package

android.database.sqlite

Contains the SQLite database management classes that an application would use to manage its own private database.

Applications use these classes to manage private databases. If creating a content provider, you will probably have to use these classes to create and manage your own database to store content. See Content Providers to learn the conventions for implementing a content provider. See the NotePadProvider class in the NotePad sample application in the SDK for an example of a content provider. Android ships with SQLite version 3.4.0

If you are working with data sent to you by a provider, you will not use these SQLite classes, but instead use the generic `android.database` classes.

Android ships with the sqlite3 database tool in the `tools/` folder. You can use this tool to browse or run SQL commands on the device. Run by typing `sqlite3` in a shell window.

Interfaces

SQLiteCursorDriver	A driver for SQLiteCursors that is used to create them and gets notified by the cursors it creates on significant events in their lifetimes.
SQLiteDatabase.CursorFactory	Used to allow returning sub-classes of `Cursor` when calling query.
SQLiteTransactionListener	A listener for transaction events.

Classes

SQLiteClosable	An object created from a SQLiteDatabase that can be closed.
SQLiteCursor	A Cursor implementation that exposes results from a query on a `SQLiteDatabase`.
SQLiteDatabase	Exposes methods to manage a SQLite database.
SQLiteOpenHelper	A helper class to manage database creation and version management.

SQLiteProgram	A base class for compiled SQLite programs.
SQLiteQuery	A SQLite program that represents a query that reads the resulting rows into a CursorWindow.
SQLiteQueryBuilder	This is a convience class that helps build SQL queries to be sent to `SQLiteDatabase` objects.
SQLiteStatement	A pre-compiled statement against a `SQLiteDatabase` that can be reused.

Exceptions

SQLiteAbortException	An exception that indicates that the SQLite program was aborted.
SQLiteAccessPermException	This exception class is used when sqlite can't access the database file due to lack of permissions on the file.
SQLiteBindOrColumnIndexOutOfRangeException	Thrown if the the bind or column parameter index is out of range
SQLiteBlobTooBigException	
SQLiteCantOpenDatabaseException	
SQLiteConstraintException	An exception that indicates that an integrity constraint was violated.
SQLiteDatabaseCorruptException	An exception that indicates that the SQLite database file is corrupt.
SQLiteDatabaseLockedException	Thrown if the database engine was unable to acquire the database locks it needs to do its job.
SQLiteDatatypeMismatchException	

SQLiteDiskIOException	An exception that indicates that an IO error occured while accessing the SQLite database file.
SQLiteDoneException	An exception that indicates that the SQLite program is done.
SQLiteException	A SQLite exception that indicates there was an error with SQL parsing or execution.
SQLiteFullException	An exception that indicates that the SQLite database is full.
SQLiteMisuseException	This error can occur if the application creates a SQLiteStatement object and allows multiple threads in the application use it at the same time.
SQLiteOutOfMemoryException	
SQLiteReadOnlyDatabaseException	
SQLiteTableLockedException	

SQLiteDatabase

extends SQLiteClosable

java.lang.Object

↳android.database.sqlite.SQLiteClosable

↳android.database.sqlite.SQLiteDatabase

Class Overview

Exposes methods to manage a SQLite database.

SQLiteDatabase has methods to create, delete, execute SQL commands, and perform other common database management tasks.

See the Notepad sample application in the SDK for an example of creating and managing a database.

Database names must be unique within an application, not across all applications.

Localized Collation - ORDER BY

In addition to SQLite's default `BINARY` collator, Android supplies two more, `LOCALIZED`, which changes with the system's current locale if you wire it up correctly (XXX a link needed!), and `UNICODE`, which is the Unicode Collation Algorithm and not tailored to the current locale.

Summary

Nested Classes		
interface	SQLiteDatabase.CursorFactory	Used to allow returning sub-classes of `Cursor` when calling query.
Constants		
int	CONFLICT_ABORT	When a constraint violation occurs,no ROLLBACK is executed so change

		commands within the same transaction are preserved.
int	CONFLICT_FAIL	When a constraint violation occurs, the command aborts with a return SQLITE_CONSTRAINT.
int	CONFLICT_IGNORE	When a constraint violation occurs, the one row that contains the cons not inserted or changed.
int	CONFLICT_NONE	use the following when no conflict action is specified.
int	CONFLICT_REPLACE	When a UNIQUE constraint violation occurs, the pre-existing rows that constraint violation are removed prior to inserting or updating the curr
int	CONFLICT_ROLLBACK	When a constraint violation occurs, an immediate ROLLBACK occurs, th current transaction, and the command aborts with a return code of SQ
int	CREATE_IF_NECESSARY	Flag for `openDatabase(String, SQLiteDatabase.CursorFacto` create the database file if it does not already exist.
int	MAX_SQL_CACHE_SIZE	absolute max value that can be set by `setMaxSqlCacheSize(int)` s prepared-statement is between 1K - 6K, depending on the complexity c statement & schema.
int	NO_LOCALIZED_COLLATORS	Flag for `openDatabase(String, SQLiteDatabase.CursorFacto` the database without support for localized collators.
int	OPEN_READONLY	Flag for `openDatabase(String, SQLiteDatabase.CursorFacto` the database for reading only.
int	OPEN_READWRITE	Flag for `openDatabase(String, SQLiteDatabase.CursorFacto` the database for reading and writing. If the disk is full, this may fail eve actually write anything.
int	SQLITE_MAX_LIKE_PATTERN_LENGTH	Maximum Length Of A LIKE Or GLOB Pattern The pattern matching algo default LIKE and GLOB implementation of SQLite can exhibit O(N^2) pe N is the number of characters in the pattern) for certain pathological ca

Public Methods

void	beginTransaction() Begins a transaction in EXCLUSIVE mode.
void	beginTransactionNonExclusive() Begins a transaction in IMMEDIATE mode.
void	beginTransactionWithListener(SQLiteTransactionListener transactionListener) Begins a transaction in EXCLUSIVE mode.
void	beginTransactionWithListenerNonExclusive(SQLiteTransactionListener transactionListener Begins a transaction in IMMEDIATE mode.
void	close() Close the database.
SQLiteStatement	compileStatement(String sql) Compiles an SQL statement into a reusable pre-compiled statement object.
static SQLiteDatabase	create(SQLiteDatabase.CursorFactory factory) Create a memory backed SQLite database.
int	delete(String table, String whereClause, String[] whereArgs) Convenience method for deleting rows in the database.
boolean	enableWriteAheadLogging() This method enables parallel execution of queries from multiple threads on the same data
void	endTransaction() End a transaction.
void	execSQL(String sql)

	Execute a single SQL statement that is NOT a SELECT or any other SQL statement that retu
void	execSQL(String sql, Object[] bindArgs) Execute a single SQL statement that is NOT a SELECT/INSERT/UPDATE/DELETE.
static String	findEditTable(String tables) Finds the name of the first table, which is editable.
List<Pair<String, String>>	getAttachedDbs() Returns list of full pathnames of all attached databases including the main database by ex
long	getMaximumSize() Returns the maximum size the database may grow to.
long	getPageSize() Returns the current database page size, in bytes.
final String	getPath() Getter for the path to the database file.
Map<String, String>	getSyncedTables() *This method is deprecated. This method no longer serves any useful purpose and has been*
int	getVersion() Gets the database version.
boolean	inTransaction() return true if there is a transaction pending
long	insert(String table, String nullColumnHack, ContentValues values) Convenience method for inserting a row into the database.

long	insertOrThrow(String table, String nullColumnHack, ContentValues values) Convenience method for inserting a row into the database.
long	insertWithOnConflict(String table, String nullColumnHack, ContentValues initialValues, int General method for inserting a row into the database.
boolean	isDatabaseIntegrityOk() Runs 'pragma integrity_check' on the given database (and all the attached databases) and databases) pass integrity_check, false otherwise.
boolean	isDbLockedByCurrentThread() Checks if the database lock is held by this thread.
boolean	isDbLockedByOtherThreads() Checks if the database is locked by another thread.
boolean	isOpen()
boolean	isReadOnly() return whether the DB is opened as read only.
void	markTableSyncable(String table, String foreignKey, String updateTable) *This method is deprecated. This method no longer serves any useful purpose and has been*
void	markTableSyncable(String table, String deletedTable) *This method is deprecated. This method no longer serves any useful purpose and has been*
boolean	needUpgrade(int newVersion)
static SQLiteDatabase	openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseError Open the database according to the flags `OPEN_READWRITE` `OPEN_READONLY` `CREATE_I`

static SQLiteDatabase	openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags) Open the database according to the flags OPEN_READWRITE OPEN_READONLY CREATE_I
static SQLiteDatabase	openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory, DatabaseErro Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY, errorHandler).
static SQLiteDatabase	openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory) Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY).
static SQLiteDatabase	openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory) Equivalent to openDatabase(file.getPath(), factory, CREATE_IF_NECESSARY).
Cursor	query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy Query the given table, returning a Cursor over the result set.
Cursor	query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy Query the given table, returning a Cursor over the result set.
Cursor	query(boolean distinct, String table, String[] columns, String selection, String[] selectionAr Query the given URL, returning a Cursor over the result set.
Cursor	queryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String group Query the given URL, returning a Cursor over the result set.
Cursor	rawQuery(String sql, String[] selectionArgs) Runs the provided SQL and returns a Cursor over the result set.
Cursor	rawQueryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, String sql, String[] sel Runs the provided SQL and returns a cursor over the result set.
static int	releaseMemory()

	Attempts to release memory that SQLite holds but does not require to operate properly.
long	replace(String table, String nullColumnHack, ContentValues initialValues) Convenience method for replacing a row in the database.
long	replaceOrThrow(String table, String nullColumnHack, ContentValues initialValues) Convenience method for replacing a row in the database.
void	setLocale(Locale locale) Sets the locale for this database.
void	setLockingEnabled(boolean lockingEnabled) Control whether or not the SQLiteDatabase is made thread-safe by using locks around crit
void	setMaxSqlCacheSize(int cacheSize) Sets the maximum size of the prepared-statement cache for this database.
long	setMaximumSize(long numBytes) Sets the maximum size the database will grow to.
void	setPageSize(long numBytes) Sets the database page size.
void	setTransactionSuccessful() Marks the current transaction as successful.
void	setVersion(int version) Sets the database version.
int	update(String table, ContentValues values, String whereClause, String[] whereArgs) Convenience method for updating rows in the database.

int	updateWithOnConflict(String table, ContentValues values, String whereClause, String[] wh Convenience method for updating rows in the database.
boolean	yieldIfContended() *This method is deprecated. if the db is locked more than once (becuase of nested transacti yieldIfContendedSafely instead.*
boolean	yieldIfContendedSafely(long sleepAfterYieldDelay) Temporarily end the transaction to let other threads run.
boolean	yieldIfContendedSafely() Temporarily end the transaction to let other threads run.

Protected Methods	
void	finalize() Invoked when the garbage collector has detected that this instance is no longer reachable.
void	onAllReferencesReleased()

[Expand] **Inherited Methods**
▶ From class android.database.sqlite.SQLiteClosable
▶ From class java.lang.Object

Constants

public static final int ***CONFLICT_ABORT***

Since: API Level 8

When a constraint violation occurs,no ROLLBACK is executed so changes from prior commands within the same transaction are preserved. This is the default behavior.

Constant Value: 2 (0x00000002)

public static final int ***CONFLICT_FAIL***

Since: API Level 8

When a constraint violation occurs, the command aborts with a return code SQLITE_CONSTRAINT. But any changes to the database that the command made prior to encountering the constraint violation are preserved and are not backed out.
Constant Value: 3 (0x00000003)

public static final int ***CONFLICT_IGNORE***

Since: API Level 8

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.
Constant Value: 4 (0x00000004)

public static final int ***CONFLICT_NONE***

Since: API Level 8

use the following when no conflict action is specified.
Constant Value: 0 (0x00000000)

public static final int ***CONFLICT_REPLACE***

Since: API Level 8

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 insert or update always occurs. The command continues executing normally. No error is returned. If a NOT NULL constraint violation occurs, the NULL value is replaced by the default value for that column. If the column has no default value, then the ABORT algorithm is used. If a CHECK constraint violation occurs then the IGNORE algorithm is used. When this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows. This behavior might change in a future release.
Constant Value: 5 (0x00000005)

public static final int ***CONFLICT_ROLLBACK***

Since: API Level 8

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.
Constant Value: 1 (0x00000001)

public static final int ***CREATE_IF_NECESSARY***

Since: API Level 1

Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to create the database file if it does not already exist.

Constant Value: 268435456 (0x10000000)

public static final int ***MAX_SQL_CACHE_SIZE***

Since: API Level 11

absolute max value that can be set by `setMaxSqlCacheSize(int)` size of each prepared-statement is between 1K - 6K, depending on the complexity of the SQL statement & schema.

Constant Value: 100 (0x00000064)

public static final int ***NO_LOCALIZED_COLLATORS***

Since: API Level 1

Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database without support for localized collators.

This causes the collator `LOCALIZED` not to be created. You must be consistent when using this flag to use the setting the database was created with. If this is set,`setLocale(Locale)` will do nothing.

Constant Value: 16 (0x00000010)

public static final int ***OPEN_READONLY***

Since: API Level 1

Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database for reading only. This is the only reliable way to open a database if the disk may be full.

Constant Value: 1 (0x00000001)

public static final int ***OPEN_READWRITE***

Since: API Level 1

Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database for reading and writing. If the disk is full, this may fail even before you actually write anything.

Note that the value of this flag is 0, so it is the default.

Constant Value: 0 (0x00000000)

public static final int ***SQLITE_MAX_LIKE_PATTERN_LENGTH***

Since: API Level 1

Maximum Length Of A LIKE Or GLOB Pattern The pattern matching algorithm used in the default LIKE and GLOB implementation of SQLite can exhibit O(N^2) performance (where N is the number of characters in the pattern) for certain pathological cases. To avoid denial-of-service attacks the length of the LIKE or GLOB pattern is limited to SQLITE_MAX_LIKE_PATTERN_LENGTH bytes. The default value of this limit is 50000. A modern workstation can evaluate even a pathological LIKE or GLOB pattern of 50000 bytes relatively quickly. The denial of service problem only comes into play when the pattern length gets into millions of bytes.

Nevertheless, since most useful LIKE or GLOB patterns are at most a few dozen bytes in length, paranoid application developers may want to reduce this parameter to something in the range of a few hundred if they know that external users are able to generate arbitrary patterns.
Constant Value: 50000 (0x0000c350)

Public Methods

*public void **beginTransaction** ()*

Since: API Level 1

Begins a transaction in EXCLUSIVE mode.

Transactions can be nested. When the outer transaction is ended all of the work done in that transaction and all of the nested transactions will be committed or rolled back. The changes will be rolled back if any transaction is ended without being marked as clean (by calling setTransactionSuccessful). Otherwise they will be committed.

Here is the standard idiom for transactions:

```
db.beginTransaction();
try {
  ...
  db.setTransactionSuccessful();
} finally {
  db.endTransaction();
}
```

*public void **beginTransactionNonExclusive** ()*

Since: API Level 11

Begins a transaction in IMMEDIATE mode. Transactions can be nested. When the outer transaction is ended all of the work done in that transaction and all of the nested transactions will be committed or rolled back. The changes will be rolled back if any transaction is ended without being marked as clean (by calling setTransactionSuccessful). Otherwise they will be committed.

Here is the standard idiom for transactions:

```
db.beginTransactionNonExclusive();
try {
  ...
  db.setTransactionSuccessful();
} finally {
  db.endTransaction();
}
```

public void ***beginTransactionWithListener*** *(SQLiteTransactionListener transactionListener)*

Since: API Level 5

Begins a transaction in EXCLUSIVE mode.

Transactions can be nested. When the outer transaction is ended all of the work done in that transaction and all of the nested transactions will be committed or rolled back. The changes will be rolled back if any transaction is ended without being marked as clean (by calling setTransactionSuccessful). Otherwise they will be committed.

Here is the standard idiom for transactions:

```
db.beginTransactionWithListener(listener);
try {
  ...
  db.setTransactionSuccessful();
} finally {
  db.endTransaction();
}
```

Parameters

transactionListener	listener that should be notified when the transaction begins, commits, or is rolled back, either explicitly or by a call to `yieldIfContendedSafely()`.

public void ***beginTransactionWithListenerNonExclusive*** *(SQLiteTransactionListener transactionListener)*

Since: API Level 11

Begins a transaction in IMMEDIATE mode. Transactions can be nested. When the outer transaction is ended all of the work done in that transaction and all of the nested transactions will be committed or rolled back. The changes will be rolled back if any transaction is ended without being marked as clean (by calling setTransactionSuccessful). Otherwise they will be committed.

Here is the standard idiom for transactions:

```
db.beginTransactionWithListenerNonExclusive(listener);
try {
  ...
  db.setTransactionSuccessful();
} finally {
  db.endTransaction();
}
```

Parameters

transactionListener	listener that should be notified when the transaction begins, commits, or is rolled back, either explicitly or by a call to `yieldIfContendedSafely()`.

public void ***close*** *()*

Since: API Level 1

Close the database.

public *SQLiteStatement* ***compileStatement*** *(String sql)*

Since: API Level 1

Compiles an SQL statement into a reusable pre-compiled statement object. The parameters are identical to `execSQL(String)`. You may put ?s in the statement and fill in those values with `bindString(int, String)` and `bindLong(int, long)` each time you want to run the statement. Statements may not return result sets larger than 1x1.

No two threads should be using the same `SQLiteStatement` at the same time.

Parameters

sql The raw SQL statement, may contain ? for unknown values to be bound later.

Returns

- A pre-compiled `SQLiteStatement` object. Note that `SQLiteStatement`s are not synchronized, see the documentation for more details.

Throws

SQLException

public static SQLiteDatabase ***create*** *(SQLiteDatabase.CursorFactory factory)*

Since: API Level 1

Create a memory backed SQLite database. Its contents will be destroyed when the database is closed.

Sets the locale of the database to the the system's current locale. Call `setLocale(Locale)` if you would like something else.

Parameters

factory an optional factory class that is called to instantiate a cursor when query is called

Returns

- a SQLiteDatabase object, or null if the database can't be created

public int ***delete*** *(String table, String whereClause, String[] whereArgs)*

Since: API Level 1

Convenience method for deleting rows in the database.

Parameters

table	the table to delete from
whereClause	the optional WHERE clause to apply when deleting. Passing null will delete all rows.

Returns

- the number of rows affected if a whereClause is passed in, 0 otherwise. To remove all rows and get a count pass "1" as the whereClause.

public boolean ***enableWriteAheadLogging*** *()*

Since: API Level 11

This method enables parallel execution of queries from multiple threads on the same database. It does this by opening multiple handles to the database and using a different database handle for each query.

If a transaction is in progress on one connection handle and say, a table is updated in the transaction, then query on the same table on another connection handle will block for the transaction to complete. But this method enables such queries to execute by having them return old version of the data from the table. Most often it is the data that existed in the table prior to the above transaction updates on that table.

Maximum number of simultaneous handles used to execute queries in parallel is dependent upon the device memory and possibly other properties.

After calling this method, execution of queries in parallel is enabled as long as this database handle is open. To disable execution of queries in parallel, database should be closed and reopened.

If a query is part of a transaction, then it is executed on the same database handle the transaction was begun.

If the database has any attached databases, then execution of queries in paralel is NOT possible. In such cases, a message is printed to logcat and false is returned.

This feature is not available for :memory: databases. In such cases, a message is printed to logcat and false is returned.

A typical way to use this method is the following:

```
     SQLiteDatabase db = SQLiteDatabase.openDatabase("db_filename",
cursorFactory,
             CREATE_IF_NECESSARY, myDatabaseErrorHandler);
     db.enableWriteAheadLogging();
```

Writers should use `beginTransactionNonExclusive()` or `beginTransactionWithListenerNonExclusive(SQ`

`LiteTransactionListener)` to start a trsnsaction. Non-exclusive mode allows database file to be in readable by threads executing queries.

Returns

- true if write-ahead-logging is set. false otherwise

*public void **endTransaction** ()*

Since: API Level 1

End a transaction. See beginTransaction for notes about how to use this and when transactions are committed and rolled back.

*public void **execSQL** (String sql)*

Since: API Level 1

Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.

It has no means to return any data (such as the number of affected rows). Instead, you're encouraged to use `insert(String, String, ContentValues)`, `update(String, ContentValues, String, String[])`, et al, when possible.

When using `enableWriteAheadLogging()`, journal_mode is automatically managed by this class. So, do not set journal_mode using "PRAGMA journal_mode'" statement if your app is using `enableWriteAheadLogging()`

Parameters

sql	the SQL statement to be executed. Multiple statements separated by semicolons are not supported.

Throws

SQLException	if the SQL string is invalid

*public void **execSQL** (String sql, Object[] bindArgs)*

Since: API Level 1

Execute a single SQL statement that is NOT a SELECT/INSERT/UPDATE/DELETE.

For INSERT statements, use any of the following instead.

- `insert(String, String, ContentValues)`
- `insertOrThrow(String, String, ContentValues)`
- `insertWithOnConflict(String, String, ContentValues, int)`

For UPDATE statements, use any of the following instead.

- `update(String, ContentValues, String, String[])`

- `updateWithOnConflict(String, ContentValues, String, String[], int)`

For DELETE statements, use any of the following instead.

- `delete(String, String, String[])`

For example, the following are good candidates for using this method:

- ALTER TABLE
- CREATE or DROP table / trigger / view / index / virtual table
- REINDEX
- RELEASE
- SAVEPOINT
- PRAGMA that returns no data

When using `enableWriteAheadLogging()`, journal_mode is automatically managed by this class. So, do not set journal_mode using "PRAGMA journal_mode'" statement if your app is using `enableWriteAheadLogging()`

Parameters

sql	the SQL statement to be executed. Multiple statements separated by semicolons are not supported.
bindArgs	only byte[], String, Long and Double are supported in bindArgs.

Throws

SQLException	if the SQL string is invalid

public static String ***findEditTable*** *(String tables)*

Since: API Level 1

Finds the name of the first table, which is editable.

Parameters

tables	a list of tables

Returns

- the first table listed

public List<Pair<String, String>> ***getAttachedDbs*** *()*

Since: API Level 11

Returns list of full pathnames of all attached databases including the main database by executing 'pragma database_list' on the database.

Returns

- ArrayList of pairs of (database name, database file path) or null if the database is not open.

public long ***getMaximumSize*** *()*

Since: API Level 1

Returns the maximum size the database may grow to.

Returns

- the new maximum database size

public long ***getPageSize*** *()*

Since: API Level 1

Returns the current database page size, in bytes.

Returns

- the database page size, in bytes

public final String ***getPath*** *()*

Since: API Level 1

Getter for the path to the database file.

Returns

- the path to our database file.

public Map<String, String> ***getSyncedTables*** *()*

Since: API Level 1

This method is deprecated.

This method no longer serves any useful purpose and has been deprecated.

public int ***getVersion*** *()*

Since: API Level 1

Gets the database version.

Returns

- the database version

public boolean ***inTransaction*** *()*

Since: API Level 1

return true if there is a transaction pending

public long ***insert*** *(String table, String nullColumnHack, ContentValues values)*

Since: API Level 1

Convenience method for inserting a row into the database.

Parameters

table	the table to insert the row into
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `values` is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your `values` is empty.
values	this map contains the initial column values for the row. The keys should be the column names and the values the column values

Returns

- the row ID of the newly inserted row, or -1 if an error occurred

public long ***insertOrThrow*** *(String table, String nullColumnHack, ContentValues values)*

Since: API Level 1

Convenience method for inserting a row into the database.

Parameters

table	the table to insert the row into
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `values` is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your `values` is empty.
values	this map contains the initial column values for the row. The keys should be the column names and the values the column values

Returns

- the row ID of the newly inserted row, or -1 if an error occurred

Throws

SQLException

[SQLException]

public long ***insertWithOnConflict*** *([String] table, [String] nullColumnHack, [ContentValues] initialValues, int conflictAlgorithm)*

Since: API Level 8

General method for inserting a row into the database.

Parameters

table	the table to insert the row into
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `initialValues`is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your `initialValues` is empty.
initialValues	this map contains the initial column values for the row. The keys should be the column names and the values the column values
conflictAlgorithm	for insert conflict resolver

Returns

- the row ID of the newly inserted row OR the primary key of the existing row if the input param 'conflictAlgorithm' = [`CONFLICT_IGNORE`] OR -1 if any error

public boolean ***isDatabaseIntegrityOk*** *()*

Since: API Level 11

Runs 'pragma integrity_check' on the given database (and all the attached databases) and returns true if the given database (and all its attached databases) pass integrity_check, false otherwise.

If the result is false, then this method logs the errors reported by the integrity_check command execution.

Note that 'pragma integrity_check' on a database can take a long time.

Returns

- true if the given database (and all its attached databases) pass integrity_check, false otherwise.

public boolean ***isDbLockedByCurrentThread*** *()*

Since: API Level 1

Checks if the database lock is held by this thread.

Returns

- true, if this thread is holding the database lock.

public boolean ***isDbLockedByOtherThreads*** *()*

Since: API Level 1

Checks if the database is locked by another thread. This is just an estimate, since this status can change at any time, including after the call is made but before the result has been acted upon.

Returns

- true, if the database is locked by another thread

public boolean ***isOpen*** *()*

Since: API Level 1

Returns

- true if the DB is currently open (has not been closed)

public boolean ***isReadOnly*** *()*

Since: API Level 1

return whether the DB is opened as read only.

Returns

- true if DB is opened as read only

public void ***markTableSyncable*** *(String table, String foreignKey, String updateTable)*

Since: API Level 1

> **This method is deprecated.**
> This method no longer serves any useful purpose and has been deprecated.

Mark this table as syncable, with the _sync_dirty residing in another table. When an update occurs in this table the _sync_dirty field of the row in updateTable with the _id in foreignKey will be set to ensure proper syncing operation.

Parameters

table	an update on this table will trigger a sync time removal

foreignKey	this is the column in table whose value is an _id in updateTable
updateTable	this is the table that will have its _sync_dirty

public void ***markTableSyncable*** *(String table, String deletedTable)*

Since: API Level 1

This method is deprecated.

This method no longer serves any useful purpose and has been deprecated.

Mark this table as syncable. When an update occurs in this table the _sync_dirty field will be set to ensure proper syncing operation.

Parameters

table	the table to mark as syncable
deletedTable	The deleted table that corresponds to the syncable table

public boolean ***needUpgrade*** *(int newVersion)*

Since: API Level 1

public static SQLiteDatabase ***openDatabase*** *(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler)*

Since: API Level 11

Open the database according to the flags `OPEN_READWRITE` `OPEN_READONLY` `CREATE_IF_NECESSARY` and/or `NO_LOCALIZED_COLLATORS`.

Sets the locale of the database to the the system's current locale. Call `setLocale(Locale)` if you would like something else.

Accepts input param: a concrete instance of `DatabaseErrorHandler` to be used to handle corruption when sqlite reports database corruption.

Parameters

path	to database file to open and/or create
factory	an optional factory class that is called to instantiate a cursor when query is called, or null for default
flags	to control database access mode
errorHandler	the `DatabaseErrorHandler` obj to be used to handle corruption when sqlite reports database corruption

Returns

- the newly opened database

Throws

SQLiteException	if the database cannot be opened

public static SQLiteDatabase ***openDatabase*** *(String path, SQLiteDatabase.CursorFactory factory, int flags)*

Since: API Level 1

Open the database according to the flags `OPEN_READWRITE` `OPEN_READONLY` `CREATE_IF_NECESSARY` and/or `NO_LOCALIZED_COLLATORS`.

Sets the locale of the database to the the system's current locale. Call `setLocale(Locale)` if you would like something else.

Parameters

path	to database file to open and/or create
factory	an optional factory class that is called to instantiate a cursor when query is called, or null for default
flags	to control database access mode

Returns

- the newly opened database

Throws

SQLiteException	if the database cannot be opened

public static SQLiteDatabase ***openOrCreateDatabase*** *(String path, SQLiteDatabase.CursorFactory factory, DatabaseErrorHandler errorHandler)*

Since: API Level 11

Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY, errorHandler).

public static SQLiteDatabase ***openOrCreateDatabase*** *(String path, SQLiteDatabase.CursorFactory factory)*

Since: API Level 1

Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY).

public static SQLiteDatabase ***openOrCreateDatabase*** *(File file, SQLiteDatabase.CursorFactory factory)*

Since: API Level 1

Equivalent to openDatabase(file.getPath(), factory, CREATE_IF_NECESSARY).

public Cursor ***query*** *(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)*

Since: API Level 1

Query the given table, returning a `Cursor` over the result set.

Parameters

table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table.
selectionArgs	You may include ?s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.
groupBy	A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped.
having	A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used.
orderBy	How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.
limit	Limits the number of rows returned by the query, formatted as LIMIT clause. Passing null denotes no LIMIT clause.

Returns

- A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

See Also

- Cursor

*public Cursor **query** (String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)*

Since: API Level 1

Query the given table, returning a Cursor over the result set.

Parameters

table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table.
selectionArgs	You may include ?s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.
groupBy	A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped.
having	A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used.
orderBy	How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.

Returns

- A Cursor object, which is positioned before the first entry. Note that Cursors are not synchronized, see the documentation for more details.

See Also

- Cursor

*public Cursor **query** (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)*

Since: API Level 1

Query the given URL, returning a `Cursor` over the result set.

Parameters

distinct	true if you want each row to be unique, false otherwise.
table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table.
selectionArgs	You may include ?s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.
groupBy	A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped.
having	A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used.
orderBy	How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.
limit	Limits the number of rows returned by the query, formatted as LIMIT clause. Passing null denotes no LIMIT clause.

Returns

- A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

See Also

- `Cursor`

public Cursor ***queryWithFactory*** *(SQLiteDatabase.CursorFactory cursorFactory, boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, StringgroupBy, String having, String orderBy, String limit)*

Since: API Level 1

Query the given URL, returning a `Cursor` over the result set.

Parameters

cursorFactory	the cursor factory to use, or null for the default factory
distinct	true if you want each row to be unique, false otherwise.
table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table.
selectionArgs	You may include ?s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.
groupBy	A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped.
having	A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used.
orderBy	How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.
limit	Limits the number of rows returned by the query, formatted as LIMIT clause. Passing null denotes no LIMIT clause.

Returns

- A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

See Also

- `Cursor`

public Cursor ***rawQuery*** *(String sql, String[] selectionArgs)*

Since: API Level 1

Runs the provided SQL and returns a `Cursor` over the result set.

Parameters

sql	the SQL query. The SQL string must not be ; terminated
selectionArgs	You may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.

Returns

- A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

public Cursor ***rawQueryWithFactory*** *(SQLiteDatabase.CursorFactory cursorFactory, String sql, String[] selectionArgs, String editTable)*

Since: API Level 1

Runs the provided SQL and returns a cursor over the result set.

Parameters

cursorFactory	the cursor factory to use, or null for the default factory
sql	the SQL query. The SQL string must not be ; terminated
selectionArgs	You may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.
editTable	the name of the first table, which is editable

Returns

- A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

public static int ***releaseMemory*** *()*

Since: API Level 1

Attempts to release memory that SQLite holds but does not require to operate properly. Typically this memory will come from the page cache.

Returns

- the number of bytes actually released

public long ***replace*** *(String table, String nullColumnHack, ContentValues initialValues)*

Since: API Level 1

Convenience method for replacing a row in the database.

Parameters

table	the table in which to replace the row
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `initialValues`is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your `initialValues` is empty.
initialValues	this map contains the initial column values for the row.

Returns

- the row ID of the newly inserted row, or -1 if an error occurred

public long ***replaceOrThrow*** *(String table, String nullColumnHack, ContentValues initialValues)*

Since: API Level 1

Convenience method for replacing a row in the database.

Parameters

table	the table in which to replace the row
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `initialValues`is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your `initialValues` is empty.
initialValues	this map contains the initial column values for the row. The key

Returns

- the row ID of the newly inserted row, or -1 if an error occurred

Throws

SQLException

SQLException

public void **setLocale** (Locale locale)

Since: API Level 1

Sets the locale for this database. Does nothing if this database has the NO_LOCALIZED_COLLATORS flag set or was opened read only.

Throws

SQLException	if the locale could not be set. The most common reason for this is that there is no collator available for the locale you requested. In this case the database remains unchanged.

public void **setLockingEnabled** (boolean lockingEnabled)

Since: API Level 1

Control whether or not the SQLiteDatabase is made thread-safe by using locks around critical sections. This is pretty expensive, so if you know that your DB will only be used by a single thread then you should set this to false. The default is true.

Parameters

lockingEnabled	set to true to enable locks, false otherwise

public void **setMaxSqlCacheSize** (int cacheSize)

Since: API Level 11

Sets the maximum size of the prepared-statement cache for this database. (size of the cache = number of compiled-sql-statements stored in the cache).

Maximum cache size can ONLY be increased from its current size (default = 10). If this method is called with smaller size than the current maximum value, then IllegalStateException is thrown.

This method is thread-safe.

Parameters

cacheSize	the size of the cache. can be (0 to `MAX_SQL_CACHE_SIZE`)

Throws

IllegalStateException	if input cacheSize > `MAX_SQL_CACHE_SIZE` or the value set with previous

setMaxSqlCacheSize() call.

public long ***setMaximumSize*** *(long numBytes)*

Since: API Level 1

Sets the maximum size the database will grow to. The maximum size cannot be set below the current size.

Parameters

numBytes	the maximum database size, in bytes

Returns

- the new maximum database size

public void ***setPageSize*** *(long numBytes)*

Since: API Level 1

Sets the database page size. The page size must be a power of two. This method does not work if any data has been written to the database file, and must be called right after the database has been created.

Parameters

numBytes	the database page size, in bytes

public void ***setTransactionSuccessful*** *()*

Since: API Level 1

Marks the current transaction as successful. Do not do any more database work between calling this and calling endTransaction. Do as little non-database work as possible in that situation too. If any errors are encountered between this and endTransaction the transaction will still be committed.

Throws

IllegalStateException	if the current thread is not in a transaction or the transaction is already marked as successful.

public void ***setVersion*** *(int version)*

Since: API Level 1

Sets the database version.

Parameters

version	the new database version

public int **update** *(String table, ContentValues values, String whereClause, String[] whereArgs)*

Since: API Level 1

Convenience method for updating rows in the database.

Parameters

table	the table to update in
values	a map from column names to new column values. null is a valid value that will be translated to NULL.
whereClause	the optional WHERE clause to apply when updating. Passing null will update all rows.

Returns

- the number of rows affected