. 223)
   - *Full Column Names* (p. 223)
   - *Legacy File Format* (p. 223)
   - *Page Size* (p. 223)
   - *Short Column Names* (p. 224)
   - *Synchronous* (p. 224)
   - *Temp Store* (p. 224)
2. Pragmas used to query the schema of the current database:
   - *Foreign Key List* (p. 225)
   - *Index Info* (p. 225)
   - *Index List* (p. 225)
   - *Table Info* (p. 226)
3. Pragmas used to query or modify the databases' two version values, the schema-version and the user-version:
   - *Schema and User Version* (p. 226)
4. Pragmas used to debug the library and verify that database files are not corrupted:
   - *Integrity Check* (p. 226)

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* (p. 232) command or *qdb_vacuum()* (p. 183) 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* (p. 183) (or the *VACUUM* (p. 232) 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 UPDATEs or DELETEs 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 doesn't 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're 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. However, 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
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 version check 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) that 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.

The scope of the affected database indexes depends on the type of database object given to the command, as summarized in the following behavioral rules:

- If you specify a *collation_name*, all the indexes in any attached databases that use the specified collation sequence are recreated.
- If you specify a [*database-name* .] *table-name* combination, all indexes associated with the table are rebuilt.
- If you specify a [*database-name* .] *index-name* combination, only this specified index is deleted and recreated.
- If don't specify a *database-name* but there exists both a collation sequence and either a table or index with the specified name, the indexes associated with the collation sequence are only reconstructed. You can avoid this ambiguity by always specifying a *database-name* when reindexing a 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* (p. 218) command. This alias is provided for compatibility with MySQL.

SELECT

Query a database

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 clause 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 because it maximizes compatibility with legacy SQL database systems.

Compound SELECT statements

A compound SELECT is formed from two or more simple SELECTs connected by one of the operators UNION, UNION ALL, INTERSECT, or EXCEPT. In a compound SELECT, all the constituent SELECTs 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 SELECTs 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 SELECTs. EXCEPT takes the result of left SELECT after removing the results of the right SELECT. When three or more SELECTs 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* (p. 218) 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 doesn'thing. 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 doesn't 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* (p. 218) 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* (p. 221). 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
```

Chapter 10
fileset Reference

Copy a file set

Synopsis:

```
fileset [-6] [-b backupdir] [-c fname] [-m] [-P perms] [-p pattern]
        [-t savetime] [-v] [-x] [load | save | test] prmdir tmpdir
```

Options:

-6

Limit DMA so it doesn't cross a 64K boundary.

-b *backupdir*

(load command)

Use this directory instead of *prmdir* as a backup if the check file is missing or bad.

-c *fname*

Create a check file called *fname*; the default name is `_FILESET_`.

-m

Create any necessary directories.

-P *perms*

Assign access permissions to backup files. For more information, see "*Access permissions*" in the *QNX Neutrino C Library Reference*.

-p *pattern*

Define a set of files to copy based on a filename pattern. You can specify multiple file sets by using up to 16 `-p` options.

By default, only the `"*"` (all files) pattern is defined. When you define a pattern for a file set, the default pattern is overridden.

-t *savetime*

(save command)

Use the provided time value for the access and modification times.

-v

Increase output verbosity. Messages are written to `stdout`. This option is cumulative, allowing you to specify up to three `-v` options for maximum verbosity.

This option is handy when you're trying to understand the operation of QDB, but when many `-v` arguments are used, the logging becomes quite significant and can change timing noticeably. The verbosity setting is good for systems under development but should probably not be used in production systems or when performance testing.

-x

Don't copy files that haven't changed. By default, all files are copied.

load

Load the set of files listed in the check file into the temporary directory. QDB issues the `load` command at startup, causing `fileset` to look for a check file in the path defined by *prmdir*. On success, each file listed in the check file is copied from *prmdir* to *tmpdir* and `fileset` returns an exit status of `0` (`EOK`).

Any of following conditions cause `fileset` to exit with an error:

- The check file doesn't exist.
- The CRC (checksum) for the check file is wrong.
- A file has a size or modification time that doesn't match the information in the check file.
- A file listed in the check file is missing from the database.
- An error occurs during a file copy.

When it encounters an erroneous condition, `fileset` sets *errno* and removes the links to any files it copied before the error occurred.

save

Save the files during shutdown or at any other time as needed. When given the `save` command, `fileset` looks for a valid check file in the path defined by *prmdir*, then performs one of these actions based on the check file status:

Missing or invalid check file

If the check file doesn't exist or the CRC (checksum) for its data is wrong, `fileset` creates a new check file, based on the patterns defined with `-p`.

Valid check file

If the check file exists and has a valid CRC (checksum), `fileset`:

1. Loads the check file (which has the list of files to copy) into memory.
2. Deletes any existing check file in the permanent directory (`prmdir`), because it's about to modify files in that directory.
3. Copies some or all files listed in the check file to the permanent directory, depending on the `-x` setting. When this last option is specified, `fileset` compares the size and modification time for each listed file with the information of the actual file in the temporary directory (`tmpdir`). If the values match, `fileset` assumes that the file is unchanged and doesn't copy it. If they don't match, `fileset` copies the file to the permanent directory. When `-x` isn't specified, all files are copied, whether they've changed or not.
4. Creates a new check file in the permanent directory to mark the directory as valid.

`test`

Test the information in the check file against the contents of the permanent directory. When given this command, `fileset` returns an exit status as follows:

`0`

The size and modification time for each file listed in the check file matches the information of the actual file in *prmdir*.

`<>0`

There's a mismatch in the size or modification time for a file listed in the check file, or there's a problem with the *prmdir* directory.

Description:

The `fileset` utility is used by QDB to copy files during database backups when the *`diocopy` compression option* (p. 20) is set in the database's configuration object.

QDB launches `fileset` when you call *qdb_backup()* (p. 108) and when the database is restored on startup.

The `fileset` binary must be in a path specified in QDB's *PATH* environment variable.

While `fileset` can copy a set of files between any directories on any filesystem (disk, flash, or tmpfs), the utility optimizes the copy path between disk and tmpfs by using DMA transfer to move files between these filesystem types. It requires about one-tenth of the CPU used to move data by a traditional read/write transfer from a program, and it can achieve better platter/interface transfer rates.

In this release, `fileset` can copy files to multiple backup directories specified in the *`BackupDir::`* (p. 19) parameter in the database configuration object.

Limitations

The efficiency of `fileset` imposes some limitations. This utility:

- Doesn't read or copy subdirectory contents.
- Doesn't create a source directory.
- Creates a destination directory only if you use `-m`.
- Either completely succeeds or completely fails. There are no partial results. If it encounters an error while copying any file, it unlinks from the destination directory any files already copied and then exits with an error.

Examples

The following is a typical sequence of events:

1. **At startup:**

   ```
   # fileset load /fs/hd0/myMediaDB /tmpfs/myMediaDB
   ```

   If this sequence fails, the system must take action; typically, this entails creating a new set of database files.

2. **At shutdown:**

   ```
   # fileset -p "*.dat" save /fs/hd0/myTunes /tmpfs/myTunes
   ```

   If there's no check file (because it's the command is being run for the first time or a serious error occurred), the system creates a check file, using the `-p` option to define the file set.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W