Lightweight Automatic Index Selection for Relational Databases

Sarah Chasins
schasi1@cs.swarthmore.edu

Phil Koonce
pkoonc1@swarthmore.edu

Abstract

Maintaining a workload-appropriate index set is crucial to achieving good performance in large databases, but automatic index selection systems are still much less effective at identifying good indexes than a skilled human database administrator. The number of possible indexes grows exponentially relative to the number of relevant columns, so exhaustive search of the index space is intractable, and research to date has not produced a sufficiently reliable method for automatic pruning. This is especially true of lightweight methods. In this paper, we conduct thorough testing of COLT, one automatic index tuning system with lower overhead than most. We reveal that its reliance on a DBMS's native query optimizer will cause it to perform poorly in circumstances where the query optimizer mis-predicts costs. We then present our own automatic index selection tool, SCISSOR, which reduces workload execution time significantly more than COLT-suggested indexes in our testing environment.

1 Introduction

Current Database Management Systems (DBMSs) perform well without human managers, contingent on having little data and receiving few queries. Unfortunately, in most modern databases, scalability at the level required by many of today's data-intensive applications is only made possible by a skilled database administrator's manual tuning of the index set. The database administrator is responsible for selecting new indexes to materialize and old indexes to drop, based on knowledge of the workload, the database's table structure, and an understanding of a database's query optimization approach. Despite the fact that all of this information is available to the DBMS itself, no open source DBMSs include automatic index tuning tools.

Modern DBMSs have complex query execution planners that automatically use indexes to drastically reduce execution costs. When a query is sent to a database server, the query is parsed to determine what data the client demands. The query optimizer identifies the tables that must be joined, the columns that must be extracted, and the conditions that must be met for a record to be placed in the result set. In standard relational databases, correctness alone does not impose an order of access for the tables involved in the query. Thus, the number of possible access paths is exponential. Given that some of these paths are orders of magnitude faster than others — in large part because of indexes — the query optimizer's task is to select the path that will answer the current query most quickly. Making good indexes available to the optimizer can drastically reduce execution cost of queries. We believe it should be possible for an automated system to identify such indexes.

In this paper, we present a performance analysis of a previous system's proposed index recommendation tool, as well as our own implementation of an index manager. Specifically, we evaluate

the performance of the COLT (Continuous On-Line Tuning) system's index recommender, developed at University of California Santa Cruz (Schnaitter et al., 2007). We design an experiment that should showcase the best possible performance of their index recommender. Our findings demonstrate that COLT's performance is reliant on the good performance of the query optimizer of the DBMS in which COLT is implemented. We further demonstrate that good performance by that optimizer is not guaranteed.

Our own original system, SCISSOR (Single Column Index SuggeStOR), presents an alternative extremely lightweight approach to index recommendation. We describe its design and evaluate it with the same tests we use to investigate our reimplementation of COLT. We find that SCISSOR indexes decreases execution time by an order of magnitude more than COLT indexes.

We first give an in-depth overview of the COLT system as presented by Schnaitter et. al. (Schnaitter et al., 2007), and brief overviews some automatic index selection tools for proprietary DBMSs (Chaudhuri and Narasayya, 1997). We then present in detail our implementation of an index tuner that uses COLT's index recommender, as well as the details of our own index recommendation system. Lastly, we present and discuss the results of testing the two systems.

2 Related Work

A significant body of work addresses the challenge of index selection. Here we briefly outline two commercial systems, and then provide an in-depth description of COLT's methods.

2.1 Commercial Systems

Several tools for index selection have been developed and deployed in commercial products. We present a high level description of two successful index selectors with very different approaches. We then describe the index selection tool that we will evaluate.

Chaudhuri and Narasayya lay out three central techniques for reducing the size of the index selection problem and avoiding the task of comparing all possible indexes. First, they prune a large number of indexes using query syntax and cost information. Second, they develop a method for efficiently evaluating the likely benefits of an index set. Third, they build up potentially useful multi-column indexes using a novel iterative approach (Chaudhuri and Narasayya, 1997).

With our focus on improving lightweight index recommendation, we are mainly interested in the first of these techniques, the pruning of candidate indexes. Their candidate index selection model begins by considering one-column indexes in the first round, then two-column indexes in the second round, and so on. We emulate the first stage of this process in our query parser, discussed in Section 5.2. The authors refer to the module's algorithm as the query-specific-best-configuration algorithm, because it forms the candidate index set from the set of indexes that are part of any individual query's best possible access path. The problem of generating indexes for a whole workload is thus reduced to determining what indexes could produce the best performance for a given query. Their approach considers all indexable columns of the query to be potential candidate indexes, which is exactly the approach we use. The authors' algorithm for enumerating possibilities is semi-greedy, with a parameter m that should reflect the optimal number of indexes to compose. Unfortunately, the correct level of m is typically unknown. Depending on this setting, their algorithm can range from full greed (if m is 0), to complete naive enumeration (if m is the total number of indexes).

Another approach involves classification of columns to assess their likely relevance. Valentin et al., in (Valentin et al., 2000) use the following categories:

EQ columns that appear in EQUAL predicates

O columns that appear in the INTERESTING ORDERS list. This includes columns from ORDER BY and GROUP BY clauses, or join predicates.

RANGE columns that appear in range predicates

SARG columns that appear in any predicates but nested sub-queries or those involving a large object (LOB).

REF Remaining columns referenced in the SQL statement.

Based on knowledge of appropriate combinations of these classifications, their algorithm proceeds by

producing combinations of the following forms:
1. EQ + O
2. EQ + O + RANGE
3. EQ + O + RANGE + SARG
4. EQ + O + RANGE + REF
5. O + EQ
6. O + EQ + RANGE
7. O + EQ + RANGE + SARG
8. O + EQ + RANGE + REF

They add an additional brute-force stage, which simply enumerates all possible indexes until it reaches a maximum number of permitted indexes. Any duplicated indexes are removed from consideration, and then the entire set is added to the schema as hypothetical indexes. The normal optimizer is run with these virtual indexes, and any virtual index which is included in the optimal access plan is returned as a recommended index.

Although these complex systems have been relatively successful, the overhead is significant, and some tuning remains necessary. The level of greed (in the first system) and the number of indexes that the brute force stage may generate (in the second system) do not have obviously correct settings. Further, erring on the side of generating too many indexes adds significantly to the time and computation costs of selecting indexes with these tools. We attempt a more lightweight approach, though emulating the classification method of Valentin et al. could give better candidate index ranking, as discussed in Section 6.2

2.2 COLT: Continuous On-line Tuning

COLT is an addition to a pre-existing DBMS that recommends indexes based on the queries it receives and selects which indexes should be materialized at any given time, based on estimations of their benefits and costs. The goal of the system is to perform as well or nearly as well as an offline tuning system on a stable workload, while also reacting dynamically to shifting workloads.

The COLT system approaches self-tuning by adding two components to a standard database management system: an extension of the typical query optimizer, and what the authors call a Self-Tuning Module. The Extended Query Optimizer (or EQO) chooses the best physical access path, as would a normal query optimizer, but allows *profiling* of indexes. That is, it returns information about how altering the materialization status of the profiled index would affect the execution time of the given query. If the index i is currently materialized in the database, the EQO estimates the amount of time that would be added to the run time of the query being optimized if i were eliminated. If the index i is only hypothetical, the EQO estimates the amount by which the run time would decrease if i were materialized. As queries pass through the COLT system, it continuously calculates index gains. It chooses the optimal plan for the current set of indexes, after which it can find the optimal plan excluding index i, or conditional on i being available.

COLT's Self-Tuning Module (or STM) is tasked with adjusting the set of materialized indexes. The STM is the component selecting the indexes which the query optimizer should profile. The STM uses the results of the EQO profiling to determine the appropriate set of indexes for the current workload, adding and removing indexes as necessary. The challenging components of its role are the choice of indexes to profile and preserving the set of indexes such that they do not surpass a space limit. To accomplish this, the STM creates batches of ten queries each, called *epochs*. During the normal processing of each epoch, the STM also collects the results of profiling candidate indexes on this ten query workload. Based on the results of this profiling, the set of materialized indexes may be adjusted at the end of the ten query period.

Because making access plans for hypothetical index sets is computationally expensive — as is access path selection in general — COLT limits the number of profiling requests permissible within a given epoch. Since the system reuses intermediate results for efficiency, their system is only permitted to gather profiling results from queries which are currently being executed. Given these restrictions, it is essential for the system to be capable of good approximations of index value, without profiling. COLT classifies particular queries as similar to each other, based on whether they use the same relations and have similarly selective predicates. They use this measure of similarity to draw conclusions about the benefit of a given index to a query for which it has not been tested. If an index i's gain has consistently been observed for queries similar to q,

the system has greater confidence that the gain will apply for to q as well, and the system is less likely to profile i on q. Instead, the system will test indexes for which a consistent pattern has not been found in the past, since it has not yet established a confident evaluation of that index's value.

At the end of a ten query epoch, the STM makes all decisions that alter the status of a given index. During operation, the STM identifies candidate indexes relevant to new queries' predicates. These are evaluated with a crude metric, until the crude metric provides enough evidence of their benefits that they may be evaluated using index profiling. This decision to start profiling, the decision to materialize an index, and the decision to remove a materialized image are all made at the end of the epoch. The STM may also remove an index from the set of indexes being profiled and considered for promotion into the database.

The COLT system in its current form has several flaws, some of which the creators mention in their papers (Schnaitter et al., 2006) (Schnaitter et al., 2007). First, as they note, they ignore update queries, assuming a read-only workload. While non-ideal, we do not consider this a fatal flaw, given that this may be an appropriate form of optimization for many applications. The critical flaw that they do not raise in either (Schnaitter et al., 2006) or (Schnaitter et al., 2007) is the failure to provide any recommendations for queries with `ORDER BY` or `GROUP BY` clauses. The authors note in a short feature guide associated with the source code that "`RECOMMEND INDEXES` is limited in its ability to handle complex queries. It should make good recommendations for queries with simple equijoins, range predicates, and projections, but may have unpredictable behavior given other syntax, such as subqueries and set operations" (Schnaitter, 2011). We believe that this weakness prevents COLT from aiding in most useful read-mostly applications and significantly limits its relevance.

The authors execute an experimental study which produces encouraging results. They implement an offline tuning system which is given full knowledge of the workload, examines all possible one-column indexes, and materializes the optimal set, subject to a space constraint. This offline tuner is compared to COLT on a 1.4 GB synthetic data set based on the TPC-H schema (see Section 4.2). They state that 18 indexes could yield improvement, and the tests specify a space limit allowing 3-6 of them to be materialized at any given point. They find that their system barely underperforms the offline system on a stable workload, and barely outperforms the offline system on a shifting workload. We believe that the restrictions placed on these tests are too strong, and misrepresent typical demands on databases. A limit small enough to allow only 3-6 indexes on a TPC-H database is tiny compared to the size of the database overall. In our own tests, materializing 8 COLT indexes requires only on the order of 176 MB, for a 1 GB database. Given that it is not uncommon for database users to dedicate more space to indexes than to the database itself, this suggests their test may not represent the common case.

We wondered whether the experiment they constructed — running COLT next to their offline tuning method — might emphasize the wrong comparison. The authors do not reveal how well COLT and the offline tuning system perform compared to an unoptimized database. Their conclusions about COLT's benefits are contingent on the preconception that the offline tuner performs well, but they present no evidence that their offline tuner significantly improves query execution time. In particular, we were concerned that the very small space limit they impose might result in the offline optimizer producing no reduction in execution time, for some queries in their experiments. This brought about our interest in reanalyzing COLT. One of our goals is to produce the missing experiments.

3 Index Selection Implementation

3.1 COLT Implementation

We create an index selection tool that uses COLT's extended query optimizer for index recommendation, and to evaluate the benefits of a given index. Our own implementation of a COLT-style index selection system is built on these two features of their original implementation.

Our version of COLT (henceforth COLT$--$) makes several modifications to the original model. First, we do not implement online tuning, but rather elect to test their index recommendation solution more directly. COLT$--$ is given a set of queries,

selects indexes to optimize for that workload, and is tested on the same workload. The method of optimization is the same method that standard COLT implements at the end of each epoch. Based on the current estimated value of all indexes, the system picks the best set of indexes that it can fit in the memory available to it. Under this formulation, the quality of the index recommender should determine performance.

The system proceeds as follows. First, COLT−− runs COLT's index recommender for each query in the workload. In the event that the system proposes a candidate index, COLT−− runs COLT's index profiling optimizer once for the current set of materialized indexes, and once including the candidate index as a hypothetical index. The COLT recommender estimates the query's time cost for both cases, and COLT−− treats the difference as the candidate index's expected value. COLT's index recommender uses statistics about the database to estimate the amount of space a given index will demand. For each candidate index, this space requirement is associated with the index as a cost. With all queries and candidate indexes examined, COLT−− is faced with the challenge of implementing the set of indexes with the highest possible value, subject to a space constraint. This will be easily recognized as an instance of the Knapsack Problem. The sum of the chosen indexes' predicted benefits is the value to maximize. The limited space available for materialized indexes is the restriction: the knapsack size. We use a standard dynamic programming solution to the knapsack problem, and return the set of indexes with the highest overall value that does not surpass the space limit. These indexes are then materialized.

An overview appears below:

```
FOR each query:
  Run COLT's RECOMMEND INDEXES to
    find candidates
  FOR each candidate:
    Run COLT's EXPLAIN INDEXES to
      find performance change and
      creation cost
  indexes to implement =
    solveKnapsack(candidates,
                  spaceLimit)
  materialize(indexes to implement)
```

3.2 SCISSOR: Single Column Index SuggeStOR

We also implemented a novel method of candidate index suggestion, inspired by the work of Chaudhuri et al. on automatic index selection for Microsoft SQL server (Chaudhuri and Narasayya, 1997). Our system, SCISSOR (Single Column Index SuggeStOR) generates and ranks single column indexes for the tables that appear in a workload. First, the system obtains the full database schema, including all tables and the columns in each table. When SCISSOR is given a query, it parses the SQL to determine which columns in which tables are examined in each query. Columns that appear in `WHERE` clauses and `ORDER BY` or `GROUP BY` clauses are all extracted and matched to the tables in the `FROM` clause. The candidates for all the queries are aggregated as counts of `<table, column>` pairs. These pairs are sorted by number of occurrences. The candidate indexes are then ranked by the number of queries they could potentially benefit. This analysis does not take into account index size, or related external information. While larger indexes are more expensive to create, the benefits are similarly more significant than the benefits of a small, inexpensive index. Thus, it seems potentially beneficial to consider indexes independent of creation cost.

The system can then issue `CREATE INDEX` commands to the database for a specified percentage of these indexes, for indexes that benefit a specified minimum number of queries, or until a specified time period has elapsed. These are different constraints than the ones used in (Schnaitter et al., 2007). The COLT system used a space limitation to determine the number of indexes to implement. In our system, and likely on most systems, there is ample disk space for indexes and materialization time may be the limiting cost. In order to be useful for this common case, SCISSOR can use information about workload size to materialize a certain number of indexes, or materialize indexes for a certain period of time.

4 Test Methodology

We briefly describe the physical systems on which we test our index selection methods, the schema of

the databases we use, and specifics of the databases themselves.

4.1 Physical Systems

We tested on 2 nearly identical machines with 4 cores at 2.80 GHz, 16 GB of memory, and at least 500 GB of hard drive space on a 7200 RPM hard drive. We ran the COLT creators' modified PostgreSQL server. We ran tests on multiple machines because of time constraints: we could not produce reliable results by simultaneously running multiple tests on a single computer, because each test would affect the timing of any other test. In addition to this, the modified server which implements COLT's `RECOMMEND INDEXES` and `EXPLAIN INDEXES` functionality frequently segfaults and restarts when asked to evaluate complex queries. To accommodate this, we needed a separate machine to run any test code that could segfault and restart the server.

4.2 TPC-H Schema

Our databases utilized synthetic data, generated in accordance with the TPC-H schema provided by the Transaction Processing Performance Council. It is a Decision Support System specification, intended to model applications that examine large amounts of data using complex queries. The queries used in our tests are generated by the query generation tool provided with the benchmark. The schema appears in Table 1. The `nation` table and the `region` tables are both of fixed size, with 25 rows and 5 rows, respectively. The other tables grow to fill the rest of the space in the database, with the `lineitem`, `orders`, and `partsupp` tables accounting for the vast majority of the records.

4.3 Test Data

We used 2 database sizes in our testing: 100 MB and 1 GB. The majority of the reported tests were run on the 1 GB database, with 100 MB tests to examine how time and space costs scaled up with more data. Running a 24 query TPC-H workload on the non-optimized 1 GB database took approximately 20 minutes, such that a single test comparing a non-optimized and an optimized method ran between 20 and 40 minutes. Larger databases tended to be too slow for us to run more than a few tests on them, given our time constraints: a 10 GB database took 2 days to populate with synthetic data. Creating larger databases using the our data generation tool and the `COPY` command was not possible without a larger main memory, nor was running queries on a larger database feasible within the scope of this project. A single query on a 20 GB database could easily run for upwards of a day.

Table Name	# Col.
`customer`	3
`lineitem`	16
`nation`	4
`orders`	9
`part`	9
`partsupp`	5
`region`	3
`supplier`	7

Table 1: Table names and column counts for the TPC-H schema.

5 Results and Discussion

5.1 COLT–

5.1.1 Online vs. Offline

It is important to note that our system does not directly implement COLT, but rather tests the value of the indexes it suggests. We do not implement online index tuning. However, if anything, our implementation favors COLT's optimization tools. Because the costs of optimization (extra query profiling calls, materialization time) are not included in the times we report, this reworking should perform better than theirs. Further, their system is constantly adjusting on the basis of the ten most recently seen queries. Our testing gives the system foreknowledge of all the queries it will answer, which should permit it to select its preferred indexes for that workload. Because we provide it a space limit that is far above the total space requirements of all indexes it suggests, it is allowed to pick any set of the indexes it considers, even all of them. In fact, in our experiments, the system implements every index which it estimates to have a positive value, for a total of 8 indexes with a space cost of 176 MB. Our environment provides COLT with the best possible chance at success, al-

lowing it to do every optimization it can generate.

A smaller space limit — like the restriction they apply in their tests — could only harm performance in this context. That is, it could only make performance worse relative to the database without indexes.

5.1.2 Execution Time

We test COLT−− on the synthetic data set described above, produced with the TPC-H schema, the same decision support benchmark used by the COLT authors. We contrast the performance of a COLT−− database with a non-optimized database, on a 1 GB data set. The non-optimized database has no indexes except the primary key indexes. The COLT– database has all COLT-recommended indexes materialized. To ensure a fair comparison, No Index and COLT trials were interleaved, with indexes materialized and dropped between workloads. We used a small workload of 24 queries, generated by the DBGEN tool provided with the TPC-H data. Each experiment was run 10 times

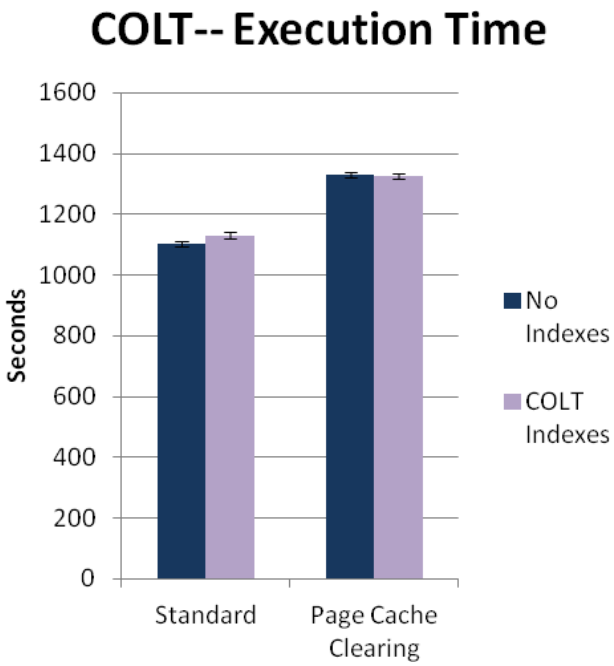

Figure 1: The average execution time in seconds for a 24 query workload, across 10 runs. Error bars reflect standard deviation.

As is evident from the Warm condition in Figure 1, the average execution time slightly increases with the introduction of COLT-recommended indexes. With a warm cache, the materialization of COLT-recommended indexes increases execution time by an average of 28.2 seconds, for a workload with a mean execution time of 1120.9 on a non-optimized database. Figure 2 gives us a closer look at the differences in run times for individual queries. Although many queries appear to benefit from COLT−− indexes, many remain unaltered. Most consequently, the run time for the final query, number 24, remains unaffected. The final query accounts for a majority of the workload's execution time. If we limit the queries under consideration to only the queries for which COLT can suggest an index, the workload is reduced to 9 queries, and COLT indexes increase run time by an average of 29.0 seconds. If we limit the queries under consideration to only the queries for which COLT implements an index, the workload is reduced to only 5 queries, and COLT indexes increase run time by an average of 30.0 seconds.

A series of `EXPLAIN` commands reveals that the data access plans do in fact change in the presence of COLT indexes, that they are being used to retrieve the data. Running `EXPLAIN` on the final, long-running query indicates that PostgreSQL's optimizer expects the introduction of the COLT−− indexes to reduce disk page fetches by 66%, from 353107911 to 117712520.

To reconcile these expected gains with the execution times reported above, we turn to the fact that index selection usually has its most significant effect on I/O time. Ideally, it affects overall execution time through this mechanism, but it is targeted specifically at reducing I/O time. To examine COLT indexes' effects on I/O time, we introduce a new set of tests, repeating the comparison of No Indexes to COLT, but ensuring a cold page cache prior to the execution of each query. Because Warm Cache No Indexes and Cold Cache No Indexes have the same execution plans, they are doing the same computation; because Warm Cache COLT Indexes and Cold Cache COLT Indexes have the same execution plans, they are doing the same computation. Thus, whatever change in execution time is introduced by clearing the page cache is a change in I/O time. If more time is added to No Indexes under the cold page cache condition than is added to COLT Indexes, this

indicates that COLT indexes are reducing I/O time.

We compare the time added in the shift from warm to cold caches, and find that the time added to workload execution time with COLT indexes is 37.00 seconds less than the time added to workload execution time without indexes. Moreover, with a Cold Cache, COLT indexes lead to a 3.79 second drop in execution time, over a non-optimized database. An unpaired t-test allows us to reject at a .01% significance level the null hypothesis that there is no difference in the amount of additional I/O time (p=.0001). The 95% confidence interval indicates that the true difference in time added likely falls between 28.516069314 to 45.498120886. The difference is highly statistically significant. We thus conclude that the presence of COLT indexes can reduce I/O time.

5.1.3 CPU Boundedness

We are left with the question of why COLT indexes can reduce I/O time and yet increase execution time overall. Analysis of CPU activity during workload execution indicates that the reason for this unexpected result is CPU boundedness. With only primary key indexes and COLT-recommended indexes, some of the queries' execution times are determined largely by computation time. This is clearest in the case of the final query, which runs for approximately the final 1,000 seconds of the workload, in each condition. While Figure 3 indicates that there is some I/O activity early in the query's execution, this is followed by a long period in which the system barely blocks on I/O. Figure 4 demonstrates that the load on the CPU is steady throughout this period, as the computation required to fulfill the request takes place, without additional information from disk. Thus, even though COLT−− reduces I/O time, computation is the dominant contributor to execution time, and the benefits of reduced disk access are rendered insignificant in comparison. In fact, COLT−− can increase the execution time of the query, if it changes the form in which the data is manipulated, and if this manipulation requires more time.

Essentially, COLT indexes do not perform well under our test conditions because the COLT index recommender relies on PostgreSQL's query optimizer, which in this case does not anticipate that computation may be a more significant bottleneck than disk I/O. Wherever PostgreSQL's query optimizer performs poorly, COLT may suggest indexes that will not confer any benefits, even though the query optimizer will predict that they do — in fact, *because* the query optimizer will predict that they do. These indexes will be used in the query execution plan, but are unlikely to reduce execution time.

5.1.4 Index Selection Time

To select indexes for the workload required an average of 0.35 seconds. Tests on a 100 MB database indicate that the shift from a 100 MB to a 1 GB database approximately doubles the index selection time. Given that this process must be repeated only relatively infrequently compared to query processing, and that the time is dwarfed by query execution time, we conclude that index selection time is not a tremendous barrier to the usefulness of the COLT system, for sufficiently large databases.

5.1.5 Materialization Time

To materialize indexes for the workload required an average of 34.84 seconds. Tests comparing materialization costs on a 100 MB database relative to a 1 GB database reveal that materialization time in the larger database is approximately 4.2 times as high as materialization time in the smaller one. It is evident that this scales up with size much more than index selection time. Although this is almost as large the amount by which COLT indexes reduce added I/O time in the cold cache condition, this is a consequence of our relatively small workload. If the same queries or similar ones were run many more times, the time saved could far surpass the materialization time. Because materialization need only be accomplished once (in our implementation), while queries would be frequent, the materialization cost does not appear to be a barrier to use. It is important to note, however, that this may not be the case in the continuous version of COLT, in which materialization may be ongoing, and these costs would accrue over time.

5.2 SCISSOR

5.2.1 Execution Time

To test SCISSOR, we used the same 24-query workload that we used to test COLT−−. System parameters controlled which SCISSOR-suggested in-

dexes were materialized, and each test was run for 5 trials. All tests were run with a warm cache, clearing the page cache only once before each 24 query workload.

The first test, the results of which are show in Figure 5, measures the full potential of SCISSOR. The execution time for each query is depicted as a percentage of the time required to run the query on a non-optimized database. The results indicate that SCISSOR is extremely effective in reducing the execution time for most queries. For two queries, it suggested indexes that caused run time to increase. This may occur if SCISSOR suggests indexes on small tables, and if the query optimizer mistakenly anticipates that index access will be faster than a scan for those tables. Looking up a record in an index when the root is not in the page cache can be slower than reading in the heap file of all the records.

The second test varied the percentage of queries materialized from the full suggested set. Figure 6 displays the execution time of the 24 query workload without indexes and after materializing 25, 50, and 100 percent of the indexes suggested by SCISSOR. Indexes are ranked by the number of queries they may benefit, such that higher ranked queries are always materialized before lower ranked ones. In the bottom half of Figure 6, we show SCISSOR execution times on a smaller scale, to demonstrate that there is a decline in performance advantages as fewer indexes are materialized.

We also ran a test materializing all indexes that benefited at least 1, 2, 3, 4 and 5 queries. Figure **??** shows the improvement seen for each set of indexes compared to the same workload executed without indexes. Execution times of the SCISSOR trials appear on a smaller scale in the bottom half of the figure, to demonstrate the small decline in performance as fewer indexes are materialized. The increases in execution time here are even smaller than the increases we observe from materializing only a quarter of the indexes.

Lastly, we ran tests materializing as many indexes (in ranked order) as could be materialized before the end of a time limit. The execution times decreased as the limit on materialization time increased, as expected. We used 60, 120, 180, 240, and 300 second thresholds, and the results are presented in Figure 8.

SCISSOR suggested a total of 53 indexes on

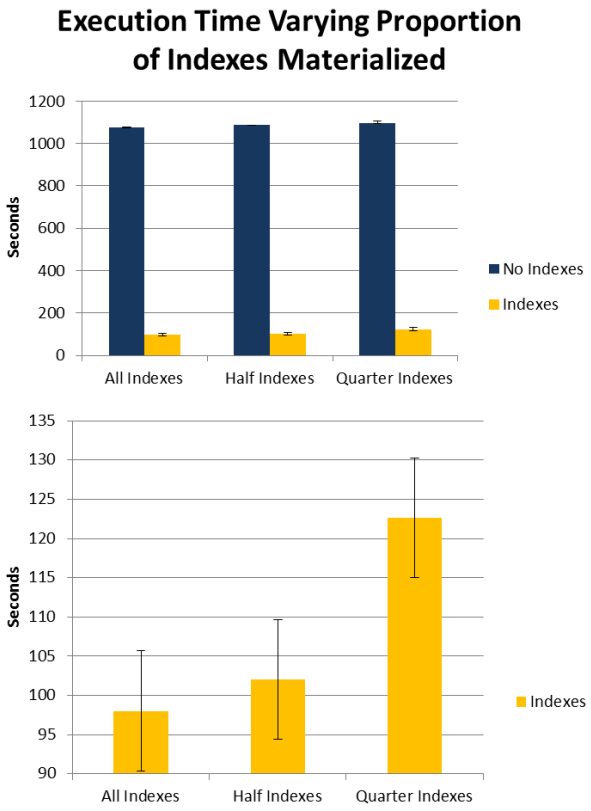

Figure 6: The average execution time in seconds for a 24 query workload, across 5 runs. Error bars reflect standard deviation. The SCISSOR runs vary the proportion of suggested indexes that are materialized.

this 24 query workload. Most were small, though 11 were on the largest table in the database, the `lineitem` table. Total index space for all 53 candidate indexes was 3.2 GB, which reflects a database:indexes size ratio that is acceptable in many real world situations. The top-ranked quarter of SCISSOR indexes, which yield hardly worse run times, require only 800 MB.

These results are very encouraging. The set of tests materializing 25, 50, and 100 percent of SCISSOR's recommended index set resulted in very little change in query execution time; the benefits were mostly gained from the first 25% most-used indexes. This supports the hypothesis that SCISSOR's ranking method, simply sorting by the number of queries that could use a given index, is effective. While execution time in the trials that materialized 50% and

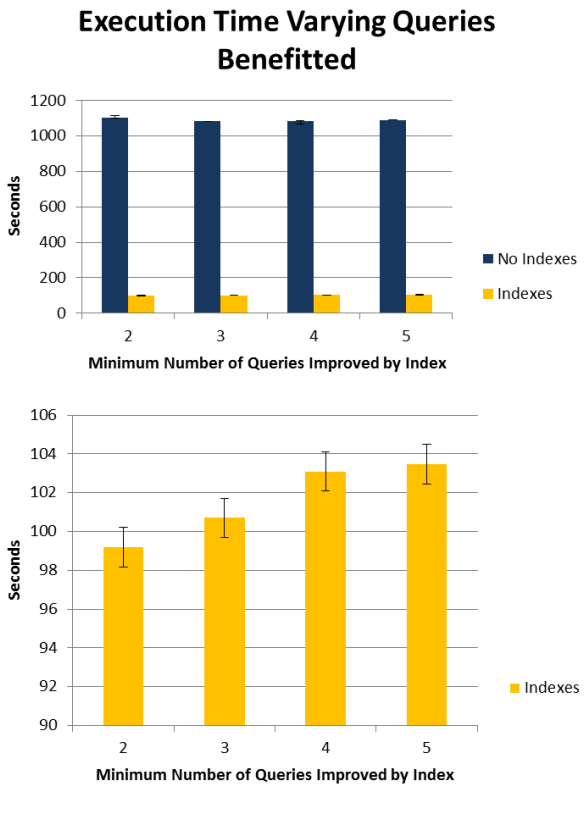

Figure 7: The average execution time in seconds for a 24 query workload, across 5 runs. Error bars reflect standard deviation. The SCISSOR runs vary the number of queries that must be potential beneficiaries of the index, in order for an index to be materialized.

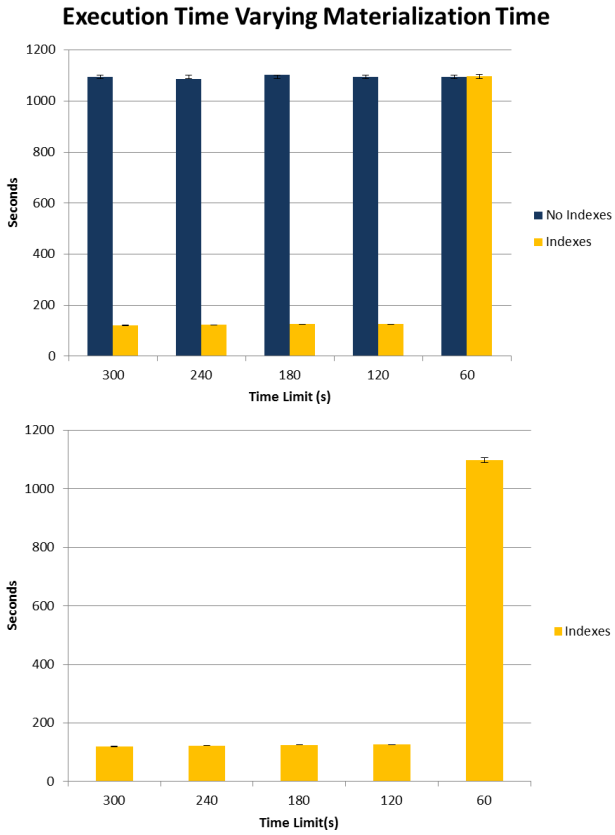

Figure 8: The average execution time in seconds for a 24 query workload, across 5 runs. Error bars reflect standard deviation. The SCISSOR runs vary the amount of time the system may spend materializing indexes.

25% was higher than in the trial with all indexes materialized, examining the No Indexes condition reveals that this slowdown is insignificant compared to the benefits of the first quarter of the indexes.

The results from the second set of tests, materializing indexes only if they benefit some threshold number of queries, are equally promising. There is some slowdown without the indexes that benefit a single query, slightly more slowdown without indexes benefiting 2 queries, and so on, but the slowdown is insignificant compared to the benefits from the indexes that *are* materialized, even with a high threshold. This test verifies that a queries helped threshold is also a valid method to cut down index materialization time and index space, while retaining nearly all the benefits for execution time.

The trials with the time limit are the most relevant to the common case. If index selection is completed once and the indexes used many times, it may not be important; but to enable the sort of online tuning that is COLT's goal, it is crucial that the time spent materializing indexes be less than the speedup achieved by the indexes. The results of these materialization time-limited trials reveal that there is either a single index that is extremely beneficial to our workload, or there may be complex interactions between multiple indexes, one or more of which cannot be materialized in only 60 seconds, but can be in 120. In all cases except the 60 second threshold condition, the time saved during workload execution far outweighs the time spent materializing indexes.

Index materialization took between 408 seconds for all 53 indexes and 187 seconds for 18 that benefited 5 or more queries. Given that execution time

improved on the order of 1000 seconds for even this small workload, we believe we can safely conclude that SCISSOR improves overall execution time under some conditions.

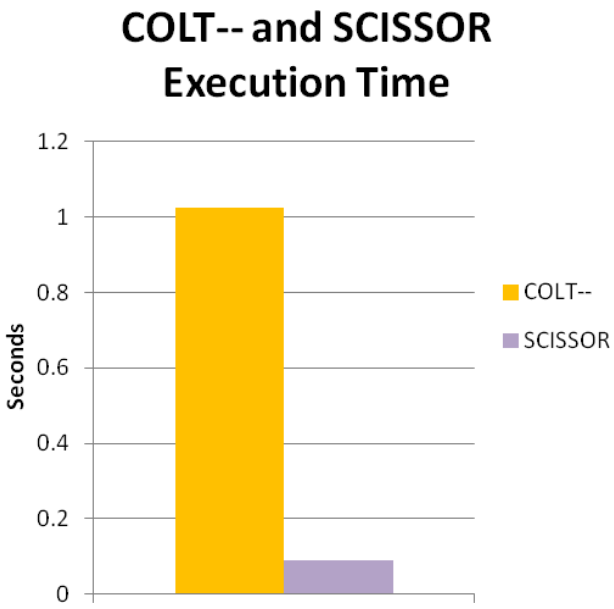

Figure 9: The relative execution times with COLT– and SCISSOR indexes, for each query in a 24 query workload. The execution times are represented as proportions of the execution time in a non-optimized database.

Figure 9 shows COLT−− and SCISSOR workload execution times as proportions of workload execution time on a non-optimized database. SCISSOR, even with its extremely lightweight approach to index selection, proves useful in a situation where more complicated systems may fail.

6 Future Work and Conclusion

6.1 COLT

We believe that COLT warrants more study. Although we are unsure of the implications of their tests, and although our results demonstrate COLT's ineffectiveness when PostgreSQL's optimizer misleads it, it would be interesting to see results when the PostgreSQL optimizer is more accurate. We would be interested in reproducing their tests more exactly, and determining whether we can obtain the same speed ups they see. We also remain interested in producing the missing comparison of COLT to a non-optimized database, in circumstances in which the workload would not be CPU bound.

6.2 SCISSOR

We believe SCISSOR could be usefully extended by the generation of multi-column indexes. While single column indexes do certainly provide some opportunities for optimization, as evidenced by our results, greater gains could be achieved by a strategy that proposes more complex indexes. This is especially true for workloads with complicated, decision-support workloads, exactly the sort of workload that is hardest to interpret as a human observer. We believe that doing this without high overhead will be the development most crucial to creating truly practical automatic database tuning systems. Striking a balance between overhead and index quality is central challenge in automatic index generation.

Another interesting experiment would be to test SCISSOR as an online tuning tool. We could run SCISSOR iteratively, every ten queries, track which indexes have been suggested most highly and most recently, and compare SCISSOR to COLT in the situation COLT was designed to handle.

6.3 Conclusion

We believe heavyweight solutions are the wrong approach to automatic index selection. The developers of commercial systems have focused large amounts of time and resources on developing solutions for tuning indexes, but always with very complex approaches. We developed an extremely lightweight tool for recommending indexes, which reduced execution time on a set of complex queries by 90.9%, with very little time and computation dedicated to index selection itself. This leads us to believe that it is worth exploring simpler, faster mechanisms for recommending indexes.

By testing COLT against a non-optimized database, excluding index selection time from our performance metrics, giving it foreknowledge of its workload and an effectively unlimited amount of memory for indexes, we attempt to give the COLT index recommender every possible advantage. We find limited performance improvements, but not because of COLT's index recommender. COLT's problems seem to stem from its reliance on the DBMS's

embedded query optimizer that hampers its performance. This too leads us to believe that there is a place for simple models of index selection, for approaches that do not rely on computationally expensive and sometimes flawed external tools. For now, however, it seems the right system has not been found. More research will have to go into index recommendation algorithms before lightweight automatic index tuning will be a feasible alternative to administrator expertise.

References

S. Chaudhuri and V. Narasayya. 1997. An efficient, cost-driven index selection tool for microsoft sql server. In *Proceedings of the International Conference on Very Large Data Bases*, pages 146–155. Citeseer.

K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. 2006. Online database tuning. Technical report, University of California Santa Cruz.

K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. 2007. On-line index selection for shifting workloads. In *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, pages 459–468. IEEE.

Karl Schnaitter. 2011. Postgres new features guide.

G. Valentin, M. Zuliani, and D. C. Zilio. 2000. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *In ICDE*, pages 101–110.

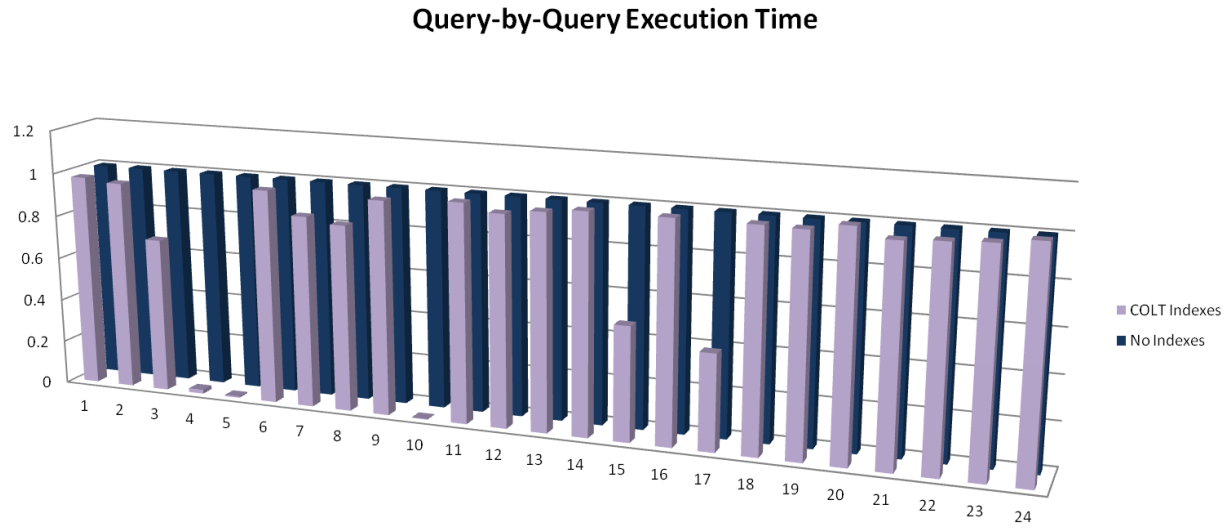

Figure 2: COLT–: The relative execution times for each query in a 24 query workload, across 10 runs. The execution time of each query in a non-optimized database has been normalized to 1.0, to enable comparing all queries on the same scale.

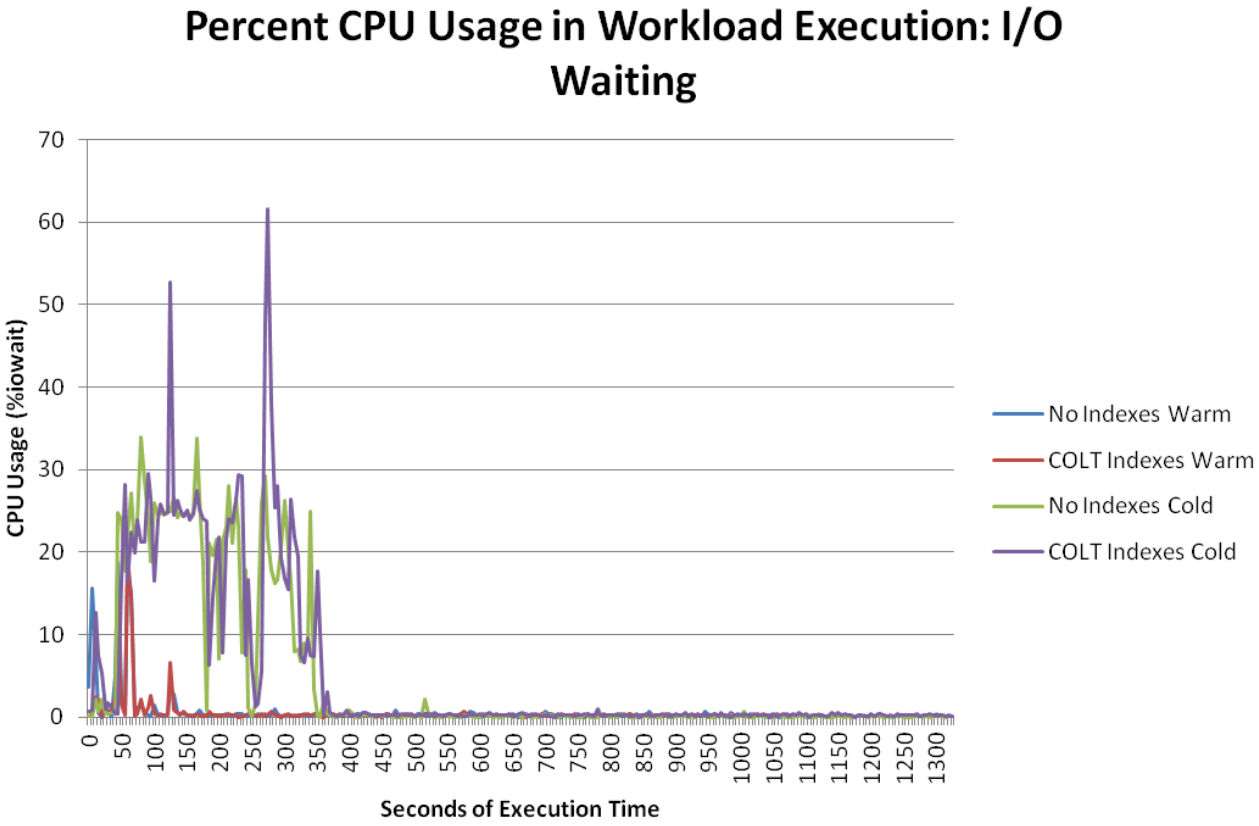

Figure 3: CPU usage sampled at 5 second intervals over the course of workload execution. This tracks the percentage of time that the CPU was idle while there was an outstanding I/O request.

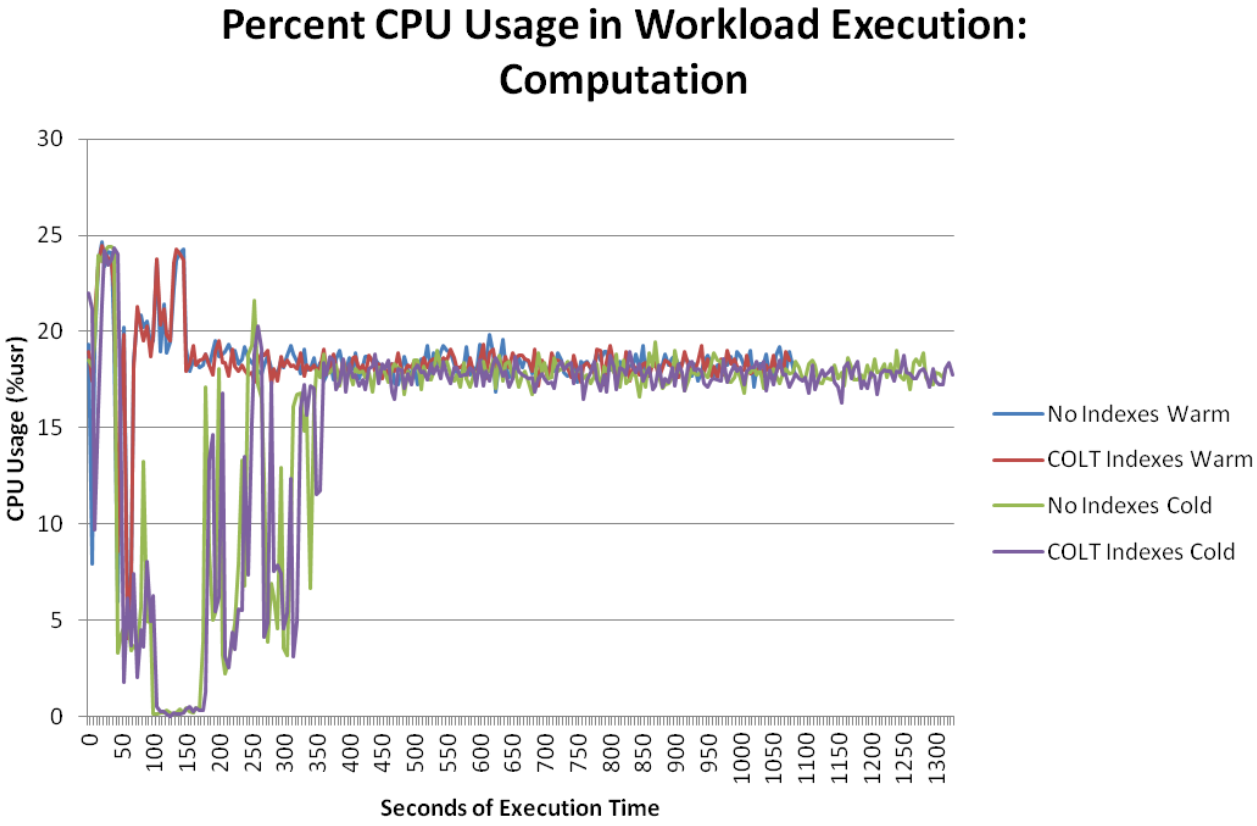

Figure 4: CPU usage sampled at 5 second intervals over the course of workload execution. This tracks the percentage of CPU utilization dedicated to user-level computation.

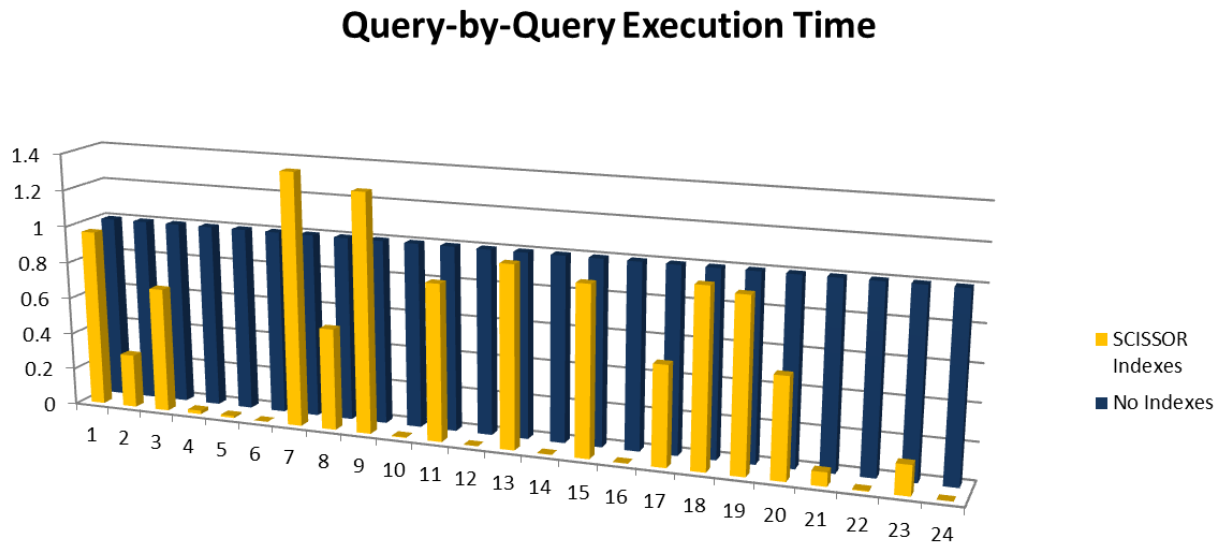

Figure 5: SCISSOR: The relative execution times for each query in a 24 query workload, across 5 runs. The execution time of each query in a non-optimized database has been normalized to 1.0, to enable comparing all queries on the same scale.