Analyzing TPC-H: Quantifying Query Results and Performance Optimizations

Jeff Bloom[1], Justin Cabral[2], Veronica Melican[3], and Catherine Merfeld[4]

[1] WPI, Worcester, Massachusetts
Jmbloom@wpi.edu
[2] WPI, Worcester, Massachusetts
jjcabral@wpi.edu
[3] WPI, Worcester, Massachusetts
vrmelican@wpi.edu
[4] WPI, Worcester, Massachusetts
cmerfeld@wpi.edu

Abstract. TPC-H is the most widely used benchmark in the industry for performing high speed, multi-dimensional analysis of large volumes of data in a database management system using 22 defined queries. The benchmark introduces choke points, such as joins across multiple tables, that can be used to evaluate the performance of the queries. Optimizing these queries can result in performance gains within the database management systems. In this paper we will quantify the results of all 22 queries based on the execution run time and perform optimizations that reduce the overall query execution time.

1 Introduction

Database management system benchmarks provide the industry with a quantifiable metric which allows for increased vendor competition, resulting in vastly faster databases and more resources allocated to solving future optimizations. These benchmarks were created by a non-profit group called the Transaction Processing Performance Council (also known as TPC) which was founded in 1988 to define transaction processing and database benchmarks. Some of the biggest database manufacturers in the world, such as Microsoft, Oracle, and IBM, are among the founding members of the TPC [1]. The utility of decision support systems in supporting businesses with organizational decision-making data led to a dramatic increase in their popularity over the course of the 1990s. By the end of the decade, the TPC-H benchmark went on to replace TPC-D due to new features such as join indices, summary tables, and materialized views. TPC-H is an ad-hoc decision support benchmark that consists of a 3rd Normal Form schema which contains 8 tables, 6 of which grow linearly with the scale of the data provided. It uses 22 queries which are complex, and 2 data refreshes, insert and delete, which are run in parallel to test concurrency [9]. To this day, TPC-H is considered a leader in both raw performance and price-performance [1].

In this paper, we quantify the performance of the 22 queries of the TPC-H benchmark on a database generated with a scale factor of 1. Based on the runtimes of the queries, we attempt to optimize slower queries by identifying performance bottlenecks, making changes to the OS settings and altering the structure of the queries. The goal of this paper is to provide the reader with a comprehensive look at the performance of the TPC-H benchmark and identify optimizations to improve it.

2 Methods

In our experiments, we used the database provided for class assignments, which had a scale factor of 1. We generated queries with default values for the substitution parameters using the QGEN tool [2] and query templates downloaded from the TPC website [8]. Queries were run serially in SQLite using a high level for loop. Results were captured for each query, including execution time and the query plan. The reason for running each query serially was to prevent caching of the data. If the same query were re-run, this would result in what's considered warm data as the page cache will have already held the past data query. It is important to use cold data (with no cache pre-populated) to arrive at consistent behavior. After running the original queries, we attempted to improve performance in two ways: modifying SQLite's configuration and changing the structures of the queries themselves.

3 Benchmark Analysis

3.1 Improving Queries by Modifying SQLite (OS) Settings

After running the original queries, we made changes to SQLite's settings to improve performance. The settings we changed along with descriptions of them are detailed below in the Appendix [**Modified SQLite settings**]:

The OS settings configured through SQLite's configuration improved performance by enabling page cache as well as journaling improvements. Along with OS settings, additional configurations were enabled to improve query plan debugging and detailed database/OS interactions.

The following table compares the average query runtime without modifications to the query runtime after OS settings were modified. Queries labeled "timed out" took longer than 15 minutes to complete. We can see that the queries 7, 9, and 21 had the most significant improvements from OS changes highlighted in **green**.

QUERY #	Time (no changes)	Avg (OS changes)
1	7.679	6.035
2	0.308	.300
3	1.952	1.353
4	0.476	0.518
5	1.125	0.872
6	1.244	1.279
7	7.127	4.648
8	2.175	1.803
9	15.783	8.933
10	0.988	0.955
11	0.684	0.663
12	2.048	2.155
13	9.155	9.234
14	1.278	1.334
15	0.452	0.000
16	0.980	0.456
17 (timed out)	0	0
18	1.247	1.311
19	1.70	1.804
20 (timed out)	0	0
21	0	4.529
22 (timed out)	0	0

Using Query 9 as an example, we can further analyze why OS changes provided an improvement over default SQL settings. Shown below is the query plan applied to Query 9 for both the default settings and the OS modifications.

```
.read q9.sql
QUERY PLAN
|--SCAN TABLE lineitem
```

```
|--SEARCH TABLE part USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE supplier USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE partsupp USING INDEX sqlite_autoindex_PARTSUPP_1 (PS_PARTKEY=? AND
PS_SUPPKEY=?)
|--SEARCH TABLE orders USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE nation USING INTEGER PRIMARY KEY (rowid=?)
|--USE TEMP B-TREE FOR GROUP BY
`--USE TEMP B-TREE FOR ORDER BY
```

The table below summarizes the statistics for Query 9 before and after the OS changes.

	Default Settings	OS Modifications
Runtime	12 seconds	6 seconds
Page cache hits	7123594	**13539550**
Page cache misses	7194963	779004
Bytes received by read()	29470598515	3190838881
Bytes sent to write()	17663508	15263
Read() system calls	7195009	779075
Write() system calls	8854	392
Cancelled write bytes	9994240	**0**

The OS modifications reduced read pressure by approximately 50% based on page cache hits highlighted in green. In addition, the modifications reduced temporary write system calls and removed all cancelled writes to the system. The reduction in runtime from 12 to 6 seconds can be attributed to page cache hits reducing OS read/write calls.

For queries 7, 9 and 21, OS setting changes improved overall performance and total time to complete. Our slowest running queries (17,20, and 22) did not see significant improvement from these adjustments. In the next section we take a deeper dive into these queries in order to determine what improvements can be made.

3.2 Improving Performance by Modifying Query Structure

Queries 17, 20, and 22 did not see a performance improvement after modifying the OS settings. We first analyzed these queries to determine which elements most contributed to the slow runtime. Then, we made changes to the structure of the queries to improve performance.

For Queries 17 and 20, we improved performance by using an index. A database index has a similar function to an index in a book. A book index points a reader to all pages related to a specific topic. In the same way, a database index points the query to all the rows in the table that are related to the specified topic. Rather than scanning an entire table to find the relevant rows, the query can use the information gathered from the index to jump directly to the rows it needs.

Each index is associated with one specific table, so all columns of the index must come from that table. Multiple indexes can be created from the same table. Indexes created in SQLite use a B-tree (balanced tree) to hold index data. B-tree indexes work best with column comparisons in expressions that use =, >, >=, <, <=, or BETWEEN operators [10]. If the index only has only one column, that column will be used as a sort key. In a multicolumn index, data is sorted by each column sequentially, meaning that any duplicate values from column A will be sorted by column B and so on. As such, column order can be very important in index creation [7]. While an index will be automatically created for many queries, explicitly creating one can significantly increase performance [7].

Below is a sample from the index we created for Q20. We generated the index using the statement, "create index idx_ps_part_sup_key on partsupp(ps_partkey, ps_suppkey);". In the original table the data is not sorted, while in the index the data is sorted first by ps_partkey and subsequently by ps_suppkey, which is consistent with the order the columns were listed in the create statement.

PARTSUPP table

rowid	ps_availqty	ps_partkey	ps_suppkey	ps_supplycost
1	3325	1	2	771.64
2	8076	1	2502	993.49
3	8895	2	3	378.49
4	4969	2	2503	915.27

IDX_PS_PART_SUP_KEY Index Based on PARTSUPP Table

ps_partkey	ps_suppkey	rowid
1	2	1
1	2502	2
2	3	3
2	2503	4

Query 17
As seen below, the original query plan for Query 17 was a large read loop scanning the lineitem table.

```
 sqlite> .read q17.sql
QUERY PLAN
|--SCAN TABLE lineitem
|--SEARCH TABLE part USING INTEGER PRIMARY KEY (rowid=?)
`--CORRELATED SCALAR SUBQUERY
   `--SCAN TABLE lineitem
```

Operation codes (opcodes) are machine language instructions that specify which operation is to be performed. They can be useful in understanding the process of a program or query. The table below shows opcodes for Query 17. The first column indicates the number of op codes in order of the query. Based on this information we focus on the number of read scan loops occurring. The scan loops are indicated (as Rewind) in #4 and #16 of the first column. The SQLite opcode documentation refers to Rewind as an opcode to traverse rowed or column or next instruction of P(1-5) and leaves the cursor configured to move in forward order to the Next location.
(**Original Query 17 OpCodes)**

addr	opcode	p1	p2	p3	p4	p5	comment
0	Init	0	34	0		00	Start as 34
1	Null	0	1	0		00	r[1..2]=NULL
2	OpenRead	0	10	2	6	00	root=10 iDb=0; LINEITEM
3	OpenRead	1	4	0	7	00	root=4 iDb=0; PART
4	Rewind	0	30	0		00	
…							
16	Rewind	2	23	0		00	
… (see full output in appendix)							

We created an index to increase the performance of this query, reducing our runtime from over 15 minutes to under 3 seconds (2.828 seconds for index creation + .133 seconds to run Query 17 = 2.961 seconds). Creating this index: 'create index idx_part_sup_key on lineitem(l_partkey, l_suppkey);' resulted in the change to the query plan shown below.

Original query plan:
```
QUERY PLAN
|--SCAN TABLE lineitem
```

```
|--SEARCH TABLE part USING INTEGER PRIMARY KEY (rowid=?)
`--CORRELATED SCALAR SUBQUERY
   `--SCAN TABLE lineitem
```

New query plan: (Note: search table uses an index versus a full scan of the table)

```
QUERY PLAN
|--SCAN TABLE part
|--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=?)
`--CORRELATED SCALAR SUBQUERY
   `--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=?)
```

Post-optimization Improvements can be seen in many areas including runtime, read() and write() system calls and cancelled write bytes.

	Before Optimization	After Optimization
Runtime	> 10 minutes	< 3 seconds
Page cache hits	13184	614713
Page cache misses	2368588	12951
Bytes received by read()	888940022	53085444
Bytes sent to write()	410124518	13684
Read() system calls	217099	13015
Write() system calls	200925	285
Cancelled write bytes	322695168	0

For full query statistics and reduced Rewind Opcode with the modified Q17 query plan, see appendix under [Q17 OpCodes with Index Modifications]

The new query plan searched via index O(1), while the original query searched via a full table scan as O(N) or B-tree O(log N) if a primary key exists.

Flame Graph for Query 17 Analysis

Flame graphs provide a visualization of hierarchical data, created to visualize stack traces of profiled software so that the most frequent code-paths can be identified quickly and accurately [3].

As with any visual tool, it's important to point out how to read and identify performance bottlenecks. For flame graphs, the x-axis calculates the runtime for each function call request, so wider spans of time indicate longer runtime. The y-axis calculates the depth of the call stack (Ie number of systems call tree). Embedded into an SVG (Scalable Vector Graphics) file, a user can navigate over a call stack and click to expand various levels for further granularity of analysis. In addition, the flame graphs colors represent a percent of the total calling stack. The colors progress from lighter shades of yellow to dark followed by lighter to darker shades of red. For our analysis, we will focus on the dark red call stack related to SQL read requests.

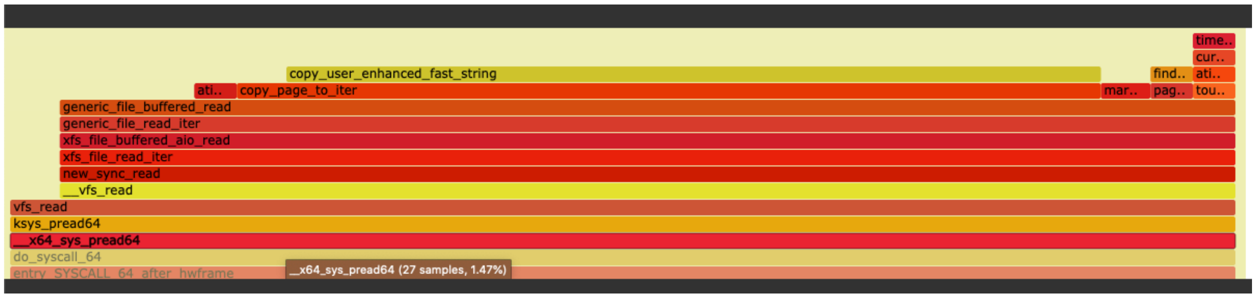
In this section we will describe how to capture flame graphs followed by comparing the original Q17 SQL query alongside an improved query with an index.

Linux command run to collect and analyze Query 17:

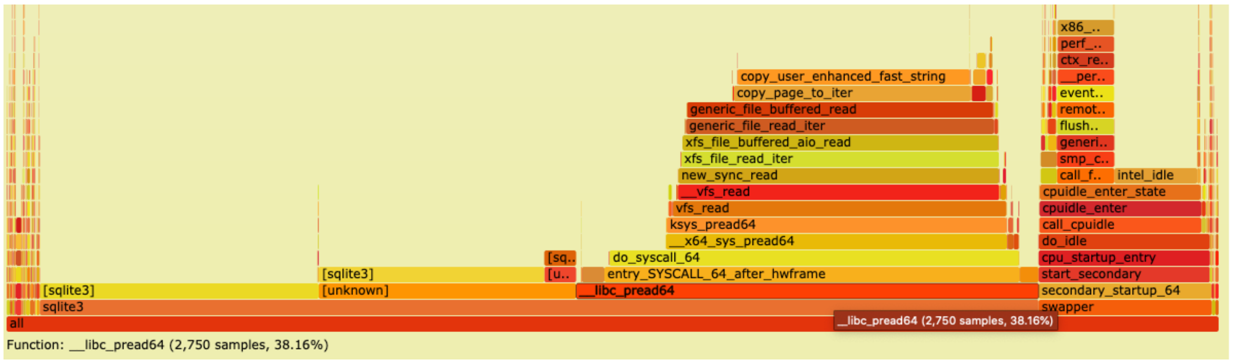
```
# Records the query via perf collecting system calls for 60 seconds
perf record -F 99 -a -g -- sleep 60
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.711 MB perf.data (1839 samples) ]

# Next fold the stats into a folded representation.
perf script | stackcollapse-perf.pl > out.perf-folded

# Last to convert into a visual hierarchal stack using flamegraph.pl.
flamegraph.pl out.perf-folded >perf-tpch-q17-idx.svg
```

Below we compare the Query 17 SQL flame graphs capturing index performance improvements followed by the original query.

Modified Q17: **create index idx_part_sup_key on lineitem(l_partkey, l_suppkey);**



The new stack clearly shows pread only required a small set of reads (1.47% total processing of sqlite3) which the index provided versus a full scan from the original query without an index below: (pread consumed 38% of sqlite3 sampling)



Query 20

Query 20 was optimized using the same index technique as Query 17. The query completed in under 3 seconds. See below for query plan differences before and after optimization.

Original query plan:

```
QUERY PLAN
|--SEARCH TABLE supplier USING INTEGER PRIMARY KEY (rowid=?)
|--LIST SUBQUERY
|  |--SEARCH TABLE partsupp USING INDEX sqlite_autoindex_PARTSUPP_1 (PS_PARTKEY=?)
|  |--LIST SUBQUERY
|  |  `--SCAN TABLE part
|  `--CORRELATED SCALAR SUBQUERY
|     `--SCAN TABLE lineitem
|--SEARCH TABLE nation USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR ORDER BY
```

New query plan:

```
QUERY PLAN
|--SEARCH TABLE supplier USING INTEGER PRIMARY KEY (rowid=?)
|--LIST SUBQUERY
|  |--SEARCH TABLE partsupp USING INDEX sqlite_autoindex_PARTSUPP_1 (PS_PARTKEY=?)
|  |--LIST SUBQUERY
|  |  `--SCAN TABLE part
|  `--CORRELATED SCALAR SUBQUERY
|     `--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=? AND
L_SUPPKEY=?)
|--SEARCH TABLE nation USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR ORDER BY
```

By creating an index idx_part_sup_key, both Query 17 and Query 20 benefit. This directly correlates to O(1) search in the new query plan versus the original O(N) via scan entire table in a loop.

Query 22

In optimizing Query 22, we tried a different approach than we used for 17 and 20. By observing the change in runtime when removing various parts of Query 22, we determined that the NOT EXISTS clause was the bottleneck. After replacing NOT EXISTS with NOT IN, we found that the runtime of the query was approximately 1.8 seconds, a vast improvement over the previous time of 15+ minutes.

However, NOT EXISTS and NOT IN will not behave the same when NULL values are present in the column that is being searched. If there are NULLs present in this column, NOT IN will return nothing. Conversely, NOT EXISTS will simply ignore NULLs and still return whether the value exists in the column [5]. The original and optimized queries returned the same results in our test database, which had no NULL values in the columns used in NOT in clause. However, had NULL values been present, the optimized query would have returned something different. Therefore, even though replace NOT EXISTS with NOT IN led to a major performance improvement, the optimized query will not work correctly in all cases so it should not be used.

4 Additional Considerations

Most of the analysis for this research focused on in-memory optimizations versus I/O to the back-end storage. This section also explores additional considerations if the database was compiled in terabytes TB versus GB. By creating a database far larger than its in-memory capabilities, read/write pressure becomes an additional bottleneck and choke point for any database query optimization. To fix these issues, here are some additional optimizations to consider:

- Linux OS settings:
    - Virtual VM's to use blk-mq (multi-threaded) versus virtio(single threaded)
    - IO scheduler to use 'none' or mq-deadline versus kyber or bfq as its cooperative and less performant.
    - Remove CPU power saving mode to improve performance at max power settings.
    - Direct IO versus async or synchronous settings.
- HW config:
    - Underpinning of HW to use NVMe drives capabable of 500K IOPS @4k block size. On a 1-U server we can deliver north of 1 million IOPS @4k block size across a small number of devices with low latency.
    - Using NVMe or SSD technology also increase queue depth of IO to and from a device versus HDDs.

5 Conclusion

Through our research we were able to successfully run and test all 22 queries required by the industry standard TPC-H benchmark. Our focus was to identify the slowest running queries, attempt performance optimizations, identify the cause of performance bottle necks, and categorize the queries based on their execution plan. In our findings we were able to determine that queries 17,20,21, and 22 were the worst performing when measured by run time. While some other queries, including 21, saw performance gains by making OS level optimizations to allow for more processing cores, larger cache and page sizes, the remaining three were unaffected.
The reason for their lack of improvement is due to their overall query plan. In query 17 we found that nested Rewind calls resulting in a quadratic growth factor, causing significant slowdown. By creating a separate index and removing the need for the second Rewind, we saw dramatic improvement. This same technique was applied to query 20 and saw similar results. Query 22 required a different approach that included removing NOT EXISTS and replacing it with NOT IN which also resulted in a significant improvement. Though we do acknowledge that if NULL values were to be present in our columns, our chosen method of optimization may not have performed the same.
In this paper we have measured the performance of all 22 queries, performed both OS level and query plan optimizations that resulted in improvements, and categorized the queries based on our findings. We have shown that the TPC-H benchmark, while starting to show its age, still is a high bar for benchmarking and for future benchmark designers. The set of bottle necks we've identified show the potential improvements for future benchmarks to build on which will lead to a stronger industry in the years to come.

References

[1] "10 Questions: The TPC-H Benchmark." *Exasol*, 25 Aug. 2020, https://www.exasol.com/resource/10-questions-about-the-tpc-h-benchmark/

[2] Electrum. "TPC-H Dbgen". *Github,* 2011, github.com/electrum/tpch-dbgen.

[3] Gregg, Brendad. "Flame Graphs." *Flame Graphs,* www.brendangregg.com/flamegraphs.html.

[4] "Pragma Statements." *SQLite*, www.sqlite.org/pragma.html.

[5] Shaw, Gail. "Not Exists vs Not In." *SQL Server Central*, 5. Dec 2011, https://www.sqlservercentral.com/blogs/not-exists-vs-not-in

[6] "The SQLite Bytecode Engine". *SQLite,* https://www.sqlite.org/opcode.html

[7] "SQLite Index: An Essential Guide to Sqlite Indexes." *SQLite Tutorial*, 11 Apr. 2020, https://www.sqlitetutorial.net/sqlite-index/

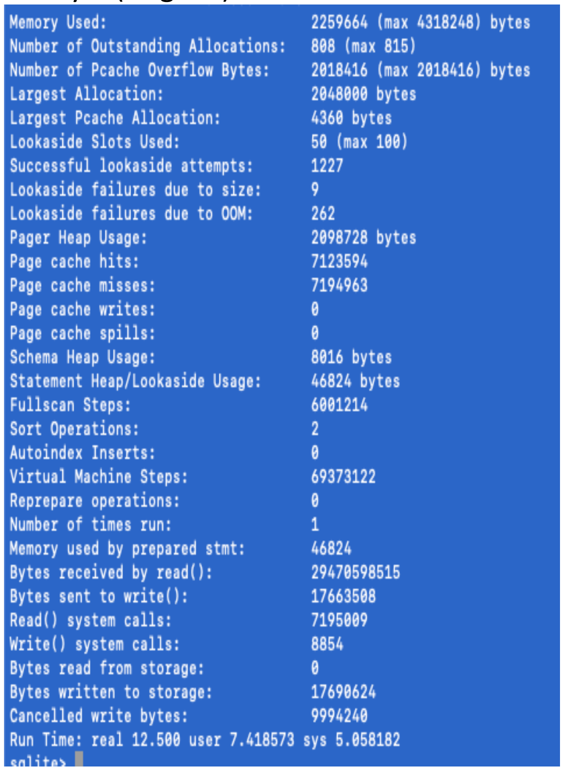
[8] TPC. "TPC-H Version 2 and Version 3." *TPC*, www.tpc.org/tpch/.

[9] "TPCH Benchmark". *Deister Software*, https://docs.deistercloud.com/content/Databases.30/TPCH%20Benchmark.90

[10] Winand, Markus. "The Balanced Search Tree (B-Tree) in SQL Databases." *Use The Index Luke: A Guide to SQL Performance for Developers*, https://use-the-index-luke.com/sql/anatomy/the-tree

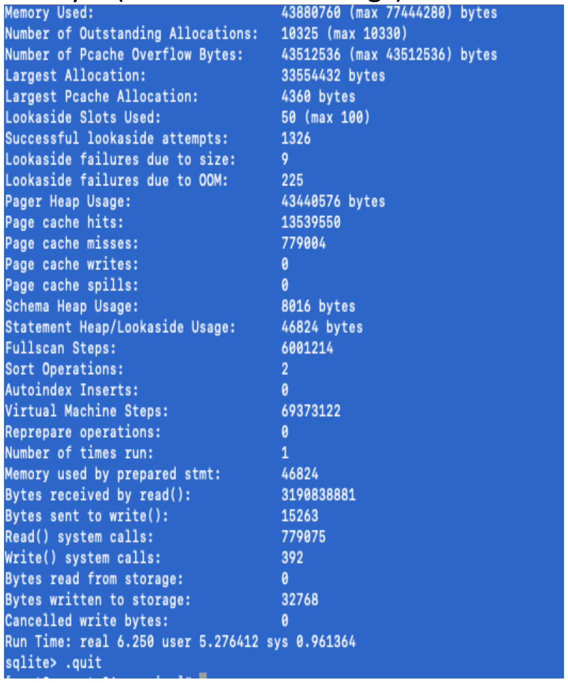
Figures and Tables

Query 9 (original)



```
Memory Used:                         2259664 (max 4318248) bytes
Number of Outstanding Allocations:   808 (max 815)
Number of Pcache Overflow Bytes:     2018416 (max 2018416) bytes
Largest Allocation:                  2048000 bytes
Largest Pcache Allocation:           4360 bytes
Lookaside Slots Used:                50 (max 100)
Successful lookaside attempts:       1227
Lookaside failures due to size:      9
Lookaside failures due to OOM:       262
Pager Heap Usage:                    2098728 bytes
Page cache hits:                     7123594
Page cache misses:                   7194963
Page cache writes:                   0
Page cache spills:                   0
Schema Heap Usage:                   8016 bytes
Statement Heap/Lookaside Usage:      46824 bytes
Fullscan Steps:                      6001214
Sort Operations:                     2
Autoindex Inserts:                   0
Virtual Machine Steps:               69373122
Reprepare operations:                0
Number of times run:                 1
Memory used by prepared stmt:        46824
Bytes received by read():            29470598515
Bytes sent to write():               17663508
Read() system calls:                 7195009
Write() system calls:                8854
Bytes read from storage:             0
Bytes written to storage:            17690624
Cancelled write bytes:               9994240
Run Time: real 12.500 user 7.418573 sys 5.058182
sqlite>
```

Query 9 (Modified OS settings)



```
Memory Used:                         43880760 (max 77444280) bytes
Number of Outstanding Allocations:   10325 (max 10330)
Number of Pcache Overflow Bytes:     43512536 (max 43512536) bytes
Largest Allocation:                  33554432 bytes
Largest Pcache Allocation:           4360 bytes
Lookaside Slots Used:                50 (max 100)
Successful lookaside attempts:       1326
Lookaside failures due to size:      9
Lookaside failures due to OOM:       225
Pager Heap Usage:                    43440576 bytes
Page cache hits:                     13539550
Page cache misses:                   779004
Page cache writes:                   0
Page cache spills:                   0
Schema Heap Usage:                   8016 bytes
Statement Heap/Lookaside Usage:      46824 bytes
Fullscan Steps:                      6001214
Sort Operations:                     2
Autoindex Inserts:                   0
Virtual Machine Steps:               69373122
Reprepare operations:                0
Number of times run:                 1
Memory used by prepared stmt:        46824
Bytes received by read():            3190838881
Bytes sent to write():               15263
Read() system calls:                 779075
Write() system calls:                392
Bytes read from storage:             0
Bytes written to storage:            32768
Cancelled write bytes:               0
Run Time: real 6.250 user 5.276412 sys 0.961364
sqlite> .quit
```

Appendix

Modified SQLite settings

```
cat ~/.sqliterc
.headers on
.mode columns
.timer on
.eqp full
.stats on

pragma threads=8;
pragma synchronous=1;
pragma journal_mode=wal;
pragma page_size=32768;
pragma cache_size=10000;
```

Configuration details [4]:

- threads – Number of threads to assist with queries. Increased from 0(default) to 8.
- Synchronous - synchronous=NORMAL(1) setting is a good choice for most applications running in WAL mode
- Journal Mode – WAL – Using a write ahead log.
- Page size – Query to hold the page size of the database in power of 2. Default is 1024 which is too small and inefficient for most systems. Setting to 32K (32768) was the most efficient setting. Bare minimum 4K to align with underlying SSD or NVMe media.
- Cache size - Query or change the suggested maximum number of database disk pages that SQLite will hold in memory at once per open database file

Sqlite commands provide additional tracing capabilities:

- Headers – Display table headers
- Mode – Mode set to columns to display table columns
- Timer – Trace system run time of each query
- Eqp – Set to “full” allowing both explain plan tracing and op codes [6]

Q17 original stats:

```
Memory Used:                         43878304 (max 289665568) bytes
Number of Outstanding Allocations:   10313 (max 10362)
Number of Pcache Overflow Bytes:     43516632 (max 43524824) bytes
Largest Allocation:                  40960000 bytes
Largest Pcache Allocation:           4360 bytes
Lookaside Slots Used:                16 (max 74)
Successful lookaside attempts:       782
Lookaside failures due to size:      10
Lookaside failures due to OOM:       0
Pager Heap Usage:                    43440576 bytes
Page cache hits:                     201957
Page cache misses:                   195199
Page cache writes:                   21819
Page cache spills:                   12820
Schema Heap Usage:                   8232 bytes
Statement Heap/Lookaside Usage:      2712 bytes
Fullscan Steps:                      0
Sort Operations:                     1
Autoindex Inserts:                   0
Virtual Machine Steps:               60012176
Reprepare operations:                0
Number of times run:                 1
Memory used by prepared stmt:        2712
Bytes received by read():            888940022
Bytes sent to write():               410124518
Read() system calls:                 217099
Write() system calls:                200925
Bytes read from storage:             0
Bytes written to storage:            502173696
Cancelled write bytes:               322695168
```

Q17 after optimizations:

```
Memory Used:                         43873720 (max 43873752) bytes
```

```
Number of Outstanding Allocations:   10297 (max 10298)
Number of Pcache Overflow Bytes:     43512536 (max 43512536) bytes
Largest Allocation:                  131072 bytes
Largest Pcache Allocation:           4360 bytes
Lookaside Slots Used:                42 (max 100)
Successful lookaside attempts:       7316
Lookaside failures due to size:      15
Lookaside failures due to OOM:       54
Pager Heap Usage:                    43440576 bytes
Page cache hits:                     614713
Page cache misses:                   12951
Page cache writes:                   0
Page cache spills:                   0
Schema Heap Usage:                   8232 bytes
Statement Heap/Lookaside Usage:      31168 bytes
Fullscan Steps:                      199999
Sort Operations:                     0
Autoindex Inserts:                   0
Virtual Machine Steps:               1672832
Reprepare operations:                0
Number of times run:                 1
Memory used by prepared stmt:        31168
Bytes received by read():            53085444
Bytes sent to write():               13684
Read() system calls:                 13015
Write() system calls:                285
Bytes read from storage:             0
Bytes written to storage:            32768
Cancelled write bytes:               0
Run Time: real 0.129 user 0.086942 sys 0.042110
```

Q17 OpCodes

```
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     34    0                    00  Start at 34
1     Null           0     1     2                    00  r[1..2]=NULL
2     OpenRead       0     10    0     6              00  root=10 iDb=0; LINEITEM
3     OpenRead       1     4     0     7              00  root=4 iDb=0; PART
4     Rewind         0     30    0                    00
5       Column         0     1     3                    00
r[3]=LINEITEM.L_PARTKEY
6       SeekRowid      1     29    3                    00  intkey=r[3]
7       Column         1     3     4                    00  r[4]=PART.P_BRAND
8       Ne             5     29    4     (BINARY)       52  if r[4]!=r[5] goto 29
9       Column         1     6     4                    00  r[4]=PART.P_CONTAINER
10      Ne             6     29    4     (BINARY)       52  if r[4]!=r[6] goto 29
11      Column         0     4     4                    00
r[4]=LINEITEM.L_QUANTITY
12      Null           0     8     8                    00  r[8..8]=NULL; Init
subquery result
13      Integer        1     9     0                    00  r[9]=1; LIMIT counter
14      Null           0     10    11                   00  r[10..11]=NULL
15      OpenRead       2     10    0     5              00  root=10 iDb=0;
LINEITEM
16      Rewind         2     23    0                    00
17        Column         2     1     12                   00
r[12]=LINEITEM.L_PARTKEY
18        Rowid          1     13    0                    00  r[13]=rowid
19        Ne             13    22    12    (BINARY)       53  if r[12]!=r[13] goto
22
20        Column         2     4     13                   00
r[13]=LINEITEM.L_QUANTITY
21        AggStep        0     13    10    avg(1)         01  accum=r[10]
step(r[13])
22      Next           2     17    0                    01
23      AggFinal       10    1     0     avg(1)         00  accum=r[10] N=1
24      Multiply       10    14    8                    00  r[8]=r[10]*r[14]
25      DecrJumpZero   9     26    0                    00  if (--r[9])==0 goto 26
26      Ge             8     29    4     (BINARY)       54  if r[4]>=r[8] goto 29
```

```
27      Column         0     5     4                    00
r[4]=LINEITEM.L_EXTENDEDPRICE
28      AggStep        0     4     1     sum(1)         01  accum=r[1] step(r[4])
29    Next           0     5     0                    01
30    AggFinal       1     1     0     sum(1)         00  accum=r[1] N=1
31    Divide         16    1     15                   00  r[15]=r[1]/r[16]
32    ResultRow      15    1     0                    00  output=r[15]
33    Halt           0     0     0                    00
34    Transaction    0     0     8     0              01  usesStmtJournal=0
35    String8        0     5     0     Brand#52       00  r[5]='Brand#52'
36    String8        0     6     0     JUMBO CAN      00  r[6]='JUMBO CAN'
37    Real           0     14    0     0.2            00  r[14]=0.2
38    Real           0     16    0     7              00  r[16]=7
39    Goto           0     1     0                    00
```

See below for the full output details: (**Q17 OpCodes with Index Modifications**)

```
sqlite> .read q17new.sql
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     34    0                    00  Start at 34
1     Noop           0     33    0                    00
2     CreateBtree    0     1     2                    00  r[1]=root iDb=0 flags=2
3     OpenWrite      0     1     0     5              00  root=1 iDb=0;
sqlite_master
4     NewRowid       0     2     0                    00  r[2]=rowid
5     String8        0     3     0     index          00  r[3]='index'
6     String8        0     4     0     idx_part_sup_key  00
r[4]='idx_part_sup_key'
7     String8        0     5     0     LINEITEM       00  r[5]='LINEITEM'
8     Copy           1     6     0                    00  r[6]=r[1]
9     String8        0     7     0     CREATE INDEX idx_part_sup_key on
lineitem(l_partkey, l_suppkey)  00  r[7]='CREATE INDEX idx_part_sup_key on
lineitem(l_partkey, l_suppkey)'
10    MakeRecord     3     5     8     BBBDB          00  r[8]=mkrec(r[3..7])
11    Insert         0     8     2                    18  intkey=r[2] data=r[8]
12    SorterOpen     3     0     2     k(3,,,)        00
13    OpenRead       1     10    0     16             00  root=10 iDb=0; LINEITEM
14    Rewind         1     21    0                    00
15      Column         1     1     10                   00
r[10]=LINEITEM.L_PARTKEY
16      Column         1     2     11                   00
r[11]=LINEITEM.L_SUPPKEY
17      Rowid          1     12    0                    00  r[12]=rowid
18      MakeRecord     10    3     9                    00  r[9]=mkrec(r[10..12])
19      SorterInsert   3     9     0                    00  key=r[9]
20    Next           1     15    0                    00
21    OpenWrite      2     1     0     k(3,,,)        11  root=1 iDb=0
22    SorterSort     3     27    0                    00
23      SorterData     3     9     2                    00  r[9]=data
24      SeekEnd        2     0     0                    00
25      IdxInsert      2     9     0                    10  key=r[9]
26    SorterNext     3     23    0                    00
27    Close          1     0     0                    00
28    Close          2     0     0                    00
29    Close          3     0     0                    00
30    SetCookie      0     1     9                    00
31    ParseSchema    0     0     0     name='idx_part_sup_key' AND type='index'
00
32    Expire         0     1     0                    00
33    Halt           0     0     0                    00
34    Transaction    0     1     8     0              01  usesStmtJournal=0
35    Goto           0     1     0                    00
```
Run Time: real 2.828 user 5.547345 sys 0.843189

```
QUERY PLAN
|--SCAN TABLE part
|--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=?)
`--CORRELATED SCALAR SUBQUERY
   `--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=?)
```

```
--- The above query completed the index in 2.8 seconds. Now to Query 17 using the
index.
```

New with the improved Index search

```
sqlite> .read q17.sql
QUERY PLAN
|--SCAN TABLE part
|--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=?)
`--CORRELATED SCALAR SUBQUERY
   `--SEARCH TABLE lineitem USING INDEX idx_part_sup_key (L_PARTKEY=?)
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     39    0                    00  Start at 39
1     Null           0     1     2                    00  r[1..2]=NULL
2     OpenRead       1     4     0     7              00  root=4 iDb=0; PART
3     OpenRead       0     10    0     6              00  root=10 iDb=0; LINEITEM
4     OpenRead       3     305876  0     k(3,,,)        02  root=305876 iDb=0;
idx_part_sup_key
5     Rewind         1     35    0                    00
6       Column         1     3     3                    00  r[3]=PART.P_BRAND
7       Ne             4     34    3     (BINARY)       52  if r[3]!=r[4] goto 34
8       Column         1     6     3                    00  r[3]=PART.P_CONTAINER
9       Ne             5     34    3     (BINARY)       52  if r[3]!=r[5] goto 34
10      Rowid          1     6     0                    00  r[6]=rowid
11      SeekGE         3     34    6     1              00  key=r[6]
12        IdxGT          3     34    6     1              00  key=r[6]
13        DeferredSeek   3     0     0                    00  Move 0 to 3.rowid if
needed
14        Column         0     4     3                    00
r[3]=LINEITEM.L_QUANTITY
15        Null           0     8     8                    00  r[8..8]=NULL; Init
subquery result
16        Integer        1     9     0                    00  r[9]=1; LIMIT
counter
17        Null           0     10    11                   00  r[10..11]=NULL
18        OpenRead       2     10    0     5              00  root=10 iDb=0;
LINEITEM
19        OpenRead       4     305876  0     k(3,,,)        02  root=305876 iDb=0;
idx_part_sup_key
20        Rowid          1     12    0                    00  r[12]=rowid
21        SeekGE         4     27    12    1              00  key=r[12]
22          IdxGT          4     27    12    1              00  key=r[12]
23          DeferredSeek   4     0     2                    00  Move 2 to 4.rowid
if needed
24          Column         2     4     13                   00
r[13]=LINEITEM.L_QUANTITY
25          AggStep        0     13    10    avg(1)         01  accum=r[10]
step(r[13])
26        Next           4     22    0                    00
27        AggFinal       10    1     0     avg(1)         00  accum=r[10] N=1
28        Multiply       10    14    8                    00  r[8]=r[10]*r[14]
29        DecrJumpZero   9     30    0                    00  if (--r[9])==0 goto
30
30        Ge             8     33    3     (BINARY)       54  if r[3]>=r[8] goto
33
31        Column         0     5     3                    00
r[3]=LINEITEM.L_EXTENDEDPRICE
32        AggStep        0     3     1     sum(1)         01  accum=r[1]
step(r[3])
33      Next           3     12    0                    00
34    Next           1     6     0                    01
35    AggFinal       1     1     0     sum(1)         00  accum=r[1] N=1
36    Divide         16    1     15                   00  r[15]=r[1]/r[16]
37    ResultRow      15    1     0                    00  output=r[15]
38    Halt           0     0     0                    00
39    Transaction    0     0     9     0              01  usesStmtJournal=0
40    String8        0     4     0     Brand#52       00  r[4]='Brand#52'
41    String8        0     5     0     JUMBO CAN      00  r[5]='JUMBO CAN'
42    Real           0     14    0     0.2            00  r[14]=0.2
43    Real           0     16    0     7              00  r[16]=7
```

```
44    Goto           0     1     0                    00
3440
```