Small. Fast. Reliable.
Choose any three.

The Next-Generation Query Planner

▼ Table Of Contents

1. Introduction

The task of the "query planner" is to figure out the best algorithm or "query plan" to accomplish an SQL statement. Beginning with SQLite version 3.8.0 (2013-08-26), the query planner component has been rewritten so that it runs faster and generates better plans. The rewrite is called the "next generation query planner" or "NGQP".

This article overviews the importance of query planning, describes some of the problems inherent to query planning, and outlines how the NGQP solves those problems.

The NGQP is almost always better than the legacy query planner. However, there may exist legacy applications that unknowingly depend on undefined and/or

suboptimal behavior in the legacy query planner, and upgrading to the NGQP on those legacy applications could cause performance regressions. This risk is considered and a checklist is provided for reducing the risk and for fixing any issues that do arise.

This document focuses on the NGQP. For a more general overview of the SQLite query planner that encompasses the entire history of SQLite, see "The SQLite Query Optimizer Overview".

2. Background

For simple queries against a single table with few indices, there is usually an obvious choice for the best algorithm. But for larger and more complex queries, such as multi-way joins with many indices and subqueries, there can be hundreds, thousands, or millions of reasonable algorithms for computing the result. The job of the query planner is to choose the single "best" query plan from this multitude of possibilities.

Query planners are what make SQL database engines so amazingly useful and powerful. (This is true of all SQL database engines, not just SQLite.) The query planner frees the programmer from the chore of selecting a particular query plan, and thereby allows the programmer to focus more mental energy on higher-level application issues and on providing more value to the end user. For simple queries where the choice of query plan is obvious, this is convenient but not hugely important. But as applications and schemas and queries grow more complex, a clever query planner can greatly speed and simplify the work of application development. There is amazing power in being about to tell the database engine what content is desired, and then let the database engine figure out the best way to retrieve that content.

Writing a good query planner is more art than science. The query planner must work with incomplete information. It cannot determine how long any particular plan will take without actually running that plan. So when comparing two or more plans to figure out which is "best", the query planner has to make some guesses and assumptions and those guesses and assumptions will sometimes be wrong. A good query planner is one that will find the correct solution often enough that application programmers rarely need to get involved.

2.1. Query Planning In SQLite

SQLite computes joins using nested loops, one loop for each table in the join. (Additional loops might be inserted for IN and OR operators in the WHERE clause.

SQLite considers those too, but for simplicity we will ignore them in this essay.) One or more indices might be used on each loop to speed the search, or a loop might be a "full table scan" that reads every row in the table. Thus query planning decomposes into two subtasks:

1. Picking the nested order of the various loops
2. Choosing good indices for each loop

Picking the nesting order is generally the more challenging problem. Once the nesting order of the join is established, the choice of indices for each loop is normally obvious.

2.2. The SQLite Query Planner Stability Guarantee

SQLite will always pick the same query plan for any given SQL statement as long as:

a. the database schema does not change in significant ways such as adding or dropping indices,
b. the ANALYZE command is not rerun,
c. SQLite is not compiled with SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4, and
d. the same version of SQLite is used.

The SQLite stability guarantee means that if all of your queries run efficiently during testing, and if your application does not change the schema, then SQLite will not suddenly decide to start using a different query plan, possibly causing a performance problem, after your application is released to users. If your application works in the lab, it will continue working the same way after deployment.

Enterprise-class client/server SQL database engines do not normally make this guarantee. In client/server SQL database engines, the server keeps track of statistics on the sizes of tables and on the quality of indices and the query planner uses those statistics to help select the best plans. As content is added, deleted, or changed in the database, the statistics will evolve and may cause the query planner to begin using a different query plan for some particular query. Usually the new plan will be better for the evolving structure of the data. But sometimes the new query plan will cause a performance reduction. With a client/server database engine, there is typically a Database Administrator (DBA) on hand to deal with these rare problems as they come up. But DBAs are not available to fix problems in an embedded database like SQLite, and hence SQLite is careful to ensure that plans do not change unexpectedly after deployment.

The SQLite stability guarantee applies equally to the legacy query planner and to the NGQP.

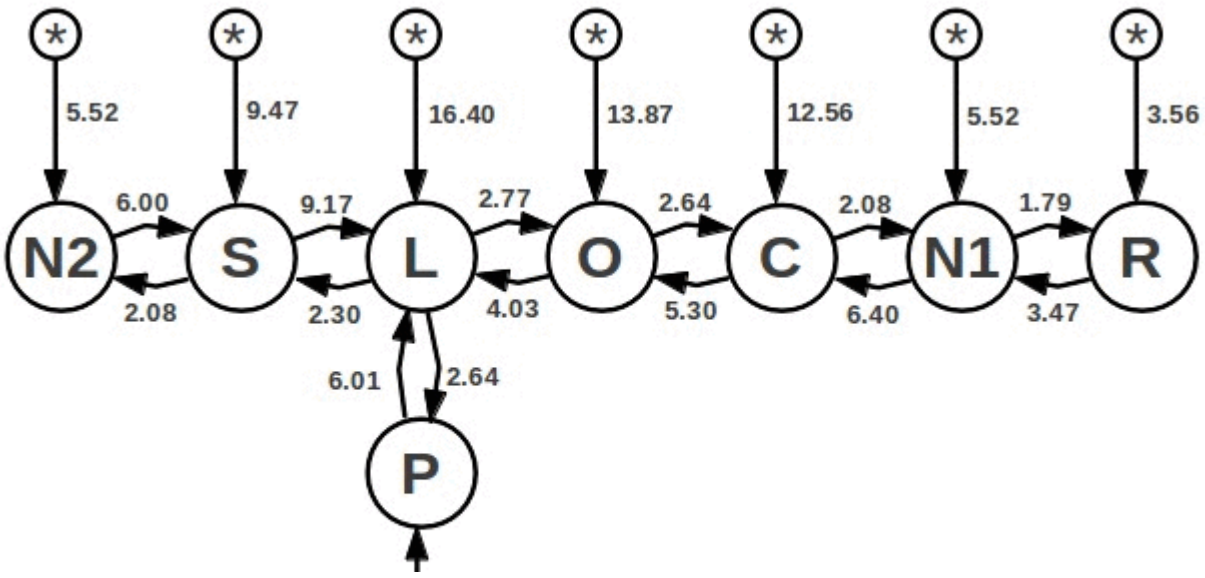
It is important to note that changing versions of SQLite might cause changes in query plans. The same version of SQLite will always pick the same query plan, but if you relink your application to use a different version of SQLite, then query plans might change. In rare cases, an SQLite version change might lead to a performance regression. This is one reason you should consider statically linking your applications against SQLite rather than use a system-wide SQLite shared library which might change without your knowledge or control.

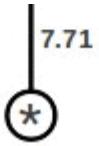
3. A Difficult Case

"TPC-H Q8" is a test query from the Transaction Processing Performance Council. The query planners in SQLite versions 3.7.17 and earlier do not choose good plans for TPC-H Q8. And it has been determined that no amount of tweaking of the legacy query planner will fix that. In order to find a good solution to the TPC-H Q8 query, and to continue improving the quality of SQLite's query planner, it became necessary to redesign the query planner. This section tries to explain why this redesign was necessary and how the NGQP is different and addresses the TPC-H Q8 problem.

3.1. Query Details

TPC-H Q8 is an eight-way join. As observed above, the main task of the query planner is to figure out the best nesting order of the eight loops in order to minimize the work needed to complete the join. A simplified model of this problem for the case of TPC-H Q8 is shown by the following diagram:





In the diagram, each of the 8 tables in the FROM clause of the query is identified by a large circle with the label of the FROM-clause term: N2, S, L, P, O, C, N1 and R. The arcs in the graph represent the estimated cost of computing each term assuming that the origin of the arc is in an outer loop. For example, the cost of running the S loop as an inner loop to L is 2.30 whereas the cost of running the S loop as an outer loop to L is 9.17.

The "cost" here is logarithmic. With nested loops, the work is multiplied, not added. But it is customary to think of graphs with additive weights and so the graph shows the logarithm of the various costs. The graph shows a cost advantage of S being inside of L of about 6.87, but this translates into the query running about 963 times faster when S loop is inside of the L loop rather than being outside of it.

The arrows from the small circles labeled with "*" indicate the cost of running each loop with no dependencies. The outer loop must use this *-cost. Inner loops have the option of using the *-cost or a cost assuming one of the other terms is in an outer loop, whichever gives the best result. One can think of the *-costs as a short-hand notation indicating multiple arcs, one from each of the other nodes in the graph. The graph is therefore "complete", meaning that there are arcs (some explicit and some implied) in both directions between every pair of nodes in the graph.

The problem of finding the best query plan is equivalent to finding a minimum-cost path through the graph that visits each node exactly once.

(Side note: The cost estimates in the TPC-H Q8 graph above were computed by the query planner in SQLite 3.7.16 and converted using a natural logarithm.)

3.2. Complications

The presentation of the query planner problem above is a simplification. The costs are estimates. We cannot know what the true cost of running a loop is until we actually run the loop. SQLite makes guesses for the cost of running a loop based on the availability of indices and constraints found in the WHERE clause. These guesses are usually pretty good, but they can sometimes be off. Using the ANALYZE command to collect additional statistical information about the database can sometimes enable SQLite to make better guesses about the cost.

The costs are comprised of multiple numbers, not a single number as shown in the graph. SQLite computes several different estimated costs for each loop that apply at different times. For example, there is a "setup" cost that is incurred just once when the query starts. The setup cost is the cost of computing an automatic index for a table that does not already have an index. Then there is the cost of running each step of the loop. Finally, there is an estimate of the number rows generated by the loop, which is information needed in estimating the costs of inner loops. Sorting costs may come into play if the query has an ORDER BY clause.

In a general query, dependencies need not be on a single loop, and hence the matrix of dependencies might not be representable as a graph. For example, one of the WHERE clause constraints might be S.a=L.b+P.c, implying that the S loop must be an inner loop of both L and P. Such dependencies cannot be drawn as a graph since there is no way for an arc to originate at two or more nodes at once.

If the query contains an ORDER BY clause or a GROUP BY clause or if the query uses the DISTINCT keyword then it is advantageous to select a path through the graph that causes rows to naturally appear in sorted order, so that no separate sorting step is required. Automatic elimination of ORDER BY clauses can make a large performance difference, so this is another factor that needs to be considered in a complete implementation.

In the TPC-H Q8 query, the setup costs are all negligible, all dependencies are between individual nodes, and there is no ORDER BY, GROUP BY, or DISTINCT clause. So for TPC-H Q8, the graph above is a reasonable representation of what needs to be computed. The general case involves a lot of extra complication, which for clarity is neglected in the remainder of this article.

3.3. Finding The Best Query Plan

Prior to version 3.8.0 (2013-08-26), SQLite always used the "Nearest Neighbor" or "NN" heuristic when searching for the best query plan. The NN heuristic makes a single traversal of the graph, always choosing the lowest-cost arc as the next step. The NN heuristic works surprisingly well in most cases. And NN is fast, so that SQLite is able to quickly find good plans for even large 64-way joins. In contrast, other SQL database engines that do more extensive searching tend to bog down when the number of tables in a join goes above 10 or 15.

Unfortunately, the query plan computed by NN for TPC-H Q8 is not optimal. The plan computed using NN is R-N1-N2-S-C-O-L-P with a cost of 36.92. The notation in the previous sentence means that the R table is run in the outer loop, N1 is in the next inner loop, N2 is in the third loop, and so forth down to P which is in the inner-most loop. The shortest path through the graph (as found via exhaustive search) is P-L-O-C-N1-R-S-N2 with a cost of 27.38. The difference might not seem

like much, but remember that the costs are logarithmic, so the shortest path is nearly 750 times faster than that path found using the NN heuristic.

One solution to this problem is to change SQLite to do an exhaustive search for the best path. But an exhaustive search requires time proportional to K! (where K is the number of tables in the join) and so when you get beyond a 10-way join, the time to run sqlite3_prepare() becomes very large.

3.4. The N Nearest Neighbors or "N3" Heuristic

The NGQP uses a new heuristic for seeking the best path through the graph: "N Nearest Neighbors" (hereafter "N3"). With N3, instead of choosing just one nearest neighbor for each step, the algorithm keeps track of the N bests paths at each step for some small integer N.

Suppose N=4. Then for the TPC-H Q8 graph, the first step finds the four shortest paths to visit any single node in the graph:

R (cost: 3.56)
N1 (cost: 5.52)
N2 (cost: 5.52)
P (cost: 7.71)

The second step finds the four shortest paths to visit two nodes beginning with one of the four paths from the previous step. In the case where two or more paths are equivalent (they have the same set of visited nodes, though possibly in a different order) only the first and lowest-cost path is retained. We have:

R-N1 (cost: 7.03)
R-N2 (cost: 9.08)
N2-N1 (cost: 11.04)
R-P {cost: 11.27}

The third step starts with the four shortest two-node paths and finds the four shortest three-node paths:

R-N1-N2 (cost: 12.55)
R-N1-C (cost: 13.43)
R-N1-P (cost: 14.74)
R-N2-S (cost: 15.08)

And so forth. There are 8 nodes in the TPC-H Q8 query, so this process repeats a total of 8 times. In the general case of a K-way join, the storage requirement is O(N) and the computation time is O(K*N), which is significantly faster than the $O(2^K)$ exact solution.

But what value to choose for N? One might try N=K. This makes the algorithm $O(K^2)$ which is actually still quite efficient, since the maximum value of K is 64 and K rarely exceeds 10. But that is not enough for the TPC-H Q8 problem. With N=8 on TPC-H Q8 the N3 algorithm finds the solution R-N1-C-O-L-S-N2-P with a cost of 29.78. That is a big improvement over NN, but it is still not optimal. N3 finds the optimal solution for TPC-H Q8 when N is 10 or greater.

The initial implementation of NGQP chooses N=1 for simple queries, N=5 for two-way joins and N=10 for all joins with three or more tables. This formula for selecting N might change in subsequent releases.

4. Hazards Of Upgrading To NGQP

For most applications, upgrading from the legacy query planner to the NGQP requires little thought or effort. Simply replace the older SQLite version with the newer version of SQLite and recompile and the application will run faster. There are no API changes nor modifications to compilation procedures.

But as with any query planner change, upgrading to the NGQP does carry a small risk of introducing performance regressions. The problem here is not that the NGQP is incorrect or buggy or inferior to the legacy query planner. Given reliable information about the selectivity of indices, the NGQP should always pick a plan that is as good or better than before. The problem is that some applications may be using low-quality and low-selectivity indices without having run ANALYZE. The older query planners look at many fewer possible implementations for each query and so they may have stumbled over a good plan by stupid luck. The NGQP, on the other hand, looks at many more query plan possibilities, and it may choose a different query plan that works better in theory, assuming good indices, but which gives a performance regression in practice, because of the shape of the data.

Key points:

- The NGQP will always find an equal or better query plan, compared to prior query planners, as long as it has access to accurate ANALYZE data in the SQLITE_STAT1 file.
- The NGQP will always find a good query plan as long as the schema does not contain indices that have more than about 10 or 20 rows with the same value in the left-most column of the index.

Not all applications meet these conditions. Fortunately, the NGQP will still usually find good query plans, even without these conditions. However, cases do arise (rarely) where performance regressions can occur.

4.1. Case Study: Upgrading Fossil to the NGQP

The Fossil DVCS is the version control system used to track all of the SQLite source code. A Fossil repository is an SQLite database file. (Readers are invited to ponder this recursion as an independent exercise.) Fossil is both the version-control system for SQLite and a test platform for SQLite. Whenever enhancements are made to SQLite, Fossil is one of the first applications to test and evaluate those enhancements. So Fossil was an early adopter of the NGQP.

Unfortunately, the NGQP caused a performance regression in Fossil.

One of the many reports that Fossil makes available is a timeline of changes to a single branch showing all merges in and out of that branch. See http://www.sqlite.org/src/timeline?nd&n=200&r=trunk for a typical example of such a report. Generating such a report normally takes just a few milliseconds. But after upgrading to the NGQP we noticed that this one report was taking closer to 10 seconds for the trunk of the repository.

The core query used to generate the branch timeline is shown below. (Readers are not expected to understand the details of this query. Commentary will follow.)

```
SELECT
     blob.rid AS blobRid,
     uuid AS uuid,
     datetime(event.mtime,'localtime') AS timestamp,
     coalesce(ecomment, comment) AS comment,
     coalesce(euser, user) AS user,
     blob.rid IN leaf AS leaf,
     bgcolor AS bgColor,
     event.type AS eventType,
     (SELECT group_concat(substr(tagname,5), ', ')
        FROM tag, tagxref
       WHERE tagname GLOB 'sym-*'
         AND tag.tagid=tagxref.tagid
         AND tagxref.rid=blob.rid
         AND tagxref.tagtype>0) AS tags,
     tagid AS tagid,
     brief AS brief,
     event.mtime AS mtime
  FROM event CROSS JOIN blob
 WHERE blob.rid=event.objid
   AND (EXISTS(SELECT 1 FROM tagxref
                WHERE tagid=11 AND tagtype>0 AND rid=blob.rid)
        OR EXISTS(SELECT 1 FROM plink JOIN tagxref ON rid=cid
                   WHERE tagid=11 AND tagtype>0 AND pid=blob.rid)
        OR EXISTS(SELECT 1 FROM plink JOIN tagxref ON rid=pid
                   WHERE tagid=11 AND tagtype>0 AND cid=blob.rid))
 ORDER BY event.mtime DESC
 LIMIT 200;
```

This query is not especially complicated, but even so it replaces hundreds or perhaps thousands of lines of procedural code. The gist of the query is this: Scan down the EVENT table looking for the most recent 200 check-ins that satisfy any

one of three conditions:

1. The check-in has a "trunk" tag.
2. The check-in has a child that has a "trunk" tag.
3. The check-in has a parent that has a "trunk" tag.

The first condition causes all of the trunk check-ins to be displayed and the second and third cause check-ins that merge into or fork from the trunk to also be included. The three conditions are implemented by the three OR-connected EXISTS statements in the WHERE clause of the query. The slowdown that occurred with the NGQP was caused by the second and third conditions. The problem is the same in each, so we will examine just the second one. The subquery of the second condition can be rewritten (with minor and immaterial simplifications) as follows:

```
SELECT 1
  FROM plink JOIN tagxref ON tagxref.rid=plink.cid
 WHERE tagxref.tagid=$trunk
   AND plink.pid=$ckid;
```

The PLINK table holds parent-child relationships between check-ins. The TAGXREF table maps tags into check-ins. For reference, the relevant portions of the schemas for these two tables is shown here:

```
CREATE TABLE plink(
  pid INTEGER REFERENCES blob,
  cid INTEGER REFERENCES blob
);
CREATE UNIQUE INDEX plink_i1 ON plink(pid,cid);

CREATE TABLE tagxref(
  tagid INTEGER REFERENCES tag,
  mtime TIMESTAMP,
  rid INTEGER REFERENCE blob,
  UNIQUE(rid, tagid)
);
CREATE INDEX tagxref_i1 ON tagxref(tagid, mtime);
```

There are only two reasonable ways to implement this query. (There are many other possible algorithms, but none of the others are contenders for being the "best" algorithm.)

1. Find all children of check-in $ckid and test each one to see if it has the $trunk tag.

2. Find all check-ins with the $trunk tag and test each one to see if it is a child of $ckid.

Intuitively, we humans understand that algorithm-1 is best. Each check-in is likely to have few children (one child is the most common case) and each child can be tested for the $trunk tag in logarithmic time. Indeed, algorithm-1 is the faster choice in practice. But the NGQP has no intuition. The NGQP must use hard math,

and algorithm-2 is slightly better mathematically. This is because, in the absence of other information, the NGQP must assume that the indices PLINK_I1 and TAGXREF_I1 are of equal quality and are equally selective. Algorithm-2 uses one field of the TAGXREF_I1 index and both fields of the PLINK_I1 index whereas algorithm-1 only uses the first field of each index. Since algorithm-2 uses more index material, the NGQP is correct to judge it to be the better algorithm. The scores are close and algorithm-2 just barely squeaks ahead of algorithm-1. But algorithm-2 really is the correct choice here.

Unfortunately, algorithm-2 is slower than algorithm-1 in this application.

The problem is that the indices are not of equal quality. A check-in is likely to only have one child. So the first field of PLINK_I1 will usually narrow down the search to just a single row. But there are thousands and thousands check-ins tagged with "trunk", so the first field of TAGXREF_I1 will be of little help in narrowing down the search.

The NGQP has no way of knowing that TAGXREF_I1 is almost useless in this query, unless ANALYZE has been run on the database. The ANALYZE command gathers statistics on the quality of the various indices and stores those statistics in SQLITE_STAT1 table. Having access to this statistical information, the NGQP easily chooses algorithm-1 as the best algorithm, by a wide margin.

Why didn't the legacy query planner choose algorithm-2? Easy: because the NN algorithm never even considered algorithm-2. Graphs of the planning problem look like this:

In the "without ANALYZE" case on the left, the NN algorithm chooses loop P (PLINK) as the outer loop because 4.9 is less than 5.2, resulting in path P-T which is algorithm-1. NN only looks at the single best choice at each step so it completely misses the fact that 5.2+4.4 makes a slightly cheaper plan than 4.9+4.8. But the N3 algorithm keeps track of the 5 best paths for a 2-way join, so it ends up selecting path T-P because of its slightly lower overall cost. Path T-P is

algorithm-2.

Note that with ANALYZE the cost estimates are better aligned with reality and algorithm-1 is selected by both NN and N3.

(Side note: The costs estimates in the two most recent graphs were computed by the NGQP using a base-2 logarithm and slightly different cost assumptions compared to the legacy query planner. Hence, the cost estimates in these latter two graphs are not directly comparable to the cost estimates in the TPC-H Q8 graph.)

4.2. Fixing The Problem

Running ANALYZE on the repository database immediately fixed the performance problem. However, we want Fossil to be robust and to always work quickly regardless of whether or not its repository has been analyzed. For this reason, the query was modified to use the CROSS JOIN operator instead of the plain JOIN operator. SQLite will not reorder the tables of a CROSS JOIN. This is a long-standing feature of SQLite that is specifically designed to allow knowledgeable programmers to enforce a particular loop nesting order. Once the join was changed to CROSS JOIN (the addition of a single keyword) the NGQP was forced to choose the faster algorithm-1 regardless of whether or not statistical information had been gathered using ANALYZE.

We say that algorithm-1 is "faster", but this is not strictly true. Algorithm-1 is faster in common repositories, but it is possible to construct a repository in which every check-in is on a different uniquely-named branch and all check-ins are children of the root check-in. In that case, TAGXREF_I1 would become more selective than PLINK_I1 and algorithm-2 really would be the faster choice. However such repositories are very unlikely to appear in practice and so hard-coding the loop nested order using the CROSS JOIN syntax is a reasonable solution to the problem in this case.

5. Checklist For Avoiding Or Fixing Query Planner Problems

1. **Don't panic!** Cases where the query planner picks an inferior plan are actually quite rare. You are unlikely to run across any problems in your application. If you are not having performance issues, you do not need to worry about any of this.

2. **Create appropriate indices.** Most SQL performance problems arise not

because of query planner issues but rather due to lack of appropriate indices. Make sure indices are available to assist all large queries. Most performance issues can be resolved by one or two CREATE INDEX commands and with no changes to application code.

3. **Avoid creating low-quality indices.**. A low-quality index (for the purpose of this checklist) is one where there are more than 10 or 20 rows in the table that have the same value for the left-most column of the index. In particular, avoid using boolean or "enum" columns as the left-most columns of your indices.

   The Fossil performance problem described in the previous section of this document arose because there were over ten-thousand entries in the TAGXREF table with the same value for the left-most column (the TAGID column) of the TAGXREF_I1 index.

4. **If you must use a low-quality index, be sure to run ANALYZE.** Low-quality indices will not confuse the query planner as long as the query planner knows that the indices are of low quality. And the way the query planner knows this is by the content of the SQLITE_STAT1 table, which is computed by the ANALYZE command.

   Of course, ANALYZE only works effectively if you have a significant amount of content in your database in the first place. When creating a new database that you expect to accumulate a lot of data, you can run the command "ANALYZE sqlite_master" to create the SQLITE_STAT1 table, then prepopulate the SQLITE_STAT1 table (using ordinary INSERT statements) with content that describes a typical database for your application - perhaps content that you extracted after running ANALYZE on a well-populated template database in the lab.

5. **Instrument your code.** Add logic that lets you know quickly and easily which queries are taking too much time. Then work on just those specific queries.

6. **Use unlikely() and likelihood() SQL functions.** SQLite normally assumes that terms in the WHERE clause that cannot be used by indices have a strong probability of being true. If this assumption is incorrect, it could lead to a suboptimal query plan. The unlikely() and likelihood() SQL functions can be used to provide hints to the query planner about WHERE clause terms that are probably not true, and thus aid the query planner in selecting the best possible plan.

7. **Use the CROSS JOIN syntax to enforce a particular loop nesting order on queries that might use low-quality indices in an unanalyzed**

database. SQLite treats the CROSS JOIN operator specially, forcing the table to the left to be an outer loop relative to the table on the right.

Avoid this step if possible, as it defeats one of the huge advantages of the whole SQL language concept, specifically that the application programmer does not need to get involved with query planning. If you do use CROSS JOIN, wait until late in your development cycle to do so, and comment the use of CROSS JOIN carefully so that you can take it out later if possible. Avoid using CROSS JOIN early in the development cycle as doing so is a premature optimization, which is well known to be the root of all evil.

8. **Use unary "+" operators to disqualify WHERE clause terms.** If the query planner insists on selecting a poor-quality index for a particular query when a much higher-quality index is available, then careful use of unary "+" operators in the WHERE clause can force the query planner away from the poor-quality index. Avoid using this trick if at all possible, and especially avoid it early in the application development cycle. Beware that adding a unary "+" operator to an equality expression might change the result of that expression if type affinity is involved.

9. **Use the INDEXED BY syntax to enforce the selection of particular indices on problem queries.** As with the previous two bullets, avoid this step if possible, and especially avoid doing this early in development as it is clearly a premature optimization.

6. Summary

The query planner in SQLite normally does a terrific job of selecting fast algorithms for running your SQL statements. This is true of the legacy query planner and even more true of the new NGQP. There may be an occasional situation where, due to incomplete information, the query planner selects a suboptimal plan. This will happen less often with the NGQP than with the legacy query planner, but it might still happen. Only in those rare cases do application developers need to get involved and help the query planner to do the right thing. In the common case, the NGQP is just a new enhancement to SQLite that makes the application run a little faster and which requires no new developer thought or action.