



SQL:Trek Automated Index Design at Airbnb

Sam Lightstone

Airbnb

Toronto, Canada

sam.lightstone@airbnb.com

Ping Wang

Airbnb

San Francisco, USA

ping.wang@airbnb.com

ABSTRACT

Automating index design has been an active area of research for decades due to the significant impact that indexes have on query performance and database efficiency. Existing approaches range from brute-force search to cost-based optimizations and, more recently, machine learning techniques. However, many suffer from high computational costs, reliance on inaccurate cost models, or the need for deep integration with database internals.

We introduce SQL:Trek, a time-efficient tool for automated index design that operates entirely as an external utility. SQL:Trek leverages query compiler cost models to identify effective indexes while mitigating false positives through execution on a lightweight simulation database. This approach enables fast, iterative index selection without modifying database internals, making it broadly applicable across relational databases, including most MySQL® and PostgreSQL® derivative databases.

Our evaluation demonstrates that SQL:Trek delivers significant query performance improvements while keeping index selection computationally efficient, with most workloads analyzed in under five minutes. Unlike many cost-based what-if analysis methods, SQL:Trek significantly improved performance of many production workloads while avoiding the majority of detrimental index recommendations caused by optimizer misestimates. These results highlight SQL:Trek as a practical, scalable solution for automated index tuning in modern database environments.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750638

PVLDB Reference Format:

Sam Lightstone, Ping Wang. SQL:Trek Automated Index Design at Airbnb. PVLDB, 18(12): 5210–5222, 2025.
doi:10.14778/3750601.3750638

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol18.html.

1 INTRODUCTION

Efficient query performance is fundamental to modern database systems, especially in large-scale enterprise and cloud-based environments where scalability and responsiveness are critical.

Index selection plays a pivotal role in query optimization, significantly impacting execution times and overall system efficiency. The presence or absence of a key index can alter query performance by orders of magnitude. However, manual index tuning requires deep expertise in database technology, workload characteristics, and resource constraints, making it both intricate and resource-intensive. As databases scale and complexity increases, automated index design has become an essential strategy for enhancing performance while reducing administrative burden.

Indexes, as auxiliary data structures, accelerate query execution by enabling efficient data retrieval. Yet, identifying an optimal index set is challenging due to trade-offs among query performance, maintenance costs, and storage overhead. Effective index selection hinges on workload characteristics—such as query frequency, join patterns, and filtering conditions—since suboptimal configurations can degrade performance, inflating latency and resource use. Notably, well-designed indexes can yield order of magnitude query latency improvements.

The problem of automated index design is inherently complex, driven by a vast search space and computational intricacy. For T tables, each with N indexable columns, the number of possible multi-column index designs is given by:

$$\left(\sum_{k=1}^N \frac{N!}{(N-k)!} \right)^T \quad (1)$$

For instance, with $N = 10$ and $T = 12$, this yields approximately 10^{84} possibilities—surpassing the estimated number of atoms in the observable universe ($\sim 10^{80}$). Practical heuristics, such as restricting candidates to query-referenced columns or prioritizing selectivity, mitigate this explosion, yet the core challenge remains NP-hard. Determining an optimal index subset that minimizes execution time while balancing storage and update costs requires combinatorial evaluation, cost estimation, and performance simulation. Moreover, dynamic workloads necessitate adaptive strategies to refine index recommendations continuously.

Traditional index selection relies on heuristics and query optimizer insights, while recent advances employ machine learning and reinforcement learning to predict index utility and scale decision-making. Cloud-based databases introduce additional complexities, including multi-tenancy, resource limits, and heterogeneous workloads, amplifying the need for robust automation. At Airbnb® we manage hundreds of databases and tens of thousands of queries, rendering manual design impractical. Indexes also incur storage and runtime overhead, necessitating a system-wide approach rather than query-specific optimization.

Addressing these challenges demands innovative algorithms, scalable optimization, and adaptive learning models. Our work at

Airbnb with multiple relational database engines underscores specific requirements: a cross-engine solution, which in turn requires minimal internal database modifications; resilience to cost-model inaccuracies; and rapid, low-cost execution. With databases where index creation on large tables spans days, trial-based evaluation becomes infeasible, further emphasizing the need for efficient automation.

1.2 Our contributions

SQL:Trek makes several contributions to the area of automated index design.

First, practically, we created the solution entirely outside of the database engine, as a viable utility for the majority of MySQL [23] and PostgreSQL [26] derived databases. This covers a huge swath of database products in the market today such as MySQL, PostgreSQL, Amazon Aurora[®] [1], TiDB[®] [25], AlloyDB[®] [13], CockroachDB[®] [7], MariaDB[®] [20], EnterpriseDB[®] [11] etc. Although we haven't tested SQL:Trek on all of these systems, we have tested it on MySQL, Amazon Aurora, PostgreSQL and TiDB with good success.

Second, SQL:Trek is designed to minimize the false positives that result in indexes being recommended that are of minimal or negative consequence to the workload. While SQL:Trek does not provide a guarantee of no false positives, in our testing we have found the presence of false positives to be extremely low. This is a large improvement over the what-if analysis approaches that leverage SQL cost models to select indexes. To be explicit, SQL:Trek aims to dramatically reduce the recommendation of indexes with negative value, not simply detect and remove them after they are deployed into production.

Third, SQL:Trek offers time-efficient processing, enabling analysis of large, production-class databases and generation of solutions—including the necessary CREATE INDEX DDL—typically in under 20 minutes for OLTP workloads. While this is certainly slower than the cost-model-only what-if approaches that use virtual indexes (indexes without data), it is adequate for our operational needs because it is fast enough that the workload analysis can be executed daily, or even a few times per day per database cluster if needed. This is also a large improvement over approaches that evaluate candidate indexes on a full-scale copy of the database. Such strategies are impractical for evaluating a large set of potential candidate indexes, especially when the source database is tens of terabytes per node. Creating a single index on a 10+ TB database can take several hours on current systems, making it infeasible to evaluate dozens of new indexes across hundreds of databases. While many index recommenders are time-efficient—often more so than SQL:Trek—they typically cannot achieve this efficiency while also maintaining a low false-positive rate.

2 BACKGROUND ON AUTOMATED INDEX DESIGN

We summarize highlights from several influential papers that have shaped industry approaches to automated index recommendation.

Index selection is a critical aspect of physical database design, significantly impacting query performance. Over the years, researchers have developed various methodologies for automated

index selection, ranging from early heuristic approaches to modern AI-driven techniques.

Finkelstein et al. [12] present one of the earliest formalizations of physical database design, emphasizing heuristic and cost-based approaches to index selection. They highlight the trade-offs between query performance and storage constraints, laying the foundation for subsequent research in automated index design.

Chaudhuri and Narasayya [5] introduce a cost-driven index selection tool for Microsoft SQL Server[®], leveraging a query optimizer to estimate the benefits of potential indexes. Their work establishes a systematic method for index recommendation, integrating workload analysis to guide index selection.

Valentin et al. [29] propose DB2 Advisor, an index recommendation system that leverages the DB2[®] query optimizer both for designing the candidate indexes as well as determining their potential benefit to the workload. Notably, the query compiler itself is used to detect columns of interest and generate candidate virtual indexes for consideration. In this architecture the SQL compiler is expanded to both design the virtual indexes and evaluate them. DB2 Advisor leverages optimizer feedback to iteratively refine index choices. This work demonstrates the effectiveness of integrating index selection into database query optimization processes. The approach builds on the strategies in [5, 12] using what-if analysis with the SQL cost model but extends this by having the SQL compiler generate the what-if candidates for evaluation.

A key insight of these early papers was to recognize that the SQL compiler is the deciding factor in whether an index will be utilized for any SQL query. Ultimately, if an index is not selected by the SQL compiler, then it is useless. Consequently, the SQL cost model is an invaluable tool in modeling the value of a potential index.

Lightstone and Bhattacharjee [18] extend automated index selection to multidimensional clustering tables. Their approach considers complex data distributions and query patterns, optimizing clustering strategies alongside index structures. This research is particularly relevant for high-dimensional analytical workloads. Notably, this work includes the use of data sampling in the process of automated physical design. While the paper thematically builds on the work of Valentine et al [29], leveraging the SQL compiler to generate candidate designs and the cost model to assess what-if scenarios—it further extends these ideas by introducing data sampling to reasonably estimate the cardinality of cells resulting from clustering multiple attributes simultaneously. This dynamically produces cardinality insights across columns that are not normally available in the system catalogs.

Chaudhuri and Narasayya [6] describe the Database Engine Tuning Advisor (DTA) used in SQL Server as of 2020. Their algorithm has several notable attributes, in particular a merge phase that allows indexes initially tuned for specific queries to be merged (subsumed) in order to produce a final recommendation for the database that provides a global optimum rather than a collection of local optimizations per query. This is achieved by pair-wise comparisons of indexes with the same leading keys, and retaining the index with the higher individual benefit if it is a superset of the keys of its pair.

Das et al. [8] shift the focus to large-scale cloud environments, proposing an automated index selection framework for Microsoft Azure SQL Database. Their system scales to millions of databases, utilizing machine learning and reinforcement learning techniques

to continuously adapt index recommendations based on evolving workloads. This paper reports the inaccuracy of the what-if modeling based on SQL cost model estimates, drawing on known literature regarding the inaccuracy and pitfalls of SQL compiler cost models [10, 16, 19], as well as reporting their own finding on the false positives in automated index design that depends on these estimates. Das et al. report a false positive rate of 11%, meaning that up to 11% of new indexes recommended by the autoindexing technology caused query regression. Their approach detects such false positives and actively removes them from the system.

The idea of leveraging the SQL compiler cost model for physical design was extended to other database design automation problems such as data clustering, table partitioning, and materialized views [3, 4, 18, 31], demonstrating its wide applicability.

Yadav et al. [30] present AIM, a practical automated index management framework for SQL databases at Meta (Facebook). Their approach emphasizes practicality and real-world deployment challenges, ensuring index recommendations remain optimal over time. Unlike the previous approaches that were based on what-if modeling of candidate indexes that are evaluated by the SQL compiler cost model, Yada et al create clones of the databases under study, and physically evaluate candidate indexes in the cloned environment. This has the advantage that the index selection is not primarily dependent on the SQL compiler’s cost model, but actually evaluated for their impact under load. The probability of false positives is dramatically reduced as a result. However, the implementation is costly since it requires a clone of the entire database under study.

The recursive strategy used in the paper by Schlosser et al [27] efficiently tackles multi-attribute index selection by breaking it into smaller subproblems, avoiding the infeasible task of evaluating all index combinations. It starts with all candidate indexes, recursively selecting subsets while assessing the cost reduction of adding each index, considering only beneficial ones. Unlike greedy methods, it accounts for index dependencies, avoiding early pruning of useful options. Intelligent pruning and heuristics eliminate unpromising combinations, enhancing scalability for workloads with many tables and attributes. This enables fast convergence to a high-quality index set without brute-force overhead.

Several papers have explored applying AI/machine learning methods to the problem of automated index design [17, 22, 32]. These have focused mostly on reinforcement learning and genetic algorithms. While effective in their results, they require large numbers of training measurements (tens of thousands, or more) that make them currently impractical for industrial use at Airbnb.

Ding et al. [10] grapples specifically with the problem of SQL compiler optimizer misestimates leading to damaging index recommendations. They create an ML classifier that takes as input two query execution plans for the same query and predicts which will run longer. The classifier is used to compare the original plan without the newly recommended indexes and the one that includes them. This serves as a second model (in addition to the SQL compiler costing) for whether the recommended indexes will regress some queries. While it greatly helps to reduce the probability of recommending indexes that regress any single query, it has the consequence of rejecting index recommendations that regress one or more queries but have a strong net positive value on the workload by improving other queries.

These papers collectively illustrate the evolution of automated index selection, transitioning from rule-based heuristics to machine learning-driven systems capable of handling large-scale, dynamic workloads. Future advancements in this domain are expected to further enhance efficiency through AI-driven optimizations [32] and cloud-native database architectures.

3 DESIGN

We present the design objectives, guiding principles, and practical design constraints of SQL:Trek, as well as both the high-level architecture and detailed design specifications.

3.1 Objectives

Index Selection Optimization Problem

The problem of finding an optimal set of indexes C^* to minimize the execution time of a workload W can be formulated as follows:

Given:

- 1) A workload $W = \{q_1, q_2, \dots, q_n\}$
- 2) $(T(q, C))$ is the execution time of a query q_i when using index configuration C .
- 3) $(T(W, C) = \sum_{q \in W} T(q, C))$ is the total execution time of the workload under configuration C .
- 4) C_{init} is the initial set of pre-existing indexes
- 5) C' is the set of new candidate indexes
- 6) C_{sub} is a subset of the candidate indexes.
- 7) C^* is the final recommended index configuration
- 8) s is the storage required for an individual index recommendation.
- 9) S_{max} is the storage constraint for a new index configuration.

We define an optimization goal to find the optimal index configuration C^* such that:

$$C^* = \arg \min_{C_{sub} \subseteq C'} T(W, C_{init} \cup C_{sub}) \quad (2)$$

subject to:

$$\sum_{s \in C_{init} \cup C_{sub}} s \leq S_{max}$$

Complexity and Constraints

This problem is NP-hard, meaning that finding the exact optimal solution requires exhaustive evaluation of all possible index configurations. Our goal is to minimize the total execution time of the workload while ensuring that the storage cost of the selected indexes does not exceed the given storage budget. We also aim to avoid selecting indexes that provide little to no benefit, or cause workloads to regress.

Execution Time Constraint: Since index recommendations are needed periodically (but not continuously), the average analysis time T_a must be kept under a reasonable limit. At Airbnb, where we manage hundreds of SQL databases, we require:

$$T_a \leq 60 \text{ minutes}$$

This ensures timely recommendations without excessive computational overhead. Practically, this means that the index recommender is fast enough that if desired it can be run daily for each database cluster in our fleet with minimal infrastructure overhead.

3.2 SQL:Trek design principles

- 1) **Performance of the index recommender process.** Index design should be periodically reevaluated to adapt to evolving query patterns, as well as changes in data volume and distribution. Frequent reassessment, such as multiple times daily, is typically unnecessary, as index configurations rarely require such rapid adjustments. A re-evaluation process completed within 60 minutes is adequate for our needs at Airbnb, enabling the analysis of dozens of databases per day on a single evaluation server.
- 2) **SQL compiler costing of candidate indexes.** Since the SQL compiler will select from existing indexes at query compile time during production operation of the database, only indexes that are chosen by the compiler are valuable to a database. It is a necessary design point that the SQL compiler must select the indexes for recommendation.
- 3) **Minimize false positives.** Modern database cost models are often inaccurate, leading to false positive index recommendations and potential deployment of poor designs. We address this by evaluating candidate indexes on representative data samples rather than relying solely on cost estimates. The sample must be sufficient to provide confidence in two key aspects: whether an index is useful, and its relative utility compared to other candidates. While data sampling introduces distortions in data size and distribution that make samples imperfectly representative of production databases, our approach is based on two core hypotheses: first, that optimizer estimates combined with actual runtime measurements on sampled data can provide stronger signals of index efficacy than cost estimates alone for most OLTP queries; second, that these measurements can help identify many indexes that would negatively impact the production database.
- 4) **No database internal changes.** The entire design process, from start to finish, must be possible without the need to modify database internals. This is necessary, as we have no access to database internals for some of the databases we use at Airbnb.
- 5) **Simulation quality.** We make use of a simulation database that clones a sample of data. Using a sample is fraught with problems, changing the cardinality of data, the IO, the relative size of tables that will be joined (even though the sample rate is the same for all tables, we

impose a min and max sample which changes the effective sample for many tables.). If the goal was to achieve an exact emulation of the system, then any sample would be problematic. Our approach is to create a “good enough” emulation that the recommended indexes will be high quality when applied to the production environment. We do not require perfection for this; in fact, we are deliberately accepting a compromised approach.

- 6) **Covering indexes as an anti-pattern.** A covering index is a secondary index that includes all the columns of a table that are referenced by a specific query. It has the benefit that the query can be fully answered by examining data in the index alone, and a lookup into the table data is not needed. While covering indexes provide maximum performance potential for a particular query, they are often specialized to the query, and therefore provide lower utility to the overall workload. Second, because they include all reference columns for the query against a table, they tend (on average) to be wide, and therefore require more storage and are maintenance intensive. For the initial implementation of SQL:Trek described here, we consider covering indexes with reference columns as an anti-pattern. This is purely a simplifying assumption, as there are common cases where such indexes are excellent choices. Reference columns therefore are explicitly not included in our index designs in this initial version, though we may relax this in the future. To illustrate this concern, consider a sales table with five columns {sale_id, product_id, customer_id, sale_date, amount}. The following query accesses all five with additional predicates on just product_id. It would be tempting to create a covering index that includes all of these columns, with product_id as the leading key part. However, doing so would double the size of the table storage with this one index, and double the write cost due to index maintenance. An index defined on just the predicate column likely provides most of the performance benefit at a fraction of the storage and maintenance overhead.

```
SELECT sale_id, product_id,
customer_id, sale_date, amount
FROM sales
WHERE product_id = 42;
```

3.3 Practical design constraints

Airbnb makes extensive use of MySQL-derived databases [23]. These systems commonly constrain the number of secondary indexes per table to 64. As a result, not only is the recommendation of new indexes constrained to a maximum of 64 (less the number of pre-existing indexes) per table, but at no point in the design exploration can more than 64 indexes be defined on a single table at one time—even for what-if analysis by the query compiler. This is sufficient for performing an exhaustive search of column combinations for up to 6 indexable columns per table.

While it may be possible to modify the MySQL open source to relax this constraint, doing so is not practical for the cloud services

we utilize, and would violate our design objective of avoiding changes to database internals. Therefore, our solution operates within this constraint and we have been able to achieve good results despite this limitation.

PostgreSQL based systems do not have this constraint, and when running SQL:Trek in these environments we relax the upper bound on candidates per table to 256, which is sufficient for performing an exhaustive search of column combinations for up to 8 indexable columns per table. We still prefer a constraint in order to limit the search space since it strongly affects the execution time.

3.4 High-level design

Much like the solutions used in Chaudhuri [5], Valentin [29] and Lightstone [18], SQL:Trek creates a large number of candidate indexes for “what-if” exploration by the query compiler and relies on the SQL compiler’s costing to detect which of these indexes are optimal. However, unlike these approaches, SQL:Trek creates these candidate indexes on a simulation database, not in a virtual catalog. SQL:Trek creates a pool of new candidate indexes for consideration C' known as the *Candidate Pool*. After the costing process has selected a subset of new indexes and unreferenced indexes are removed from the simulation, the query workload is executed on the simulation database to estimate the relative benefit of these indexes. Indexes observed to have minimal or negative impact when queries are run on the simulation are removed from the *Candidate Pool*. The query workload is rerun in the simulation environment until a stable solution is derived where all of the recommended new indexes provide clear value, resulting in the *Winners Pool* C^* that is the union of the initial pre-existing indexes C_{init} and the surviving remaining indexes from C' . Figure 1 illustrates the main elements of this architecture.

The simulation database is populated with a sample of the source databases, up to a maximum data size. For our implementation we have used a 5% sample, with a minimum of 20K rows per table, and a maximum of 1M rows per table as the constraints. The principle behind these choices is that we want the simulation to have enough data to reasonably model the benefit of new secondary indexes, but not be so large that the simulation will take a long time to populate. The choice of sampling parameters generally results in a simulation database that is modestly sized, allowing for dozens of candidate indexes to be created in seconds.

What-if analysis is the process of presenting candidate indexes to the SQL compiler and allowing the compilation process to assess the potential benefit of these indexes. We perform this what-if analysis by creating candidate indexes on the simulation database and using EXPLAIN to detect whether the cost model chooses any of the candidate indexes. Unlike prior art that uses virtual indexes to perform this what-if analysis, where the indexes exist in definition only and are not populated, our approach uses materialized indexes. Having real indexes has two critical advantages. First, it allows us to use the database management system without code modification, since most commercially available database systems do not currently support creating virtual indexes. PostgreSQL is a notable exception, and we could special-case SQL:Trek for PostgreSQL to use virtual indexes in this step—an optimization we have not yet implemented. Second, after indexes are selected by the cost model, we are able to evaluate their impact by actually executing queries with the new indexes on the simulation database. Because the

candidate indexes are created on a small simulation database which is created on a separate server, the index creation process is fast, and does not interfere with production workloads at all. The storage space required, even for dozens of indexes, is small since the entire simulation database is by design created from a small sample.

Notably, we make use of EXPLAIN to detect whether a candidate index is selected by the query optimizer, but we do not make use of any compiler costing data beyond this. SQL:Trek is informed by EXPLAIN whether an index is optimal for a query, but the relative merit of indexes is modeled entirely based on the query improvement observed when running queries on the simulation database.

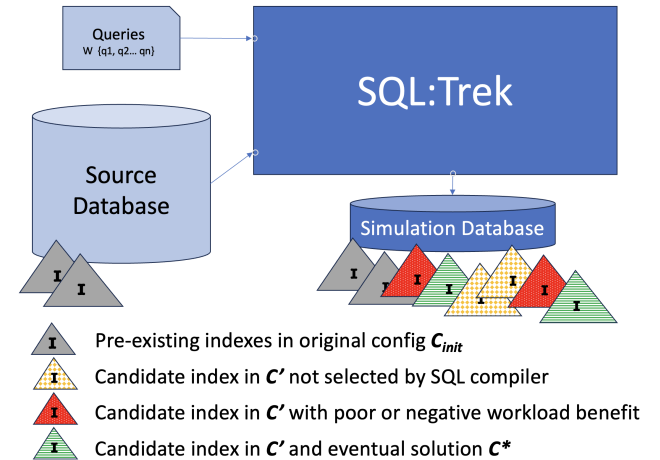


Figure 1: SQL:Trek at a glance.

3.5 Design details

The proposed method provides an automated approach for index selection in SQL databases based on workload analysis. The user supplies the database name, credentials, and a set of SQL queries for assessment. The queries can be provided either by an input text file, or optionally (on MySQL) retrieved from the database’s query history. Additionally, the user may specify a storage constraint for the total space allocated to indexes as a percentage of the table size. If no constraint is provided, the system defaults to allowing unlimited indexes.

The index design follows eleven steps.

Step 1: SQL Workload Analysis

The system parses the SQL workload to identify which columns are used in operations such as:

- 1) Equality comparisons and IN-list conditions
- 2) Order by
- 3) Joins
- 4) Distinct
- 5) Inequality, between and like
- 6) Grouping and aggregation




Since direct access to database internals is not assumed, an open-source MySQL and PostgreSQL compatible SQL parser is used. After evaluating JSQLParser [15] and Apache Calcite SQL [2], we chose

Calcite for its robustness and extensibility. We found multiple query patterns that were simpler to handle with Calcite, including subqueries, expressions with brackets, etc.

Step 2: Cloning the Database Schema

A temporary simulation database is created by cloning the schema, including all table definitions that are referenced in the SQL workload. Existing secondary index definitions are also replicated. Constraints, including foreign keys, uniqueness, cascade, etc., are also cloned. At this stage, data is not cloned – the tables and indexes are empty.

Step 3: Candidate Index Design

Starting with the database configuration C, which includes an initial set of indexes designed by human experts, SQL:Trek designs a larger domain of additional indexes C', known as the *Candidate Pool* (described in more detail in section 3.6). For each table referenced in the workload the system generates a candidate index design based on column usage patterns. Higher priority is given to columns involved in highly selective operations (equality predicates), followed by those used in less selective operations. The index designs in the *Candidate Pool* are shown in Figure 1 by the icons {, , }.

Step 4: Data Sampling

To reduce computational overhead, a sample of the data is extracted from the source database. The sampling strategy involves selecting 5% of the data, with a minimum sample of 20,000 rows and a cap of 1 million rows per table. This range was chosen in order to keep the simulation database size modest, regardless of the size of the source database. This is dense enough to surface the relative CPU and IO effects of new indexes and provide some insight on how they may benefit the production system. 5% is the default, and the sampling rate is configurable.

The use of real data has the consequence that data distribution and range will generally follow that of data in the source database both for the column data as well as the correlation of data between columns and between tables. Sampling data from base tables can alter the underlying data distributions and introduce inaccuracies in SQL optimizer cost estimates and the physical response of the simulation. This, in turn, reduces the fidelity with which the simulation populated with a sample reflects the behavior of the complete database. Nonetheless, certain performance trends generally persist—for example, highly selective index lookups tend to be more efficient than full table scans, and implicit ordering via indexes is typically faster than explicit sorting. While the resulting costing and performance estimates may be imperfect, the use of real (albeit partial) data often offers a better method for detecting negative indexes than cost model alone with comprehensive stats, as our experimental results illustrate.

Step 5: Establishing a Performance Baseline

The SQL workload is executed against the simulation database to measure baseline query performance before any indexing modifications.


Step 6: Creating Candidate Indexes

Candidate indexes in C' generated in Step 2 are physically created on the simulation database.

Step 7: Index Evaluation with EXPLAIN

Rather than executing queries, the system compiles the workload using the EXPLAIN statement to determine which candidate indexes the SQL optimizer considers beneficial. The use of EXPLAIN injects the cost model into the selection process. A large number of candidates may be include in C', from which the SQL compiler will assess the benefits of these candidates to individual queries. Ideally this step would be performed entirely with virtual indexes (indexes that exist in the catalog space, but are not materialized), but since that is not widely available in MySQL, we use actual indexes on the simulation database. The key purpose of this step is to leverage the SQL compiler's cost model to identify which indexes in C' are useful to the workload based entirely on SQL compiler costing. Note, at this point we are only concerned with a decision on whether each index is useful or not, and we do not use the cost model to assess "how much" benefit each index may provide.

Step 8: Filtering Unused Candidate Indexes

Indexes that are not selected by EXPLAIN for any queries in the workload are physically dropped from the simulation. This typically prunes the majority of candidates from consideration (more than 80%). These are the index candidates shown in Figure 1 with the icon .

Step 9: Performance Evaluation with New Indexes in Simulation


The remaining candidate indexes were each selected for use by EXPLAIN for at least one query in the workload. The SQL workload is re-executed in the simulation database to measure performance changes per query with the new indexes. The purpose of this step is to quantify the relative benefit with higher confidence than the cost model alone would allow.

The speedup or regression of each SQL query is attributed to each new index referenced by the query. The total benefit (which may be negative) for an index is the sum of all gains and losses from all queries that reference it.

Running a physical measurement, even on a sample, allows us to identify and subsequently remove indexes with minimal or negative impact. Execution of these queries in the simulation has a large advantage over cost modeling alone, since the cost model is based on table statistics, which generally exclude correlation modeling between columns and across tables. Similarly, the cost model can estimate physical resource usage (CPU, IO latency, etc.) while the simulation run provides an actual observation.

Step 10: Iterative Index Refinement

Indexes that provide minimal or negative performance improvements are removed, and the workload is reevaluated since the removal of some indexes may significantly impact the value and

usage of the remaining indexes. These indexes are shown in Figure 1 as . The process is repeated until only beneficial indexes remain. We use the following criteria for this detection:

- 1) Recommended indexes should provide at least 10% performance benefit in the simulation environment to at least one query.
- 2) Recommended indexes should have a net positive impact on the entire workload.

Step 11: Optimized Index Selection for Storage Constraints Using the Knapsack Approach

If a storage constraint is defined, the system applies a knapsack-based optimization to select the most valuable indexes while staying within the allowed storage limit. Each candidate index is evaluated based on:

- 1) Benefit: Improvement in query performance (measured in workload execution time saved).
- 2) Cost: Storage space required for the index.

The selection process follows the 0/1 knapsack model, where the goal is to maximize total performance gain without exceeding the available storage budget. The system prioritizes indexes that provide the highest performance-per-byte ratio, defined as:

$$\left(\sum_{i \in C'} p_i x_i \right) \quad (3)$$

Subject to

$$\sum_{i \in C'} s_i x_i \leq S$$

$$x_i \in \{0,1\}, \quad \forall i \in C'$$

Where:

- C' is the set of candidate indexes remaining in the *Candidate Pool* at the start of the knapsack algorithm.
- p_i is the performance gain of index i .
- s_i is the storage cost of index i .
- S is the total available storage budget.
- x_i is a binary decision variable (1 if the index is selected, 0 otherwise).

The system prioritizes indexes based on their performance-per-byte ratio:

$$\frac{p_i}{s_i} \quad (4)$$

Using a greedy approximation, the system determines the optimal set of indexes to retain, ensuring the best possible query performance within the given storage constraints. After removing indexes from C' that cause workload regression or have minimal value, and then applying the knapsack algorithm to select the highest value remaining candidate indexes, the final recommendation C^* is produced, known as the *Winners Pool*.

This approach provides a systematic, automated method for optimizing database indexing based on workload characteristics and

storage constraints. By leveraging query optimizer feedback and iteratively refining index selection, the system ensures performance gains without unnecessary index bloat.

3.6 Index candidate design

The processing of designing new indexes for the *Candidate Pool* begins with parsing each query in the workload using the Calcite parser. Calcite builds an abstract syntax tree (AST) that represents the hierarchical structure of the SQL query. The AST can then be searched to produce the list of columns in each table referenced by the query, and what indexable purpose they were used for, such as equality, in-list, inequality, between, group by, order by, distinct, join, etc. Due to the strict limitations on MySQL that constrain secondary indexes to a maximum of 64 per table, we use an approach for exploring the space that caps the number of candidate indexes. Since the number of combinations of columns explodes exponentially, it was necessary to apply heuristics to ensure good coverage of the search space without exceeding this narrow limit. We use an approach similar to Valentin's SAEFIS model [29] that organizes columns of interest (columns that may be beneficial to index) into categories such as equality, inequality and range, order by, distinct, etc. Candidates are designed by selecting all strict subsets of these categories in priority sequence, i.e. all single column combinations, followed by all two column combinations, etc. until the constraint is reached. The categories are defined as follows:

- 1) EQ: columns used in equality predicates, and in-lists
- 2) ORDER: columns used in order by
- 3) JOIN: columns used in joins
- 4) DISTINCT: columns used in distinct and count distinct
- 5) RANGE: columns used in inequalities, range (between), and like
- 6) GROUP: columns used in group by

If a column appears in multiple categories, it is retained exclusively in the first category and removed from the others in order to limit the search space.

To illustrate the process, consider a table with columns $c1$ through $c5$ that appear as columns of interest within a workload as follows:

- 1) EQ: $c2, c4$
- 2) ORDER: $c5$
- 3) JOIN: $c1$
- 4) DISTINCT: none
- 5) RANGE: $c2, c3$
- 6) GROUP: $c4$

Columns in the EQ category are sorted into decreasing order of cardinality, which is determined by sampling. The columns are then serialized into a list, without duplication. This starts with the EQ category and proceeds in sequence to GROUP. This results in the following list for our example:

{ $c2, c4, c5, c1, c3$ }

Note that $c2$ and $c4$ appear in multiple categories but they are placed in the list once without duplication. Next, all strict subsets of the list are created, defining possible indexes, starting with single

column indexes, and proceeding to two and then three columns, etc. This produces the following 30 candidate indexes for our example:

{c2}	{c4, c1}	{c2, c1, c3}
{c4}	{c4, c3}	{c4, c5, c1}
{c5}	{c5, c1}	{c4, c5, c3}
{c1}	{c5, c3}	{c4, c1, c3}
{c3}	{c1, c3}	{c5, c1, c3}
{c2, c4}	{c2, c4, c5}	{c2, c4, c5, c1}
{c2, c5}	{c2, c4, c1}	{c2, c4, c5, c3}
{c2, c1}	{c2, c4, c3}	{c2, c4, c1, c3}
{c2, c3}	{c2, c5, c1}	{c2, c5, c1, c3}
{c4, c5}	{c2, c5, c3}	{c4, c5, c1, c3}

All of these indexes are considered during the evaluation, up to the limit of the secondary indexes constraint (64 for MySQL and 256 for PostgreSQL).

Our solution currently excludes reference columns from the select list when designing indexes, focusing only on columns used in predicates, grouping, and sorting. While covering indexes that include all query columns can provide secondary benefits, the most significant performance gains come from indexing filtering and sorting columns. This approach substantially reduces the search space and enables efficient batch creation of candidate index definitions for optimizer evaluation. We may relax this in future versions of SQL:Trek.

3.7 Denoising index evaluation

After selecting initial candidates using what-if analysis that leverages that SQL compiler’s cost model, we evaluate the relative benefit of indexes by executing them on a simulation environment. Unlike the cost model, which is purely mathematical, the simulation environment is a real functional system subject to run-to-run variance. We found the variance to be significant especially for short-running queries where fractions of a second may represent a high percentage of the total query runtime. This is important for our environment at Airbnb, where many of the SQL workloads in our online service path are OLTP workloads. To improve the stability of the evaluation, we ran all short running queries ($\leq 2s$) 9 times and long running queries ($> 2s$) 5 times, using the median response time as representative. Second, we rounded all query execution times to the nearest $1/20^{\text{th}}$ of a millisecond.

3.8 Additional features

We implemented the following options on the utility that have been helpful:

- 1) **Simulation reuse.** Since populating the simulation database is the single most time-consuming step in the process (see section 4.3), we created an option to reuse the simulation database from prior runs, which allows experiments to be run on multiple query sets in a fraction of the time.
- 2) **Sampling rate.** The sampling rate, as well as a max sample size constraint, can be defined as input parameters.
- 3) **Storage constraint.** The storage constraint for indexes can be defined on input as a % of data size.
- 4) **Query frequency.** The frequency of query execution is optionally modeled, so that the benefit of an index is scaled by the number of times its associated query runs. For example, an index that speeds up query #1 by 0.5ms and is executed 10 times in the workload is considered equally valuable as an index that accelerates query #2 by 5ms but is executed only once.
- 5) **Workload analysis.** SQL:Trek also provides a workload analysis mode that reports on the characteristics of the database under study, including size, top tables, DDL features such as number and size of tables and indexes. It further details the size and cardinality of the largest tables, as well as the use of specific features such as autoincrement columns, enums, BLOBs, UDFs, and triggers. SQL query patterns are reported such as the frequency of GROUP BY, joins, distinct, scalar functions, UNION, etc. and many other attributes. This has been useful in helping us understand the usage patterns of workloads across the company, as well as in detecting anti-patterns in our SQL workloads. For example, from this feature we were able to detect several cases where applications were pushing complex business logic or very large joins into SQL that are anti-patterns for our OLTP applications at Airbnb.

4 EXPERIMENTAL RESULTS

Here we present our results on 120 databases, each comprising several queries and organized into three suites, as summarized in Table 1. Each database represents a distinct workload.

Databases in the *Product Serving* group are used by the Airbnb site directly. They are in the code path of the online user experience. These are the most sensitive and highly tuned databases in the company. Because of Airbnb’s business as a consumer-consumer workflow, many queries in these workloads are simple point lookups and range queries. The rate of complex SQL expressions involving grouping, joins, distinct, case, etc. is much lower than the other workload categories. Less than 1/3 of workloads in this category have a single query with any of join, group by or distinct.

Databases in the *Internal Use* category are production databases at Airbnb outside the product workflow. These databases are not accessed by the online user experience. On average these workloads have more complex SQL, with a higher rate of complex language elements. Half of the workloads in this category have queries with join, group by or distinct.

Finally, the *SQL:Trek Test DB* is a test suite of 18 queries hand-crafted on top of a TPCC schema [28]. These queries are designed to cover a broad range of the query patterns across the company. This represents a more varied query set than any individual production query workload typically expresses. Queries in this set range from simple point lookups and in-lists, to more complex queries with joins, unions, grouping, subqueries, scalar functions, date-time math, etc.

Table 1: Three categories of test workloads.

Category	Number of databases	Number of queries
Product Serving DBs	47	1728
Internal Use DBs	72	958
SQL:Trek Test DB	1	18

4.1 Initial testing with the SQL:Trek Test DB

Initial testing and refinement of SQL:Trek was performed using BenchBase [9] with standard TPCC queries. We then replaced the TPCC queries with increasingly complex queries over time to ensure the index recommendation analysis could handle additional SQL language complexity, including in-lists, inequality predicates, range predicates (i.e. between), like, joins, unions, subselects, scalar functions, grouping and aggregation, distinct and ordering. The system under test included a 10GB TPCC database created using the BenchBase tooling, and Macbook Pro with 64GB RAM and M1 CPU.

Figure 2 shows the query performance improvement in the simulation environment. However, because the simulation is based on a sample of data, the benefits are generally higher when applied to the production environment. When we compared the estimated benefits of new indexes that SQL:Trek produces, which are based on evaluations run against the simulation database, against the same indexes and queries run against the full source database, we found the benefit were generally equivalent or greater on the full database environment. We found no cases of bad indexes being recommended. SQL:Trek pruned 11% of indexes that were identified in the what-if phase of the processing, meaning that indexes that appeared beneficial to the SQL cost model were found during the evaluation on the simulation database to be either not beneficial or to cause a workload regression. This is similar to the false-positive rate reported by Das [8].

After reporting the performance improvement seen in the simulation, SQL:Trek lists the CREATE INDEX DDL for the recommended indexes. For the example in Figure 2, this included 8 new indexes.

The performance of the queries on the full database, before and after applying the recommended indexes, is shown in Table 2. The absolute improvement of some queries is many times higher than what was achieved in the simulation run, usually because a scan became a point lookup; the former costs much more on the full database, while the latter is near constant time. The total execution time for all queries when applying the new index design improved by a factor of 6.6x. As expected, the recommendations from SQL:Trek did not provide accurate predictions of performance improvement, but were very successful in predicting which indexes would be highly beneficial.

Speed Test Results (rounded to 1/20th of a millisecond)			
Query No.	Original Speed	New Speed	Relative Speedup
1	0.05ms	0.05ms	1.00x
2	43.60ms	0.10ms	436.00x
3	95.80ms	0.50ms	191.60x
4	22.45ms	0.55ms	40.82x
5	0.15ms	0.15ms	1.00x
6	0.15ms	0.15ms	1.00x
7	0.55ms	0.55ms	1.00x
8	29.90ms	0.20ms	149.50x
9	42.40ms	0.15ms	282.67x
10	0.40ms	0.40ms	1.00x
11	0.35ms	0.35ms	1.00x
12	34.20ms	34.20ms	1.00x
13	22.75ms	4.00ms	5.69x
14	50.20ms	0.10ms	502.00x
15	0.95ms	0.10ms	9.50x
16	270.05ms	0.10ms	2700.50x
17	49.65ms	49.65ms	1.00x
18	40.65ms	40.65ms	1.00x

Figure 2: Results from a SQL:Trek test run.

Table 2: Individual query improvement for the SQL:Trek Test DB.

Query	Baseline (ms)	With new indexes (ms)	Speedup
1	0.05	0.05	1.0x
2	1343.80	0.10	13438.0x
3	2411.65	27.75	86.9x
4	497.25	7.55	65.9x
5	0.20	0.05	4.0x
6	0.15	0.05	3.0x
7	0.50	0.45	1.1x
8	1044.60	0.55	1899.3x
9	1309.75	0.05	26195.0x
10	0.40	0.25	1.6x
11	0.40	0.25	1.6x
12	1130.30	1125.55	1.0x
13	516.40	88.30	5.8x
14	1494.00	0.20	7470.0x
15	1.00	0.15	6.7x
16	5775.35	0.10	57753.5x
17	41.40	40.90	1.0x
18	1265.30	1256.60	1.0x

4.2 Workload benefit on production workloads

We evaluated SQL:Trek with data from 119 production databases, all MySQL 8.0 runtimes. The systems under test are summarized in Table 3.

For each workload under study, we report the improvement in execution time as the ratio of the sum of all query execution times before new indexes to the sum of all query execution times after new indexes were added. Each unique query in the workload is counted once for this assessment. For example, if the sum of all query execution times before adding new indexes was 3s, and the sum after adding new indexes was 1.5s, the improvement is shown as 2x.

Table 3: Systems under test.

System	Configuration
Product Serving DBs	DB Count: 47 Sizes: 10 x 0-100GB 7 x 100-1000GB 17 x 1,000-10,000GB 13 x >10,000GB Database runtime: MySQL 8.0
Internal Use DBs	DB Count: 72 Sizes: 51 x 0-100GB 16 x 100-1000GB 3 x 1000-10000GB 2 x >10,000GB Database runtime: MySQL 8.0
Simulation databases	Hardware: AWS db.r6i.xlarge - db.r6i.16xlarge 2-64 vCPUs 16-512GB RAM. Database runtime: MySQL 8.0

Results for databases in the *Product Serving* group are shown in Figure 3. As expected, these databases are among the best tuned in the company. Of the 47 databases studied, SQL:Trek identified 6 databases that benefited from additional indexing. One of these improved by more than 50,000x. The average workload improvement, over 8,000x, is skewed by the single large result. The median gain was 16.4x. Notably, in workload 2, which is comprised of 238 queries, a single query had a severe regression due to a compiler execution plan difference between the simulation environment and the full production data set.

Results for databases in the *Internal Use* group are shown in Figure 4. Of the 72 workloads under study, 28 benefited from new indexes. The average workload improvement was 54.6x and a median gain of 1.6x. Here as well, a single query in one workload experienced a large regression. In this case there was no execution plan change; the plan selected by the optimizer on both the simulation and the full production data was identical but performed poorly on the latter.

Even in workloads with overall performance near unity, there were several instances of individual queries improving dramatically—by two and three orders of magnitude. However, the total workload time was dominated by a few long-running queries. In such cases, the recommended indexes may still be particularly valuable to the application.

4.3 Execution time of the index recommender

Our goal for the index recommender was to generally complete the entire process of recommending indexes for a workload in less than 60 minutes. This is fast enough that it would allow us to run the evaluation for hundreds of databases several times per week on a

single evaluation server. Across the 119 production databases we studied (47 *Product Serving* and 72 *Internal Use*), SQL:Trek was able to complete the analysis with an average completion time of 16.1 minutes. A summary of the execution times for each design step, for a workload near the average duration time, is shown in Table 4.

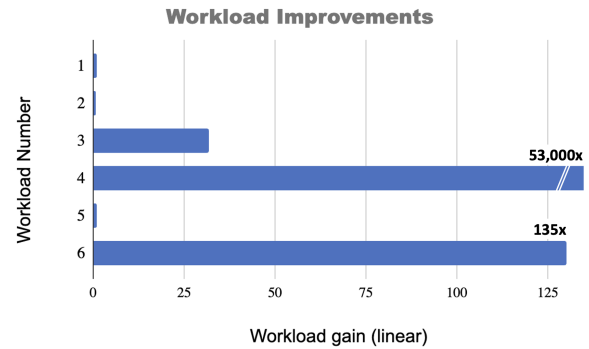


Figure 3: Product Serving DBs - Improved query execution time by workload.

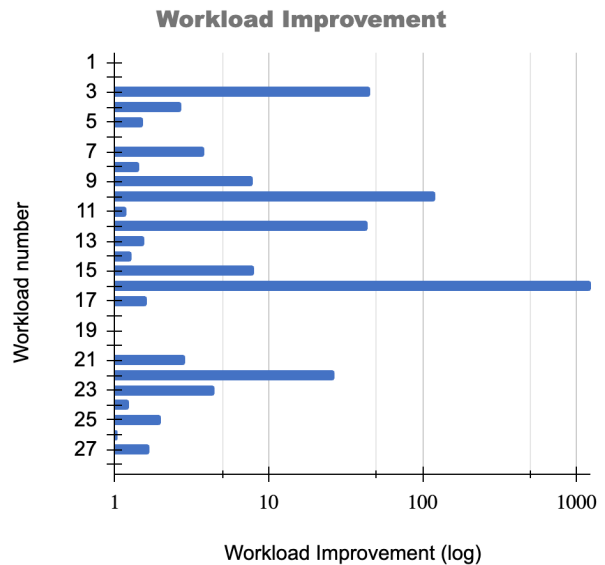


Figure 4: Internal Use DBs - Improved query execution time by workload.

The execution time for SQL:Trek is usually dominated by two steps, “Data sampling” and “Creating candidate indexes”. The duration of each phase of the processing varies based on the number of tables under study, the data volume and the number and complexity of queries in the workload. Because the simulation database is based on a sample and is size constrained, candidate index creation was generally quite fast. Similarly, query evaluation was usually fast, with most executing in less than 1ms and a small number running for 1–6s. This efficiency is driven in large part by the small size of the simulation database.

Table 4: Time spent in the SQL:Trek index recommender by step

Step	Execution time (s)
SQL workload analysis	0.3
Cloning the database schema	0.7
Candidate index design	0.2
Data sampling	448.5
Establish performance baseline	7.0
Creating candidate indexes	461.1
Index eval w. EXPLAIN	1.1
Filtering ineffective indexes	7.8
Performance eval in simulation	1.4
Iterative index refinement	4.7
Optimize index selection for storage constraints	0.3

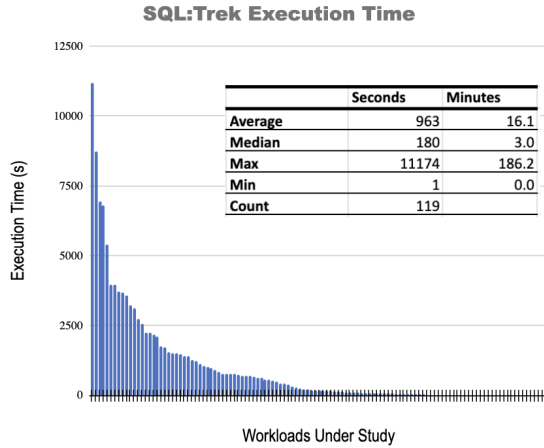


Figure 5: Execution time for the index recommender.

The distribution time of 119 index recommender runs at Airbnb, drawn from both the *Product Serving* and *Internal Use* groups sorted by execution time is shown in Figure 5. Across these 119 production databases, we see a median run time duration of 3 min, an average duration of just 16.1 min, and a maximum run time of 186.2 min. Both the median and average are well below the 60 min target, with 7.6% of databases exceeding it.

4.4 Rate of false positive recommendations

A key design objective for SQL:Trek was to minimize the rate of recommending indexes that have net-negative impact on a workload. We examined the rate of detecting and removing ineffective indexes on both the *Internal Use* and *SQL:Trek Test* workloads that were recommended by the SQL compiler. In the *Internal Use* workload, 48% of indexes selected by the database SQL optimizer (step 7 of the design) were found to be ineffective or negative, and SQL:Trek dropped these from consideration (step 10 of the design). On *SQL:Trek Test* the number was 11%. After applying the final index recommendations, we found the rate of query regression on *Internal Use* workloads that had index recommendations was just 0.1%, and on the *SQL:Trek Test* it was

0%. The percentage of negative indexes being recommended was 3.6% and 0% respectively. These are early tests, but strong indicators that executing queries on the simulation was effective at identifying indexes that appear useful from costing, but are not

4.5 Sampling rate sensitivity analysis

We examined the impact of varying sampling rates on 32 databases from the *Internal Use DB* category that were candidates for new index recommendations. This included all 28 databases with index recommendations in section 4.2, as well as four additional workloads. These workloads comprised approximately 700 queries. We tested sampling rates of 1%, 5%, 10%, 20%, and 40%, scaling both the sampling rate and the per-table row cap proportionately. To minimize noise in our measurements, we applied strict regression criteria: queries were classified as regressed only when both the absolute regression exceeded 0.2ms and the relative regression exceeded 10%. Index recommendations were classified as negative when they provided net-negative benefit to the overall workload and caused at least one query to meet our regression criteria.

As sample size increased, the execution time of the recommender increased roughly linearly. This is expected, since the time is largely dominated by populating the simulation database with the data sample and candidate index creation—both of which are proportional to data size.

The set of recommended indexes varied modestly across the tests with a coefficient of variation of 6.47% in the number of recommended indexes. Overall workload improvement trended higher at higher sampling rates, from 45x up to 58x as expected (see Figure 6). Median workload improvement was largely stable, remaining steady between 1.3x-1.5x for all sampling rates.

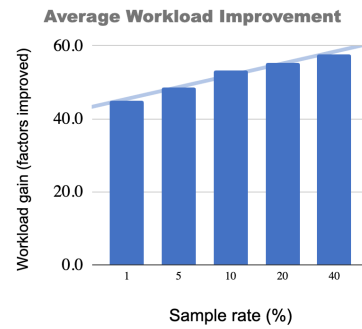


Figure 6: Impact of sampling rate.

The total number of false positives in the final recommendation, i.e. negative indexes that SQL:Trek eventually recommended because they demonstrated benefit in the simulation, but which in fact were damaging when applied to the full data, remained steady for all sample sizes. Specifically, the same negative indexes were recommended at each sampling rate, representing a false positive recommendation rate between 3.2% and 3.8%.

The rate at which the SQL compiler selected negative (damaging the workload) and/or ineffective (not significantly helpful) indexes remained quite significant, as measured in the simulation. The average for these across sampling rates was 47.2% with a coefficient

of variation of 6.06%. There was no clear trend for how this varied with sample rate.

5 FUTURE WORK

This paper presents our initial work on SQL:Trek, a utility for automated index design and workload analysis. The system is actively being developed at Airbnb, and we identify several key areas for future improvement:

- 1) **Virtual statistics.** Many database engines provide the ability to override the table statistics generated by ANALYZE. Where this is available it would be highly beneficial to override the statistics in the simulation with the statistics that exist in the source database. This would further reduce the risk of false positive index recommendations.
- 2) **Query Weighting** – Incorporate query execution frequency analysis to better capture workload characteristics and prioritize index recommendations accordingly.
- 3) **Scalability of What-If Analysis** – Extend multi-pass evaluation techniques to support more than 64 candidate indexes per table, overcoming current system limitations on MySQL.
- 4) **Selective Inclusion of Reference Columns** – Our initial approach excludes reference columns based on the principle that covering indexes often lead to excessive specialization. However, in specific cases, selectively including reference columns could be beneficial, provided they generalize well and do not lead to overfitting indexes to a narrow query pattern.
- 5) **Identification and Removal of Unused Indexes** – Develop mechanisms to detect indexes that are no longer utilized by the workload and recommend their removal. These may include legacy indexes created for outdated queries or indexes that remain unused due to current query optimizer cost estimations.
- 6) **Impact Assessment on Write Performance** – Model the effects of newly recommended indexes on INSERT, UPDATE, and DELETE operations to balance query performance with write overhead.
- 7) **Simulation Sampling Rate Optimization** – Our initial investigation utilized a 5% sampling rate, with minimum and maximum row constraints set at 20K and 1M rows, respectively. These parameters have proven effective for the current database configurations under study. However, future efforts will focus on determining the minimal sample size that can be employed without degrading the quality of the recommendation output. Preliminary observations indicate that reducing the sampling rate significantly accelerates recommendation computation time, exhibiting a near-linear correlation. Further exploration of this relationship will aim to optimize performance while maintaining recommendation accuracy.

These enhancements will refine SQL:Trek's recommendations, making it more effective in real-world database environments.

6 CONCLUSIONS

In this paper, we presented SQL:Trek, an index recommendation tool for relational databases that combines what-if analysis using the SQL compiler's cost model with query execution on sampled datasets. SQL:Trek typically delivers high-quality index recommendations in under 20 minutes while minimizing false positives. This approach offers substantial advantages over prior methods by achieving rapid analysis without requiring modifications to the underlying database system.

Implemented as an external tool, SQL:Trek is compatible with MySQL- and PostgreSQL-based databases and can be extended to other platforms including Oracle [24], DB2 [14], and SQL Server [21]. Our evaluation across 120 databases demonstrated significant performance improvements, with query workload execution times improved by substantial factors in multiple cases. These results demonstrate SQL:Trek's effectiveness as a practical utility for database index optimization.

ACKNOWLEDGMENTS

This work would not be possible without the support of several colleagues including Dave Nagle, Abhishek Parmar, Zheng Liu, and Erluo Li. All trademarks are the properties of their respective owners. Any use of these are for identification purposes only and do not imply sponsorship or endorsement.

REFERENCES

- [1] Amazon Web Services. (n.d.). Amazon Aurora MySQL reference. Retrieved March 14, 2025, from <https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-mysql-reference.html>
- [2] Apache Software Foundation. 2023. Apache Calcite: A Dynamic Data Management Framework. Apache Calcite Official Website. Available online: <https://calcite.apache.org/>
- [3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004)*, Toronto, Canada, August 31 – September 3, 2004. Morgan Kaufmann, 1110–1121. <https://doi.org/10.1016/B978-012088469-8.50097-8>
- [4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000)*, Cairo, Egypt, September 10–14, 2000. VLDB Endowment, 496–505. <http://www.vldb.org/conf/2000/P496.pdf>
- [5] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*, Athens, Greece, August 25–29, 1997. Morgan Kaufmann, 146–155.
- [6] Surajit Chaudhuri and Vivek Narasayya. 2020. Database Tuning Advisor for Microsoft SQL Server, *Microsoft Research* <https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/>.
- [7] Cockroach Labs. 2024. CockroachDB: The Distributed SQL Database. <https://www.cockroachlabs.com/>
- [8] Sudipto Das, Miroslav Grbic, Igor Ilıc, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*, Amsterdam, The Netherlands, June 30–July 5, 2019, 666–679. ACM. <https://doi.org/10.1145/3299869.3314035>

- [9] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288. <http://www.vldb.org/pvldb/vol7/p277-difallah.pdf> <https://doi.org/10.14778/2732240.2732246>
- [10] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019)*, Amsterdam, The Netherlands, June 30 – July 5, 2019, 1241–1258. <https://doi.org/10.1145/3299869.3324957>
- [11] EnterpriseDB Corporation. 2024. EDB Postgres Advanced Server. <https://www.enterprisedb.com/>
- [12] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.* 13, 1 (1988), 91–128. <https://doi.org/10.1145/42201.42205>
- [13] Google Cloud. 2024. AlloyDB for PostgreSQL: Fully managed PostgreSQL-compatible database service. <https://cloud.google.com/alloydb>
- [14] IBM Corporation. 2024. IBM Db2 for Linux, UNIX and Windows. <https://www.ibm.com/products/db2>
- [15] JSQLParser. JSQLParser: A SQL statement parser. 2023. Retrieved May 26, 2025, from <https://github.com/JSQLParser/JSQLParser>
- [16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. <http://www.vldb.org/pvldb/vol9/p204-leis.pdf> <https://doi.org/10.14778/2850583.2850594>
- [17] Gabriel Paludo Licks and Felipe Meneguzzi. 2020. Automated Database Indexing using Model-free Reinforcement Learning. *CoRR*, abs/2007.14244. <https://arxiv.org/abs/2007.14244>
- [18] Sam Lightstone and Bishwaranjan Bhattacharjee. 2004. Automated design of multidimensional clustering tables for relational databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004)*, Toronto, Canada, August 31 – September 3, 2004. Morgan Kaufmann, 1170–1181. <https://doi.org/10.1016/B978-012088469-8.50102-9>
- [19] Guy Lohman. 2014. Is Query Optimization a “Solved” Problem? <http://wp.sigmod.org/?p=1075>
- [20] MariaDB Corporation. 2024. MariaDB Server: The Open Source Relational Database. <https://mariadb.org/>
- [21] Microsoft Corporation. 2024. Microsoft SQL Server. <https://www.microsoft.com/en-us/sql-server/>
- [22] Priscilla Neuhaus, Julia Couto, Jonatas Wehrmann, Duncan Dubugras Alcoba Ruiz, and Felipe Meneguzzi. 2019. GADIS: A Genetic Algorithm for Database Index Selection. In *Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering (SEKE 2019)*, Lisbon, Portugal, July 10–12, 2019, 39–54. <https://doi.org/10.18293/SEKE2019-135>
- [23] Oracle Corporation, MySQL 8.0 Reference Manual, MySQL 8.0.36, Jan. 2025. [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/>
- [24] Oracle Corporation, Oracle® Database Documentation, [Online]. Available: <https://docs.oracle.com/en/database/oracle/oracle-database/>
- [25] PingCAP. 2024. TiDB: An open-source, cloud-native, distributed, MySQL-Compatible database for elastic scale and real-time analytics. <https://www.pingcap.com/tidb/>
- [26] The PostgreSQL Global Development Group. 2024. PostgreSQL: The World’s Most Advanced Open Source Relational Database. <https://www.postgresql.org/>
- [27] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE 2019)*, Macao, China, April 8–11, 2019, 1238–1249. IEEE. <https://doi.org/10.1109/ICDE.2019.00113>
- [28] Transaction Processing Performance Council (TPC), TPC Benchmark C Standard Specification, TPC, San Francisco, CA, USA, Available: <http://www.tpc.org/tpcc/>
- [29] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, San Diego, CA, USA, February 28 – March 3, 2000. IEEE Computer Society, 101–110. <https://doi.org/10.1109/ICDE.2000.839397>
- [30] Ritwik Yadav, Satyanarayana R. Valluri, and Mohamed Zait. 2023. AIM: A practical approach to automated index management for SQL databases. In *Proceedings of the 39th IEEE International Conference on Data Engineering (ICDE 2023)*, Anaheim, CA, USA, April 3–7, 2023, 3349–3362. IEEE. <https://doi.org/10.1109/ICDE55515.2023.00257>
- [31] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004)*, Toronto, Canada, August 31 – September 3, 2004. *VLDB Endowment*, 1087–1097. <http://www.vldb.org/conf/2004/IND4P1.PDF>. <https://doi.org/10.1016/B978-012088469-8.50095-4>
- [32] Benyuan Zou, Jinguo You, Quankun Wang, Xinxian Wen, and Lianyin Jia. 2022. Survey on Learnable Databases: A Machine Learning Perspective. *Big Data Research* 27 (2022), 100304. <https://doi.org/10.1016/j.bdr.2021.100304>