The SQLite Bytecode Engine

1. Executive Summary

SQLite works by translating SQL statements into bytecode and then running that bytecode in a virtual machine. This document describes how the bytecode engine works.

This document describes SQLite internals. The information provided here is not needed for routine application development using SQLite. This document is intended for people who want to delve more deeply into the internal operation of SQLite.

The bytecode engine is not an API of SQLite. Details about the bytecode engine change from one release of SQLite to the next. Applications that use SQLite should not depend on any of the details found in this document.

2. Introduction

SQLite works by translating each SQL statement into bytecode and then running that bytecode. A prepared statement in SQLite is mostly just the bytecode needed to implement the corresponding SQL. The sqlite3_prepare_v2() interface is a compiler that translates SQL into bytecode. The sqlite3_step() interface is the virtual machine that runs the bytecode contained within the prepared statement.

The bytecode virtual machine is the heart of SQLite. Programmers who want to understand how SQLite operates internally must be familiar with the bytecode engine.

Historically, the bytecode engine in SQLite is called the "Virtual DataBase Engine" or "VDBE". This website uses the terms "bytecode engine", "VDBE", "virtual machine", and "bytecode virtual machine" interchangeably, as they all mean the same thing.

This article also uses the terms "bytecode program" and "prepared statement" interchangeably, as they are mostly the same thing.

2.1. VDBE Source Code

The source code to the bytecode engine is in the vdbe.c source file. The opcode definitions in this document are derived from comments in that source file. The source code comments are the canonical source of information about the bytecode engine. When in doubt, refer to the source code.

In addition to the primary vdbe.c source code file, there are other helper code files in the source tree, all of whose names begin with "vdbe" - short for "Virtual DataBase Engine".

Remember that the names and meanings of opcodes often change from one release of SQLite to the next. So if you are studying the EXPLAIN output from SQLite, you should reference the version of this document (or the vdbe.c source code) that corresponds to the version of SQLite that ran the EXPLAIN. Otherwise, the description of the opcodes may not be accurate. This document is derived from SQLite version 3.39.2 check-in 698edb77537b6 dated 2022-07-21.

2.2. Instruction Format

A bytecoded program in SQLite consists of one or more instructions. Each instruction has an opcode and five operands named P1, P2 P3, P4, and P5. The P1, P2, and P3 operands are 32-bit signed integers. These operands often refer to registers. For instructions that operate on b-tree cursors, the P1 operand is usually the cursor number. For jump instructions, P2 is usually the jump destination. P4 may be a 32-bit signed integer, a 64-bit signed integer, a 64-bit floating point value, a string literal, a Blob literal, a pointer to a collating sequence comparison function, or a pointer to the implementation of an application-defined SQL function, or various other things. P5 is an 16-bit unsigned integer normally used to hold flags. Bits of the P5 flag can sometimes affect the opcode in subtle ways. For example, if the SQLITE_NULLEQ (0x0080) bit of the P5 operand is set on the Eq opcode, then the NULL values compare equal to one another. Otherwise NULL values compare different from one another.

Some opcodes use all five operands. Some opcodes use one or two. Some opcodes use none of the operands.

The bytecode engine begins execution on instruction number 0. Execution continues until a Halt instruction is seen, or until the program counter becomes greater than the address of last instruction, or until there is an error. When the bytecode engine halts, all memory that it allocated is released and all database cursors it may have had open are closed. If the execution stopped due to an error, any pending transactions are terminated and changes made to the database are rolled back.

The ResultRow opcode causes the bytecode engine to pause and the corresponding sqlite3_step() call to return SQLITE_ROW. Before invoking ResultRow, the bytecoded program will have loaded the results for a single row of a query into a series of registers. C-language APIs such as sqlite3_column_int() or sqlite3_column_text() extract the query results from those registers. The bytecode engine resumes with the next instruction after the ResultRow on the next call to sqlite3_step().

2.3. Registers

Every bytecode program has a fixed (but potentially large) number of registers. A single register can hold a variety of objects:

- A NULL value
- A signed 64-bit integer
- An IEEE double-precision (64-bit) floating point number
- An arbitrary length strings
- An arbitrary length BLOB
- A RowSet object (See the RowSetAdd, RowSetRead, and RowSetTest opcodes)
- A Frame object (Used by subprograms - see Program)

A register can also be "Undefined" meaning that it holds no value at all. Undefined is different from NULL. Depending on compile-time options, an attempt to read an undefined register will usually cause a run-time error. If the code generator (sqlite3_prepare_v2()) ever generates a prepared statement that reads an Undefined register, that is a bug in the code generator.

Registers are numbered beginning with 0. Most opcodes refer to at least one register.

The number of registers in a single prepared statement is fixed at compile-time. The content of all registers is cleared when a prepared statement is reset or finalized.

The internal Mem object stores the value for a single register. The abstract sqlite3_value object that is exposed in the API is really just a Mem object or register.

2.4. B-Tree Cursors

A prepared statement can have zero or more open cursors. Each cursor is identified by a small integer, which is usually the P1 parameter to the opcode that uses the cursor. There can be multiple cursors open on the same index or table. All cursors operate independently, even cursors pointing to the same indices or tables. The only way for the virtual machine to interact with a database file is through a cursor. Instructions in the virtual machine can create a new cursor (ex: OpenRead or OpenWrite), read data from a cursor (Column), advance the cursor to the next entry in the table (ex: Next or Prev), and so forth. All cursors are automatically closed when the prepared statement is reset or finalized.

2.5. Subroutines, Coroutines, and Subprograms

The bytecode engine has no stack on which to store the return address of a subroutine. Return addresses must be stored in registers. Hence, bytecode subroutines are not reentrant.

The Gosub opcode stores the current program counter into register P1 then jumps to address P2. The Return opcode jumps to address P1+1. Hence, every subroutine is associated with two integers: the address of the entry point in the subroutine and the register number that is used to hold the return address.

The Yield opcode swaps the value of the program counter with the integer value in

register P1. This opcode is used to implement coroutines. Coroutines are often used to implement subqueries from which content is pulled on an as-needed basis.

Triggers need to be reentrant. Since bytecode subroutines are not reentrant a different mechanism must be used to implement triggers. Each trigger is implemented using a separate bytecode program with its own opcodes, program counter, and register set. The Program opcode invokes the trigger subprogram. The Program instruction allocates and initializes a fresh register set for each invocation of the subprogram, so subprograms can be reentrant and recursive. The Param opcode is used by subprograms to access content in registers of the calling bytecode program.

2.6. Self-Altering Code

Some opcodes are self-altering. For example, the Init opcode (which is always the first opcode in every bytecode program) increments its P1 operand. Subsequent Once opcodes compare their P1 operands to the P1 value for the Init opcode in order to determine if the one-time initialization code that follows should be skipped. Another example is the String8 opcode which converts its P4 operand from UTF-8 into the correct database string encoding, then converts itself into a String opcode.

3. Viewing The Bytecode

Every SQL statement that SQLite interprets results in a program for the virtual machine. But if the SQL statement begins with the keyword EXPLAIN the virtual machine will not execute the program. Instead, the instructions of the program will be returned, one instruction per row, like a query result. This feature is useful for debugging and for learning how the virtual machine operates. For example:

```
$ sqlite3 ex1.db
sqlite> explain delete from tbl1 where two<20;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     12    0                    00  Start at 12
1     Null           0     1     0                    00  r[1]=NULL
2     OpenWrite      0     2     0     3              00  root=2 iDb=0; tbl1
3     Rewind         0     10    0                    00
4       Column         0     1     2                    00  r[2]=tbl1.two
5       Ge             3     9     2     (BINARY)       51  if r[2]>=r[3] goto 9
6       Rowid          0     4     0                    00  r[4]=rowid
7       Once           0     8     0                    00
8       Delete         0     1     0     tbl1           02
9     Next           0     4     0                    01
10    Noop           0     0     0                    00
11    Halt           0     0     0                    00
12    Transaction    0     1     1     0              01  usesStmtJournal=0
13    TableLock      0     2     1     tbl1           00  iDb=0 root=2 write=1
14    Integer        20    3     0                    00  r[3]=20
15    Goto           0     1     0                    00
```

Any application can run an EXPLAIN query to get output similar to the above. However, indentation to show the loop structure is not generated by the SQLite core. The command-line shell contains extra logic for indenting loops. Also, the "comment"

column in the EXPLAIN output is only provided if SQLite is compiled with the -DSQLITE_ENABLE_EXPLAIN_COMMENTS options.

When SQLite is compiled with the SQLITE_DEBUG compile-time option, extra PRAGMA commands are available that are useful for debugging and for exploring the operation of the VDBE. For example the vdbe_trace pragma can be enabled to cause a disassembly of each VDBE opcode to be printed on standard output as the opcode is executed. These debugging pragmas include:

- PRAGMA parser_trace
- PRAGMA vdbe_addoptrace
- PRAGMA vdbe_debug
- PRAGMA vdbe_listing
- PRAGMA vdbe_trace

4. The Opcodes

There are currently 186 opcodes defined by the virtual machine. All currently defined opcodes are described in the table below. This table was generated automatically by scanning the source code from the file vdbe.c.

Remember: The VDBE opcodes are not part of the interface definition for SQLite. The number of opcodes and their names and meanings change from one release of SQLite to the next. The opcodes shown in the table below are valid for SQLite version 3.39.2 check-in 698edb77537b6 dated 2022-07-21.

Opcode Name	Description
Abortable	Verify that an Abort can happen. Assert if an Abort at this point might cause database corruption. This opcode only appears in debugging builds. An Abort is safe if either there have been no writes, or if there is an active statement journal.
Add	Add the value in register P1 to the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
AddImm	Add the constant P2 to the value in register P1. The result is always an integer. To force any register to be an integer, just add 0.
Affinity	Apply affinities to a range of P2 registers starting with P1. P4 is a string that is P2 characters long. The N-th

	character of the string indicates the column affinity that should be used for the N-th memory cell in the range.
AggFinal	P1 is the memory location that is the accumulator for an aggregate or window function. Execute the finalizer function for an aggregate and store the result in P1. P2 is the number of arguments that the step function takes and P4 is a pointer to the FuncDef for this function. The P2 argument is not used by this opcode. It is only there to disambiguate functions that can take varying numbers of arguments. The P4 argument is only needed for the case where the step function was not previously called.
AggInverse	Execute the xInverse function for an aggregate. The function has P5 arguments. P4 is a pointer to the FuncDef structure that specifies the function. Register P3 is the accumulator. The P5 arguments are taken from register P2 and its successors.
AggStep	Execute the xStep function for an aggregate. The function has P5 arguments. P4 is a pointer to the FuncDef structure that specifies the function. Register P3 is the accumulator. The P5 arguments are taken from register P2 and its successors.
AggStep1	Execute the xStep (if P1==0) or xInverse (if P1!=0) function for an aggregate. The function has P5 arguments. P4 is a pointer to the FuncDef structure that specifies the function. Register P3 is the accumulator. The P5 arguments are taken from register P2 and its successors. This opcode is initially coded as OP_AggStep0. On first evaluation, the FuncDef stored in P4 is converted into an sqlite3_context and the opcode is changed. In this way, the initialization of the sqlite3_context only happens once, instead of on each call to the step function.

AggValue	Invoke the xValue() function and store the result in register P3. P2 is the number of arguments that the step function takes and P4 is a pointer to the FuncDef for this function. The P2 argument is not used by this opcode. It is only there to disambiguate functions that can take varying numbers of arguments. The P4 argument is only needed for the case where the step function was not previously called.
And	Take the logical AND of the values in registers P1 and P2 and write the result into register P3. If either P1 or P2 is 0 (false) then the result is 0 even if the other input is NULL. A NULL and true or two NULLs give a NULL output.
AutoCommit	Set the database auto-commit flag to P1 (1 or 0). If P2 is true, roll back any currently active btree transactions. If there are any active VMs (apart from this one), then a ROLLBACK fails. A COMMIT fails if there are active writing VMs or active VMs that use shared cache. This instruction causes the VM to halt.
BeginSubrtn	Mark the beginning of a subroutine that can be entered in-line or that can be called using Gosub. The subroutine should be terminated by an Return instruction that has a P1 operand that is the same as the P2 operand to this opcode and that has P3 set to 1. If the subroutine is entered in-line, then the Return will simply fall through. But if the subroutine is entered using Gosub, then the Return will jump back to the first instruction after the Gosub. This routine works by loading a NULL into the P2 register. When the return address register contains a NULL, the Return instruction is a no-op that simply falls through to the next instruction (assuming that the Return opcode has a P3 value of 1). Thus if the subroutine is entered in-line, then the Return will cause in-line execution to continue. But if the subroutine is entered via Gosub, then the Return will cause a return to the address following the Gosub. This opcode is identical to Null. It has a different name

	only to make the byte code easier to read and verify.
BitAnd	Take the bit-wise AND of the values in register P1 and P2 and store the result in register P3. If either input is NULL, the result is NULL.
BitNot	Interpret the content of register P1 as an integer. Store the ones-complement of the P1 value into register P2. If P1 holds a NULL then store a NULL in P2.
BitOr	Take the bit-wise OR of the values in register P1 and P2 and store the result in register P3. If either input is NULL, the result is NULL.
Blob	P4 points to a blob of data P1 bytes long. Store this blob in register P2. If P4 is a NULL pointer, then construct a zero-filled blob that is P1 bytes long in P2.
Cast	Force the value in register P1 to be the type defined by P2. • P2=='A' → BLOB • P2=='B' → TEXT • P2=='C' → NUMERIC • P2=='D' → INTEGER • P2=='E' → REAL A NULL value is not changed by this routine. It remains NULL.
Checkpoint	Checkpoint database P1. This is a no-op if P1 is not currently in WAL mode. Parameter P2 is one of SQLITE_CHECKPOINT_PASSIVE, FULL, RESTART, or TRUNCATE. Write 1 or 0 into mem[P3] if the checkpoint returns SQLITE_BUSY or not, respectively. Write the number of pages in the WAL after the checkpoint into mem[P3+1] and the number of pages in the WAL that have been checkpointed after the checkpoint completes into mem[P3+2]. However on an error, mem[P3+1] and mem[P3+2] are initialized to -1.
Clear	Delete all contents of the database table or index whose root page in the database file is given by P1. But, unlike Destroy, do not remove the table or index from the database file. The table being clear is in the main database file if P2==0. If P2==1 then the table to be clear is in the

	auxiliary database file that is used to store tables create using CREATE TEMPORARY TABLE. If the P3 value is non-zero, then the row change count is incremented by the number of rows in the table being cleared. If P3 is greater than zero, then the value stored in register P3 is also incremented by the number of rows in the table being cleared. See also: Destroy
Close	Close a cursor previously opened as P1. If P1 is not currently open, this instruction is a no-op.
ClrSubtype	Clear the subtype from register P1.
CollSeq	P4 is a pointer to a CollSeq object. If the next call to a user function or aggregate calls sqlite3GetFuncCollSeq(), this collation sequence will be returned. This is used by the built-in min(), max() and nullif() functions. If P1 is not zero, then it is a register that a subsequent min() or max() aggregate will set to 1 if the current row is not the minimum or maximum. The P1 register is initialized to 0 by this instruction. The interface used by the implementation of the aforementioned functions to retrieve the collation sequence set by this opcode is not available publicly. Only built-in functions have access to this feature.
Column	Interpret the data that cursor P1 points to as a structure built using the MakeRecord instruction. (See the MakeRecord opcode for additional information about the format of the data.) Extract the P2-th column from this record. If there are less that (P2+1) values in the record, extract a NULL. The value extracted is stored in register P3. If the record contains fewer than P2 fields, then extract a NULL. Or, if the P4 argument is a P4_MEM use the value of the P4 argument as the result. If the OPFLAG_LENGTHARG and OPFLAG_TYPEOFARG bits are set on P5 then the result is guaranteed to only be used as the argument of a length() or typeof() function, respectively. The loading of large blobs can be

	skipped for length() and all content loading can be skipped for typeof().
ColumnsUsed	This opcode (which only exists if SQLite was compiled with SQLITE_ENABLE_COLUMN_USED_MASK) identifies which columns of the table or index for cursor P1 are used. P4 is a 64-bit integer (P4_INT64) in which the first 63 bits are one for each of the first 63 columns of the table or index that are actually used by the cursor. The high-order bit is set if any column after the 64th is used.
Compare	Compare two vectors of registers in reg(P1)..reg(P1+P3-1) (call this vector "A") and in reg(P2)..reg(P2+P3-1) ("B"). Save the result of the comparison for use by the next Jump instruct. If P5 has the OPFLAG_PERMUTE bit set, then the order of comparison is determined by the most recent Permutation operator. If the OPFLAG_PERMUTE bit is clear, then register are compared in sequential order. P4 is a KeyInfo structure that defines collating sequences and sort orders for the comparison. The permutation applies to registers only. The KeyInfo elements are used sequentially. The comparison is a sort comparison, so NULLs compare equal, NULLs are less than numbers, numbers are less than strings, and strings are less than blobs. This opcode must be immediately followed by an Jump opcode.
Concat	Add the text in register P1 onto the end of the text in register P2 and store the result in register P3. If either the P1 or P2 text are NULL then store NULL in P3. P3 = P2 || P1 It is illegal for P1 and P3 to be the same register. Sometimes, if P3 is the same register as P2, the implementation is able to avoid a memcpy().
Copy	Make a copy of registers P1..P1+P3 into registers P2..P2+P3. If the 0x0002 bit of P5 is set then also clear the

	MEM_Subtype flag in the destination. The 0x0001 bit of P5 indicates that this Copy opcode cannot be merged. The 0x0001 bit is used by the query planner and does not come into play during query execution. This instruction makes a deep copy of the value. A duplicate is made of any string or blob constant. See also SCopy.
Count	Store the number of entries (an integer value) in the table or index opened by cursor P1 in register P2. If P3==0, then an exact count is obtained, which involves visiting every btree page of the table. But if P3 is non-zero, an estimate is returned based on the current cursor position.
CreateBtree	Allocate a new b-tree in the main database file if P1==0 or in the TEMP database file if P1==1 or in an attached database if P1>1. The P3 argument must be 1 (BTREE_INTKEY) for a rowid table it must be 2 (BTREE_BLOBKEY) for an index or WITHOUT ROWID table. The root page number of the new b-tree is stored in register P2.
CursorHint	Provide a hint to cursor P1 that it only needs to return rows that satisfy the Expr in P4. TK_REGISTER terms in the P4 expression refer to values currently held in registers. TK_COLUMN terms in the P4 expression refer to columns in the b-tree to which cursor P1 is pointing.
CursorLock	Lock the btree to which cursor P1 is pointing so that the btree cannot be written by an other cursor.
CursorUnlock	Unlock the btree to which cursor P1 is pointing so that it can be written by other cursors.
DecrJumpZero	Register P1 must hold an integer. Decrement the value in P1 and jump to P2 if the new value is exactly zero.
DeferredSeek	P1 is an open index cursor and P3 is a cursor on the corresponding table. This opcode does a deferred seek of the P3 table cursor to the row that corresponds to the current row of P1. This is a deferred seek. Nothing actually happens until the cursor is used to read a record. That way, if no reads occur, no unnecessary I/O happens.

	P4 may be an array of integers (type P4_INTARRAY) containing one entry for each column in the P3 table. If array entry a(i) is non-zero, then reading column a(i)-1 from cursor P3 is equivalent to performing the deferred seek and then reading column i from P1. This information is stored in P3 and used to redirect reads against P3 over to P1, thus possibly avoiding the need to seek and read cursor P3.

Delete	Delete the record at which the P1 cursor is currently pointing. If the OPFLAG_SAVEPOSITION bit of the P5 parameter is set, then the cursor will be left pointing at either the next or the previous record in the table. If it is left pointing at the next record, then the next Next instruction will be a no-op. As a result, in this case it is ok to delete a record from within a Next loop. If OPFLAG_SAVEPOSITION bit of P5 is clear, then the cursor will be left in an undefined state. If the OPFLAG_AUXDELETE bit is set on P5, that indicates that this delete one of several associated with deleting a table row and all its associated index entries. Exactly one of those deletes is the "primary" delete. The others are all on OPFLAG_FORDELETE cursors or else are marked with the AUXDELETE flag. If the OPFLAG_NCHANGE flag of P2 (NB: P2 not P5) is set, then the row change count is incremented (otherwise not). P1 must not be pseudo-table. It has to be a real table with multiple rows. If P4 is not NULL then it points to a Table object. In this case either the update or pre-update hook, or both, may be invoked. The P1 cursor must have been positioned using NotFound prior to invoking this opcode in this case. Specifically, if one is configured, the pre-update hook is invoked if P4 is not NULL. The update-hook is invoked if one is configured, P4 is not NULL, and the OPFLAG_NCHANGE flag is set in P2. If the OPFLAG_ISUPDATE flag is set in P2, then P3 contains the address of the memory cell that contains the value that the rowid of the row will be set to by the update.
Destroy	Delete an entire database table or index whose root page in the database file is given by P1. The table being destroyed is in the main database file if P3==0. If P3==1 then the table to be clear is in the auxiliary database file that is used to store tables create using CREATE TEMPORARY TABLE. If AUTOVACUUM is enabled then it is possible that

	another root page might be moved into the newly deleted root page in order to keep all root pages contiguous at the beginning of the database. The former value of the root page that moved - its value before the move occurred - is stored in register P2. If no page movement was required (because the table being dropped was already the last one in the database) then a zero is stored in register P2. If AUTOVACUUM is disabled then a zero is stored in register P2. This opcode throws an error if there are any active reader VMs when it is invoked. This is done to avoid the difficulty associated with updating existing cursors when a root page is moved in an AUTOVACUUM database. This error is thrown even if the database is not an AUTOVACUUM db in order to avoid introducing an incompatibility between autovacuum and non-autovacuum modes. See also: Clear
Divide	Divide the value in register P1 by the value in register P2 and store the result in register P3 (P3=P2/P1). If the value in register P1 is zero, then the result is NULL. If either input is NULL, the result is NULL.
DropIndex	Remove the internal (in-memory) data structures that describe the index named P4 in database P1. This is called after an index is dropped from disk (using the Destroy opcode) in order to keep the internal representation of the schema consistent with what is on disk.
DropTable	Remove the internal (in-memory) data structures that describe the table named P4 in database P1. This is called after a table is dropped from disk (using the Destroy opcode) in order to keep the internal representation of the schema consistent with what is on disk.
DropTrigger	Remove the internal (in-memory) data structures that describe the trigger named P4 in database P1. This is called after a trigger is dropped from disk (using the Destroy opcode) in order to keep the internal representation of the schema consistent with what is on disk.

ElseEq	This opcode must follow an Lt or Gt comparison operator. There can be zero or more OP_ReleaseReg opcodes intervening, but no other opcodes are allowed to occur between this instruction and the previous Lt or Gt. If result of an Eq comparison on the same two operands as the prior Lt or Gt would have been true, then jump to P2. If the result of an Eq comparison on the two previous operands would have been false or NULL, then fall through.
EndCoroutine	The instruction at the address in register P1 is a Yield. Jump to the P2 parameter of that Yield. After the jump, register P1 becomes undefined. See also: InitCoroutine
Eq	Compare the values in register P1 and P3. If reg(P3)==reg(P1) then jump to address P2. The SQLITE_AFF_MASK portion of P5 must be an affinity character - SQLITE_AFF_TEXT, SQLITE_AFF_INTEGER, and so forth. An attempt is made to coerce both inputs according to this affinity before the comparison is made. If the SQLITE_AFF_MASK is 0x00, then numeric affinity is used. Note that the affinity conversions are stored back into the input registers P1 and P3. So this opcode can cause persistent changes to registers P1 and P3. Once any conversions have taken place, and neither value is NULL, the values are compared. If both values are blobs then memcmp() is used to determine the results of the comparison. If both values are text, then the appropriate collating function specified in P4 is used to do the comparison. If P4 is not specified then memcmp() is used to compare text string. If both values are numeric, then a numeric comparison is used. If the two values are of different types, then numbers are considered less than strings and strings are considered less than blobs. If SQLITE_NULLEQ is set in P5 then the result of comparison is always either true or false and is never NULL. If both operands are NULL then the result of comparison is true. If either operand is NULL then the result is false. If neither operand is NULL the result is the same as it would be if the SQLITE_NULLEQ flag

	were omitted from P5. This opcode saves the result of comparison for use by the new Jump opcode.
Expire	Cause precompiled statements to expire. When an expired statement is executed using sqlite3_step() it will either automatically reprepare itself (if it was originally created using sqlite3_prepare_v2()) or it will fail with SQLITE_SCHEMA. If P1 is 0, then all SQL statements become expired. If P1 is non-zero, then only the currently executing statement is expired. If P2 is 0, then SQL statements are expired immediately. If P2 is 1, then running SQL statements are allowed to continue to run to completion. The P2==1 case occurs when a CREATE INDEX or similar schema change happens that might help the statement run faster but which does not affect the correctness of operation.
Filter	Compute a hash on the key contained in the P4 registers starting with r[P3]. Check to see if that hash is found in the bloom filter hosted by register P1. If it is not present then maybe jump to P2. Otherwise fall through. False negatives are harmless. It is always safe to fall through, even if the value is in the bloom filter. A false negative causes more CPU cycles to be used, but it should still yield the correct answer. However, an incorrect answer may well arise from a false positive - if the jump is taken when it should fall through.
FilterAdd	Compute a hash on the P4 registers starting with r[P3] and add that hash to the bloom filter contained in r[P1].
FinishSeek	If cursor P1 was previously moved via DeferredSeek, complete that seek operation now, without further delay. If the cursor seek has already occurred, this instruction is a no-op.
FkCheck	Halt with an SQLITE_CONSTRAINT error if there are any unresolved foreign key constraint violations. If there are no foreign key constraint violations, this is a no-op.

	FK constraint violations are also checked when the prepared statement exits. This opcode is used to raise foreign key constraint errors prior to returning results such as a row change count or the result of a RETURNING clause.
FkCounter	Increment a "constraint counter" by P2 (P2 may be negative or positive). If P1 is non-zero, the database constraint counter is incremented (deferred foreign key constraints). Otherwise, if P1 is zero, the statement counter is incremented (immediate foreign key constraints).
FkIfZero	This opcode tests if a foreign key constraint-counter is currently zero. If so, jump to instruction P2. Otherwise, fall through to the next instruction. If P1 is non-zero, then the jump is taken if the database constraint-counter is zero (the one that counts deferred constraint violations). If P1 is zero, the jump is taken if the statement constraint-counter is zero (immediate foreign key constraint violations).
Found	If P4==0 then register P3 holds a blob constructed by MakeRecord. If P4>0 then register P3 is the first of P4 registers that form an unpacked record. Cursor P1 is on an index btree. If the record identified by P3 and P4 is a prefix of any entry in P1 then a jump is made to P2 and P1 is left pointing at the matching entry. This operation leaves the cursor in a state where it can be advanced in the forward direction. The Next instruction will work, but not the Prev instruction. See also: NotFound, NoConflict, NotExists. SeekGe
Function	Invoke a user function (P4 is a pointer to an sqlite3_context object that contains a pointer to the function to be run) with arguments taken from register P2 and successors. The number of arguments is in the sqlite3_context object that P4 points to. The result of the function is stored in register P3. Register P3 must not be one of the function inputs. P1 is a 32-bit bitmask indicating whether or not each

	argument to the function was determined to be constant at compile time. If the first argument was constant then bit 0 of P1 is set. This is used to determine whether meta data associated with a user function argument using the sqlite3_set_auxdata() API may be safely retained until the next invocation of this opcode. See also: AggStep, AggFinal, PureFunc
Ge	This works just like the Lt opcode except that the jump is taken if the content of register P3 is greater than or equal to the content of register P1. See the Lt opcode for additional information.
Gosub	Write the current address onto register P1 and then jump to address P2.
Goto	An unconditional jump to address P2. The next instruction executed will be the one at index P2 from the beginning of the program. The P1 parameter is not actually used by this opcode. However, it is sometimes set to 1 instead of 0 as a hint to the command-line shell that this Goto is the bottom of a loop and that the lines from P2 down to the current line should be indented for EXPLAIN output.
Gt	This works just like the Lt opcode except that the jump is taken if the content of register P3 is greater than the content of register P1. See the Lt opcode for additional information.
Halt	Exit immediately. All open cursors, etc are closed automatically. P1 is the result code returned by sqlite3_exec(), sqlite3_reset(), or sqlite3_finalize(). For a normal halt, this should be SQLITE_OK (0). For errors, it can be some other value. If P1!=0 then P2 will determine whether or not to rollback the current transaction. Do not rollback if P2==OE_Fail. Do the rollback if P2==OE_Rollback. If P2==OE_Abort, then back out all changes that have occurred during this execution of the VDBE, but do not rollback the transaction. If P4 is not null then it is an error message string. P5 is a value between 0 and 4, inclusive, that modifies

	the P4 string. 0: (no change) 1: NOT NULL contraint failed: P4 2: UNIQUE constraint failed: P4 3: CHECK constraint failed: P4 4: FOREIGN KEY constraint failed: P4 If P5 is not zero and P4 is NULL, then everything after the ":" is omitted. There is an implied "Halt 0 0 0" instruction inserted at the very end of every program. So a jump past the last instruction of the program is the same as executing Halt.
HaltIfNull	Check the value in register P3. If it is NULL then Halt using parameter P1, P2, and P4 as if this were a Halt instruction. If the value in register P3 is not NULL, then this routine is a no-op. The P5 parameter should be 1.
IdxDelete	The content of P3 registers starting at register P2 form an unpacked index key. This opcode removes that entry from the index opened by cursor P1. If P5 is not zero, then raise an SQLITE_CORRUPT_INDEX error if no matching index entry is found. This happens when running an UPDATE or DELETE statement and the index entry to be updated or deleted is not found. For some uses of IdxDelete (example: the EXCEPT operator) it does not matter that no matching entry is found. For those cases, P5 is zero. Also, do not raise this (self-correcting and non-critical) error if in writable_schema mode.
IdxGE	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID fields at the end. If the P1 index entry is greater than or equal to the key value then jump to P2. Otherwise fall through to the next instruction.
IdxGT	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID fields at the end.

	If the P1 index entry is greater than the key value then jump to P2. Otherwise fall through to the next instruction.
IdxInsert	Register P2 holds an SQL index key made using the MakeRecord instructions. This opcode writes that key into the index P1. Data for the entry is nil. If P4 is not zero, then it is the number of values in the unpacked key of reg(P2). In that case, P3 is the index of the first register for the unpacked key. The availability of the unpacked key can sometimes be an optimization. If P5 has the OPFLAG_APPEND bit set, that is a hint to the b-tree layer that this insert is likely to be an append. If P5 has the OPFLAG_NCHANGE bit set, then the change counter is incremented by this instruction. If the OPFLAG_NCHANGE bit is clear, then the change counter is unchanged. If the OPFLAG_USESEEKRESULT flag of P5 is set, the implementation might run faster by avoiding an unnecessary seek on cursor P1. However, the OPFLAG_USESEEKRESULT flag must only be set if there have been no prior seeks on the cursor or if the most recent seek used a key equivalent to P2. This instruction only works for indices. The equivalent instruction for tables is Insert.
IdxLE	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY or ROWID. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID on the P1 index. If the P1 index entry is less than or equal to the key value then jump to P2. Otherwise fall through to the next instruction.
IdxLT	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY or ROWID. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID on the P1 index.

	If the P1 index entry is less than the key value then jump to P2. Otherwise fall through to the next instruction.
IdxRowid	Write into register P2 an integer which is the last entry in the record at the end of the index key pointed to by cursor P1. This integer should be the rowid of the table entry to which this index entry points. See also: Rowid, MakeRecord.
If	Jump to P2 if the value in register P1 is true. The value is considered true if it is numeric and non-zero. If the value in P1 is NULL then take the jump if and only if P3 is non-zero.

IfNoHope	Register P3 is the first of P4 registers that form an unpacked record. Cursor P1 is an index btree. P2 is a jump destination. In other words, the operands to this opcode are the same as the operands to NotFound and IdxGT. This opcode is an optimization attempt only. If this opcode always falls through, the correct answer is still obtained, but extra works is performed. A value of N in the seekHit flag of cursor P1 means that there exists a key P3:N that will match some record in the index. We want to know if it is possible for a record P3:P4 to match some record in the index. If it is not possible, we can skips some work. So if seekHit is less than P4, attempt to find out if a match is possible by running NotFound. This opcode is used in IN clause processing for a multi-column key. If an IN clause is attached to an element of the key other than the left-most element, and if there are no matches on the most recent seek over the whole key, then it might be that one of the key element to the left is prohibiting a match, and hence there is "no hope" of any match regardless of how many IN clause elements are checked. In such a case, we abandon the IN clause search early, using this opcode. The opcode name comes from the fact that the jump is taken if there is "no hope" of achieving a match. See also: NotFound, SeekHit
IfNot	Jump to P2 if the value in register P1 is False. The value is considered false if it has a numeric value of zero. If the value in P1 is NULL then take the jump if and only if P3 is non-zero.
IfNotOpen	If cursor P1 is not open, jump to instruction P2. Otherwise, fall through.
IfNotZero	Register P1 must contain an integer. If the content of register P1 is initially greater than zero, then decrement the value in register P1. If it is non-zero (negative or positive) and then also jump to P2. If register P1 is initially zero, leave it unchanged and fall through.
IfNullRow	Check the cursor P1 to see if it is currently pointing at a NULL row. If it is, then set register P3 to NULL and jump immediately to P2. If P1 is not on a NULL row, then fall

	through without making any changes.
IfPos	Register P1 must contain an integer. If the value of register P1 is 1 or greater, subtract P3 from the value in P1 and jump to P2. If the initial value of register P1 is less than 1, then the value is unchanged and control passes through to the next instruction.
IfSmaller	Estimate the number of rows in the table P1. Jump to P2 if that estimate is less than approximately 2**(0.1*P3).
IncrVacuum	Perform a single step of the incremental vacuum procedure on the P1 database. If the vacuum has finished, jump to instruction P2. Otherwise, fall through to the next instruction.
Init	Programs contain a single instance of this opcode as the very first opcode. If tracing is enabled (by the sqlite3_trace()) interface, then the UTF-8 string contained in P4 is emitted on the trace callback. Or if P4 is blank, use the string returned by sqlite3_sql(). If P2 is not zero, jump to instruction P2. Increment the value of P1 so that Once opcodes will jump the first time they are evaluated for this run. If P3 is not zero, then it is an address to jump to if an SQLITE_CORRUPT error is encountered.
InitCoroutine	Set up register P1 so that it will Yield to the coroutine located at address P3. If P2!=0 then the coroutine implementation immediately follows this opcode. So jump over the coroutine implementation to address P2. See also: EndCoroutine
Insert	Write an entry into the table of cursor P1. A new entry is created if it doesn't already exist or the data for an existing entry is overwritten. The data is the value MEM_Blob stored in register number P2. The key is

	stored in register P3. The key must be a MEM_Int. If the OPFLAG_NCHANGE flag of P5 is set, then the row change count is incremented (otherwise not). If the OPFLAG_LASTROWID flag of P5 is set, then rowid is stored for subsequent return by the sqlite3_last_insert_rowid() function (otherwise it is unmodified). If the OPFLAG_USESEEKRESULT flag of P5 is set, the implementation might run faster by avoiding an unnecessary seek on cursor P1. However, the OPFLAG_USESEEKRESULT flag must only be set if there have been no prior seeks on the cursor or if the most recent seek used a key equal to P3. If the OPFLAG_ISUPDATE flag is set, then this opcode is part of an UPDATE operation. Otherwise (if the flag is clear) then this opcode is part of an INSERT operation. The difference is only important to the update hook. Parameter P4 may point to a Table structure, or may be NULL. If it is not NULL, then the update-hook (sqlite3.xUpdateCallback) is invoked following a successful insert. (WARNING/TODO: If P1 is a pseudo-cursor and P2 is dynamically allocated, then ownership of P2 is transferred to the pseudo-cursor and register P2 becomes ephemeral. If the cursor is changed, the value of register P2 will then change. Make sure this does not cause any problems.) This instruction only works on tables. The equivalent instruction for indices is IdxInsert.
Int64	P4 is a pointer to a 64-bit integer value. Write that value into register P2.
IntCopy	Transfer the integer value held in register P1 into register P2. This is an optimized version of SCopy that works only for integer values.
Integer	The 32-bit integer value P1 is written into register P2.
IntegrityCk	Do an analysis of the currently open database. Store in register P1 the text of an error message describing any

	problems. If no problems are found, store a NULL in register P1. The register P3 contains one less than the maximum number of allowed errors. At most reg(P3) errors will be reported. In other words, the analysis stops as soon as reg(P1) errors are seen. Reg(P1) is updated with the number of errors remaining. The root page numbers of all tables in the database are integers stored in P4_INTARRAY argument. If P5 is not zero, the check is done on the auxiliary database file, not the main database file. This opcode is used to implement the integrity_check pragma.
IsNull	Jump to P2 if the value in register P1 is NULL.
IsNullOrType	Jump to P2 if the value in register P1 is NULL or has a datatype P3. P3 is an integer which should be one of SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_BLOB, SQLITE_NULL, or SQLITE_TEXT.
IsTrue	This opcode implements the IS TRUE, IS FALSE, IS NOT TRUE, and IS NOT FALSE operators. Interpret the value in register P1 as a boolean value. Store that boolean (a 0 or 1) in register P2. Or if the value in register P1 is NULL, then the P3 is stored in register P2. Invert the answer if P4 is 1. The logic is summarized like this: • If P3==0 and P4==0 then r[P2] := r[P1] IS TRUE • If P3==1 and P4==1 then r[P2] := r[P1] IS FALSE • If P3==0 and P4==1 then r[P2] := r[P1] IS NOT TRUE • If P3==1 and P4==0 then r[P2] := r[P1] IS NOT FALSE
JournalMode	Change the journal mode of database P1 to P3. P3 must be one of the PAGER_JOURNALMODE_XXX values. If changing between the various rollback modes (delete, truncate, persist, off and memory), this is a simple operation. No IO is required. If changing into or out of WAL mode the procedure is

	more complicated. Write a string containing the final journal-mode to register P2.
Jump	Jump to the instruction at address P1, P2, or P3 depending on whether in the most recent Compare instruction the P1 vector was less than equal to, or greater than the P2 vector, respectively. This opcode must immediately follow an Compare opcode.
Last	The next use of the Rowid or Column or Prev instruction for P1 will refer to the last entry in the database table or index. If the table or index is empty and P2>0, then jump immediately to P2. If P2 is 0 or if the table or index is not empty, fall through to the following instruction. This opcode leaves the cursor configured to move in reverse order, from the end toward the beginning. In other words, the cursor is configured to use Prev, not Next.
Le	This works just like the Lt opcode except that the jump is taken if the content of register P3 is less than or equal to the content of register P1. See the Lt opcode for additional information.
LoadAnalysis	Read the sqlite_stat1 table for database P1 and load the content of that table into the internal index hash table. This will cause the analysis to be used when preparing all subsequent queries.
Lt	Compare the values in register P1 and P3. If reg(P3) <reg(P1) then jump to address P2. If the SQLITE_JUMPIFNULL bit of P5 is set and either reg(P1) or reg(P3) is NULL then the take the jump. If the SQLITE_JUMPIFNULL bit is clear then fall through if either operand is NULL. The SQLITE_AFF_MASK portion of P5 must be an affinity character - SQLITE_AFF_TEXT, SQLITE_AFF_INTEGER, and so forth. An attempt is made to coerce both inputs according to this affinity before the comparison is made. If the SQLITE_AFF_MASK is 0x00, then numeric affinity

	is used. Note that the affinity conversions are stored back into the input registers P1 and P3. So this opcode can cause persistent changes to registers P1 and P3. Once any conversions have taken place, and neither value is NULL, the values are compared. If both values are blobs then memcmp() is used to determine the results of the comparison. If both values are text, then the appropriate collating function specified in P4 is used to do the comparison. If P4 is not specified then memcmp() is used to compare text string. If both values are numeric, then a numeric comparison is used. If the two values are of different types, then numbers are considered less than strings and strings are considered less than blobs. This opcode saves the result of comparison for use by the new Jump opcode.
MakeRecord	Convert P2 registers beginning with P1 into the record format use as a data record in a database table or as a key in an index. The Column opcode can decode the record later. P4 may be a string that is P2 characters long. The N-th character of the string indicates the column affinity that should be used for the N-th field of the index key. The mapping from character to affinity is given by the SQLITE_AFF_ macros defined in sqliteInt.h. If P4 is NULL then all index fields have the affinity BLOB. The meaning of P5 depends on whether or not the SQLITE_ENABLE_NULL_TRIM compile-time option is enabled: * If SQLITE_ENABLE_NULL_TRIM is enabled, then the P5 is the index of the right-most table that can be null-trimmed. * If SQLITE_ENABLE_NULL_TRIM is omitted, then P5 has the value OPFLAG_NOCHNG_MAGIC if the MakeRecord opcode is allowed to accept no-change records with serial_type 10. This value is only used inside an assert() and does not affect the end result.
MaxPgcnt	Try to set the maximum page count for database P1 to the value in P3. Do not let the maximum page count fall

	below the current page count and do not change the maximum page count value if P3==0. Store the maximum page count after the change in register P2.
MemMax	P1 is a register in the root frame of this VM (the root frame is different from the current frame if this instruction is being executed within a sub-program). Set the value of register P1 to the maximum of its current value and the value in register P2. This instruction throws an error if the memory cell is not initially an integer.
Move	Move the P3 values in register P1..P1+P3-1 over into registers P2..P2+P3-1. Registers P1..P1+P3-1 are left holding a NULL. It is an error for register ranges P1..P1+P3-1 and P2..P2+P3-1 to overlap. It is an error for P3 to be less than 1.
Multiply	Multiply the value in register P1 by the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
MustBeInt	Force the value in register P1 to be an integer. If the value in P1 is not an integer and cannot be converted into an integer without data loss, then jump immediately to P2, or if P2==0 raise an SQLITE_MISMATCH exception.
Ne	This works just like the Eq opcode except that the jump is taken if the operands in registers P1 and P3 are not equal. See the Eq opcode for additional information.
NewRowid	Get a new integer record number (a.k.a "rowid") used as the key to a table. The record number is not previously used as a key in the database table that cursor P1 points to. The new record number is written written to register P2. If P3>0 then P3 is a register in the root frame of this VDBE that holds the largest previously generated record number. No new record numbers are allowed to be less than this value. When this value reaches its maximum, an SQLITE_FULL error is generated. The P3 register is updated with the ' generated record number. This P3 mechanism is used to help implement the

	AUTOINCREMENT feature.
Next	Advance cursor P1 so that it points to the next key/data pair in its table or index. If there are no more key/value pairs then fall through to the following instruction. But if the cursor advance was successful, jump immediately to P2. The Next opcode is only valid following an SeekGT, SeekGE, or Rewind opcode used to position the cursor. Next is not allowed to follow SeekLT, SeekLE, or Last. The P1 cursor must be for a real table, not a pseudo-table. P1 must have been opened prior to this opcode or the program will segfault. The P3 value is a hint to the btree implementation. If P3==1, that means P1 is an SQL index and that this instruction could have been omitted if that index had been unique. P3 is usually 0. P3 is always either 0 or 1. If P5 is positive and the jump is taken, then event counter number P5-1 in the prepared statement is incremented. See also: Prev
NoConflict	If P4==0 then register P3 holds a blob constructed by MakeRecord. If P4>0 then register P3 is the first of P4 registers that form an unpacked record. Cursor P1 is on an index btree. If the record identified by P3 and P4 contains any NULL value, jump immediately to P2. If all terms of the record are not-NULL then a check is done to determine if any row in the P1 index btree has a matching key prefix. If there are no matches, jump immediately to P2. If there is a match, fall through and leave the P1 cursor pointing to the matching row. This opcode is similar to NotFound with the exceptions that the branch is always taken if any part of the search key input is NULL. This operation leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes do not work after this operation.

	See also: NotFound, Found, NotExists
Noop	Do nothing. This instruction is often useful as a jump destination.
Not	Interpret the value in register P1 as a boolean value. Store the boolean complement in register P2. If the value in register P1 is NULL, then a NULL is stored in P2.
NotExists	P1 is the index of a cursor open on an SQL table btree (with integer keys). P3 is an integer rowid. If P1 does not contain a record with rowid P3 then jump immediately to P2. Or, if P2 is 0, raise an SQLITE_CORRUPT error. If P1 does contain a record with rowid P3 then leave the cursor pointing at that record and fall through to the next instruction. The SeekRowid opcode performs the same operation but also allows the P3 register to contain a non-integer value, in which case the jump is always taken. This opcode requires that P3 always contain an integer. The NotFound opcode performs the same operation on index btrees (with arbitrary multi-value keys). This opcode leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes will not work following this opcode. See also: Found, NotFound, NoConflict, SeekRowid
NotFound	If P4==0 then register P3 holds a blob constructed by MakeRecord. If P4>0 then register P3 is the first of P4 registers that form an unpacked record. Cursor P1 is on an index btree. If the record identified by P3 and P4 is not the prefix of any entry in P1 then a jump is made to P2. If P1 does contain an entry whose prefix matches the P3/P4 record then control falls through to the next instruction and P1 is left pointing at the matching entry. This operation leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes do not work after this operation.

	See also: Found, NotExists, NoConflict, IfNoHope
NotNull	Jump to P2 if the value in register P1 is not NULL.
Null	Write a NULL into registers P2. If P3 greater than P2, then also write NULL into register P3 and every register in between P2 and P3. If P3 is less than P2 (typically P3 is zero) then only register P2 is set to NULL. If the P1 value is non-zero, then also set the MEM_Cleared flag so that NULL values will not compare equal even if SQLITE_NULLEQ is set on Ne or Eq.
NullRow	Move the cursor P1 to a null row. Any Column operations that occur while the cursor is on the null row will always write a NULL. If cursor P1 is not previously opened, open it now to a special pseudo-cursor that always returns NULL for every column.
Offset	Store in register r[P3] the byte offset into the database file that is the start of the payload for the record at which that cursor P1 is currently pointing. P2 is the column number for the argument to the sqlite_offset() function. This opcode does not use P2 itself, but the P2 value is used by the code generator. The P1, P2, and P3 operands to this opcode are the same as for Column. This opcode is only available if SQLite is compiled with the -DSQLITE_ENABLE_OFFSET_SQL_FUNC option.
OffsetLimit	This opcode performs a commonly used computation associated with LIMIT and OFFSET process. r[P1] holds the limit counter. r[P3] holds the offset counter. The opcode computes the combined value of the LIMIT and OFFSET and stores that value in r[P2]. The r[P2] value computed is the total number of rows that will need to be visited in order to complete the query. If r[P3] is zero or negative, that means there is no OFFSET and r[P2] is set to be the value of the LIMIT, r[P1].

	if r[P1] is zero or negative, that means there is no LIMIT and r[P2] is set to -1. Otherwise, r[P2] is set to the sum of r[P1] and r[P3].
Once	Fall through to the next instruction the first time this opcode is encountered on each invocation of the byte-code program. Jump to P2 on the second and all subsequent encounters during the same invocation. Top-level programs determine first invocation by comparing the P1 operand against the P1 operand on the Init opcode at the beginning of the program. If the P1 values differ, then fall through and make the P1 of this opcode equal to the P1 of Init. If P1 values are the same then take the jump. For subprograms, there is a bitmask in the VdbeFrame that determines whether or not the jump should be taken. The bitmask is necessary because the self-altering code trick does not work for recursive triggers.
OpenAutoindex	This opcode works the same as OpenEphemeral. It has a different name to distinguish its use. Tables created using by this opcode will be used for automatically created transient indices in joins.
OpenDup	Open a new cursor P1 that points to the same ephemeral table as cursor P2. The P2 cursor must have been opened by a prior OpenEphemeral opcode. Only ephemeral cursors may be duplicated. Duplicate ephemeral cursors are used for self-joins of materialized views.
OpenEphemeral	Open a new cursor P1 to a transient table. The cursor is always opened read/write even if the main database is read-only. The ephemeral table is deleted automatically when the cursor is closed. If the cursor P1 is already opened on an ephemeral table, the table is cleared (all content is erased). P2 is the number of columns in the ephemeral table. The cursor points to a BTree table if P4==0 and to a BTree index if P4 is not 0. If P4 is not NULL, it points to a KeyInfo structure that defines the format of keys in

	the index. The P5 parameter can be a mask of the BTREE_* flags defined in btree.h. These flags control aspects of the operation of the btree. The BTREE_OMIT_JOURNAL and BTREE_SINGLE flags are added automatically. If P3 is positive, then reg[P3] is modified slightly so that it can be used as zero-length data for Insert. This is an optimization that avoids an extra Blob opcode to initialize that register.
OpenPseudo	Open a new cursor that points to a fake table that contains a single row of data. The content of that one row is the content of memory register P2. In other words, cursor P1 becomes an alias for the MEM_Blob content contained in register P2. A pseudo-table created by this opcode is used to hold a single row output from the sorter so that the row can be decomposed into individual columns using the Column opcode. The Column opcode is the only cursor opcode that works with a pseudo-table. P3 is the number of fields in the records that will be stored by the pseudo-table.
OpenRead	Open a read-only cursor for the database table whose root page is P2 in a database file. The database file is determined by P3. P3==0 means the main database, P3==1 means the database used for temporary tables, and P3>1 means used the corresponding attached database. Give the new cursor an identifier of P1. The P1 values need not be contiguous but all P1 values should be small integers. It is an error for P1 to be negative. Allowed P5 bits: • **0x02 OPFLAG_SEEKEQ**: This cursor will only be used for equality lookups (implemented as a pair of opcodes SeekGE/IdxGT of SeekLE/IdxLT) The P4 value may be either an integer (P4_INT32) or a pointer to a KeyInfo structure (P4_KEYINFO). If it is a pointer to a KeyInfo object, then table being opened must be an index b-tree where the KeyInfo object defines the content and collating sequence of that index b-tree. Otherwise, if P4 is an integer value, then the

	table being opened must be a table b-tree with a number of columns no less than the value of P4. See also: OpenWrite, ReopenIdx
OpenWrite	Open a read/write cursor named P1 on the table or index whose root page is P2 (or whose root page is held in register P2 if the OPFLAG_P2ISREG bit is set in P5 - see below). The P4 value may be either an integer (P4_INT32) or a pointer to a KeyInfo structure (P4_KEYINFO). If it is a pointer to a KeyInfo object, then table being opened must be an index b-tree where the KeyInfo object defines the content and collating sequence of that index b-tree. Otherwise, if P4 is an integer value, then the table being opened must be a table b-tree with a number of columns no less than the value of P4. Allowed P5 bits: • **0x02 OPFLAG_SEEKEQ**: This cursor will only be used for equality lookups (implemented as a pair of opcodes SeekGE/IdxGT of SeekLE/IdxLT) • **0x08 OPFLAG_FORDELETE**: This cursor is used only to seek and subsequently delete entries in an index btree. This is a hint to the storage engine that the storage engine is allowed to ignore. The hint is not used by the official SQLite b*tree storage engine, but is used by COMDB2. • **0x10 OPFLAG_P2ISREG**: Use the content of register P2 as the root page, not the value of P2 itself. This instruction works like OpenRead except that it opens the cursor in read/write mode. See also: OpenRead, ReopenIdx
Or	Take the logical OR of the values in register P1 and P2 and store the answer in register P3. If either P1 or P2 is nonzero (true) then the result is 1 (true) even if the other input is NULL. A NULL and false or two NULLs give a NULL output.

Pagecount	Write the current number of pages in database P1 to memory cell P2.
Param	This opcode is only ever present in sub-programs called via the Program instruction. Copy a value currently stored in a memory cell of the calling (parent) frame to cell P2 in the current frames address space. This is used by trigger programs to access the new.* and old.* values. The address of the cell in the parent frame is determined by adding the value of the P1 argument to the value of the P1 argument to the calling Program instruction.
ParseSchema	Read and parse all entries from the schema table of database P1 that match the WHERE clause P4. If P4 is a NULL pointer, then the entire schema for P1 is reparsed. This opcode invokes the parser to create a new virtual machine, then runs the new virtual machine. It is thus a re-entrant opcode.
Permutation	Set the permutation used by the Compare operator in the next instruction. The permutation is stored in the P4 operand. The permutation is only valid for the next opcode which must be an Compare that has the OPFLAG_PERMUTE bit set in P5. The first integer in the P4 integer array is the length of the array and does not become part of the permutation.
Prev	Back up cursor P1 so that it points to the previous key/data pair in its table or index. If there is no previous key/value pairs then fall through to the following instruction. But if the cursor backup was successful, jump immediately to P2. The Prev opcode is only valid following an SeekLT, SeekLE, or Last opcode used to position the cursor. Prev is not allowed to follow SeekGT, SeekGE, or Rewind. The P1 cursor must be for a real table, not a pseudo-table. If P1 is not open then the behavior is undefined.

	The P3 value is a hint to the btree implementation. If P3==1, that means P1 is an SQL index and that this instruction could have been omitted if that index had been unique. P3 is usually 0. P3 is always either 0 or 1. If P5 is positive and the jump is taken, then event counter number P5-1 in the prepared statement is incremented.
Program	Execute the trigger program passed as P4 (type P4_SUBPROGRAM). P1 contains the address of the memory cell that contains the first memory cell in an array of values used as arguments to the sub-program. P2 contains the address to jump to if the sub-program throws an IGNORE exception using the RAISE() function. Register P3 contains the address of a memory cell in this (the parent) VM that is used to allocate the memory required by the sub-vdbe at runtime. P4 is a pointer to the VM containing the trigger program. If P5 is non-zero, then recursive program invocation is enabled.
PureFunc	Invoke a user function (P4 is a pointer to an sqlite3_context object that contains a pointer to the function to be run) with arguments taken from register P2 and successors. The number of arguments is in the sqlite3_context object that P4 points to. The result of the function is stored in register P3. Register P3 must not be one of the function inputs. P1 is a 32-bit bitmask indicating whether or not each argument to the function was determined to be constant at compile time. If the first argument was constant then bit 0 of P1 is set. This is used to determine whether meta data associated with a user function argument using the sqlite3_set_auxdata() API may be safely retained until the next invocation of this opcode. This opcode works exactly like Function. The only difference is in its name. This opcode is used in places where the function must be purely non-deterministic. Some built-in date/time functions can be either determinitic of non-deterministic, depending on their

	arguments. When those function are used in a non-deterministic way, they will check to see if they were called using PureFunc instead of Function, and if they were, they throw an error. See also: AggStep, AggFinal, Function
ReadCookie	Read cookie number P3 from database P1 and write it into register P2. P3==1 is the schema version. P3==2 is the database format. P3==3 is the recommended pager cache size, and so forth. P1==0 is the main database file and P1==1 is the database file used to store temporary tables. There must be a read-lock on the database (either a transaction must be started or there must be an open cursor) before executing this instruction.
Real	P4 is a pointer to a 64-bit floating point value. Write that value into register P2.
RealAffinity	If register P1 holds an integer convert it to a real value. This opcode is used when extracting information from a column that has REAL affinity. Such column values may still be stored as integers, for space efficiency, but after extraction we want them to have only a real value.
ReleaseReg	Release registers from service. Any content that was in the the registers is unreliable after this opcode completes. The registers released will be the P2 registers starting at P1, except if bit ii of P3 set, then do not release register P1+ii. In other words, P3 is a mask of registers to preserve. Releasing a register clears the Mem.pScopyFrom pointer. That means that if the content of the released register was set using SCopy, a change to the value of the source register for the SCopy will no longer generate an assertion fault in sqlite3VdbeMemAboutToChange(). If P5 is set, then all released registers have their type set to MEM_Undefined so that any subsequent attempt to read the released register (before it is reinitialized) will generate an assertion fault.

	P5 ought to be set on every call to this opcode. However, there are places in the code generator will release registers before their are used, under the (valid) assumption that the registers will not be reallocated for some other purpose before they are used and hence are safe to release. This opcode is only available in testing and debugging builds. It is not generated for release builds. The purpose of this opcode is to help validate the generated bytecode. This opcode does not actually contribute to computing an answer.
Remainder	Compute the remainder after integer register P2 is divided by register P1 and store the result in register P3. If the value in register P1 is zero the result is NULL. If either operand is NULL, the result is NULL.
ReopenIdx	The ReopenIdx opcode works like OpenRead except that it first checks to see if the cursor on P1 is already open on the same b-tree and if it is this opcode becomes a no-op. In other words, if the cursor is already open, do not reopen it. The ReopenIdx opcode may only be used with P5==0 or P5==OPFLAG_SEEKEQ and with P4 being a P4_KEYINFO object. Furthermore, the P3 value must be the same as every other ReopenIdx or OpenRead for the same cursor number. Allowed P5 bits: • **0x02 OPFLAG_SEEKEQ**: This cursor will only be used for equality lookups (implemented as a pair of opcodes SeekGE/IdxGT of SeekLE/IdxLT) See also: OpenRead, OpenWrite
ResetCount	The value of the change counter is copied to the database handle change counter (returned by subsequent calls to sqlite3_changes()). Then the VMs internal change counter resets to 0. This is used by trigger programs.
ResetSorter	Delete all contents from the ephemeral table or sorter that is open on cursor P1. This opcode only works for cursors used for sorting and

	opened with OpenEphemeral or SorterOpen.
ResultRow	The registers P1 through P1+P2-1 contain a single row of results. This opcode causes the sqlite3_step() call to terminate with an SQLITE_ROW return code and it sets up the sqlite3_stmt structure to provide access to the r(P1)..r(P1+P2-1) values as the result row.
Return	Jump to the address stored in register P1. If P1 is a return address register, then this accomplishes a return from a subroutine. If P3 is 1, then the jump is only taken if register P1 holds an integer values, otherwise execution falls through to the next opcode, and the Return becomes a no-op. If P3 is 0, then register P1 must hold an integer or else an assert() is raised. P3 should be set to 1 when this opcode is used in combination with BeginSubrtn, and set to 0 otherwise. The value in register P1 is unchanged by this opcode. P2 is not used by the byte-code engine. However, if P2 is positive and also less than the current address, then the "EXPLAIN" output formatter in the CLI will indent all opcodes from the P2 opcode up to be not including the current Return. P2 should be the first opcode in the subroutine from which this opcode is returning. Thus the P2 value is a byte-code indentation hint. See tag-20220407a in wherecode.c and shell.c.
Rewind	The next use of the Rowid or Column or Next instruction for P1 will refer to the first entry in the database table or index. If the table or index is empty, jump immediately to P2. If the table or index is not empty, fall through to the following instruction. This opcode leaves the cursor configured to move in forward order, from the beginning toward the end. In other words, the cursor is configured to use Next, not Prev.
RowCell	P1 and P2 are both open cursors. Both must be opened on the same type of table - intkey or index. This opcode is used as part of copying the current row from P2 into P1. If the cursors are opened on intkey tables, register P3 contains the rowid to use with the new record in P1.

	If they are opened on index tables, P3 is not used. This opcode must be followed by either an Insert or InsertIdx opcode with the OPFLAG_PREFORMAT flag set to complete the insert operation.
RowData	Write into register P2 the complete row content for the row at which cursor P1 is currently pointing. There is no interpretation of the data. It is just copied onto the P2 register exactly as it is found in the database file. If cursor P1 is an index, then the content is the key of the row. If cursor P2 is a table, then the content extracted is the data. If the P1 cursor must be pointing to a valid row (not a NULL row) of a real table, not a pseudo-table. If P3!=0 then this opcode is allowed to make an ephemeral pointer into the database page. That means that the content of the output register will be invalidated as soon as the cursor moves - including moves caused by other cursors that "save" the current cursors position in order that they can write to the same table. If P3==0 then a copy of the data is made into memory. P3!=0 is faster, but P3==0 is safer. If P3!=0 then the content of the P2 register is unsuitable for use in OP_Result and any OP_Result will invalidate the P2 register content. The P2 register content is invalidated by opcodes like Function or by any use of another cursor pointing to the same table.
Rowid	Store in register P2 an integer which is the key of the table entry that P1 is currently point to. P1 can be either an ordinary table or a virtual table. There used to be a separate OP_VRowid opcode for use with virtual tables, but this one opcode now works for both table types.
RowSetAdd	Insert the integer value held by register P2 into a RowSet object held in register P1. An assertion fails if P2 is not an integer.

RowSetRead	Extract the smallest value from the RowSet object in P1 and put that value into register P3. Or, if RowSet object P1 is initially empty, leave P3 unchanged and jump to instruction P2.
RowSetTest	Register P3 is assumed to hold a 64-bit integer value. If register P1 contains a RowSet object and that RowSet object contains the value held in P3, jump to register P2. Otherwise, insert the integer in P3 into the RowSet and continue on to the next opcode. The RowSet object is optimized for the case where sets of integers are inserted in distinct phases, which each set contains no duplicates. Each set is identified by a unique P4 value. The first set must have P4==0, the final set must have P4==-1, and for all other sets must have P4>0. This allows optimizations: (a) when P4==0 there is no need to test the RowSet object for P3, as it is guaranteed not to contain it, (b) when P4==-1 there is no need to insert the value, as it will never be tested for, and (c) when a value that is part of set X is inserted, there is no need to search to see if the same value was previously inserted as part of set X (only if it was previously inserted as part of some other set).
Savepoint	Open, release or rollback the savepoint named by parameter P4, depending on the value of P1. To open a new savepoint set P1==0 (SAVEPOINT_BEGIN). To release (commit) an existing savepoint set P1==1 (SAVEPOINT_RELEASE). To rollback an existing savepoint set P1==2 (SAVEPOINT_ROLLBACK).
SCopy	Make a shallow copy of register P1 into register P2. This instruction makes a shallow copy of the value. If the value is a string or blob, then the copy is only a pointer to the original and hence if the original changes so will the copy. Worse, if the original is deallocated, the copy becomes invalid. Thus the program must guarantee that the original will not change during the lifetime of the copy. Use Copy to make a complete copy.
SeekEnd	Position cursor P1 at the end of the btree for the purpose of appending a new entry onto the btree. It is assumed that the cursor is used only for appending

	and so if the cursor is valid, then the cursor must already be pointing at the end of the btree and so no changes are made to the cursor.
SeekGE	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as the key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key. Reposition cursor P1 so that it points to the smallest entry that is greater than or equal to the key value. If there are no records greater than or equal to the key and P2 is not zero, then jump to P2. If the cursor P1 was opened using the OPFLAG_SEEKEQ flag, then this opcode will either land on a record that exactly matches the key, or else it will cause a jump to P2. When the cursor is OPFLAG_SEEKEQ, this opcode must be followed by an IdxLE opcode with the same arguments. The IdxGT opcode will be skipped if this opcode succeeds, but the IdxGT opcode will be used on subsequent loop iterations. The OPFLAG_SEEKEQ flags is a hint to the btree layer to say that this is an equality search. This opcode leaves the cursor configured to move in forward order, from the beginning toward the end. In other words, the cursor is configured to use Next, not Prev. See also: Found, NotFound, SeekLt, SeekGt, SeekLe
SeekGT	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as a key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key. Reposition cursor P1 so that it points to the smallest entry that is greater than the key value. If there are no records greater than the key and P2 is not zero, then jump to P2. This opcode leaves the cursor configured to move in forward order, from the beginning toward the end. In other words, the cursor is configured to use Next, not Prev.

	See also: Found, NotFound, SeekLt, SeekGe, SeekLe
SeekHit	Increase or decrease the seekHit value for cursor P1, if necessary, so that it is no less than P2 and no greater than P3. The seekHit integer represents the maximum of terms in an index for which there is known to be at least one match. If the seekHit value is smaller than the total number of equality terms in an index lookup, then the IfNoHope opcode might run to see if the IN loop can be abandoned early, thus saving work. This is part of the IN-early-out optimization. P1 must be a valid b-tree cursor.
SeekLE	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as a key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key. Reposition cursor P1 so that it points to the largest entry that is less than or equal to the key value. If there are no records less than or equal to the key and P2 is not zero, then jump to P2. This opcode leaves the cursor configured to move in reverse order, from the end toward the beginning. In other words, the cursor is configured to use Prev, not Next. If the cursor P1 was opened using the OPFLAG_SEEKEQ flag, then this opcode will either land on a record that exactly matches the key, or else it will cause a jump to P2. When the cursor is OPFLAG_SEEKEQ, this opcode must be followed by an IdxLE opcode with the same arguments. The IdxGE opcode will be skipped if this opcode succeeds, but the IdxGE opcode will be used on subsequent loop iterations. The OPFLAG_SEEKEQ flags is a hint to the btree layer to say that this is an equality search. See also: Found, NotFound, SeekGt, SeekGe, SeekLt
SeekLT	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as a key. If

	cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key. Reposition cursor P1 so that it points to the largest entry that is less than the key value. If there are no records less than the key and P2 is not zero, then jump to P2. This opcode leaves the cursor configured to move in reverse order, from the end toward the beginning. In other words, the cursor is configured to use Prev, not Next. See also: Found, NotFound, SeekGt, SeekGe, SeekLe
SeekRowid	P1 is the index of a cursor open on an SQL table btree (with integer keys). If register P3 does not contain an integer or if P1 does not contain a record with rowid P3 then jump immediately to P2. Or, if P2 is 0, raise an SQLITE_CORRUPT error. If P1 does contain a record with rowid P3 then leave the cursor pointing at that record and fall through to the next instruction. The NotExists opcode performs the same operation, but with NotExists the P3 register must be guaranteed to contain an integer value. With this opcode, register P3 might not contain an integer. The NotFound opcode performs the same operation on index btrees (with arbitrary multi-value keys). This opcode leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes will not work following this opcode. See also: Found, NotFound, NoConflict, SeekRowid
SeekScan	This opcode is a prefix opcode to SeekGE. In other words, this opcode must be immediately followed by SeekGE. This constraint is checked by assert() statements. This opcode uses the P1 through P4 operands of the subsequent SeekGE. In the text that follows, the operands of the subsequent SeekGE opcode are denoted as SeekOP.P1 through SeekOP.P4. Only the P1 and P2 operands of this opcode are also used, and are called

	This.P1 and This.P2. This opcode helps to optimize IN operators on a multi-column index where the IN operator is on the later terms of the index by avoiding unnecessary seeks on the btree, substituting steps to the next row of the b-tree instead. A correct answer is obtained if this opcode is omitted or is a no-op. The SeekGE.P3 and SeekGE.P4 operands identify an unpacked key which is the desired entry that we want the cursor SeekGE.P1 to be pointing to. Call this SeekGE.P4/P5 row the "target". If the SeekGE.P1 cursor is not currently pointing to a valid row, then this opcode is a no-op and control passes through into the SeekGE. If the SeekGE.P1 cursor is pointing to a valid row, then that row might be the target row, or it might be near and slightly before the target row. This opcode attempts to position the cursor on the target row by, perhaps by invoking sqlite3BtreeStep() on the cursor between 0 and This.P1 times. There are three possible outcomes from this opcode: 1. If after This.P1 steps, the cursor is still pointing to a place that is earlier in the btree than the target row, then fall through into the subsquence SeekGE opcode. 2. If the cursor is successfully moved to the target row by 0 or more sqlite3BtreeNext() calls, then jump to This.P2, which will land just past the IdxGT or IdxGE opcode that follows the SeekGE. 3. If the cursor ends up past the target row (indicating the the target row does not exist in the btree) then jump to SeekOP.P2.
Sequence	Find the next available sequence number for cursor P1. Write the sequence number into register P2. The sequence number on the cursor is incremented after this instruction.
SequenceTest	P1 is a sorter cursor. If the sequence counter is currently zero, jump to P2. Regardless of whether or not the jump is taken, increment the the sequence value.

SetCookie	Write the integer value P3 into cookie number P2 of database P1. P2==1 is the schema version. P2==2 is the database format. P2==3 is the recommended pager cache size, and so forth. P1==0 is the main database file and P1==1 is the database file used to store temporary tables. A transaction must be started before executing this opcode. If P2 is the SCHEMA_VERSION cookie (cookie number 1) then the internal schema version is set to P3-P5. The "PRAGMA schema_version=N" statement has P5 set to 1, so that the internal schema version will be different from the database schema version, resulting in a schema reset.
ShiftLeft	Shift the integer value in register P2 to the left by the number of bits specified by the integer in register P1. Store the result in register P3. If either input is NULL, the result is NULL.
ShiftRight	Shift the integer value in register P2 to the right by the number of bits specified by the integer in register P1. Store the result in register P3. If either input is NULL, the result is NULL.
SoftNull	Set register P1 to have the value NULL as seen by the MakeRecord instruction, but do not free any string or blob memory associated with the register, so that if the value was a string or blob that was previously copied using SCopy, the copies will continue to be valid.
Sort	This opcode does exactly the same thing as Rewind except that it increments an undocumented global variable used for testing. Sorting is accomplished by writing records into a sorting index, then rewinding that index and playing it back from beginning to end. We use the Sort opcode instead of Rewind to do the rewinding so that the global variable will be incremented and regression tests can determine whether or not the optimizer is correctly optimizing out sorts.
SorterCompare	P1 is a sorter cursor. This instruction compares a prefix of the record blob in register P3 against a prefix of the entry that the sorter cursor currently points to. Only the

	first P4 fields of r[P3] and the sorter record are compared. If either P3 or the sorter contains a NULL in one of their significant fields (not counting the P4 fields at the end which are ignored) then the comparison is assumed to be equal. Fall through to next instruction if the two records compare equal to each other. Jump to P2 if they are different.
SorterData	Write into register P2 the current sorter data for sorter cursor P1. Then clear the column header cache on cursor P3. This opcode is normally use to move a record out of the sorter and into a register that is the source for a pseudo-table cursor created using OpenPseudo. That pseudo-table cursor is the one that is identified by parameter P3. Clearing the P3 column cache as part of this opcode saves us from having to issue a separate NullRow instruction to clear that cache.
SorterInsert	Register P2 holds an SQL index key made using the MakeRecord instructions. This opcode writes that key into the sorter P1. Data for the entry is nil.
SorterNext	This opcode works just like Next except that P1 must be a sorter object for which the SorterSort opcode has been invoked. This opcode advances the cursor to the next sorted record, or jumps to P2 if there are no more sorted records.
SorterOpen	This opcode works like OpenEphemeral except that it opens a transient index that is specifically designed to sort large tables using an external merge-sort algorithm. If argument P3 is non-zero, then it indicates that the sorter may assume that a stable sort considering the first P3 fields of each key is sufficient to produce the required results.
SorterSort	After all records have been inserted into the Sorter object identified by P1, invoke this opcode to actually do the sorting. Jump to P2 if there are no records to be

	sorted. This opcode is an alias for Sort and Rewind that is used for Sorter objects.
SqlExec	Run the SQL statement or statements specified in the P4 string.
String	The string value P4 of length P1 (bytes) is stored in register P2. If P3 is not zero and the content of register P3 is equal to P5, then the datatype of the register P2 is converted to BLOB. The content is the same sequence of bytes, it is merely interpreted as a BLOB instead of a string, as if it had been CAST. In other words: if(P3!=0 and reg[P3]==P5) reg[P2] := CAST(reg[P2] as BLOB)
String8	P4 points to a nul terminated UTF-8 string. This opcode is transformed into a String opcode before it is executed for the first time. During this transformation, the length of string P4 is computed and stored as the P1 parameter.
Subtract	Subtract the value in register P1 from the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
TableLock	Obtain a lock on a particular table. This instruction is only used when the shared-cache feature is enabled. P1 is the index of the database in sqlite3.aDb[] of the database on which the lock is acquired. A readlock is obtained if P3==0 or a write lock if P3==1. P2 contains the root-page of the table to lock. P4 contains a pointer to the name of the table being locked. This is only used to generate an error message if the lock cannot be obtained.
Trace	Write P4 on the statement trace output if statement tracing is enabled. Operand P1 must be 0x7fffffff and P2 must positive.

Transaction	Begin a transaction on database P1 if a transaction is not already active. If P2 is non-zero, then a write-transaction is started, or if a read-transaction is already active, it is upgraded to a write-transaction. If P2 is zero, then a read-transaction is started. If P2 is 2 or more then an exclusive transaction is started. P1 is the index of the database file on which the transaction is started. Index 0 is the main database file and index 1 is the file used for temporary tables. Indices of 2 or more are used for attached databases. If a write-transaction is started and the Vdbe.usesStmtJournal flag is true (this flag is set if the Vdbe may modify more than one row and may throw an ABORT exception), a statement transaction may also be opened. More specifically, a statement transaction is opened iff the database connection is currently not in autocommit mode, or if there are other active statements. A statement transaction allows the changes made by this VDBE to be rolled back after an error without having to roll back the entire transaction. If no error is encountered, the statement transaction will automatically commit when the VDBE halts. If P5!=0 then this opcode also checks the schema cookie against P3 and the schema generation counter against P4. The cookie changes its value whenever the database schema changes. This operation is used to detect when that the cookie has changed and that the current process needs to reread the schema. If the schema cookie in P3 differs from the schema cookie in the database header or if the schema generation counter in P4 differs from the current generation counter, then an SQLITE_SCHEMA error is raised and execution halts. The sqlite3_step() wrapper function might then reprepare the statement and rerun it from the beginning.
TypeCheck	Apply affinities to the range of P2 registers beginning with P1. Take the affinities from the Table object in P4. If any value cannot be coerced into the correct type, then raise an error. This opcode is similar to Affinity except that this opcode forces the register type to the Table column type. This is used to implement "strict affinity". GENERATED ALWAYS AS ... STATIC columns are only

	checked if P3 is zero. When P3 is non-zero, no type checking occurs for static generated columns. Virtual columns are computed at query time and so they are never checked. Preconditions: • P2 should be the number of non-virtual columns in the table of P4. • Table P4 should be a STRICT table. If any precondition is false, an assertion fault occurs.
Vacuum	Vacuum the entire database P1. P1 is 0 for "main", and 2 or more for an attached database. The "temp" database may not be vacuumed. If P2 is not zero, then it is a register holding a string which is the file into which the result of vacuum should be written. When P2 is zero, the vacuum overwrites the original database.
Variable	Transfer the values of bound parameter P1 into register P2 If the parameter is named, then its name appears in P4. The P4 value is used by sqlite3_bind_parameter_name().
VBegin	P4 may be a pointer to an sqlite3_vtab structure. If so, call the xBegin method for that table. Also, whether or not P4 is set, check that this is not being called from within a callback to a virtual table xSync() method. If it is, the error code will be set to SQLITE_LOCKED.
VColumn	Store in register P3 the value of the P2-th column of the current row of the virtual-table of cursor P1. If the VColumn opcode is being used to fetch the value of an unchanging column during an UPDATE operation, then the P5 value is OPFLAG_NOCHNG. This will cause the sqlite3_vtab_nochange() function to return true inside the xColumn method of the virtual table implementation. The P5 column might also contain other bits (OPFLAG_LENGTHARG or

	OPFLAG_TYPEOFARG) but those bits are unused by VColumn.
VCreate	P2 is a register that holds the name of a virtual table in database P1. Call the xCreate method for that table.
VDestroy	P4 is the name of a virtual table in database P1. Call the xDestroy method of that table.
VFilter	P1 is a cursor opened using VOpen. P2 is an address to jump to if the filtered result set is empty. P4 is either NULL or a string that was generated by the xBestIndex method of the module. The interpretation of the P4 string is left to the module implementation. This opcode invokes the xFilter method on the virtual table specified by P1. The integer query plan parameter to xFilter is stored in register P3. Register P3+1 stores the argc parameter to be passed to the xFilter method. Registers P3+2..P3+1+argc are the argc additional parameters which are passed to xFilter as argv. Register P3+2 becomes argv[0] when passed to xFilter. A jump is made to P2 if the result set after filtering would be empty.
VInitIn	Set register P2 to be a pointer to a ValueList object for cursor P1 with cache register P3 and output register P3+1. This ValueList object can be used as the first argument to sqlite3_vtab_in_first() and sqlite3_vtab_in_next() to extract all of the values stored in the P1 cursor. Register P3 is used to hold the values returned by sqlite3_vtab_in_first() and sqlite3_vtab_in_next().
VNext	Advance virtual table P1 to the next row in its result set and jump to instruction P2. Or, if the virtual table has reached the end of its result set, then fall through to the next instruction.
VOpen	P4 is a pointer to a virtual table object, an sqlite3_vtab structure. P1 is a cursor number. This opcode opens a cursor to the virtual table and stores that cursor in P1.
VRename	P4 is a pointer to a virtual table object, an sqlite3_vtab structure. This opcode invokes the corresponding xRename method. The value in register P1 is passed as

	the zName argument to the xRename method.
VUpdate	P4 is a pointer to a virtual table object, an sqlite3_vtab structure. This opcode invokes the corresponding xUpdate method. P2 values are contiguous memory cells starting at P3 to pass to the xUpdate invocation. The value in register (P3+P2-1) corresponds to the p2th element of the argv array passed to xUpdate. The xUpdate method will do a DELETE or an INSERT or both. The argv[0] element (which corresponds to memory cell P3) is the rowid of a row to delete. If argv[0] is NULL then no deletion occurs. The argv[1] element is the rowid of the new row. This can be NULL to have the virtual table select the new rowid for itself. The subsequent elements in the array are the values of columns in the new row. If P2==1 then no insert is performed. argv[0] is the rowid of a row to delete. P1 is a boolean flag. If it is set to true and the xUpdate call is successful, then the value returned by sqlite3_last_insert_rowid() is set to the value of the rowid for the row just inserted. P5 is the error actions (OE_Replace, OE_Fail, OE_Ignore, etc) to apply in the case of a constraint failure on an insert or update.
Yield	Swap the program counter with the value in register P1. This has the effect of yielding to a coroutine. If the coroutine that is launched by this instruction ends with Yield or Return then continue to the next instruction. But if the coroutine launched by this instruction ends with EndCoroutine, then jump to P2 rather than continuing with the next instruction. See also: InitCoroutine
ZeroOrNull	If all both registers P1 and P3 are NOT NULL, then store a zero in register P2. If either registers P1 or P3 are NULL then put a NULL in register P2.