Android APIs API level: 17

android.content
android.content.pm
android.content.res
android.database
android.database.sqlite
android.drm
android.gesture
android.graphics
android.graphics.drawa
android.graphics.drawa
android.hardware
android.hardware.displ
android.hardware.inpu
android.hardware.usb

SQLiteCursor
SQLiteDatabase
SQLiteOpenHelper
SQLiteProgram

Use Tree Navigation

Summary: Nested Classes | Constants | Methods | Protected Methods | Inherited Methods | [Expand All]

public final class

SQLiteDatabase

extends SQLiteClosable

Added in API level 1

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, 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 changes from prior commands within the same transaction are preserved.
int	CONFLICT_FAIL	When a constraint violation occurs, the command aborts with a return code SQLITE_CONSTRAINT.
int	CONFLICT_IGNORE	When a constraint violation occurs, the one row that contains the constraint violation is 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 are causing the constraint violation are removed prior to inserting or updating the current row.
int	CONFLICT_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.
int	CREATE_IF_NECESSARY	Open flag: Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to create the database file if it does not already exist.
int	ENABLE_WRITE_AHEAD_LOGGING	Open flag: Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database file with write-ahead logging enabled by default.
int	MAX_SQL_CACHE_SIZE	Absolute max value that can be set by `setMaxSqlCacheSize(int)`.
int	NO_LOCALIZED_COLLATORS	Open flag: Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database without support for localized collators.
int	OPEN_READONLY	Open flag: Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database for reading only.
int	OPEN_READWRITE	Open flag: 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.
int	SQLITE_MAX_LIKE_PATTERN_LENGTH	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.

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.
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.

static boolean	deleteDatabase(File file) Deletes a database including its journal file and other auxiliary files that may have been created by the database engine.
void	disableWriteAheadLogging() This method disables the features enabled by `enableWriteAheadLogging()`.
boolean	enableWriteAheadLogging() This method enables parallel execution of queries from multiple threads on the same database.
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 returns data.
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 executing 'pragma database_list' on the database.
long	getMaximumSize() Returns the maximum size the database may grow to.
long	getPageSize() Returns the current database page size, in bytes.
final String	getPath() Gets the path to the database file.
Map<String, String>	getSyncedTables() *This method was deprecated in API level 11. This method no longer serves any useful purpose and has been deprecated.*
int	getVersion() Gets the database version.
boolean	inTransaction() Returns true if the current thread has 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 conflictAlgorithm) 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 returns true if the given database (and all its attached databases) pass integrity_check, false otherwise.
boolean	isDbLockedByCurrentThread() Returns true if the current thread is holding an active connection to the database.
boolean	isDbLockedByOtherThreads() *This method was deprecated in API level 16. Always returns false. Do not use this method.*
boolean	isOpen() Returns true if the database is currently open.
boolean	isReadOnly() Returns true if the database is opened as read only.
boolean	isWriteAheadLoggingEnabled() Returns true if write-ahead logging has been enabled for this database.
void	markTableSyncable(String table, String foreignKey, String updateTable) *This method was deprecated in API level 11. This method no longer serves any useful purpose and has been deprecated.*
void	markTableSyncable(String table, String deletedTable) *This method was deprecated in API level 11. This method no longer serves any useful purpose and has been deprecated.*
boolean	needUpgrade(int newVersion) Returns true if the new version code is greater than the current database version.
static SQLiteDatabase	openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler) Open the database according to the flags `OPEN_READWRITE` `OPEN_READONLY` `CREATE_IF_NECESSARY` and/or `NO_LOCALIZED_COLLATORS`.
static SQLiteDatabase	openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags) Open the database according to the flags `OPEN_READWRITE` `OPEN_READONLY` `CREATE_IF_NECESSARY` and/or `NO_LOCALIZED_COLLATORS`.
static SQLiteDatabase	openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory, DatabaseErrorHandler errorHandler) 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, String having, String orderBy, String limit) Query the given table, returning a `Cursor` over the result set.
Cursor	query(boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit, CancellationSignal cancellationSignal) Query the given URL, returning a `Cursor` over the result set.
Cursor	query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy) Query the given table, returning a `Cursor` over the result set.
Cursor	query(boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit) 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 groupBy, String having, String orderBy, String limit, CancellationSignal cancellationSignal) 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 groupBy, String having, String orderBy, String limit) Query the given URL, returning a `Cursor` over the result set.
Cursor	rawQuery(String sql, String[] selectionArgs, CancellationSignal cancellationSignal) Runs the provided SQL and returns 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[] selectionArgs, String editTable) Runs the provided SQL and returns a cursor over the result set.
Cursor	rawQueryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, String sql, String[] selectionArgs, String editTable, CancellationSignal cancellationSignal) 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	setForeignKeyConstraintsEnabled(boolean enable) Sets whether foreign key constraints are enabled for the database.
void	setLocale(Locale locale) Sets the locale for this database.
void	setLockingEnabled(boolean lockingEnabled) *This method was deprecated in API level 16. This method now does nothing. Do not use.*
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.
String	toString() Returns a string containing a concise, human-readable description of this object.
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[] whereArgs, int conflictAlgorithm) Convenience method for updating rows in the database.
boolean	yieldIfContended() *This method was deprecated in API level 3. if the db is locked more than once (becuase of nested transactions) then the lock will not be yielded. Use 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() Called when the last reference to the object was released by a call to `releaseReference()` or `close()`.

Inherited Methods [Expand]

- From class android.database.sqlite.SQLiteClosable
- From class java.lang.Object
- From interface java.io.Closeable

Constants

public static final int **CONFLICT_ABORT**
Added in 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**
Added in 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**
Added in 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**
Added in API level 8

Use the following when no conflict action is specified.

Constant Value: 0 (0x00000000)

public static final int **CONFLICT_REPLACE**
Added in 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**
Added in 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**
Added in API level 1

Open flag: 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 **ENABLE_WRITE_AHEAD_LOGGING**
Added in API level 16

Open flag: Flag for `openDatabase(String, SQLiteDatabase.CursorFactory, int)` to open the database file with write-ahead logging enabled by default. Using this flag is more efficient than calling `enableWriteAheadLogging()`. Write-ahead logging cannot be used with read-only databases so the value of this flag is ignored if the database is opened read-only.

See Also
`enableWriteAheadLogging()`

Constant Value: 536870912 (0x20000000)

public static final int **MAX_SQL_CACHE_SIZE**
Added in API level 11

Absolute max value that can be set by `setMaxSqlCacheSize(int)`. Each prepared-statement is between 1K - 6K, depending on the complexity of the SQL statement & schema. A large SQL cache may use a significant amount of memory.

Constant Value: 100 (0x00000064)

public static final int **NO_LOCALIZED_COLLATORS**
Added in API level 1

Open flag: 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**
Added in API level 1

Open flag: 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**

Added in API level 1

Open flag: 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**

Added in 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** ()

Added in 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** ()

Added in 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)

Added in 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)

Added in 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 SQLiteStatement **compileStatement** (String sql)

Added in 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)

Added in 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)

Added in 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 static boolean **deleteDatabase** (File file)

Added in API level 16

Deletes a database including its journal file and other auxiliary files that may have been created by the database engine.

Parameters

file The database file path.

Returns

True if the database was successfully deleted.

public void **disableWriteAheadLogging** ()

Added in API level 16

This method disables the features enabled by `enableWriteAheadLogging()`.

Throws

IllegalStateException if there are transactions in progress at the time this method is called. WAL mode can only be changed when there are no transactions in progress.

See Also

`enableWriteAheadLogging()`

public boolean **enableWriteAheadLogging** ()

Added in API level 11

This method enables parallel execution of queries from multiple threads on the same database. It does this by opening multiple connections to the database and using a different database connection for each query. The database journal mode is also changed to enable writes to proceed concurrently with reads.

When write-ahead logging is not enabled (the default), it is not possible for reads and writes to occur on the database at the same time. Before modifying the database, the writer implicitly acquires an exclusive lock on the database which prevents readers from accessing the database until the write is completed.

In contrast, when write-ahead logging is enabled (by calling this method), write operations occur in a separate log file which allows reads to proceed concurrently. While a write is in progress, readers on other threads will perceive the state of the database as it was before the write began. When the write completes, readers on other threads will then perceive the new state of the database.

It is a good idea to enable write-ahead logging whenever a database will be concurrently accessed and modified by multiple threads at the same time. However, write-ahead logging uses significantly more memory than ordinary journaling because there are multiple connections to the same database. So if a database will only be used by a single thread, or if optimizing concurrency is not very important, then write-ahead logging should be disabled.

After calling this method, execution of queries in parallel is enabled as long as the database remains open. To disable execution of queries in parallel, either call `disableWriteAheadLogging()` or close the database and reopen it.

The maximum number of connections used to execute queries in parallel is dependent upon the device memory and possibly other properties.

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

Writers should use `beginTransactionNonExclusive()` or `beginTransactionWithListenerNonExclusive(SQLiteTransactionListener)` to start a transaction. Non-exclusive mode allows database file to be in readable by other threads executing queries.

If the database has any attached databases, then execution of queries in parallel is NOT possible. Likewise, write-ahead logging is not supported for read-only databases or memory databases. In such cases, `enableWriteAheadLogging()` returns false.

The best way to enable write-ahead logging is to pass the `ENABLE_WRITE_AHEAD_LOGGING` flag to `openDatabase(String, SQLiteDatabase.CursorFactory, int)`. This is more efficient than calling `enableWriteAheadLogging()`.

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

Another way to enable write-ahead logging is to call `enableWriteAheadLogging()` after opening the database.

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

See also SQLite Write-Ahead Logging for more details about how write-ahead logging works.

Returns
True if write-ahead logging is enabled.

Throws

IllegalStateException	if there are transactions in progress at the time this method is called. WAL mode can only be changed when there are no transactions in progress.

See Also
`ENABLE_WRITE_AHEAD_LOGGING`
`disableWriteAheadLogging()`

public void **endTransaction** ()

Added in 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)

Added in 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)

Added in 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) Added in 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** () Added in 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** () Added in API level 1

Returns the maximum size the database may grow to.

Returns

the new maximum database size

public long **getPageSize** () Added in API level 1

Returns the current database page size, in bytes.

Returns

the database page size, in bytes

public final String **getPath** () Added in API level 1

Gets the path to the database file.

Returns

The path to the database file.

public Map<String, String> **getSyncedTables** () Added in API level 1

This method was deprecated in API level 11.
This method no longer serves any useful purpose and has been deprecated.

Deprecated.

public int **getVersion** () Added in API level 1

Gets the database version.

Returns

the database version

public boolean **inTransaction** () Added in API level 1

Returns true if the current thread has a transaction pending.

Returns
True if the current thread is in a transaction.

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

Added in 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)

Added in 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)

Added in 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** ()

Added in 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** ()

Added in API level 1

Returns true if the current thread is holding an active connection to the database.

The name of this method comes from a time when having an active connection to the database meant that the thread was holding an actual lock on the database. Nowadays, there is no longer a true "database lock" although threads may block if they cannot acquire a database connection to perform a particular operation.

Returns
True if the current thread is holding an active connection to the database.

public boolean **isDbLockedByOtherThreads** ()

Added in API level 1

This method was deprecated in API level 16.
Always returns false. Do not use this method.

Always returns false.

There is no longer the concept of a database lock, so this method always returns false.

Returns
False.

public boolean **isOpen** () Added in API level 1

Returns true if the database is currently open.

Returns
True if the database is currently open (has not been closed).

public boolean **isReadOnly** () Added in API level 1

Returns true if the database is opened as read only.

Returns
True if database is opened as read only.

public boolean **isWriteAheadLoggingEnabled** () Added in API level 16

Returns true if write-ahead logging has been enabled for this database.

Returns
True if write-ahead logging has been enabled for this database.

See Also
`enableWriteAheadLogging()`
`ENABLE_WRITE_AHEAD_LOGGING`

public void **markTableSyncable** (String table, String foreignKey, String updateTable) Added in API level 1

This method was deprecated in API level 11.
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) Added in API level 1

This method was deprecated in API level 11.
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) Added in API level 1

Returns true if the new version code is greater than the current database version.

Parameters

newVersion	The new version code.

Returns
True if the new version code is greater than the current database version.

public static SQLiteDatabase **openDatabase** (String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler) Added in 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) Added in 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) Added in API level 11

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

public static SQLiteDatabase **openOrCreateDatabase** (String path, SQLiteDatabase.CursorFactory factory) Added in API level 1

Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY).

public static SQLiteDatabase **openOrCreateDatabase** (File file, SQLiteDatabase.CursorFactory factory) Added in 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) Added in 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** (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit, CancellationSignal cancellationSignal) Added in API level 16

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.
cancellationSignal	A signal to cancel the operation in progress, or null if none. If the operation is canceled, then `OperationCanceledException` will be thrown when the query is executed.

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) Added in 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 `Cursor`s 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) Added in 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, String groupBy, String having, String orderBy, String limit, CancellationSignal cancellationSignal) Added in API level 16

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.
cancellationSignal	A signal to cancel the operation in progress, or null if none. If the operation is canceled, then `OperationCanceledException` will be thrown when the query is executed.

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, String groupBy, String having, String orderBy, String limit) Added in 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, CancellationSignal cancellationSignal) Added in API level 16

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.
cancellationSignal	A signal to cancel the operation in progress, or null if none. If the operation is canceled, then `OperationCanceledException` will be thrown when the query is executed.

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 **rawQuery** (String sql, String[] selectionArgs) Added in 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) Added in 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 Cursor **rawQueryWithFactory** (SQLiteDatabase.CursorFactory cursorFactory, String sql, String[] selectionArgs, String editTable, CancellationSignal cancellationSignal) Added in API level 16

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
cancellationSignal	A signal to cancel the operation in progress, or null if none. If the operation is canceled, then `OperationCanceledException` will be thrown when the query is executed.

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** () Added in 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) Added in 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)

Added in 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 **setForeignKeyConstraintsEnabled** (boolean enable)

Added in API level 16

Sets whether foreign key constraints are enabled for the database.

By default, foreign key constraints are not enforced by the database. This method allows an application to enable foreign key constraints. It must be called each time the database is opened to ensure that foreign key constraints are enabled for the session.

A good time to call this method is right after calling `openOrCreateDatabase(File, SQLiteDatabase.CursorFactory)` or in the `onConfigure(SQLiteDatabase)` callback.

When foreign key constraints are disabled, the database does not check whether changes to the database will violate foreign key constraints. Likewise, when foreign key constraints are disabled, the database will not execute cascade delete or update triggers. As a result, it is possible for the database state to become inconsistent. To perform a database integrity check, call `isDatabaseIntegrityOk()`.

This method must not be called while a transaction is in progress.

See also SQLite Foreign Key Constraints for more details about foreign key constraint support.

Parameters

enable	True to enable foreign key constraints, false to disable them.

Throws

IllegalStateException	if the are transactions is in progress when this method is called.

public void **setLocale** (Locale locale)

Added in 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.

Parameters

locale	The new locale.

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)

Added in API level 1

This method was deprecated in API level 16.
This method now does nothing. Do not use.

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)

Added in 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`.

public long **setMaximumSize** (long numBytes)

Added in 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)

Added in 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** ()

Added in 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)

Added in API level 1

Sets the database version.

Parameters

version	the new database version

public String **toString** ()

Added in API level 1

Returns a string containing a concise, human-readable description of this object. Subclasses are encouraged to override this method and provide an implementation that takes into account the object's type and data. The default implementation is equivalent to the following expression:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

See Writing a useful `toString` method if you intend implementing your own `toString` method.

Returns
a printable representation of this object.

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

Added in 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

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

Added in API level 8

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.
conflictAlgorithm	for update conflict resolver

Returns
the number of rows affected

public boolean **yieldIfContended** ()

Added in API level 1

This method was deprecated in API level 3.
if the db is locked more than once (becuase of nested transactions) then the lock will not be yielded. Use yieldIfContendedSafely instead.

Temporarily end the transaction to let other threads run. The transaction is assumed to be successful so far. Do not call setTransactionSuccessful before calling this. When this returns a new transaction will have been created but not marked as successful.

Returns
true if the transaction was yielded

public boolean **yieldIfContendedSafely** (long sleepAfterYieldDelay)

Added in API level 5

Temporarily end the transaction to let other threads run. The transaction is assumed to be successful so far. Do not call setTransactionSuccessful before calling this. When this returns a new transaction will have been created but not marked as successful. This assumes that there are no nested transactions (beginTransaction has only been called once) and will throw an exception if that is not the case.

Parameters

sleepAfterYieldDelay	if > 0, sleep this long before starting a new transaction if the lock was actually yielded. This will allow other background threads to make some more progress than they would if we started the transaction immediately.

Returns

true if the transaction was yielded

public boolean **yieldIfContendedSafely** () Added in API level 3

Temporarily end the transaction to let other threads run. The transaction is assumed to be successful so far. Do not call setTransactionSuccessful before calling this. When this returns a new transaction will have been created but not marked as successful. This assumes that there are no nested transactions (beginTransaction has only been called once) and will throw an exception if that is not the case.

Returns

true if the transaction was yielded

Protected Methods

protected void **finalize** () Added in API level 1

Invoked when the garbage collector has detected that this instance is no longer reachable. The default implementation does nothing, but this method can be overridden to free resources.

Note that objects that override `finalize` are significantly more expensive than objects that don't. Finalizers may be run a long time after the object is no longer reachable, depending on memory pressure, so it's a bad idea to rely on them for cleanup. Note also that finalizers are run on a single VM-wide finalizer thread, so doing blocking work in a finalizer is a bad idea. A finalizer is usually only necessary for a class that has a native peer and needs to call a native method to destroy that peer. Even then, it's better to provide an explicit `close` method (and implement `Closeable`), and insist that callers manually dispose of instances. This works well for something like files, but less well for something like a `BigInteger` where typical calling code would have to deal with lots of temporaries. Unfortunately, code that creates lots of temporaries is the worst kind of code from the point of view of the single finalizer thread.

If you *must* use finalizers, consider at least providing your own `ReferenceQueue` and having your own thread process that queue.

Unlike constructors, finalizers are not automatically chained. You are responsible for calling `super.finalize()` yourself.

Uncaught exceptions thrown by finalizers are ignored and do not terminate the finalizer thread. See *Effective Java* Item 7, "Avoid finalizers" for more.

Throws

Throwable

protected void **onAllReferencesReleased** () Added in API level 1

Called when the last reference to the object was released by a call to `releaseReference()` or `close()`.