C-language Interface Specification for SQLite

This page is intended to be a precise and detailed specification. For a tutorial introductions, see instead:

- SQLite In 3 Minutes Or Less and/or
- the Introduction To The SQLite C/C++ Interface.

This same content is also available split out into lots of small pages.

Experimental And Deprecated Interfaces

SQLite interfaces can be subdivided into three categories:

1. Stable
2. Experimental
3. Deprecated

Stable interfaces will be maintained indefinitely in a backwards compatible way. An application that uses only stable interfaces should always be able to relink against a newer version of SQLite without any changes.

Experimental interfaces are subject to change. Applications that use experimental interfaces may need to be modified when upgrading to a newer SQLite release, though this is rare. When new interfaces are added to SQLite, they generally begin as experimental interfaces. After an interface has been in use for a while and the developers are confident that the design of the interface is sound and worthy of long-term support, the interface is marked as stable.

Deprecated interfaces have been superceded by better methods of accomplishing the same thing and should be avoided in new applications. Deprecated interfaces continue to be supported for the sake of backwards compatibility. At some point in the future, it is possible that deprecated interfaces may be removed.

Key points:

- Experimental interfaces are subject to change and/or removal at any time.
- Deprecated interfaces should not be used in new code and might be removed in some future release.

List Of Objects:

- sqlite3
- sqlite3_api_routines
- sqlite3_backup
- sqlite3_blob
- sqlite3_context
- sqlite3_data_directory
- sqlite3_file
- sqlite3_index_info
- sqlite3_int64
- sqlite3_uint64
- sqlite_int64
- sqlite_uint64
- sqlite3_io_methods
- sqlite3_mem_methods
- sqlite3_module
- sqlite3_mutex
- sqlite3_mutex_methods
- sqlite3_pcache
- sqlite3_pcache_methods2
- sqlite3_pcache_page
- sqlite3_stmt
- sqlite3_str
- sqlite3_temp_directory
- sqlite3_value
- sqlite3_vfs
- sqlite3_vtab
- sqlite3_vtab_cursor

List Of Constants:

Also available: list of error codes

- SQLITE_ABORT
- SQLITE_ABORT_ROLLBACK
- SQLITE_ACCESS_EXISTS
- SQLITE_ACCESS_READ
- SQLITE_ACCESS_READWRITE
- SQLITE_ALTER_TABLE
- SQLITE_ANALYZE
- SQLITE_ANY
- SQLITE_ATTACH
- SQLITE_AUTH
- SQLITE_AUTH_USER
- SQLITE_BLOB
- SQLITE_BUSY
- SQLITE_BUSY_RECOVERY
- SQLITE_BUSY_SNAPSHOT
- SQLITE_CANTOPEN
- SQLITE_CANTOPEN_CONVPATH
- SQLITE_CANTOPEN_DIRTYWAL
- SQLITE_CANTOPEN_FULLPATH
- SQLITE_CANTOPEN_ISDIR
- SQLITE_FCNTL_FILE_POINTER
- SQLITE_FCNTL_GET_LOCKPROXYFILE
- SQLITE_FCNTL_HAS_MOVED
- SQLITE_FCNTL_JOURNAL_POINTER
- SQLITE_FCNTL_LAST_ERRNO
- SQLITE_FCNTL_LOCKSTATE
- SQLITE_FCNTL_LOCK_TIMEOUT
- SQLITE_FCNTL_MMAP_SIZE
- SQLITE_FCNTL_OVERWRITE
- SQLITE_FCNTL_PDB
- SQLITE_FCNTL_PERSIST_WAL
- SQLITE_FCNTL_POWERSAFE_OVERWRITE
- SQLITE_FCNTL_PRAGMA
- SQLITE_FCNTL_RBU
- SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE
- SQLITE_FCNTL_SET_LOCKPROXYFILE
- SQLITE_FCNTL_SIZE_HINT
- SQLITE_FCNTL_SIZE_LIMIT
- SQLITE_FCNTL_SYNC
- SQLITE_FCNTL_SYNC_OMITTED
- SQLITE_NOTFOUND
- SQLITE_NOTICE
- SQLITE_NOTICE_RECOVER_ROL
- SQLITE_NOTICE_RECOVER_WAL
- SQLITE_NULL
- SQLITE_OK
- SQLITE_OK_LOAD_PERMANENTL
- SQLITE_OK_SYMLINK
- SQLITE_OPEN_AUTOPROXY
- SQLITE_OPEN_CREATE
- SQLITE_OPEN_DELETEONCLOSE
- SQLITE_OPEN_EXCLUSIVE
- SQLITE_OPEN_FULLMUTEX
- SQLITE_OPEN_MAIN_DB
- SQLITE_OPEN_MAIN_JOURNAL
- SQLITE_OPEN_MASTER_JOURNA
- SQLITE_OPEN_MEMORY
- SQLITE_OPEN_NOFOLLOW
- SQLITE_OPEN_NOMUTEX
- SQLITE_OPEN_PRIVATECACHE

- SQLITE_CANTOPEN_NOTEMPDIR
- SQLITE_CANTOPEN_SYMLINK
- SQLITE_CHECKPOINT_FULL
- SQLITE_CHECKPOINT_PASSIVE
- SQLITE_CHECKPOINT_RESTART
- SQLITE_CHECKPOINT_TRUNCATE
- SQLITE_CONFIG_COVERING_INDEX_SCAN
- SQLITE_CONFIG_GETMALLOC
- SQLITE_CONFIG_GETMUTEX
- SQLITE_CONFIG_GETPCACHE
- SQLITE_CONFIG_GETPCACHE2
- SQLITE_CONFIG_HEAP
- SQLITE_CONFIG_LOG
- SQLITE_CONFIG_LOOKASIDE
- SQLITE_CONFIG_MALLOC
- SQLITE_CONFIG_MEMDB_MAXSIZE
- SQLITE_CONFIG_MEMSTATUS
- SQLITE_CONFIG_MMAP_SIZE
- SQLITE_CONFIG_MULTITHREAD
- SQLITE_CONFIG_MUTEX
- SQLITE_CONFIG_PAGECACHE
- SQLITE_CONFIG_PCACHE
- SQLITE_CONFIG_PCACHE2
- SQLITE_CONFIG_PCACHE_HDRSZ
- SQLITE_CONFIG_PMASZ
- SQLITE_CONFIG_SCRATCH
- SQLITE_CONFIG_SERIALIZED
- SQLITE_CONFIG_SINGLETHREAD
- SQLITE_CONFIG_SMALL_MALLOC
- SQLITE_CONFIG_SORTERREF_SIZE
- SQLITE_CONFIG_SQLLOG
- SQLITE_CONFIG_STMTJRNL_SPILL
- SQLITE_CONFIG_URI
- SQLITE_CONFIG_WIN32_HEAPSIZE
- SQLITE_CONSTRAINT
- SQLITE_CONSTRAINT_CHECK
- SQLITE_CONSTRAINT_COMMITHOOK
- SQLITE_CONSTRAINT_FOREIGNKEY
- SQLITE_CONSTRAINT_FUNCTION
- SQLITE_CONSTRAINT_NOTNULL
- SQLITE_CONSTRAINT_PINNED
- SQLITE_CONSTRAINT_PRIMARYKEY
- SQLITE_CONSTRAINT_ROWID
- SQLITE_CONSTRAINT_TRIGGER
- SQLITE_CONSTRAINT_UNIQUE
- SQLITE_CONSTRAINT_VTAB
- SQLITE_COPY
- SQLITE_CORRUPT
- SQLITE_CORRUPT_SEQUENCE
- SQLITE_CORRUPT_VTAB
- SQLITE_CREATE_INDEX
- SQLITE_CREATE_TABLE
- SQLITE_CREATE_TEMP_INDEX
- SQLITE_CREATE_TEMP_TABLE
- SQLITE_CREATE_TEMP_TRIGGER
- SQLITE_CREATE_TEMP_VIEW
- SQLITE_CREATE_TRIGGER
- SQLITE_CREATE_VIEW
- SQLITE_CREATE_VTABLE
- SQLITE_DBCONFIG_DEFENSIVE
- SQLITE_DBCONFIG_DQS_DDL
- SQLITE_DBCONFIG_DQS_DML
- SQLITE_DBCONFIG_ENABLE_FKEY
- SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER
- SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION
- SQLITE_DBCONFIG_ENABLE_QPSG
- SQLITE_DBCONFIG_ENABLE_TRIGGER
- SQLITE_DBCONFIG_ENABLE_VIEW
- SQLITE_DBCONFIG_LEGACY_ALTER_TABLE
- SQLITE_DBCONFIG_LEGACY_FILE_FORMAT
- SQLITE_DBCONFIG_LOOKASIDE
- SQLITE_DBCONFIG_MAINDBNAME
- SQLITE_DBCONFIG_MAX
- SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE
- SQLITE_DBCONFIG_RESET_DATABASE
- SQLITE_DBCONFIG_TRIGGER_EQP
- SQLITE_DBCONFIG_TRUSTED_SCHEMA
- SQLITE_DBCONFIG_WRITABLE_SCHEMA
- SQLITE_DBSTATUS_CACHE_HIT
- SQLITE_DBSTATUS_CACHE_MISS
- SQLITE_DBSTATUS_CACHE_SPILL
- SQLITE_DBSTATUS_CACHE_USED
- SQLITE_FCNTL_TEMPFILENAME
- SQLITE_FCNTL_TRACE
- SQLITE_FCNTL_VFSNAME
- SQLITE_FCNTL_VFS_POINTER
- SQLITE_FCNTL_WAL_BLOCK
- SQLITE_FCNTL_WIN32_AV_RETRY
- SQLITE_FCNTL_WIN32_GET_HANDLE
- SQLITE_FCNTL_WIN32_SET_HANDLE
- SQLITE_FCNTL_ZIPVFS
- SQLITE_FLOAT
- SQLITE_FORMAT
- SQLITE_FULL
- SQLITE_FUNCTION
- SQLITE_IGNORE
- SQLITE_INDEX_CONSTRAINT_EQ
- SQLITE_INDEX_CONSTRAINT_FUNCTION
- SQLITE_INDEX_CONSTRAINT_GE
- SQLITE_INDEX_CONSTRAINT_GLOB
- SQLITE_INDEX_CONSTRAINT_GT
- SQLITE_INDEX_CONSTRAINT_IS
- SQLITE_INDEX_CONSTRAINT_ISNOT
- SQLITE_INDEX_CONSTRAINT_ISNOTNULL
- SQLITE_INDEX_CONSTRAINT_ISNULL
- SQLITE_INDEX_CONSTRAINT_LE
- SQLITE_INDEX_CONSTRAINT_LIKE
- SQLITE_INDEX_CONSTRAINT_LT
- SQLITE_INDEX_CONSTRAINT_MATCH
- SQLITE_INDEX_CONSTRAINT_NE
- SQLITE_INDEX_CONSTRAINT_REGEXP
- SQLITE_INDEX_SCAN_UNIQUE
- SQLITE_INNOCUOUS
- SQLITE_INSERT
- SQLITE_INTEGER
- SQLITE_INTERNAL
- SQLITE_INTERRUPT
- SQLITE_IOCAP_ATOMIC
- SQLITE_IOCAP_ATOMIC16K
- SQLITE_IOCAP_ATOMIC1K
- SQLITE_IOCAP_ATOMIC2K
- SQLITE_IOCAP_ATOMIC32K
- SQLITE_IOCAP_ATOMIC4K
- SQLITE_IOCAP_ATOMIC512
- SQLITE_IOCAP_ATOMIC64K
- SQLITE_IOCAP_ATOMIC8K
- SQLITE_IOCAP_BATCH_ATOMIC
- SQLITE_IOCAP_IMMUTABLE
- SQLITE_IOCAP_POWERSAFE_OVERWRITE
- SQLITE_IOCAP_SAFE_APPEND
- SQLITE_IOCAP_SEQUENTIAL
- SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN
- SQLITE_IOERR
- SQLITE_IOERR_ACCESS
- SQLITE_IOERR_AUTH
- SQLITE_IOERR_BEGIN_ATOMIC
- SQLITE_IOERR_BLOCKED
- SQLITE_IOERR_CHECKRESERVEDLOCK
- SQLITE_IOERR_CLOSE
- SQLITE_IOERR_COMMIT_ATOMIC
- SQLITE_IOERR_CONVPATH
- SQLITE_IOERR_DELETE
- SQLITE_IOERR_DELETE_NOENT
- SQLITE_IOERR_DIR_CLOSE
- SQLITE_IOERR_DIR_FSYNC
- SQLITE_IOERR_FSTAT
- SQLITE_IOERR_FSYNC
- SQLITE_IOERR_GETTEMPPATH
- SQLITE_IOERR_LOCK
- SQLITE_IOERR_MMAP
- SQLITE_IOERR_NOMEM
- SQLITE_IOERR_RDLOCK
- SQLITE_IOERR_READ
- SQLITE_IOERR_ROLLBACK_ATOMIC
- SQLITE_IOERR_SEEK
- SQLITE_IOERR_SHMLOCK
- SQLITE_IOERR_SHMMAP
- SQLITE_IOERR_SHMOPEN
- SQLITE_IOERR_SHMSIZE
- SQLITE_IOERR_SHORT_READ
- SQLITE_IOERR_TRUNCATE
- SQLITE_IOERR_UNLOCK
- SQLITE_IOERR_VNODE
- SQLITE_IOERR_WRITE
- SQLITE_OPEN_READONLY
- SQLITE_OPEN_READWRITE
- SQLITE_OPEN_SHAREDCACHE
- SQLITE_OPEN_SUBJOURNAL
- SQLITE_OPEN_TEMP_DB
- SQLITE_OPEN_TEMP_JOURNAL
- SQLITE_OPEN_TRANSIENT_DB
- SQLITE_OPEN_URI
- SQLITE_OPEN_WAL
- SQLITE_PERM
- SQLITE_PRAGMA
- SQLITE_PREPARE_NORMALIZE
- SQLITE_PREPARE_NO_VTAB
- SQLITE_PREPARE_PERSISTENT
- SQLITE_PROTOCOL
- SQLITE_RANGE
- SQLITE_READ
- SQLITE_READONLY
- SQLITE_READONLY_CANTINIT
- SQLITE_READONLY_CANTLOCK
- SQLITE_READONLY_DBMOVED
- SQLITE_READONLY_DIRECTORY
- SQLITE_READONLY_RECOVERY
- SQLITE_READONLY_ROLLBACK
- SQLITE_RECURSIVE
- SQLITE_REINDEX
- SQLITE_REPLACE
- SQLITE_ROLLBACK
- SQLITE_ROW
- SQLITE_SAVEPOINT
- SQLITE_SCANSTAT_EST
- SQLITE_SCANSTAT_EXPLAIN
- SQLITE_SCANSTAT_NAME
- SQLITE_SCANSTAT_NLOOP
- SQLITE_SCANSTAT_NVISIT
- SQLITE_SCANSTAT_SELECTID
- SQLITE_SCHEMA
- SQLITE_SELECT
- SQLITE_SERIALIZE_NOCOPY
- SQLITE_SHM_EXCLUSIVE
- SQLITE_SHM_LOCK
- SQLITE_SHM_NLOCK
- SQLITE_SHM_SHARED
- SQLITE_SHM_UNLOCK
- SQLITE_SOURCE_ID
- SQLITE_STATIC
- SQLITE_STATUS_MALLOC_COUN
- SQLITE_STATUS_MALLOC_SIZE
- SQLITE_STATUS_MEMORY_USEI
- SQLITE_STATUS_PAGECACHE_O
- SQLITE_STATUS_PAGECACHE_S
- SQLITE_STATUS_PAGECACHE_U
- SQLITE_STATUS_PARSER_STAC
- SQLITE_STATUS_SCRATCH_OVE
- SQLITE_STATUS_SCRATCH_SIZ
- SQLITE_STATUS_SCRATCH_USE
- SQLITE_STMTSTATUS_AUTOIND
- SQLITE_STMTSTATUS_FULLSCA
- SQLITE_STMTSTATUS_MEMUSEI
- SQLITE_STMTSTATUS_REPREPA
- SQLITE_STMTSTATUS_RUN
- SQLITE_STMTSTATUS_SORT
- SQLITE_STMTSTATUS_VM_STEP
- SQLITE_SUBTYPE
- SQLITE_SYNC_DATAONLY
- SQLITE_SYNC_FULL
- SQLITE_SYNC_NORMAL
- SQLITE_TESTCTRL_ALWAYS
- SQLITE_TESTCTRL_ASSERT
- SQLITE_TESTCTRL_BENIGN_MA
- SQLITE_TESTCTRL_BITVEC_TES
- SQLITE_TESTCTRL_BYTEORDER
- SQLITE_TESTCTRL_EXPLAIN_ST
- SQLITE_TESTCTRL_EXTRA_SCH
- SQLITE_TESTCTRL_FAULT_INST
- SQLITE_TESTCTRL_FIRST
- SQLITE_TESTCTRL_IMPOSTER
- SQLITE_TESTCTRL_INTERNAL_F
- SQLITE_TESTCTRL_ISINIT
- SQLITE_TESTCTRL_ISKEYWORD
- SQLITE_TESTCTRL_LAST
- SQLITE_TESTCTRL_LOCALTIME_

- SQLITE_DBSTATUS_CACHE_USED_SHARED
- SQLITE_DBSTATUS_CACHE_WRITE
- SQLITE_DBSTATUS_DEFERRED_FKS
- SQLITE_DBSTATUS_LOOKASIDE_HIT
- SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL
- SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE
- SQLITE_DBSTATUS_LOOKASIDE_USED
- SQLITE_DBSTATUS_MAX
- SQLITE_DBSTATUS_SCHEMA_USED
- SQLITE_DBSTATUS_STMT_USED
- SQLITE_DELETE
- SQLITE_DENY
- SQLITE_DESERIALIZE_FREEONCLOSE
- SQLITE_DESERIALIZE_READONLY
- SQLITE_DESERIALIZE_RESIZEABLE
- SQLITE_DETACH
- SQLITE_DETERMINISTIC
- SQLITE_DIRECTONLY
- SQLITE_DONE
- SQLITE_DROP_INDEX
- SQLITE_DROP_TABLE
- SQLITE_DROP_TEMP_INDEX
- SQLITE_DROP_TEMP_TABLE
- SQLITE_DROP_TEMP_TRIGGER
- SQLITE_DROP_TEMP_VIEW
- SQLITE_DROP_TRIGGER
- SQLITE_DROP_VIEW
- SQLITE_DROP_VTABLE
- SQLITE_EMPTY
- SQLITE_ERROR
- SQLITE_ERROR_MISSING_COLLSEQ
- SQLITE_ERROR_RETRY
- SQLITE_ERROR_SNAPSHOT
- SQLITE_FAIL
- SQLITE_FCNTL_BEGIN_ATOMIC_WRITE
- SQLITE_FCNTL_BUSYHANDLER
- SQLITE_FCNTL_CHUNK_SIZE
- SQLITE_FCNTL_CKPT_DONE
- SQLITE_FCNTL_COMMIT_ATOMIC_WRITE
- SQLITE_FCNTL_COMMIT_PHASETWO
- SQLITE_FCNTL_DATA_VERSION
- SQLITE_LIMIT_ATTACHED
- SQLITE_LIMIT_COLUMN
- SQLITE_LIMIT_COMPOUND_SELECT
- SQLITE_LIMIT_EXPR_DEPTH
- SQLITE_LIMIT_FUNCTION_ARG
- SQLITE_LIMIT_LENGTH
- SQLITE_LIMIT_LIKE_PATTERN_LENGTH
- SQLITE_LIMIT_SQL_LENGTH
- SQLITE_LIMIT_TRIGGER_DEPTH
- SQLITE_LIMIT_VARIABLE_NUMBER
- SQLITE_LIMIT_VDBE_OP
- SQLITE_LIMIT_WORKER_THREADS
- SQLITE_LOCKED
- SQLITE_LOCKED_SHAREDCACHE
- SQLITE_LOCKED_VTAB
- SQLITE_LOCK_EXCLUSIVE
- SQLITE_LOCK_NONE
- SQLITE_LOCK_PENDING
- SQLITE_LOCK_RESERVED
- SQLITE_LOCK_SHARED
- SQLITE_MISMATCH
- SQLITE_MISUSE
- SQLITE_MUTEX_FAST
- SQLITE_MUTEX_RECURSIVE
- SQLITE_MUTEX_STATIC_APP1
- SQLITE_MUTEX_STATIC_APP2
- SQLITE_MUTEX_STATIC_APP3
- SQLITE_MUTEX_STATIC_LRU
- SQLITE_MUTEX_STATIC_LRU2
- SQLITE_MUTEX_STATIC_MASTER
- SQLITE_MUTEX_STATIC_MEM
- SQLITE_MUTEX_STATIC_MEM2
- SQLITE_MUTEX_STATIC_OPEN
- SQLITE_MUTEX_STATIC_PMEM
- SQLITE_MUTEX_STATIC_PRNG
- SQLITE_MUTEX_STATIC_VFS1
- SQLITE_MUTEX_STATIC_VFS2
- SQLITE_MUTEX_STATIC_VFS3
- SQLITE_NOLFS
- SQLITE_NOMEM
- SQLITE_NOTADB
- SQLITE_TESTCTRL_NEVER_COR
- SQLITE_TESTCTRL_ONCE_RESE
- SQLITE_TESTCTRL_OPTIMIZATI
- SQLITE_TESTCTRL_PARSER_CO
- SQLITE_TESTCTRL_PENDING_B
- SQLITE_TESTCTRL_PRNG_RESE
- SQLITE_TESTCTRL_PRNG_REST
- SQLITE_TESTCTRL_PRNG_SAVE
- SQLITE_TESTCTRL_PRNG_SEED
- SQLITE_TESTCTRL_RESERVE
- SQLITE_TESTCTRL_RESULT_INT
- SQLITE_TESTCTRL_SCRATCHMA
- SQLITE_TESTCTRL_SORTER_MM
- SQLITE_TESTCTRL_VDBE_COVE
- SQLITE_TEXT
- SQLITE_TOOBIG
- SQLITE_TRACE
- SQLITE_TRACE_CLOSE
- SQLITE_TRACE_PROFILE
- SQLITE_TRACE_ROW
- SQLITE_TRACE_STMT
- SQLITE_TRANSACTION
- SQLITE_TRANSIENT
- SQLITE_UPDATE
- SQLITE_UTF16
- SQLITE_UTF16BE
- SQLITE_UTF16LE
- SQLITE_UTF16_ALIGNED
- SQLITE_UTF8
- SQLITE_VERSION
- SQLITE_VERSION_NUMBER
- SQLITE_VTAB_CONSTRAINT_SU
- SQLITE_VTAB_DIRECTONLY
- SQLITE_VTAB_INNOCUOUS
- SQLITE_WARNING
- SQLITE_WARNING_AUTOINDEX
- SQLITE_WIN32_DATA_DIRECTO
- SQLITE_WIN32_TEMP_DIRECTO

List Of Functions:

Note: Functions marked with "*(exp)*" are experimental and functions whose names are ~~struck through~~ are deprecated.

- sqlite3_aggregate_context
- ~~sqlite3_aggregate_count~~
- sqlite3_auto_extension
- sqlite3_backup_finish
- sqlite3_backup_init
- sqlite3_backup_pagecount
- sqlite3_backup_remaining
- sqlite3_backup_step
- sqlite3_bind_blob
- sqlite3_bind_blob64
- sqlite3_bind_double
- sqlite3_bind_int
- sqlite3_bind_int64
- sqlite3_bind_null
- sqlite3_bind_parameter_count
- sqlite3_bind_parameter_index
- sqlite3_bind_parameter_name
- sqlite3_bind_pointer
- sqlite3_bind_text
- sqlite3_bind_text16
- sqlite3_bind_text64
- sqlite3_bind_value
- sqlite3_bind_zeroblob
- sqlite3_bind_zeroblob64
- sqlite3_blob_bytes
- sqlite3_blob_close
- sqlite3_blob_open
- sqlite3_blob_read
- sqlite3_blob_reopen
- sqlite3_blob_write
- sqlite3_busy_handler
- sqlite3_busy_timeout
- sqlite3_cancel_auto_extension
- sqlite3_create_function
- sqlite3_create_function16
- sqlite3_create_function_v2
- sqlite3_create_module
- sqlite3_create_module_v2
- sqlite3_create_window_function
- sqlite3_data_count
- sqlite3_db_cacheflush
- sqlite3_db_config
- sqlite3_db_filename
- sqlite3_db_handle
- sqlite3_db_mutex
- sqlite3_db_readonly
- sqlite3_db_release_memory
- sqlite3_db_status
- sqlite3_declare_vtab
- sqlite3_deserialize
- sqlite3_drop_modules
- sqlite3_enable_load_extension
- sqlite3_enable_shared_cache
- sqlite3_errcode
- sqlite3_errmsg
- sqlite3_errmsg16
- sqlite3_errstr
- sqlite3_exec
- sqlite3_expanded_sql
- ~~sqlite3_expired~~
- sqlite3_extended_errcode
- sqlite3_extended_result_codes
- sqlite3_file_control
- sqlite3_filename_database
- sqlite3_filename_journal
- sqlite3_filename_wal
- sqlite3_open_v2
- sqlite3_os_end
- sqlite3_os_init
- sqlite3_overload_function
- sqlite3_prepare
- sqlite3_prepare16
- sqlite3_prepare16_v2
- sqlite3_prepare16_v3
- sqlite3_prepare_v2
- sqlite3_prepare_v3
- sqlite3_preupdate_count
- sqlite3_preupdate_depth
- sqlite3_preupdate_hook
- sqlite3_preupdate_new
- sqlite3_preupdate_old
- sqlite3_profile
- sqlite3_progress_handler
- sqlite3_randomness
- sqlite3_realloc
- sqlite3_realloc64
- sqlite3_release_memory
- sqlite3_reset
- sqlite3_reset_auto_extension
- sqlite3_result_blob
- sqlite3_result_blob64
- sqlite3_result_double
- sqlite3_result_error
- sqlite3_result_error16
- sqlite3_result_error_code
- sqlite3_result_error_nomem
- sqlite3_result_error_toobig
- sqlite3_result_int
- sqlite3_result_int64
- sqlite3_str_append
- sqlite3_str_appendall
- sqlite3_str_appendchar
- sqlite3_str_appendf
- sqlite3_str_errcode
- sqlite3_str_finish
- sqlite3_str_length
- sqlite3_str_new
- sqlite3_str_reset
- sqlite3_str_value
- sqlite3_str_vappendf
- sqlite3_strglob
- sqlite3_stricmp
- sqlite3_strlike
- sqlite3_strnicmp
- sqlite3_system_errno
- sqlite3_table_column_me
- sqlite3_test_control
- ~~sqlite3_thread_cleanup~~
- sqlite3_threadsafe
- sqlite3_total_changes
- sqlite3_trace
- sqlite3_trace_v2
- ~~sqlite3_transfer_bindings~~
- sqlite3_unlock_notify
- sqlite3_update_hook
- sqlite3_uri_boolean
- sqlite3_uri_int64
- sqlite3_uri_key
- sqlite3_uri_parameter
- sqlite3_user_data
- sqlite3_value_blob
- sqlite3_value_bytes

- sqlite3_changes
- sqlite3_clear_bindings
- sqlite3_close
- sqlite3_close_v2
- sqlite3_collation_needed
- sqlite3_collation_needed16
- sqlite3_column_blob
- sqlite3_column_bytes
- sqlite3_column_bytes16
- sqlite3_column_count
- sqlite3_column_database_name
- sqlite3_column_database_name16
- sqlite3_column_decltype
- sqlite3_column_decltype16
- sqlite3_column_double
- sqlite3_column_int
- sqlite3_column_int64
- sqlite3_column_name
- sqlite3_column_name16
- sqlite3_column_origin_name
- sqlite3_column_origin_name16
- sqlite3_column_table_name
- sqlite3_column_table_name16
- sqlite3_column_text
- sqlite3_column_text16
- sqlite3_column_type
- sqlite3_column_value
- sqlite3_commit_hook
- sqlite3_compileoption_get
- sqlite3_compileoption_used
- sqlite3_complete
- sqlite3_complete16
- sqlite3_config
- sqlite3_context_db_handle
- sqlite3_create_collation
- sqlite3_create_collation16
- sqlite3_create_collation_v2
- sqlite3_finalize
- sqlite3_free
- sqlite3_free_table
- sqlite3_get_autocommit
- sqlite3_get_auxdata
- sqlite3_get_table
- ~~sqlite3_global_recover~~
- sqlite3_hard_heap_limit64
- sqlite3_initialize
- sqlite3_interrupt
- sqlite3_keyword_check
- sqlite3_keyword_count
- sqlite3_keyword_name
- sqlite3_last_insert_rowid
- sqlite3_libversion
- sqlite3_libversion_number
- sqlite3_limit
- sqlite3_load_extension
- sqlite3_log
- sqlite3_malloc
- sqlite3_malloc64
- ~~sqlite3_memory_alarm~~
- sqlite3_memory_highwater
- sqlite3_memory_used
- sqlite3_mprintf
- sqlite3_msize
- sqlite3_mutex_alloc
- sqlite3_mutex_enter
- sqlite3_mutex_free
- sqlite3_mutex_held
- sqlite3_mutex_leave
- sqlite3_mutex_notheld
- sqlite3_mutex_try
- sqlite3_next_stmt
- sqlite3_normalized_sql
- sqlite3_open
- sqlite3_open16
- sqlite3_result_null
- sqlite3_result_pointer
- sqlite3_result_subtype
- sqlite3_result_text
- sqlite3_result_text16
- sqlite3_result_text16be
- sqlite3_result_text16le
- sqlite3_result_text64
- sqlite3_result_value
- sqlite3_result_zeroblob
- sqlite3_result_zeroblob64
- sqlite3_rollback_hook
- sqlite3_serialize
- sqlite3_set_authorizer
- sqlite3_set_auxdata
- sqlite3_set_last_insert_rowid
- sqlite3_shutdown
- sqlite3_sleep
- sqlite3_snapshot_cmp
- sqlite3_snapshot_free
- sqlite3_snapshot_get
- sqlite3_snapshot_open
- sqlite3_snapshot_recover
- sqlite3_snprintf
- ~~sqlite3_soft_heap_limit~~
- sqlite3_soft_heap_limit64
- sqlite3_sourceid
- sqlite3_sql
- sqlite3_status
- sqlite3_status64
- sqlite3_step
- sqlite3_stmt_busy
- sqlite3_stmt_isexplain
- sqlite3_stmt_readonly
- sqlite3_stmt_scanstatus
- sqlite3_stmt_scanstatus_reset
- sqlite3_stmt_status
- sqlite3_value_bytes16
- sqlite3_value_double
- sqlite3_value_dup
- sqlite3_value_free
- sqlite3_value_frombind
- sqlite3_value_int
- sqlite3_value_int64
- sqlite3_value_nochange
- sqlite3_value_numeric_ty
- sqlite3_value_pointer
- sqlite3_value_subtype
- sqlite3_value_text
- sqlite3_value_text16
- sqlite3_value_text16be
- sqlite3_value_text16le
- sqlite3_value_type
- sqlite3_version
- sqlite3_vfs_find
- sqlite3_vfs_register
- sqlite3_vfs_unregister
- sqlite3_vmprintf
- sqlite3_vsnprintf
- sqlite3_vtab_collation
- sqlite3_vtab_config
- sqlite3_vtab_nochange
- sqlite3_vtab_on_conflict
- sqlite3_wal_autocheckpoi
- sqlite3_wal_checkpoint
- sqlite3_wal_checkpoint_v
- sqlite3_wal_hook
- sqlite3_win32_set_directo
- sqlite3_win32_set_directo
- sqlite3_win32_set_directo

Virtual Table Scan Flags

```
#define SQLITE_INDEX_SCAN_UNIQUE      1     /* Scan visits at most 1 row */
```

Virtual table implementations are allowed to set the sqlite3_index_info.idxFlags field to some combination of these bits.

Flags for sqlite3_serialize

```
#define SQLITE_SERIALIZE_NOCOPY 0x001   /* Do no memory allocations */
```

Zero or more of the following constants can be OR-ed together for the F argument to sqlite3_serialize(D,S,P,F).

SQLITE_SERIALIZE_NOCOPY means that sqlite3_serialize() will return a pointer to contiguous in-memory database that it is currently using, without making a copy of the database. If SQLite is not currently using a contiguous in-memory database, then this option causes sqlite3_serialize() to return a NULL pointer. SQLite will only be using a contiguous in-memory database if it has been initialized by a prior call to sqlite3_deserialize().

Maximum xShmLock index

```
#define SQLITE_SHM_NLOCK        8
```

The xShmLock method on sqlite3_io_methods may use values between 0 and this upper bound as its "offset" argument. The SQLite core will never attempt to acquire or release a lock outside of this range

---

Loadable Extension Thunk

```
typedef struct sqlite3_api_routines sqlite3_api_routines;
```

A pointer to the opaque sqlite3_api_routines structure is passed as the third parameter to entry points of loadable extensions. This structure must be typedefed in order to work around compiler warnings on some platforms.

Online Backup Object

```
typedef struct sqlite3_backup sqlite3_backup;
```

The sqlite3_backup object records state information about an ongoing online backup operation. The sqlite3_backup object is created by a call to sqlite3_backup_init() and is destroyed by a call to sqlite3_backup_finish().

See Also: Using the SQLite Online Backup API

SQL Function Context Object

```
typedef struct sqlite3_context sqlite3_context;
```

The context in which an SQL function executes is stored in an sqlite3_context object. A pointer to an sqlite3_context object is always first parameter to application-defined SQL functions. The application-defined SQL function implementation will pass this pointer through into calls to sqlite3_result(), sqlite3_aggregate_context(), sqlite3_user_data(), sqlite3_context_db_handle(), sqlite3_get_auxdata(), and/or sqlite3_set_auxdata().

Methods:

- sqlite3_aggregate_context
- sqlite3_context_db_handle
- sqlite3_get_auxdata
- sqlite3_result_blob
- sqlite3_result_blob64
- sqlite3_result_double
- sqlite3_result_error
- sqlite3_result_error16
- sqlite3_result_error_code
- sqlite3_result_error_nomem
- sqlite3_result_error_toobig
- sqlite3_result_int
- sqlite3_result_int64
- sqlite3_result_null
- sqlite3_result_pointer
- sqlite3_result_subtype
- sqlite3_result_text
- sqlite3_result_text16
- sqlite3_result_text16be
- sqlite3_result_text16le
- sqlite3_result_text64
- sqlite3_result_value
- sqlite3_result_zeroblob
- sqlite3_result_zeroblob64
- sqlite3_set_auxdata
- sqlite3_user_data

Name Of The Folder Holding Database Files

```
SQLITE_EXTERN char *sqlite3_data_directory;
```

If this global variable is made to point to a string which is the name of a folder (a.k.a. directory), then all database files specified with a relative pathname and created or accessed by SQLite when using a built-in windows VFS will be assumed to be relative to that directory. If this variable is a NULL pointer, then SQLite assumes that all database files specified with a relative pathname are relative to the current directory for the process. Only the windows VFS makes use of this global variable; it is ignored by the unix VFS.

Changing the value of this variable while a database connection is open can result in a corrupt database.

It is not safe to read or modify this variable in more than one thread at a time. It is not safe to read or modify this variable if a database connection is being used at the same time in a separate thread. It is intended that this variable be set once as part of process initialization and before any SQLite interface routines have been called and that this variable remain unchanged thereafter.

The data_store_directory pragma may modify this variable and cause it to point to memory obtained from sqlite3_malloc. Furthermore, the data_store_directory pragma always assumes that any string that this variable points to is held in memory obtained from sqlite3_malloc and the pragma may attempt to free that memory using sqlite3_free. Hence, if this variable is modified directly, either it should be made NULL or made to point to memory obtained from sqlite3_malloc or else the use of the data_store_directory pragma should be avoided.

OS Interface Open File Handle

```
typedef struct sqlite3_file sqlite3_file;
struct sqlite3_file {
  const struct sqlite3_io_methods *pMethods;  /* Methods for an open file */
};
```

An sqlite3_file object represents an open file in the OS interface layer. Individual OS interface implementations will want to subclass this object by appending additional fields for their own use. The pMethods entry is a pointer to an sqlite3_io_methods object that defines methods for performing I/O operations on the open file.

Virtual Table Indexing Information

```
struct sqlite3_index_info {
  /* Inputs */
  int nConstraint;           /* Number of entries in aConstraint */
  struct sqlite3_index_constraint {
     int iColumn;              /* Column constrained.  -1 for ROWID */
     unsigned char op;         /* Constraint operator */
     unsigned char usable;     /* True if this constraint is usable */
     int iTermOffset;          /* Used internally - xBestIndex should ignore */
  } *aConstraint;            /* Table of WHERE clause constraints */
  int nOrderBy;              /* Number of terms in the ORDER BY clause */
  struct sqlite3_index_orderby {
     int iColumn;              /* Column number */
     unsigned char desc;       /* True for DESC.  False for ASC. */
  } *aOrderBy;               /* The ORDER BY clause */
  /* Outputs */
  struct sqlite3_index_constraint_usage {
    int argvIndex;           /* if >0, constraint is part of argv to xFilter */
    unsigned char omit;      /* Do not code a test for this constraint */
  } *aConstraintUsage;
  int idxNum;                /* Number used to identify the index */
  char *idxStr;              /* String, possibly obtained from sqlite3_malloc */
  int needToFreeIdxStr;      /* Free idxStr using sqlite3_free() if true */
  int orderByConsumed;       /* True if output is already ordered */
  double estimatedCost;           /* Estimated cost of using this index */
  /* Fields below are only available in SQLite 3.8.2 and later */
  sqlite3_int64 estimatedRows;    /* Estimated number of rows returned */
  /* Fields below are only available in SQLite 3.9.0 and later */
```

```
  int idxFlags;              /* Mask of SQLITE_INDEX_SCAN_* flags */
  /* Fields below are only available in SQLite 3.10.0 and later */
  sqlite3_uint64 colUsed;    /* Input: Mask of columns used by statement */
};
```

The sqlite3_index_info structure and its substructures is used as part of the virtual table interface to pass information into and receive the reply from the xBestIndex method of a virtual table module. The fields under **Inputs** are the inputs to xBestIndex and are read-only. xBestIndex inserts its results into the **Outputs** fields.

The aConstraint[] array records WHERE clause constraints of the form:

column OP expr

where OP is =, <, <=, >, or >=. The particular operator is stored in aConstraint[].op using one of the SQLITE_INDEX_CONSTRAINT_ values. The index of the column is stored in aConstraint[].iColumn. aConstraint[].usable is TRUE if the expr on the right-hand side can be evaluated (and thus the constraint is usable) and false if it cannot.

The optimizer automatically inverts terms of the form "expr OP column" and makes other simplifications to the WHERE clause in an attempt to get as many WHERE clause terms into the form shown above as possible. The aConstraint[] array only reports WHERE clause terms that are relevant to the particular virtual table being queried.

Information about the ORDER BY clause is stored in aOrderBy[]. Each term of aOrderBy records a column of the ORDER BY clause.

The colUsed field indicates which columns of the virtual table may be required by the current scan. Virtual table columns are numbered from zero in the order in which they appear within the CREATE TABLE statement passed to sqlite3_declare_vtab(). For the first 63 columns (columns 0-62), the corresponding bit is set within the colUsed mask if the column may be required by SQLite. If the table has at least 64 columns and any column to the right of the first 63 is required, then bit 63 of colUsed is also set. In other words, column iCol may be required if the expression (colUsed & ((sqlite3_uint64)1 << (iCol>=63 ? 63 : iCol))) evaluates to non-zero.

The xBestIndex method must fill aConstraintUsage[] with information about what parameters to pass to xFilter. If argvIndex>0 then the right-hand side of the corresponding aConstraint[] is evaluated and becomes the argvIndex-th entry in argv. If aConstraintUsage[].omit is true, then the constraint is assumed to be fully handled by the virtual table and might not be checked again by the byte code. The aConstraintUsage[].omit flag is an optimization hint. When the omit flag is left in its default setting of false, the constraint will always be checked separately in byte code. If the omit flag is change to true, then the constraint may or may not be checked in byte code. In other words, when the omit flag is true there is no guarantee that the constraint will not be checked again using byte code.

The idxNum and idxPtr values are recorded and passed into the xFilter method. sqlite3_free() is used to free idxPtr if and only if needToFreeIdxPtr is true.

The orderByConsumed means that output from xFilter/xNext will occur in the correct order to satisfy the ORDER BY clause so that no separate sorting step is required.

The estimatedCost value is an estimate of the cost of a particular strategy. A cost of N indicates that the cost of the strategy is similar to a linear scan of an SQLite table with N rows. A cost of log(N) indicates that the expense of the operation is similar to that of a binary search on a unique indexed field of an SQLite table with N rows.

The estimatedRows value is an estimate of the number of rows that will be returned by the strategy.

The xBestIndex method may optionally populate the idxFlags field with a mask of SQLITE_INDEX_SCAN_* flags. Currently there is only one such flag - SQLITE_INDEX_SCAN_UNIQUE. If the xBestIndex method sets this flag, SQLite assumes that the strategy may visit at most one row.

Additionally, if xBestIndex sets the SQLITE_INDEX_SCAN_UNIQUE flag, then SQLite also assumes that if a call to the xUpdate() method is made as part of the same statement to delete or update a virtual table row and the implementation returns SQLITE_CONSTRAINT, then there is no need to rollback any database changes. In other words, if the xUpdate() returns SQLITE_CONSTRAINT, the database contents must be exactly as they were before xUpdate was called. By contrast, if SQLITE_INDEX_SCAN_UNIQUE is not set and xUpdate returns SQLITE_CONSTRAINT, any database changes made by the xUpdate method are automatically rolled back by SQLite.

IMPORTANT: The estimatedRows field was added to the sqlite3_index_info structure for SQLite version 3.8.2 (2013-12-06). If a virtual table extension is used with an SQLite version earlier than 3.8.2, the results of attempting to read or write the estimatedRows field are undefined (but are likely to include crashing the application). The estimatedRows field should therefore only be used if sqlite3_libversion_number() returns a value greater than or equal to 3008002. Similarly, the idxFlags field was added for version 3.9.0 (2015-10-14). It may therefore only be used if sqlite3_libversion_number() returns a value greater than or equal to 3009000.

OS Interface File Virtual Methods Object

```
typedef struct sqlite3_io_methods sqlite3_io_methods;
struct sqlite3_io_methods {
  int iVersion;
  int (*xClose)(sqlite3_file*);
  int (*xRead)(sqlite3_file*, void*, int iAmt, sqlite3_int64 iOfst);
  int (*xWrite)(sqlite3_file*, const void*, int iAmt, sqlite3_int64 iOfst);
  int (*xTruncate)(sqlite3_file*, sqlite3_int64 size);
  int (*xSync)(sqlite3_file*, int flags);
  int (*xFileSize)(sqlite3_file*, sqlite3_int64 *pSize);
  int (*xLock)(sqlite3_file*, int);
  int (*xUnlock)(sqlite3_file*, int);
  int (*xCheckReservedLock)(sqlite3_file*, int *pResOut);
  int (*xFileControl)(sqlite3_file*, int op, void *pArg);
  int (*xSectorSize)(sqlite3_file*);
  int (*xDeviceCharacteristics)(sqlite3_file*);
  /* Methods above are valid for version 1 */
  int (*xShmMap)(sqlite3_file*, int iPg, int pgsz, int, void volatile**);
  int (*xShmLock)(sqlite3_file*, int offset, int n, int flags);
```

```
  void (*xShmBarrier)(sqlite3_file*);
  int (*xShmUnmap)(sqlite3_file*, int deleteFlag);
  /* Methods above are valid for version 2 */
  int (*xFetch)(sqlite3_file*, sqlite3_int64 iOfst, int iAmt, void **pp);
  int (*xUnfetch)(sqlite3_file*, sqlite3_int64 iOfst, void *p);
  /* Methods above are valid for version 3 */
  /* Additional methods may be added in future releases */
};
```

Every file opened by the sqlite3_vfs.xOpen method populates an sqlite3_file object (or, more commonly, a subclass of the sqlite3_file object) with a pointer to an instance of this object. This object defines the methods used to perform various operations against the open file represented by the sqlite3_file object.

If the sqlite3_vfs.xOpen method sets the sqlite3_file.pMethods element to a non-NULL pointer, then the sqlite3_io_methods.xClose method may be invoked even if the sqlite3_vfs.xOpen reported that it failed. The only way to prevent a call to xClose following a failed sqlite3_vfs.xOpen is for the sqlite3_vfs.xOpen to set the sqlite3_file.pMethods element to NULL.

The flags argument to xSync may be one of SQLITE_SYNC_NORMAL or SQLITE_SYNC_FULL. The first choice is the normal fsync(). The second choice is a Mac OS X style fullsync. The SQLITE_SYNC_DATAONLY flag may be ORed in to indicate that only the data of the file and not its inode needs to be synced.

The integer values to xLock() and xUnlock() are one of

- SQLITE_LOCK_NONE,
- SQLITE_LOCK_SHARED,
- SQLITE_LOCK_RESERVED,
- SQLITE_LOCK_PENDING, or
- SQLITE_LOCK_EXCLUSIVE.

xLock() increases the lock. xUnlock() decreases the lock. The xCheckReservedLock() method checks whether any database connection, either in this process or in some other process, is holding a RESERVED, PENDING, or EXCLUSIVE lock on the file. It returns true if such a lock exists and false otherwise.

The xFileControl() method is a generic interface that allows custom VFS implementations to directly control an open file using the sqlite3_file_control() interface. The second "op" argument is an integer opcode. The third argument is a generic pointer intended to point to a structure that may contain arguments or space in which to write return values. Potential uses for xFileControl() might be functions to enable blocking locks with timeouts, to change the locking strategy (for example to use dot-file locks), to inquire about the status of a lock, or to break stale locks. The SQLite core reserves all opcodes less than 100 for its own use. A list of opcodes less than 100 is available. Applications that define a custom xFileControl method should use opcodes greater than 100 to avoid conflicts. VFS implementations should return SQLITE_NOTFOUND for file control opcodes that they do not recognize.

The xSectorSize() method returns the sector size of the device that underlies the file. The sector size is the minimum write that can be performed without disturbing other bytes in the file. The xDeviceCharacteristics() method returns a bit vector describing behaviors of the underlying device:

- SQLITE_IOCAP_ATOMIC
- SQLITE_IOCAP_ATOMIC512
- SQLITE_IOCAP_ATOMIC1K
- SQLITE_IOCAP_ATOMIC2K
- SQLITE_IOCAP_ATOMIC4K
- SQLITE_IOCAP_ATOMIC8K
- SQLITE_IOCAP_ATOMIC16K
- SQLITE_IOCAP_ATOMIC32K
- SQLITE_IOCAP_ATOMIC64K
- SQLITE_IOCAP_SAFE_APPEND
- SQLITE_IOCAP_SEQUENTIAL
- SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN
- SQLITE_IOCAP_POWERSAFE_OVERWRITE
- SQLITE_IOCAP_IMMUTABLE
- SQLITE_IOCAP_BATCH_ATOMIC

The SQLITE_IOCAP_ATOMIC property means that all writes of any size are atomic. The SQLITE_IOCAP_ATOMICnnn values mean that writes of blocks that are nnn bytes in size and are aligned to an address which is an integer multiple of nnn are atomic. The SQLITE_IOCAP_SAFE_APPEND value means that when data is appended to a file, the data is appended first then the size of the file is extended, never the other way around. The SQLITE_IOCAP_SEQUENTIAL property means that information is written to disk in the same order as calls to xWrite().

If xRead() returns SQLITE_IOERR_SHORT_READ it must also fill in the unread portions of the buffer with zeros. A VFS that fails to zero-fill short reads might seem to work. However, failure to zero-fill short reads will eventually lead to database corruption.

Memory Allocation Routines

```
typedef struct sqlite3_mem_methods sqlite3_mem_methods;
struct sqlite3_mem_methods {
  void *(*xMalloc)(int);         /* Memory allocation function */
  void (*xFree)(void*);          /* Free a prior allocation */
  void *(*xRealloc)(void*,int);  /* Resize an allocation */
  int (*xSize)(void*);           /* Return the size of an allocation */
  int (*xRoundup)(int);          /* Round up request size to allocation size */
  int (*xInit)(void*);           /* Initialize the memory allocator */
  void (*xShutdown)(void*);      /* Deinitialize the memory allocator */
  void *pAppData;                /* Argument to xInit() and xShutdown() */
};
```

An instance of this object defines the interface between SQLite and low-level memory allocation routines.

This object is used in only one place in the SQLite interface. A pointer to an instance of this object is the argument to sqlite3_config() when the configuration option is SQLITE_CONFIG_MALLOC or SQLITE_CONFIG_GETMALLOC. By creating an instance of this object and passing it to sqlite3_config(SQLITE_CONFIG_MALLOC) during configuration, an application can specify an alternative memory allocation subsystem for SQLite to use for all of its dynamic memory needs.

Note that SQLite comes with several built-in memory allocators that are perfectly adequate for the overwhelming majority of applications and that this object is only useful to a tiny minority of applications with specialized memory allocation requirements. This object is also used during testing of SQLite in order to specify an alternative memory allocator that simulates memory out-of-memory conditions in order to verify that SQLite recovers gracefully from such conditions.

The xMalloc, xRealloc, and xFree methods must work like the malloc(), realloc() and free() functions from the standard C library. SQLite guarantees that the second argument to xRealloc is always a value returned by a prior call to xRoundup.

xSize should return the allocated size of a memory allocation previously obtained from xMalloc or xRealloc. The allocated size is always at least as big as the requested size but may be larger.

The xRoundup method returns what would be the allocated size of a memory allocation given a particular requested size. Most memory allocators round up memory allocations at least to the next multiple of 8. Some allocators round up to a larger multiple or to a power of 2. Every memory allocation request coming in through sqlite3_malloc() or sqlite3_realloc() first calls xRoundup. If xRoundup returns 0, that causes the corresponding memory allocation to fail.

The xInit method initializes the memory allocator. For example, it might allocate any required mutexes or initialize internal data structures. The xShutdown method is invoked (indirectly) by sqlite3_shutdown() and should deallocate any resources acquired by xInit. The pAppData pointer is used as the only parameter to xInit and xShutdown.

SQLite holds the SQLITE_MUTEX_STATIC_MASTER mutex when it invokes the xInit method, so the xInit method need not be threadsafe. The xShutdown method is only called from sqlite3_shutdown() so it does not need to be threadsafe either. For all other methods, SQLite holds the SQLITE_MUTEX_STATIC_MEM mutex as long as the SQLITE_CONFIG_MEMSTATUS configuration option is turned on (which it is by default) and so the methods are automatically serialized. However, if SQLITE_CONFIG_MEMSTATUS is disabled, then the other methods must be threadsafe or else make their own arrangements for serialization.

SQLite will never invoke xInit() more than once without an intervening call to xShutdown().

Mutex Handle

```
typedef struct sqlite3_mutex sqlite3_mutex;
```

The mutex module within SQLite defines sqlite3_mutex to be an abstract type for a mutex object. The SQLite core never looks at the internal representation of an sqlite3_mutex. It only deals with pointers to the sqlite3_mutex object.

Mutexes are created using sqlite3_mutex_alloc().

Mutex Methods Object

```
typedef struct sqlite3_mutex_methods sqlite3_mutex_methods;
struct sqlite3_mutex_methods {
  int (*xMutexInit)(void);
  int (*xMutexEnd)(void);
  sqlite3_mutex *(*xMutexAlloc)(int);
  void (*xMutexFree)(sqlite3_mutex *);
  void (*xMutexEnter)(sqlite3_mutex *);
  int (*xMutexTry)(sqlite3_mutex *);
  void (*xMutexLeave)(sqlite3_mutex *);
  int (*xMutexHeld)(sqlite3_mutex *);
  int (*xMutexNotheld)(sqlite3_mutex *);
};
```

An instance of this structure defines the low-level routines used to allocate and use mutexes.

Usually, the default mutex implementations provided by SQLite are sufficient, however the application has the option of substituting a custom implementation for specialized deployments or systems for which SQLite does not provide a suitable implementation. In this case, the application creates and populates an instance of this structure to pass to sqlite3_config() along with the SQLITE_CONFIG_MUTEX option. Additionally, an instance of this structure can be used as an output variable when querying the system for the current mutex implementation, using the SQLITE_CONFIG_GETMUTEX option.

The xMutexInit method defined by this structure is invoked as part of system initialization by the sqlite3_initialize() function. The xMutexInit routine is called by SQLite exactly once for each effective call to sqlite3_initialize().

The xMutexEnd method defined by this structure is invoked as part of system shutdown by the sqlite3_shutdown() function. The implementation of this method is expected to release all outstanding resources obtained by the mutex methods implementation, especially those obtained by the xMutexInit method. The xMutexEnd() interface is invoked exactly once for each call to sqlite3_shutdown().

The remaining seven methods defined by this structure (xMutexAlloc, xMutexFree, xMutexEnter, xMutexTry, xMutexLeave, xMutexHeld and xMutexNotheld) implement the following interfaces (respectively):

- sqlite3_mutex_alloc()
- sqlite3_mutex_free()
- sqlite3_mutex_enter()
- sqlite3_mutex_try()
- sqlite3_mutex_leave()
- sqlite3_mutex_held()
- sqlite3_mutex_notheld()

The only difference is that the public sqlite3_XXX functions enumerated above silently ignore any invocations that pass a NULL pointer instead of a valid mutex handle. The implementations of the methods defined by this structure are not required to handle this case. The results of passing a NULL pointer instead of a valid mutex handle are undefined (i.e. it is acceptable to provide an implementation that segfaults if it is passed a NULL pointer).

The xMutexInit() method must be threadsafe. It must be harmless to invoke xMutexInit() multiple times within the same process and without intervening calls to xMutexEnd(). Second and subsequent calls to xMutexInit() must be no-ops.

xMutexInit() must not use SQLite memory allocation (sqlite3_malloc() and its associates). Similarly, xMutexAlloc() must not use SQLite memory allocation for a static mutex. However xMutexAlloc() may use SQLite memory allocation for a fast or recursive mutex.

SQLite will invoke the xMutexEnd() method when sqlite3_shutdown() is called, but only if the prior call to xMutexInit returned SQLITE_OK. If xMutexInit fails in any way, it is expected to clean up after itself prior to returning.

Custom Page Cache Object

```
typedef struct sqlite3_pcache sqlite3_pcache;
```

The sqlite3_pcache type is opaque. It is implemented by the pluggable module. The SQLite core has no knowledge of its size or internal structure and never deals with the sqlite3_pcache object except by holding and passing pointers to the object.

See sqlite3_pcache_methods2 for additional information.

Custom Page Cache Object

```
typedef struct sqlite3_pcache_page sqlite3_pcache_page;
struct sqlite3_pcache_page {
  void *pBuf;        /* The content of the page */
  void *pExtra;      /* Extra information associated with the page */
};
```

The sqlite3_pcache_page object represents a single page in the page cache. The page cache will allocate instances of this object. Various methods of the page cache use pointers to instances of this object as parameters or as their return value.

See sqlite3_pcache_methods2 for additional information.

Name Of The Folder Holding Temporary Files

```
SQLITE_EXTERN char *sqlite3_temp_directory;
```

If this global variable is made to point to a string which is the name of a folder (a.k.a. directory), then all temporary files created by SQLite when using a built-in VFS will be placed in that directory. If this variable is a NULL pointer, then SQLite performs a search for an appropriate temporary file directory.

Applications are strongly discouraged from using this global variable. It is required to set a temporary folder on Windows Runtime (WinRT). But for all other platforms, it is highly recommended that applications neither read nor write this variable. This global variable is a relic that exists for backwards compatibility of legacy applications and should be avoided in new projects.

It is not safe to read or modify this variable in more than one thread at a time. It is not safe to read or modify this variable if a database connection is being used at the same time in a separate thread. It is intended that this variable be set once as part of process initialization and before any SQLite interface routines have been called and that this variable remain unchanged thereafter.

The temp_store_directory pragma may modify this variable and cause it to point to memory obtained from sqlite3_malloc. Furthermore, the temp_store_directory pragma always assumes that any string that this variable points to is held in memory obtained from sqlite3_malloc and the pragma may attempt to free that memory using sqlite3_free. Hence, if this variable is modified directly, either it should be made NULL or made to point to memory obtained from sqlite3_malloc or else the use of the temp_store_directory pragma should be avoided. Except when requested by the temp_store_directory pragma, SQLite does not free the memory that sqlite3_temp_directory points to. If the application wants that memory to be freed, it must do so itself, taking care to only do so after all database connection objects have been destroyed.

Note to Windows Runtime users: The temporary directory must be set prior to calling sqlite3_open or sqlite3_open_v2. Otherwise, various features that require the use of temporary files may fail. Here is an example of how to do this using C++ with the Windows Runtime:

```
LPCWSTR zPath = Windows::Storage::ApplicationData::Current->
      TemporaryFolder->Path->Data();
char zPathBuf[MAX_PATH + 1];
memset(zPathBuf, 0, sizeof(zPathBuf));
WideCharToMultiByte(CP_UTF8, 0, zPath, -1, zPathBuf, sizeof(zPathBuf),
      NULL, NULL);
sqlite3_temp_directory = sqlite3_mprintf("%s", zPathBuf);
```

OS Interface Object

```
typedef struct sqlite3_vfs sqlite3_vfs;
typedef void (*sqlite3_syscall_ptr)(void);
struct sqlite3_vfs {
  int iVersion;            /* Structure version number (currently 3) */
  int szOsFile;            /* Size of subclassed sqlite3_file */
  int mxPathname;          /* Maximum file pathname length */
  sqlite3_vfs *pNext;      /* Next registered VFS */
  const char *zName;       /* Name of this virtual file system */
```

```
  void *pAppData;          /* Pointer to application-specific data */
  int (*xOpen)(sqlite3_vfs*, const char *zName, sqlite3_file*,
               int flags, int *pOutFlags);
  int (*xDelete)(sqlite3_vfs*, const char *zName, int syncDir);
  int (*xAccess)(sqlite3_vfs*, const char *zName, int flags, int *pResOut);
  int (*xFullPathname)(sqlite3_vfs*, const char *zName, int nOut, char *zOut);
  void *(*xDlOpen)(sqlite3_vfs*, const char *zFilename);
  void (*xDlError)(sqlite3_vfs*, int nByte, char *zErrMsg);
  void (*(*xDlSym)(sqlite3_vfs*,void*, const char *zSymbol))(void);
  void (*xDlClose)(sqlite3_vfs*, void*);
  int (*xRandomness)(sqlite3_vfs*, int nByte, char *zOut);
  int (*xSleep)(sqlite3_vfs*, int microseconds);
  int (*xCurrentTime)(sqlite3_vfs*, double*);
  int (*xGetLastError)(sqlite3_vfs*, int, char *);
  /*
  ** The methods above are in version 1 of the sqlite_vfs object
  ** definition.  Those that follow are added in version 2 or later
  */
  int (*xCurrentTimeInt64)(sqlite3_vfs*, sqlite3_int64*);
  /*
  ** The methods above are in versions 1 and 2 of the sqlite_vfs object.
  ** Those below are for version 3 and greater.
  */
  int (*xSetSystemCall)(sqlite3_vfs*, const char *zName, sqlite3_syscall_ptr);
  sqlite3_syscall_ptr (*xGetSystemCall)(sqlite3_vfs*, const char *zName);
  const char *(*xNextSystemCall)(sqlite3_vfs*, const char *zName);
  /*
  ** The methods above are in versions 1 through 3 of the sqlite_vfs object.
  ** New fields may be appended in future versions.  The iVersion
  ** value will increment whenever this happens.
  */
};
```

An instance of the sqlite3_vfs object defines the interface between the SQLite core and the underlying operating system. The "vfs" in the name of the object stands for "virtual file system". See the VFS documentation for further information.

The VFS interface is sometimes extended by adding new methods onto the end. Each time such an extension occurs, the iVersion field is incremented. The iVersion value started out as 1 in SQLite version 3.5.0 on 2007-09-04, then increased to 2 with SQLite version 3.7.0 on 2010-07-21, and then increased to 3 with SQLite version 3.7.6 on 2011-04-12. Additional fields may be appended to the sqlite3_vfs object and the iVersion value may increase again in future versions of SQLite. Note that due to an oversight, the structure of the sqlite3_vfs object changed in the transition from SQLite version 3.5.9 to version 3.6.0 on 2008-07-16 and yet the iVersion field was not increased.

The szOsFile field is the size of the subclassed sqlite3_file structure used by this VFS. mxPathname is the maximum length of a pathname in this VFS.

Registered sqlite3_vfs objects are kept on a linked list formed by the pNext pointer. The sqlite3_vfs_register() and sqlite3_vfs_unregister() interfaces manage this list in a thread-safe way. The sqlite3_vfs_find() interface searches the list. Neither the application code nor the VFS implementation should use the pNext pointer.

The pNext field is the only field in the sqlite3_vfs structure that SQLite will ever modify. SQLite will only access or modify this field while holding a particular static mutex. The application should never modify anything within the sqlite3_vfs object once the object has been registered.

The zName field holds the name of the VFS module. The name must be unique across all VFS modules.

SQLite guarantees that the zFilename parameter to xOpen is either a NULL pointer or string obtained from xFullPathname() with an optional suffix added. If a suffix is added to the zFilename parameter, it will consist of a single "-" character followed by no more than 11 alphanumeric and/or "-" characters. SQLite further guarantees that the string will be valid and unchanged until xClose() is called. Because of the previous sentence, the sqlite3_file can safely store a pointer to the filename if it needs to remember the filename for some reason. If the zFilename parameter to xOpen is a NULL pointer then xOpen must invent its own temporary name for the file. Whenever the xFilename parameter is NULL it will also be the case that the flags parameter will include SQLITE_OPEN_DELETEONCLOSE.

The flags argument to xOpen() includes all bits set in the flags argument to sqlite3_open_v2(). Or if sqlite3_open() or sqlite3_open16() is used, then flags includes at least SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE. If xOpen() opens a file read-only then it sets *pOutFlags to include SQLITE_OPEN_READONLY. Other bits in *pOutFlags may be set.

SQLite will also add one of the following flags to the xOpen() call, depending on the object being opened:

- SQLITE_OPEN_MAIN_DB
- SQLITE_OPEN_MAIN_JOURNAL
- SQLITE_OPEN_TEMP_DB
- SQLITE_OPEN_TEMP_JOURNAL
- SQLITE_OPEN_TRANSIENT_DB
- SQLITE_OPEN_SUBJOURNAL
- SQLITE_OPEN_MASTER_JOURNAL
- SQLITE_OPEN_WAL

The file I/O implementation can use the object type flags to change the way it deals with files. For example, an application that does not care about crash recovery or rollback might make the open of a journal file a no-op. Writes to this journal would also be no-ops, and any attempt to read the journal would return SQLITE_IOERR. Or the implementation might recognize that a database file will be doing page-aligned sector reads and writes in a random order and set up its I/O subsystem accordingly.

SQLite might also add one of the following flags to the xOpen method:

- SQLITE_OPEN_DELETEONCLOSE
- SQLITE_OPEN_EXCLUSIVE

The SQLITE_OPEN_DELETEONCLOSE flag means the file should be deleted when it is closed. The SQLITE_OPEN_DELETEONCLOSE will be set for TEMP databases and their journals, transient databases, and subjournals.

The SQLITE_OPEN_EXCLUSIVE flag is always used in conjunction with the SQLITE_OPEN_CREATE flag, which are both directly analogous to the O_EXCL and O_CREAT flags of the POSIX open() API. The SQLITE_OPEN_EXCLUSIVE flag, when paired with the SQLITE_OPEN_CREATE, is used to indicate that file should always be created, and that it is an error if it already exists. It is *not* used to indicate the file should be opened for exclusive access.

At least szOsFile bytes of memory are allocated by SQLite to hold the sqlite3_file structure passed as the third argument to xOpen. The xOpen method does not have to allocate the structure; it should just fill it in. Note that the xOpen method must set the sqlite3_file.pMethods to either a valid sqlite3_io_methods object or to NULL. xOpen must do this even if the open fails. SQLite expects that the sqlite3_file.pMethods element will be valid after xOpen returns regardless of the success or failure of the xOpen call.

The flags argument to xAccess() may be SQLITE_ACCESS_EXISTS to test for the existence of a file, or SQLITE_ACCESS_READWRITE to test whether a file is readable and writable, or SQLITE_ACCESS_READ to test whether a file is at least readable. The SQLITE_ACCESS_READ flag is never actually used and is not implemented in the built-in VFSes of SQLite. The file is named by the second argument and can be a directory. The xAccess method returns SQLITE_OK on success or some non-zero error code if there is an I/O error or if the name of the file given in the second argument is illegal. If SQLITE_OK is returned, then non-zero or zero is written into *pResOut to indicate whether or not the file is accessible.

SQLite will always allocate at least mxPathname+1 bytes for the output buffer xFullPathname. The exact size of the output buffer is also passed as a parameter to both methods. If the output buffer is not large enough, SQLITE_CANTOPEN should be returned. Since this is handled as a fatal error by SQLite, vfs implementations should endeavor to prevent this by setting mxPathname to a sufficiently large value.

The xRandomness(), xSleep(), xCurrentTime(), and xCurrentTimeInt64() interfaces are not strictly a part of the filesystem, but they are included in the VFS structure for completeness. The xRandomness() function attempts to return nBytes bytes of good-quality randomness into zOut. The return value is the actual number of bytes of randomness obtained. The xSleep() method causes the calling thread to sleep for at least the number of microseconds given. The xCurrentTime() method returns a Julian Day Number for the current date and time as a floating point value. The xCurrentTimeInt64() method returns, as an integer, the Julian Day Number multiplied by 86400000 (the number of milliseconds in a 24-hour day). SQLite will use the xCurrentTimeInt64() method to get the current date and time if that method is available (if iVersion is 2 or greater and the function pointer is not NULL) and will fall back to xCurrentTime() if xCurrentTimeInt64() is unavailable.

The xSetSystemCall(), xGetSystemCall(), and xNestSystemCall() interfaces are not used by the SQLite core. These optional interfaces are provided by some VFSes to facilitate testing of the VFS code. By overriding system calls with functions under its control, a test program can simulate faults and error conditions that would otherwise be difficult or impossible to induce. The set of system calls that can be overridden varies from one VFS to another, and from one version of the same VFS to the next. Applications that use these interfaces must be prepared for any or all of these interfaces to be NULL or for their behavior to change from one release to the next. Applications must not attempt to access any of these methods if the iVersion of the VFS is less than 3.

Virtual Table Instance Object

```
struct sqlite3_vtab {
  const sqlite3_module *pModule;  /* The module for this virtual table */
  int nRef;                       /* Number of open cursors */
  char *zErrMsg;                  /* Error message from sqlite3_mprintf() */
  /* Virtual table implementations will typically add additional fields */
};
```

Every virtual table module implementation uses a subclass of this object to describe a particular instance of the virtual table. Each subclass will be tailored to the specific needs of the module implementation. The purpose of this superclass is to define certain fields that are common to all module implementations.

Virtual tables methods can set an error message by assigning a string obtained from sqlite3_mprintf() to zErrMsg. The method should take care that any prior string is freed by a call to sqlite3_free() prior to assigning a new string to zErrMsg. After the error message is delivered up to the client application, the string will be automatically freed by sqlite3_free() and the zErrMsg field will be zeroed.

Obtain Aggregate Function Context

```
void *sqlite3_aggregate_context(sqlite3_context*, int nBytes);
```

Implementations of aggregate SQL functions use this routine to allocate memory for storing their state.

The first time the sqlite3_aggregate_context(C,N) routine is called for a particular aggregate function, SQLite allocates N bytes of memory, zeroes out that memory, and returns a pointer to the new memory. On second and subsequent calls to sqlite3_aggregate_context() for the same aggregate function instance, the same buffer is returned. Sqlite3_aggregate_context() is normally called once for each invocation of the xStep callback and then one last time when the xFinal callback is invoked. When no rows match an aggregate query, the xStep() callback of the aggregate function implementation is never called and xFinal() is called exactly once. In those cases, sqlite3_aggregate_context() might be called for the first time from within xFinal().

The sqlite3_aggregate_context(C,N) routine returns a NULL pointer when first called if N is less than or equal to zero or if a memory allocate error occurs.

The amount of space allocated by sqlite3_aggregate_context(C,N) is determined by the N parameter on first successful call. Changing the value of N in any subsequents call to sqlite3_aggregate_context() within the same aggregate function instance will not resize the memory allocation. Within the xFinal callback, it is customary to set N=0 in calls to sqlite3_aggregate_context(C,N) so that no pointless memory allocations occur.

SQLite automatically frees the memory allocated by sqlite3_aggregate_context() when the aggregate query concludes.

The first parameter must be a copy of the SQL function context that is the first parameter to the xStep or xFinal callback routine that implements the aggregate function.

This routine must be called from the same thread in which the aggregate SQL function is running.

Automatically Load Statically Linked Extensions

```
int sqlite3_auto_extension(void(*xEntryPoint)(void));
```

This interface causes the xEntryPoint() function to be invoked for each new database connection that is created. The idea here is that xEntryPoint() is the entry point for a statically linked SQLite extension that is to be automatically loaded into all new database connections.

Even though the function prototype shows that xEntryPoint() takes no arguments and returns void, SQLite invokes xEntryPoint() with three arguments and expects an integer result as if the signature of the entry point where as follows:

```
int xEntryPoint(
  sqlite3 *db,
  const char **pzErrMsg,
  const struct sqlite3_api_routines *pThunk
);
```

If the xEntryPoint routine encounters an error, it should make *pzErrMsg point to an appropriate error message (obtained from sqlite3_mprintf()) and return an appropriate error code. SQLite ensures that *pzErrMsg is NULL before calling the xEntryPoint(). SQLite will invoke sqlite3_free() on *pzErrMsg after xEntryPoint() returns. If any xEntryPoint() returns an error, the sqlite3_open(), sqlite3_open16(), or sqlite3_open_v2() call that provoked the xEntryPoint() will fail.

Calling sqlite3_auto_extension(X) with an entry point X that is already on the list of automatic extensions is a harmless no-op. No entry point will be called more than once for each database connection that is opened.

See also: sqlite3_reset_auto_extension() and sqlite3_cancel_auto_extension()

Number Of SQL Parameters

```
int sqlite3_bind_parameter_count(sqlite3_stmt*);
```

This routine can be used to find the number of SQL parameters in a prepared statement. SQL parameters are tokens of the form "?", "?NNN", ":AAA", "$AAA", or "@AAA" that serve as placeholders for values that are bound to the parameters at a later time.

This routine actually returns the index of the largest (rightmost) parameter. For all forms except ?NNN, this will correspond to the number of unique parameters. If parameters of the ?NNN form are used, there may be gaps in the list.

See also: sqlite3_bind(), sqlite3_bind_parameter_name(), and sqlite3_bind_parameter_index().

Index Of A Parameter With A Given Name

```
int sqlite3_bind_parameter_index(sqlite3_stmt*, const char *zName);
```

Return the index of an SQL parameter given its name. The index value returned is suitable for use as the second parameter to sqlite3_bind(). A zero is returned if no matching parameter is found. The parameter name must be given in UTF-8 even if the original statement was prepared from UTF-16 text using sqlite3_prepare16_v2() or sqlite3_prepare16_v3().

See also: sqlite3_bind(), sqlite3_bind_parameter_count(), and sqlite3_bind_parameter_name().

Name Of A Host Parameter

```
const char *sqlite3_bind_parameter_name(sqlite3_stmt*, int);
```

The sqlite3_bind_parameter_name(P,N) interface returns the name of the N-th SQL parameter in the prepared statement P. SQL parameters of the form "?NNN" or ":AAA" or "@AAA" or "$AAA" have a name which is the string "?NNN" or ":AAA" or "@AAA" or "$AAA" respectively. In other words, the initial ":" or "$" or "@" or "?" is included as part of the name. Parameters of the form "?" without a following integer have no name and are referred to as "nameless" or "anonymous parameters".

The first host parameter has an index of 1, not 0.

If the value N is out of range or if the N-th parameter is nameless, then NULL is returned. The returned string is always in UTF-8 encoding even if the named parameter was originally specified as UTF-16 in sqlite3_prepare16(), sqlite3_prepare16_v2(), or sqlite3_prepare16_v3().

See also: sqlite3_bind(), sqlite3_bind_parameter_count(), and sqlite3_bind_parameter_index().

Return The Size Of An Open BLOB

```
int sqlite3_blob_bytes(sqlite3_blob *);
```

Returns the size in bytes of the BLOB accessible via the successfully opened BLOB handle in its only argument. The incremental blob I/O routines can only read or overwriting existing blob content; they cannot change the size of a blob.

This routine only works on a BLOB handle which has been created by a prior successful call to sqlite3_blob_open() and which has not been closed by sqlite3_blob_close(). Passing any other pointer in to this routine results in undefined and probably undesirable behavior.

Close A BLOB Handle

```
int sqlite3_blob_close(sqlite3_blob *);
```

This function closes an open BLOB handle. The BLOB handle is closed unconditionally. Even if this routine returns an error code, the handle is still closed.

If the blob handle being closed was opened for read-write access, and if the database is in auto-commit mode and there are no other open read-write blob handles or active write statements, the current transaction is committed. If an error occurs while committing the transaction, an error code is returned and the transaction rolled back.

Calling this function with an argument that is not a NULL pointer or an open blob handle results in undefined behaviour. Calling this routine with a null pointer (such as would be returned by a failed call to sqlite3_blob_open()) is a harmless no-op. Otherwise, if this function is passed a valid open blob handle, the values returned by the sqlite3_errcode() and sqlite3_errmsg() functions are set before returning.

Open A BLOB For Incremental I/O

```
int sqlite3_blob_open(
  sqlite3*,
  const char *zDb,
  const char *zTable,
  const char *zColumn,
  sqlite3_int64 iRow,
  int flags,
  sqlite3_blob **ppBlob
);
```

This interfaces opens a handle to the BLOB located in row iRow, column zColumn, table zTable in database zDb; in other words, the same BLOB that would be selected by:

```
SELECT zColumn FROM zDb.zTable WHERE rowid = iRow;
```

Parameter zDb is not the filename that contains the database, but rather the symbolic name of the database. For attached databases, this is the name that appears after the AS keyword in the ATTACH statement. For the main database file, the database name is "main". For TEMP tables, the database name is "temp".

If the flags parameter is non-zero, then the BLOB is opened for read and write access. If the flags parameter is zero, the BLOB is opened for read-only access.

On success, SQLITE_OK is returned and the new BLOB handle is stored in *ppBlob. Otherwise an error code is returned and, unless the error code is SQLITE_MISUSE, *ppBlob is set to NULL. This means that, provided the API is not misused, it is always safe to call sqlite3_blob_close() on *ppBlob after this function it returns.

This function fails with SQLITE_ERROR if any of the following are true:

- Database zDb does not exist,
- Table zTable does not exist within database zDb,
- Table zTable is a WITHOUT ROWID table,
- Column zColumn does not exist,
- Row iRow is not present in the table,
- The specified column of row iRow contains a value that is not a TEXT or BLOB value,
- Column zColumn is part of an index, PRIMARY KEY or UNIQUE constraint and the blob is being opened for read/write access,
- Foreign key constraints are enabled, column zColumn is part of a child key definition and the blob is being opened for read/write access.

Unless it returns SQLITE_MISUSE, this function sets the database connection error code and message accessible via sqlite3_errcode() and sqlite3_errmsg() and related functions.

A BLOB referenced by sqlite3_blob_open() may be read using the sqlite3_blob_read() interface and modified by using sqlite3_blob_write(). The BLOB handle can be moved to a different row of the same table using the sqlite3_blob_reopen() interface. However, the column, table, or database of a BLOB handle cannot be changed after the BLOB handle is opened.

If the row that a BLOB handle points to is modified by an UPDATE, DELETE, or by ON CONFLICT side-effects then the BLOB handle is marked as "expired". This is true if any column of the row is changed, even a column other than the one the BLOB handle is open on. Calls to sqlite3_blob_read() and sqlite3_blob_write() for an expired BLOB handle fail with a return code of SQLITE_ABORT. Changes written into a BLOB prior to the BLOB expiring are not rolled back by the expiration of the BLOB. Such changes will eventually commit if the transaction continues to completion.

Use the sqlite3_blob_bytes() interface to determine the size of the opened blob. The size of a blob may not be changed by this interface. Use the UPDATE SQL command to change the size of a blob.

The sqlite3_bind_zeroblob() and sqlite3_result_zeroblob() interfaces and the built-in zeroblob SQL function may be used to create a zero-filled blob to read or write using the incremental-blob interface.

To avoid a resource leak, every open BLOB handle should eventually be released by a call to sqlite3_blob_close().

See also: sqlite3_blob_close(), sqlite3_blob_reopen(), sqlite3_blob_read(), sqlite3_blob_bytes(), sqlite3_blob_write().

Read Data From A BLOB Incrementally

```
int sqlite3_blob_read(sqlite3_blob *, void *Z, int N, int iOffset);
```

This function is used to read data from an open BLOB handle into a caller-supplied buffer. N bytes of data are copied into buffer Z from the open BLOB, starting at offset iOffset.

If offset iOffset is less than N bytes from the end of the BLOB, SQLITE_ERROR is returned and no data is read. If N or iOffset is less than zero, SQLITE_ERROR is returned and no data is read. The size of the blob (and hence the maximum value of N+iOffset) can be determined using the sqlite3_blob_bytes() interface.

An attempt to read from an expired BLOB handle fails with an error code of SQLITE_ABORT.

On success, sqlite3_blob_read() returns SQLITE_OK. Otherwise, an error code or an extended error code is returned.

This routine only works on a BLOB handle which has been created by a prior successful call to sqlite3_blob_open() and which has not been closed by sqlite3_blob_close(). Passing any other pointer in to this routine results in undefined and probably undesirable behavior.

See also: sqlite3_blob_write().

Move a BLOB Handle to a New Row

```
int sqlite3_blob_reopen(sqlite3_blob *, sqlite3_int64);
```

This function is used to move an existing BLOB handle so that it points to a different row of the same database table. The new row is identified by the rowid value passed as the second argument. Only the row can be changed. The database, table and column on which the blob handle is open remain the same. Moving an existing BLOB handle to a new row is faster than closing the existing handle and opening a new one.

The new row must meet the same criteria as for sqlite3_blob_open() - it must exist and there must be either a blob or text value stored in the nominated column. If the new row is not present in the table, or if it does not contain a blob or text value, or if another error occurs, an SQLite error code is returned and the blob handle is considered aborted. All subsequent calls to sqlite3_blob_read(), sqlite3_blob_write() or sqlite3_blob_reopen() on an aborted blob handle immediately return SQLITE_ABORT. Calling sqlite3_blob_bytes() on an aborted blob handle always returns zero.

This function sets the database handle error code and message.

Write Data Into A BLOB Incrementally

```
int sqlite3_blob_write(sqlite3_blob *, const void *z, int n, int iOffset);
```

This function is used to write data into an open BLOB handle from a caller-supplied buffer. N bytes of data are copied from the buffer Z into the open BLOB, starting at offset iOffset.

On success, sqlite3_blob_write() returns SQLITE_OK. Otherwise, an error code or an extended error code is returned. Unless SQLITE_MISUSE is returned, this function sets the database connection error code and message accessible via sqlite3_errcode() and sqlite3_errmsg() and related functions.

If the BLOB handle passed as the first argument was not opened for writing (the flags parameter to sqlite3_blob_open() was zero), this function returns SQLITE_READONLY.

This function may only modify the contents of the BLOB; it is not possible to increase the size of a BLOB using this API. If offset iOffset is less than N bytes from the end of the BLOB, SQLITE_ERROR is returned and no data is written. The size of the BLOB (and hence the maximum value of N+iOffset) can be determined using the sqlite3_blob_bytes() interface. If N or iOffset are less than zero SQLITE_ERROR is returned and no data is written.

An attempt to write to an expired BLOB handle fails with an error code of SQLITE_ABORT. Writes to the BLOB that occurred before the BLOB handle expired are not rolled back by the expiration of the handle, though of course those changes might have been overwritten by the statement that expired the BLOB handle or by other independent statements.

This routine only works on a BLOB handle which has been created by a prior successful call to sqlite3_blob_open() and which has not been closed by sqlite3_blob_close(). Passing any other pointer in to this routine results in undefined and probably undesirable behavior.

See also: sqlite3_blob_read().

Set A Busy Timeout

```
int sqlite3_busy_timeout(sqlite3*, int ms);
```

This routine sets a busy handler that sleeps for a specified amount of time when a table is locked. The handler will sleep multiple times until at least "ms" milliseconds of sleeping have accumulated. After at least "ms" milliseconds of sleeping, the handler returns 0 which causes sqlite3_step() to return SQLITE_BUSY.

Calling this routine with an argument less than or equal to zero turns off all busy handlers.

There can only be a single busy handler for a particular database connection at any given moment. If another busy handler was defined (using sqlite3_busy_handler()) prior to calling this routine, that other busy handler is cleared.

See also: PRAGMA busy_timeout

Cancel Automatic Extension Loading

```
int sqlite3_cancel_auto_extension(void(*xEntryPoint)(void));
```

The sqlite3_cancel_auto_extension(X) interface unregisters the initialization routine X that was registered using a prior call to sqlite3_auto_extension(X). The sqlite3_cancel_auto_extension(X) routine returns 1 if initialization routine X was successfully unregistered and it returns 0 if X was not on the list of initialization routines.

Count The Number Of Rows Modified

```
int sqlite3_changes(sqlite3*);
```

This function returns the number of rows modified, inserted or deleted by the most recently completed INSERT, UPDATE or DELETE statement on the database connection specified by the only parameter. Executing any other type of SQL statement does not modify the value returned by this function.

Only changes made directly by the INSERT, UPDATE or DELETE statement are considered - auxiliary changes caused by triggers, foreign key actions or REPLACE constraint resolution are not counted.

Changes to a view that are intercepted by INSTEAD OF triggers are not counted. The value returned by sqlite3_changes() immediately after an INSERT, UPDATE or DELETE statement run on a view is always zero. Only changes made to real tables are counted.

Things are more complicated if the sqlite3_changes() function is executed while a trigger program is running. This may happen if the program uses the changes() SQL function, or if some other callback function invokes sqlite3_changes() directly. Essentially:

- Before entering a trigger program the value returned by sqlite3_changes() function is saved. After the trigger program has finished, the original value is restored.
- Within a trigger program each INSERT, UPDATE and DELETE statement sets the value returned by sqlite3_changes() upon completion as normal. Of course, this value will not include any changes performed by sub-triggers, as the sqlite3_changes() value will be saved and restored after each sub-trigger has run.

This means that if the changes() SQL function (or similar) is used by the first INSERT, UPDATE or DELETE statement within a trigger, it returns the value as set when the calling statement began executing. If it is used by the second or subsequent such statement within a trigger program, the value returned reflects the number of rows modified by the previous INSERT, UPDATE or DELETE statement within the same trigger.

If a separate thread makes changes on the same database connection while sqlite3_changes() is running then the value returned is unpredictable and not meaningful.

See also:

- the sqlite3_total_changes() interface
- the count_changes pragma
- the changes() SQL function
- the data_version pragma

Reset All Bindings On A Prepared Statement

```
int sqlite3_clear_bindings(sqlite3_stmt*);
```

Contrary to the intuition of many, sqlite3_reset() does not reset the bindings on a prepared statement. Use this routine to reset all host parameters to NULL.

Number Of Columns In A Result Set

```
int sqlite3_column_count(sqlite3_stmt *pStmt);
```

Return the number of columns in the result set returned by the prepared statement. If this routine returns 0, that means the prepared statement returns no data (for example an UPDATE). However, just because this routine returns a positive number does not mean that one or more rows of data will be returned. A SELECT statement will always have a positive sqlite3_column_count() but depending on the WHERE clause constraints and the table content, it might return no rows.

See also: sqlite3_data_count()

Configuring The SQLite Library

```
int sqlite3_config(int, ...);
```

The sqlite3_config() interface is used to make global configuration changes to SQLite in order to tune SQLite to the specific needs of the application. The default configuration is recommended for most applications and so this routine is usually not necessary. It is provided to support rare applications with unusual needs.

The sqlite3_config() interface is not threadsafe. The application must ensure that no other SQLite interfaces are invoked by other threads while sqlite3_config() is running.

The sqlite3_config() interface may only be invoked prior to library initialization using sqlite3_initialize() or after shutdown by sqlite3_shutdown(). If sqlite3_config() is called after sqlite3_initialize() and before sqlite3_shutdown() then it will return SQLITE_MISUSE. Note, however, that sqlite3_config() can be called as part of the implementation of an application-defined sqlite3_os_init().

The first argument to sqlite3_config() is an integer configuration option that determines what property of SQLite is to be configured. Subsequent arguments vary depending on the configuration option in the first argument.

When a configuration option is set, sqlite3_config() returns SQLITE_OK. If the option is unknown or SQLite is unable to set the option then this routine returns a non-zero error code.

Database Connection For Functions

```
sqlite3 *sqlite3_context_db_handle(sqlite3_context*);
```

The sqlite3_context_db_handle() interface returns a copy of the pointer to the database connection (the 1st parameter) of the sqlite3_create_function() and sqlite3_create_function16() routines that originally registered the application defined function.

Number of columns in a result set

```
int sqlite3_data_count(sqlite3_stmt *pStmt);
```

The sqlite3_data_count(P) interface returns the number of columns in the current row of the result set of prepared statement P. If prepared statement P does not have results ready to return (via calls to the sqlite3_column() family of interfaces) then sqlite3_data_count(P) returns 0. The sqlite3_data_count(P) routine also returns 0 if P is a NULL pointer. The sqlite3_data_count(P) routine returns 0 if the previous call to sqlite3_step(P) returned SQLITE_DONE. The sqlite3_data_count(P) will return non-zero if previous call to sqlite3_step(P) returned SQLITE_ROW, except in the case of the PRAGMA incremental_vacuum where it always returns zero since each step of that multi-step pragma returns 0 columns of data.

See also: sqlite3_column_count()

Flush caches to disk mid-transaction

```
int sqlite3_db_cacheflush(sqlite3*);
```

If a write-transaction is open on database connection D when the sqlite3_db_cacheflush(D) interface invoked, any dirty pages in the pager-cache that are not currently in use are written out to disk. A dirty page may be in use if a database cursor created by an active SQL statement is reading from it, or if it is page 1 of a database file (page 1 is always "in use"). The sqlite3_db_cacheflush(D) interface flushes caches for all schemas - "main", "temp", and any attached databases.

If this function needs to obtain extra database locks before dirty pages can be flushed to disk, it does so. If those locks cannot be obtained immediately and there is a busy-handler callback configured, it is invoked in the usual manner. If the required lock still cannot be obtained, then the database is skipped and an attempt made to flush any dirty pages belonging to the next (if any) database. If any databases are skipped because locks cannot be obtained, but no other error occurs, this function returns SQLITE_BUSY.

If any other error occurs while flushing dirty pages to disk (for example an IO error or out-of-memory condition), then processing is abandoned and an SQLite error code is returned to the caller immediately.

Otherwise, if no error occurs, sqlite3_db_cacheflush() returns SQLITE_OK.

This function does not set the database handle error code or message returned by the sqlite3_errcode() and sqlite3_errmsg() functions.

Configure database connections

```
int sqlite3_db_config(sqlite3*, int op, ...);
```

The sqlite3_db_config() interface is used to make configuration changes to a database connection. The interface is similar to sqlite3_config() except that the changes apply to a single database connection (specified in the first argument).

The second argument to sqlite3_db_config(D,V,...) is the configuration verb - an integer code that indicates what aspect of the database connection is being configured. Subsequent arguments vary depending on the configuration verb.

Calls to sqlite3_db_config() return SQLITE_OK if and only if the call is considered successful.

Return The Filename For A Database Connection

```
const char *sqlite3_db_filename(sqlite3 *db, const char *zDbName);
```

The sqlite3_db_filename(D,N) interface returns a pointer to the filename associated with database N of connection D. If there is no attached database N on the database connection D, or if database N is a temporary or in-memory database, then this function will return either a NULL pointer or an empty string.

The string value returned by this routine is owned and managed by the database connection. The value will be valid until the database N is DETACH-ed or until the database connection closes.

The filename returned by this function is the output of the xFullPathname method of the VFS. In other words, the filename will be an absolute pathname, even if the filename used to open the database originally was a URI or relative pathname.

If the filename pointer returned by this routine is not NULL, then it can be used as the filename input parameter to these routines:

- sqlite3_uri_parameter()

- sqlite3_uri_boolean()
- sqlite3_uri_int64()
- sqlite3_filename_database()
- sqlite3_filename_journal()
- sqlite3_filename_wal()

Find The Database Handle Of A Prepared Statement

```
sqlite3 *sqlite3_db_handle(sqlite3_stmt*);
```

The sqlite3_db_handle interface returns the database connection handle to which a prepared statement belongs. The database connection returned by sqlite3_db_handle is the same database connection that was the first argument to the sqlite3_prepare_v2() call (or its variants) that was used to create the statement in the first place.

Retrieve the mutex for a database connection

```
sqlite3_mutex *sqlite3_db_mutex(sqlite3*);
```

This interface returns a pointer the sqlite3_mutex object that serializes access to the database connection given in the argument when the threading mode is Serialized. If the threading mode is Single-thread or Multi-thread then this routine returns a NULL pointer.

Determine if a database is read-only

```
int sqlite3_db_readonly(sqlite3 *db, const char *zDbName);
```

The sqlite3_db_readonly(D,N) interface returns 1 if the database N of connection D is read-only, 0 if it is read/write, or -1 if N is not the name of a database on connection D.

Free Memory Used By A Database Connection

```
int sqlite3_db_release_memory(sqlite3*);
```

The sqlite3_db_release_memory(D) interface attempts to free as much heap memory as possible from database connection D. Unlike the sqlite3_release_memory() interface, this interface is in effect even when the SQLITE_ENABLE_MEMORY_MANAGEMENT compile-time option is omitted.

See also: sqlite3_release_memory()

Database Connection Status

```
int sqlite3_db_status(sqlite3*, int op, int *pCur, int *pHiwtr, int resetFlg);
```

This interface is used to retrieve runtime status information about a single database connection. The first argument is the database connection object to be interrogated. The second argument is an integer constant, taken from the set of SQLITE_DBSTATUS options, that determines the parameter to interrogate. The set of SQLITE_DBSTATUS options is likely to grow in future releases of SQLite.

The current value of the requested parameter is written into *pCur and the highest instantaneous value is written into *pHiwtr. If the resetFlg is true, then the highest instantaneous value is reset back down to the current value.

The sqlite3_db_status() routine returns SQLITE_OK on success and a non-zero error code on failure.

See also: sqlite3_status() and sqlite3_stmt_status().

Declare The Schema Of A Virtual Table

```
int sqlite3_declare_vtab(sqlite3*, const char *zSQL);
```

The xCreate and xConnect methods of a virtual table module call this interface to declare the format (the names and datatypes of the columns) of the virtual tables they implement.

Deserialize a database

```
int sqlite3_deserialize(
  sqlite3 *db,            /* The database connection */
  const char *zSchema,    /* Which DB to reopen with the deserialization */
  unsigned char *pData,   /* The serialized database content */
  sqlite3_int64 szDb,     /* Number bytes in the deserialization */
  sqlite3_int64 szBuf,    /* Total size of buffer pData[] */
  unsigned mFlags         /* Zero or more SQLITE_DESERIALIZE_* flags */
);
```

The sqlite3_deserialize(D,S,P,N,M,F) interface causes the database connection D to disconnect from database S and then reopen S as an in-memory database based on the serialization contained in P. The serialized database P is N bytes in size. M is the size of

the buffer P, which might be larger than N. If M is larger than N, and the SQLITE_DESERIALIZE_READONLY bit is not set in F, then SQLite is permitted to add content to the in-memory database as long as the total size does not exceed M bytes.

If the SQLITE_DESERIALIZE_FREEONCLOSE bit is set in F, then SQLite will invoke sqlite3_free() on the serialization buffer when the database connection closes. If the SQLITE_DESERIALIZE_RESIZEABLE bit is set, then SQLite will try to increase the buffer size using sqlite3_realloc64() if writes on the database cause it to grow larger than M bytes.

The sqlite3_deserialize() interface will fail with SQLITE_BUSY if the database is currently in a read transaction or is involved in a backup operation.

If sqlite3_deserialize(D,S,P,N,M,F) fails for any reason and if the SQLITE_DESERIALIZE_FREEONCLOSE bit is set in argument F, then sqlite3_free() is invoked on argument P prior to returning.

This interface is only available if SQLite is compiled with the SQLITE_ENABLE_DESERIALIZE option.

Remove Unnecessary Virtual Table Implementations

```
int sqlite3_drop_modules(
  sqlite3 *db,               /* Remove modules from this connection */
  const char **azKeep        /* Except, do not remove the ones named here */
);
```

The sqlite3_drop_modules(D,L) interface removes all virtual table modules from database connection D except those named on list L. The L parameter must be either NULL or a pointer to an array of pointers to strings where the array is terminated by a single NULL pointer. If the L parameter is NULL, then all virtual table modules are removed.

See also: sqlite3_create_module()

Enable Or Disable Extension Loading

```
int sqlite3_enable_load_extension(sqlite3 *db, int onoff);
```

So as not to open security holes in older applications that are unprepared to deal with extension loading, and as a means of disabling extension loading while evaluating user-entered SQL, the following API is provided to turn the sqlite3_load_extension() mechanism on and off.

Extension loading is off by default. Call the sqlite3_enable_load_extension() routine with onoff==1 to turn extension loading on and call it with onoff==0 to turn it back off again.

This interface enables or disables both the C-API sqlite3_load_extension() and the SQL function load_extension(). Use sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION,..) to enable or disable only the C-API.

Security warning: It is recommended that extension loading be enabled using the SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION method rather than this interface, so the load_extension() SQL function remains disabled. This will prevent SQL injections from giving attackers access to extension loading capabilities.

Enable Or Disable Shared Pager Cache

```
int sqlite3_enable_shared_cache(int);
```

This routine enables or disables the sharing of the database cache and schema data structures between connections to the same database. Sharing is enabled if the argument is true and disabled if the argument is false.

Cache sharing is enabled and disabled for an entire process. This is a change as of SQLite version 3.5.0 (2007-09-04). In prior versions of SQLite, sharing was enabled or disabled for each thread separately.

The cache sharing mode set by this interface effects all subsequent calls to sqlite3_open(), sqlite3_open_v2(), and sqlite3_open16(). Existing database connections continue to use the sharing mode that was in effect at the time they were opened.

This routine returns SQLITE_OK if shared cache was enabled or disabled successfully. An error code is returned otherwise.

Shared cache is disabled by default. It is recommended that it stay that way. In other words, do not use this routine. This interface continues to be provided for historical compatibility, but its use is discouraged. Any use of shared cache is discouraged. If shared cache must be used, it is recommended that shared cache only be enabled for individual database connections using the sqlite3_open_v2() interface with the SQLITE_OPEN_SHAREDCACHE flag.

Note: This method is disabled on MacOS X 10.7 and iOS version 5.0 and will always return SQLITE_MISUSE. On those systems, shared cache mode should be enabled per-database connection via sqlite3_open_v2() with SQLITE_OPEN_SHAREDCACHE.

This interface is threadsafe on processors where writing a 32-bit integer is atomic.

See Also: SQLite Shared-Cache Mode

One-Step Query Execution Interface

```
int sqlite3_exec(
  sqlite3*,                                  /* An open database */
  const char *sql,                           /* SQL to be evaluated */
  int (*callback)(void*,int,char**,char**),  /* Callback function */
  void *,                                    /* 1st argument to callback */
  char **errmsg                              /* Error msg written here */
);
```

The sqlite3_exec() interface is a convenience wrapper around sqlite3_prepare_v2(), sqlite3_step(), and sqlite3_finalize(), that allows an application to run multiple statements of SQL without having to use a lot of C code.

The sqlite3_exec() interface runs zero or more UTF-8 encoded, semicolon-separate SQL statements passed into its 2nd argument, in the context of the database connection passed in as its 1st argument. If the callback function of the 3rd argument to sqlite3_exec() is not NULL, then it is invoked for each result row coming out of the evaluated SQL statements. The 4th argument to sqlite3_exec() is relayed through to the 1st argument of each callback invocation. If the callback pointer to sqlite3_exec() is NULL, then no callback is ever invoked and result rows are ignored.

If an error occurs while evaluating the SQL statements passed into sqlite3_exec(), then execution of the current statement stops and subsequent statements are skipped. If the 5th parameter to sqlite3_exec() is not NULL then any error message is written into memory obtained from sqlite3_malloc() and passed back through the 5th parameter. To avoid memory leaks, the application should invoke sqlite3_free() on error message strings returned through the 5th parameter of sqlite3_exec() after the error message string is no longer needed. If the 5th parameter to sqlite3_exec() is not NULL and no errors occur, then sqlite3_exec() sets the pointer in its 5th parameter to NULL before returning.

If an sqlite3_exec() callback returns non-zero, the sqlite3_exec() routine returns SQLITE_ABORT without invoking the callback again and without running any subsequent SQL statements.

The 2nd argument to the sqlite3_exec() callback function is the number of columns in the result. The 3rd argument to the sqlite3_exec() callback is an array of pointers to strings obtained as if from sqlite3_column_text(), one for each column. If an element of a result row is NULL then the corresponding string pointer for the sqlite3_exec() callback is a NULL pointer. The 4th argument to the sqlite3_exec() callback is an array of pointers to strings where each entry represents the name of corresponding result column as obtained from sqlite3_column_name().

If the 2nd parameter to sqlite3_exec() is a NULL pointer, a pointer to an empty string, or a pointer that contains only whitespace and/or SQL comments, then no SQL statements are evaluated and the database is not changed.

Restrictions:

- The application must ensure that the 1st parameter to sqlite3_exec() is a valid and open database connection.
- The application must not close the database connection specified by the 1st parameter to sqlite3_exec() while sqlite3_exec() is running.
- The application must not modify the SQL statement text passed into the 2nd parameter of sqlite3_exec() while sqlite3_exec() is running.

Enable Or Disable Extended Result Codes

```
int sqlite3_extended_result_codes(sqlite3*, int onoff);
```

The sqlite3_extended_result_codes() routine enables or disables the extended result codes feature of SQLite. The extended result codes are disabled by default for historical compatibility.

Destroy A Prepared Statement Object

```
int sqlite3_finalize(sqlite3_stmt *pStmt);
```

The sqlite3_finalize() function is called to delete a prepared statement. If the most recent evaluation of the statement encountered no errors or if the statement is never been evaluated, then sqlite3_finalize() returns SQLITE_OK. If the most recent evaluation of statement S failed, then sqlite3_finalize(S) returns the appropriate error code or extended error code.

The sqlite3_finalize(S) routine can be called at any point during the life cycle of prepared statement S: before statement S is ever evaluated, after one or more calls to sqlite3_reset(), or after any call to sqlite3_step() regardless of whether or not the statement has completed execution.

Invoking sqlite3_finalize() on a NULL pointer is a harmless no-op.

The application must finalize every prepared statement in order to avoid resource leaks. It is a grievous error for the application to try to use a prepared statement after it has been finalized. Any use of a prepared statement after it has been finalized can result in undefined and undesirable behavior such as segfaults and heap corruption.

Interrupt A Long-Running Query

```
void sqlite3_interrupt(sqlite3*);
```

This function causes any pending database operation to abort and return at its earliest opportunity. This routine is typically called in response to a user action such as pressing "Cancel" or Ctrl-C where the user wants a long query operation to halt immediately.

It is safe to call this routine from a thread different from the thread that is currently running the database operation. But it is not safe to call this routine with a database connection that is closed or might close before sqlite3_interrupt() returns.

If an SQL operation is very nearly finished at the time when sqlite3_interrupt() is called, then it might not have an opportunity to be interrupted and might continue to completion.

An SQL operation that is interrupted will return SQLITE_INTERRUPT. If the interrupted SQL operation is an INSERT, UPDATE, or DELETE that is inside an explicit transaction, then the entire transaction will be rolled back automatically.

The sqlite3_interrupt(D) call is in effect until all currently running SQL statements on database connection D complete. Any new SQL statements that are started after the sqlite3_interrupt() call and before the running statement count reaches zero are interrupted as if they had been running prior to the sqlite3_interrupt() call. New SQL statements that are started after the running statement count reaches zero are not effected by the sqlite3_interrupt(). A call to sqlite3_interrupt(D) that occurs when there are

no running SQL statements is a no-op and has no effect on SQL statements that are started after the sqlite3_interrupt() call returns.

Last Insert Rowid

```
sqlite3_int64 sqlite3_last_insert_rowid(sqlite3*);
```

Each entry in most SQLite tables (except for WITHOUT ROWID tables) has a unique 64-bit signed integer key called the "rowid". The rowid is always available as an undeclared column named ROWID, OID, or _ROWID_ as long as those names are not also used by explicitly declared columns. If the table has a column of type INTEGER PRIMARY KEY then that column is another alias for the rowid.

The sqlite3_last_insert_rowid(D) interface usually returns the rowid of the most recent successful INSERT into a rowid table or virtual table on database connection D. Inserts into WITHOUT ROWID tables are not recorded. If no successful INSERTs into rowid tables have ever occurred on the database connection D, then sqlite3_last_insert_rowid(D) returns zero.

As well as being set automatically as rows are inserted into database tables, the value returned by this function may be set explicitly by sqlite3_set_last_insert_rowid()

Some virtual table implementations may INSERT rows into rowid tables as part of committing a transaction (e.g. to flush data accumulated in memory to disk). In this case subsequent calls to this function return the rowid associated with these internal INSERT operations, which leads to unintuitive results. Virtual table implementations that do write to rowid tables in this way can avoid this problem by restoring the original rowid value using sqlite3_set_last_insert_rowid() before returning control to the user.

If an INSERT occurs within a trigger then this routine will return the rowid of the inserted row as long as the trigger is running. Once the trigger program ends, the value returned by this routine reverts to what it was before the trigger was fired.

An INSERT that fails due to a constraint violation is not a successful INSERT and does not change the value returned by this routine. Thus INSERT OR FAIL, INSERT OR IGNORE, INSERT OR ROLLBACK, and INSERT OR ABORT make no changes to the return value of this routine when their insertion fails. When INSERT OR REPLACE encounters a constraint violation, it does not fail. The INSERT continues to completion after deleting rows that caused the constraint problem so INSERT OR REPLACE will always change the return value of this interface.

For the purposes of this routine, an INSERT is considered to be successful even if it is subsequently rolled back.

This function is accessible to SQL statements via the last_insert_rowid() SQL function.

If a separate thread performs a new INSERT on the same database connection while the sqlite3_last_insert_rowid() function is running and thus changes the last insert rowid, then the value returned by sqlite3_last_insert_rowid() is unpredictable and might not equal either the old or the new last insert rowid.

Run-time Limits

```
int sqlite3_limit(sqlite3*, int id, int newVal);
```

This interface allows the size of various constructs to be limited on a connection by connection basis. The first parameter is the database connection whose limit is to be set or queried. The second parameter is one of the limit categories that define a class of constructs to be size limited. The third parameter is the new limit for that construct.

If the new limit is a negative number, the limit is unchanged. For each limit category SQLITE_LIMIT_*NAME* there is a hard upper bound set at compile-time by a C preprocessor macro called SQLITE_MAX_*NAME*. (The "_LIMIT_" in the name is changed to "_MAX_".) Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound.

Regardless of whether or not the limit was changed, the sqlite3_limit() interface returns the prior value of the limit. Hence, to find the current value of a limit without changing it, simply invoke this interface with the third parameter set to -1.

Run-time limits are intended for use in applications that manage both their own internal database and also databases that are controlled by untrusted external sources. An example application might be a web browser that has its own databases for storing history and separate databases controlled by JavaScript applications downloaded off the Internet. The internal databases can be given the large, default limits. Databases managed by external sources can be given much smaller limits designed to prevent a denial of service attack. Developers might also want to use the sqlite3_set_authorizer() interface to further control untrusted SQL. The size of the database created by an untrusted script can be contained using the max_page_count PRAGMA.

New run-time limit categories may be added in future releases.

Load An Extension

```
int sqlite3_load_extension(
  sqlite3 *db,          /* Load the extension into this database connection */
  const char *zFile,    /* Name of the shared library containing extension */
  const char *zProc,    /* Entry point.  Derived from zFile if 0 */
  char **pzErrMsg       /* Put error message here if not 0 */
);
```

This interface loads an SQLite extension library from the named file.

The sqlite3_load_extension() interface attempts to load an SQLite extension library contained in the file zFile. If the file cannot be loaded directly, attempts are made to load with various operating-system specific extensions added. So for example, if "samplelib" cannot be loaded, then names like "samplelib.so" or "samplelib.dylib" or "samplelib.dll" might be tried also.

The entry point is zProc. zProc may be 0, in which case SQLite will try to come up with an entry point name on its own. It first tries "sqlite3_extension_init". If that does not work, it constructs a name "sqlite3_X_init" where the X is consists of the lower-case

equivalent of all ASCII alphabetic characters in the filename from the last "/" to the first following "." and omitting any initial "lib". The sqlite3_load_extension() interface returns SQLITE_OK on success and SQLITE_ERROR if something goes wrong. If an error occurs and pzErrMsg is not 0, then the sqlite3_load_extension() interface shall attempt to fill *pzErrMsg with error message text stored in memory obtained from sqlite3_malloc(). The calling function should free this memory by calling sqlite3_free().

Extension loading must be enabled using sqlite3_enable_load_extension() or sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION,1,NULL) prior to calling this API, otherwise an error will be returned.

Security warning: It is recommended that the SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION method be used to enable only this interface. The use of the sqlite3_enable_load_extension() interface should be avoided. This will keep the SQL function load_extension() disabled and prevent SQL injections from giving attackers access to extension loading capabilities.

See also the load_extension() SQL function.

Error Logging Interface

```
void sqlite3_log(int iErrCode, const char *zFormat, ...);
```

The sqlite3_log() interface writes a message into the error log established by the SQLITE_CONFIG_LOG option to sqlite3_config(). If logging is enabled, the zFormat string and subsequent arguments are used with sqlite3_snprintf() to generate the final output string.

The sqlite3_log() interface is intended for use by extensions such as virtual tables, collating functions, and SQL functions. While there is nothing to prevent an application from calling sqlite3_log(), doing so is considered bad form.

The zFormat string must not be NULL.

To avoid deadlocks and other threading problems, the sqlite3_log() routine will not use dynamically allocated memory. The log message is stored in a fixed-length buffer on the stack. If the log message is longer than a few hundred characters, it will be truncated to the length of the buffer.

Find the next prepared statement

```
sqlite3_stmt *sqlite3_next_stmt(sqlite3 *pDb, sqlite3_stmt *pStmt);
```

This interface returns a pointer to the next prepared statement after pStmt associated with the database connection pDb. If pStmt is NULL then this interface returns a pointer to the first prepared statement associated with the database connection pDb. If no prepared statement satisfies the conditions of this routine, it returns NULL.

The database connection pointer D in a call to sqlite3_next_stmt(D,S) must refer to an open database connection and in particular must not be a NULL pointer.

Overload A Function For A Virtual Table

```
int sqlite3_overload_function(sqlite3*, const char *zFuncName, int nArg);
```

Virtual tables can provide alternative implementations of functions using the xFindFunction method of the virtual table module. But global versions of those functions must exist in order to be overloaded.

This API makes sure a global version of a function with a particular name and number of parameters exists. If no such function exists before this API is called, a new function is created. The implementation of the new function always causes an exception to be thrown. So the new function is not good for anything by itself. Its only purpose is to be a placeholder function that can be overloaded by a virtual table.

Query Progress Callbacks

```
void sqlite3_progress_handler(sqlite3*, int, int(*)(void*), void*);
```

The sqlite3_progress_handler(D,N,X,P) interface causes the callback function X to be invoked periodically during long running calls to sqlite3_exec(), sqlite3_step() and sqlite3_get_table() for database connection D. An example use for this interface is to keep a GUI updated during a large query.

The parameter P is passed through as the only parameter to the callback function X. The parameter N is the approximate number of virtual machine instructions that are evaluated between successive invocations of the callback X. If N is less than one then the progress handler is disabled.

Only a single progress handler may be defined at one time per database connection; setting a new progress handler cancels the old one. Setting parameter X to NULL disables the progress handler. The progress handler is also disabled by setting N to a value less than 1.

If the progress callback returns non-zero, the operation is interrupted. This feature can be used to implement a "Cancel" button on a GUI progress dialog box.

The progress handler callback must not do anything that will modify the database connection that invoked the progress handler. Note that sqlite3_prepare_v2() and sqlite3_step() both modify their database connections for the meaning of "modify" in this paragraph.

Pseudo-Random Number Generator

```
void sqlite3_randomness(int N, void *P);
```

SQLite contains a high-quality pseudo-random number generator (PRNG) used to select random ROWIDs when inserting new records into a table that already uses the largest possible ROWID. The PRNG is also used for the built-in random() and randomblob() SQL functions. This interface allows applications to access the same PRNG for other purposes.

A call to this routine stores N bytes of randomness into buffer P. The P parameter can be a NULL pointer.

If this routine has not been previously called or if the previous call had N less than one or a NULL pointer for P, then the PRNG is seeded using randomness obtained from the xRandomness method of the default sqlite3_vfs object. If the previous call to this routine had an N of 1 or more and a non-NULL P then the pseudo-randomness is generated internally and without recourse to the sqlite3_vfs xRandomness method.

Attempt To Free Heap Memory

```
int sqlite3_release_memory(int);
```

The sqlite3_release_memory() interface attempts to free N bytes of heap memory by deallocating non-essential memory allocations held by the database library. Memory used to cache database pages to improve performance is an example of non-essential memory. sqlite3_release_memory() returns the number of bytes actually freed, which might be more or less than the amount requested. The sqlite3_release_memory() routine is a no-op returning zero if SQLite is not compiled with SQLITE_ENABLE_MEMORY_MANAGEMENT.

See also: sqlite3_db_release_memory()

Reset A Prepared Statement Object

```
int sqlite3_reset(sqlite3_stmt *pStmt);
```

The sqlite3_reset() function is called to reset a prepared statement object back to its initial state, ready to be re-executed. Any SQL statement variables that had values bound to them using the sqlite3_bind_*() API retain their values. Use sqlite3_clear_bindings() to reset the bindings.

The sqlite3_reset(S) interface resets the prepared statement S back to the beginning of its program.

If the most recent call to sqlite3_step(S) for the prepared statement S returned SQLITE_ROW or SQLITE_DONE, or if sqlite3_step(S) has never before been called on S, then sqlite3_reset(S) returns SQLITE_OK.

If the most recent call to sqlite3_step(S) for the prepared statement S indicated an error, then sqlite3_reset(S) returns an appropriate error code.

The sqlite3_reset(S) interface does not change the values of any bindings on the prepared statement S.

Reset Automatic Extension Loading

```
void sqlite3_reset_auto_extension(void);
```

This interface disables all automatic extensions previously registered using sqlite3_auto_extension().

Setting The Subtype Of An SQL Function

```
void sqlite3_result_subtype(sqlite3_context*,unsigned int);
```

The sqlite3_result_subtype(C,T) function causes the subtype of the result from the application-defined SQL function with sqlite3_context C to be the value T. Only the lower 8 bits of the subtype T are preserved in current versions of SQLite; higher order bits are discarded. The number of subtype bytes preserved by SQLite might increase in future releases of SQLite.

Serialize a database

```
unsigned char *sqlite3_serialize(
  sqlite3 *db,           /* The database connection */
  const char *zSchema,   /* Which DB to serialize. ex: "main", "temp", ... */
  sqlite3_int64 *piSize, /* Write size of the DB here, if not NULL */
  unsigned int mFlags    /* Zero or more SQLITE_SERIALIZE_* flags */
);
```

The sqlite3_serialize(D,S,P,F) interface returns a pointer to memory that is a serialization of the S database on database connection D. If P is not a NULL pointer, then the size of the database in bytes is written into *P.

For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "TEMP" database, the serialization is the same sequence of bytes which would be written to disk if that database where backed up to disk.

The usual case is that sqlite3_serialize() copies the serialization of the database into memory obtained from sqlite3_malloc64() and returns a pointer to that memory. The caller is responsible for freeing the returned value to avoid a memory leak. However, if the F argument contains the SQLITE_SERIALIZE_NOCOPY bit, then no memory allocations are made, and the sqlite3_serialize() function will return a pointer to the contiguous memory representation of the database that SQLite is currently using for that database, or NULL if the no such contiguous memory representation of the database exists. A contiguous memory representation of the

database will usually only exist if there has been a prior call to sqlite3_deserialize(D,S,...) with the same values of D and S. The size of the database is written into *P even if the SQLITE_SERIALIZE_NOCOPY bit is set but no contiguous copy of the database exists.

A call to sqlite3_serialize(D,S,P,F) might return NULL even if the SQLITE_SERIALIZE_NOCOPY bit is omitted from argument F if a memory allocation error occurs.

This interface is only available if SQLite is compiled with the SQLITE_ENABLE_DESERIALIZE option.

Set the Last Insert Rowid value.

```
void sqlite3_set_last_insert_rowid(sqlite3*,sqlite3_int64);
```

The sqlite3_set_last_insert_rowid(D, R) method allows the application to set the value returned by calling sqlite3_last_insert_rowid(D) to R without inserting a row into the database.

Suspend Execution For A Short Time

```
int sqlite3_sleep(int);
```

The sqlite3_sleep() function causes the current thread to suspend execution for at least a number of milliseconds specified in its parameter.

If the operating system does not support sleep requests with millisecond time resolution, then the time will be rounded up to the nearest second. The number of milliseconds of sleep actually requested from the operating system is returned.

SQLite implements this interface by calling the xSleep() method of the default sqlite3_vfs object. If the xSleep() method of the default VFS is not implemented correctly, or not implemented at all, then the behavior of sqlite3_sleep() may deviate from the description in the previous paragraphs.

Compare the ages of two snapshot handles.

```
int sqlite3_snapshot_cmp(
  sqlite3_snapshot *p1,
  sqlite3_snapshot *p2
);
```

The sqlite3_snapshot_cmp(P1, P2) interface is used to compare the ages of two valid snapshot handles.

If the two snapshot handles are not associated with the same database file, the result of the comparison is undefined.

Additionally, the result of the comparison is only valid if both of the snapshot handles were obtained by calling sqlite3_snapshot_get() since the last time the wal file was deleted. The wal file is deleted when the database is changed back to rollback mode or when the number of database clients drops to zero. If either snapshot handle was obtained before the wal file was last deleted, the value returned by this function is undefined.

Otherwise, this API returns a negative value if P1 refers to an older snapshot than P2, zero if the two handles refer to the same database snapshot, and a positive value if P1 is a newer snapshot than P2.

This interface is only available if SQLite is compiled with the SQLITE_ENABLE_SNAPSHOT option.

Destroy a snapshot

```
void sqlite3_snapshot_free(sqlite3_snapshot*);
```

The sqlite3_snapshot_free(P) interface destroys sqlite3_snapshot P. The application must eventually free every sqlite3_snapshot object using this routine to avoid a memory leak.

The sqlite3_snapshot_free() interface is only available when the SQLITE_ENABLE_SNAPSHOT compile-time option is used.

Record A Database Snapshot

```
int sqlite3_snapshot_get(
  sqlite3 *db,
  const char *zSchema,
  sqlite3_snapshot **ppSnapshot
);
```

The sqlite3_snapshot_get(D,S,P) interface attempts to make a new sqlite3_snapshot object that records the current state of schema S in database connection D. On success, the sqlite3_snapshot_get(D,S,P) interface writes a pointer to the newly created sqlite3_snapshot object into *P and returns SQLITE_OK. If there is not already a read-transaction open on schema S when this function is called, one is opened automatically.

The following must be true for this function to succeed. If any of the following statements are false when sqlite3_snapshot_get() is called, SQLITE_ERROR is returned. The final value of *P is undefined in this case.

- The database handle must not be in autocommit mode.
- Schema S of database connection D must be a WAL mode database.

- There must not be a write transaction open on schema S of database connection D.
- One or more transactions must have been written to the current wal file since it was created on disk (by any connection). This means that a snapshot cannot be taken on a wal mode database with no wal file immediately after it is first opened. At least one transaction must be written to it first.

This function may also return SQLITE_NOMEM. If it is called with the database handle in autocommit mode but fails for some other reason, whether or not a read transaction is opened on schema S is undefined.

The sqlite3_snapshot object returned from a successful call to sqlite3_snapshot_get() must be freed using sqlite3_snapshot_free() to avoid a memory leak.

The sqlite3_snapshot_get() interface is only available when the SQLITE_ENABLE_SNAPSHOT compile-time option is used.

Start a read transaction on an historical snapshot

```
int sqlite3_snapshot_open(
  sqlite3 *db,
  const char *zSchema,
  sqlite3_snapshot *pSnapshot
);
```

The sqlite3_snapshot_open(D,S,P) interface either starts a new read transaction or upgrades an existing one for schema S of database connection D such that the read transaction refers to historical snapshot P, rather than the most recent change to the database. The sqlite3_snapshot_open() interface returns SQLITE_OK on success or an appropriate error code if it fails.

In order to succeed, the database connection must not be in autocommit mode when sqlite3_snapshot_open(D,S,P) is called. If there is already a read transaction open on schema S, then the database handle must have no active statements (SELECT statements that have been passed to sqlite3_step() but not sqlite3_reset() or sqlite3_finalize()). SQLITE_ERROR is returned if either of these conditions is violated, or if schema S does not exist, or if the snapshot object is invalid.

A call to sqlite3_snapshot_open() will fail to open if the specified snapshot has been overwritten by a checkpoint. In this case SQLITE_ERROR_SNAPSHOT is returned.

If there is already a read transaction open when this function is invoked, then the same read transaction remains open (on the same database snapshot) if SQLITE_ERROR, SQLITE_BUSY or SQLITE_ERROR_SNAPSHOT is returned. If another error code - for example SQLITE_PROTOCOL or an SQLITE_IOERR error code - is returned, then the final state of the read transaction is undefined. If SQLITE_OK is returned, then the read transaction is now open on database snapshot P.

A call to sqlite3_snapshot_open(D,S,P) will fail if the database connection D does not know that the database file for schema S is in WAL mode. A database connection might not know that the database file is in WAL mode if there has been no prior I/O on that database connection, or if the database entered WAL mode after the most recent I/O on the database connection. (Hint: Run "PRAGMA application_id" against a newly opened database connection in order to make it ready to use snapshots.)

The sqlite3_snapshot_open() interface is only available when the SQLITE_ENABLE_SNAPSHOT compile-time option is used.

Recover snapshots from a wal file

```
int sqlite3_snapshot_recover(sqlite3 *db, const char *zDb);
```

If a WAL file remains on disk after all database connections close (either through the use of the SQLITE_FCNTL_PERSIST_WAL file control or because the last process to have the database opened exited without calling sqlite3_close()) and a new connection is subsequently opened on that database and WAL file, the sqlite3_snapshot_open() interface will only be able to open the last transaction added to the WAL file even though the WAL file contains other valid transactions.

This function attempts to scan the WAL file associated with database zDb of database handle db and make all valid snapshots available to sqlite3_snapshot_open(). It is an error if there is already a read transaction open on the database, or if the database is not a WAL mode database.

SQLITE_OK is returned if successful, or an SQLite error code otherwise.

This interface is only available if SQLite is compiled with the SQLITE_ENABLE_SNAPSHOT option.

Deprecated Soft Heap Limit Interface

```
void sqlite3_soft_heap_limit(int N);
```

This is a deprecated version of the sqlite3_soft_heap_limit64() interface. This routine is provided for historical compatibility only. All new applications should use the sqlite3_soft_heap_limit64() interface rather than this one.

Evaluate An SQL Statement

```
int sqlite3_step(sqlite3_stmt*);
```

After a prepared statement has been prepared using any of sqlite3_prepare_v2(), sqlite3_prepare_v3(), sqlite3_prepare16_v2(), or sqlite3_prepare16_v3() or one of the legacy interfaces sqlite3_prepare() or sqlite3_prepare16(), this function must be called one or more times to evaluate the statement.

The details of the behavior of the sqlite3_step() interface depend on whether the statement was prepared using the newer "vX" interfaces sqlite3_prepare_v3(), sqlite3_prepare_v2(), sqlite3_prepare16_v3(), sqlite3_prepare16_v2() or the older legacy

interfaces sqlite3_prepare() and sqlite3_prepare16(). The use of the new "vX" interface is recommended for new applications but the legacy interface will continue to be supported.

In the legacy interface, the return value will be either SQLITE_BUSY, SQLITE_DONE, SQLITE_ROW, SQLITE_ERROR, or SQLITE_MISUSE. With the "v2" interface, any of the other result codes or extended result codes might be returned as well.

SQLITE_BUSY means that the database engine was unable to acquire the database locks it needs to do its job. If the statement is a COMMIT or occurs outside of an explicit transaction, then you can retry the statement. If the statement is not a COMMIT and occurs within an explicit transaction then you should rollback the transaction before continuing.

SQLITE_DONE means that the statement has finished executing successfully. sqlite3_step() should not be called again on this virtual machine without first calling sqlite3_reset() to reset the virtual machine back to its initial state.

If the SQL statement being executed returns any data, then SQLITE_ROW is returned each time a new row of data is ready for processing by the caller. The values may be accessed using the column access functions. sqlite3_step() is called again to retrieve the next row of data.

SQLITE_ERROR means that a run-time error (such as a constraint violation) has occurred. sqlite3_step() should not be called again on the VM. More information may be found by calling sqlite3_errmsg(). With the legacy interface, a more specific error code (for example, SQLITE_INTERRUPT, SQLITE_SCHEMA, SQLITE_CORRUPT, and so forth) can be obtained by calling sqlite3_reset() on the prepared statement. In the "v2" interface, the more specific error code is returned directly by sqlite3_step().

SQLITE_MISUSE means that the this routine was called inappropriately. Perhaps it was called on a prepared statement that has already been finalized or on one that had previously returned SQLITE_ERROR or SQLITE_DONE. Or it could be the case that the same database connection is being used by two or more threads at the same moment in time.

For all versions of SQLite up to and including 3.6.23.1, a call to sqlite3_reset() was required after sqlite3_step() returned anything other than SQLITE_ROW before any subsequent invocation of sqlite3_step(). Failure to reset the prepared statement using sqlite3_reset() would result in an SQLITE_MISUSE return from sqlite3_step(). But after version 3.6.23.1 (2010-03-26, sqlite3_step() began calling sqlite3_reset() automatically in this circumstance rather than returning SQLITE_MISUSE. This is not considered a compatibility break because any application that ever receives an SQLITE_MISUSE error is broken by definition. The SQLITE_OMIT_AUTORESET compile-time option can be used to restore the legacy behavior.

Goofy Interface Alert: In the legacy interface, the sqlite3_step() API always returns a generic error code, SQLITE_ERROR, following any error other than SQLITE_BUSY and SQLITE_MISUSE. You must call sqlite3_reset() or sqlite3_finalize() in order to find one of the specific error codes that better describes the error. We admit that this is a goofy design. The problem has been fixed with the "v2" interface. If you prepare all of your SQL statements using sqlite3_prepare_v3() or sqlite3_prepare_v2() or sqlite3_prepare16_v2() or sqlite3_prepare16_v3() instead of the legacy sqlite3_prepare() and sqlite3_prepare16() interfaces, then the more specific error codes are returned directly by sqlite3_step(). The use of the "vX" interfaces is recommended.

Determine If A Prepared Statement Has Been Reset

```
int sqlite3_stmt_busy(sqlite3_stmt*);
```

The sqlite3_stmt_busy(S) interface returns true (non-zero) if the prepared statement S has been stepped at least once using sqlite3_step(S) but has neither run to completion (returned SQLITE_DONE from sqlite3_step(S)) nor been reset using sqlite3_reset(S). The sqlite3_stmt_busy(S) interface returns false if S is a NULL pointer. If S is not a NULL pointer and is not a pointer to a valid prepared statement object, then the behavior is undefined and probably undesirable.

This interface can be used in combination sqlite3_next_stmt() to locate all prepared statements associated with a database connection that are in need of being reset. This can be used, for example, in diagnostic routines to search for prepared statements that are holding a transaction open.

Query The EXPLAIN Setting For A Prepared Statement

```
int sqlite3_stmt_isexplain(sqlite3_stmt *pStmt);
```

The sqlite3_stmt_isexplain(S) interface returns 1 if the prepared statement S is an EXPLAIN statement, or 2 if the statement S is an EXPLAIN QUERY PLAN. The sqlite3_stmt_isexplain(S) interface returns 0 if S is an ordinary statement or a NULL pointer.

Determine If An SQL Statement Writes The Database

```
int sqlite3_stmt_readonly(sqlite3_stmt *pStmt);
```

The sqlite3_stmt_readonly(X) interface returns true (non-zero) if and only if the prepared statement X makes no direct changes to the content of the database file.

Note that application-defined SQL functions or virtual tables might change the database indirectly as a side effect. For example, if an application defines a function "eval()" that calls sqlite3_exec(), then the following SQL statement would change the database file through side-effects:

```
SELECT eval('DELETE FROM t1') FROM t2;
```

But because the SELECT statement does not change the database file directly, sqlite3_stmt_readonly() would still return true.

Transaction control statements such as BEGIN, COMMIT, ROLLBACK, SAVEPOINT, and RELEASE cause sqlite3_stmt_readonly() to return true, since the statements themselves do not actually modify the database but rather they control the timing of when other statements modify the database. The ATTACH and DETACH statements also cause sqlite3_stmt_readonly() to return true since, while those statements change the configuration of a database connection, they do not make changes to the content of the database files on disk. The sqlite3_stmt_readonly() interface returns true for BEGIN since BEGIN merely sets internal flags, but the

BEGIN IMMEDIATE and BEGIN EXCLUSIVE commands do touch the database and so sqlite3_stmt_readonly() returns false for those commands.

Prepared Statement Scan Status

```
int sqlite3_stmt_scanstatus(
  sqlite3_stmt *pStmt,      /* Prepared statement for which info desired */
  int idx,                  /* Index of loop to report on */
  int iScanStatusOp,        /* Information desired.  SQLITE_SCANSTAT_* */
  void *pOut                /* Result written here */
);
```

This interface returns information about the predicted and measured performance for pStmt. Advanced applications can use this interface to compare the predicted and the measured performance and issue warnings and/or rerun ANALYZE if discrepancies are found.

Since this interface is expected to be rarely used, it is only available if SQLite is compiled using the SQLITE_ENABLE_STMT_SCANSTATUS compile-time option.

The "iScanStatusOp" parameter determines which status information to return. The "iScanStatusOp" must be one of the scanstatus options or the behavior of this interface is undefined. The requested measurement is written into a variable pointed to by the "pOut" parameter. Parameter "idx" identifies the specific loop to retrieve statistics for. Loops are numbered starting from zero. If idx is out of range - less than zero or greater than or equal to the total number of loops used to implement the statement - a non-zero value is returned and the variable that pOut points to is unchanged.

Statistics might not be available for all loops in all statements. In cases where there exist loops with no available statistics, this function behaves as if the loop did not exist - it returns non-zero and leave the variable that pOut points to unchanged.

See also: sqlite3_stmt_scanstatus_reset()

Zero Scan-Status Counters

```
void sqlite3_stmt_scanstatus_reset(sqlite3_stmt*);
```

Zero all sqlite3_stmt_scanstatus() related event counters.

This API is only available if the library is built with pre-processor symbol SQLITE_ENABLE_STMT_SCANSTATUS defined.

Prepared Statement Status

```
int sqlite3_stmt_status(sqlite3_stmt*, int op,int resetFlg);
```

Each prepared statement maintains various SQLITE_STMTSTATUS counters that measure the number of times it has performed specific operations. These counters can be used to monitor the performance characteristics of the prepared statements. For example, if the number of table steps greatly exceeds the number of table searches or result rows, that would tend to indicate that the prepared statement is using a full table scan rather than an index.

This interface is used to retrieve and reset counter values from a prepared statement. The first argument is the prepared statement object to be interrogated. The second argument is an integer code for a specific SQLITE_STMTSTATUS counter to be interrogated. The current value of the requested counter is returned. If the resetFlg is true, then the counter is reset to zero after this interface call returns.

See also: sqlite3_status() and sqlite3_db_status().

Finalize A Dynamic String

```
char *sqlite3_str_finish(sqlite3_str*);
```

The sqlite3_str_finish(X) interface destroys the sqlite3_str object X and returns a pointer to a memory buffer obtained from sqlite3_malloc64() that contains the constructed string. The calling application should pass the returned value to sqlite3_free() to avoid a memory leak. The sqlite3_str_finish(X) interface may return a NULL pointer if any errors were encountered during construction of the string. The sqlite3_str_finish(X) interface will also return a NULL pointer if the string in sqlite3_str object X is zero bytes long.

Create A New Dynamic String Object

```
sqlite3_str *sqlite3_str_new(sqlite3*);
```

The sqlite3_str_new(D) interface allocates and initializes a new sqlite3_str object. To avoid memory leaks, the object returned by sqlite3_str_new() must be freed by a subsequent call to sqlite3_str_finish(X).

The sqlite3_str_new(D) interface always returns a pointer to a valid sqlite3_str object, though in the event of an out-of-memory error the returned object might be a special singleton that will silently reject new text, always return SQLITE_NOMEM from sqlite3_str_errcode(), always return 0 for sqlite3_str_length(), and always return NULL from sqlite3_str_finish(X). It is always safe to use the value returned by sqlite3_str_new(D) as the sqlite3_str parameter to any of the other sqlite3_str methods.

The D parameter to sqlite3_str_new(D) may be NULL. If the D parameter in sqlite3_str_new(D) is not NULL, then the maximum length of the string contained in the sqlite3_str object will be the value set for sqlite3_limit(D,SQLITE_LIMIT_LENGTH) instead of SQLITE_MAX_LENGTH.

String Globbing

```
int sqlite3_strglob(const char *zGlob, const char *zStr);
```

The sqlite3_strglob(P,X) interface returns zero if and only if string X matches the GLOB pattern P. The definition of GLOB pattern matching used in sqlite3_strglob(P,X) is the same as for the "X GLOB P" operator in the SQL dialect understood by SQLite. The sqlite3_strglob(P,X) function is case sensitive.

Note that this routine returns zero on a match and non-zero if the strings do not match, the same as sqlite3_stricmp() and sqlite3_strnicmp().

See also: sqlite3_strlike().

String LIKE Matching

```
int sqlite3_strlike(const char *zGlob, const char *zStr, unsigned int cEsc);
```

The sqlite3_strlike(P,X,E) interface returns zero if and only if string X matches the LIKE pattern P with escape character E. The definition of LIKE pattern matching used in sqlite3_strlike(P,X,E) is the same as for the "X LIKE P ESCAPE E" operator in the SQL dialect understood by SQLite. For "X LIKE P" without the ESCAPE clause, set the E parameter of sqlite3_strlike(P,X,E) to 0. As with the LIKE operator, the sqlite3_strlike(P,X,E) function is case insensitive - equivalent upper and lower case ASCII characters match one another.

The sqlite3_strlike(P,X,E) function matches Unicode characters, though only ASCII characters are case folded.

Note that this routine returns zero on a match and non-zero if the strings do not match, the same as sqlite3_stricmp() and sqlite3_strnicmp().

See also: sqlite3_strglob().

Low-level system error code

```
int sqlite3_system_errno(sqlite3*);
```

Attempt to return the underlying operating system error code or error number that caused the most recent I/O error or failure to open a file. The return value is OS-dependent. For example, on unix systems, after sqlite3_open_v2() returns SQLITE_CANTOPEN, this interface could be called to get back the underlying "errno" that caused the problem, such as ENOSPC, EAUTH, EISDIR, and so forth.

Extract Metadata About A Column Of A Table

```
int sqlite3_table_column_metadata(
  sqlite3 *db,                /* Connection handle */
  const char *zDbName,        /* Database name or NULL */
  const char *zTableName,     /* Table name */
  const char *zColumnName,    /* Column name */
  char const **pzDataType,    /* OUTPUT: Declared data type */
  char const **pzCollSeq,     /* OUTPUT: Collation sequence name */
  int *pNotNull,              /* OUTPUT: True if NOT NULL constraint exists */
  int *pPrimaryKey,           /* OUTPUT: True if column part of PK */
  int *pAutoinc               /* OUTPUT: True if column is auto-increment */
);
```

The sqlite3_table_column_metadata(X,D,T,C,....) routine returns information about column C of table T in database D on database connection X. The sqlite3_table_column_metadata() interface returns SQLITE_OK and fills in the non-NULL pointers in the final five arguments with appropriate values if the specified column exists. The sqlite3_table_column_metadata() interface returns SQLITE_ERROR if the specified column does not exist. If the column-name parameter to sqlite3_table_column_metadata() is a NULL pointer, then this routine simply checks for the existence of the table and returns SQLITE_OK if the table exists and SQLITE_ERROR if it does not. If the table name parameter T in a call to sqlite3_table_column_metadata(X,D,T,C,...) is NULL then the result is undefined behavior.

The column is identified by the second, third and fourth parameters to this function. The second parameter is either the name of the database (i.e. "main", "temp", or an attached database) containing the specified table or NULL. If it is NULL, then all attached databases are searched for the table using the same algorithm used by the database engine to resolve unqualified table references.

The third and fourth parameters to this function are the table and column name of the desired column, respectively.

Metadata is returned by writing to the memory locations passed as the 5th and subsequent parameters to this function. Any of these arguments may be NULL, in which case the corresponding element of metadata is omitted.

Parameter	Output Type	Description
5th	const char*	Data type
6th	const char*	Name of default collation sequence
7th	int	True if column has a NOT NULL constraint
8th	int	True if column is part of the PRIMARY KEY
9th	int	True if column is AUTOINCREMENT

The memory pointed to by the character pointers returned for the declaration type and collation sequence is valid until the next call to any SQLite API function.

If the specified table is actually a view, an error code is returned.

If the specified column is "rowid", "oid" or "_rowid_" and the table is not a WITHOUT ROWID table and an INTEGER PRIMARY KEY column has been explicitly declared, then the output parameters are set for the explicitly declared column. If there is no INTEGER PRIMARY KEY column, then the outputs for the rowid are set as follows:

```
data type: "INTEGER"
collation sequence: "BINARY"
not null: 0
primary key: 1
auto increment: 0
```

This function causes all database schemas to be read from disk and parsed, if that has not already been done, and returns an error if any errors are encountered while loading the schema.

Testing Interface

```
int sqlite3_test_control(int op, ...);
```

The sqlite3_test_control() interface is used to read out internal state of SQLite and to inject faults into SQLite for testing purposes. The first parameter is an operation code that determines the number, meaning, and operation of all subsequent parameters.

This interface is not for use by applications. It exists solely for verifying the correct operation of the SQLite library. Depending on how the SQLite library is compiled, this interface might not exist.

The details of the operation codes, their meanings, the parameters they take, and what they do are all subject to change without notice. Unlike most of the SQLite API, this function is not guaranteed to operate consistently from one release to the next.

Test To See If The Library Is Threadsafe

```
int sqlite3_threadsafe(void);
```

The sqlite3_threadsafe() function returns zero if and only if SQLite was compiled with mutexing code omitted due to the SQLITE_THREADSAFE compile-time option being set to 0.

SQLite can be compiled with or without mutexes. When the SQLITE_THREADSAFE C preprocessor macro is 1 or 2, mutexes are enabled and SQLite is threadsafe. When the SQLITE_THREADSAFE macro is 0, the mutexes are omitted. Without the mutexes, it is not safe to use SQLite concurrently from more than one thread.

Enabling mutexes incurs a measurable performance penalty. So if speed is of utmost importance, it makes sense to disable the mutexes. But for maximum safety, mutexes should be enabled. The default behavior is for mutexes to be enabled.

This interface can be used by an application to make sure that the version of SQLite that it is linking against was compiled with the desired setting of the SQLITE_THREADSAFE macro.

This interface only reports on the compile-time mutex setting of the SQLITE_THREADSAFE flag. If SQLite is compiled with SQLITE_THREADSAFE=1 or =2 then mutexes are enabled by default but can be fully or partially disabled using a call to sqlite3_config() with the verbs SQLITE_CONFIG_SINGLETHREAD, SQLITE_CONFIG_MULTITHREAD, or SQLITE_CONFIG_SERIALIZED. The return value of the sqlite3_threadsafe() function shows only the compile-time setting of thread safety, not any run-time changes to that setting made by sqlite3_config(). In other words, the return value from sqlite3_threadsafe() is unchanged by calls to sqlite3_config().

See the threading mode documentation for additional information.

Total Number Of Rows Modified

```
int sqlite3_total_changes(sqlite3*);
```

This function returns the total number of rows inserted, modified or deleted by all INSERT, UPDATE or DELETE statements completed since the database connection was opened, including those executed as part of trigger programs. Executing any other type of SQL statement does not affect the value returned by sqlite3_total_changes().

Changes made as part of foreign key actions are included in the count, but those made as part of REPLACE constraint resolution are not. Changes to a view that are intercepted by INSTEAD OF triggers are not counted.

The sqlite3_total_changes(D) interface only reports the number of rows that changed due to SQL statement run against database connection D. Any changes by other database connections are ignored. To detect changes against a database file from other database connections use the PRAGMA data_version command or the SQLITE_FCNTL_DATA_VERSION file control.

If a separate thread makes changes on the same database connection while sqlite3_total_changes() is running then the value returned is unpredictable and not meaningful.

See also:

- the sqlite3_changes() interface
- the count_changes pragma
- the changes() SQL function
- the data_version pragma
- the SQLITE_FCNTL_DATA_VERSION file control

SQL Trace Hook

```
int sqlite3_trace_v2(
  sqlite3*,
  unsigned uMask,
  int(*xCallback)(unsigned,void*,void*,void*),
  void *pCtx
);
```

The sqlite3_trace_v2(D,M,X,P) interface registers a trace callback function X against database connection D, using property mask M and context pointer P. If the X callback is NULL or if the M mask is zero, then tracing is disabled. The M argument should be the bitwise OR-ed combination of zero or more SQLITE_TRACE constants.

Each call to either sqlite3_trace() or sqlite3_trace_v2() overrides (cancels) any prior calls to sqlite3_trace() or sqlite3_trace_v2().

The X callback is invoked whenever any of the events identified by mask M occur. The integer return value from the callback is currently ignored, though this may change in future releases. Callback implementations should return zero to ensure future compatibility.

A trace callback is invoked with four arguments: callback(T,C,P,X). The T argument is one of the SQLITE_TRACE constants to indicate why the callback was invoked. The C argument is a copy of the context pointer. The P and X arguments are pointers whose meanings depend on T.

The sqlite3_trace_v2() interface is intended to replace the legacy interfaces sqlite3_trace() and sqlite3_profile(), both of which are deprecated.

Unlock Notification

```
int sqlite3_unlock_notify(
  sqlite3 *pBlocked,                          /* Waiting connection */
  void (*xNotify)(void **apArg, int nArg),    /* Callback function to invoke */
  void *pNotifyArg                            /* Argument to pass to xNotify */
);
```

When running in shared-cache mode, a database operation may fail with an SQLITE_LOCKED error if the required locks on the shared-cache or individual tables within the shared-cache cannot be obtained. See SQLite Shared-Cache Mode for a description of shared-cache locking. This API may be used to register a callback that SQLite will invoke when the connection currently holding the required lock relinquishes it. This API is only available if the library was compiled with the SQLITE_ENABLE_UNLOCK_NOTIFY C-preprocessor symbol defined.

See Also: Using the SQLite Unlock Notification Feature.

Shared-cache locks are released when a database connection concludes its current transaction, either by committing it or rolling it back.

When a connection (known as the blocked connection) fails to obtain a shared-cache lock and SQLITE_LOCKED is returned to the caller, the identity of the database connection (the blocking connection) that has locked the required resource is stored internally. After an application receives an SQLITE_LOCKED error, it may call the sqlite3_unlock_notify() method with the blocked connection handle as the first argument to register for a callback that will be invoked when the blocking connections current transaction is concluded. The callback is invoked from within the sqlite3_step or sqlite3_close call that concludes the blocking connection's transaction.

If sqlite3_unlock_notify() is called in a multi-threaded application, there is a chance that the blocking connection will have already concluded its transaction by the time sqlite3_unlock_notify() is invoked. If this happens, then the specified callback is invoked immediately, from within the call to sqlite3_unlock_notify().

If the blocked connection is attempting to obtain a write-lock on a shared-cache table, and more than one other connection currently holds a read-lock on the same table, then SQLite arbitrarily selects one of the other connections to use as the blocking connection.

There may be at most one unlock-notify callback registered by a blocked connection. If sqlite3_unlock_notify() is called when the blocked connection already has a registered unlock-notify callback, then the new callback replaces the old. If sqlite3_unlock_notify() is called with a NULL pointer as its second argument, then any existing unlock-notify callback is canceled. The blocked connections unlock-notify callback may also be canceled by closing the blocked connection using sqlite3_close().

The unlock-notify callback is not reentrant. If an application invokes any sqlite3_xxx API functions from within an unlock-notify callback, a crash or deadlock may be the result.

Unless deadlock is detected (see below), sqlite3_unlock_notify() always returns SQLITE_OK.

Callback Invocation Details

When an unlock-notify callback is registered, the application provides a single void* pointer that is passed to the callback when it is invoked. However, the signature of the callback function allows SQLite to pass it an array of void* context pointers. The first argument passed to an unlock-notify callback is a pointer to an array of void* pointers, and the second is the number of entries in the array.

When a blocking connection's transaction is concluded, there may be more than one blocked connection that has registered for an unlock-notify callback. If two or more such blocked connections have specified the same callback function, then instead of invoking the callback function multiple times, it is invoked once with the set of void* context pointers specified by the blocked connections bundled together into an array. This gives the application an opportunity to prioritize any actions related to the set of unblocked database connections.

Deadlock Detection

Assuming that after registering for an unlock-notify callback a database waits for the callback to be issued before taking any further action (a reasonable assumption), then using this API may cause the application to deadlock. For example, if connection X is waiting for connection Y's transaction to be concluded, and similarly connection Y is waiting on connection X's transaction, then neither connection will proceed and the system may remain deadlocked indefinitely.

To avoid this scenario, the sqlite3_unlock_notify() performs deadlock detection. If a given call to sqlite3_unlock_notify() would put the system in a deadlocked state, then SQLITE_LOCKED is returned and no unlock-notify callback is registered. The system is said to be in a deadlocked state if connection A has registered for an unlock-notify callback on the conclusion of connection B's transaction, and connection B has itself registered for an unlock-notify callback when connection A's transaction is concluded. Indirect deadlock is also detected, so the system is also considered to be deadlocked if connection B has registered for an unlock-notify callback on the conclusion of connection C's transaction, where connection C is waiting on connection A. Any number of levels of indirection are allowed.

The "DROP TABLE" Exception

When a call to sqlite3_step() returns SQLITE_LOCKED, it is almost always appropriate to call sqlite3_unlock_notify(). There is however, one exception. When executing a "DROP TABLE" or "DROP INDEX" statement, SQLite checks if there are any currently executing SELECT statements that belong to the same connection. If there are, SQLITE_LOCKED is returned. In this case there is no "blocking connection", so invoking sqlite3_unlock_notify() results in the unlock-notify callback being invoked immediately. If the application then re-attempts the "DROP TABLE" or "DROP INDEX" query, an infinite loop might be the result.

One way around this problem is to check the extended error code returned by an sqlite3_step() call. If there is a blocking connection, then the extended error code is set to SQLITE_LOCKED_SHAREDCACHE. Otherwise, in the special "DROP TABLE/INDEX" case, the extended error code is just SQLITE_LOCKED.

Data Change Notification Callbacks

```
void *sqlite3_update_hook(
  sqlite3*,
  void(*)(void *,int ,char const *,char const *,sqlite3_int64),
  void*
);
```

The sqlite3_update_hook() interface registers a callback function with the database connection identified by the first argument to be invoked whenever a row is updated, inserted or deleted in a rowid table. Any callback set by a previous call to this function for the same database connection is overridden.

The second argument is a pointer to the function to invoke when a row is updated, inserted or deleted in a rowid table. The first argument to the callback is a copy of the third argument to sqlite3_update_hook(). The second callback argument is one of SQLITE_INSERT, SQLITE_DELETE, or SQLITE_UPDATE, depending on the operation that caused the callback to be invoked. The third and fourth arguments to the callback contain pointers to the database and table name containing the affected row. The final callback parameter is the rowid of the row. In the case of an update, this is the rowid after the update takes place.

The update hook is not invoked when internal system tables are modified (i.e. sqlite_master and sqlite_sequence). The update hook is not invoked when WITHOUT ROWID tables are modified.

In the current implementation, the update hook is not invoked when conflicting rows are deleted because of an ON CONFLICT REPLACE clause. Nor is the update hook invoked when rows are deleted using the truncate optimization. The exceptions defined in this paragraph might change in a future release of SQLite.

The update hook implementation must not do anything that will modify the database connection that invoked the update hook. Any actions to modify the database connection must be deferred until after the completion of the sqlite3_step() call that triggered the update hook. Note that sqlite3_prepare_v2() and sqlite3_step() both modify their database connections for the meaning of "modify" in this paragraph.

The sqlite3_update_hook(D,C,P) function returns the P argument from the previous call on the same database connection D, or NULL for the first call on D.

See also the sqlite3_commit_hook(), sqlite3_rollback_hook(), and sqlite3_preupdate_hook() interfaces.

User Data For Functions

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

The sqlite3_user_data() interface returns a copy of the pointer that was the pUserData parameter (the 5th parameter) of the sqlite3_create_function() and sqlite3_create_function16() routines that originally registered the application defined function.

This routine must be called from the same thread in which the application-defined function is running.

Finding The Subtype Of SQL Values

```
unsigned int sqlite3_value_subtype(sqlite3_value*);
```

The sqlite3_value_subtype(V) function returns the subtype for an application-defined SQL function argument V. The subtype information can be used to pass a limited amount of context from one SQL function to another. Use the sqlite3_result_subtype() routine to set the subtype for the return value of an SQL function.

Determine The Collation For a Virtual Table Constraint

```
const char *sqlite3_vtab_collation(sqlite3_index_info*,int);
```

This function may only be called from within a call to the xBestIndex method of a virtual table.

The first argument must be the sqlite3_index_info object that is the first parameter to the xBestIndex() method. The second argument must be an index into the aConstraint[] array belonging to the sqlite3_index_info structure passed to xBestIndex. This function returns a pointer to a buffer containing the name of the collation sequence for the corresponding constraint.

Virtual Table Interface Configuration

```
int sqlite3_vtab_config(sqlite3*, int op, ...);
```

This function may be called by either the xConnect or xCreate method of a virtual table implementation to configure various facets of the virtual table interface.

If this interface is invoked outside the context of an xConnect or xCreate virtual table method then the behavior is undefined.

In the call sqlite3_vtab_config(D,C,...) the D parameter is the database connection in which the virtual table is being created and which is passed in as the first argument to the xConnect or xCreate method that is invoking sqlite3_vtab_config(). The C parameter is one of the virtual table configuration options. The presence and meaning of parameters after C depend on which virtual table configuration option is used.

Determine If Virtual Table Column Access Is For UPDATE

```
int sqlite3_vtab_nochange(sqlite3_context*);
```

If the sqlite3_vtab_nochange(X) routine is called within the xColumn method of a virtual table, then it returns true if and only if the column is being fetched as part of an UPDATE operation during which the column value will not change. Applications might use this to substitute a return value that is less expensive to compute and that the corresponding xUpdate method understands as a "no-change" value.

If the xColumn method calls sqlite3_vtab_nochange() and finds that the column is not changed by the UPDATE statement, then the xColumn method can optionally return without setting a result, without calling any of the sqlite3_result_xxxxx() interfaces. In that case, sqlite3_value_nochange(X) will return true for the same column in the xUpdate method.

Determine The Virtual Table Conflict Policy

```
int sqlite3_vtab_on_conflict(sqlite3 *);
```

This function may only be called from within a call to the xUpdate method of a virtual table implementation for an INSERT or UPDATE operation. The value returned is one of SQLITE_ROLLBACK, SQLITE_IGNORE, SQLITE_FAIL, SQLITE_ABORT, or SQLITE_REPLACE, according to the ON CONFLICT mode of the SQL statement that triggered the call to the xUpdate method of the virtual table.

Configure an auto-checkpoint

```
int sqlite3_wal_autocheckpoint(sqlite3 *db, int N);
```

The sqlite3_wal_autocheckpoint(D,N) is a wrapper around sqlite3_wal_hook() that causes any database on database connection D to automatically checkpoint after committing a transaction if there are N or more frames in the write-ahead log file. Passing zero or a negative value as the nFrame parameter disables automatic checkpoints entirely.

The callback registered by this function replaces any existing callback registered using sqlite3_wal_hook(). Likewise, registering a callback using sqlite3_wal_hook() disables the automatic checkpoint mechanism configured by this function.

The wal_autocheckpoint pragma can be used to invoke this interface from SQL.

Checkpoints initiated by this mechanism are PASSIVE.

Every new database connection defaults to having the auto-checkpoint enabled with a threshold of 1000 or SQLITE_DEFAULT_WAL_AUTOCHECKPOINT pages. The use of this interface is only necessary if the default setting is found to be suboptimal for a particular application.

Checkpoint a database

```
int sqlite3_wal_checkpoint(sqlite3 *db, const char *zDb);
```

The sqlite3_wal_checkpoint(D,X) is equivalent to sqlite3_wal_checkpoint_v2(D,X,SQLITE_CHECKPOINT_PASSIVE,0,0).

In brief, sqlite3_wal_checkpoint(D,X) causes the content in the write-ahead log for database X on database connection D to be transferred into the database file and for the write-ahead log to be reset. See the checkpointing documentation for addition information.

This interface used to be the only way to cause a checkpoint to occur. But then the newer and more powerful sqlite3_wal_checkpoint_v2() interface was added. This interface is retained for backwards compatibility and as a convenience for applications that need to manually start a callback but which do not need the full power (and corresponding complication) of sqlite3_wal_checkpoint_v2().

Checkpoint a database

```
int sqlite3_wal_checkpoint_v2(
  sqlite3 *db,                    /* Database handle */
  const char *zDb,                /* Name of attached database (or NULL) */
  int eMode,                      /* SQLITE_CHECKPOINT_* value */
  int *pnLog,                     /* OUT: Size of WAL log in frames */
  int *pnCkpt                     /* OUT: Total number of frames checkpointed */
);
```

The sqlite3_wal_checkpoint_v2(D,X,M,L,C) interface runs a checkpoint operation on database X of database connection D in mode M. Status information is written back into integers pointed to by L and C. The M parameter must be a valid checkpoint mode:

SQLITE_CHECKPOINT_PASSIVE
: Checkpoint as many frames as possible without waiting for any database readers or writers to finish, then sync the database file if all frames in the log were checkpointed. The busy-handler callback is never invoked in the SQLITE_CHECKPOINT_PASSIVE mode. On the other hand, passive mode might leave the checkpoint unfinished if there are concurrent readers or writers.

SQLITE_CHECKPOINT_FULL
: This mode blocks (it invokes the busy-handler callback) until there is no database writer and all readers are reading from the most recent database snapshot. It then checkpoints all frames in the log file and syncs the database file. This mode blocks new database writers while it is pending, but new database readers are allowed to continue unimpeded.

SQLITE_CHECKPOINT_RESTART
: This mode works the same way as SQLITE_CHECKPOINT_FULL with the addition that after checkpointing the log file it blocks (calls the busy-handler callback) until all readers are reading from the database file only. This ensures that the next writer will restart the log file from the beginning. Like SQLITE_CHECKPOINT_FULL, this mode blocks new database writer attempts while it is pending, but does not impede readers.

SQLITE_CHECKPOINT_TRUNCATE
: This mode works the same way as SQLITE_CHECKPOINT_RESTART with the addition that it also truncates the log file to zero bytes just prior to a successful return.

If pnLog is not NULL, then *pnLog is set to the total number of frames in the log file or to -1 if the checkpoint could not run because of an error or because the database is not in WAL mode. If pnCkpt is not NULL,then *pnCkpt is set to the total number of checkpointed frames in the log file (including any that were already checkpointed before the function was called) or to -1 if the checkpoint could not run due to an error or because the database is not in WAL mode. Note that upon successful completion of an SQLITE_CHECKPOINT_TRUNCATE, the log file will have been truncated to zero bytes and so both *pnLog and *pnCkpt will be set to zero.

All calls obtain an exclusive "checkpoint" lock on the database file. If any other process is running a checkpoint operation at the same time, the lock cannot be obtained and SQLITE_BUSY is returned. Even if there is a busy-handler configured, it will not be invoked in this case.

The SQLITE_CHECKPOINT_FULL, RESTART and TRUNCATE modes also obtain the exclusive "writer" lock on the database file. If the writer lock cannot be obtained immediately, and a busy-handler is configured, it is invoked and the writer lock retried until either the busy-handler returns 0 or the lock is successfully obtained. The busy-handler is also invoked while waiting for database readers as described above. If the busy-handler returns 0 before the writer lock is obtained or while waiting for database readers, the checkpoint operation proceeds from that point in the same way as SQLITE_CHECKPOINT_PASSIVE - checkpointing as many frames as possible without blocking any further. SQLITE_BUSY is returned in this case.

If parameter zDb is NULL or points to a zero length string, then the specified operation is attempted on all WAL databases attached to database connection db. In this case the values written to output parameters *pnLog and *pnCkpt are undefined. If an SQLITE_BUSY error is encountered when processing one or more of the attached WAL databases, the operation is still attempted on any remaining attached databases and SQLITE_BUSY is returned at the end. If any other error occurs while processing an attached database, processing is abandoned and the error code is returned to the caller immediately. If no error (SQLITE_BUSY or otherwise) is encountered while processing the attached databases, SQLITE_OK is returned.

If database zDb is the name of an attached database that is not in WAL mode, SQLITE_OK is returned and both *pnLog and *pnCkpt set to -1. If zDb is not NULL (or a zero length string) and is not the name of any attached database, SQLITE_ERROR is returned to the caller.

Unless it returns SQLITE_MISUSE, the sqlite3_wal_checkpoint_v2() interface sets the error information that is queried by sqlite3_errcode() and sqlite3_errmsg().

The PRAGMA wal_checkpoint command can be used to invoke this interface from SQL.

Write-Ahead Log Commit Hook

```
void *sqlite3_wal_hook(
  sqlite3*,
  int(*)(void *,sqlite3*,const char*,int),
  void*
);
```

The sqlite3_wal_hook() function is used to register a callback that is invoked each time data is committed to a database in wal mode.

The callback is invoked by SQLite after the commit has taken place and the associated write-lock on the database released, so the implementation may read, write or checkpoint the database as required.

The first parameter passed to the callback function when it is invoked is a copy of the third parameter passed to sqlite3_wal_hook() when registering the callback. The second is a copy of the database handle. The third parameter is the name of

the database that was written to - either "main" or the name of an ATTACH-ed database. The fourth parameter is the number of pages currently in the write-ahead log file, including those that were just committed.

The callback function should normally return SQLITE_OK. If an error code is returned, that error will propagate back up through the SQLite code base to cause the statement that provoked the callback to report an error, though the commit will have still occurred. If the callback returns SQLITE_ROW or SQLITE_DONE, or if it returns a value that does not correspond to any valid SQLite error code, the results are undefined.

A single database handle may have at most a single write-ahead log callback registered at one time. Calling sqlite3_wal_hook() replaces any previously registered write-ahead log callback. Note that the sqlite3_wal_autocheckpoint() interface and the wal_autocheckpoint pragma both invoke sqlite3_wal_hook() and will overwrite any prior sqlite3_wal_hook() settings.

Database Snapshot

```
typedef struct sqlite3_snapshot {
  unsigned char hidden[48];
} sqlite3_snapshot;
```

An instance of the snapshot object records the state of a WAL mode database for some specific point in history.

In WAL mode, multiple database connections that are open on the same database file can each be reading a different historical version of the database file. When a database connection begins a read transaction, that connection sees an unchanging copy of the database as it existed for the point in time when the transaction first started. Subsequent changes to the database from other connections are not seen by the reader until a new read transaction is started.

The sqlite3_snapshot object records state information about an historical version of the database file so that it is possible to later open a new read transaction that sees that historical version of the database rather than the most recent version.

Constructor: sqlite3_snapshot_get()

Destructor: sqlite3_snapshot_free()

Methods: sqlite3_snapshot_cmp(), sqlite3_snapshot_open(), sqlite3_snapshot_recover()

Result Codes

```
#define SQLITE_OK           0   /* Successful result */
/* beginning-of-error-codes */
#define SQLITE_ERROR        1   /* Generic error */
#define SQLITE_INTERNAL     2   /* Internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
#define SQLITE_INTERRUPT    9   /* Operation terminated by sqlite3_interrupt()*/
#define SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT     11   /* The database disk image is malformed */
#define SQLITE_NOTFOUND    12   /* Unknown opcode in sqlite3_file_control() */
#define SQLITE_FULL        13   /* Insertion failed because database is full */
#define SQLITE_CANTOPEN    14   /* Unable to open the database file */
#define SQLITE_PROTOCOL    15   /* Database lock protocol error */
#define SQLITE_EMPTY       16   /* Internal use only */
#define SQLITE_SCHEMA      17   /* The database schema changed */
#define SQLITE_TOOBIG      18   /* String or BLOB exceeds size limit */
#define SQLITE_CONSTRAINT  19   /* Abort due to constraint violation */
#define SQLITE_MISMATCH    20   /* Data type mismatch */
#define SQLITE_MISUSE      21   /* Library used incorrectly */
#define SQLITE_NOLFS       22   /* Uses OS features not supported on host */
#define SQLITE_AUTH        23   /* Authorization denied */
#define SQLITE_FORMAT      24   /* Not used */
#define SQLITE_RANGE       25   /* 2nd parameter to sqlite3_bind out of range */
#define SQLITE_NOTADB      26   /* File opened that is not a database file */
#define SQLITE_NOTICE      27   /* Notifications from sqlite3_log() */
#define SQLITE_WARNING     28   /* Warnings from sqlite3_log() */
#define SQLITE_ROW         100  /* sqlite3_step() has another row ready */
#define SQLITE_DONE        101  /* sqlite3_step() has finished executing */
/* end-of-error-codes */
```

Many SQLite functions return an integer result code from the set shown here in order to indicate success or failure.

New error codes may be added in future versions of SQLite.

See also: extended result code definitions

Extended Result Codes

```
#define SQLITE_ERROR_MISSING_COLLSEQ   (SQLITE_ERROR | (1<<8))
#define SQLITE_ERROR_RETRY             (SQLITE_ERROR | (2<<8))
#define SQLITE_ERROR_SNAPSHOT          (SQLITE_ERROR | (3<<8))
#define SQLITE_IOERR_READ              (SQLITE_IOERR | (1<<8))
#define SQLITE_IOERR_SHORT_READ        (SQLITE_IOERR | (2<<8))
#define SQLITE_IOERR_WRITE             (SQLITE_IOERR | (3<<8))
#define SQLITE_IOERR_FSYNC             (SQLITE_IOERR | (4<<8))
#define SQLITE_IOERR_DIR_FSYNC         (SQLITE_IOERR | (5<<8))
#define SQLITE_IOERR_TRUNCATE          (SQLITE_IOERR | (6<<8))
#define SQLITE_IOERR_FSTAT             (SQLITE_IOERR | (7<<8))
#define SQLITE_IOERR_UNLOCK            (SQLITE_IOERR | (8<<8))
```

```
#define SQLITE_IOERR_RDLOCK            (SQLITE_IOERR | (9<<8))
#define SQLITE_IOERR_DELETE            (SQLITE_IOERR | (10<<8))
#define SQLITE_IOERR_BLOCKED           (SQLITE_IOERR | (11<<8))
#define SQLITE_IOERR_NOMEM             (SQLITE_IOERR | (12<<8))
#define SQLITE_IOERR_ACCESS            (SQLITE_IOERR | (13<<8))
#define SQLITE_IOERR_CHECKRESERVEDLOCK (SQLITE_IOERR | (14<<8))
#define SQLITE_IOERR_LOCK              (SQLITE_IOERR | (15<<8))
#define SQLITE_IOERR_CLOSE             (SQLITE_IOERR | (16<<8))
#define SQLITE_IOERR_DIR_CLOSE         (SQLITE_IOERR | (17<<8))
#define SQLITE_IOERR_SHMOPEN           (SQLITE_IOERR | (18<<8))
#define SQLITE_IOERR_SHMSIZE           (SQLITE_IOERR | (19<<8))
#define SQLITE_IOERR_SHMLOCK           (SQLITE_IOERR | (20<<8))
#define SQLITE_IOERR_SHMMAP            (SQLITE_IOERR | (21<<8))
#define SQLITE_IOERR_SEEK              (SQLITE_IOERR | (22<<8))
#define SQLITE_IOERR_DELETE_NOENT      (SQLITE_IOERR | (23<<8))
#define SQLITE_IOERR_MMAP              (SQLITE_IOERR | (24<<8))
#define SQLITE_IOERR_GETTEMPPATH       (SQLITE_IOERR | (25<<8))
#define SQLITE_IOERR_CONVPATH          (SQLITE_IOERR | (26<<8))
#define SQLITE_IOERR_VNODE             (SQLITE_IOERR | (27<<8))
#define SQLITE_IOERR_AUTH              (SQLITE_IOERR | (28<<8))
#define SQLITE_IOERR_BEGIN_ATOMIC      (SQLITE_IOERR | (29<<8))
#define SQLITE_IOERR_COMMIT_ATOMIC     (SQLITE_IOERR | (30<<8))
#define SQLITE_IOERR_ROLLBACK_ATOMIC   (SQLITE_IOERR | (31<<8))
#define SQLITE_LOCKED_SHAREDCACHE      (SQLITE_LOCKED |  (1<<8))
#define SQLITE_LOCKED_VTAB             (SQLITE_LOCKED |  (2<<8))
#define SQLITE_BUSY_RECOVERY           (SQLITE_BUSY   |  (1<<8))
#define SQLITE_BUSY_SNAPSHOT           (SQLITE_BUSY   |  (2<<8))
#define SQLITE_CANTOPEN_NOTEMPDIR      (SQLITE_CANTOPEN | (1<<8))
#define SQLITE_CANTOPEN_ISDIR          (SQLITE_CANTOPEN | (2<<8))
#define SQLITE_CANTOPEN_FULLPATH       (SQLITE_CANTOPEN | (3<<8))
#define SQLITE_CANTOPEN_CONVPATH       (SQLITE_CANTOPEN | (4<<8))
#define SQLITE_CANTOPEN_DIRTYWAL       (SQLITE_CANTOPEN | (5<<8)) /* Not Used */
#define SQLITE_CANTOPEN_SYMLINK        (SQLITE_CANTOPEN | (6<<8))
#define SQLITE_CORRUPT_VTAB            (SQLITE_CORRUPT | (1<<8))
#define SQLITE_CORRUPT_SEQUENCE        (SQLITE_CORRUPT | (2<<8))
#define SQLITE_READONLY_RECOVERY       (SQLITE_READONLY | (1<<8))
#define SQLITE_READONLY_CANTLOCK       (SQLITE_READONLY | (2<<8))
#define SQLITE_READONLY_ROLLBACK       (SQLITE_READONLY | (3<<8))
#define SQLITE_READONLY_DBMOVED        (SQLITE_READONLY | (4<<8))
#define SQLITE_READONLY_CANTINIT       (SQLITE_READONLY | (5<<8))
#define SQLITE_READONLY_DIRECTORY      (SQLITE_READONLY | (6<<8))
#define SQLITE_ABORT_ROLLBACK          (SQLITE_ABORT | (2<<8))
#define SQLITE_CONSTRAINT_CHECK        (SQLITE_CONSTRAINT | (1<<8))
#define SQLITE_CONSTRAINT_COMMITHOOK   (SQLITE_CONSTRAINT | (2<<8))
#define SQLITE_CONSTRAINT_FOREIGNKEY   (SQLITE_CONSTRAINT | (3<<8))
#define SQLITE_CONSTRAINT_FUNCTION     (SQLITE_CONSTRAINT | (4<<8))
#define SQLITE_CONSTRAINT_NOTNULL      (SQLITE_CONSTRAINT | (5<<8))
#define SQLITE_CONSTRAINT_PRIMARYKEY   (SQLITE_CONSTRAINT | (6<<8))
#define SQLITE_CONSTRAINT_TRIGGER      (SQLITE_CONSTRAINT | (7<<8))
#define SQLITE_CONSTRAINT_UNIQUE       (SQLITE_CONSTRAINT | (8<<8))
#define SQLITE_CONSTRAINT_VTAB         (SQLITE_CONSTRAINT | (9<<8))
#define SQLITE_CONSTRAINT_ROWID        (SQLITE_CONSTRAINT |(10<<8))
#define SQLITE_CONSTRAINT_PINNED       (SQLITE_CONSTRAINT |(11<<8))
#define SQLITE_NOTICE_RECOVER_WAL      (SQLITE_NOTICE | (1<<8))
#define SQLITE_NOTICE_RECOVER_ROLLBACK (SQLITE_NOTICE | (2<<8))
#define SQLITE_WARNING_AUTOINDEX       (SQLITE_WARNING | (1<<8))
#define SQLITE_AUTH_USER               (SQLITE_AUTH | (1<<8))
#define SQLITE_OK_LOAD_PERMANENTLY     (SQLITE_OK | (1<<8))
#define SQLITE_OK_SYMLINK              (SQLITE_OK | (2<<8))
```

In its default configuration, SQLite API routines return one of 30 integer result codes. However, experience has shown that many of these result codes are too coarse-grained. They do not provide as much information about problems as programmers might like. In an effort to address this, newer versions of SQLite (version 3.3.8 2006-10-09 and later) include support for additional result codes that provide more detailed information about errors. These extended result codes are enabled or disabled on a per database connection basis using the sqlite3_extended_result_codes() API. Or, the extended code for the most recent error can be obtained using sqlite3_extended_errcode().

Flags for the xAccess VFS method

```
#define SQLITE_ACCESS_EXISTS    0
#define SQLITE_ACCESS_READWRITE 1   /* Used by PRAGMA temp_store_directory */
#define SQLITE_ACCESS_READ      2   /* Unused */
```

These integer constants can be used as the third parameter to the xAccess method of an sqlite3_vfs object. They determine what kind of permissions the xAccess method is looking for. With SQLITE_ACCESS_EXISTS, the xAccess method simply checks whether the file exists. With SQLITE_ACCESS_READWRITE, the xAccess method checks whether the named directory is both readable and writable (in other words, if files can be added, removed, and renamed within the directory). The SQLITE_ACCESS_READWRITE constant is currently used only by the temp_store_directory pragma, though this could change in a future release of SQLite. With SQLITE_ACCESS_READ, the xAccess method checks whether the file is readable. The SQLITE_ACCESS_READ constant is currently unused, though it might be used in a future release of SQLite.

Authorizer Action Codes

```
/****************************************** 3rd ************ 4th ***********/
#define SQLITE_CREATE_INDEX          1   /* Index Name      Table Name      */
#define SQLITE_CREATE_TABLE          2   /* Table Name      NULL            */
#define SQLITE_CREATE_TEMP_INDEX     3   /* Index Name      Table Name      */
#define SQLITE_CREATE_TEMP_TABLE     4   /* Table Name      NULL            */
#define SQLITE_CREATE_TEMP_TRIGGER   5   /* Trigger Name    Table Name      */
#define SQLITE_CREATE_TEMP_VIEW      6   /* View Name       NULL            */
#define SQLITE_CREATE_TRIGGER        7   /* Trigger Name    Table Name      */
#define SQLITE_CREATE_VIEW           8   /* View Name       NULL            */
#define SQLITE_DELETE                9   /* Table Name      NULL            */
#define SQLITE_DROP_INDEX           10   /* Index Name      Table Name      */
#define SQLITE_DROP_TABLE           11   /* Table Name      NULL            */
```

```
#define SQLITE_DROP_TEMP_INDEX      12   /* Index Name      Table Name      */
#define SQLITE_DROP_TEMP_TABLE      13   /* Table Name      NULL            */
#define SQLITE_DROP_TEMP_TRIGGER    14   /* Trigger Name    Table Name      */
#define SQLITE_DROP_TEMP_VIEW       15   /* View Name       NULL            */
#define SQLITE_DROP_TRIGGER         16   /* Trigger Name    Table Name      */
#define SQLITE_DROP_VIEW            17   /* View Name       NULL            */
#define SQLITE_INSERT               18   /* Table Name      NULL            */
#define SQLITE_PRAGMA               19   /* Pragma Name     1st arg or NULL */
#define SQLITE_READ                 20   /* Table Name      Column Name     */
#define SQLITE_SELECT               21   /* NULL            NULL            */
#define SQLITE_TRANSACTION          22   /* Operation       NULL            */
#define SQLITE_UPDATE               23   /* Table Name      Column Name     */
#define SQLITE_ATTACH               24   /* Filename        NULL            */
#define SQLITE_DETACH               25   /* Database Name   NULL            */
#define SQLITE_ALTER_TABLE          26   /* Database Name   Table Name      */
#define SQLITE_REINDEX              27   /* Index Name      NULL            */
#define SQLITE_ANALYZE              28   /* Table Name      NULL            */
#define SQLITE_CREATE_VTABLE        29   /* Table Name      Module Name     */
#define SQLITE_DROP_VTABLE          30   /* Table Name      Module Name     */
#define SQLITE_FUNCTION             31   /* NULL            Function Name   */
#define SQLITE_SAVEPOINT            32   /* Operation       Savepoint Name  */
#define SQLITE_COPY                  0   /* No longer used */
#define SQLITE_RECURSIVE            33   /* NULL            NULL            */
```

The sqlite3_set_authorizer() interface registers a callback function that is invoked to authorize certain SQL statement actions. The second parameter to the callback is an integer code that specifies what action is being authorized. These are the integer action codes that the authorizer callback may be passed.

These action code values signify what kind of operation is to be authorized. The 3rd and 4th parameters to the authorization callback function will be parameters or NULL depending on which of these codes is used as the second parameter. The 5th parameter to the authorizer callback is the name of the database ("main", "temp", etc.) if applicable. The 6th parameter to the authorizer callback is the name of the inner-most trigger or view that is responsible for the access attempt or NULL if this access attempt is directly from top-level SQL code.

Text Encodings

```
#define SQLITE_UTF8           1    /* IMP: R-37514-35566 */
#define SQLITE_UTF16LE        2    /* IMP: R-03371-37637 */
#define SQLITE_UTF16BE        3    /* IMP: R-51971-34154 */
#define SQLITE_UTF16          4    /* Use native byte order */
#define SQLITE_ANY            5    /* Deprecated */
#define SQLITE_UTF16_ALIGNED  8    /* sqlite3_create_collation only */
```

These constant define integer codes that represent the various text encodings supported by SQLite.

Fundamental Datatypes

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_BLOB     4
#define SQLITE_NULL     5
#ifdef SQLITE_TEXT
# undef SQLITE_TEXT
#else
# define SQLITE_TEXT     3
#endif
#define SQLITE3_TEXT     3
```

Every value in SQLite has one of five fundamental datatypes:

- 64-bit signed integer
- 64-bit IEEE floating point number
- string
- BLOB
- NULL

These constants are codes for each of those types.

Note that the SQLITE_TEXT constant was also used in SQLite version 2 for a completely different meaning. Software that links against both SQLite version 2 and SQLite version 3 should use SQLITE3_TEXT, not SQLITE_TEXT.

Checkpoint Mode Values

```
#define SQLITE_CHECKPOINT_PASSIVE  0  /* Do as much as possible w/o blocking */
#define SQLITE_CHECKPOINT_FULL     1  /* Wait for writers, then checkpoint */
#define SQLITE_CHECKPOINT_RESTART  2  /* Like FULL but wait for for readers */
#define SQLITE_CHECKPOINT_TRUNCATE 3  /* Like RESTART but also truncate WAL */
```

These constants define all valid values for the "checkpoint mode" passed as the third parameter to the sqlite3_wal_checkpoint_v2() interface. See the sqlite3_wal_checkpoint_v2() documentation for details on the meaning of each of these checkpoint modes.

Configuration Options

```
#define SQLITE_CONFIG_SINGLETHREAD  1  /* nil */
#define SQLITE_CONFIG_MULTITHREAD   2  /* nil */
#define SQLITE_CONFIG_SERIALIZED    3  /* nil */
```

```
#define SQLITE_CONFIG_MALLOC        4  /* sqlite3_mem_methods* */
#define SQLITE_CONFIG_GETMALLOC     5  /* sqlite3_mem_methods* */
#define SQLITE_CONFIG_SCRATCH       6  /* No longer used */
#define SQLITE_CONFIG_PAGECACHE     7  /* void*, int sz, int N */
#define SQLITE_CONFIG_HEAP          8  /* void*, int nByte, int min */
#define SQLITE_CONFIG_MEMSTATUS     9  /* boolean */
#define SQLITE_CONFIG_MUTEX        10  /* sqlite3_mutex_methods* */
#define SQLITE_CONFIG_GETMUTEX     11  /* sqlite3_mutex_methods* */
/* previously SQLITE_CONFIG_CHUNKALLOC 12 which is now unused. */
#define SQLITE_CONFIG_LOOKASIDE    13  /* int int */
#define SQLITE_CONFIG_PCACHE       14  /* no-op */
#define SQLITE_CONFIG_GETPCACHE    15  /* no-op */
#define SQLITE_CONFIG_LOG          16  /* xFunc, void* */
#define SQLITE_CONFIG_URI          17  /* int */
#define SQLITE_CONFIG_PCACHE2      18  /* sqlite3_pcache_methods2* */
#define SQLITE_CONFIG_GETPCACHE2   19  /* sqlite3_pcache_methods2* */
#define SQLITE_CONFIG_COVERING_INDEX_SCAN 20  /* int */
#define SQLITE_CONFIG_SQLLOG       21  /* xSqllog, void* */
#define SQLITE_CONFIG_MMAP_SIZE    22  /* sqlite3_int64, sqlite3_int64 */
#define SQLITE_CONFIG_WIN32_HEAPSIZE      23  /* int nByte */
#define SQLITE_CONFIG_PCACHE_HDRSZ        24  /* int *psz */
#define SQLITE_CONFIG_PMASZ               25  /* unsigned int szPma */
#define SQLITE_CONFIG_STMTJRNL_SPILL      26  /* int nByte */
#define SQLITE_CONFIG_SMALL_MALLOC        27  /* boolean */
#define SQLITE_CONFIG_SORTERREF_SIZE      28  /* int nByte */
#define SQLITE_CONFIG_MEMDB_MAXSIZE       29  /* sqlite3_int64 */
```

These constants are the available integer configuration options that can be passed as the first argument to the sqlite3_config() interface.

New configuration options may be added in future releases of SQLite. Existing configuration options might be discontinued. Applications should check the return code from sqlite3_config() to make sure that the call worked. The sqlite3_config() interface will return a non-zero error code if a discontinued or unsupported configuration option is invoked.

SQLITE_CONFIG_SINGLETHREAD
: There are no arguments to this option. This option sets the threading mode to Single-thread. In other words, it disables all mutexing and puts SQLite into a mode where it can only be used by a single thread. If SQLite is compiled with the SQLITE_THREADSAFE=0 compile-time option then it is not possible to change the threading mode from its default value of Single-thread and so sqlite3_config() will return SQLITE_ERROR if called with the SQLITE_CONFIG_SINGLETHREAD configuration option.

SQLITE_CONFIG_MULTITHREAD
: There are no arguments to this option. This option sets the threading mode to Multi-thread. In other words, it disables mutexing on database connection and prepared statement objects. The application is responsible for serializing access to database connections and prepared statements. But other mutexes are enabled so that SQLite will be safe to use in a multi-threaded environment as long as no two threads attempt to use the same database connection at the same time. If SQLite is compiled with the SQLITE_THREADSAFE=0 compile-time option then it is not possible to set the Multi-thread threading mode and sqlite3_config() will return SQLITE_ERROR if called with the SQLITE_CONFIG_MULTITHREAD configuration option.

SQLITE_CONFIG_SERIALIZED
: There are no arguments to this option. This option sets the threading mode to Serialized. In other words, this option enables all mutexes including the recursive mutexes on database connection and prepared statement objects. In this mode (which is the default when SQLite is compiled with SQLITE_THREADSAFE=1) the SQLite library will itself serialize access to database connections and prepared statements so that the application is free to use the same database connection or the same prepared statement in different threads at the same time. If SQLite is compiled with the SQLITE_THREADSAFE=0 compile-time option then it is not possible to set the Serialized threading mode and sqlite3_config() will return SQLITE_ERROR if called with the SQLITE_CONFIG_SERIALIZED configuration option.

SQLITE_CONFIG_MALLOC
: The SQLITE_CONFIG_MALLOC option takes a single argument which is a pointer to an instance of the sqlite3_mem_methods structure. The argument specifies alternative low-level memory allocation routines to be used in place of the memory allocation routines built into SQLite. SQLite makes its own private copy of the content of the sqlite3_mem_methods structure before the sqlite3_config() call returns.

SQLITE_CONFIG_GETMALLOC
: The SQLITE_CONFIG_GETMALLOC option takes a single argument which is a pointer to an instance of the sqlite3_mem_methods structure. The sqlite3_mem_methods structure is filled with the currently defined memory allocation routines. This option can be used to overload the default memory allocation routines with a wrapper that simulations memory allocation failure or tracks memory usage, for example.

SQLITE_CONFIG_SMALL_MALLOC
: The SQLITE_CONFIG_SMALL_MALLOC option takes single argument of type int, interpreted as a boolean, which if true provides a hint to SQLite that it should avoid large memory allocations if possible. SQLite will run faster if it is free to make large memory allocations, but some application might prefer to run slower in exchange for guarantees about memory fragmentation that are possible if large allocations are avoided. This hint is normally off.

SQLITE_CONFIG_MEMSTATUS
: The SQLITE_CONFIG_MEMSTATUS option takes single argument of type int, interpreted as a boolean, which enables or disables the collection of memory allocation statistics. When memory allocation statistics are disabled, the following SQLite interfaces become non-operational:

  - sqlite3_hard_heap_limit64()
  - sqlite3_memory_used()
  - sqlite3_memory_highwater()
  - sqlite3_soft_heap_limit64()
  - sqlite3_status64()

  Memory allocation statistics are enabled by default unless SQLite is compiled with SQLITE_DEFAULT_MEMSTATUS=0 in which case memory allocation statistics are disabled by default.

SQLITE_CONFIG_SCRATCH
The SQLITE_CONFIG_SCRATCH option is no longer used.

SQLITE_CONFIG_PAGECACHE
The SQLITE_CONFIG_PAGECACHE option specifies a memory pool that SQLite can use for the database page cache with the default page cache implementation. This configuration option is a no-op if an application-defined page cache implementation is loaded using the SQLITE_CONFIG_PCACHE2. There are three arguments to SQLITE_CONFIG_PAGECACHE: A pointer to 8-byte aligned memory (pMem), the size of each page cache line (sz), and the number of cache lines (N). The sz argument should be the size of the largest database page (a power of two between 512 and 65536) plus some extra bytes for each page header. The number of extra bytes needed by the page header can be determined using SQLITE_CONFIG_PCACHE_HDRSZ. It is harmless, apart from the wasted memory, for the sz parameter to be larger than necessary. The pMem argument must be either a NULL pointer or a pointer to an 8-byte aligned block of memory of at least sz*N bytes, otherwise subsequent behavior is undefined. When pMem is not NULL, SQLite will strive to use the memory provided to satisfy page cache needs, falling back to sqlite3_malloc() if a page cache line is larger than sz bytes or if all of the pMem buffer is exhausted. If pMem is NULL and N is non-zero, then each database connection does an initial bulk allocation for page cache memory from sqlite3_malloc() sufficient for N cache lines if N is positive or of -1024*N bytes if N is negative, . If additional page cache memory is needed beyond what is provided by the initial allocation, then SQLite goes to sqlite3_malloc() separately for each additional cache line.

SQLITE_CONFIG_HEAP
The SQLITE_CONFIG_HEAP option specifies a static memory buffer that SQLite will use for all of its dynamic memory allocation needs beyond those provided for by SQLITE_CONFIG_PAGECACHE. The SQLITE_CONFIG_HEAP option is only available if SQLite is compiled with either SQLITE_ENABLE_MEMSYS3 or SQLITE_ENABLE_MEMSYS5 and returns SQLITE_ERROR if invoked otherwise. There are three arguments to SQLITE_CONFIG_HEAP: An 8-byte aligned pointer to the memory, the number of bytes in the memory buffer, and the minimum allocation size. If the first pointer (the memory pointer) is NULL, then SQLite reverts to using its default memory allocator (the system malloc() implementation), undoing any prior invocation of SQLITE_CONFIG_MALLOC. If the memory pointer is not NULL then the alternative memory allocator is engaged to handle all of SQLites memory allocation needs. The first pointer (the memory pointer) must be aligned to an 8-byte boundary or subsequent behavior of SQLite will be undefined. The minimum allocation size is capped at 2**12. Reasonable values for the minimum allocation size are 2**5 through 2**8.

SQLITE_CONFIG_MUTEX
The SQLITE_CONFIG_MUTEX option takes a single argument which is a pointer to an instance of the sqlite3_mutex_methods structure. The argument specifies alternative low-level mutex routines to be used in place the mutex routines built into SQLite. SQLite makes a copy of the content of the sqlite3_mutex_methods structure before the call to sqlite3_config() returns. If SQLite is compiled with the SQLITE_THREADSAFE=0 compile-time option then the entire mutexing subsystem is omitted from the build and hence calls to sqlite3_config() with the SQLITE_CONFIG_MUTEX configuration option will return SQLITE_ERROR.

SQLITE_CONFIG_GETMUTEX
The SQLITE_CONFIG_GETMUTEX option takes a single argument which is a pointer to an instance of the sqlite3_mutex_methods structure. The sqlite3_mutex_methods structure is filled with the currently defined mutex routines. This option can be used to overload the default mutex allocation routines with a wrapper used to track mutex usage for performance profiling or testing, for example. If SQLite is compiled with the SQLITE_THREADSAFE=0 compile-time option then the entire mutexing subsystem is omitted from the build and hence calls to sqlite3_config() with the SQLITE_CONFIG_GETMUTEX configuration option will return SQLITE_ERROR.

SQLITE_CONFIG_LOOKASIDE
The SQLITE_CONFIG_LOOKASIDE option takes two arguments that determine the default size of lookaside memory on each database connection. The first argument is the size of each lookaside buffer slot and the second is the number of slots allocated to each database connection. SQLITE_CONFIG_LOOKASIDE sets the *default* lookaside size. The SQLITE_DBCONFIG_LOOKASIDE option to sqlite3_db_config() can be used to change the lookaside configuration on individual connections.

SQLITE_CONFIG_PCACHE2
The SQLITE_CONFIG_PCACHE2 option takes a single argument which is a pointer to an sqlite3_pcache_methods2 object. This object specifies the interface to a custom page cache implementation. SQLite makes a copy of the sqlite3_pcache_methods2 object.

SQLITE_CONFIG_GETPCACHE2
The SQLITE_CONFIG_GETPCACHE2 option takes a single argument which is a pointer to an sqlite3_pcache_methods2 object. SQLite copies of the current page cache implementation into that object.

SQLITE_CONFIG_LOG
The SQLITE_CONFIG_LOG option is used to configure the SQLite global error log. (The SQLITE_CONFIG_LOG option takes two arguments: a pointer to a function with a call signature of void(*)(void*,int,const char*), and a pointer to void. If the function pointer is not NULL, it is invoked by sqlite3_log() to process each logging event. If the function pointer is NULL, the sqlite3_log() interface becomes a no-op. The void pointer that is the second argument to SQLITE_CONFIG_LOG is passed through as the first parameter to the application-defined logger function whenever that function is invoked. The second parameter to the logger function is a copy of the first parameter to the corresponding sqlite3_log() call and is intended to be a result code or an extended result code. The third parameter passed to the logger is log message after formatting via sqlite3_snprintf(). The SQLite logging interface is not reentrant; the logger function supplied by the application must not invoke any SQLite interface. In a multi-threaded application, the application-defined logger function must be threadsafe.

SQLITE_CONFIG_URI
The SQLITE_CONFIG_URI option takes a single argument of type int. If non-zero, then URI handling is globally enabled. If the parameter is zero, then URI handling is globally disabled. If URI handling is globally enabled, all filenames passed to sqlite3_open(), sqlite3_open_v2(), sqlite3_open16() or specified as part of ATTACH commands are interpreted as URIs, regardless of whether or not the SQLITE_OPEN_URI flag is set when the database connection is opened. If it is globally disabled, filenames are only interpreted as URIs if the SQLITE_OPEN_URI flag is set when the database connection is opened. By default, URI handling is globally disabled. The default value may be changed by compiling with the SQLITE_USE_URI symbol defined.

SQLITE_CONFIG_COVERING_INDEX_SCAN

The SQLITE_CONFIG_COVERING_INDEX_SCAN option takes a single integer argument which is interpreted as a boolean in order to enable or disable the use of covering indices for full table scans in the query optimizer. The default setting is determined by the SQLITE_ALLOW_COVERING_INDEX_SCAN compile-time option, or is "on" if that compile-time option is omitted. The ability to disable the use of covering indices for full table scans is because some incorrectly coded legacy applications might malfunction when the optimization is enabled. Providing the ability to disable the optimization allows the older, buggy application code to work without change even with newer versions of SQLite.

SQLITE_CONFIG_PCACHE and SQLITE_CONFIG_GETPCACHE
: These options are obsolete and should not be used by new code. They are retained for backwards compatibility but are now no-ops.

SQLITE_CONFIG_SQLLOG
: This option is only available if sqlite is compiled with the SQLITE_ENABLE_SQLLOG pre-processor macro defined. The first argument should be a pointer to a function of type void(*)(void*,sqlite3*,const char*, int). The second should be of type (void*). The callback is invoked by the library in three separate circumstances, identified by the value passed as the fourth parameter. If the fourth parameter is 0, then the database connection passed as the second argument has just been opened. The third argument points to a buffer containing the name of the main database file. If the fourth parameter is 1, then the SQL statement that the third parameter points to has just been executed. Or, if the fourth parameter is 2, then the connection being passed as the second parameter is being closed. The third parameter is passed NULL In this case. An example of using this configuration option can be seen in the "test_sqllog.c" source file in the canonical SQLite source tree.

SQLITE_CONFIG_MMAP_SIZE
: SQLITE_CONFIG_MMAP_SIZE takes two 64-bit integer (sqlite3_int64) values that are the default mmap size limit (the default setting for PRAGMA mmap_size) and the maximum allowed mmap size limit. The default setting can be overridden by each database connection using either the PRAGMA mmap_size command, or by using the SQLITE_FCNTL_MMAP_SIZE file control. The maximum allowed mmap size will be silently truncated if necessary so that it does not exceed the compile-time maximum mmap size set by the SQLITE_MAX_MMAP_SIZE compile-time option. If either argument to this option is negative, then that argument is changed to its compile-time default.

SQLITE_CONFIG_WIN32_HEAPSIZE
: The SQLITE_CONFIG_WIN32_HEAPSIZE option is only available if SQLite is compiled for Windows with the SQLITE_WIN32_MALLOC pre-processor macro defined. SQLITE_CONFIG_WIN32_HEAPSIZE takes a 32-bit unsigned integer value that specifies the maximum size of the created heap.

SQLITE_CONFIG_PCACHE_HDRSZ
: The SQLITE_CONFIG_PCACHE_HDRSZ option takes a single parameter which is a pointer to an integer and writes into that integer the number of extra bytes per page required for each page in SQLITE_CONFIG_PAGECACHE. The amount of extra space required can change depending on the compiler, target platform, and SQLite version.

SQLITE_CONFIG_PMASZ
: The SQLITE_CONFIG_PMASZ option takes a single parameter which is an unsigned integer and sets the "Minimum PMA Size" for the multithreaded sorter to that integer. The default minimum PMA Size is set by the SQLITE_SORTER_PMASZ compile-time option. New threads are launched to help with sort operations when multithreaded sorting is enabled (using the PRAGMA threads command) and the amount of content to be sorted exceeds the page size times the minimum of the PRAGMA cache_size setting and this value.

SQLITE_CONFIG_STMTJRNL_SPILL
: The SQLITE_CONFIG_STMTJRNL_SPILL option takes a single parameter which becomes the statement journal spill-to-disk threshold. Statement journals are held in memory until their size (in bytes) exceeds this threshold, at which point they are written to disk. Or if the threshold is -1, statement journals are always held exclusively in memory. Since many statement journals never become large, setting the spill threshold to a value such as 64KiB can greatly reduce the amount of I/O required to support statement rollback. The default value for this setting is controlled by the SQLITE_STMTJRNL_SPILL compile-time option.

SQLITE_CONFIG_SORTERREF_SIZE
: The SQLITE_CONFIG_SORTERREF_SIZE option accepts a single parameter of type (int) - the new value of the sorter-reference size threshold. Usually, when SQLite uses an external sort to order records according to an ORDER BY clause, all fields required by the caller are present in the sorted records. However, if SQLite determines based on the declared type of a table column that its values are likely to be very large - larger than the configured sorter-reference size threshold - then a reference is stored in each sorted record and the required column values loaded from the database as records are returned in sorted order. The default value for this option is to never use this optimization. Specifying a negative value for this option restores the default behaviour. This option is only available if SQLite is compiled with the SQLITE_ENABLE_SORTER_REFERENCES compile-time option.

SQLITE_CONFIG_MEMDB_MAXSIZE
: The SQLITE_CONFIG_MEMDB_MAXSIZE option accepts a single parameter sqlite3_int64 parameter which is the default maximum size for an in-memory database created using sqlite3_deserialize(). This default maximum size can be adjusted up or down for individual databases using the SQLITE_FCNTL_SIZE_LIMIT file-control. If this configuration setting is never used, then the default maximum is determined by the SQLITE_MEMDB_DEFAULT_MAXSIZE compile-time option. If that compile-time option is not set, then the default maximum is 1073741824.

Database Connection Configuration Options

```
#define SQLITE_DBCONFIG_MAINDBNAME            1000 /* const char* */
#define SQLITE_DBCONFIG_LOOKASIDE             1001 /* void* int int */
#define SQLITE_DBCONFIG_ENABLE_FKEY           1002 /* int int* */
#define SQLITE_DBCONFIG_ENABLE_TRIGGER        1003 /* int int* */
#define SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER 1004 /* int int* */
#define SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION 1005 /* int int* */
#define SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE      1006 /* int int* */
#define SQLITE_DBCONFIG_ENABLE_QPSG           1007 /* int int* */
#define SQLITE_DBCONFIG_TRIGGER_EQP           1008 /* int int* */
#define SQLITE_DBCONFIG_RESET_DATABASE        1009 /* int int* */
#define SQLITE_DBCONFIG_DEFENSIVE             1010 /* int int* */
#define SQLITE_DBCONFIG_WRITABLE_SCHEMA       1011 /* int int* */
```

```
#define SQLITE_DBCONFIG_LEGACY_ALTER_TABLE    1012 /* int int* */
#define SQLITE_DBCONFIG_DQS_DML               1013 /* int int* */
#define SQLITE_DBCONFIG_DQS_DDL               1014 /* int int* */
#define SQLITE_DBCONFIG_ENABLE_VIEW           1015 /* int int* */
#define SQLITE_DBCONFIG_LEGACY_FILE_FORMAT    1016 /* int int* */
#define SQLITE_DBCONFIG_TRUSTED_SCHEMA        1017 /* int int* */
#define SQLITE_DBCONFIG_MAX                   1017 /* Largest DBCONFIG */
```

These constants are the available integer configuration options that can be passed as the second argument to the sqlite3_db_config() interface.

New configuration options may be added in future releases of SQLite. Existing configuration options might be discontinued. Applications should check the return code from sqlite3_db_config() to make sure that the call worked. The sqlite3_db_config() interface will return a non-zero error code if a discontinued or unsupported configuration option is invoked.

SQLITE_DBCONFIG_LOOKASIDE
: This option takes three additional arguments that determine the lookaside memory allocator configuration for the database connection. The first argument (the third parameter to sqlite3_db_config()) is a pointer to a memory buffer to use for lookaside memory. The first argument after the SQLITE_DBCONFIG_LOOKASIDE verb may be NULL in which case SQLite will allocate the lookaside buffer itself using sqlite3_malloc(). The second argument is the size of each lookaside buffer slot. The third argument is the number of slots. The size of the buffer in the first argument must be greater than or equal to the product of the second and third arguments. The buffer must be aligned to an 8-byte boundary. If the second argument to SQLITE_DBCONFIG_LOOKASIDE is not a multiple of 8, it is internally rounded down to the next smaller multiple of 8. The lookaside memory configuration for a database connection can only be changed when that connection is not currently using lookaside memory, or in other words when the "current value" returned by sqlite3_db_status(D,SQLITE_CONFIG_LOOKASIDE,...) is zero. Any attempt to change the lookaside memory configuration when lookaside memory is in use leaves the configuration unchanged and returns SQLITE_BUSY.

SQLITE_DBCONFIG_ENABLE_FKEY
: This option is used to enable or disable the enforcement of foreign key constraints. There should be two additional arguments. The first argument is an integer which is 0 to disable FK enforcement, positive to enable FK enforcement or negative to leave FK enforcement unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether FK enforcement is off or on following this call. The second parameter may be a NULL pointer, in which case the FK enforcement setting is not reported back.

SQLITE_DBCONFIG_ENABLE_TRIGGER
: This option is used to enable or disable triggers. There should be two additional arguments. The first argument is an integer which is 0 to disable triggers, positive to enable triggers or negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether triggers are disabled or enabled following this call. The second parameter may be a NULL pointer, in which case the trigger setting is not reported back.

SQLITE_DBCONFIG_ENABLE_VIEW
: This option is used to enable or disable views. There should be two additional arguments. The first argument is an integer which is 0 to disable views, positive to enable views or negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether views are disabled or enabled following this call. The second parameter may be a NULL pointer, in which case the view setting is not reported back.

SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER
: This option is used to enable or disable the fts3_tokenizer() function which is part of the FTS3 full-text search engine extension. There should be two additional arguments. The first argument is an integer which is 0 to disable fts3_tokenizer() or positive to enable fts3_tokenizer() or negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether fts3_tokenizer is disabled or enabled following this call. The second parameter may be a NULL pointer, in which case the new setting is not reported back.

SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION
: This option is used to enable or disable the sqlite3_load_extension() interface independently of the load_extension() SQL function. The sqlite3_enable_load_extension() API enables or disables both the C-API sqlite3_load_extension() and the SQL function load_extension(). There should be two additional arguments. When the first argument to this interface is 1, then only the C-API is enabled and the SQL function remains disabled. If the first argument to this interface is 0, then both the C-API and the SQL function are disabled. If the first argument is -1, then no changes are made to state of either the C-API or the SQL function. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether sqlite3_load_extension() interface is disabled or enabled following this call. The second parameter may be a NULL pointer, in which case the new setting is not reported back.

SQLITE_DBCONFIG_MAINDBNAME
: This option is used to change the name of the "main" database schema. The sole argument is a pointer to a constant UTF8 string which will become the new schema name in place of "main". SQLite does not make a copy of the new main schema name string, so the application must ensure that the argument passed into this DBCONFIG option is unchanged until after the database connection closes.

SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE
: Usually, when a database in wal mode is closed or detached from a database handle, SQLite checks if this will mean that there are now no connections at all to the database. If so, it performs a checkpoint operation before closing the connection. This option may be used to override this behaviour. The first parameter passed to this operation is an integer - positive to disable checkpoints-on-close, or zero (the default) to enable them, and negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether checkpoints-on-close have been disabled - 0 if they are not disabled, 1 if they are.

SQLITE_DBCONFIG_ENABLE_QPSG
: The SQLITE_DBCONFIG_ENABLE_QPSG option activates or deactivates the query planner stability guarantee (QPSG). When the QPSG is active, a single SQL query statement will always use the same algorithm regardless of values of bound parameters. The QPSG disables some query optimizations that look at the values of bound parameters, which can make some queries slower. But the QPSG has the advantage of more predictable behavior. With the QPSG active, SQLite will always use the same query plan in the field as was used during testing in the lab. The first argument to this setting is an integer which is

0 to disable the QPSG, positive to enable QPSG, or negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether the QPSG is disabled or enabled following this call.

SQLITE_DBCONFIG_TRIGGER_EQP
By default, the output of EXPLAIN QUERY PLAN commands does not include output for any operations performed by trigger programs. This option is used to set or clear (the default) a flag that governs this behavior. The first parameter passed to this operation is an integer - positive to enable output for trigger programs, or zero to disable it, or negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether output-for-triggers has been disabled - 0 if it is not disabled, 1 if it is.

SQLITE_DBCONFIG_RESET_DATABASE
Set the SQLITE_DBCONFIG_RESET_DATABASE flag and then run VACUUM in order to reset a database back to an empty database with no schema and no content. The following process works even for a badly corrupted database file:

1. If the database connection is newly opened, make sure it has read the database schema by preparing then discarding some query against the database, or calling sqlite3_table_column_metadata(), ignoring any errors. This step is only necessary if the application desires to keep the database in WAL mode after the reset if it was in WAL mode before the reset.
2. sqlite3_db_config(db, SQLITE_DBCONFIG_RESET_DATABASE, 1, 0);
3. sqlite3_exec(db, "VACUUM", 0, 0, 0);
4. sqlite3_db_config(db, SQLITE_DBCONFIG_RESET_DATABASE, 0, 0);

Because resetting a database is destructive and irreversible, the process requires the use of this obscure API and multiple steps to help ensure that it does not happen by accident.

SQLITE_DBCONFIG_DEFENSIVE
The SQLITE_DBCONFIG_DEFENSIVE option activates or deactivates the "defensive" flag for a database connection. When the defensive flag is enabled, language features that allow ordinary SQL to deliberately corrupt the database file are disabled. The disabled features include but are not limited to the following:

- The PRAGMA writable_schema=ON statement.
- The PRAGMA journal_mode=OFF statement.
- Writes to the sqlite_dbpage virtual table.
- Direct writes to shadow tables.

SQLITE_DBCONFIG_WRITABLE_SCHEMA
The SQLITE_DBCONFIG_WRITABLE_SCHEMA option activates or deactivates the "writable_schema" flag. This has the same effect and is logically equivalent to setting PRAGMA writable_schema=ON or PRAGMA writable_schema=OFF. The first argument to this setting is an integer which is 0 to disable the writable_schema, positive to enable writable_schema, or negative to leave the setting unchanged. The second parameter is a pointer to an integer into which is written 0 or 1 to indicate whether the writable_schema is enabled or disabled following this call.

SQLITE_DBCONFIG_LEGACY_ALTER_TABLE
The SQLITE_DBCONFIG_LEGACY_ALTER_TABLE option activates or deactivates the legacy behavior of the ALTER TABLE RENAME command such it behaves as it did prior to version 3.24.0 (2018-06-04). See the "Compatibility Notice" on the ALTER TABLE RENAME documentation for additional information. This feature can also be turned on and off using the PRAGMA legacy_alter_table statement.

SQLITE_DBCONFIG_DQS_DML
The SQLITE_DBCONFIG_DQS_DML option activates or deactivates the legacy double-quoted string literal misfeature for DML statements only, that is DELETE, INSERT, SELECT, and UPDATE statements. The default value of this setting is determined by the -DSQLITE_DQS compile-time option.

SQLITE_DBCONFIG_DQS_DDL
The SQLITE_DBCONFIG_DQS option activates or deactivates the legacy double-quoted string literal misfeature for DDL statements, such as CREATE TABLE and CREATE INDEX. The default value of this setting is determined by the -DSQLITE_DQS compile-time option.

SQLITE_DBCONFIG_TRUSTED_SCHEMA
The SQLITE_DBCONFIG_TRUSTED_SCHEMA option tells SQLite to assume that database schemas (the contents of the sqlite_master tables) are untainted by malicious content. When the SQLITE_DBCONFIG_TRUSTED_SCHEMA option is disabled, SQLite takes additional defensive steps to protect the application from harm including:

- Prohibit the use of SQL functions inside triggers, views, CHECK constraints, DEFAULT clauses, expression indexes, partial indexes, or generated columns unless those functions are tagged with SQLITE_INNOCUOUS.
- Prohibit the use of virtual tables inside of triggers or views unless those virtual tables are tagged with SQLITE_VTAB_INNOCUOUS.

This setting defaults to "on" for legacy compatibility, however all applications are advised to turn it off if possible. This setting can also be controlled using the PRAGMA trusted_schema statement.

SQLITE_DBCONFIG_LEGACY_FILE_FORMAT
The SQLITE_DBCONFIG_LEGACY_FILE_FORMAT option activates or deactivates the legacy file format flag. When activated, this flag causes all newly created database file to have a schema format version number (the 4-byte integer found at offset 44 into the database header) of 1. This in turn means that the resulting database file will be readable and writable by any SQLite version back to 3.0.0 (2004-06-18). Without this setting, newly created databases are generally not understandable by SQLite versions prior to 3.3.0 (2006-01-11). As these words are written, there is now scarcely any need to generated database files that are compatible all the way back to version 3.0.0, and so this setting is of little practical use, but is provided so that SQLite can continue to claim the ability to generate new database files that are compatible with version 3.0.0.

Note that when the SQLITE_DBCONFIG_LEGACY_FILE_FORMAT setting is on, the VACUUM command will fail with an obscure error when attempting to process a table with generated columns and a descending index. This is not considered a bug since SQLite versions 3.3.0 and earlier do not support either generated columns or decending indexes.

Authorizer Return Codes

```
#define SQLITE_DENY   1   /* Abort the SQL statement with an error */
#define SQLITE_IGNORE 2   /* Don't allow access, but don't generate an error */
```

The authorizer callback function must return either SQLITE_OK or one of these two constants in order to signal SQLite whether or not the action is permitted. See the authorizer documentation for additional information.

Note that SQLITE_IGNORE is also used as a conflict resolution mode returned from the sqlite3_vtab_on_conflict() interface.

Flags for sqlite3_deserialize()

```
#define SQLITE_DESERIALIZE_FREEONCLOSE 1 /* Call sqlite3_free() on close */
#define SQLITE_DESERIALIZE_RESIZEABLE  2 /* Resize using sqlite3_realloc64() */
#define SQLITE_DESERIALIZE_READONLY    4 /* Database is read-only */
```

The following are allowed values for 6th argument (the F argument) to the sqlite3_deserialize(D,S,P,N,M,F) interface.

The SQLITE_DESERIALIZE_FREEONCLOSE means that the database serialization in the P argument is held in memory obtained from sqlite3_malloc64() and that SQLite should take ownership of this memory and automatically free it when it has finished using it. Without this flag, the caller is responsible for freeing any dynamically allocated memory.

The SQLITE_DESERIALIZE_RESIZEABLE flag means that SQLite is allowed to grow the size of the database using calls to sqlite3_realloc64(). This flag should only be used if SQLITE_DESERIALIZE_FREEONCLOSE is also used. Without this flag, the deserialized database cannot increase in size beyond the number of bytes specified by the M parameter.

The SQLITE_DESERIALIZE_READONLY flag means that the deserialized database should be treated as read-only.

Function Flags

```
#define SQLITE_DETERMINISTIC    0x000000800
#define SQLITE_DIRECTONLY       0x000080000
#define SQLITE_SUBTYPE          0x000100000
#define SQLITE_INNOCUOUS        0x000200000
```

These constants may be ORed together with the preferred text encoding as the fourth argument to sqlite3_create_function(), sqlite3_create_function16(), or sqlite3_create_function_v2().

SQLITE_DETERMINISTIC
: The SQLITE_DETERMINISTIC flag means that the new function always gives the same output when the input parameters are the same. The abs() function is deterministic, for example, but randomblob() is not. Functions must be deterministic in order to be used in certain contexts such as with the WHERE clause of partial indexes or in generated columns. SQLite might also optimize deterministic functions by factoring them out of inner loops.

SQLITE_DIRECTONLY
: The SQLITE_DIRECTONLY flag means that the function may only be invoked from top-level SQL, and cannot be used in VIEWs or TRIGGERs nor in schema structures such as CHECK constraints, DEFAULT clauses, expression indexes, partial indexes, or generated columns. The SQLITE_DIRECTONLY flags is a security feature which is recommended for all application-defined SQL functions, and especially for functions that have side-effects or that could potentially leak sensitive information.

SQLITE_INNOCUOUS
: The SQLITE_INNOCUOUS flag means that the function is unlikely to cause problems even if misused. An innocuous function should have no side effects and should not depend on any values other than its input parameters. The abs() function is an example of an innocuous function. The load_extension() SQL function is not innocuous because of its side effects.

  SQLITE_INNOCUOUS is similar to SQLITE_DETERMINISTIC, but is not exactly the same. The random() function is an example of a function that is innocuous but not deterministic.

  Some heightened security settings (SQLITE_DBCONFIG_TRUSTED_SCHEMA and PRAGMA trusted_schema=OFF) disable the use of SQL functions inside views and triggers and in schema structures such as CHECK constraints, DEFAULT clauses, expression indexes, partial indexes, and generated columns unless the function is tagged with SQLITE_INNOCUOUS. Most built-in functions are innocuous. Developers are advised to avoid using the SQLITE_INNOCUOUS flag for application-defined functions unless the function has been carefully audited and found to be free of potentially security-adverse side-effects and information-leaks.

SQLITE_SUBTYPE
: The SQLITE_SUBTYPE flag indicates to SQLite that a function may call sqlite3_value_subtype() to inspect the sub-types of its arguments. Specifying this flag makes no difference for scalar or aggregate user functions. However, if it is not specified for a user-defined window function, then any sub-types belonging to arguments passed to the window function may be discarded before the window function is called (i.e. sqlite3_value_subtype() will always return 0).

Conflict resolution modes

```
#define SQLITE_ROLLBACK 1
/* #define SQLITE_IGNORE 2 // Also used by sqlite3_authorizer() callback */
#define SQLITE_FAIL     3
/* #define SQLITE_ABORT 4  // Also an error code */
#define SQLITE_REPLACE  5
```

These constants are returned by sqlite3_vtab_on_conflict() to inform a virtual table implementation what the ON CONFLICT mode is for the SQL statement being evaluated.

Note that the SQLITE_IGNORE constant is also used as a potential return value from the sqlite3_set_authorizer() callback and that SQLITE_ABORT is also a result code.

Standard File Control Opcodes

```
#define SQLITE_FCNTL_LOCKSTATE               1
#define SQLITE_FCNTL_GET_LOCKPROXYFILE       2
#define SQLITE_FCNTL_SET_LOCKPROXYFILE       3
#define SQLITE_FCNTL_LAST_ERRNO              4
#define SQLITE_FCNTL_SIZE_HINT               5
#define SQLITE_FCNTL_CHUNK_SIZE              6
#define SQLITE_FCNTL_FILE_POINTER            7
#define SQLITE_FCNTL_SYNC_OMITTED            8
#define SQLITE_FCNTL_WIN32_AV_RETRY          9
#define SQLITE_FCNTL_PERSIST_WAL            10
#define SQLITE_FCNTL_OVERWRITE              11
#define SQLITE_FCNTL_VFSNAME                12
#define SQLITE_FCNTL_POWERSAFE_OVERWRITE    13
#define SQLITE_FCNTL_PRAGMA                 14
#define SQLITE_FCNTL_BUSYHANDLER            15
#define SQLITE_FCNTL_TEMPFILENAME           16
#define SQLITE_FCNTL_MMAP_SIZE              18
#define SQLITE_FCNTL_TRACE                  19
#define SQLITE_FCNTL_HAS_MOVED              20
#define SQLITE_FCNTL_SYNC                   21
#define SQLITE_FCNTL_COMMIT_PHASETWO        22
#define SQLITE_FCNTL_WIN32_SET_HANDLE       23
#define SQLITE_FCNTL_WAL_BLOCK              24
#define SQLITE_FCNTL_ZIPVFS                 25
#define SQLITE_FCNTL_RBU                    26
#define SQLITE_FCNTL_VFS_POINTER            27
#define SQLITE_FCNTL_JOURNAL_POINTER        28
#define SQLITE_FCNTL_WIN32_GET_HANDLE       29
#define SQLITE_FCNTL_PDB                    30
#define SQLITE_FCNTL_BEGIN_ATOMIC_WRITE     31
#define SQLITE_FCNTL_COMMIT_ATOMIC_WRITE    32
#define SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE  33
#define SQLITE_FCNTL_LOCK_TIMEOUT           34
#define SQLITE_FCNTL_DATA_VERSION           35
#define SQLITE_FCNTL_SIZE_LIMIT             36
#define SQLITE_FCNTL_CKPT_DONE              37
```

These integer constants are opcodes for the xFileControl method of the sqlite3_io_methods object and for the sqlite3_file_control() interface.

- The SQLITE_FCNTL_LOCKSTATE opcode is used for debugging. This opcode causes the xFileControl method to write the current state of the lock (one of SQLITE_LOCK_NONE, SQLITE_LOCK_SHARED, SQLITE_LOCK_RESERVED, SQLITE_LOCK_PENDING, or SQLITE_LOCK_EXCLUSIVE) into an integer that the pArg argument points to. This capability is used during testing and is only available when the SQLITE_TEST compile-time option is used.

- The SQLITE_FCNTL_SIZE_HINT opcode is used by SQLite to give the VFS layer a hint of how large the database file will grow to be during the current transaction. This hint is not guaranteed to be accurate but it is often close. The underlying VFS might choose to preallocate database file space based on this hint in order to help writes to the database file run faster.

- The SQLITE_FCNTL_SIZE_LIMIT opcode is used by in-memory VFS that implements sqlite3_deserialize() to set an upper bound on the size of the in-memory database. The argument is a pointer to a sqlite3_int64. If the integer pointed to is negative, then it is filled in with the current limit. Otherwise the limit is set to the larger of the value of the integer pointed to and the current database size. The integer pointed to is set to the new limit.

- The SQLITE_FCNTL_CHUNK_SIZE opcode is used to request that the VFS extends and truncates the database file in chunks of a size specified by the user. The fourth argument to sqlite3_file_control() should point to an integer (type int) containing the new chunk-size to use for the nominated database. Allocating database file space in large chunks (say 1MB at a time), may reduce file-system fragmentation and improve performance on some systems.

- The SQLITE_FCNTL_FILE_POINTER opcode is used to obtain a pointer to the sqlite3_file object associated with a particular database connection. See also SQLITE_FCNTL_JOURNAL_POINTER.

- The SQLITE_FCNTL_JOURNAL_POINTER opcode is used to obtain a pointer to the sqlite3_file object associated with the journal file (either the rollback journal or the write-ahead log) for a particular database connection. See also SQLITE_FCNTL_FILE_POINTER.

- No longer in use.

- The SQLITE_FCNTL_SYNC opcode is generated internally by SQLite and sent to the VFS immediately before the xSync method is invoked on a database file descriptor. Or, if the xSync method is not invoked because the user has configured SQLite with PRAGMA synchronous=OFF it is invoked in place of the xSync method. In most cases, the pointer argument passed with this file-control is NULL. However, if the database file is being synced as part of a multi-database commit, the argument points to a nul-terminated string containing the transactions master-journal file name. VFSes that do not need this signal should silently ignore this opcode. Applications should not call sqlite3_file_control() with this opcode as doing so may disrupt the operation of the specialized VFSes that do require it.

- The SQLITE_FCNTL_COMMIT_PHASETWO opcode is generated internally by SQLite and sent to the VFS after a transaction has been committed immediately but before the database is unlocked. VFSes that do not need this signal should silently ignore this opcode. Applications should not call sqlite3_file_control() with this opcode as doing so may disrupt the operation of the specialized VFSes that do require it.

- The SQLITE_FCNTL_WIN32_AV_RETRY opcode is used to configure automatic retry counts and intervals for certain disk I/O operations for the windows VFS in order to provide robustness in the presence of anti-virus programs. By default, the windows VFS will retry file read, file write, and file delete operations up to 10 times, with a delay of 25 milliseconds before the first retry and with the delay increasing by an additional 25 milliseconds with each subsequent retry. This opcode allows these

two values (10 retries and 25 milliseconds of delay) to be adjusted. The values are changed for all database connections within the same process. The argument is a pointer to an array of two integers where the first integer is the new retry count and the second integer is the delay. If either integer is negative, then the setting is not changed but instead the prior value of that setting is written into the array entry, allowing the current retry settings to be interrogated. The zDbName parameter is ignored.

- The SQLITE_FCNTL_PERSIST_WAL opcode is used to set or query the persistent Write Ahead Log setting. By default, the auxiliary write ahead log (WAL file) and shared memory files used for transaction control are automatically deleted when the latest connection to the database closes. Setting persistent WAL mode causes those files to persist after close. Persisting the files is useful when other processes that do not have write permission on the directory containing the database file want to read the database file, as the WAL and shared memory files must exist in order for the database to be readable. The fourth parameter to sqlite3_file_control() for this opcode should be a pointer to an integer. That integer is 0 to disable persistent WAL mode or 1 to enable persistent WAL mode. If the integer is -1, then it is overwritten with the current WAL persistence setting.

- The SQLITE_FCNTL_POWERSAFE_OVERWRITE opcode is used to set or query the persistent "powersafe-overwrite" or "PSOW" setting. The PSOW setting determines the SQLITE_IOCAP_POWERSAFE_OVERWRITE bit of the xDeviceCharacteristics methods. The fourth parameter to sqlite3_file_control() for this opcode should be a pointer to an integer. That integer is 0 to disable zero-damage mode or 1 to enable zero-damage mode. If the integer is -1, then it is overwritten with the current zero-damage mode setting.

- The SQLITE_FCNTL_OVERWRITE opcode is invoked by SQLite after opening a write transaction to indicate that, unless it is rolled back for some reason, the entire database file will be overwritten by the current transaction. This is used by VACUUM operations.

- The SQLITE_FCNTL_VFSNAME opcode can be used to obtain the names of all VFSes in the VFS stack. The names are of all VFS shims and the final bottom-level VFS are written into memory obtained from sqlite3_malloc() and the result is stored in the char* variable that the fourth parameter of sqlite3_file_control() points to. The caller is responsible for freeing the memory when done. As with all file-control actions, there is no guarantee that this will actually do anything. Callers should initialize the char* variable to a NULL pointer in case this file-control is not implemented. This file-control is intended for diagnostic use only.

- The SQLITE_FCNTL_VFS_POINTER opcode finds a pointer to the top-level VFSes currently in use. The argument X in sqlite3_file_control(db,SQLITE_FCNTL_VFS_POINTER,X) must be of type "sqlite3_vfs **". This opcodes will set *X to a pointer to the top-level VFS. When there are multiple VFS shims in the stack, this opcode finds the upper-most shim only.

- Whenever a PRAGMA statement is parsed, an SQLITE_FCNTL_PRAGMA file control is sent to the open sqlite3_file object corresponding to the database file to which the pragma statement refers. The argument to the SQLITE_FCNTL_PRAGMA file control is an array of pointers to strings (char**) in which the second element of the array is the name of the pragma and the third element is the argument to the pragma or NULL if the pragma has no argument. The handler for an SQLITE_FCNTL_PRAGMA file control can optionally make the first element of the char** argument point to a string obtained from sqlite3_mprintf() or the equivalent and that string will become the result of the pragma or the error message if the pragma fails. If the SQLITE_FCNTL_PRAGMA file control returns SQLITE_NOTFOUND, then normal PRAGMA processing continues. If the SQLITE_FCNTL_PRAGMA file control returns SQLITE_OK, then the parser assumes that the VFS has handled the PRAGMA itself and the parser generates a no-op prepared statement if result string is NULL, or that returns a copy of the result string if the string is non-NULL. If the SQLITE_FCNTL_PRAGMA file control returns any result code other than SQLITE_OK or SQLITE_NOTFOUND, that means that the VFS encountered an error while handling the PRAGMA and the compilation of the PRAGMA fails with an error. The SQLITE_FCNTL_PRAGMA file control occurs at the beginning of pragma statement analysis and so it is able to override built-in PRAGMA statements.

- The SQLITE_FCNTL_BUSYHANDLER file-control may be invoked by SQLite on the database file handle shortly after it is opened in order to provide a custom VFS with access to the connection's busy-handler callback. The argument is of type (void**) - an array of two (void *) values. The first (void *) actually points to a function of type (int (*)(void *)). In order to invoke the connection's busy-handler, this function should be invoked with the second (void *) in the array as the only argument. If it returns non-zero, then the operation should be retried. If it returns zero, the custom VFS should abandon the current operation.

- Applications can invoke the SQLITE_FCNTL_TEMPFILENAME file-control to have SQLite generate a temporary filename using the same algorithm that is followed to generate temporary filenames for TEMP tables and other internal uses. The argument should be a char** which will be filled with the filename written into memory obtained from sqlite3_malloc(). The caller should invoke sqlite3_free() on the result to avoid a memory leak.

- The SQLITE_FCNTL_MMAP_SIZE file control is used to query or set the maximum number of bytes that will be used for memory-mapped I/O. The argument is a pointer to a value of type sqlite3_int64 that is an advisory maximum number of bytes in the file to memory map. The pointer is overwritten with the old value. The limit is not changed if the value originally pointed to is negative, and so the current limit can be queried by passing in a pointer to a negative number. This file-control is used internally to implement PRAGMA mmap_size.

- The SQLITE_FCNTL_TRACE file control provides advisory information to the VFS about what the higher layers of the SQLite stack are doing. This file control is used by some VFS activity tracing shims. The argument is a zero-terminated string. Higher layers in the SQLite stack may generate instances of this file control if the SQLITE_USE_FCNTL_TRACE compile-time option is enabled.

- The SQLITE_FCNTL_HAS_MOVED file control interprets its argument as a pointer to an integer and it writes a boolean into that integer depending on whether or not the file has been renamed, moved, or deleted since it was first opened.

- The SQLITE_FCNTL_WIN32_GET_HANDLE opcode can be used to obtain the underlying native file handle associated with a file handle. This file control interprets its argument as a pointer to a native file handle and writes the resulting value there.

- The SQLITE_FCNTL_WIN32_SET_HANDLE opcode is used for debugging. This opcode causes the xFileControl method to swap the file handle with the one pointed to by the pArg argument. This capability is used during testing and only needs to be supported when SQLITE_TEST is defined.

- The SQLITE_FCNTL_WAL_BLOCK is a signal to the VFS layer that it might be advantageous to block on the next WAL lock if the lock is not immediately available. The WAL subsystem issues this signal during rare circumstances in order to fix a problem with priority inversion. Applications should *not* use this file-control.

- The SQLITE_FCNTL_ZIPVFS opcode is implemented by zipvfs only. All other VFS should return SQLITE_NOTFOUND for this opcode.

- The SQLITE_FCNTL_RBU opcode is implemented by the special VFS used by the RBU extension only. All other VFS should return SQLITE_NOTFOUND for this opcode.

- If the SQLITE_FCNTL_BEGIN_ATOMIC_WRITE opcode returns SQLITE_OK, then the file descriptor is placed in "batch write mode", which means all subsequent write operations will be deferred and done atomically at the next SQLITE_FCNTL_COMMIT_ATOMIC_WRITE. Systems that do not support batch atomic writes will return SQLITE_NOTFOUND. Following a successful SQLITE_FCNTL_BEGIN_ATOMIC_WRITE and prior to the closing SQLITE_FCNTL_COMMIT_ATOMIC_WRITE or SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE, SQLite will make no VFS interface calls on the same sqlite3_file file descriptor except for calls to the xWrite method and the xFileControl method with SQLITE_FCNTL_SIZE_HINT.

- The SQLITE_FCNTL_COMMIT_ATOMIC_WRITE opcode causes all write operations since the previous successful call to SQLITE_FCNTL_BEGIN_ATOMIC_WRITE to be performed atomically. This file control returns SQLITE_OK if and only if the writes were all performed successfully and have been committed to persistent storage. Regardless of whether or not it is successful, this file control takes the file descriptor out of batch write mode so that all subsequent write operations are independent. SQLite will never invoke SQLITE_FCNTL_COMMIT_ATOMIC_WRITE without a prior successful call to SQLITE_FCNTL_BEGIN_ATOMIC_WRITE.

- The SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE opcode causes all write operations since the previous successful call to SQLITE_FCNTL_BEGIN_ATOMIC_WRITE to be rolled back. This file control takes the file descriptor out of batch write mode so that all subsequent write operations are independent. SQLite will never invoke SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE without a prior successful call to SQLITE_FCNTL_BEGIN_ATOMIC_WRITE.

- The SQLITE_FCNTL_LOCK_TIMEOUT opcode causes attempts to obtain a file lock using the xLock or xShmLock methods of the VFS to wait for up to M milliseconds before failing, where M is the single unsigned integer parameter.

- The SQLITE_FCNTL_DATA_VERSION opcode is used to detect changes to a database file. The argument is a pointer to a 32-bit unsigned integer. The "data version" for the pager is written into the pointer. The "data version" changes whenever any change occurs to the corresponding database file, either through SQL statements on the same database connection or through transactions committed by separate database connections possibly in other processes. The sqlite3_total_changes() interface can be used to find if any database on the connection has changed, but that interface responds to changes on TEMP as well as MAIN and does not provide a mechanism to detect changes to MAIN only. Also, the sqlite3_total_changes() interface responds to internal changes only and omits changes made by other database connections. The PRAGMA data_version command provides a mechanism to detect changes to a single attached database that occur due to other database connections, but omits changes implemented by the database connection on which it is called. This file control is the only mechanism to detect changes that happen either internally or externally and that are associated with a particular attached database.

- The SQLITE_FCNTL_CKPT_DONE opcode is invoked from within a checkpoint in wal mode after the client has finished copying pages from the wal file to the database file, but before the *-shm file is updated to record the fact that the pages have been checkpointed.

Virtual Table Constraint Operator Codes

```
#define SQLITE_INDEX_CONSTRAINT_EQ         2
#define SQLITE_INDEX_CONSTRAINT_GT         4
#define SQLITE_INDEX_CONSTRAINT_LE         8
#define SQLITE_INDEX_CONSTRAINT_LT        16
#define SQLITE_INDEX_CONSTRAINT_GE        32
#define SQLITE_INDEX_CONSTRAINT_MATCH     64
#define SQLITE_INDEX_CONSTRAINT_LIKE      65
#define SQLITE_INDEX_CONSTRAINT_GLOB      66
#define SQLITE_INDEX_CONSTRAINT_REGEXP    67
#define SQLITE_INDEX_CONSTRAINT_NE        68
#define SQLITE_INDEX_CONSTRAINT_ISNOT     69
#define SQLITE_INDEX_CONSTRAINT_ISNOTNULL 70
#define SQLITE_INDEX_CONSTRAINT_ISNULL    71
#define SQLITE_INDEX_CONSTRAINT_IS        72
#define SQLITE_INDEX_CONSTRAINT_FUNCTION 150
```

These macros define the allowed values for the sqlite3_index_info.aConstraint[].op field. Each value represents an operator that is part of a constraint term in the wHERE clause of a query that uses a virtual table.

Device Characteristics

```
#define SQLITE_IOCAP_ATOMIC                 0x00000001
#define SQLITE_IOCAP_ATOMIC512              0x00000002
#define SQLITE_IOCAP_ATOMIC1K               0x00000004
#define SQLITE_IOCAP_ATOMIC2K               0x00000008
#define SQLITE_IOCAP_ATOMIC4K               0x00000010
#define SQLITE_IOCAP_ATOMIC8K               0x00000020
#define SQLITE_IOCAP_ATOMIC16K              0x00000040
#define SQLITE_IOCAP_ATOMIC32K              0x00000080
#define SQLITE_IOCAP_ATOMIC64K              0x00000100
#define SQLITE_IOCAP_SAFE_APPEND            0x00000200
#define SQLITE_IOCAP_SEQUENTIAL             0x00000400
#define SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN  0x00000800
#define SQLITE_IOCAP_POWERSAFE_OVERWRITE    0x00001000
#define SQLITE_IOCAP_IMMUTABLE              0x00002000
#define SQLITE_IOCAP_BATCH_ATOMIC           0x00004000
```

The xDeviceCharacteristics method of the sqlite3_io_methods object returns an integer which is a vector of these bit values expressing I/O characteristics of the mass storage device that holds the file that the sqlite3_io_methods refers to.

The SQLITE_IOCAP_ATOMIC property means that all writes of any size are atomic. The SQLITE_IOCAP_ATOMICnnn values mean that writes of blocks that are nnn bytes in size and are aligned to an address which is an integer multiple of nnn are atomic. The SQLITE_IOCAP_SAFE_APPEND value means that when data is appended to a file, the data is appended first then the size of the file is extended, never the other way around. The SQLITE_IOCAP_SEQUENTIAL property means that information is written to disk in the same order as calls to xWrite(). The SQLITE_IOCAP_POWERSAFE_OVERWRITE property means that after reboot following a crash or power loss, the only bytes in a file that were written at the application level might have changed and that adjacent bytes, even bytes within the same sector are guaranteed to be unchanged. The SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN flag indicates that a file cannot be deleted when open. The SQLITE_IOCAP_IMMUTABLE flag indicates that the file is on read-only media and cannot be changed even by processes with elevated privileges.

The SQLITE_IOCAP_BATCH_ATOMIC property means that the underlying filesystem supports doing multiple write operations atomically when those write operations are bracketed by SQLITE_FCNTL_BEGIN_ATOMIC_WRITE and SQLITE_FCNTL_COMMIT_ATOMIC_WRITE.

File Locking Levels

```
#define SQLITE_LOCK_NONE          0
#define SQLITE_LOCK_SHARED        1
#define SQLITE_LOCK_RESERVED      2
#define SQLITE_LOCK_PENDING       3
#define SQLITE_LOCK_EXCLUSIVE     4
```

SQLite uses one of these integer values as the second argument to calls it makes to the xLock() and xUnlock() methods of an sqlite3_io_methods object.

Mutex Types

```
#define SQLITE_MUTEX_FAST             0
#define SQLITE_MUTEX_RECURSIVE        1
#define SQLITE_MUTEX_STATIC_MASTER    2
#define SQLITE_MUTEX_STATIC_MEM       3  /* sqlite3_malloc() */
#define SQLITE_MUTEX_STATIC_MEM2      4  /* NOT USED */
#define SQLITE_MUTEX_STATIC_OPEN      4  /* sqlite3BtreeOpen() */
#define SQLITE_MUTEX_STATIC_PRNG      5  /* sqlite3_randomness() */
#define SQLITE_MUTEX_STATIC_LRU       6  /* lru page list */
#define SQLITE_MUTEX_STATIC_LRU2      7  /* NOT USED */
#define SQLITE_MUTEX_STATIC_PMEM      7  /* sqlite3PageMalloc() */
#define SQLITE_MUTEX_STATIC_APP1      8  /* For use by application */
#define SQLITE_MUTEX_STATIC_APP2      9  /* For use by application */
#define SQLITE_MUTEX_STATIC_APP3     10  /* For use by application */
#define SQLITE_MUTEX_STATIC_VFS1     11  /* For use by built-in VFS */
#define SQLITE_MUTEX_STATIC_VFS2     12  /* For use by extension VFS */
#define SQLITE_MUTEX_STATIC_VFS3     13  /* For use by application VFS */
```

The sqlite3_mutex_alloc() interface takes a single argument which is one of these integer constants.

The set of static mutexes may change from one SQLite release to the next. Applications that override the built-in mutex logic must be prepared to accommodate additional static mutexes.

Flags For File Open Operations

```
#define SQLITE_OPEN_READONLY         0x00000001  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_READWRITE        0x00000002  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_CREATE           0x00000004  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_DELETEONCLOSE    0x00000008  /* VFS only */
#define SQLITE_OPEN_EXCLUSIVE        0x00000010  /* VFS only */
#define SQLITE_OPEN_AUTOPROXY        0x00000020  /* VFS only */
#define SQLITE_OPEN_URI              0x00000040  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_MEMORY           0x00000080  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_MAIN_DB          0x00000100  /* VFS only */
#define SQLITE_OPEN_TEMP_DB          0x00000200  /* VFS only */
#define SQLITE_OPEN_TRANSIENT_DB     0x00000400  /* VFS only */
#define SQLITE_OPEN_MAIN_JOURNAL     0x00000800  /* VFS only */
#define SQLITE_OPEN_TEMP_JOURNAL     0x00001000  /* VFS only */
#define SQLITE_OPEN_SUBJOURNAL       0x00002000  /* VFS only */
#define SQLITE_OPEN_MASTER_JOURNAL   0x00004000  /* VFS only */
#define SQLITE_OPEN_NOMUTEX          0x00008000  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_FULLMUTEX        0x00010000  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_SHAREDCACHE      0x00020000  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_PRIVATECACHE     0x00040000  /* Ok for sqlite3_open_v2() */
#define SQLITE_OPEN_WAL              0x00080000  /* VFS only */
#define SQLITE_OPEN_NOFOLLOW         0x01000000  /* Ok for sqlite3_open_v2() */
```

These bit values are intended for use in the 3rd parameter to the sqlite3_open_v2() interface and in the 4th parameter to the sqlite3_vfs.xOpen method.

Prepare Flags

```
#define SQLITE_PREPARE_PERSISTENT             0x01
#define SQLITE_PREPARE_NORMALIZE              0x02
#define SQLITE_PREPARE_NO_VTAB                0x04
```

These constants define various flags that can be passed into "prepFlags" parameter of the sqlite3_prepare_v3() and sqlite3_prepare16_v3() interfaces.

New flags may be added in future releases of SQLite.

SQLITE_PREPARE_PERSISTENT
: The SQLITE_PREPARE_PERSISTENT flag is a hint to the query planner that the prepared statement will be retained for a long time and probably reused many times. Without this flag, sqlite3_prepare_v3() and sqlite3_prepare16_v3() assume that the prepared statement will be used just once or at most a few times and then destroyed using sqlite3_finalize() relatively soon. The current implementation acts on this hint by avoiding the use of lookaside memory so as not to deplete the limited store of lookaside memory. Future versions of SQLite may act on this hint differently.

SQLITE_PREPARE_NORMALIZE
: The SQLITE_PREPARE_NORMALIZE flag is a no-op. This flag used to be required for any prepared statement that wanted to use the sqlite3_normalized_sql() interface. However, the sqlite3_normalized_sql() interface is now available to all prepared statements, regardless of whether or not they use this flag.

SQLITE_PREPARE_NO_VTAB
: The SQLITE_PREPARE_NO_VTAB flag causes the SQL compiler to return an error (error code SQLITE_ERROR) if the statement uses any virtual tables.

Prepared Statement Scan Status Opcodes

```
#define SQLITE_SCANSTAT_NLOOP    0
#define SQLITE_SCANSTAT_NVISIT   1
#define SQLITE_SCANSTAT_EST      2
#define SQLITE_SCANSTAT_NAME     3
#define SQLITE_SCANSTAT_EXPLAIN  4
#define SQLITE_SCANSTAT_SELECTID 5
```

The following constants can be used for the T parameter to the sqlite3_stmt_scanstatus(S,X,T,V) interface. Each constant designates a different metric for sqlite3_stmt_scanstatus() to return.

When the value returned to V is a string, space to hold that string is managed by the prepared statement S and will be automatically freed when S is finalized.

SQLITE_SCANSTAT_NLOOP
: The sqlite3_int64 variable pointed to by the V parameter will be set to the total number of times that the X-th loop has run.

SQLITE_SCANSTAT_NVISIT
: The sqlite3_int64 variable pointed to by the V parameter will be set to the total number of rows examined by all iterations of the X-th loop.

SQLITE_SCANSTAT_EST
: The "double" variable pointed to by the V parameter will be set to the query planner's estimate for the average number of rows output from each iteration of the X-th loop. If the query planner's estimates was accurate, then this value will approximate the quotient NVISIT/NLOOP and the product of this value for all prior loops with the same SELECTID will be the NLOOP value for the current loop.

SQLITE_SCANSTAT_NAME
: The "const char *" variable pointed to by the V parameter will be set to a zero-terminated UTF-8 string containing the name of the index or table used for the X-th loop.

SQLITE_SCANSTAT_EXPLAIN
: The "const char *" variable pointed to by the V parameter will be set to a zero-terminated UTF-8 string containing the EXPLAIN QUERY PLAN description for the X-th loop.

SQLITE_SCANSTAT_SELECT
: The "int" variable pointed to by the V parameter will be set to the "select-id" for the X-th loop. The select-id identifies which query or subquery the loop is part of. The main query has a select-id of zero. The select-id is the same value as is output in the first column of an EXPLAIN QUERY PLAN query.

Flags for the xShmLock VFS method

```
#define SQLITE_SHM_UNLOCK       1
#define SQLITE_SHM_LOCK         2
#define SQLITE_SHM_SHARED       4
#define SQLITE_SHM_EXCLUSIVE    8
```

These integer constants define the various locking operations allowed by the xShmLock method of sqlite3_io_methods. The following are the only legal combinations of flags to the xShmLock method:

- SQLITE_SHM_LOCK | SQLITE_SHM_SHARED
- SQLITE_SHM_LOCK | SQLITE_SHM_EXCLUSIVE
- SQLITE_SHM_UNLOCK | SQLITE_SHM_SHARED
- SQLITE_SHM_UNLOCK | SQLITE_SHM_EXCLUSIVE

When unlocking, the same SHARED or EXCLUSIVE flag must be supplied as was given on the corresponding lock.

The xShmLock method can transition between unlocked and SHARED or between unlocked and EXCLUSIVE. It cannot transition between SHARED and EXCLUSIVE.

Compile-Time Library Version Numbers

```
#define SQLITE_VERSION        "3.31.1"
#define SQLITE_VERSION_NUMBER 3031001
#define SQLITE_SOURCE_ID      "2020-01-27 19:55:54 3bfa9cc97da10598521b342961df8f5f68c7388fa117345eeb516eaa837bb4d6"
```

The SQLITE_VERSION C preprocessor macro in the sqlite3.h header evaluates to a string literal that is the SQLite version in the format "X.Y.Z" where X is the major version number (always 3 for SQLite3) and Y is the minor version number and Z is the release number. The SQLITE_VERSION_NUMBER C preprocessor macro resolves to an integer with the value (X*1000000 + Y*1000 + Z) where X, Y, and Z are the same numbers used in SQLITE_VERSION. The SQLITE_VERSION_NUMBER for any given release of SQLite will also be larger than the release from which it is derived. Either Y will be held constant and Z will be incremented or else Y will be incremented and Z will be reset to zero.

Since version 3.6.18 (2009-09-11), SQLite source code has been stored in the Fossil configuration management system. The SQLITE_SOURCE_ID macro evaluates to a string which identifies a particular check-in of SQLite within its configuration management system. The SQLITE_SOURCE_ID string contains the date and time of the check-in (UTC) and a SHA1 or SHA3-256 hash of the entire source tree. If the source code has been edited in any way since it was last checked in, then the last four hexadecimal digits of the hash may be modified.

See also: sqlite3_libversion(), sqlite3_libversion_number(), sqlite3_sourceid(), sqlite_version() and sqlite_source_id().

Constants Defining Special Destructor Behavior

```
typedef void (*sqlite3_destructor_type)(void*);
#define SQLITE_STATIC      ((sqlite3_destructor_type)0)
#define SQLITE_TRANSIENT   ((sqlite3_destructor_type)-1)
```

These are special values for the destructor that is passed in as the final argument to routines like sqlite3_result_blob(). If the destructor argument is SQLITE_STATIC, it means that the content pointer is constant and will never change. It does not need to be destroyed. The SQLITE_TRANSIENT value means that the content will likely change in the near future and that SQLite should make its own private copy of the content before returning.

The typedef is necessary to work around problems in certain C++ compilers.

Status Parameters

```
#define SQLITE_STATUS_MEMORY_USED          0
#define SQLITE_STATUS_PAGECACHE_USED       1
#define SQLITE_STATUS_PAGECACHE_OVERFLOW   2
#define SQLITE_STATUS_SCRATCH_USED         3  /* NOT USED */
#define SQLITE_STATUS_SCRATCH_OVERFLOW     4  /* NOT USED */
#define SQLITE_STATUS_MALLOC_SIZE          5
#define SQLITE_STATUS_PARSER_STACK         6
#define SQLITE_STATUS_PAGECACHE_SIZE       7
#define SQLITE_STATUS_SCRATCH_SIZE         8  /* NOT USED */
#define SQLITE_STATUS_MALLOC_COUNT         9
```

These integer constants designate various run-time status parameters that can be returned by sqlite3_status().

SQLITE_STATUS_MEMORY_USED
: This parameter is the current amount of memory checked out using sqlite3_malloc(), either directly or indirectly. The figure includes calls made to sqlite3_malloc() by the application and internal memory usage by the SQLite library. Auxiliary page-cache memory controlled by SQLITE_CONFIG_PAGECACHE is not included in this parameter. The amount returned is the sum of the allocation sizes as reported by the xSize method in sqlite3_mem_methods.

SQLITE_STATUS_MALLOC_SIZE
: This parameter records the largest memory allocation request handed to sqlite3_malloc() or sqlite3_realloc() (or their internal equivalents). Only the value returned in the *pHighwater parameter to sqlite3_status() is of interest. The value written into the *pCurrent parameter is undefined.

SQLITE_STATUS_MALLOC_COUNT
: This parameter records the number of separate memory allocations currently checked out.

SQLITE_STATUS_PAGECACHE_USED
: This parameter returns the number of pages used out of the pagecache memory allocator that was configured using SQLITE_CONFIG_PAGECACHE. The value returned is in pages, not in bytes.

SQLITE_STATUS_PAGECACHE_OVERFLOW
: This parameter returns the number of bytes of page cache allocation which could not be satisfied by the SQLITE_CONFIG_PAGECACHE buffer and where forced to overflow to sqlite3_malloc(). The returned value includes allocations that overflowed because they where too large (they were larger than the "sz" parameter to SQLITE_CONFIG_PAGECACHE) and allocations that overflowed because no space was left in the page cache.

SQLITE_STATUS_PAGECACHE_SIZE
: This parameter records the largest memory allocation request handed to the pagecache memory allocator. Only the value returned in the *pHighwater parameter to sqlite3_status() is of interest. The value written into the *pCurrent parameter is undefined.

SQLITE_STATUS_SCRATCH_USED
: No longer used.

SQLITE_STATUS_SCRATCH_OVERFLOW
: No longer used.

SQLITE_STATUS_SCRATCH_SIZE
: No longer used.

SQLITE_STATUS_PARSER_STACK
: The *pHighwater parameter records the deepest parser stack. The *pCurrent value is undefined. The *pHighwater value is only meaningful if SQLite is compiled with YYTRACKMAXSTACKDEPTH.

New status parameters may be added from time to time.

Synchronization Type Flags

```
#define SQLITE_SYNC_NORMAL        0x00002
#define SQLITE_SYNC_FULL          0x00003
#define SQLITE_SYNC_DATAONLY      0x00010
```

When SQLite invokes the xSync() method of an sqlite3_io_methods object it uses a combination of these integer values as the second argument.

When the SQLITE_SYNC_DATAONLY flag is used, it means that the sync operation only needs to flush data to mass storage. Inode information need not be flushed. If the lower four bits of the flag equal SQLITE_SYNC_NORMAL, that means to use normal fsync() semantics. If the lower four bits equal SQLITE_SYNC_FULL, that means to use Mac OS X style fullsync instead of fsync().

Do not confuse the SQLITE_SYNC_NORMAL and SQLITE_SYNC_FULL flags with the PRAGMA synchronous=NORMAL and PRAGMA synchronous=FULL settings. The synchronous pragma determines when calls to the xSync VFS method occur and applies uniformly across all platforms. The SQLITE_SYNC_NORMAL and SQLITE_SYNC_FULL flags determine how energetic or rigorous or forceful the sync operations are and only make a difference on Mac OSX for the default SQLite code. (Third-party VFS implementations might also make the distinction between SQLITE_SYNC_NORMAL and SQLITE_SYNC_FULL, but among the operating systems natively supported by SQLite, only Mac OSX cares about the difference.)

Testing Interface Operation Codes

```
#define SQLITE_TESTCTRL_FIRST                    5
#define SQLITE_TESTCTRL_PRNG_SAVE                5
#define SQLITE_TESTCTRL_PRNG_RESTORE             6
#define SQLITE_TESTCTRL_PRNG_RESET               7  /* NOT USED */
#define SQLITE_TESTCTRL_BITVEC_TEST              8
#define SQLITE_TESTCTRL_FAULT_INSTALL            9
#define SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS     10
#define SQLITE_TESTCTRL_PENDING_BYTE            11
#define SQLITE_TESTCTRL_ASSERT                  12
#define SQLITE_TESTCTRL_ALWAYS                  13
#define SQLITE_TESTCTRL_RESERVE                 14
#define SQLITE_TESTCTRL_OPTIMIZATIONS           15
#define SQLITE_TESTCTRL_ISKEYWORD               16  /* NOT USED */
#define SQLITE_TESTCTRL_SCRATCHMALLOC           17  /* NOT USED */
#define SQLITE_TESTCTRL_INTERNAL_FUNCTIONS      17
#define SQLITE_TESTCTRL_LOCALTIME_FAULT         18
#define SQLITE_TESTCTRL_EXPLAIN_STMT            19  /* NOT USED */
#define SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD    19
#define SQLITE_TESTCTRL_NEVER_CORRUPT           20
#define SQLITE_TESTCTRL_VDBE_COVERAGE           21
#define SQLITE_TESTCTRL_BYTEORDER               22
#define SQLITE_TESTCTRL_ISINIT                  23
#define SQLITE_TESTCTRL_SORTER_MMAP             24
#define SQLITE_TESTCTRL_IMPOSTER                25
#define SQLITE_TESTCTRL_PARSER_COVERAGE         26
#define SQLITE_TESTCTRL_RESULT_INTREAL          27
#define SQLITE_TESTCTRL_PRNG_SEED               28
#define SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS     29
#define SQLITE_TESTCTRL_LAST                    29  /* Largest TESTCTRL */
```

These constants are the valid operation code parameters used as the first argument to sqlite3_test_control().

These parameters and their meanings are subject to change without notice. These values are for testing purposes only. Applications should not use any of these parameters or the sqlite3_test_control() interface.

SQL Trace Event Codes

```
#define SQLITE_TRACE_STMT       0x01
#define SQLITE_TRACE_PROFILE    0x02
#define SQLITE_TRACE_ROW        0x04
#define SQLITE_TRACE_CLOSE      0x08
```

These constants identify classes of events that can be monitored using the sqlite3_trace_v2() tracing logic. The M argument to sqlite3_trace_v2(D,M,X,P) is an OR-ed combination of one or more of the following constants. The first argument to the trace callback is one of the following constants.

New tracing constants may be added in future releases.

A trace callback has four arguments: xCallback(T,C,P,X). The T argument is one of the integer type codes above. The C argument is a copy of the context pointer passed in as the fourth argument to sqlite3_trace_v2(). The P and X arguments are pointers whose meanings depend on T.

SQLITE_TRACE_STMT
: An SQLITE_TRACE_STMT callback is invoked when a prepared statement first begins running and possibly at other times during the execution of the prepared statement, such as at the start of each trigger subprogram. The P argument is a pointer to the prepared statement. The X argument is a pointer to a string which is the unexpanded SQL text of the prepared statement or an SQL comment that indicates the invocation of a trigger. The callback can compute the same text that would have been returned by the legacy sqlite3_trace() interface by using the X argument when X begins with "--" and invoking sqlite3_expanded_sql(P) otherwise.

SQLITE_TRACE_PROFILE
: An SQLITE_TRACE_PROFILE callback provides approximately the same information as is provided by the sqlite3_profile() callback. The P argument is a pointer to the prepared statement and the X argument points to a 64-bit integer which is the estimated of the number of nanosecond that the prepared statement took to run. The SQLITE_TRACE_PROFILE callback is invoked when the statement finishes.

SQLITE_TRACE_ROW
: An SQLITE_TRACE_ROW callback is invoked whenever a prepared statement generates a single row of result. The P argument is a pointer to the prepared statement and the X argument is unused.

SQLITE_TRACE_CLOSE
: An SQLITE_TRACE_CLOSE callback is invoked when a database connection closes. The P argument is a pointer to the database connection object and the X argument is unused.

Virtual Table Configuration Options

```
#define SQLITE_VTAB_CONSTRAINT_SUPPORT 1
#define SQLITE_VTAB_INNOCUOUS          2
#define SQLITE_VTAB_DIRECTONLY         3
```

These macros define the various options to the sqlite3_vtab_config() interface that virtual table implementations can use to customize and optimize their behavior.

SQLITE_VTAB_CONSTRAINT_SUPPORT
: Calls of the form sqlite3_vtab_config(db,SQLITE_VTAB_CONSTRAINT_SUPPORT,X) are supported, where X is an integer. If X is zero, then the virtual table whose xCreate or xConnect method invoked sqlite3_vtab_config() does not support constraints. In this configuration (which is the default) if a call to the xUpdate method returns SQLITE_CONSTRAINT, then the entire statement is rolled back as if OR ABORT had been specified as part of the users SQL statement, regardless of the actual ON CONFLICT mode specified.

  If X is non-zero, then the virtual table implementation guarantees that if xUpdate returns SQLITE_CONSTRAINT, it will do so before any modifications to internal or persistent data structures have been made. If the ON CONFLICT mode is ABORT, FAIL, IGNORE or ROLLBACK, SQLite is able to roll back a statement or database transaction, and abandon or continue processing the current SQL statement as appropriate. If the ON CONFLICT mode is REPLACE and the xUpdate method returns SQLITE_CONSTRAINT, SQLite handles this as if the ON CONFLICT mode had been ABORT.

  Virtual table implementations that are required to handle OR REPLACE must do so within the xUpdate method. If a call to the sqlite3_vtab_on_conflict() function indicates that the current ON CONFLICT policy is REPLACE, the virtual table implementation should silently replace the appropriate rows within the xUpdate callback and return SQLITE_OK. Or, if this is not possible, it may return SQLITE_CONSTRAINT, in which case SQLite falls back to OR ABORT constraint handling.

SQLITE_VTAB_DIRECTONLY
: Calls of the form sqlite3_vtab_config(db,SQLITE_VTAB_DIRECTONLY) from within the the xConnect or xCreate methods of a virtual table implmentation prohibits that virtual table from being used from within triggers and views.

SQLITE_VTAB_INNOCUOUS
: Calls of the form sqlite3_vtab_config(db,SQLITE_VTAB_INNOCUOUS) from within the the xConnect or xCreate methods of a virtual table implmentation identify that virtual table as being safe to use from within triggers and views. Conceptually, the SQLITE_VTAB_INNOCUOUS tag means that the virtual table can do no serious harm even if it is controlled by a malicious hacker. Developers should avoid setting the SQLITE_VTAB_INNOCUOUS flag unless absolutely necessary.

Win32 Directory Types

```
#define SQLITE_WIN32_DATA_DIRECTORY_TYPE  1
#define SQLITE_WIN32_TEMP_DIRECTORY_TYPE  2
```

These macros are only available on Windows. They define the allowed values for the type argument to the sqlite3_win32_set_directory interface.

Run-Time Limit Categories

```
#define SQLITE_LIMIT_LENGTH                    0
#define SQLITE_LIMIT_SQL_LENGTH                1
#define SQLITE_LIMIT_COLUMN                    2
#define SQLITE_LIMIT_EXPR_DEPTH                3
#define SQLITE_LIMIT_COMPOUND_SELECT           4
#define SQLITE_LIMIT_VDBE_OP                   5
#define SQLITE_LIMIT_FUNCTION_ARG              6
#define SQLITE_LIMIT_ATTACHED                  7
#define SQLITE_LIMIT_LIKE_PATTERN_LENGTH       8
#define SQLITE_LIMIT_VARIABLE_NUMBER           9
#define SQLITE_LIMIT_TRIGGER_DEPTH            10
#define SQLITE_LIMIT_WORKER_THREADS           11
```

These constants define various performance limits that can be lowered at run-time using sqlite3_limit(). The synopsis of the meanings of the various limits is shown below. Additional information is available at Limits in SQLite.

SQLITE_LIMIT_LENGTH
: The maximum size of any string or BLOB or table row, in bytes.

SQLITE_LIMIT_SQL_LENGTH
: The maximum length of an SQL statement, in bytes.

SQLITE_LIMIT_COLUMN
The maximum number of columns in a table definition or in the result set of a SELECT or the maximum number of columns in an index or in an ORDER BY or GROUP BY clause.

SQLITE_LIMIT_EXPR_DEPTH
The maximum depth of the parse tree on any expression.

SQLITE_LIMIT_COMPOUND_SELECT
The maximum number of terms in a compound SELECT statement.

SQLITE_LIMIT_VDBE_OP
The maximum number of instructions in a virtual machine program used to implement an SQL statement. If sqlite3_prepare_v2() or the equivalent tries to allocate space for more than this many opcodes in a single prepared statement, an SQLITE_NOMEM error is returned.

SQLITE_LIMIT_FUNCTION_ARG
The maximum number of arguments on a function.

SQLITE_LIMIT_ATTACHED
The maximum number of attached databases.

SQLITE_LIMIT_LIKE_PATTERN_LENGTH
The maximum length of the pattern argument to the LIKE or GLOB operators.

SQLITE_LIMIT_VARIABLE_NUMBER
The maximum index number of any parameter in an SQL statement.

SQLITE_LIMIT_TRIGGER_DEPTH
The maximum depth of recursion for triggers.

SQLITE_LIMIT_WORKER_THREADS
The maximum number of auxiliary worker threads that a single prepared statement may start.

Status Parameters for database connections

```
#define SQLITE_DBSTATUS_LOOKASIDE_USED       0
#define SQLITE_DBSTATUS_CACHE_USED           1
#define SQLITE_DBSTATUS_SCHEMA_USED          2
#define SQLITE_DBSTATUS_STMT_USED            3
#define SQLITE_DBSTATUS_LOOKASIDE_HIT        4
#define SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE  5
#define SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL  6
#define SQLITE_DBSTATUS_CACHE_HIT            7
#define SQLITE_DBSTATUS_CACHE_MISS           8
#define SQLITE_DBSTATUS_CACHE_WRITE          9
#define SQLITE_DBSTATUS_DEFERRED_FKS        10
#define SQLITE_DBSTATUS_CACHE_USED_SHARED   11
#define SQLITE_DBSTATUS_CACHE_SPILL         12
#define SQLITE_DBSTATUS_MAX                 12   /* Largest defined DBSTATUS */
```

These constants are the available integer "verbs" that can be passed as the second argument to the sqlite3_db_status() interface.

New verbs may be added in future releases of SQLite. Existing verbs might be discontinued. Applications should check the return code from sqlite3_db_status() to make sure that the call worked. The sqlite3_db_status() interface will return a non-zero error code if a discontinued or unsupported verb is invoked.

SQLITE_DBSTATUS_LOOKASIDE_USED
This parameter returns the number of lookaside memory slots currently checked out.

SQLITE_DBSTATUS_LOOKASIDE_HIT
This parameter returns the number of malloc attempts that were satisfied using lookaside memory. Only the high-water value is meaningful; the current value is always zero.

SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE
This parameter returns the number malloc attempts that might have been satisfied using lookaside memory but failed due to the amount of memory requested being larger than the lookaside slot size. Only the high-water value is meaningful; the current value is always zero.

SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL
This parameter returns the number malloc attempts that might have been satisfied using lookaside memory but failed due to all lookaside memory already being in use. Only the high-water value is meaningful; the current value is always zero.

SQLITE_DBSTATUS_CACHE_USED
This parameter returns the approximate number of bytes of heap memory used by all pager caches associated with the database connection. The highwater mark associated with SQLITE_DBSTATUS_CACHE_USED is always 0.

SQLITE_DBSTATUS_CACHE_USED_SHARED
This parameter is similar to DBSTATUS_CACHE_USED, except that if a pager cache is shared between two or more connections the bytes of heap memory used by that pager cache is divided evenly between the attached connections. In other words, if none of the pager caches associated with the database connection are shared, this request returns the same value as DBSTATUS_CACHE_USED. Or, if one or more or the pager caches are shared, the value returned by this call will be smaller than that returned by DBSTATUS_CACHE_USED. The highwater mark associated with SQLITE_DBSTATUS_CACHE_USED_SHARED is always 0.

SQLITE_DBSTATUS_SCHEMA_USED

This parameter returns the approximate number of bytes of heap memory used to store the schema for all databases associated with the connection - main, temp, and any ATTACH-ed databases. The full amount of memory used by the schemas is reported, even if the schema memory is shared with other database connections due to shared cache mode being enabled. The highwater mark associated with SQLITE_DBSTATUS_SCHEMA_USED is always 0.

SQLITE_DBSTATUS_STMT_USED
This parameter returns the approximate number of bytes of heap and lookaside memory used by all prepared statements associated with the database connection. The highwater mark associated with SQLITE_DBSTATUS_STMT_USED is always 0.

SQLITE_DBSTATUS_CACHE_HIT
This parameter returns the number of pager cache hits that have occurred. The highwater mark associated with SQLITE_DBSTATUS_CACHE_HIT is always 0.

SQLITE_DBSTATUS_CACHE_MISS
This parameter returns the number of pager cache misses that have occurred. The highwater mark associated with SQLITE_DBSTATUS_CACHE_MISS is always 0.

SQLITE_DBSTATUS_CACHE_WRITE
This parameter returns the number of dirty cache entries that have been written to disk. Specifically, the number of pages written to the wal file in wal mode databases, or the number of pages written to the database file in rollback mode databases. Any pages written as part of transaction rollback or database recovery operations are not included. If an IO or other error occurs while writing a page to disk, the effect on subsequent SQLITE_DBSTATUS_CACHE_WRITE requests is undefined. The highwater mark associated with SQLITE_DBSTATUS_CACHE_WRITE is always 0.

SQLITE_DBSTATUS_CACHE_SPILL
This parameter returns the number of dirty cache entries that have been written to disk in the middle of a transaction due to the page cache overflowing. Transactions are more efficient if they are written to disk all at once. When pages spill mid-transaction, that introduces additional overhead. This parameter can be used help identify inefficiencies that can be resolved by increasing the cache size.

SQLITE_DBSTATUS_DEFERRED_FKS
This parameter returns zero for the current value if and only if all foreign key constraints (deferred or immediate) have been resolved. The highwater mark is always 0.

Status Parameters for prepared statements

```
#define SQLITE_STMTSTATUS_FULLSCAN_STEP     1
#define SQLITE_STMTSTATUS_SORT              2
#define SQLITE_STMTSTATUS_AUTOINDEX         3
#define SQLITE_STMTSTATUS_VM_STEP           4
#define SQLITE_STMTSTATUS_REPREPARE         5
#define SQLITE_STMTSTATUS_RUN               6
#define SQLITE_STMTSTATUS_MEMUSED           99
```

These preprocessor macros define integer codes that name counter values associated with the sqlite3_stmt_status() interface. The meanings of the various counters are as follows:

SQLITE_STMTSTATUS_FULLSCAN_STEP
This is the number of times that SQLite has stepped forward in a table as part of a full table scan. Large numbers for this counter may indicate opportunities for performance improvement through careful use of indices.

SQLITE_STMTSTATUS_SORT
This is the number of sort operations that have occurred. A non-zero value in this counter may indicate an opportunity to improvement performance through careful use of indices.

SQLITE_STMTSTATUS_AUTOINDEX
This is the number of rows inserted into transient indices that were created automatically in order to help joins run faster. A non-zero value in this counter may indicate an opportunity to improvement performance by adding permanent indices that do not need to be reinitialized each time the statement is run.

SQLITE_STMTSTATUS_VM_STEP
This is the number of virtual machine operations executed by the prepared statement if that number is less than or equal to 2147483647. The number of virtual machine operations can be used as a proxy for the total work done by the prepared statement. If the number of virtual machine operations exceeds 2147483647 then the value returned by this statement status code is undefined.

SQLITE_STMTSTATUS_REPREPARE
This is the number of times that the prepare statement has been automatically regenerated due to schema changes or changes to bound parameters that might affect the query plan.

SQLITE_STMTSTATUS_RUN
This is the number of times that the prepared statement has been run. A single "run" for the purposes of this counter is one or more calls to sqlite3_step() followed by a call to sqlite3_reset(). The counter is incremented on the first sqlite3_step() call of each cycle.

SQLITE_STMTSTATUS_MEMUSED
This is the approximate number of bytes of heap memory used to store the prepared statement. This value is not actually a counter, and so the resetFlg parameter to sqlite3_stmt_status() is ignored when the opcode is SQLITE_STMTSTATUS_MEMUSED.

64-Bit Integer Types

```
#ifdef SQLITE_INT64_TYPE
  typedef SQLITE_INT64_TYPE sqlite_int64;
# ifdef SQLITE_UINT64_TYPE
    typedef SQLITE_UINT64_TYPE sqlite_uint64;
# else
    typedef unsigned SQLITE_INT64_TYPE sqlite_uint64;
# endif
#elif defined(_MSC_VER) || defined(__BORLANDC__)
  typedef __int64 sqlite_int64;
  typedef unsigned __int64 sqlite_uint64;
#else
  typedef long long int sqlite_int64;
  typedef unsigned long long int sqlite_uint64;
#endif
typedef sqlite_int64 sqlite3_int64;
typedef sqlite_uint64 sqlite3_uint64;
```

Because there is no cross-platform way to specify 64-bit integer types SQLite includes typedefs for 64-bit signed and unsigned integers.

The sqlite3_int64 and sqlite3_uint64 are the preferred type definitions. The sqlite_int64 and sqlite_uint64 types are supported for backwards compatibility only.

The sqlite3_int64 and sqlite_int64 types can store integer values between -9223372036854775808 and +9223372036854775807 inclusive. The sqlite3_uint64 and sqlite_uint64 types can store integer values between 0 and +18446744073709551615 inclusive.

Virtual Table Object

```
struct sqlite3_module {
  int iVersion;
  int (*xCreate)(sqlite3*, void *pAux,
               int argc, const char *const*argv,
               sqlite3_vtab **ppVTab, char**);
  int (*xConnect)(sqlite3*, void *pAux,
               int argc, const char *const*argv,
               sqlite3_vtab **ppVTab, char**);
  int (*xBestIndex)(sqlite3_vtab *pVTab, sqlite3_index_info*);
  int (*xDisconnect)(sqlite3_vtab *pVTab);
  int (*xDestroy)(sqlite3_vtab *pVTab);
  int (*xOpen)(sqlite3_vtab *pVTab, sqlite3_vtab_cursor **ppCursor);
  int (*xClose)(sqlite3_vtab_cursor*);
  int (*xFilter)(sqlite3_vtab_cursor*, int idxNum, const char *idxStr,
                int argc, sqlite3_value **argv);
  int (*xNext)(sqlite3_vtab_cursor*);
  int (*xEof)(sqlite3_vtab_cursor*);
  int (*xColumn)(sqlite3_vtab_cursor*, sqlite3_context*, int);
  int (*xRowid)(sqlite3_vtab_cursor*, sqlite3_int64 *pRowid);
  int (*xUpdate)(sqlite3_vtab *, int, sqlite3_value **, sqlite3_int64 *);
  int (*xBegin)(sqlite3_vtab *pVTab);
  int (*xSync)(sqlite3_vtab *pVTab);
  int (*xCommit)(sqlite3_vtab *pVTab);
  int (*xRollback)(sqlite3_vtab *pVTab);
  int (*xFindFunction)(sqlite3_vtab *pVtab, int nArg, const char *zName,
                       void (**pxFunc)(sqlite3_context*,int,sqlite3_value**),
                       void **ppArg);
  int (*xRename)(sqlite3_vtab *pVtab, const char *zNew);
  /* The methods above are in version 1 of the sqlite_module object. Those
  ** below are for version 2 and greater. */
  int (*xSavepoint)(sqlite3_vtab *pVTab, int);
  int (*xRelease)(sqlite3_vtab *pVTab, int);
  int (*xRollbackTo)(sqlite3_vtab *pVTab, int);
  /* The methods above are in versions 1 and 2 of the sqlite_module object.
  ** Those below are for version 3 and greater. */
  int (*xShadowName)(const char*);
};
```

This structure, sometimes called a "virtual table module", defines the implementation of a virtual table. This structure consists mostly of methods for the module.

A virtual table module is created by filling in a persistent instance of this structure and passing a pointer to that instance to sqlite3_create_module() or sqlite3_create_module_v2(). The registration remains valid until it is replaced by a different module or until the database connection closes. The content of this structure must not change while it is registered with any database connection.

Virtual Table Cursor Object

```
struct sqlite3_vtab_cursor {
  sqlite3_vtab *pVtab;      /* Virtual table of this cursor */
  /* Virtual table implementations will typically add additional fields */
};
```

Every virtual table module implementation uses a subclass of the following structure to describe cursors that point into the virtual table and are used to loop through the virtual table. Cursors are created using the xOpen method of the module and are destroyed by the xClose method. Cursors are used by the xFilter, xNext, xEof, xColumn, and xRowid methods of the module. Each module implementation will define the content of a cursor structure to suit its own needs.

This superclass exists in order to define fields of the cursor that are common to all implementations.

A Handle To An Open BLOB

```
typedef struct sqlite3_blob sqlite3_blob;
```

An instance of this object represents an open BLOB on which incremental BLOB I/O can be performed. Objects of this type are created by sqlite3_blob_open() and destroyed by sqlite3_blob_close(). The sqlite3_blob_read() and sqlite3_blob_write() interfaces can be used to read or write small subsections of the BLOB. The sqlite3_blob_bytes() interface returns the size of the BLOB in bytes.

Constructor: sqlite3_blob_open()

Destructor: sqlite3_blob_close()

Methods: sqlite3_blob_bytes(), sqlite3_blob_read(), sqlite3_blob_reopen(), sqlite3_blob_write()

Database Connection Handle

```
typedef struct sqlite3 sqlite3;
```

Each open SQLite database is represented by a pointer to an instance of the opaque structure named "sqlite3". It is useful to think of an sqlite3 pointer as an object. The sqlite3_open(), sqlite3_open16(), and sqlite3_open_v2() interfaces are its constructors, and sqlite3_close() and sqlite3_close_v2() are its destructors. There are many other interfaces (such as sqlite3_prepare_v2(), sqlite3_create_function(), and sqlite3_busy_timeout() to name but three) that are methods on an sqlite3 object.

Constructors: sqlite3_open(), sqlite3_open16(), sqlite3_open_v2()

Destructors: sqlite3_close(), sqlite3_close_v2()

Methods:

- sqlite3_blob_open
- sqlite3_busy_handler
- sqlite3_busy_timeout
- sqlite3_changes
- sqlite3_collation_needed
- sqlite3_collation_needed16
- sqlite3_commit_hook
- sqlite3_create_collation
- sqlite3_create_collation16
- sqlite3_create_collation_v2
- sqlite3_create_function
- sqlite3_create_function16
- sqlite3_create_function_v2
- sqlite3_create_module
- sqlite3_create_module_v2
- sqlite3_create_window_function
- sqlite3_db_config
- sqlite3_db_filename
- sqlite3_db_mutex
- sqlite3_db_readonly
- sqlite3_db_release_memory
- sqlite3_db_status
- sqlite3_drop_modules
- sqlite3_enable_load_extension
- sqlite3_errcode
- sqlite3_errmsg
- sqlite3_errmsg16
- sqlite3_errstr
- sqlite3_exec
- sqlite3_extended_errcode
- sqlite3_extended_result_codes
- sqlite3_file_control
- sqlite3_free_table
- sqlite3_get_autocommit
- sqlite3_get_table
- sqlite3_interrupt
- sqlite3_last_insert_rowid
- sqlite3_limit
- sqlite3_load_extension
- sqlite3_next_stmt
- sqlite3_overload_function
- sqlite3_prepare
- sqlite3_prepare16
- sqlite3_prepare16_v2
- sqlite3_prepare16_v3
- sqlite3_prepare_v2
- sqlite3_prepare_v3
- sqlite3_profile
- sqlite3_progress_handler
- sqlite3_rollback_hook
- sqlite3_set_authorizer
- sqlite3_set_last_insert_rowid
- sqlite3_table_column_metadata
- sqlite3_total_changes
- sqlite3_trace
- sqlite3_trace_v2
- sqlite3_unlock_notify
- sqlite3_update_hook
- sqlite3_wal_autocheckpoint
- sqlite3_wal_checkpoint
- sqlite3_wal_checkpoint_v2
- sqlite3_wal_hook

Dynamic String Object

```
typedef struct sqlite3_str sqlite3_str;
```

An instance of the sqlite3_str object contains a dynamically-sized string under construction.

The lifecycle of an sqlite3_str object is as follows:

1. The sqlite3_str object is created using sqlite3_str_new().
2. Text is appended to the sqlite3_str object using various methods, such as sqlite3_str_appendf().
3. The sqlite3_str object is destroyed and the string it created is returned using the sqlite3_str_finish() interface.

Constructor: sqlite3_str_new()

Destructor: sqlite3_str_finish()

Methods:

- sqlite3_str_append
- sqlite3_str_appendall
- sqlite3_str_appendchar
- sqlite3_str_appendf
- sqlite3_str_errcode
- sqlite3_str_length
- sqlite3_str_reset
- sqlite3_str_value
- sqlite3_str_vappendf

Application Defined Page Cache.

```
typedef struct sqlite3_pcache_methods2 sqlite3_pcache_methods2;
struct sqlite3_pcache_methods2 {
  int iVersion;
  void *pArg;
  int (*xInit)(void*);
  void (*xShutdown)(void*);
  sqlite3_pcache *(*xCreate)(int szPage, int szExtra, int bPurgeable);
  void (*xCachesize)(sqlite3_pcache*, int nCachesize);
  int (*xPagecount)(sqlite3_pcache*);
  sqlite3_pcache_page *(*xFetch)(sqlite3_pcache*, unsigned key, int createFlag);
  void (*xUnpin)(sqlite3_pcache*, sqlite3_pcache_page*, int discard);
  void (*xRekey)(sqlite3_pcache*, sqlite3_pcache_page*,
      unsigned oldKey, unsigned newKey);
  void (*xTruncate)(sqlite3_pcache*, unsigned iLimit);
```

```
  void (*xDestroy)(sqlite3_pcache*);
  void (*xShrink)(sqlite3_pcache*);
};
```

The sqlite3_config(SQLITE_CONFIG_PCACHE2, ...) interface can register an alternative page cache implementation by passing in an instance of the sqlite3_pcache_methods2 structure. In many applications, most of the heap memory allocated by SQLite is used for the page cache. By implementing a custom page cache using this API, an application can better control the amount of memory consumed by SQLite, the way in which that memory is allocated and released, and the policies used to determine exactly which parts of a database file are cached and for how long.

The alternative page cache mechanism is an extreme measure that is only needed by the most demanding applications. The built-in page cache is recommended for most uses.

The contents of the sqlite3_pcache_methods2 structure are copied to an internal buffer by SQLite within the call to sqlite3_config. Hence the application may discard the parameter after the call to sqlite3_config() returns.

The xInit() method is called once for each effective call to sqlite3_initialize() (usually only once during the lifetime of the process). The xInit() method is passed a copy of the sqlite3_pcache_methods2.pArg value. The intent of the xInit() method is to set up global data structures required by the custom page cache implementation. If the xInit() method is NULL, then the built-in default page cache is used instead of the application defined page cache.

The xShutdown() method is called by sqlite3_shutdown(). It can be used to clean up any outstanding resources before process shutdown, if required. The xShutdown() method may be NULL.

SQLite automatically serializes calls to the xInit method, so the xInit method need not be threadsafe. The xShutdown method is only called from sqlite3_shutdown() so it does not need to be threadsafe either. All other methods must be threadsafe in multithreaded applications.

SQLite will never invoke xInit() more than once without an intervening call to xShutdown().

SQLite invokes the xCreate() method to construct a new cache instance. SQLite will typically create one cache instance for each open database file, though this is not guaranteed. The first parameter, szPage, is the size in bytes of the pages that must be allocated by the cache. szPage will always a power of two. The second parameter szExtra is a number of bytes of extra storage associated with each page cache entry. The szExtra parameter will a number less than 250. SQLite will use the extra szExtra bytes on each page to store metadata about the underlying database page on disk. The value passed into szExtra depends on the SQLite version, the target platform, and how SQLite was compiled. The third argument to xCreate(), bPurgeable, is true if the cache being created will be used to cache database pages of a file stored on disk, or false if it is used for an in-memory database. The cache implementation does not have to do anything special based with the value of bPurgeable; it is purely advisory. On a cache where bPurgeable is false, SQLite will never invoke xUnpin() except to deliberately delete a page. In other words, calls to xUnpin() on a cache with bPurgeable set to false will always have the "discard" flag set to true. Hence, a cache created with bPurgeable false will never contain any unpinned pages.

The xCachesize() method may be called at any time by SQLite to set the suggested maximum cache-size (number of pages stored by) the cache instance passed as the first argument. This is the value configured using the SQLite "PRAGMA cache_size" command. As with the bPurgeable parameter, the implementation is not required to do anything with this value; it is advisory only.

The xPagecount() method must return the number of pages currently stored in the cache, both pinned and unpinned.

The xFetch() method locates a page in the cache and returns a pointer to an sqlite3_pcache_page object associated with that page, or a NULL pointer. The pBuf element of the returned sqlite3_pcache_page object will be a pointer to a buffer of szPage bytes used to store the content of a single database page. The pExtra element of sqlite3_pcache_page will be a pointer to the szExtra bytes of extra storage that SQLite has requested for each entry in the page cache.

The page to be fetched is determined by the key. The minimum key value is 1. After it has been retrieved using xFetch, the page is considered to be "pinned".

If the requested page is already in the page cache, then the page cache implementation must return a pointer to the page buffer with its content intact. If the requested page is not already in the cache, then the cache implementation should use the value of the createFlag parameter to help it determined what action to take:

createFlag	Behavior when page is not already in cache
0	Do not allocate a new page. Return NULL.
1	Allocate a new page if it easy and convenient to do so. Otherwise return NULL.
2	Make every effort to allocate a new page. Only return NULL if allocating a new page is effectively impossible.

SQLite will normally invoke xFetch() with a createFlag of 0 or 1. SQLite will only use a createFlag of 2 after a prior call with a createFlag of 1 failed. In between the xFetch() calls, SQLite may attempt to unpin one or more cache pages by spilling the content of pinned pages to disk and synching the operating system disk cache.

xUnpin() is called by SQLite with a pointer to a currently pinned page as its second argument. If the third parameter, discard, is non-zero, then the page must be evicted from the cache. If the discard parameter is zero, then the page may be discarded or retained at the discretion of page cache implementation. The page cache implementation may choose to evict unpinned pages at any time.

The cache must not perform any reference counting. A single call to xUnpin() unpins the page regardless of the number of prior calls to xFetch().

The xRekey() method is used to change the key value associated with the page passed as the second argument. If the cache previously contains an entry associated with newKey, it must be discarded. Any prior cache entry associated with newKey is guaranteed not to be pinned.

When SQLite calls the xTruncate() method, the cache must discard all existing cache entries with page numbers (keys) greater than or equal to the value of the iLimit parameter passed to xTruncate(). If any of these pages are pinned, they are implicitly

unpinned, meaning that they can be safely discarded.

The xDestroy() method is used to delete a cache allocated by xCreate(). All resources associated with the specified cache should be freed. After calling the xDestroy() method, SQLite considers the sqlite3_pcache* handle invalid, and will not use it with any other sqlite3_pcache_methods2 functions.

SQLite invokes the xShrink() method when it wants the page cache to free up as much of heap memory as possible. The page cache implementation is not obligated to free any memory, but well-behaved implementations should do their best.

Prepared Statement Object

```
typedef struct sqlite3_stmt sqlite3_stmt;
```

An instance of this object represents a single SQL statement that has been compiled into binary form and is ready to be evaluated.

Think of each SQL statement as a separate computer program. The original SQL text is source code. A prepared statement object is the compiled object code. All SQL must be converted into a prepared statement before it can be run.

The life-cycle of a prepared statement object usually goes like this:

1. Create the prepared statement object using sqlite3_prepare_v2().
2. Bind values to parameters using the sqlite3_bind_*() interfaces.
3. Run the SQL by calling sqlite3_step() one or more times.
4. Reset the prepared statement using sqlite3_reset() then go back to step 2. Do this zero or more times.
5. Destroy the object using sqlite3_finalize().

Constructors:

- sqlite3_prepare
- sqlite3_prepare16
- sqlite3_prepare16_v2
- sqlite3_prepare16_v3
- sqlite3_prepare_v2
- sqlite3_prepare_v3

Destructor: sqlite3_finalize()

Methods:

- sqlite3_bind_blob
- sqlite3_bind_blob64
- sqlite3_bind_double
- sqlite3_bind_int
- sqlite3_bind_int64
- sqlite3_bind_null
- sqlite3_bind_parameter_count
- sqlite3_bind_parameter_index
- sqlite3_bind_parameter_name
- sqlite3_bind_pointer
- sqlite3_bind_text
- sqlite3_bind_text16
- sqlite3_bind_text64
- sqlite3_bind_value
- sqlite3_bind_zeroblob
- sqlite3_bind_zeroblob64
- sqlite3_clear_bindings
- sqlite3_column_blob
- sqlite3_column_bytes
- sqlite3_column_bytes16
- sqlite3_column_count
- sqlite3_column_database_name
- sqlite3_column_database_name16
- sqlite3_column_decltype
- sqlite3_column_decltype16
- sqlite3_column_double
- sqlite3_column_int
- sqlite3_column_int64
- sqlite3_column_name
- sqlite3_column_name16
- sqlite3_column_origin_name
- sqlite3_column_origin_name16
- sqlite3_column_table_name
- sqlite3_column_table_name16
- sqlite3_column_text
- sqlite3_column_text16
- sqlite3_column_type
- sqlite3_column_value
- sqlite3_data_count
- sqlite3_db_handle
- sqlite3_expanded_sql
- sqlite3_normalized_sql
- sqlite3_reset
- sqlite3_sql
- sqlite3_step
- sqlite3_stmt_busy
- sqlite3_stmt_isexplain
- sqlite3_stmt_readonly
- sqlite3_stmt_scanstatus
- sqlite3_stmt_scanstatus_r
- sqlite3_stmt_status

Dynamically Typed Value Object

```
typedef struct sqlite3_value sqlite3_value;
```

SQLite uses the sqlite3_value object to represent all values that can be stored in a database table. SQLite uses dynamic typing for the values it stores. Values stored in sqlite3_value objects can be integers, floating point values, strings, BLOBs, or NULL.

An sqlite3_value object may be either "protected" or "unprotected". Some interfaces require a protected sqlite3_value. Other interfaces will accept either a protected or an unprotected sqlite3_value. Every interface that accepts sqlite3_value arguments specifies whether or not it requires a protected sqlite3_value. The sqlite3_value_dup() interface can be used to construct a new protected sqlite3_value from an unprotected sqlite3_value.

The terms "protected" and "unprotected" refer to whether or not a mutex is held. An internal mutex is held for a protected sqlite3_value object but no mutex is held for an unprotected sqlite3_value object. If SQLite is compiled to be single-threaded (with SQLITE_THREADSAFE=0 and with sqlite3_threadsafe() returning 0) or if SQLite is run in one of reduced mutex modes SQLITE_CONFIG_SINGLETHREAD or SQLITE_CONFIG_MULTITHREAD then there is no distinction between protected and unprotected sqlite3_value objects and they can be used interchangeably. However, for maximum code portability it is recommended that applications still make the distinction between protected and unprotected sqlite3_value objects even when not strictly required.

The sqlite3_value objects that are passed as parameters into the implementation of application-defined SQL functions are protected. The sqlite3_value object returned by sqlite3_column_value() is unprotected. Unprotected sqlite3_value objects may only be used as arguments to sqlite3_result_value(), sqlite3_bind_value(), and sqlite3_value_dup(). The sqlite3_value_type() family of interfaces require protected sqlite3_value objects.

Methods:

- sqlite3_value_blob
- sqlite3_value_bytes
- sqlite3_value_bytes16
- sqlite3_value_free
- sqlite3_value_frombind
- sqlite3_value_int
- sqlite3_value_numeric_type
- sqlite3_value_pointer
- sqlite3_value_subtype
- sqlite3_value_text16be
- sqlite3_value_text16le
- sqlite3_value_type

- sqlite3_value_double
- sqlite3_value_dup
- sqlite3_value_int64
- sqlite3_value_nochange
- sqlite3_value_text
- sqlite3_value_text16

Deprecated Functions

```
#ifndef SQLITE_OMIT_DEPRECATED
int sqlite3_aggregate_count(sqlite3_context*);
int sqlite3_expired(sqlite3_stmt*);
int sqlite3_transfer_bindings(sqlite3_stmt*, sqlite3_stmt*);
int sqlite3_global_recover(void);
void sqlite3_thread_cleanup(void);
int sqlite3_memory_alarm(void(*)(void*,sqlite3_int64,int),
                         void*,sqlite3_int64);
#endif
```

These functions are deprecated. In order to maintain backwards compatibility with older code, these functions continue to be supported. However, new applications should avoid the use of these functions. To encourage programmers to avoid these functions, we will not explain what they do.

Online Backup API.

```
sqlite3_backup *sqlite3_backup_init(
  sqlite3 *pDest,                        /* Destination database handle */
  const char *zDestName,                 /* Destination database name */
  sqlite3 *pSource,                      /* Source database handle */
  const char *zSourceName                /* Source database name */
);
int sqlite3_backup_step(sqlite3_backup *p, int nPage);
int sqlite3_backup_finish(sqlite3_backup *p);
int sqlite3_backup_remaining(sqlite3_backup *p);
int sqlite3_backup_pagecount(sqlite3_backup *p);
```

The backup API copies the content of one database into another. It is useful either for creating backups of databases or for copying in-memory databases to or from persistent files.

See Also: Using the SQLite Online Backup API

SQLite holds a write transaction open on the destination database file for the duration of the backup operation. The source database is read-locked only while it is being read; it is not locked continuously for the entire backup operation. Thus, the backup may be performed on a live source database without preventing other database connections from reading or writing to the source database while the backup is underway.

To perform a backup operation:

1. **sqlite3_backup_init()** is called once to initialize the backup,
2. **sqlite3_backup_step()** is called one or more times to transfer the data between the two databases, and finally
3. **sqlite3_backup_finish()** is called to release all resources associated with the backup operation.

There should be exactly one call to sqlite3_backup_finish() for each successful call to sqlite3_backup_init().

sqlite3_backup_init()

The D and N arguments to sqlite3_backup_init(D,N,S,M) are the database connection associated with the destination database and the database name, respectively. The database name is "main" for the main database, "temp" for the temporary database, or the name specified after the AS keyword in an ATTACH statement for an attached database. The S and M arguments passed to sqlite3_backup_init(D,N,S,M) identify the database connection and database name of the source database, respectively. The source and destination database connections (parameters S and D) must be different or else sqlite3_backup_init(D,N,S,M) will fail with an error.

A call to sqlite3_backup_init() will fail, returning NULL, if there is already a read or read-write transaction open on the destination database.

If an error occurs within sqlite3_backup_init(D,N,S,M), then NULL is returned and an error code and error message are stored in the destination database connection D. The error code and message for the failed call to sqlite3_backup_init() can be retrieved using the sqlite3_errcode(), sqlite3_errmsg(), and/or sqlite3_errmsg16() functions. A successful call to sqlite3_backup_init() returns a pointer to an sqlite3_backup object. The sqlite3_backup object may be used with the sqlite3_backup_step() and sqlite3_backup_finish() functions to perform the specified backup operation.

sqlite3_backup_step()

Function sqlite3_backup_step(B,N) will copy up to N pages between the source and destination databases specified by sqlite3_backup object B. If N is negative, all remaining source pages are copied. If sqlite3_backup_step(B,N) successfully copies N pages and there are still more pages to be copied, then the function returns SQLITE_OK. If sqlite3_backup_step(B,N) successfully finishes copying all pages from source to destination, then it returns SQLITE_DONE. If an error occurs while running sqlite3_backup_step(B,N), then an error code is returned. As well as SQLITE_OK and SQLITE_DONE, a call to sqlite3_backup_step() may return SQLITE_READONLY, SQLITE_NOMEM, SQLITE_BUSY, SQLITE_LOCKED, or an SQLITE_IOERR_XXX extended error code.

The sqlite3_backup_step() might return SQLITE_READONLY if

1. the destination database was opened read-only, or
2. the destination database is using write-ahead-log journaling and the destination and source page sizes differ, or
3. the destination database is an in-memory database and the destination and source page sizes differ.

If sqlite3_backup_step() cannot obtain a required file-system lock, then the busy-handler function is invoked (if one is specified). If the busy-handler returns non-zero before the lock is available, then SQLITE_BUSY is returned to the caller. In this case the call to sqlite3_backup_step() can be retried later. If the source database connection is being used to write to the source database when sqlite3_backup_step() is called, then SQLITE_LOCKED is returned immediately. Again, in this case the call to sqlite3_backup_step() can be retried later on. If SQLITE_IOERR_XXX, SQLITE_NOMEM, or SQLITE_READONLY is returned, then there is no point in retrying the call to sqlite3_backup_step(). These errors are considered fatal. The application must accept that the backup operation has failed and pass the backup operation handle to the sqlite3_backup_finish() to release associated resources.

The first call to sqlite3_backup_step() obtains an exclusive lock on the destination file. The exclusive lock is not released until either sqlite3_backup_finish() is called or the backup operation is complete and sqlite3_backup_step() returns SQLITE_DONE. Every call to sqlite3_backup_step() obtains a shared lock on the source database that lasts for the duration of the sqlite3_backup_step() call. Because the source database is not locked between calls to sqlite3_backup_step(), the source database may be modified mid-way through the backup process. If the source database is modified by an external process or via a database connection other than the one being used by the backup operation, then the backup will be automatically restarted by the next call to sqlite3_backup_step(). If the source database is modified by the using the same database connection as is used by the backup operation, then the backup database is automatically updated at the same time.

sqlite3_backup_finish()

When sqlite3_backup_step() has returned SQLITE_DONE, or when the application wishes to abandon the backup operation, the application should destroy the sqlite3_backup by passing it to sqlite3_backup_finish(). The sqlite3_backup_finish() interfaces releases all resources associated with the sqlite3_backup object. If sqlite3_backup_step() has not yet returned SQLITE_DONE, then any active write-transaction on the destination database is rolled back. The sqlite3_backup object is invalid and may not be used following a call to sqlite3_backup_finish().

The value returned by sqlite3_backup_finish is SQLITE_OK if no sqlite3_backup_step() errors occurred, regardless or whether or not sqlite3_backup_step() completed. If an out-of-memory condition or IO error occurred during any prior sqlite3_backup_step() call on the same sqlite3_backup object, then sqlite3_backup_finish() returns the corresponding error code.

A return of SQLITE_BUSY or SQLITE_LOCKED from sqlite3_backup_step() is not a permanent error and does not affect the return value of sqlite3_backup_finish().

sqlite3_backup_remaining() and sqlite3_backup_pagecount()

The sqlite3_backup_remaining() routine returns the number of pages still to be backed up at the conclusion of the most recent sqlite3_backup_step(). The sqlite3_backup_pagecount() routine returns the total number of pages in the source database at the conclusion of the most recent sqlite3_backup_step(). The values returned by these functions are only updated by sqlite3_backup_step(). If the source database is modified in a way that changes the size of the source database or the number of pages remaining, those changes are not reflected in the output of sqlite3_backup_pagecount() and sqlite3_backup_remaining() until after the next sqlite3_backup_step().

Concurrent Usage of Database Handles

The source database connection may be used by the application for other purposes while a backup operation is underway or being initialized. If SQLite is compiled and configured to support threadsafe database connections, then the source database connection may be used concurrently from within other threads.

However, the application must guarantee that the destination database connection is not passed to any other API (by any thread) after sqlite3_backup_init() is called and before the corresponding call to sqlite3_backup_finish(). SQLite does not currently check to see if the application incorrectly accesses the destination database connection and so no error code is reported, but the operations may malfunction nevertheless. Use of the destination database connection while a backup is in progress might also also cause a mutex deadlock.

If running in shared cache mode, the application must guarantee that the shared cache used by the destination database is not accessed while the backup is running. In practice this means that the application must guarantee that the disk file being backed up to is not accessed by any connection within the process, not just the specific connection that was passed to sqlite3_backup_init().

The sqlite3_backup object itself is partially threadsafe. Multiple threads may safely make multiple concurrent calls to sqlite3_backup_step(). However, the sqlite3_backup_remaining() and sqlite3_backup_pagecount() APIs are not strictly speaking threadsafe. If they are invoked at the same time as another thread is invoking sqlite3_backup_step() it is possible that they return invalid values.

Closing A Database Connection

```
int sqlite3_close(sqlite3*);
int sqlite3_close_v2(sqlite3*);
```

The sqlite3_close() and sqlite3_close_v2() routines are destructors for the sqlite3 object. Calls to sqlite3_close() and sqlite3_close_v2() return SQLITE_OK if the sqlite3 object is successfully destroyed and all associated resources are deallocated.

If the database connection is associated with unfinalized prepared statements or unfinished sqlite3_backup objects then sqlite3_close() will leave the database connection open and return SQLITE_BUSY. If sqlite3_close_v2() is called with unfinalized prepared statements and/or unfinished sqlite3_backups, then the database connection becomes an unusable "zombie" which will automatically be deallocated when the last prepared statement is finalized or the last sqlite3_backup is finished. The sqlite3_close_v2() interface is intended for use with host languages that are garbage collected, and where the order in which destructors are called is arbitrary.

Applications should finalize all prepared statements, close all BLOB handles, and finish all sqlite3_backup objects associated with the sqlite3 object prior to attempting to close the object. If sqlite3_close_v2() is called on a database connection that still has outstanding prepared statements, BLOB handles, and/or sqlite3_backup objects then it returns SQLITE_OK and the deallocation of resources is deferred until all prepared statements, BLOB handles, and sqlite3_backup objects are also destroyed.

If an sqlite3 object is destroyed while a transaction is open, the transaction is automatically rolled back.

The C parameter to sqlite3_close(C) and sqlite3_close_v2(C) must be either a NULL pointer or an sqlite3 object pointer obtained from sqlite3_open(), sqlite3_open16(), or sqlite3_open_v2(), and not previously closed. Calling sqlite3_close() or sqlite3_close_v2() with a NULL pointer argument is a harmless no-op.

Collation Needed Callbacks

```
int sqlite3_collation_needed(
  sqlite3*,
  void*,
  void(*)(void*,sqlite3*,int eTextRep,const char*)
);
int sqlite3_collation_needed16(
  sqlite3*,
  void*,
  void(*)(void*,sqlite3*,int eTextRep,const void*)
);
```

To avoid having to register all collation sequences before a database can be used, a single callback function may be registered with the database connection to be invoked whenever an undefined collation sequence is required.

If the function is registered using the sqlite3_collation_needed() API, then it is passed the names of undefined collation sequences as strings encoded in UTF-8. If sqlite3_collation_needed16() is used, the names are passed as UTF-16 in machine native byte order. A call to either function replaces the existing collation-needed callback.

When the callback is invoked, the first argument passed is a copy of the second argument to sqlite3_collation_needed() or sqlite3_collation_needed16(). The second argument is the database connection. The third argument is one of SQLITE_UTF8, SQLITE_UTF16BE, or SQLITE_UTF16LE, indicating the most desirable form of the collation sequence function required. The fourth parameter is the name of the required collation sequence.

The callback function should register the desired collation using sqlite3_create_collation(), sqlite3_create_collation16(), or sqlite3_create_collation_v2().

Source Of Data In A Query Result

```
const char *sqlite3_column_database_name(sqlite3_stmt*,int);
const void *sqlite3_column_database_name16(sqlite3_stmt*,int);
const char *sqlite3_column_table_name(sqlite3_stmt*,int);
const void *sqlite3_column_table_name16(sqlite3_stmt*,int);
const char *sqlite3_column_origin_name(sqlite3_stmt*,int);
const void *sqlite3_column_origin_name16(sqlite3_stmt*,int);
```

These routines provide a means to determine the database, table, and table column that is the origin of a particular result column in SELECT statement. The name of the database or table or column can be returned as either a UTF-8 or UTF-16 string. The _database_ routines return the database name, the _table_ routines return the table name, and the origin_ routines return the column name. The returned string is valid until the prepared statement is destroyed using sqlite3_finalize() or until the statement is automatically reprepared by the first call to sqlite3_step() for a particular run or until the same information is requested again in a different encoding.

The names returned are the original un-aliased names of the database, table, and column.

The first argument to these interfaces is a prepared statement. These functions return information about the Nth result column returned by the statement, where N is the second function argument. The left-most column is column 0 for these routines.

If the Nth column returned by the statement is an expression or subquery and is not a column value, then all of these functions return NULL. These routines might also return NULL if a memory allocation error occurs. Otherwise, they return the name of the attached database, table, or column that query result column was extracted from.

As with all other SQLite APIs, those whose names end with "16" return UTF-16 encoded strings and the other functions return UTF-8.

These APIs are only available if the library was compiled with the SQLITE_ENABLE_COLUMN_METADATA C-preprocessor symbol.

If two or more threads call one or more column metadata interfaces for the same prepared statement and result column at the same time then the results are undefined.

Declared Datatype Of A Query Result

```
const char *sqlite3_column_decltype(sqlite3_stmt*,int);
const void *sqlite3_column_decltype16(sqlite3_stmt*,int);
```

The first parameter is a prepared statement. If this statement is a SELECT statement and the Nth column of the returned result set of that SELECT is a table column (not an expression or subquery) then the declared type of the table column is returned. If the Nth column of the result set is an expression or subquery, then a NULL pointer is returned. The returned string is always UTF-8 encoded.

For example, given the database schema:

CREATE TABLE t1(c1 VARIANT);

and the following statement to be compiled:

SELECT c1 + 1, c1 FROM t1;

this routine would return the string "VARIANT" for the second result column (i==1), and a NULL pointer for the first result column (i==0).

SQLite uses dynamic run-time typing. So just because a column is declared to contain a particular type does not mean that the data stored in that column is of the declared type. SQLite is strongly typed, but the typing is dynamic not static. Type is associated with individual values, not with the containers used to hold those values.

Column Names In A Result Set

```
const char *sqlite3_column_name(sqlite3_stmt*, int N);
const void *sqlite3_column_name16(sqlite3_stmt*, int N);
```

These routines return the name assigned to a particular column in the result set of a SELECT statement. The sqlite3_column_name() interface returns a pointer to a zero-terminated UTF-8 string and sqlite3_column_name16() returns a pointer to a zero-terminated UTF-16 string. The first parameter is the prepared statement that implements the SELECT statement. The second parameter is the column number. The leftmost column is number 0.

The returned string pointer is valid until either the prepared statement is destroyed by sqlite3_finalize() or until the statement is automatically reprepared by the first call to sqlite3_step() for a particular run or until the next call to sqlite3_column_name() or sqlite3_column_name16() on the same column.

If sqlite3_malloc() fails during the processing of either routine (for example during a conversion from UTF-8 to UTF-16) then a NULL pointer is returned.

The name of a result column is the value of the "AS" clause for that column, if there is an AS clause. If there is no AS clause then the name of the column is unspecified and may change from one release of SQLite to the next.

Commit And Rollback Notification Callbacks

```
void *sqlite3_commit_hook(sqlite3*, int(*)(void*), void*);
void *sqlite3_rollback_hook(sqlite3*, void(*)(void *), void*);
```

The sqlite3_commit_hook() interface registers a callback function to be invoked whenever a transaction is committed. Any callback set by a previous call to sqlite3_commit_hook() for the same database connection is overridden. The sqlite3_rollback_hook() interface registers a callback function to be invoked whenever a transaction is rolled back. Any callback set by a previous call to sqlite3_rollback_hook() for the same database connection is overridden. The pArg argument is passed through to the callback. If the callback on a commit hook function returns non-zero, then the commit is converted into a rollback.

The sqlite3_commit_hook(D,C,P) and sqlite3_rollback_hook(D,C,P) functions return the P argument from the previous call of the same function on the same database connection D, or NULL for the first call for each function on D.

The commit and rollback hook callbacks are not reentrant. The callback implementation must not do anything that will modify the database connection that invoked the callback. Any actions to modify the database connection must be deferred until after the completion of the sqlite3_step() call that triggered the commit or rollback hook in the first place. Note that running any other SQL statements, including SELECT statements, or merely calling sqlite3_prepare_v2() and sqlite3_step() will modify the database connections for the meaning of "modify" in this paragraph.

Registering a NULL function disables the callback.

When the commit hook callback routine returns zero, the COMMIT operation is allowed to continue normally. If the commit hook returns non-zero, then the COMMIT is converted into a ROLLBACK. The rollback hook is invoked on a rollback that results from a commit hook returning non-zero, just as it would be with any other rollback.

For the purposes of this API, a transaction is said to have been rolled back if an explicit "ROLLBACK" statement is executed, or an error or constraint causes an implicit rollback to occur. The rollback callback is not invoked if a transaction is automatically rolled back because the database connection is closed.

See also the sqlite3_update_hook() interface.

Run-Time Library Compilation Options Diagnostics

```
#ifndef SQLITE_OMIT_COMPILEOPTION_DIAGS
int sqlite3_compileoption_used(const char *zOptName);
const char *sqlite3_compileoption_get(int N);
#else
# define sqlite3_compileoption_used(X) 0
# define sqlite3_compileoption_get(X)  ((void*)0)
#endif
```

The sqlite3_compileoption_used() function returns 0 or 1 indicating whether the specified option was defined at compile time. The SQLITE_ prefix may be omitted from the option name passed to sqlite3_compileoption_used().

The sqlite3_compileoption_get() function allows iterating over the list of options that were defined at compile time by returning the N-th compile time option string. If N is out of range, sqlite3_compileoption_get() returns a NULL pointer. The SQLITE_ prefix is omitted from any strings returned by sqlite3_compileoption_get().

Support for the diagnostic functions sqlite3_compileoption_used() and sqlite3_compileoption_get() may be omitted by specifying the SQLITE_OMIT_COMPILEOPTION_DIAGS option at compile time.

See also: SQL functions sqlite_compileoption_used() and sqlite_compileoption_get() and the compile_options pragma.

Determine If An SQL Statement Is Complete

```
int sqlite3_complete(const char *sql);
int sqlite3_complete16(const void *sql);
```

These routines are useful during command-line input to determine if the currently entered text seems to form a complete SQL statement or if additional input is needed before sending the text into SQLite for parsing. These routines return 1 if the input string appears to be a complete SQL statement. A statement is judged to be complete if it ends with a semicolon token and is not a prefix of a well-formed CREATE TRIGGER statement. Semicolons that are embedded within string literals or quoted identifier names or comments are not independent tokens (they are part of the token in which they are embedded) and thus do not count as a statement terminator. Whitespace and comments that follow the final semicolon are ignored.

These routines return 0 if the statement is incomplete. If a memory allocation fails, then SQLITE_NOMEM is returned.

These routines do not parse the SQL statements thus will not detect syntactically incorrect SQL.

If SQLite has not been initialized using sqlite3_initialize() prior to invoking sqlite3_complete16() then sqlite3_initialize() is invoked automatically by sqlite3_complete16(). If that initialization fails, then the return value from sqlite3_complete16() will be non-zero regardless of whether or not the input SQL is complete.

The input to sqlite3_complete() must be a zero-terminated UTF-8 string.

The input to sqlite3_complete16() must be a zero-terminated UTF-16 string in native byte order.

Define New Collating Sequences

```
int sqlite3_create_collation(
  sqlite3*,
  const char *zName,
  int eTextRep,
  void *pArg,
  int(*xCompare)(void*,int,const void*,int,const void*)
);
int sqlite3_create_collation_v2(
  sqlite3*,
  const char *zName,
  int eTextRep,
  void *pArg,
  int(*xCompare)(void*,int,const void*,int,const void*),
  void(*xDestroy)(void*)
);
int sqlite3_create_collation16(
  sqlite3*,
  const void *zName,
  int eTextRep,
  void *pArg,
  int(*xCompare)(void*,int,const void*,int,const void*)
);
```

These functions add, remove, or modify a collation associated with the database connection specified as the first argument.

The name of the collation is a UTF-8 string for sqlite3_create_collation() and sqlite3_create_collation_v2() and a UTF-16 string in native byte order for sqlite3_create_collation16(). Collation names that compare equal according to sqlite3_strnicmp() are considered to be the same name.

The third argument (eTextRep) must be one of the constants:

- SQLITE_UTF8,
- SQLITE_UTF16LE,
- SQLITE_UTF16BE,
- SQLITE_UTF16, or
- SQLITE_UTF16_ALIGNED.

The eTextRep argument determines the encoding of strings passed to the collating function callback, xCompare. The SQLITE_UTF16 and SQLITE_UTF16_ALIGNED values for eTextRep force strings to be UTF16 with native byte order. The SQLITE_UTF16_ALIGNED value for eTextRep forces strings to begin on an even byte address.

The fourth argument, pArg, is an application data pointer that is passed through as the first argument to the collating function callback.

The fifth argument, xCompare, is a pointer to the collating function. Multiple collating functions can be registered using the same name but with different eTextRep parameters and SQLite will use whichever function requires the least amount of data transformation. If the xCompare argument is NULL then the collating function is deleted. When all collating functions having the same name are deleted, that collation is no longer usable.

The collating function callback is invoked with a copy of the pArg application data pointer and with two strings in the encoding specified by the eTextRep argument. The two integer parameters to the collating function callback are the length of the two strings, in bytes. The collating function must return an integer that is negative, zero, or positive if the first string is less than, equal to, or greater than the second, respectively. A collating function must always return the same answer given the same inputs. If two or more collating functions are registered to the same collation name (using different eTextRep values) then all must give an equivalent answer when invoked with equivalent strings. The collating function must obey the following properties for all strings A, B, and C:

1. If A==B then B==A.
2. If A==B and B==C then A==C.
3. If A<B THEN B>A.
4. If A<B and B<C then A<C.

If a collating function fails any of the above constraints and that collating function is registered and used, then the behavior of SQLite is undefined.

The sqlite3_create_collation_v2() works like sqlite3_create_collation() with the addition that the xDestroy callback is invoked on pArg when the collating function is deleted. Collating functions are deleted when they are overridden by later calls to the collation creation functions or when the database connection is closed using sqlite3_close().

The xDestroy callback is not called if the sqlite3_create_collation_v2() function fails. Applications that invoke sqlite3_create_collation_v2() with a non-NULL xDestroy argument should check the return code and dispose of the application data pointer themselves rather than expecting SQLite to deal with it for them. This is different from every other SQLite interface. The inconsistency is unfortunate but cannot be changed without breaking backwards compatibility.

See also: sqlite3_collation_needed() and sqlite3_collation_needed16().

Register A Virtual Table Implementation

```
int sqlite3_create_module(
  sqlite3 *db,               /* SQLite connection to register module with */
  const char *zName,         /* Name of the module */
  const sqlite3_module *p,   /* Methods for the module */
  void *pClientData          /* Client data for xCreate/xConnect */
);
int sqlite3_create_module_v2(
  sqlite3 *db,               /* SQLite connection to register module with */
  const char *zName,         /* Name of the module */
  const sqlite3_module *p,   /* Methods for the module */
  void *pClientData,         /* Client data for xCreate/xConnect */
  void(*xDestroy)(void*)     /* Module destructor function */
);
```

These routines are used to register a new virtual table module name. Module names must be registered before creating a new virtual table using the module and before using a preexisting virtual table for the module.

The module name is registered on the database connection specified by the first parameter. The name of the module is given by the second parameter. The third parameter is a pointer to the implementation of the virtual table module. The fourth parameter is an arbitrary client data pointer that is passed through into the xCreate and xConnect methods of the virtual table module when a new virtual table is be being created or reinitialized.

The sqlite3_create_module_v2() interface has a fifth parameter which is a pointer to a destructor for the pClientData. SQLite will invoke the destructor function (if it is not NULL) when SQLite no longer needs the pClientData pointer. The destructor will also be invoked if the call to sqlite3_create_module_v2() fails. The sqlite3_create_module() interface is equivalent to sqlite3_create_module_v2() with a NULL destructor.

If the third parameter (the pointer to the sqlite3_module object) is NULL then no new module is create and any existing modules with the same name are dropped.

See also: sqlite3_drop_modules()

Error Codes And Messages

```
int sqlite3_errcode(sqlite3 *db);
int sqlite3_extended_errcode(sqlite3 *db);
const char *sqlite3_errmsg(sqlite3*);
const void *sqlite3_errmsg16(sqlite3*);
const char *sqlite3_errstr(int);
```

If the most recent sqlite3_* API call associated with database connection D failed, then the sqlite3_errcode(D) interface returns the numeric result code or extended result code for that API call. The sqlite3_extended_errcode() interface is the same except that it always returns the extended result code even when extended result codes are disabled.

The values returned by sqlite3_errcode() and/or sqlite3_extended_errcode() might change with each API call. Except, there are some interfaces that are guaranteed to never change the value of the error code. The error-code preserving interfaces are:

- sqlite3_errcode()
- sqlite3_extended_errcode()
- sqlite3_errmsg()
- sqlite3_errmsg16()

The sqlite3_errmsg() and sqlite3_errmsg16() return English-language text that describes the error, as either UTF-8 or UTF-16 respectively. Memory to hold the error message string is managed internally. The application does not need to worry about freeing the result. However, the error string might be overwritten or deallocated by subsequent calls to other SQLite interface functions.

The sqlite3_errstr() interface returns the English-language text that describes the result code, as UTF-8. Memory to hold the error message string is managed internally and must not be freed by the application.

When the serialized threading mode is in use, it might be the case that a second error occurs on a separate thread in between the time of the first error and the call to these interfaces. When that happens, the second error will be reported since these interfaces always report the most recent result. To avoid this, each thread can obtain exclusive use of the database connection D by invoking sqlite3_mutex_enter(sqlite3_db_mutex(D)) before beginning to use D and invoking sqlite3_mutex_leave(sqlite3_db_mutex(D)) after all calls to the interfaces listed here are completed.

If an interface fails with SQLITE_MISUSE, that means the interface was invoked incorrectly by the application. In that case, the error code and message may or may not be set.

Retrieving Statement SQL

```
const char *sqlite3_sql(sqlite3_stmt *pStmt);
char *sqlite3_expanded_sql(sqlite3_stmt *pStmt);
const char *sqlite3_normalized_sql(sqlite3_stmt *pStmt);
```

The sqlite3_sql(P) interface returns a pointer to a copy of the UTF-8 SQL text used to create prepared statement P if P was created by sqlite3_prepare_v2(), sqlite3_prepare_v3(), sqlite3_prepare16_v2(), or sqlite3_prepare16_v3(). The sqlite3_expanded_sql(P) interface returns a pointer to a UTF-8 string containing the SQL text of prepared statement P with bound parameters expanded. The sqlite3_normalized_sql(P) interface returns a pointer to a UTF-8 string containing the normalized SQL text of prepared statement P. The semantics used to normalize a SQL statement are unspecified and subject to change. At a minimum, literal values will be replaced with suitable placeholders.

For example, if a prepared statement is created using the SQL text "SELECT $abc,:xyz" and if parameter $abc is bound to integer 2345 and parameter :xyz is unbound, then sqlite3_sql() will return the original string, "SELECT $abc,:xyz" but sqlite3_expanded_sql() will return "SELECT 2345,NULL".

The sqlite3_expanded_sql() interface returns NULL if insufficient memory is available to hold the result, or if the result would exceed the the maximum string length determined by the SQLITE_LIMIT_LENGTH.

The SQLITE_TRACE_SIZE_LIMIT compile-time option limits the size of bound parameter expansions. The SQLITE_OMIT_TRACE compile-time option causes sqlite3_expanded_sql() to always return NULL.

The strings returned by sqlite3_sql(P) and sqlite3_normalized_sql(P) are managed by SQLite and are automatically freed when the prepared statement is finalized. The string returned by sqlite3_expanded_sql(P), on the other hand, is obtained from sqlite3_malloc() and must be free by the application by passing it to sqlite3_free().

Translate filenames

```
const char *sqlite3_filename_database(const char*);
const char *sqlite3_filename_journal(const char*);
const char *sqlite3_filename_wal(const char*);
```

These routines are available to custom VFS implementations for translating filenames between the main database file, the journal file, and the WAL file.

If F is the name of an sqlite database file, journal file, or WAL file passed by the SQLite core into the VFS, then sqlite3_filename_database(F) returns the name of the corresponding database file.

If F is the name of an sqlite database file, journal file, or WAL file passed by the SQLite core into the VFS, or if F is a database filename obtained from sqlite3_db_filename(), then sqlite3_filename_journal(F) returns the name of the corresponding rollback journal file.

If F is the name of an sqlite database file, journal file, or WAL file that was passed by the SQLite core into the VFS, or if F is a database filename obtained from sqlite3_db_filename(), then sqlite3_filename_wal(F) returns the name of the corresponding WAL file.

In all of the above, if F is not the name of a database, journal or WAL filename passed into the VFS from the SQLite core and F is not the return value from sqlite3_db_filename(), then the result is undefined and is likely a memory access violation.

Memory Allocation Subsystem

```
void *sqlite3_malloc(int);
void *sqlite3_malloc64(sqlite3_uint64);
void *sqlite3_realloc(void*, int);
void *sqlite3_realloc64(void*, sqlite3_uint64);
void sqlite3_free(void*);
sqlite3_uint64 sqlite3_msize(void*);
```

The SQLite core uses these three routines for all of its own internal memory allocation needs. "Core" in the previous sentence does not include operating-system specific VFS implementation. The Windows VFS uses native malloc() and free() for some operations.

The sqlite3_malloc() routine returns a pointer to a block of memory at least N bytes in length, where N is the parameter. If sqlite3_malloc() is unable to obtain sufficient free memory, it returns a NULL pointer. If the parameter N to sqlite3_malloc() is zero or negative then sqlite3_malloc() returns a NULL pointer.

The sqlite3_malloc64(N) routine works just like sqlite3_malloc(N) except that N is an unsigned 64-bit integer instead of a signed 32-bit integer.

Calling sqlite3_free() with a pointer previously returned by sqlite3_malloc() or sqlite3_realloc() releases that memory so that it might be reused. The sqlite3_free() routine is a no-op if is called with a NULL pointer. Passing a NULL pointer to sqlite3_free() is harmless. After being freed, memory should neither be read nor written. Even reading previously freed memory might result in a segmentation fault or other severe error. Memory corruption, a segmentation fault, or other severe error might result if sqlite3_free() is called with a non-NULL pointer that was not obtained from sqlite3_malloc() or sqlite3_realloc().

The sqlite3_realloc(X,N) interface attempts to resize a prior memory allocation X to be at least N bytes. If the X parameter to sqlite3_realloc(X,N) is a NULL pointer then its behavior is identical to calling sqlite3_malloc(N). If the N parameter to sqlite3_realloc(X,N) is zero or negative then the behavior is exactly the same as calling sqlite3_free(X). sqlite3_realloc(X,N) returns a pointer to a memory allocation of at least N bytes in size or NULL if insufficient memory is available. If M is the size of the prior allocation, then min(N,M) bytes of the prior allocation are copied into the beginning of buffer returned by sqlite3_realloc(X,N) and the prior allocation is freed. If sqlite3_realloc(X,N) returns NULL and N is positive, then the prior allocation is not freed.

The sqlite3_realloc64(X,N) interfaces works the same as sqlite3_realloc(X,N) except that N is a 64-bit unsigned integer instead of a 32-bit signed integer.

If X is a memory allocation previously obtained from sqlite3_malloc(), sqlite3_malloc64(), sqlite3_realloc(), or sqlite3_realloc64(), then sqlite3_msize(X) returns the size of that memory allocation in bytes. The value returned by sqlite3_msize(X) might be larger than the number of bytes requested when X was allocated. If X is a NULL pointer then sqlite3_msize(X) returns zero. If X points to something that is not the beginning of memory allocation, or if it points to a formerly valid memory allocation that has now been freed, then the behavior of sqlite3_msize(X) is undefined and possibly harmful.

The memory returned by sqlite3_malloc(), sqlite3_realloc(), sqlite3_malloc64(), and sqlite3_realloc64() is always aligned to at least an 8 byte boundary, or to a 4 byte boundary if the SQLITE_4_BYTE_ALIGNED_MALLOC compile-time option is used.

The pointer arguments to sqlite3_free() and sqlite3_realloc() must be either NULL or else pointers obtained from a prior invocation of sqlite3_malloc() or sqlite3_realloc() that have not yet been released.

The application must not read or write any part of a block of memory after it has been released using sqlite3_free() or sqlite3_realloc().

Convenience Routines For Running Queries

```
int sqlite3_get_table(
  sqlite3 *db,          /* An open database */
  const char *zSql,     /* SQL to be evaluated */
  char ***pazResult,    /* Results of the query */
  int *pnRow,           /* Number of result rows written here */
  int *pnColumn,        /* Number of result columns written here */
  char **pzErrmsg       /* Error msg written here */
);
void sqlite3_free_table(char **result);
```

This is a legacy interface that is preserved for backwards compatibility. Use of this interface is not recommended.

Definition: A **result table** is memory data structure created by the sqlite3_get_table() interface. A result table records the complete query results from one or more queries.

The table conceptually has a number of rows and columns. But these numbers are not part of the result table itself. These numbers are obtained separately. Let N be the number of rows and M be the number of columns.

A result table is an array of pointers to zero-terminated UTF-8 strings. There are (N+1)*M elements in the array. The first M pointers point to zero-terminated strings that contain the names of the columns. The remaining entries all point to query results. NULL values result in NULL pointers. All other values are in their UTF-8 zero-terminated string representation as returned by sqlite3_column_text().

A result table might consist of one or more memory allocations. It is not safe to pass a result table directly to sqlite3_free(). A result table should be deallocated using sqlite3_free_table().

As an example of the result table format, suppose a query result is as follows:

```
Name        | Age
-----------------------
Alice       | 43
Bob         | 28
Cindy       | 21
```

There are two columns (M==2) and three rows (N==3). Thus the result table has 8 entries. Suppose the result table is stored in an array named azResult. Then azResult holds this content:

```
azResult[0] = "Name";
azResult[1] = "Age";
azResult[2] = "Alice";
azResult[3] = "43";
azResult[4] = "Bob";
azResult[5] = "28";
azResult[6] = "Cindy";
azResult[7] = "21";
```

The sqlite3_get_table() function evaluates one or more semicolon-separated SQL statements in the zero-terminated UTF-8 string of its 2nd parameter and returns a result table to the pointer given in its 3rd parameter.

After the application has finished with the result from sqlite3_get_table(), it must pass the result table pointer to sqlite3_free_table() in order to release the memory that was malloced. Because of the way the sqlite3_malloc() happens within sqlite3_get_table(), the calling function must not try to call sqlite3_free() directly. Only sqlite3_free_table() is able to release the memory properly and safely.

The sqlite3_get_table() interface is implemented as a wrapper around sqlite3_exec(). The sqlite3_get_table() routine does not have access to any internal data structures of SQLite. It uses only the public interface defined here. As a consequence, errors that occur in the wrapper layer outside of the internal sqlite3_exec() call are not reflected in subsequent calls to sqlite3_errcode() or sqlite3_errmsg().

Function Auxiliary Data

```
void *sqlite3_get_auxdata(sqlite3_context*, int N);
void sqlite3_set_auxdata(sqlite3_context*, int N, void*, void (*)(void*));
```

These functions may be used by (non-aggregate) SQL functions to associate metadata with argument values. If the same value is passed to multiple invocations of the same SQL function during query execution, under some circumstances the associated metadata may be preserved. An example of where this might be useful is in a regular-expression matching function. The compiled version of the regular expression can be stored as metadata associated with the pattern string. Then as long as the pattern string remains the same, the compiled regular expression can be reused on multiple invocations of the same function.

The sqlite3_get_auxdata(C,N) interface returns a pointer to the metadata associated by the sqlite3_set_auxdata(C,N,P,X) function with the Nth argument value to the application-defined function. N is zero for the left-most function argument. If there is no metadata associated with the function argument, the sqlite3_get_auxdata(C,N) interface returns a NULL pointer.

The sqlite3_set_auxdata(C,N,P,X) interface saves P as metadata for the N-th argument of the application-defined function. Subsequent calls to sqlite3_get_auxdata(C,N) return P from the most recent sqlite3_set_auxdata(C,N,P,X) call if the metadata is still valid or NULL if the metadata has been discarded. After each call to sqlite3_set_auxdata(C,N,P,X) where X is not NULL, SQLite will invoke the destructor function X with parameter P exactly once, when the metadata is discarded. SQLite is free to discard the metadata at any time, including:

- when the corresponding function parameter changes, or
- when sqlite3_reset() or sqlite3_finalize() is called for the SQL statement, or
- when sqlite3_set_auxdata() is invoked again on the same parameter, or
- during the original sqlite3_set_auxdata() call when a memory allocation error occurs.

Note the last bullet in particular. The destructor X in sqlite3_set_auxdata(C,N,P,X) might be called immediately, before the sqlite3_set_auxdata() interface even returns. Hence sqlite3_set_auxdata() should be called near the end of the function implementation and the function implementation should not make any use of P after sqlite3_set_auxdata() has been called.

In practice, metadata is preserved between function calls for function parameters that are compile-time constants, including literal values and parameters and expressions composed from the same.

The value of the N parameter to these interfaces should be non-negative. Future enhancements may make use of negative N values to define new kinds of function caching behavior.

These routines must be called from the same thread in which the SQL function is running.

Impose A Limit On Heap Size

```
sqlite3_int64 sqlite3_soft_heap_limit64(sqlite3_int64 N);
sqlite3_int64 sqlite3_hard_heap_limit64(sqlite3_int64 N);
```

These interfaces impose limits on the amount of heap memory that will be by all database connections within a single process.

The sqlite3_soft_heap_limit64() interface sets and/or queries the soft limit on the amount of heap memory that may be allocated by SQLite. SQLite strives to keep heap memory utilization below the soft heap limit by reducing the number of pages held in the page cache as heap memory usages approaches the limit. The soft heap limit is "soft" because even though SQLite strives to stay below the limit, it will exceed the limit rather than generate an SQLITE_NOMEM error. In other words, the soft heap limit is advisory only.

The sqlite3_hard_heap_limit64(N) interface sets a hard upper bound of N bytes on the amount of memory that will be allocated. The sqlite3_hard_heap_limit64(N) interface is similar to sqlite3_soft_heap_limit64(N) except that memory allocations will fail when the hard heap limit is reached.

The return value from both sqlite3_soft_heap_limit64() and sqlite3_hard_heap_limit64() is the size of the heap limit prior to the call, or negative in the case of an error. If the argument N is negative then no change is made to the heap limit. Hence, the current size of heap limits can be determined by invoking sqlite3_soft_heap_limit64(-1) or sqlite3_hard_heap_limit(-1).

Setting the heap limits to zero disables the heap limiter mechanism.

The soft heap limit may not be greater than the hard heap limit. If the hard heap limit is enabled and if sqlite3_soft_heap_limit(N) is invoked with a value of N that is greater than the hard heap limit, the the soft heap limit is set to the value of the hard heap limit. The soft heap limit is automatically enabled whenever the hard heap limit is enabled. When sqlite3_hard_heap_limit64(N) is invoked and the soft heap limit is outside the range of 1..N, then the soft heap limit is set to N. Invoking sqlite3_soft_heap_limit64(0) when the hard heap limit is enabled makes the soft heap limit equal to the hard heap limit.

The memory allocation limits can also be adjusted using PRAGMA soft_heap_limit and PRAGMA hard_heap_limit.

The heap limits are not enforced in the current implementation if one or more of following conditions are true:

- The limit value is set to zero.
- Memory accounting is disabled using a combination of the sqlite3_config(SQLITE_CONFIG_MEMSTATUS,...) start-time option and the SQLITE_DEFAULT_MEMSTATUS compile-time option.
- An alternative page cache implementation is specified using sqlite3_config(SQLITE_CONFIG_PCACHE2,...).
- The page cache allocates from its own memory pool supplied by sqlite3_config(SQLITE_CONFIG_PAGECACHE,...) rather than from the heap.

The circumstances under which SQLite will enforce the heap limits may changes in future releases of SQLite.

Initialize The SQLite Library

```
int sqlite3_initialize(void);
int sqlite3_shutdown(void);
int sqlite3_os_init(void);
int sqlite3_os_end(void);
```

The sqlite3_initialize() routine initializes the SQLite library. The sqlite3_shutdown() routine deallocates any resources that were allocated by sqlite3_initialize(). These routines are designed to aid in process initialization and shutdown on embedded systems. Workstation applications using SQLite normally do not need to invoke either of these routines.

A call to sqlite3_initialize() is an "effective" call if it is the first time sqlite3_initialize() is invoked during the lifetime of the process, or if it is the first time sqlite3_initialize() is invoked following a call to sqlite3_shutdown(). Only an effective call of sqlite3_initialize() does any initialization. All other calls are harmless no-ops.

A call to sqlite3_shutdown() is an "effective" call if it is the first call to sqlite3_shutdown() since the last sqlite3_initialize(). Only an effective call to sqlite3_shutdown() does any deinitialization. All other valid calls to sqlite3_shutdown() are harmless no-ops.

The sqlite3_initialize() interface is threadsafe, but sqlite3_shutdown() is not. The sqlite3_shutdown() interface must only be called from a single thread. All open database connections must be closed and all other SQLite resources must be deallocated prior to invoking sqlite3_shutdown().

Among other things, sqlite3_initialize() will invoke sqlite3_os_init(). Similarly, sqlite3_shutdown() will invoke sqlite3_os_end().

The sqlite3_initialize() routine returns SQLITE_OK on success. If for some reason, sqlite3_initialize() is unable to initialize the library (perhaps it is unable to allocate a needed resource such as a mutex) it returns an error code other than SQLITE_OK.

The sqlite3_initialize() routine is called internally by many other SQLite interfaces so that an application usually does not need to invoke sqlite3_initialize() directly. For example, sqlite3_open() calls sqlite3_initialize() so the SQLite library will be automatically initialized when sqlite3_open() is called if it has not be initialized already. However, if SQLite is compiled with the SQLITE_OMIT_AUTOINIT compile-time option, then the automatic calls to sqlite3_initialize() are omitted and the application must call sqlite3_initialize() directly prior to using any other SQLite interface. For maximum portability, it is recommended that applications always invoke sqlite3_initialize() directly prior to using any other SQLite interface. Future releases of SQLite may require this. In other words, the behavior exhibited when SQLite is compiled with SQLITE_OMIT_AUTOINIT might become the default behavior in some future release of SQLite.

The sqlite3_os_init() routine does operating-system specific initialization of the SQLite library. The sqlite3_os_end() routine undoes the effect of sqlite3_os_init(). Typical tasks performed by these routines include allocation or deallocation of static resources, initialization of global variables, setting up a default sqlite3_vfs module, or setting up a default configuration using sqlite3_config().

The application should never invoke either sqlite3_os_init() or sqlite3_os_end() directly. The application should only invoke sqlite3_initialize() and sqlite3_shutdown(). The sqlite3_os_init() interface is called automatically by sqlite3_initialize() and sqlite3_os_end() is called by sqlite3_shutdown(). Appropriate implementations for sqlite3_os_init() and sqlite3_os_end() are built into SQLite when it is compiled for Unix, Windows, or OS/2. When built for other platforms (using the SQLITE_OS_OTHER=1 compile-time option) the application must supply a suitable implementation for sqlite3_os_init() and sqlite3_os_end(). An application-supplied implementation of sqlite3_os_init() or sqlite3_os_end() must return SQLITE_OK on success and some other error code upon failure.

SQL Keyword Checking

```
int sqlite3_keyword_count(void);
int sqlite3_keyword_name(int,const char**,int*);
int sqlite3_keyword_check(const char*,int);
```

These routines provide access to the set of SQL language keywords recognized by SQLite. Applications can uses these routines to determine whether or not a specific identifier needs to be escaped (for example, by enclosing in double-quotes) so as not to confuse the parser.

The sqlite3_keyword_count() interface returns the number of distinct keywords understood by SQLite.

The sqlite3_keyword_name(N,Z,L) interface finds the N-th keyword and makes *Z point to that keyword expressed as UTF8 and writes the number of bytes in the keyword into *L. The string that *Z points to is not zero-terminated. The sqlite3_keyword_name(N,Z,L) routine returns SQLITE_OK if N is within bounds and SQLITE_ERROR if not. If either Z or L are NULL or invalid pointers then calls to sqlite3_keyword_name(N,Z,L) result in undefined behavior.

The sqlite3_keyword_check(Z,L) interface checks to see whether or not the L-byte UTF8 identifier that Z points to is a keyword, returning non-zero if it is and zero if not.

The parser used by SQLite is forgiving. It is often possible to use a keyword as an identifier as long as such use does not result in a parsing ambiguity. For example, the statement "CREATE TABLE BEGIN(REPLACE,PRAGMA,END);" is accepted by SQLite, and creates a new table named "BEGIN" with three columns named "REPLACE", "PRAGMA", and "END". Nevertheless, best practice is to avoid using keywords as identifiers. Common techniques used to avoid keyword name collisions include:

- Put all identifier names inside double-quotes. This is the official SQL way to escape identifier names.
- Put identifier names inside [...]. This is not standard SQL, but it is what SQL Server does and so lots of programmers use this technique.
- Begin every identifier with the letter "Z" as no SQL keywords start with "Z".
- Include a digit somewhere in every identifier name.

Note that the number of keywords understood by SQLite can depend on compile-time options. For example, "VACUUM" is not a keyword if SQLite is compiled with the -DSQLITE_OMIT_VACUUM option. Also, new keywords may be added to future releases of SQLite.

Run-Time Library Version Numbers

```
SQLITE_EXTERN const char sqlite3_version[];
const char *sqlite3_libversion(void);
const char *sqlite3_sourceid(void);
int sqlite3_libversion_number(void);
```

These interfaces provide the same information as the SQLITE_VERSION, SQLITE_VERSION_NUMBER, and SQLITE_SOURCE_ID C preprocessor macros but are associated with the library instead of the header file. Cautious programmers might include assert() statements in their application to verify that values returned by these interfaces match the macros in the header, and thus ensure that the application is compiled with matching library and header files.

```
assert( sqlite3_libversion_number()==SQLITE_VERSION_NUMBER );
assert( strncmp(sqlite3_sourceid(),SQLITE_SOURCE_ID,80)==0 );
assert( strcmp(sqlite3_libversion(),SQLITE_VERSION)==0 );
```

The sqlite3_version[] string constant contains the text of SQLITE_VERSION macro. The sqlite3_libversion() function returns a pointer to the to the sqlite3_version[] string constant. The sqlite3_libversion() function is provided for use in DLLs since DLL users usually do not have direct access to string constants within the DLL. The sqlite3_libversion_number() function returns an integer equal to SQLITE_VERSION_NUMBER. The sqlite3_sourceid() function returns a pointer to a string constant whose value is the same as the SQLITE_SOURCE_ID C preprocessor macro. Except if SQLite is built using an edited copy of the amalgamation, then the last four characters of the hash might be different from SQLITE_SOURCE_ID.

See also: sqlite_version() and sqlite_source_id().

Memory Allocator Statistics

```
sqlite3_int64 sqlite3_memory_used(void);
sqlite3_int64 sqlite3_memory_highwater(int resetFlag);
```

SQLite provides these two interfaces for reporting on the status of the sqlite3_malloc(), sqlite3_free(), and sqlite3_realloc() routines, which form the built-in memory allocation subsystem.

The sqlite3_memory_used() routine returns the number of bytes of memory currently outstanding (malloced but not freed). The sqlite3_memory_highwater() routine returns the maximum value of sqlite3_memory_used() since the high-water mark was last reset. The values returned by sqlite3_memory_used() and sqlite3_memory_highwater() include any overhead added by SQLite in its implementation of sqlite3_malloc(), but not overhead added by the any underlying system library routines that sqlite3_malloc() may call.

The memory high-water mark is reset to the current value of sqlite3_memory_used() if and only if the parameter to sqlite3_memory_highwater() is true. The value returned by sqlite3_memory_highwater(1) is the high-water mark prior to the reset.

Formatted String Printing Functions

```
char *sqlite3_mprintf(const char*,...);
char *sqlite3_vmprintf(const char*, va_list);
char *sqlite3_snprintf(int,char*,const char*, ...);
char *sqlite3_vsnprintf(int,char*,const char*, va_list);
```

These routines are work-alikes of the "printf()" family of functions from the standard C library. These routines understand most of the common formatting options from the standard library printf() plus some additional non-standard formats (%q, %Q, %w, and %z). See the built-in printf() documentation for details.

The sqlite3_mprintf() and sqlite3_vmprintf() routines write their results into memory obtained from sqlite3_malloc64(). The strings returned by these two routines should be released by sqlite3_free(). Both routines return a NULL pointer if sqlite3_malloc64() is unable to allocate enough memory to hold the resulting string.

The sqlite3_snprintf() routine is similar to "snprintf()" from the standard C library. The result is written into the buffer supplied as the second parameter whose size is given by the first parameter. Note that the order of the first two parameters is reversed from snprintf(). This is an historical accident that cannot be fixed without breaking backwards compatibility. Note also that sqlite3_snprintf() returns a pointer to its buffer instead of the number of characters actually written into the buffer. We admit that the number of characters written would be a more useful return value but we cannot change the implementation of sqlite3_snprintf() now without breaking compatibility.

As long as the buffer size is greater than zero, sqlite3_snprintf() guarantees that the buffer is always zero-terminated. The first parameter "n" is the total size of the buffer, including space for the zero terminator. So the longest string that can be completely written will be n-1 characters.

The sqlite3_vsnprintf() routine is a varargs version of sqlite3_snprintf().

See also: built-in printf(), printf() SQL function

Mutexes

```
sqlite3_mutex *sqlite3_mutex_alloc(int);
void sqlite3_mutex_free(sqlite3_mutex*);
void sqlite3_mutex_enter(sqlite3_mutex*);
int sqlite3_mutex_try(sqlite3_mutex*);
void sqlite3_mutex_leave(sqlite3_mutex*);
```

The SQLite core uses these routines for thread synchronization. Though they are intended for internal use by SQLite, code that links against SQLite is permitted to use any of these routines.

The SQLite source code contains multiple implementations of these mutex routines. An appropriate implementation is selected automatically at compile-time. The following implementations are available in the SQLite core:

- SQLITE_MUTEX_PTHREADS
- SQLITE_MUTEX_W32
- SQLITE_MUTEX_NOOP

The SQLITE_MUTEX_NOOP implementation is a set of routines that does no real locking and is appropriate for use in a single-threaded application. The SQLITE_MUTEX_PTHREADS and SQLITE_MUTEX_W32 implementations are appropriate for use on Unix and Windows.

If SQLite is compiled with the SQLITE_MUTEX_APPDEF preprocessor macro defined (with "-DSQLITE_MUTEX_APPDEF=1"), then no mutex implementation is included with the library. In this case the application must supply a custom mutex implementation using

the SQLITE_CONFIG_MUTEX option of the sqlite3_config() function before calling sqlite3_initialize() or any other public sqlite3_ function that calls sqlite3_initialize().

The sqlite3_mutex_alloc() routine allocates a new mutex and returns a pointer to it. The sqlite3_mutex_alloc() routine returns NULL if it is unable to allocate the requested mutex. The argument to sqlite3_mutex_alloc() must one of these integer constants:

- SQLITE_MUTEX_FAST
- SQLITE_MUTEX_RECURSIVE
- SQLITE_MUTEX_STATIC_MASTER
- SQLITE_MUTEX_STATIC_MEM
- SQLITE_MUTEX_STATIC_OPEN
- SQLITE_MUTEX_STATIC_PRNG
- SQLITE_MUTEX_STATIC_LRU
- SQLITE_MUTEX_STATIC_PMEM
- SQLITE_MUTEX_STATIC_APP1
- SQLITE_MUTEX_STATIC_APP2
- SQLITE_MUTEX_STATIC_APP3
- SQLITE_MUTEX_STATIC_VFS1
- SQLITE_MUTEX_STATIC_VFS2
- SQLITE_MUTEX_STATIC_VFS3

The first two constants (SQLITE_MUTEX_FAST and SQLITE_MUTEX_RECURSIVE) cause sqlite3_mutex_alloc() to create a new mutex. The new mutex is recursive when SQLITE_MUTEX_RECURSIVE is used but not necessarily so when SQLITE_MUTEX_FAST is used. The mutex implementation does not need to make a distinction between SQLITE_MUTEX_RECURSIVE and SQLITE_MUTEX_FAST if it does not want to. SQLite will only request a recursive mutex in cases where it really needs one. If a faster non-recursive mutex implementation is available on the host platform, the mutex subsystem might return such a mutex in response to SQLITE_MUTEX_FAST.

The other allowed parameters to sqlite3_mutex_alloc() (anything other than SQLITE_MUTEX_FAST and SQLITE_MUTEX_RECURSIVE) each return a pointer to a static preexisting mutex. Nine static mutexes are used by the current version of SQLite. Future versions of SQLite may add additional static mutexes. Static mutexes are for internal use by SQLite only. Applications that use SQLite mutexes should use only the dynamic mutexes returned by SQLITE_MUTEX_FAST or SQLITE_MUTEX_RECURSIVE.

Note that if one of the dynamic mutex parameters (SQLITE_MUTEX_FAST or SQLITE_MUTEX_RECURSIVE) is used then sqlite3_mutex_alloc() returns a different mutex on every call. For the static mutex types, the same mutex is returned on every call that has the same type number.

The sqlite3_mutex_free() routine deallocates a previously allocated dynamic mutex. Attempting to deallocate a static mutex results in undefined behavior.

The sqlite3_mutex_enter() and sqlite3_mutex_try() routines attempt to enter a mutex. If another thread is already within the mutex, sqlite3_mutex_enter() will block and sqlite3_mutex_try() will return SQLITE_BUSY. The sqlite3_mutex_try() interface returns SQLITE_OK upon successful entry. Mutexes created using SQLITE_MUTEX_RECURSIVE can be entered multiple times by the same thread. In such cases, the mutex must be exited an equal number of times before another thread can enter. If the same thread tries to enter any mutex other than an SQLITE_MUTEX_RECURSIVE more than once, the behavior is undefined.

Some systems (for example, Windows 95) do not support the operation implemented by sqlite3_mutex_try(). On those systems, sqlite3_mutex_try() will always return SQLITE_BUSY. The SQLite core only ever uses sqlite3_mutex_try() as an optimization so this is acceptable behavior.

The sqlite3_mutex_leave() routine exits a mutex that was previously entered by the same thread. The behavior is undefined if the mutex is not currently entered by the calling thread or is not currently allocated.

If the argument to sqlite3_mutex_enter(), sqlite3_mutex_try(), or sqlite3_mutex_leave() is a NULL pointer, then all three routines behave as no-ops.

See also: sqlite3_mutex_held() and sqlite3_mutex_notheld().

Mutex Verification Routines

```
#ifndef NDEBUG
int sqlite3_mutex_held(sqlite3_mutex*);
int sqlite3_mutex_notheld(sqlite3_mutex*);
#endif
```

The sqlite3_mutex_held() and sqlite3_mutex_notheld() routines are intended for use inside assert() statements. The SQLite core never uses these routines except inside an assert() and applications are advised to follow the lead of the core. The SQLite core only provides implementations for these routines when it is compiled with the SQLITE_DEBUG flag. External mutex implementations are only required to provide these routines if SQLITE_DEBUG is defined and if NDEBUG is not defined.

These routines should return true if the mutex in their argument is held or not held, respectively, by the calling thread.

The implementation is not required to provide versions of these routines that actually work. If the implementation does not provide working versions of these routines, it should at least provide stubs that always return true so that one does not get spurious assertion failures.

If the argument to sqlite3_mutex_held() is a NULL pointer then the routine should return 1. This seems counter-intuitive since clearly the mutex cannot be held if it does not exist. But the reason the mutex does not exist is because the build is not using mutexes. And we do not want the assert() containing the call to sqlite3_mutex_held() to fail, so a non-zero return is the appropriate thing to do. The sqlite3_mutex_notheld() interface should also return 1 when given a NULL pointer.

Opening A New Database Connection

```
int sqlite3_open(
  const char *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);
int sqlite3_open16(
  const void *filename,   /* Database filename (UTF-16) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);
int sqlite3_open_v2(
  const char *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb,         /* OUT: SQLite db handle */
  int flags,              /* Flags */
  const char *zVfs        /* Name of VFS module to use */
);
```

These routines open an SQLite database file as specified by the filename argument. The filename argument is interpreted as UTF-8 for sqlite3_open() and sqlite3_open_v2() and as UTF-16 in the native byte order for sqlite3_open16(). A database connection handle is usually returned in *ppDb, even if an error occurs. The only exception is that if SQLite is unable to allocate memory to hold the sqlite3 object, a NULL will be written into *ppDb instead of a pointer to the sqlite3 object. If the database is opened (and/or created) successfully, then SQLITE_OK is returned. Otherwise an error code is returned. The sqlite3_errmsg() or sqlite3_errmsg16() routines can be used to obtain an English language description of the error following a failure of any of the sqlite3_open() routines.

The default encoding will be UTF-8 for databases created using sqlite3_open() or sqlite3_open_v2(). The default encoding for databases created using sqlite3_open16() will be UTF-16 in the native byte order.

Whether or not an error occurs when it is opened, resources associated with the database connection handle should be released by passing it to sqlite3_close() when it is no longer required.

The sqlite3_open_v2() interface works like sqlite3_open() except that it accepts two additional parameters for additional control over the new database connection. The flags parameter to sqlite3_open_v2() must include, at a minimum, one of the following three flag combinations:

SQLITE_OPEN_READONLY
: The database is opened in read-only mode. If the database does not already exist, an error is returned.

SQLITE_OPEN_READWRITE
: The database is opened for reading and writing if possible, or reading only if the file is write protected by the operating system. In either case the database must already exist, otherwise an error is returned.

SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE
: The database is opened for reading and writing, and is created if it does not already exist. This is the behavior that is always used for sqlite3_open() and sqlite3_open16().

In addition to the required flags, the following optional flags are also supported:

SQLITE_OPEN_URI
: The filename can be interpreted as a URI if this flag is set.

SQLITE_OPEN_MEMORY
: The database will be opened as an in-memory database. The database is named by the "filename" argument for the purposes of cache-sharing, if shared cache mode is enabled, but the "filename" is otherwise ignored.

SQLITE_OPEN_NOMUTEX
: The new database connection will use the "multi-thread" threading mode. This means that separate threads are allowed to use SQLite at the same time, as long as each thread is using a different database connection.

SQLITE_OPEN_FULLMUTEX
: The new database connection will use the "serialized" threading mode. This means the multiple threads can safely attempt to use the same database connection at the same time. (Mutexes will block any actual concurrency, but in this mode there is no harm in trying.)

SQLITE_OPEN_SHAREDCACHE
: The database is opened shared cache enabled, overriding the default shared cache setting provided by sqlite3_enable_shared_cache().

SQLITE_OPEN_PRIVATECACHE
: The database is opened shared cache disabled, overriding the default shared cache setting provided by sqlite3_enable_shared_cache().

SQLITE_OPEN_NOFOLLOW
: The database filename is not allowed to be a symbolic link

If the 3rd parameter to sqlite3_open_v2() is not one of the required combinations shown above optionally combined with other SQLITE_OPEN_* bits then the behavior is undefined.

The fourth parameter to sqlite3_open_v2() is the name of the sqlite3_vfs object that defines the operating system interface that the new database connection should use. If the fourth parameter is a NULL pointer then the default sqlite3_vfs object is used.

If the filename is ":memory:", then a private, temporary in-memory database is created for the connection. This in-memory database will vanish when the database connection is closed. Future versions of SQLite might make use of additional special filenames that begin with the ":" character. It is recommended that when a database filename actually does begin with a ":" character you should prefix the filename with a pathname such as "./" to avoid ambiguity.

If the filename is an empty string, then a private, temporary on-disk database will be created. This private database will be automatically deleted as soon as the database connection is closed.

URI Filenames

If URI filename interpretation is enabled, and the filename argument begins with "file:", then the filename is interpreted as a URI. URI filename interpretation is enabled if the SQLITE_OPEN_URI flag is set in the third argument to sqlite3_open_v2(), or if it has been enabled globally using the SQLITE_CONFIG_URI option with the sqlite3_config() method or by the SQLITE_USE_URI compile-time option. URI filename interpretation is turned off by default, but future releases of SQLite might enable URI filename interpretation by default. See "URI filenames" for additional information.

URI filenames are parsed according to RFC 3986. If the URI contains an authority, then it must be either an empty string or the string "localhost". If the authority is not an empty string or "localhost", an error is returned to the caller. The fragment component of a URI, if present, is ignored.

SQLite uses the path component of the URI as the name of the disk file which contains the database. If the path begins with a '/' character, then it is interpreted as an absolute path. If the path does not begin with a '/' (meaning that the authority section is omitted from the URI) then the path is interpreted as a relative path. On windows, the first component of an absolute path is a drive specification (e.g. "C:").

The query component of a URI may contain parameters that are interpreted either by SQLite itself, or by a custom VFS implementation. SQLite and its built-in VFSes interpret the following query parameters:

- **vfs**: The "vfs" parameter may be used to specify the name of a VFS object that provides the operating system interface that should be used to access the database file on disk. If this option is set to an empty string the default VFS object is used. Specifying an unknown VFS is an error. If sqlite3_open_v2() is used and the vfs option is present, then the VFS specified by the option takes precedence over the value passed as the fourth parameter to sqlite3_open_v2().
- **mode**: The mode parameter may be set to either "ro", "rw", "rwc", or "memory". Attempting to set it to any other value is an error. If "ro" is specified, then the database is opened for read-only access, just as if the SQLITE_OPEN_READONLY flag had been set in the third argument to sqlite3_open_v2(). If the mode option is set to "rw", then the database is opened for read-write (but not create) access, as if SQLITE_OPEN_READWRITE (but not SQLITE_OPEN_CREATE) had been set. Value "rwc" is equivalent to setting both SQLITE_OPEN_READWRITE and SQLITE_OPEN_CREATE. If the mode option is set to "memory" then a pure in-memory database that never reads or writes from disk is used. It is an error to specify a value for the mode parameter that is less restrictive than that specified by the flags passed in the third parameter to sqlite3_open_v2().
- **cache**: The cache parameter may be set to either "shared" or "private". Setting it to "shared" is equivalent to setting the SQLITE_OPEN_SHAREDCACHE bit in the flags argument passed to sqlite3_open_v2(). Setting the cache parameter to "private" is equivalent to setting the SQLITE_OPEN_PRIVATECACHE bit. If sqlite3_open_v2() is used and the "cache" parameter is present in a URI filename, its value overrides any behavior requested by setting SQLITE_OPEN_PRIVATECACHE or SQLITE_OPEN_SHAREDCACHE flag.
- **psow**: The psow parameter indicates whether or not the powersafe overwrite property does or does not apply to the storage media on which the database file resides.
- **nolock**: The nolock parameter is a boolean query parameter which if set disables file locking in rollback journal modes. This is useful for accessing a database on a filesystem that does not support locking. Caution: Database corruption might result if two or more processes write to the same database and any one of those processes uses nolock=1.
- **immutable**: The immutable parameter is a boolean query parameter that indicates that the database file is stored on read-only media. When immutable is set, SQLite assumes that the database file cannot be changed, even by a process with higher privilege, and so the database is opened read-only and all locking and change detection is disabled. Caution: Setting the immutable property on a database file that does in fact change can result in incorrect query results and/or SQLITE_CORRUPT errors. See also: SQLITE_IOCAP_IMMUTABLE.

Specifying an unknown parameter in the query component of a URI is not an error. Future versions of SQLite might understand additional query parameters. See "query parameters with special meaning to SQLite" for additional information.

URI filename examples

URI filenames	Results
file:data.db	Open the file "data.db" in the current directory.
file:/home/fred/data.db file:///home/fred/data.db file://localhost/home/fred/data.db	Open the database file "/home/fred/data.db".
file://darkstar/home/fred/data.db	An error. "darkstar" is not a recognized authority.
file:///C:/Documents%20and%20Settings/fred/Desktop/data.db	Windows only: Open the file "data.db" on fred's desktop on drive C:. Note that the %20 escaping in this example is not strictly necessary - space characters can be used literally in URI filenames.
file:data.db?mode=ro&cache=private	Open file "data.db" in the current directory for read-only access. Regardless of whether or not shared-cache mode is enabled by default, use a private cache.
file:/home/fred/data.db?vfs=unix-dotfile	Open file "/home/fred/data.db". Use the special VFS "unix-dotfile" that uses dot-files in place of posix advisory locking.
file:data.db?mode=readonly	An error. "readonly" is not a valid option for the "mode" parameter.

URI hexadecimal escape sequences (%HH) are supported within the path and query components of a URI. A hexadecimal escape sequence consists of a percent sign - "%" - followed by exactly two hexadecimal digits specifying an octet value. Before the path or

query components of a URI filename are interpreted, they are encoded using UTF-8 and all hexadecimal escape sequences replaced by a single byte containing the corresponding octet. If this process generates an invalid UTF-8 encoding, the results are undefined.

Note to Windows users: The encoding used for the filename argument of sqlite3_open() and sqlite3_open_v2() must be UTF-8, not whatever codepage is currently defined. Filenames containing international characters must be converted to UTF-8 prior to passing them into sqlite3_open() or sqlite3_open_v2().

Note to Windows Runtime users: The temporary directory must be set prior to calling sqlite3_open() or sqlite3_open_v2(). Otherwise, various features that require the use of temporary files may fail.

See also: sqlite3_temp_directory

The pre-update hook.

```
#if defined(SQLITE_ENABLE_PREUPDATE_HOOK)
void *sqlite3_preupdate_hook(
  sqlite3 *db,
  void(*xPreUpdate)(
    void *pCtx,                   /* Copy of third arg to preupdate_hook() */
    sqlite3 *db,                  /* Database handle */
    int op,                       /* SQLITE_UPDATE, DELETE or INSERT */
    char const *zDb,              /* Database name */
    char const *zName,            /* Table name */
    sqlite3_int64 iKey1,          /* Rowid of row about to be deleted/updated */
    sqlite3_int64 iKey2           /* New rowid value (for a rowid UPDATE) */
  ),
  void*
);
int sqlite3_preupdate_old(sqlite3 *, int, sqlite3_value **);
int sqlite3_preupdate_count(sqlite3 *);
int sqlite3_preupdate_depth(sqlite3 *);
int sqlite3_preupdate_new(sqlite3 *, int, sqlite3_value **);
#endif
```

These interfaces are only available if SQLite is compiled using the SQLITE_ENABLE_PREUPDATE_HOOK compile-time option.

The sqlite3_preupdate_hook() interface registers a callback function that is invoked prior to each INSERT, UPDATE, and DELETE operation on a database table. At most one preupdate hook may be registered at a time on a single database connection; each call to sqlite3_preupdate_hook() overrides the previous setting. The preupdate hook is disabled by invoking sqlite3_preupdate_hook() with a NULL pointer as the second parameter. The third parameter to sqlite3_preupdate_hook() is passed through as the first parameter to callbacks.

The preupdate hook only fires for changes to real database tables; the preupdate hook is not invoked for changes to virtual tables or to system tables like sqlite_master or sqlite_stat1.

The second parameter to the preupdate callback is a pointer to the database connection that registered the preupdate hook. The third parameter to the preupdate callback is one of the constants SQLITE_INSERT, SQLITE_DELETE, or SQLITE_UPDATE to identify the kind of update operation that is about to occur. The fourth parameter to the preupdate callback is the name of the database within the database connection that is being modified. This will be "main" for the main database or "temp" for TEMP tables or the name given after the AS keyword in the ATTACH statement for attached databases. The fifth parameter to the preupdate callback is the name of the table that is being modified.

For an UPDATE or DELETE operation on a rowid table, the sixth parameter passed to the preupdate callback is the initial rowid of the row being modified or deleted. For an INSERT operation on a rowid table, or any operation on a WITHOUT ROWID table, the value of the sixth parameter is undefined. For an INSERT or UPDATE on a rowid table the seventh parameter is the final rowid value of the row being inserted or updated. The value of the seventh parameter passed to the callback function is not defined for operations on WITHOUT ROWID tables, or for INSERT operations on rowid tables.

The sqlite3_preupdate_old(), sqlite3_preupdate_new(), sqlite3_preupdate_count(), and sqlite3_preupdate_depth() interfaces provide additional information about a preupdate event. These routines may only be called from within a preupdate callback. Invoking any of these routines from outside of a preupdate callback or with a database connection pointer that is different from the one supplied to the preupdate callback results in undefined and probably undesirable behavior.

The sqlite3_preupdate_count(D) interface returns the number of columns in the row that is being inserted, updated, or deleted.

The sqlite3_preupdate_old(D,N,P) interface writes into P a pointer to a protected sqlite3_value that contains the value of the Nth column of the table row before it is updated. The N parameter must be between 0 and one less than the number of columns or the behavior will be undefined. This must only be used within SQLITE_UPDATE and SQLITE_DELETE preupdate callbacks; if it is used by an SQLITE_INSERT callback then the behavior is undefined. The sqlite3_value that P points to will be destroyed when the preupdate callback returns.

The sqlite3_preupdate_new(D,N,P) interface writes into P a pointer to a protected sqlite3_value that contains the value of the Nth column of the table row after it is updated. The N parameter must be between 0 and one less than the number of columns or the behavior will be undefined. This must only be used within SQLITE_INSERT and SQLITE_UPDATE preupdate callbacks; if it is used by an SQLITE_DELETE callback then the behavior is undefined. The sqlite3_value that P points to will be destroyed when the preupdate callback returns.

The sqlite3_preupdate_depth(D) interface returns 0 if the preupdate callback was invoked as a result of a direct insert, update, or delete operation; or 1 for inserts, updates, or deletes invoked by top-level triggers; or 2 for changes resulting from triggers called by top-level triggers; and so forth.

See also: sqlite3_update_hook()

Tracing And Profiling Functions

```
void *sqlite3_trace(sqlite3*,
   void(*xTrace)(void*,const char*), void*);
void *sqlite3_profile(sqlite3*,
   void(*xProfile)(void*,const char*,sqlite3_uint64), void*);
```

These routines are deprecated. Use the sqlite3_trace_v2() interface instead of the routines described here.

These routines register callback functions that can be used for tracing and profiling the execution of SQL statements.

The callback function registered by sqlite3_trace() is invoked at various times when an SQL statement is being run by sqlite3_step(). The sqlite3_trace() callback is invoked with a UTF-8 rendering of the SQL statement text as the statement first begins executing. Additional sqlite3_trace() callbacks might occur as each triggered subprogram is entered. The callbacks for triggers contain a UTF-8 SQL comment that identifies the trigger.

The SQLITE_TRACE_SIZE_LIMIT compile-time option can be used to limit the length of bound parameter expansion in the output of sqlite3_trace().

The callback function registered by sqlite3_profile() is invoked as each SQL statement finishes. The profile callback contains the original statement text and an estimate of wall-clock time of how long that statement took to run. The profile callback time is in units of nanoseconds, however the current implementation is only capable of millisecond resolution so the six least significant digits in the time are meaningless. Future versions of SQLite might provide greater resolution on the profiler callback. Invoking either sqlite3_trace() or sqlite3_trace_v2() will cancel the profile callback.

Setting The Result Of An SQL Function

```
void sqlite3_result_blob(sqlite3_context*, const void*, int, void(*)(void*));
void sqlite3_result_blob64(sqlite3_context*,const void*,
                           sqlite3_uint64,void(*)(void*));
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_error(sqlite3_context*, const char*, int);
void sqlite3_result_error16(sqlite3_context*, const void*, int);
void sqlite3_result_error_toobig(sqlite3_context*);
void sqlite3_result_error_nomem(sqlite3_context*);
void sqlite3_result_error_code(sqlite3_context*, int);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, sqlite3_int64);
void sqlite3_result_null(sqlite3_context*);
void sqlite3_result_text(sqlite3_context*, const char*, int, void(*)(void*));
void sqlite3_result_text64(sqlite3_context*, const char*,sqlite3_uint64,
                           void(*)(void*), unsigned char encoding);
void sqlite3_result_text16(sqlite3_context*, const void*, int, void(*)(void*));
void sqlite3_result_text16le(sqlite3_context*, const void*, int,void(*)(void*));
void sqlite3_result_text16be(sqlite3_context*, const void*, int,void(*)(void*));
void sqlite3_result_value(sqlite3_context*, sqlite3_value*);
void sqlite3_result_pointer(sqlite3_context*, void*,const char*,void(*)(void*));
void sqlite3_result_zeroblob(sqlite3_context*, int n);
int sqlite3_result_zeroblob64(sqlite3_context*, sqlite3_uint64 n);
```

These routines are used by the xFunc or xFinal callbacks that implement SQL functions and aggregates. See sqlite3_create_function() and sqlite3_create_function16() for additional information.

These functions work very much like the parameter binding family of functions used to bind values to host parameters in prepared statements. Refer to the SQL parameter documentation for additional information.

The sqlite3_result_blob() interface sets the result from an application-defined function to be the BLOB whose content is pointed to by the second parameter and which is N bytes long where N is the third parameter.

The sqlite3_result_zeroblob(C,N) and sqlite3_result_zeroblob64(C,N) interfaces set the result of the application-defined function to be a BLOB containing all zero bytes and N bytes in size.

The sqlite3_result_double() interface sets the result from an application-defined function to be a floating point value specified by its 2nd argument.

The sqlite3_result_error() and sqlite3_result_error16() functions cause the implemented SQL function to throw an exception. SQLite uses the string pointed to by the 2nd parameter of sqlite3_result_error() or sqlite3_result_error16() as the text of an error message. SQLite interprets the error message string from sqlite3_result_error() as UTF-8. SQLite interprets the string from sqlite3_result_error16() as UTF-16 in native byte order. If the third parameter to sqlite3_result_error() or sqlite3_result_error16() is negative then SQLite takes as the error message all text up through the first zero character. If the third parameter to sqlite3_result_error() or sqlite3_result_error16() is non-negative then SQLite takes that many bytes (not characters) from the 2nd parameter as the error message. The sqlite3_result_error() and sqlite3_result_error16() routines make a private copy of the error message text before they return. Hence, the calling function can deallocate or modify the text after they return without harm. The sqlite3_result_error_code() function changes the error code returned by SQLite as a result of an error in a function. By default, the error code is SQLITE_ERROR. A subsequent call to sqlite3_result_error() or sqlite3_result_error16() resets the error code to SQLITE_ERROR.

The sqlite3_result_error_toobig() interface causes SQLite to throw an error indicating that a string or BLOB is too long to represent.

The sqlite3_result_error_nomem() interface causes SQLite to throw an error indicating that a memory allocation failed.

The sqlite3_result_int() interface sets the return value of the application-defined function to be the 32-bit signed integer value given in the 2nd argument. The sqlite3_result_int64() interface sets the return value of the application-defined function to be the 64-bit signed integer value given in the 2nd argument.

The sqlite3_result_null() interface sets the return value of the application-defined function to be NULL.

The sqlite3_result_text(), sqlite3_result_text16(), sqlite3_result_text16le(), and sqlite3_result_text16be() interfaces set the return value of the application-defined function to be a text string which is represented as UTF-8, UTF-16 native byte order, UTF-16 little endian, or UTF-16 big endian, respectively. The sqlite3_result_text64() interface sets the return value of an application-defined

function to be a text string in an encoding specified by the fifth (and last) parameter, which must be one of SQLITE_UTF8, SQLITE_UTF16, SQLITE_UTF16BE, or SQLITE_UTF16LE. SQLite takes the text result from the application from the 2nd parameter of the sqlite3_result_text* interfaces. If the 3rd parameter to the sqlite3_result_text* interfaces is negative, then SQLite takes result text from the 2nd parameter through the first zero character. If the 3rd parameter to the sqlite3_result_text* interfaces is non-negative, then as many bytes (not characters) of the text pointed to by the 2nd parameter are taken as the application-defined function result. If the 3rd parameter is non-negative, then it must be the byte offset into the string where the NUL terminator would appear if the string where NUL terminated. If any NUL characters occur in the string at a byte offset that is less than the value of the 3rd parameter, then the resulting string will contain embedded NULs and the result of expressions operating on strings with embedded NULs is undefined. If the 4th parameter to the sqlite3_result_text* interfaces or sqlite3_result_blob is a non-NULL pointer, then SQLite calls that function as the destructor on the text or BLOB result when it has finished using that result. If the 4th parameter to the sqlite3_result_text* interfaces or to sqlite3_result_blob is the special constant SQLITE_STATIC, then SQLite assumes that the text or BLOB result is in constant space and does not copy the content of the parameter nor call a destructor on the content when it has finished using that result. If the 4th parameter to the sqlite3_result_text* interfaces or sqlite3_result_blob is the special constant SQLITE_TRANSIENT then SQLite makes a copy of the result into space obtained from sqlite3_malloc() before it returns.

The sqlite3_result_value() interface sets the result of the application-defined function to be a copy of the unprotected sqlite3_value object specified by the 2nd parameter. The sqlite3_result_value() interface makes a copy of the sqlite3_value so that the sqlite3_value specified in the parameter may change or be deallocated after sqlite3_result_value() returns without harm. A protected sqlite3_value object may always be used where an unprotected sqlite3_value object is required, so either kind of sqlite3_value object can be used with this interface.

The sqlite3_result_pointer(C,P,T,D) interface sets the result to an SQL NULL value, just like sqlite3_result_null(C), except that it also associates the host-language pointer P or type T with that NULL value such that the pointer can be retrieved within an application-defined SQL function using sqlite3_value_pointer(). If the D parameter is not NULL, then it is a pointer to a destructor for the P parameter. SQLite invokes D with P as its only argument when SQLite is finished with P. The T parameter should be a static string and preferably a string literal. The sqlite3_result_pointer() routine is part of the pointer passing interface added for SQLite 3.20.0.

If these routines are called from within the different thread than the one containing the application-defined function that received the sqlite3_context pointer, the results are undefined.

SQLite Runtime Status

```
int sqlite3_status(int op, int *pCurrent, int *pHighwater, int resetFlag);
int sqlite3_status64(
  int op,
  sqlite3_int64 *pCurrent,
  sqlite3_int64 *pHighwater,
  int resetFlag
);
```

These interfaces are used to retrieve runtime status information about the performance of SQLite, and optionally to reset various highwater marks. The first argument is an integer code for the specific parameter to measure. Recognized integer codes are of the form SQLITE_STATUS_... The current value of the parameter is returned into *pCurrent. The highest recorded value is returned in *pHighwater. If the resetFlag is true, then the highest record value is reset after *pHighwater is written. Some parameters do not record the highest value. For those parameters nothing is written into *pHighwater and the resetFlag is ignored. Other parameters record only the highwater mark and not the current value. For these latter parameters nothing is written into *pCurrent.

The sqlite3_status() and sqlite3_status64() routines return SQLITE_OK on success and a non-zero error code on failure.

If either the current value or the highwater mark is too large to be represented by a 32-bit integer, then the values returned by sqlite3_status() are undefined.

See also: sqlite3_db_status()

Add Content To A Dynamic String

```
void sqlite3_str_appendf(sqlite3_str*, const char *zFormat, ...);
void sqlite3_str_vappendf(sqlite3_str*, const char *zFormat, va_list);
void sqlite3_str_append(sqlite3_str*, const char *zIn, int N);
void sqlite3_str_appendall(sqlite3_str*, const char *zIn);
void sqlite3_str_appendchar(sqlite3_str*, int N, char C);
void sqlite3_str_reset(sqlite3_str*);
```

These interfaces add content to an sqlite3_str object previously obtained from sqlite3_str_new().

The sqlite3_str_appendf(X,F,...) and sqlite3_str_vappendf(X,F,V) interfaces uses the built-in printf functionality of SQLite to append formatted text onto the end of sqlite3_str object X.

The sqlite3_str_append(X,S,N) method appends exactly N bytes from string S onto the end of the sqlite3_str object X. N must be non-negative. S must contain at least N non-zero bytes of content. To append a zero-terminated string in its entirety, use the sqlite3_str_appendall() method instead.

The sqlite3_str_appendall(X,S) method appends the complete content of zero-terminated string S onto the end of sqlite3_str object X.

The sqlite3_str_appendchar(X,N,C) method appends N copies of the single-byte character C onto the end of sqlite3_str object X. This method can be used, for example, to add whitespace indentation.

The sqlite3_str_reset(X) method resets the string under construction inside sqlite3_str object X back to zero bytes in length.

These methods do not return a result code. If an error occurs, that fact is recorded in the sqlite3_str object and can be recovered by a subsequent call to sqlite3_str_errcode(X).

Status Of A Dynamic String

```
int sqlite3_str_errcode(sqlite3_str*);
int sqlite3_str_length(sqlite3_str*);
char *sqlite3_str_value(sqlite3_str*);
```

These interfaces return the current status of an sqlite3_str object.

If any prior errors have occurred while constructing the dynamic string in sqlite3_str X, then the sqlite3_str_errcode(X) method will return an appropriate error code. The sqlite3_str_errcode(X) method returns SQLITE_NOMEM following any out-of-memory error, or SQLITE_TOOBIG if the size of the dynamic string exceeds SQLITE_MAX_LENGTH, or SQLITE_OK if there have been no errors.

The sqlite3_str_length(X) method returns the current length, in bytes, of the dynamic string under construction in sqlite3_str object X. The length returned by sqlite3_str_length(X) does not include the zero-termination byte.

The sqlite3_str_value(X) method returns a pointer to the current content of the dynamic string under construction in X. The value returned by sqlite3_str_value(X) is managed by the sqlite3_str object X and might be freed or altered by any subsequent method on the same sqlite3_str object. Applications must not used the pointer returned sqlite3_str_value(X) after any subsequent method call on the same object. Applications may change the content of the string returned by sqlite3_str_value(X) as long as they do not write into any bytes outside the range of 0 to sqlite3_str_length(X) and do not read or write any byte after any subsequent sqlite3_str method call.

String Comparison

```
int sqlite3_stricmp(const char *, const char *);
int sqlite3_strnicmp(const char *, const char *, int);
```

The sqlite3_stricmp() and sqlite3_strnicmp() APIs allow applications and extensions to compare the contents of two buffers containing UTF-8 strings in a case-independent fashion, using the same definition of "case independence" that SQLite uses internally when comparing identifiers.

Obtain Values For URI Parameters

```
const char *sqlite3_uri_parameter(const char *zFilename, const char *zParam);
int sqlite3_uri_boolean(const char *zFile, const char *zParam, int bDefault);
sqlite3_int64 sqlite3_uri_int64(const char*, const char*, sqlite3_int64);
const char *sqlite3_uri_key(const char *zFilename, int N);
```

These are utility routines, useful to custom VFS implementations, that check if a database file was a URI that contained a specific query parameter, and if so obtains the value of that query parameter.

If F is the database filename pointer passed into the xOpen() method of a VFS implementation or it is the return value of sqlite3_db_filename() and if P is the name of the query parameter, then sqlite3_uri_parameter(F,P) returns the value of the P parameter if it exists or a NULL pointer if P does not appear as a query parameter on F. If P is a query parameter of F and it has no explicit value, then sqlite3_uri_parameter(F,P) returns a pointer to an empty string.

The sqlite3_uri_boolean(F,P,B) routine assumes that P is a boolean parameter and returns true (1) or false (0) according to the value of P. The sqlite3_uri_boolean(F,P,B) routine returns true (1) if the value of query parameter P is one of "yes", "true", or "on" in any case or if the value begins with a non-zero number. The sqlite3_uri_boolean(F,P,B) routines returns false (0) if the value of query parameter P is one of "no", "false", or "off" in any case or if the value begins with a numeric zero. If P is not a query parameter on F or if the value of P does not match any of the above, then sqlite3_uri_boolean(F,P,B) returns (B!=0).

The sqlite3_uri_int64(F,P,D) routine converts the value of P into a 64-bit signed integer and returns that integer, or D if P does not exist. If the value of P is something other than an integer, then zero is returned.

The sqlite3_uri_key(F,N) returns a pointer to the name (not the value) of the N-th query parameter for filename F, or a NULL pointer if N is less than zero or greater than the number of query parameters minus 1. The N value is zero-based so N should be 0 to obtain the name of the first query parameter, 1 for the second parameter, and so forth.

If F is a NULL pointer, then sqlite3_uri_parameter(F,P) returns NULL and sqlite3_uri_boolean(F,P,B) returns B. If F is not a NULL pointer and is not a database file pathname pointer that the SQLite core passed into the xOpen VFS method, then the behavior of this routine is undefined and probably undesirable.

Beginning with SQLite version 3.31.0 (2020-01-22) the input F parameter can also be the name of a rollback journal file or WAL file in addition to the main database file. Prior to version 3.31.0, these routines would only work if F was the name of the main database file. When the F parameter is the name of the rollback journal or WAL file, it has access to all the same query parameters as were found on the main database file.

See the URI filename documentation for additional information.

Obtaining SQL Values

```
const void *sqlite3_value_blob(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
int sqlite3_value_int(sqlite3_value*);
sqlite3_int64 sqlite3_value_int64(sqlite3_value*);
void *sqlite3_value_pointer(sqlite3_value*, const char*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
const void *sqlite3_value_text16(sqlite3_value*);
const void *sqlite3_value_text16le(sqlite3_value*);
const void *sqlite3_value_text16be(sqlite3_value*);
int sqlite3_value_bytes(sqlite3_value*);
```

```
int sqlite3_value_bytes16(sqlite3_value*);
int sqlite3_value_type(sqlite3_value*);
int sqlite3_value_numeric_type(sqlite3_value*);
int sqlite3_value_nochange(sqlite3_value*);
int sqlite3_value_frombind(sqlite3_value*);
```

Summary:

sqlite3_value_blob	→	BLOB value
sqlite3_value_double	→	REAL value
sqlite3_value_int	→	32-bit INTEGER value
sqlite3_value_int64	→	64-bit INTEGER value
sqlite3_value_pointer	→	Pointer value
sqlite3_value_text	→	UTF-8 TEXT value
sqlite3_value_text16	→	UTF-16 TEXT value in the native byteorder
sqlite3_value_text16be	→	UTF-16be TEXT value
sqlite3_value_text16le	→	UTF-16le TEXT value
sqlite3_value_bytes	→	Size of a BLOB or a UTF-8 TEXT in bytes
sqlite3_value_bytes16	→	Size of UTF-16 TEXT in bytes
sqlite3_value_type	→	Default datatype of the value
sqlite3_value_numeric_type	→	Best numeric datatype of the value
sqlite3_value_nochange	→	True if the column is unchanged in an UPDATE against a virtual table.
sqlite3_value_frombind	→	True if value originated from a bound parameter

Details:

These routines extract type, size, and content information from protected sqlite3_value objects. Protected sqlite3_value objects are used to pass parameter information into the functions that implement application-defined SQL functions and virtual tables.

These routines work only with protected sqlite3_value objects. Any attempt to use these routines on an unprotected sqlite3_value is not threadsafe.

These routines work just like the corresponding column access functions except that these routines take a single protected sqlite3_value object pointer instead of a sqlite3_stmt* pointer and an integer column number.

The sqlite3_value_text16() interface extracts a UTF-16 string in the native byte-order of the host machine. The sqlite3_value_text16be() and sqlite3_value_text16le() interfaces extract UTF-16 strings as big-endian and little-endian respectively.

If sqlite3_value object V was initialized using sqlite3_bind_pointer(S,I,P,X,D) or sqlite3_result_pointer(C,P,X,D) and if X and Y are strings that compare equal according to strcmp(X,Y), then sqlite3_value_pointer(V,Y) will return the pointer P. Otherwise, sqlite3_value_pointer(V,Y) returns a NULL. The sqlite3_bind_pointer() routine is part of the pointer passing interface added for SQLite 3.20.0.

The sqlite3_value_type(V) interface returns the datatype code for the initial datatype of the sqlite3_value object V. The returned value is one of SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_TEXT, SQLITE_BLOB, or SQLITE_NULL. Other interfaces might change the datatype for an sqlite3_value object. For example, if the datatype is initially SQLITE_INTEGER and sqlite3_value_text(V) is called to extract a text value for that integer, then subsequent calls to sqlite3_value_type(V) might return SQLITE_TEXT. Whether or not a persistent internal datatype conversion occurs is undefined and may change from one release of SQLite to the next.

The sqlite3_value_numeric_type() interface attempts to apply numeric affinity to the value. This means that an attempt is made to convert the value to an integer or floating point. If such a conversion is possible without loss of information (in other words, if the value is a string that looks like a number) then the conversion is performed. Otherwise no conversion occurs. The datatype after conversion is returned.

Within the xUpdate method of a virtual table, the sqlite3_value_nochange(X) interface returns true if and only if the column corresponding to X is unchanged by the UPDATE operation that the xUpdate method call was invoked to implement and if and the prior xColumn method call that was invoked to extracted the value for that column returned without setting a result (probably because it queried sqlite3_vtab_nochange() and found that the column was unchanging). Within an xUpdate method, any value for which sqlite3_value_nochange(X) is true will in all other respects appear to be a NULL value. If sqlite3_value_nochange(X) is invoked anywhere other than within an xUpdate method call for an UPDATE statement, then the return value is arbitrary and meaningless.

The sqlite3_value_frombind(X) interface returns non-zero if the value X originated from one of the sqlite3_bind() interfaces. If X comes from an SQL literal value, or a table column, or an expression, then sqlite3_value_frombind(X) returns zero.

Please pay particular attention to the fact that the pointer returned from sqlite3_value_blob(), sqlite3_value_text(), or sqlite3_value_text16() can be invalidated by a subsequent call to sqlite3_value_bytes(), sqlite3_value_bytes16(), sqlite3_value_text(), or sqlite3_value_text16().

These routines must be called from the same thread as the SQL function that supplied the sqlite3_value* parameters.

As long as the input parameter is correct, these routines can only fail if an out-of-memory error occurs during a format conversion. Only the following subset of interfaces are subject to out-of-memory errors:

- sqlite3_value_blob()
- sqlite3_value_text()
- sqlite3_value_text16()
- sqlite3_value_text16le()
- sqlite3_value_text16be()
- sqlite3_value_bytes()
- sqlite3_value_bytes16()

If an out-of-memory error occurs, then the return value from these routines is the same as if the column had contained an SQL NULL value. Valid SQL NULL returns can be distinguished from out-of-memory errors by invoking the sqlite3_errcode() immediately after the suspect return value is obtained and before any other SQLite interface is called on the same database connection.

Copy And Free SQL Values

```
sqlite3_value *sqlite3_value_dup(const sqlite3_value*);
void sqlite3_value_free(sqlite3_value*);
```

The sqlite3_value_dup(V) interface makes a copy of the sqlite3_value object D and returns a pointer to that copy. The sqlite3_value returned is a protected sqlite3_value object even if the input is not. The sqlite3_value_dup(V) interface returns NULL if V is NULL or if a memory allocation fails.

The sqlite3_value_free(V) interface frees an sqlite3_value object previously obtained from sqlite3_value_dup(). If V is a NULL pointer then sqlite3_value_free(V) is a harmless no-op.

Virtual File System Objects

```
sqlite3_vfs *sqlite3_vfs_find(const char *zVfsName);
int sqlite3_vfs_register(sqlite3_vfs*, int makeDflt);
int sqlite3_vfs_unregister(sqlite3_vfs*);
```

A virtual filesystem (VFS) is an sqlite3_vfs object that SQLite uses to interact with the underlying operating system. Most SQLite builds come with a single default VFS that is appropriate for the host computer. New VFSes can be registered and existing VFSes can be unregistered. The following interfaces are provided.

The sqlite3_vfs_find() interface returns a pointer to a VFS given its name. Names are case sensitive. Names are zero-terminated UTF-8 strings. If there is no match, a NULL pointer is returned. If zVfsName is NULL then the default VFS is returned.

New VFSes are registered with sqlite3_vfs_register(). Each new VFS becomes the default VFS if the makeDflt flag is set. The same VFS can be registered multiple times without injury. To make an existing VFS into the default VFS, register it again with the makeDflt flag set. If two different VFSes with the same name are registered, the behavior is undefined. If a VFS is registered with a name that is NULL or an empty string, then the behavior is undefined.

Unregister a VFS with the sqlite3_vfs_unregister() interface. If the default VFS is unregistered, another VFS is chosen as the default. The choice for the new VFS is arbitrary.

Win32 Specific Interface

```
int sqlite3_win32_set_directory(
  unsigned long type, /* Identifier for directory being set or reset */
  void *zValue        /* New value for directory being set or reset */
);
int sqlite3_win32_set_directory8(unsigned long type, const char *zValue);
int sqlite3_win32_set_directory16(unsigned long type, const void *zValue);
```

These interfaces are available only on Windows. The sqlite3_win32_set_directory interface is used to set the value associated with the sqlite3_temp_directory or sqlite3_data_directory variable, to zValue, depending on the value of the type parameter. The zValue parameter should be NULL to cause the previous value to be freed via sqlite3_free; a non-NULL value will be copied into memory obtained from sqlite3_malloc prior to being used. The sqlite3_win32_set_directory interface returns SQLITE_OK to indicate success, SQLITE_ERROR if the type is unsupported, or SQLITE_NOMEM if memory could not be allocated. The value of the sqlite3_data_directory variable is intended to act as a replacement for the current directory on the sub-platforms of Win32 where that concept is not present, e.g. WinRT and UWP. The sqlite3_win32_set_directory8 and sqlite3_win32_set_directory16 interfaces behave exactly the same as the sqlite3_win32_set_directory interface except the string parameter must be UTF-8 or UTF-16, respectively.

Binding Values To Prepared Statements

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_blob64(sqlite3_stmt*, int, const void*, sqlite3_uint64,
                        void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, sqlite3_int64);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*,int,const char*,int,void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int, void(*)(void*));
int sqlite3_bind_text64(sqlite3_stmt*, int, const char*, sqlite3_uint64,
                         void(*)(void*), unsigned char encoding);
int sqlite3_bind_value(sqlite3_stmt*, int, const sqlite3_value*);
int sqlite3_bind_pointer(sqlite3_stmt*, int, void*, const char*,void(*)(void*));
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
int sqlite3_bind_zeroblob64(sqlite3_stmt*, int, sqlite3_uint64);
```

In the SQL statement text input to sqlite3_prepare_v2() and its variants, literals may be replaced by a parameter that matches one of following templates:

- ?
- ?NNN
- :VVV
- @VVV
- $VVV

In the templates above, NNN represents an integer literal, and VVV represents an alphanumeric identifier. The values of these parameters (also called "host parameter names" or "SQL parameters") can be set using the sqlite3_bind_*() routines defined here.

The first argument to the sqlite3_bind_*() routines is always a pointer to the sqlite3_stmt object returned from sqlite3_prepare_v2() or its variants.

The second argument is the index of the SQL parameter to be set. The leftmost SQL parameter has an index of 1. When the same named SQL parameter is used more than once, second and subsequent occurrences have the same index as the first occurrence. The index for named parameters can be looked up using the sqlite3_bind_parameter_index() API if desired. The index for "?NNN" parameters is the value of NNN. The NNN value must be between 1 and the sqlite3_limit() parameter SQLITE_LIMIT_VARIABLE_NUMBER (default value: 999).

The third argument is the value to bind to the parameter. If the third parameter to sqlite3_bind_text() or sqlite3_bind_text16() or sqlite3_bind_blob() is a NULL pointer then the fourth parameter is ignored and the end result is the same as sqlite3_bind_null().

In those routines that have a fourth argument, its value is the number of bytes in the parameter. To be clear: the value is the number of bytes in the value, not the number of characters. If the fourth parameter to sqlite3_bind_text() or sqlite3_bind_text16() is negative, then the length of the string is the number of bytes up to the first zero terminator. If the fourth parameter to sqlite3_bind_blob() is negative, then the behavior is undefined. If a non-negative fourth parameter is provided to sqlite3_bind_text() or sqlite3_bind_text16() or sqlite3_bind_text64() then that parameter must be the byte offset where the NUL terminator would occur assuming the string were NUL terminated. If any NUL characters occur at byte offsets less than the value of the fourth parameter then the resulting string value will contain embedded NULs. The result of expressions involving strings with embedded NULs is undefined.

The fifth argument to the BLOB and string binding interfaces is a destructor used to dispose of the BLOB or string after SQLite has finished with it. The destructor is called to dispose of the BLOB or string even if the call to the bind API fails, except the destructor is not called if the third parameter is a NULL pointer or the fourth parameter is negative. If the fifth argument is the special value SQLITE_STATIC, then SQLite assumes that the information is in static, unmanaged space and does not need to be freed. If the fifth argument has the value SQLITE_TRANSIENT, then SQLite makes its own private copy of the data immediately, before the sqlite3_bind_*() routine returns.

The sixth argument to sqlite3_bind_text64() must be one of SQLITE_UTF8, SQLITE_UTF16, SQLITE_UTF16BE, or SQLITE_UTF16LE to specify the encoding of the text in the third parameter. If the sixth argument to sqlite3_bind_text64() is not one of the allowed values shown above, or if the text encoding is different from the encoding specified by the sixth parameter, then the behavior is undefined.

The sqlite3_bind_zeroblob() routine binds a BLOB of length N that is filled with zeroes. A zeroblob uses a fixed amount of memory (just an integer to hold its size) while it is being processed. Zeroblobs are intended to serve as placeholders for BLOBs whose content is later written using incremental BLOB I/O routines. A negative value for the zeroblob results in a zero-length BLOB.

The sqlite3_bind_pointer(S,I,P,T,D) routine causes the I-th parameter in prepared statement S to have an SQL value of NULL, but to also be associated with the pointer P of type T. D is either a NULL pointer or a pointer to a destructor function for P. SQLite will invoke the destructor D with a single argument of P when it is finished using P. The T parameter should be a static string, preferably a string literal. The sqlite3_bind_pointer() routine is part of the pointer passing interface added for SQLite 3.20.0.

If any of the sqlite3_bind_*() routines are called with a NULL pointer for the prepared statement or with a prepared statement for which sqlite3_step() has been called more recently than sqlite3_reset(), then the call will return SQLITE_MISUSE. If any sqlite3_bind_() routine is passed a prepared statement that has been finalized, the result is undefined and probably harmful.

Bindings are not cleared by the sqlite3_reset() routine. Unbound parameters are interpreted as NULL.

The sqlite3_bind_* routines return SQLITE_OK on success or an error code if anything goes wrong. SQLITE_TOOBIG might be returned if the size of a string or BLOB exceeds limits imposed by sqlite3_limit(SQLITE_LIMIT_LENGTH) or SQLITE_MAX_LENGTH. SQLITE_RANGE is returned if the parameter index is out of range. SQLITE_NOMEM is returned if malloc() fails.

See also: sqlite3_bind_parameter_count(), sqlite3_bind_parameter_name(), and sqlite3_bind_parameter_index().

Compiling An SQL Statement

```
int sqlite3_prepare(
  sqlite3 *db,            /* Database handle */
  const char *zSql,       /* SQL statement, UTF-8 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
int sqlite3_prepare_v2(
  sqlite3 *db,            /* Database handle */
  const char *zSql,       /* SQL statement, UTF-8 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
int sqlite3_prepare_v3(
  sqlite3 *db,            /* Database handle */
  const char *zSql,       /* SQL statement, UTF-8 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  unsigned int prepFlags, /* Zero or more SQLITE_PREPARE_ flags */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
int sqlite3_prepare16(
  sqlite3 *db,            /* Database handle */
  const void *zSql,       /* SQL statement, UTF-16 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const void **pzTail     /* OUT: Pointer to unused portion of zSql */
);
int sqlite3_prepare16_v2(
```

```
  sqlite3 *db,            /* Database handle */
  const void *zSql,       /* SQL statement, UTF-16 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const void **pzTail     /* OUT: Pointer to unused portion of zSql */
);
int sqlite3_prepare16_v3(
  sqlite3 *db,            /* Database handle */
  const void *zSql,       /* SQL statement, UTF-16 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  unsigned int prepFlags, /* Zero or more SQLITE_PREPARE_ flags */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const void **pzTail     /* OUT: Pointer to unused portion of zSql */
);
```

To execute an SQL statement, it must first be compiled into a byte-code program using one of these routines. Or, in other words, these routines are constructors for the prepared statement object.

The preferred routine to use is sqlite3_prepare_v2(). The sqlite3_prepare() interface is legacy and should be avoided. sqlite3_prepare_v3() has an extra "prepFlags" option that is used for special purposes.

The use of the UTF-8 interfaces is preferred, as SQLite currently does all parsing using UTF-8. The UTF-16 interfaces are provided as a convenience. The UTF-16 interfaces work by converting the input text into UTF-8, then invoking the corresponding UTF-8 interface.

The first argument, "db", is a database connection obtained from a prior successful call to sqlite3_open(), sqlite3_open_v2() or sqlite3_open16(). The database connection must not have been closed.

The second argument, "zSql", is the statement to be compiled, encoded as either UTF-8 or UTF-16. The sqlite3_prepare(), sqlite3_prepare_v2(), and sqlite3_prepare_v3() interfaces use UTF-8, and sqlite3_prepare16(), sqlite3_prepare16_v2(), and sqlite3_prepare16_v3() use UTF-16.

If the nByte argument is negative, then zSql is read up to the first zero terminator. If nByte is positive, then it is the number of bytes read from zSql. If nByte is zero, then no prepared statement is generated. If the caller knows that the supplied string is nul-terminated, then there is a small performance advantage to passing an nByte parameter that is the number of bytes in the input string *including* the nul-terminator.

If pzTail is not NULL then *pzTail is made to point to the first byte past the end of the first SQL statement in zSql. These routines only compile the first statement in zSql, so *pzTail is left pointing to what remains uncompiled.

*ppStmt is left pointing to a compiled prepared statement that can be executed using sqlite3_step(). If there is an error, *ppStmt is set to NULL. If the input text contains no SQL (if the input is an empty string or a comment) then *ppStmt is set to NULL. The calling procedure is responsible for deleting the compiled SQL statement using sqlite3_finalize() after it has finished with it. ppStmt may not be NULL.

On success, the sqlite3_prepare() family of routines return SQLITE_OK; otherwise an error code is returned.

The sqlite3_prepare_v2(), sqlite3_prepare_v3(), sqlite3_prepare16_v2(), and sqlite3_prepare16_v3() interfaces are recommended for all new programs. The older interfaces (sqlite3_prepare() and sqlite3_prepare16()) are retained for backwards compatibility, but their use is discouraged. In the "vX" interfaces, the prepared statement that is returned (the sqlite3_stmt object) contains a copy of the original SQL text. This causes the sqlite3_step() interface to behave differently in three ways:

1. If the database schema changes, instead of returning SQLITE_SCHEMA as it always used to do, sqlite3_step() will automatically recompile the SQL statement and try to run it again. As many as SQLITE_MAX_SCHEMA_RETRY retries will occur before sqlite3_step() gives up and returns an error.

2. When an error occurs, sqlite3_step() will return one of the detailed error codes or extended error codes. The legacy behavior was that sqlite3_step() would only return a generic SQLITE_ERROR result code and the application would have to make a second call to sqlite3_reset() in order to find the underlying cause of the problem. With the "v2" prepare interfaces, the underlying reason for the error is returned immediately.

3. If the specific value bound to a host parameter in the WHERE clause might influence the choice of query plan for a statement, then the statement will be automatically recompiled, as if there had been a schema change, on the first sqlite3_step() call following any change to the bindings of that parameter. The specific value of a WHERE-clause parameter might influence the choice of query plan if the parameter is the left-hand side of a LIKE or GLOB operator or if the parameter is compared to an indexed column and the SQLITE_ENABLE_STAT4 compile-time option is enabled.

sqlite3_prepare_v3() differs from sqlite3_prepare_v2() only in having the extra prepFlags parameter, which is a bit array consisting of zero or more of the SQLITE_PREPARE_* flags. The sqlite3_prepare_v2() interface works exactly the same as sqlite3_prepare_v3() with a zero prepFlags parameter.

Compile-Time Authorization Callbacks

```
int sqlite3_set_authorizer(
  sqlite3*,
  int (*xAuth)(void*,int,const char*,const char*,const char*,const char*),
  void *pUserData
);
```

This routine registers an authorizer callback with a particular database connection, supplied in the first argument. The authorizer callback is invoked as SQL statements are being compiled by sqlite3_prepare() or its variants sqlite3_prepare_v2(), sqlite3_prepare_v3(), sqlite3_prepare16(), sqlite3_prepare16_v2(), and sqlite3_prepare16_v3(). At various points during the compilation process, as logic is being created to perform various actions, the authorizer callback is invoked to see if those actions are allowed. The authorizer callback should return SQLITE_OK to allow the action, SQLITE_IGNORE to disallow the specific action but allow the SQL statement to continue to be compiled, or SQLITE_DENY to cause the entire SQL statement to be rejected with an error. If the authorizer callback returns any value other than SQLITE_IGNORE, SQLITE_OK, or SQLITE_DENY then the sqlite3_prepare_v2() or equivalent call that triggered the authorizer will fail with an error message.

When the callback returns SQLITE_OK, that means the operation requested is ok. When the callback returns SQLITE_DENY, the sqlite3_prepare_v2() or equivalent call that triggered the authorizer will fail with an error message explaining that access is denied.

The first parameter to the authorizer callback is a copy of the third parameter to the sqlite3_set_authorizer() interface. The second parameter to the callback is an integer action code that specifies the particular action to be authorized. The third through sixth parameters to the callback are either NULL pointers or zero-terminated strings that contain additional details about the action to be authorized. Applications must always be prepared to encounter a NULL pointer in any of the third through the sixth parameters of the authorization callback.

If the action code is SQLITE_READ and the callback returns SQLITE_IGNORE then the prepared statement statement is constructed to substitute a NULL value in place of the table column that would have been read if SQLITE_OK had been returned. The SQLITE_IGNORE return can be used to deny an untrusted user access to individual columns of a table. When a table is referenced by a SELECT but no column values are extracted from that table (for example in a query like "SELECT count(*) FROM tab") then the SQLITE_READ authorizer callback is invoked once for that table with a column name that is an empty string. If the action code is SQLITE_DELETE and the callback returns SQLITE_IGNORE then the DELETE operation proceeds but the truncate optimization is disabled and all rows are deleted individually.

An authorizer is used when preparing SQL statements from an untrusted source, to ensure that the SQL statements do not try to access data they are not allowed to see, or that they do not try to execute malicious statements that damage the database. For example, an application may allow a user to enter arbitrary SQL queries for evaluation by a database. But the application does not want the user to be able to make arbitrary changes to the database. An authorizer could then be put in place while the user-entered SQL is being prepared that disallows everything except SELECT statements.

Applications that need to process SQL from untrusted sources might also consider lowering resource limits using sqlite3_limit() and limiting database size using the max_page_count PRAGMA in addition to using an authorizer.

Only a single authorizer can be in place on a database connection at a time. Each call to sqlite3_set_authorizer overrides the previous call. Disable the authorizer by installing a NULL callback. The authorizer is disabled by default.

The authorizer callback must not do anything that will modify the database connection that invoked the authorizer callback. Note that sqlite3_prepare_v2() and sqlite3_step() both modify their database connections for the meaning of "modify" in this paragraph.

When sqlite3_prepare_v2() is used to prepare a statement, the statement might be re-prepared during sqlite3_step() due to a schema change. Hence, the application should ensure that the correct authorizer callback remains in place during the sqlite3_step().

Note that the authorizer callback is invoked only during sqlite3_prepare() or its variants. Authorization is not performed during statement evaluation in sqlite3_step(), unless as stated in the previous paragraph, sqlite3_step() invokes sqlite3_prepare_v2() to reprepare a statement after a schema change.

Test For Auto-Commit Mode

```
int sqlite3_get_autocommit(sqlite3*);
```

The sqlite3_get_autocommit() interface returns non-zero or zero if the given database connection is or is not in autocommit mode, respectively. Autocommit mode is on by default. Autocommit mode is disabled by a BEGIN statement. Autocommit mode is re-enabled by a COMMIT or ROLLBACK.

If certain kinds of errors occur on a statement within a multi-statement transaction (errors including SQLITE_FULL, SQLITE_IOERR, SQLITE_NOMEM, SQLITE_BUSY, and SQLITE_INTERRUPT) then the transaction might be rolled back automatically. The only way to find out whether SQLite automatically rolled back the transaction after an error is to use this function.

If another thread changes the autocommit status of the database connection while this routine is running, then the return value is undefined.

Register A Callback To Handle SQLITE_BUSY Errors

```
int sqlite3_busy_handler(sqlite3*,int(*)(void*,int),void*);
```

The sqlite3_busy_handler(D,X,P) routine sets a callback function X that might be invoked with argument P whenever an attempt is made to access a database table associated with database connection D when another thread or process has the table locked. The sqlite3_busy_handler() interface is used to implement sqlite3_busy_timeout() and PRAGMA busy_timeout.

If the busy callback is NULL, then SQLITE_BUSY is returned immediately upon encountering the lock. If the busy callback is not NULL, then the callback might be invoked with two arguments.

The first argument to the busy handler is a copy of the void* pointer which is the third argument to sqlite3_busy_handler(). The second argument to the busy handler callback is the number of times that the busy handler has been invoked previously for the same locking event. If the busy callback returns 0, then no additional attempts are made to access the database and SQLITE_BUSY is returned to the application. If the callback returns non-zero, then another attempt is made to access the database and the cycle repeats.

The presence of a busy handler does not guarantee that it will be invoked when there is lock contention. If SQLite determines that invoking the busy handler could result in a deadlock, it will go ahead and return SQLITE_BUSY to the application instead of invoking the busy handler. Consider a scenario where one process is holding a read lock that it is trying to promote to a reserved lock and a second process is holding a reserved lock that it is trying to promote to an exclusive lock. The first process cannot proceed because it is blocked by the second and the second process cannot proceed because it is blocked by the first. If both processes invoke the busy handlers, neither will make any progress. Therefore, SQLite returns SQLITE_BUSY for the first process, hoping that this will induce the first process to release its read lock and allow the second process to proceed.

The default busy callback is NULL.

There can only be a single busy handler defined for each database connection. Setting a new busy handler clears any previously set handler. Note that calling sqlite3_busy_timeout() or evaluating PRAGMA busy_timeout=N will change the busy handler and thus clear any previously set busy handler.

The busy callback should not take any actions which modify the database connection that invoked the busy handler. In other words, the busy handler is not reentrant. Any such actions result in undefined behavior.

A busy handler must not close the database connection or prepared statement that invoked the busy handler.

Result Values From A Query

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
sqlite3_int64 sqlite3_column_int64(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
sqlite3_value *sqlite3_column_value(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
```

Summary:

sqlite3_column_blob	→	BLOB result
sqlite3_column_double	→	REAL result
sqlite3_column_int	→	32-bit INTEGER result
sqlite3_column_int64	→	64-bit INTEGER result
sqlite3_column_text	→	UTF-8 TEXT result
sqlite3_column_text16	→	UTF-16 TEXT result
sqlite3_column_value	→	The result as an unprotected sqlite3_value object.
sqlite3_column_bytes	→	Size of a BLOB or a UTF-8 TEXT result in bytes
sqlite3_column_bytes16	→	Size of UTF-16 TEXT in bytes
sqlite3_column_type	→	Default datatype of the result

Details:

These routines return information about a single column of the current result row of a query. In every case the first argument is a pointer to the prepared statement that is being evaluated (the sqlite3_stmt* that was returned from sqlite3_prepare_v2() or one of its variants) and the second argument is the index of the column for which information should be returned. The leftmost column of the result set has the index 0. The number of columns in the result can be determined using sqlite3_column_count().

If the SQL statement does not currently point to a valid row, or if the column index is out of range, the result is undefined. These routines may only be called when the most recent call to sqlite3_step() has returned SQLITE_ROW and neither sqlite3_reset() nor sqlite3_finalize() have been called subsequently. If any of these routines are called after sqlite3_reset() or sqlite3_finalize() or after sqlite3_step() has returned something other than SQLITE_ROW, the results are undefined. If sqlite3_step() or sqlite3_reset() or sqlite3_finalize() are called from a different thread while any of these routines are pending, then the results are undefined.

The first six interfaces (_blob, _double, _int, _int64, _text, and _text16) each return the value of a result column in a specific data format. If the result column is not initially in the requested format (for example, if the query returns an integer but the sqlite3_column_text() interface is used to extract the value) then an automatic type conversion is performed.

The sqlite3_column_type() routine returns the datatype code for the initial data type of the result column. The returned value is one of SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_TEXT, SQLITE_BLOB, or SQLITE_NULL. The return value of sqlite3_column_type() can be used to decide which of the first six interface should be used to extract the column value. The value returned by sqlite3_column_type() is only meaningful if no automatic type conversions have occurred for the value in question. After a type conversion, the result of calling sqlite3_column_type() is undefined, though harmless. Future versions of SQLite may change the behavior of sqlite3_column_type() following a type conversion.

If the result is a BLOB or a TEXT string, then the sqlite3_column_bytes() or sqlite3_column_bytes16() interfaces can be used to determine the size of that BLOB or string.

If the result is a BLOB or UTF-8 string then the sqlite3_column_bytes() routine returns the number of bytes in that BLOB or string. If the result is a UTF-16 string, then sqlite3_column_bytes() converts the string to UTF-8 and then returns the number of bytes. If the result is a numeric value then sqlite3_column_bytes() uses sqlite3_snprintf() to convert that value to a UTF-8 string and returns the number of bytes in that string. If the result is NULL, then sqlite3_column_bytes() returns zero.

If the result is a BLOB or UTF-16 string then the sqlite3_column_bytes16() routine returns the number of bytes in that BLOB or string. If the result is a UTF-8 string, then sqlite3_column_bytes16() converts the string to UTF-16 and then returns the number of bytes. If the result is a numeric value then sqlite3_column_bytes16() uses sqlite3_snprintf() to convert that value to a UTF-16 string and returns the number of bytes in that string. If the result is NULL, then sqlite3_column_bytes16() returns zero.

The values returned by sqlite3_column_bytes() and sqlite3_column_bytes16() do not include the zero terminators at the end of the string. For clarity: the values returned by sqlite3_column_bytes() and sqlite3_column_bytes16() are the number of bytes in the string, not the number of characters.

Strings returned by sqlite3_column_text() and sqlite3_column_text16(), even empty strings, are always zero-terminated. The return value from sqlite3_column_blob() for a zero-length BLOB is a NULL pointer.

Warning: The object returned by sqlite3_column_value() is an unprotected sqlite3_value object. In a multithreaded environment, an unprotected sqlite3_value object may only be used safely with sqlite3_bind_value() and sqlite3_result_value(). If the

unprotected sqlite3_value object returned by sqlite3_column_value() is used in any other way, including calls to routines like sqlite3_value_int(), sqlite3_value_text(), or sqlite3_value_bytes(), the behavior is not threadsafe. Hence, the sqlite3_column_value() interface is normally only useful within the implementation of application-defined SQL functions or virtual tables, not within top-level application code.

The these routines may attempt to convert the datatype of the result. For example, if the internal representation is FLOAT and a text result is requested, sqlite3_snprintf() is used internally to perform the conversion automatically. The following table details the conversions that are applied:

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0
NULL	FLOAT	Result is 0.0
NULL	TEXT	Result is a NULL pointer
NULL	BLOB	Result is a NULL pointer
INTEGER	FLOAT	Convert from integer to float
INTEGER	TEXT	ASCII rendering of the integer
INTEGER	BLOB	Same as INTEGER->TEXT
FLOAT	INTEGER	CAST to INTEGER
FLOAT	TEXT	ASCII rendering of the float
FLOAT	BLOB	CAST to BLOB
TEXT	INTEGER	CAST to INTEGER
TEXT	FLOAT	CAST to REAL
TEXT	BLOB	No change
BLOB	INTEGER	CAST to INTEGER
BLOB	FLOAT	CAST to REAL
BLOB	TEXT	Add a zero terminator if needed

Note that when type conversions occur, pointers returned by prior calls to sqlite3_column_blob(), sqlite3_column_text(), and/or sqlite3_column_text16() may be invalidated. Type conversions and pointer invalidations might occur in the following cases:

- The initial content is a BLOB and sqlite3_column_text() or sqlite3_column_text16() is called. A zero-terminator might need to be added to the string.
- The initial content is UTF-8 text and sqlite3_column_bytes16() or sqlite3_column_text16() is called. The content must be converted to UTF-16.
- The initial content is UTF-16 text and sqlite3_column_bytes() or sqlite3_column_text() is called. The content must be converted to UTF-8.

Conversions between UTF-16be and UTF-16le are always done in place and do not invalidate a prior pointer, though of course the content of the buffer that the prior pointer references will have been modified. Other kinds of conversion are done in place when it is possible, but sometimes they are not possible and in those cases prior pointers are invalidated.

The safest policy is to invoke these routines in one of the following ways:

- sqlite3_column_text() followed by sqlite3_column_bytes()
- sqlite3_column_blob() followed by sqlite3_column_bytes()
- sqlite3_column_text16() followed by sqlite3_column_bytes16()

In other words, you should call sqlite3_column_text(), sqlite3_column_blob(), or sqlite3_column_text16() first to force the result into the desired format, then invoke sqlite3_column_bytes() or sqlite3_column_bytes16() to find the size of the result. Do not mix calls to sqlite3_column_text() or sqlite3_column_blob() with calls to sqlite3_column_bytes16(), and do not mix calls to sqlite3_column_text16() with calls to sqlite3_column_bytes().

The pointers returned are valid until a type conversion occurs as described above, or until sqlite3_step() or sqlite3_reset() or sqlite3_finalize() is called. The memory space used to hold strings and BLOBs is freed automatically. Do not pass the pointers returned from sqlite3_column_blob(), sqlite3_column_text(), etc. into sqlite3_free().

As long as the input parameters are correct, these routines will only fail if an out-of-memory error occurs during a format conversion. Only the following subset of interfaces are subject to out-of-memory errors:

- sqlite3_column_blob()
- sqlite3_column_text()
- sqlite3_column_text16()
- sqlite3_column_bytes()
- sqlite3_column_bytes16()

If an out-of-memory error occurs, then the return value from these routines is the same as if the column had contained an SQL NULL value. Valid SQL NULL returns can be distinguished from out-of-memory errors by invoking the sqlite3_errcode() immediately after the suspect return value is obtained and before any other SQLite interface is called on the same database connection.

Low-Level Control Of Database Files

```
int sqlite3_file_control(sqlite3*, const char *zDbName, int op, void*);
```

The sqlite3_file_control() interface makes a direct call to the xFileControl method for the sqlite3_io_methods object associated with a particular database identified by the second argument. The name of the database is "main" for the main database or "temp" for the TEMP database, or the name that appears after the AS keyword for databases that are added using the ATTACH SQL

command. A NULL pointer can be used in place of "main" to refer to the main database file. The third and fourth parameters to this routine are passed directly through to the second and third parameters of the xFileControl method. The return value of the xFileControl method becomes the return value of this routine.

A few opcodes for sqlite3_file_control() are handled directly by the SQLite core and never invoke the sqlite3_io_methods.xFileControl method. The SQLITE_FCNTL_FILE_POINTER value for the op parameter causes a pointer to the underlying sqlite3_file object to be written into the space pointed to by the 4th parameter. The SQLITE_FCNTL_JOURNAL_POINTER works similarly except that it returns the sqlite3_file object associated with the journal file instead of the main database. The SQLITE_FCNTL_VFS_POINTER opcode returns a pointer to the underlying sqlite3_vfs object for the file. The SQLITE_FCNTL_DATA_VERSION returns the data version counter from the pager.

If the second parameter (zDbName) does not match the name of any open database file, then SQLITE_ERROR is returned. This error code is not remembered and will not be recalled by sqlite3_errcode() or sqlite3_errmsg(). The underlying xFileControl method might also return SQLITE_ERROR. There is no way to distinguish between an incorrect zDbName and an SQLITE_ERROR return from the underlying xFileControl method.

See also: file control opcodes

Create Or Redefine SQL Functions

```
int sqlite3_create_function(
  sqlite3 *db,
  const char *zFunctionName,
  int nArg,
  int eTextRep,
  void *pApp,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
int sqlite3_create_function16(
  sqlite3 *db,
  const void *zFunctionName,
  int nArg,
  int eTextRep,
  void *pApp,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
int sqlite3_create_function_v2(
  sqlite3 *db,
  const char *zFunctionName,
  int nArg,
  int eTextRep,
  void *pApp,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*),
  void(*xDestroy)(void*)
);
int sqlite3_create_window_function(
  sqlite3 *db,
  const char *zFunctionName,
  int nArg,
  int eTextRep,
  void *pApp,
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*),
  void (*xValue)(sqlite3_context*),
  void (*xInverse)(sqlite3_context*,int,sqlite3_value**),
  void(*xDestroy)(void*)
);
```

These functions (collectively known as "function creation routines") are used to add SQL functions or aggregates or to redefine the behavior of existing SQL functions or aggregates. The only differences between the three "sqlite3_create_function*" routines are the text encoding expected for the second parameter (the name of the function being created) and the presence or absence of a destructor callback for the application data pointer. Function sqlite3_create_window_function() is similar, but allows the user to supply the extra callback functions needed by aggregate window functions.

The first parameter is the database connection to which the SQL function is to be added. If an application uses more than one database connection then application-defined SQL functions must be added to each database connection separately.

The second parameter is the name of the SQL function to be created or redefined. The length of the name is limited to 255 bytes in a UTF-8 representation, exclusive of the zero-terminator. Note that the name length limit is in UTF-8 bytes, not characters nor UTF-16 bytes. Any attempt to create a function with a longer name will result in SQLITE_MISUSE being returned.

The third parameter (nArg) is the number of arguments that the SQL function or aggregate takes. If this parameter is -1, then the SQL function or aggregate may take any number of arguments between 0 and the limit set by sqlite3_limit(SQLITE_LIMIT_FUNCTION_ARG). If the third parameter is less than -1 or greater than 127 then the behavior is undefined.

The fourth parameter, eTextRep, specifies what text encoding this SQL function prefers for its parameters. The application should set this parameter to SQLITE_UTF16LE if the function implementation invokes sqlite3_value_text16le() on an input, or SQLITE_UTF16BE if the implementation invokes sqlite3_value_text16be() on an input, or SQLITE_UTF16 if sqlite3_value_text16() is used, or SQLITE_UTF8 otherwise. The same SQL function may be registered multiple times using different preferred text encodings, with different implementations for each encoding. When multiple implementations of the same function are available, SQLite will pick the one that involves the least amount of data conversion.

The fourth parameter may optionally be ORed with SQLITE_DETERMINISTIC to signal that the function will always return the same result given the same inputs within a single SQL statement. Most SQL functions are deterministic. The built-in random() SQL

function is an example of a function that is not deterministic. The SQLite query planner is able to perform additional optimizations on deterministic functions, so use of the SQLITE_DETERMINISTIC flag is recommended where possible.

The fourth parameter may also optionally include the SQLITE_DIRECTONLY flag, which if present prevents the function from being invoked from within VIEWs, TRIGGERs, CHECK constraints, generated column expressions, index expressions, or the WHERE clause of partial indexes.

For best security, the SQLITE_DIRECTONLY flag is recommended for all application-defined SQL functions that do not need to be used inside of triggers, view, CHECK constraints, or other elements of the database schema. This flags is especially recommended for SQL functions that have side effects or reveal internal application state. Without this flag, an attacker might be able to modify the schema of a database file to include invocations of the function with parameters chosen by the attacker, which the application will then execute when the database file is opened and read.

The fifth parameter is an arbitrary pointer. The implementation of the function can gain access to this pointer using sqlite3_user_data().

The sixth, seventh and eighth parameters passed to the three "sqlite3_create_function*" functions, xFunc, xStep and xFinal, are pointers to C-language functions that implement the SQL function or aggregate. A scalar SQL function requires an implementation of the xFunc callback only; NULL pointers must be passed as the xStep and xFinal parameters. An aggregate SQL function requires an implementation of xStep and xFinal and NULL pointer must be passed for xFunc. To delete an existing SQL function or aggregate, pass NULL pointers for all three function callbacks.

The sixth, seventh, eighth and ninth parameters (xStep, xFinal, xValue and xInverse) passed to sqlite3_create_window_function are pointers to C-language callbacks that implement the new function. xStep and xFinal must both be non-NULL. xValue and xInverse may either both be NULL, in which case a regular aggregate function is created, or must both be non-NULL, in which case the new function may be used as either an aggregate or aggregate window function. More details regarding the implementation of aggregate window functions are available here.

If the final parameter to sqlite3_create_function_v2() or sqlite3_create_window_function() is not NULL, then it is destructor for the application data pointer. The destructor is invoked when the function is deleted, either by being overloaded or when the database connection closes. The destructor is also invoked if the call to sqlite3_create_function_v2() fails. When the destructor callback is invoked, it is passed a single argument which is a copy of the application data pointer which was the fifth parameter to sqlite3_create_function_v2().

It is permitted to register multiple implementations of the same functions with the same name but with either differing numbers of arguments or differing preferred text encodings. SQLite will use the implementation that most closely matches the way in which the SQL function is used. A function implementation with a non-negative nArg parameter is a better match than a function implementation with a negative nArg. A function where the preferred text encoding matches the database encoding is a better match than a function where the encoding is different. A function where the encoding difference is between UTF16le and UTF16be is a closer match than a function where the encoding difference is between UTF8 and UTF16.

Built-in functions may be overloaded by new application-defined functions.

An application-defined function is permitted to call other SQLite interfaces. However, such calls must not close the database connection nor finalize or reset the prepared statement in which the function is running.
