Small. Fast. Reliable.
Choose any three.

Home Menu About Documentation Download License Support
Purchase Search

The SQLite Query Planner

This document provides overview of how the query planner and optimizer for SQLite works.

Given a single SQL statement, there might be dozens, hundreds, or even thousands of ways to implement that statement, depending on the complexity of the statement itself and of the underlying database schema. The task of the query planner is to select an algorithm from among the many choices that provides the answer with a minimum of disk I/O and CPU overhead.

Additional background information is available in the indexing tutorial document.

With release 3.8.0, the SQLite query planner was reimplemented as the Next Generation Query Planner or "NGQP". All of the features, techniques, and algorithms described in this document are applicable to both the pre-3.8.0 legacy query planner and to the NGQP. For further information on how the NGQP differs from the legacy query planner, see the detailed description of the NGQP.

1.0 WHERE clause analysis

The WHERE clause on a query is broken up into "terms" where each term is separated from the others by an AND operator. If the WHERE clause is composed of constraints separate by the OR operator then the entire clause is considered to be a single "term" to which the OR-clause optimization is applied.

All terms of the WHERE clause are analyzed to see if they can be satisfied using indices. To be usable by an index a term must be of one of the following forms:

```
column = expression
column IS expression
column > expression
column >= expression
column < expression
column <= expression
```

```
expression = column
expression > column
expression >= column
expression < column
expression <= column
column IN (expression-list)
column IN (subquery)
column IS NULL
```

If an index is created using a statement like this:

```
CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e,...,y,z);
```

Then the index might be used if the initial columns of the index (columns a, b, and so forth) appear in WHERE clause terms. The initial columns of the index must be used with the **=** or **IN** or **IS** operators. The right-most column that is used can employ inequalities. For the right-most column of an index that is used, there can be up to two inequalities that must sandwich the allowed values of the column between two extremes.

It is not necessary for every column of an index to appear in a WHERE clause term in order for that index to be used. But there cannot be gaps in the columns of the index that are used. Thus for the example index above, if there is no WHERE clause term that constraints column c, then terms that constrain columns a and b can be used with the index but not terms that constraint columns d through z. Similarly, index columns will not normally be used (for indexing purposes) if they are to the right of a column that is constrained only by inequalities. (See the skip-scan optimization below for the exception.)

In the case of indexes on expressions, whenever the word "column" is used in the foregoing text, one can substitute "indexed expression" (meaning a copy of the expression that appears in the CREATE INDEX statement) and everything will work the same.

1.1 Index term usage examples

For the index above and WHERE clause like this:

```
... WHERE a=5 AND b IN (1,2,3) AND c IS NULL AND d='hello'
```

The first four columns a, b, c, and d of the index would be usable since those four columns form a prefix of the index and are all bound by equality constraints.

For the index above and WHERE clause like this:

```
... WHERE a=5 AND b IN (1,2,3) AND c>12 AND d='hello'
```

Only columns a, b, and c of the index would be usable. The d column would not be usable because it occurs to the right of c and c is constrained only by inequalities.

For the index above and WHERE clause like this:

```
... WHERE a=5 AND b IN (1,2,3) AND d='hello'
```

Only columns a and b of the index would be usable. The d column would not be usable because column c is not constrained and there can be no gaps in the set of columns that usable by the index.

For the index above and WHERE clause like this:

```
... WHERE b IN (1,2,3) AND c NOT NULL AND d='hello'
```

The index is not usable at all because the left-most column of the index (column "a") is not constrained. Assuming there are no other indices, the query above would result in a full table scan.

For the index above and WHERE clause like this:

```
... WHERE a=5 OR b IN (1,2,3) OR c NOT NULL OR d='hello'
```

The index is not usable because the WHERE clause terms are connected by OR instead of AND. This query would result in a full table scan. However, if three additional indices where added that contained columns b, c, and d as their left-most columns, then the OR-clause optimization might apply.

2.0 The BETWEEN optimization

If a term of the WHERE clause is of the following form:

expr1 **BETWEEN** *expr2* **AND** *expr3*

Then two "virtual" terms are added as follows:

expr1 **>=** *expr2* **AND** *expr1* **<=** *expr3*

Virtual terms are used for analysis only and do not cause any VDBE code to be generated. If both virtual terms end up being used as constraints on an index, then the original BETWEEN term is omitted and the corresponding test is not performed on input rows. Thus if the BETWEEN term ends up being used as an index constraint no tests are ever performed on that term. On the other hand, the virtual terms themselves never causes tests to be performed on input rows. Thus if the BETWEEN term is not used as an index constraint and instead must be used to test input rows, the *expr1* expression is only evaluated once.

3.0 OR optimizations

WHERE clause constraints that are connected by OR instead of AND can be handled in two different ways. If a term consists of multiple subterms containing a common column name and separated by OR, like this:

```
column = expr1 OR column = expr2 OR column = expr3 OR ...
```

Then that term is rewritten as follows:

```
column IN (expr1,expr2,expr3,...)
```

The rewritten term then might go on to constrain an index using the normal rules for **IN** operators. Note that *column* must be the same column in every OR-connected subterm, although the column can occur on either the left or the right side of the = operator.

If and only if the previously described conversion of OR to an IN operator does not work, the second OR-clause optimization is attempted. Suppose the OR clause consists of multiple subterms as follows:

```
expr1 OR expr2 OR expr3
```

Individual subterms might be a single comparison expression like **a=5** or **x>y** or they can be LIKE or BETWEEN expressions, or a subterm can be a parenthesized list of AND-connected sub-subterms. Each subterm is analyzed as if it were itself the entire WHERE clause in order to see if the subterm is indexable by itself. If every subterm of an OR clause is separately indexable then the OR clause might be coded such that a separate index is used to evaluate each term of the OR clause. One way to think about how SQLite uses separate indices for each OR clause term is to imagine that the WHERE clause where rewritten as follows:

```
rowid IN (SELECT rowid FROM table WHERE expr1
          UNION SELECT rowid FROM table WHERE expr2
          UNION SELECT rowid FROM table WHERE expr3)
```

The rewritten expression above is conceptual; WHERE clauses containing OR are not really rewritten this way. The actual implementation of the OR clause uses a mechanism that is more efficient and that works even for WITHOUT ROWID tables or tables in which the "rowid" is inaccessible. But the essence of the implementation is captured by the statement above: Separate indices are used to find candidate result rows from each OR clause term and the final result is the union of those rows.

Note that in most cases, SQLite will only use a single index for each table in the FROM clause of a query. The second OR-clause optimization described here is the exception to that rule. With an OR-clause, a different index might be used for each subterm in the OR-clause.

For any given query, the fact that the OR-clause optimization described here can be used does not guarantee that it will be used. SQLite uses a cost-based query planner that estimates the CPU and disk I/O costs of various competing query plans and chooses the plan that it thinks will be the fastest. If there are many OR terms in the WHERE clause or if some of the indices on individual OR-clause subterms are not very selective, then SQLite might decide that it is faster to use a different query algorithm, or even a full-table scan. Application developers can use the EXPLAIN QUERY PLAN prefix on a statement to get a high-level overview of the chosen query strategy.

4.0 The LIKE optimization

A WHERE-clause term that uses the LIKE or GLOB operator can sometimes be used with an index to do a range search, almost as if the LIKE or GLOB were an alternative to a BETWEEN operator. There are many conditions on this optimization:

1. The right-hand side of the LIKE or GLOB must be either a string literal or a parameter bound to a string literal that does not begin with a wildcard character.
2. The ESCAPE clause cannot appear on the LIKE operator.
3. It must not be possible to make the LIKE or GLOB operator true by having a numeric value (instead of a string or blob) on the left-hand side. This means that either:
   A. the left-hand side of the LIKE or GLOB operator is the name of an indexed column with TEXT affinity, or
   B. the right-hand side pattern argument does not begin with a minus sign ("-") or a digit.

   This constraint arises from the fact that numbers do not sort in lexicographical order. For example: 9<10 but '9'>'10'.
4. The built-in functions used to implement LIKE and GLOB must not have been overloaded using the sqlite3_create_function() API.
5. For the GLOB operator, the column must be indexed using the built-in BINARY collating sequence.
6. For the LIKE operator, if case_sensitive_like mode is enabled then the column must indexed using BINARY collating sequence, or if case_sensitive_like mode is disabled then the column must indexed using built-in NOCASE collating sequence.

The LIKE operator has two modes that can be set by a pragma. The default mode is for LIKE comparisons to be insensitive to differences of case for latin1 characters. Thus, by default, the following expression is true:

```
'a' LIKE 'A'
```

But if the case_sensitive_like pragma is enabled as follows:

```
PRAGMA case_sensitive_like=ON;
```

Then the LIKE operator pays attention to case and the example above would evaluate to false. Note that case insensitivity only applies to latin1 characters - basically the upper and lower case letters of English in the lower 127 byte codes of ASCII. International character sets are case sensitive in SQLite unless an application-defined collating sequence and like() SQL function are provided that take non-ASCII characters into account. But if an application-defined collating sequence and/or like() SQL function are provided, the LIKE optimization described here will never be taken.

The LIKE operator is case insensitive by default because this is what the SQL standard requires. You can change the default behavior at compile time by using the SQLITE_CASE_SENSITIVE_LIKE command-line option to the compiler.

The LIKE optimization might occur if the column named on the left of the operator is indexed using the built-in BINARY collating sequence and case_sensitive_like is turned on. Or the optimization might occur if the column is indexed using the built-in NOCASE collating sequence and the case_sensitive_like mode is off. These are the only two combinations under which LIKE operators will be optimized.

The GLOB operator is always case sensitive. The column on the left side of the GLOB operator must always use the built-in BINARY collating sequence or no attempt will be made to optimize that operator with indices.

The LIKE optimization will only be attempted if the right-hand side of the GLOB or LIKE operator is either literal string or a parameter that has been bound to a string literal. The string literal must not begin with a wildcard; if the right-hand side begins with a wildcard character then this optimization is attempted. If the right-hand side is a parameter that is bound to a string, then this optimization is only attempted if the prepared statement containing the expression was compiled with sqlite3_prepare_v2() or sqlite3_prepare16_v2(). The LIKE optimization is not attempted if the right-hand side is a parameter and the statement was prepared using sqlite3_prepare() or sqlite3_prepare16(). The LIKE optimization is not attempted if there is an ESCAPE phrase on the LIKE operator.

Suppose the initial sequence of non-wildcard characters on the right-hand side of the LIKE or GLOB operator is *x*. We are using a single character to denote this non-wildcard prefix but the reader should understand that the prefix can consist of more than 1 character. Let *y* be the smallest string that is the same length as /x/

but which compares greater than *x*. For example, if *x* is **`hello`** then *y* would be **`hellp`**. The LIKE and GLOB optimizations consist of adding two virtual terms like this:

```
column >= x AND column < y
```

Under most circumstances, the original LIKE or GLOB operator is still tested against each input row even if the virtual terms are used to constrain an index. This is because we do not know what additional constraints may be imposed by characters to the right of the *x* prefix. However, if there is only a single global wildcard to the right of *x*, then the original LIKE or GLOB test is disabled. In other words, if the pattern is like this:

```
column LIKE x%
column GLOB x*
```

then the original LIKE or GLOB tests are disabled when the virtual terms constrain an index because in that case we know that all of the rows selected by the index will pass the LIKE or GLOB test.

Note that when the right-hand side of a LIKE or GLOB operator is a parameter and the statement is prepared using sqlite3_prepare_v2() or sqlite3_prepare16_v2() then the statement is automatically reparsed and recompiled on the first sqlite3_step() call of each run if the binding to the right-hand side parameter has changed since the previous run. This reparse and recompile is essentially the same action that occurs following a schema change. The recompile is necessary so that the query planner can examine the new value bound to the right-hand side of the LIKE or GLOB operator and determine whether or not to employ the optimization described above.

5.0 The Skip-Scan Optimization

The general rule is that indexes are only useful if there are WHERE-clause constraints on the left-most columns of the index. However, in some cases, SQLite is able to use an index even if the first few columns of the index are omitted from the WHERE clause but later columns are included.

Consider a table such as the following:

```
CREATE TABLE people(
  name TEXT PRIMARY KEY,
  role TEXT NOT NULL,
  height INT NOT NULL, -- in cm
  CHECK( role IN ('student','teacher') )
);
CREATE INDEX people_idx1 ON people(role, height);
```

The people table has one entry for each person in a large organization. Each person is either a "student" or a "teacher", as determined by the "role" field. And we record the height in centimeters of each person. The role and height are indexed. Notice that the left-most column of the index is not very selective - it only contains two possible values.

Now consider a query to find the names of everyone in the organization that is 180cm tall or taller:

```
SELECT name FROM people WHERE height>=180;
```

Because the left-most column of the index does not appear in the WHERE clause of the query, one is tempted to conclude that the index is not usable here. But SQLite is able to use the index. Conceptually, SQLite uses the index as if the query were more like the following:

```
SELECT name FROM people
 WHERE role IN (SELECT DISTINCT role FROM people)
   AND height>=180;
```

Or this:

```
SELECT name FROM people WHERE role='teacher' AND height>=180
UNION ALL
SELECT name FROM people WHERE role='student' AND height>=180;
```

The alternative query formulations shown above are conceptual only. SQLite does not really transform the query. The actual query plan is like this: SQLite locates the first possible value for "role", which it can do by rewinding the "people_idx1" index to the beginning and reading the first record. SQLite stores this first "role" value in an internal variable that we will here call "$role". Then SQLite runs a query like: "SELECT name FROM people WHERE role=$role AND height>=180". This query has an equality constraint on the left-most column of the index and so the index can be used to resolve that query. Once that query is finished, SQLite then uses the "people_idx1" index to locate the next value of the "role" column, using code that is logically similar to "SELECT role FROM people WHERE role>$role LIMIT 1". This new "role" value overwrites the $role variable, and the process repeats until all possible values for "role" have been examined.

We call this kind of index usage a "skip-scan" because the database engine is basically doing a full scan of the index but it optimizes the scan (making it less than "full") by occasionally skipping ahead to the next candidate value.

SQLite might use a skip-scan on an index if it knows that the first one or more columns contain many duplication values. If there are too few duplicates in the left-most columns of the index, then it would be faster to simply step ahead to the next value, and thus do a full table scan, than to do a binary search on an index to locate the next left-column value.

The only way that SQLite can know that the left-most columns of an index have many duplicate is if the ANALYZE command has been run on the database. Without the results of ANALYZE, SQLite has to guess at the "shape" of the data in the table, and the default guess is that there are an average of 10 duplicates for every value in the left-most column of the index. But skip-scan only becomes profitable (it only gets to be faster than a full table scan) when the number of duplicates is about 18 or more. Hence, a skip-scan is never used on a database that has not been analyzed.

6.0 Joins

The ON and USING clauses of an inner join are converted into additional terms of the WHERE clause prior to WHERE clause analysis described above in paragraph 1.0. Thus with SQLite, there is no computational advantage to use the newer SQL92 join syntax over the older SQL89 comma-join syntax. They both end up accomplishing exactly the same thing on inner joins.

For a LEFT OUTER JOIN the situation is more complex. The following two queries are not equivalent:

```
SELECT * FROM tab1 LEFT JOIN tab2 ON tab1.x=tab2.y;
SELECT * FROM tab1 LEFT JOIN tab2 WHERE tab1.x=tab2.y;
```

For an inner join, the two queries above would be identical. But special processing applies to the ON and USING clauses of an OUTER join: specifically, the constraints in an ON or USING clause do not apply if the right table of the join is on a null row, but the constraints do apply in the WHERE clause. The net effect is that putting the ON or USING clause expressions for a LEFT JOIN in the WHERE clause effectively converts the query to an ordinary INNER JOIN - albeit an inner join that runs more slowly.

6.1 Order of tables in a join

The current implementation of SQLite uses only loop joins. That is to say, joins are implemented as nested loops.

The default order of the nested loops in a join is for the left-most table in the FROM clause to form the outer loop and the right-most table to form the inner loop. However, SQLite will nest the loops in a different order if doing so will help it to select better indices.

Inner joins can be freely reordered. However a left outer join is neither commutative nor associative and hence will not be reordered. Inner joins to the left and right of the outer join might be reordered if the optimizer thinks that is advantageous but the outer joins are always evaluated in the order in which they occur.

SQLite treats the CROSS JOIN operator specially. The CROSS JOIN operator is commutative in theory. But SQLite chooses to never reorder tables in a CROSS JOIN. This provides a mechanism by which the programmer can force SQLite to choose a particular loop nesting order.

When selecting the order of tables in a join, SQLite uses an efficient polynomial-time algorithm. Because of this, SQLite is able to plan queries with 50- or 60-way joins in a matter of microseconds.

Join reordering is automatic and usually works well enough that programmers do not have to think about it, especially if ANALYZE has been used to gather statistics about the available indices. But occasionally some hints from the programmer are needed. Consider, for example, the following schema:

```
CREATE TABLE node(
   id INTEGER PRIMARY KEY,
   name TEXT
);
CREATE INDEX node_idx ON node(name);
CREATE TABLE edge(
   orig INTEGER REFERENCES node,
   dest INTEGER REFERENCES node,
   PRIMARY KEY(orig, dest)
);
CREATE INDEX edge_idx ON edge(dest,orig);
```

The schema above defines a directed graph with the ability to store a name at each node. Now consider a query against this schema:

```
SELECT *
  FROM edge AS e,
       node AS n1,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

This query asks for is all information about edges that go from nodes labeled "alice" to nodes labeled "bob". The query optimizer in SQLite has basically two choices on how to implement this query. (There are actually six different choices, but we will only consider two of them here.) Pseudocode below demonstrating these two choices.

Option 1:

```
foreach n1 where n1.name='alice' do:
  foreach n2 where n2.name='bob' do:
    foreach e where e.orig=n1.id and e.dest=n2.id
      return n1.*, n2.*, e.*
    end
  end
end
```

Option 2:

```
  foreach n1 where n1.name='alice' do:
    foreach e where e.orig=n1.id do:
      foreach n2 where n2.id=e.dest and n2.name='bob' do:
        return n1.*, n2.*, e.*
      end
    end
  end
```

The same indices are used to speed up every loop in both implementation options. The only difference in these two query plans is the order in which the loops are nested.

So which query plan is better? It turns out that the answer depends on what kind of data is found in the node and edge tables.

Let the number of alice nodes be M and the number of bob nodes be N. Consider two scenarios. In the first scenario, M and N are both 2 but there are thousands of edges on each node. In this case, option 1 is preferred. With option 1, the inner loop checks for the existence of an edge between a pair of nodes and outputs the result if found. But because there are only 2 alice and bob nodes each, the inner loop only has to run 4 times and the query is very quick. Option 2 would take much longer here. The outer loop of option 2 only executes twice, but because there are a large number of edges leaving each alice node, the middle loop has to iterate many thousands of times. It will be much slower. So in the first scenario, we prefer to use option 1.

Now consider the case where M and N are both 3500. Alice nodes are abundant. But suppose each of these nodes is connected by only one or two edges. In this case, option 2 is preferred. With option 2, the outer loop still has to run 3500 times, but the middle loop only runs once or twice for each outer loop and the inner loop will only run once for each middle loop, if at all. So the total number of iterations of the inner loop is around 7000. Option 1, on the other hand, has to run both its outer loop and its middle loop 3500 times each, resulting in 12 million iterations of the middle loop. Thus in the second scenario, option 2 is nearly 2000 times faster than option 1.

So you can see that depending on how the data is structured in the table, either query plan 1 or query plan 2 might be better. Which plan does SQLite choose by default? As of version 3.6.18, without running ANALYZE, SQLite will choose option 2. But if the ANALYZE command is run in order to gather statistics, a different choice might be made if the statistics indicate that the alternative is likely to run faster.

6.2 Manual Control Of Query Plans Using SQLITE_STAT Tables

SQLite provides the ability for advanced programmers to exercise control over the query plan chosen by the optimizer. One method for doing this is to fudge the ANALYZE results in the sqlite_stat1, sqlite_stat3, and/or sqlite_stat4 tables. That approach is not recommended except for the one scenario described in the next paragraph.

For a program that uses an SQLite database as its application file-format, when a new database instance is first created the ANALYZE command is ineffective because the database contain no data from which to gather statistics. In that case, one could construct a large prototype database containing typical data during development and run the ANALYZE command on this prototype database to gather statistics, then save the prototype statistics as part of the application. After deployment, when the application goes to create a new database file, it can run the ANALYZE command in order to create the statistics tables, then copy the precomputed statistics obtained from the prototype database into these new statistics tables. In that way, statistics from large working data sets can be preloaded into newly created application files.

6.3 Manual Control Of Query Plans Using CROSS JOIN

Programmers can force SQLite to use a particular loop nesting order for a join by using the CROSS JOIN operator instead of just JOIN, INNER JOIN, NATURAL JOIN, or a "," join. Though CROSS JOINs are commutative in theory, SQLite chooses to never reorder the tables in a CROSS JOIN. Hence, the left table of a CROSS JOIN will always be in an outer loop relative to the right table.

In the following query, the optimizer is free to reorder the tables of FROM clause anyway it sees fit:

```
SELECT *
  FROM node AS n1,
       edge AS e,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

But in the following logically equivalent formulation of the same query, the substitution of "CROSS JOIN" for the "," means that the order of tables must be N1, E, N2.

```
SELECT *
  FROM node AS n1 CROSS JOIN
       edge AS e CROSS JOIN
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

In the latter query, the query plan must be option 2. Note that you must use the keyword "CROSS" in order to disable the table reordering optimization; INNER JOIN, NATURAL JOIN, JOIN, and other similar combinations work just like a comma join in that the optimizer is free to reorder tables as it sees fit. (Table reordering is also disabled on an outer join, but that is because outer joins are not associative or commutative. Reordering tables in OUTER JOIN changes the result.)

See "The Fossil NGQP Upgrade Case Study" for another real-world example of using CROSS JOIN to manually control the nesting order of a join. The query planner checklist found later in the same document provides further guidance on manual control of the query planner.

7.0 Choosing between multiple indices

Each table in the FROM clause of a query can use at most one index (except when the OR-clause optimization comes into play) and SQLite strives to use at least one index on each table. Sometimes, two or more indices might be candidates for use on a single table. For example:

```
CREATE TABLE ex2(x,y,z);
CREATE INDEX ex2i1 ON ex2(x);
CREATE INDEX ex2i2 ON ex2(y);
SELECT z FROM ex2 WHERE x=5 AND y=6;
```

For the SELECT statement above, the optimizer can use the ex2i1 index to lookup rows of ex2 that contain x=5 and then test each row against the y=6 term. Or it can use the ex2i2 index to lookup rows of ex2 that contain y=6 then test each of those rows against the x=5 term.

When faced with a choice of two or more indices, SQLite tries to estimate the total amount of work needed to perform the query using each option. It then selects the option that gives the least estimated work.

To help the optimizer get a more accurate estimate of the work involved in using various indices, the user may optionally run the ANALYZE command. The ANALYZE command scans all indices of database where there might be a choice between two or more indices and gathers statistics on the selectiveness of those indices. The statistics gathered by this scan are stored in special database tables names shows names all begin with "**sqlite_stat**". The content of these tables is not updated as the database changes so after making significant changes it might be prudent to rerun ANALYZE. The results of an ANALYZE command are only available to database connections that are opened after the ANALYZE command completes.

The various **sqlite_stat*N*** tables contain information on how selective the various indices are. For example, the sqlite_stat1 table might indicate that an equality constraint on column x reduces the search space to 10 rows on average, whereas an equality constraint on column y reduces the search space to 3 rows on average. In that case, SQLite would prefer to use index ex2i2 since that index is more selective.

7.1 Disqualifying WHERE Clause Terms Using Unary-"+"

Terms of the WHERE clause can be manually disqualified for use with indices by prepending a unary **+** operator to the column name. The unary **+** is a no-op and will not generate any byte code in the prepared statement. But the unary **+** operator will prevent the term from constraining an index. So, in the example above, if the query were rewritten as:

```
SELECT z FROM ex2 WHERE +x=5 AND y=6;
```

The **+** operator on the **x** column will prevent that term from constraining an index. This would force the use of the ex2i2 index.

Note that the unary **+** operator also removes type affinity from an expression, and in some cases this can cause subtle changes in the meaning of an expression. In the example above, if column **x** has TEXT affinity then the comparison "x=5" will be done as text. But the **+** operator removes the affinity. So the comparison "+x=5" will compare the text in column **x** with the numeric value 5 and will always be false.

7.2 Range Queries

Consider a slightly different scenario:

```
CREATE TABLE ex2(x,y,z);
CREATE INDEX ex2i1 ON ex2(x);
CREATE INDEX ex2i2 ON ex2(y);
SELECT z FROM ex2 WHERE x BETWEEN 1 AND 100 AND y BETWEEN 1 AND 100;
```

Further suppose that column x contains values spread out between 0 and 1,000,000 and column y contains values that span between 0 and 1,000. In that scenario, the range constraint on column x should reduce the search space by a factor of 10,000 whereas the range constraint on column y should reduce the search space by a factor of only 10. So the ex2i1 index should be preferred.

SQLite will make this determination, but only if it has been compiled with

SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4. The SQLITE_ENABLE_STAT3 and SQLITE_ENABLE_STAT4 options causes the ANALYZE command to collect a histogram of column content in the sqlite_stat3 or sqlite_stat4 tables and to use this histogram to make a better guess at the best query to use for range constraints such as the above. The main difference between STAT3 and STAT4 is that STAT3 records histogram data for only the left-most column of an index whereas STAT4 records histogram data for all columns of an index. For single-column indexes, STAT3 and STAT4 work the same.

The histogram data is only useful if the right-hand side of the constraint is a simple compile-time constant or parameter and not an expression.

Another limitation of the histogram data is that it only applies to the left-most column on an index. Consider this scenario:

```
CREATE TABLE ex3(w,x,y,z);
CREATE INDEX ex3i1 ON ex2(w, x);
CREATE INDEX ex3i2 ON ex2(w, y);
SELECT z FROM ex3 WHERE w=5 AND x BETWEEN 1 AND 100 AND y BETWEEN 1 AND 100;
```

Here the inequalities are on columns x and y which are not the left-most index columns. Hence, the histogram data which is collected no left-most column of indices is useless in helping to choose between the range constraints on columns x and y.

8.0 Covering Indices

When doing an indexed lookup of a row, the usual procedure is to do a binary search on the index to find the index entry, then extract the rowid from the index and use that rowid to do a binary search on the original table. Thus a typical indexed lookup involves two binary searches. If, however, all columns that were to be fetched from the table are already available in the index itself, SQLite will use the values contained in the index and will never look up the original table row. This saves one binary search for each row and can make many queries run twice as fast.

When an index contains all of the data needed for a query and when the original table never needs to be consulted, we call that index a "covering index".

9.0 ORDER BY optimizations

SQLite attempts to use an index to satisfy the ORDER BY clause of a query when possible. When faced with the choice of using an index to satisfy WHERE clause constraints or satisfying an ORDER BY clause, SQLite does the same cost analysis described above and chooses the index that it believes will result in the fastest answer.

SQLite will also attempt to use indices to help satisfy GROUP BY clauses and the DISTINCT keyword. If the nested loops of the join can be arranged such that rows that are equivalent for the GROUP BY or for the DISTINCT are consecutive, then the GROUP BY or DISTINCT logic can determine if the current row is part of the same group or if the current row is distinct simply by comparing the current row to the previous row. This can be much faster than the alternative of comparing each row to all prior rows.

9.1 Partial ORDER BY Via Index

If a query contains an ORDER BY clause with multiple terms, it might be that SQLite can use indices to cause rows to come out in the order of some prefix of the terms in the ORDER BY but that later terms in the ORDER BY are not satisfied. In that case, SQLite does block sorting. Suppose the ORDER BY clause has four terms and the natural order of the query results in rows appearing in order of the first two terms. As each row is output by the query engine and enters the sorter, the outputs in the current row corresponding to the first two terms of the ORDER BY are compared against the previous row. If they have changed, the current sort is finished and output and a new sort is started. This results in a slightly faster sort. But the bigger advantages are that many fewer rows need to be held in memory, reducing memory requirements, and outputs can begin to appear before the core query has run to completion.

10.0 Subquery flattening

When a subquery occurs in the FROM clause of a SELECT, the simplest behavior is to evaluate the subquery into a transient table, then run the outer SELECT against the transient table. But such a plan can be suboptimal since the transient table will not have any indices and the outer query (which is likely a join) will be forced to do a full table scan on the transient table.

To overcome this problem, SQLite attempts to flatten subqueries in the FROM clause of a SELECT. This involves inserting the FROM clause of the subquery into the FROM clause of the outer query and rewriting expressions in the outer query that refer to the result set of the subquery. For example:

```
SELECT a FROM (SELECT x+y AS a FROM t1 WHERE z<100) WHERE a>5
```

Would be rewritten using query flattening as:

```
SELECT x+y AS a FROM t1 WHERE z<100 AND a>5
```

There is a long list of conditions that must all be met in order for query flattening to occur. Some of the constraints are marked as obsolete by italic text. These extra constraints are retained in the documentation to preserve the numbering of the other constraints.

1. The subquery and the outer query do not both use aggregates.
2. The subquery is not an aggregate or the outer query is not a join.
3. The subquery is not the right operand of a left outer join.
4. The subquery is not DISTINCT.
5. *(Subsumed into constraint 4)*
6. The subquery does not use aggregates or the outer query is not DISTINCT.
7. The subquery has a FROM clause.
8. The subquery does not use LIMIT or the outer query is not a join.
9. The subquery does not use LIMIT or the outer query does not use aggregates.
10. The subquery does not use aggregates or the outer query does not use LIMIT.
11. The subquery and the outer query do not both have ORDER BY clauses.
12. *(Subsumed into constraint 3)*
13. The subquery and outer query do not both use LIMIT.
14. The subquery does not use OFFSET.
15. The outer query is not part of a compound select or the subquery does not have a LIMIT clause.
16. The outer query is not an aggregate or the subquery does not contain ORDER BY.
17. The sub-query is not a compound select, or it is a UNION ALL compound clause made up entirely of non-aggregate queries, and the parent query:
    - is not itself part of a compound select,
    - is not an aggregate or DISTINCT query, and
    - is not a join.

    The parent and sub-query may contain WHERE clauses. Subject to rules (11), (12) and (13), they may also contain ORDER BY, LIMIT and OFFSET clauses.
18. If the sub-query is a compound select, then all terms of the ORDER by clause of the parent must be simple references to columns of the sub-query.
19. The subquery does not use LIMIT or the outer query does not have a WHERE clause.
20. If the sub-query is a compound select, then it must not use an ORDER BY clause.
21. The subquery does not use LIMIT or the outer query is not DISTINCT.
22. The subquery is not a recursive CTE.
23. The parent is not a recursive CTE, or the sub-query is not a compound query.

The casual reader is not expected to understand or remember any part of the list above. The point of this list is to demonstrate that the decision of whether or not to flatten a query is complex.

Query flattening is an important optimization when views are used as each use of a view is translated into a subquery.

11.0 The MIN/MAX optimization

Queries that contain a single MIN() or MAX() aggregate function whose argument is the left-most column of an index might be satisfied by doing a single index lookup rather than by scanning the entire table. Examples:

```
SELECT MIN(x) FROM table;
SELECT MAX(x)+1 FROM table;
```

12.0 Automatic Indexes

When no indices are available to aid the evaluation of a query, SQLite might create an automatic index that lasts only for the duration of a single SQL statement. Since the cost of constructing the automatic index is O(NlogN) (where N is the number of entries in the table) and the cost of doing a full table scan is only O(N), an automatic index will only be created if SQLite expects that the lookup will be run more than logN times during the course of the SQL statement. Consider an example:

```
CREATE TABLE t1(a,b);
CREATE TABLE t2(c,d);
-- Insert many rows into both t1 and t2
SELECT * FROM t1, t2 WHERE a=c;
```

In the query above, if both t1 and t2 have approximately N rows, then without any indices the query will require O(N*N) time. On the other hand, creating an index on table t2 requires O(NlogN) time and then using that index to evaluate the query requires an additional O(NlogN) time. In the absence of ANALYZE information, SQLite guesses that N is one million and hence it believes that constructing the automatic index will be the cheaper approach.

An automatic index might also be used for a subquery:

```
CREATE TABLE t1(a,b);
CREATE TABLE t2(c,d);
-- Insert many rows into both t1 and t2
SELECT a, (SELECT d FROM t2 WHERE c=b) FROM t1;
```

In this example, the t2 table is used in a subquery to translate values of the t1.b column. If each table contains N rows, SQLite expects that the subquery will run N times, and hence it will believe it is faster to construct an automatic, transient index on t2 first and then using that index to satisfy the N instances of the subquery.

The automatic indexing capability can be disabled at run-time using the automatic_index pragma. Automatic indexing is turned on by default, but this can be changed so that automatic indexing is off by default using the SQLITE_DEFAULT_AUTOMATIC_INDEX compile-time option. The ability to create automatic indices can be completely disabled by compiling with the SQLITE_OMIT_AUTOMATIC_INDEX compile-time option.

In SQLite version 3.8.0 (2013-08-26) and later, an SQLITE_WARNING_AUTOINDEX message is sent to the error log every time a statement is prepared that uses an automatic index. Application developers can and should use these warnings to identify the need for new persistent indices in the schema.

Do not confuse automatic indexes with the internal indexes (having names like "sqlite_autoindex_*table*_*N*") that are sometimes created to implement a PRIMARY KEY constraint or UNIQUE constraint. The automatic indexes described here exist only for the duration of a single query, are never persisted to disk, and are only visible to a single database connection. Internal indexes are part of the implementation of PRIMARY KEY and UNIQUE constraints, are long-lasting and persisted to disk, and are visible to all database connections. The term "autoindex" appears in the names of internal indexes for legacy reasons and does not indicate that internal indexes and automatic indexes are related.