AIM: A practical approach to automated index management for SQL databases

Ritwik Yadav
Database Engineering
Meta Platforms, Inc.
Menlo Park, USA
ritwikyadav@fb.com

Satyanarayana R. Valluri
Database Engineering
Meta Platforms, Inc.
Menlo Park, USA
satyav@fb.com

Mohamed Zaït
Database Engineering
Meta Platforms, Inc.
Menlo Park, USA
mzait@fb.com

***Abstract*—This paper describes AIM (Automatic Index Manager), a configurable index management system, which identifies impactful secondary indexes for SQL databases to efficiently use available resources such as CPU, I/O and storage. It has been validated on thousands of databases which support production systems. With AIM, the physical design of the database adapts itself to the changes in the workload.**

We lay out the end to end design of AIM while calling out the guarantees and tradeoffs associated with our design choices. Some of the salient features of AIM include fast convergence even while recommending wide composite indexes, reduced reliance on the query optimizer and a "no regression" guarantee for production workloads. Each index recommendation from AIM is accompanied with a metrics driven explanation, making it easier to verify machine driven changes.

AIM is one of the few industrial strength index recommendation engines that is deployed on production databases at a large scale. The experimental results show that AIM is quick in identifying the most effective indexes and the resulting physical design is close to optimal.

***Index Terms*—automatic indexing, self managing databases, workload adaptivity**

I. Introduction

A. Motivation

A significant challenge in managing databases is the maintenance of efficient physical structures to facilitate data access for evolving workloads. Historically, this was done manually by database administrators with significant domain expertise. However, the emergence of cloud managed databases and pay-as-you-go models has resulted in automation of database tuning processes. The resulting decline in human involvement has made physical design tuning significantly more efficient at scale.

A carefully selected set of secondary indexes can have a defining impact on the applications powered by relational databases. The tradeoff involves using more storage in exchange for reduced CPU and I/O utilization. Reduced I/O during query execution vastly improves its latency. Decades of research [1]–[8] has explored a variety of ways to navigate and choose from the space of available indexes. However, very few industrial strength solutions exist today that have been validated on a variety of production workloads and are capable of identifying wide secondary indexes in a *reasonable* amount of time.

B. Challenge

Selection of the optimal set of indexes for a given workload is an NP-hard problem [9]. In fact, optimal selection of indexes for facilitating a single join query is a complex problem in itself since determining the optimal join order is an NP-hard problem as well [10].

Most state-of-the-art solutions [11], [12] rely on utilizing the query optimizer to evaluate the cost of executing queries with hypothetical configurations of indexes. The number of these configurations explodes for real-life production workloads placing significant restraints on the solution space [13]. These restraints limit the ability of modern algorithms to identify wide composite indexes *in time* and prevent severe performance degradation for input workloads. Prior work [14] also suggests that there is significant utility in optimizing the evaluation of these hypothetical configurations.

C. Approach & Contribution

In this paper, we approach automatic index management from a practical viewpoint. Our design falls back on first priniciples of the utility of indexes. The benefit of using an index always results from reduced I/O operations. It can be attributed to efficient selection of records, reading only relevant attributes of selected records and / or by eliminating the need to sort selected records. The distinguishing contributions of our work are as follows:

- Utilization of the *structural properties* of individual queries to significantly restrict the search space of candidate indexes. The query structures are used to generate targeted index candidates rather than attempt to explore the entire space.
- Efficient exploration of *wide multi-column indexes* becomes more feasible. Most modern techniques have to restrict the index width [13] in order to prevent the number of index configurations from exploding.
- To the best of our knowledge, AIM is one of the few widely deployed production algorithms which treats *complex join queries* systematically.
- *Reduced reliance* on the database query optimizer. Unlike its predecessors, AIM does not query the optimizer for data distribution statistics every time it has to append a

new column to a multi-column candidate index. This is reflected in AIM's runtime which is orders of magnitude smaller than other modern algorithms.
- Our experimental study shows that AIM can both fill the gap left by manual tuning and build secondary indexes from scratch with comparable (or better) performance. It can also detect and drop (parts of) unused indexes.

D. Structure of the paper

Section II defines the index tuning problem which is followed by a detailed description of the general algorithm and system architecture used for solving the problem in section III and section IV. The algorithm described is general enough to be implemented by any SQL database and its salient features are called out in section V. Conclusions drawn from experiments conducted on production workloads and synthetic benchmarks are presented in section VI, which demonstrate the impact of the design choices made when implementing AIM. It also compares AIM against other modern algorithms and highlights the significantly better runtime of AIM. External components that play a crucial role in AIM's architecture are described in section VII. Section VIII calls out the lessons learned in the process of implementing AIM while section IX draws a comparison with existing work and lists ideas for future work.

II. Problem Definition

The index tuning problem focuses on selecting an optimal configuration $\mathcal{X}$ for a relational database D such that the execution cost of the workload running on it ($\mathcal{W}$) can be minimized under certain constraints. For the scope of this problem, the configuration $\mathcal{X}$ is comprised of a set of secondary indexes, each of which is materialized on one of the tables present in D. The workload $\mathcal{W}$ is comprised of queries running on D. Each query, q, has an associated weight w_q. w_q can be based on query q's execution frequency, its overall CPU consumption or a manually specified importance measure provided by query authors.

A. Static Index Tuning Problem

The static index tuning problem (also referred to as bootstrapping) involves finding the optimal subset of possible secondary indexes that minimize the computational cost of serving a constant workload. It is formally defined as follows.

Given a constant workload $\mathcal{W}$ (consisting of queries $q \in \mathcal{W}$), a set of all possible indexes $\mathcal{U}$ and a storage budget B, find $\mathcal{X}^*$ such that

$$\mathcal{X}^* = \operatorname*{argmin}_{\mathcal{X} \in 2^{\mathcal{U}}} \Big(\sum_{q \in \mathcal{W}} w_q cost(q, \mathcal{X}) \Big) \tag{1}$$

where w_q is the weight associated with query q and $cost(q, \mathcal{X})$ is the execution cost of q with index configuration $\mathcal{X}$ such that $\mathcal{X}$ fits in B.

B. Continuous Index Tuning Problem

The continuous index tuning problem is a slightly different variant of the bootstrapping problem. In fact, this variant is closer to the real world since most developers deploy their databases with an initial set of indexes. This initial set might not always be the most efficient choice but it serves as a starting point to improve upon. The problem statement is as follows.

Given a workload $\mathcal{W}$ (consisting of queries $q \in \mathcal{W}$), a set of pre-existing indexes $\mathcal{C}$ and a storage budget B, find $\mathcal{C}'$ such that

$$\sum_{q \in \mathcal{W}} w_q cost(q, \mathcal{C}') \leq (1 + \lambda_1) \sum_{q \in \mathcal{W}} w_q cost(q, \mathcal{X}^*) \tag{2}$$

$$\exists q \in \mathcal{W} \bullet w_q cost(q, \mathcal{C}') \leq (1 - \lambda_2)(w_q cost(q, \mathcal{C})) \tag{3}$$

$$\forall q \in \mathcal{W} \bullet w_q cost(q, \mathcal{C}') \leq (1 + \lambda_3)(w_q cost(q, \mathcal{C})) \tag{4}$$

where $0 \leq \lambda_1, \lambda_2, \lambda_3 < 1$ are parameters that can be set to achieve the desired goals. Equation 2 keeps the overall cost in check with respect to the configuration $\mathcal{X}*$ detected while bootstrapping, equation 3 mandates the improvement in performance of at least one query and equation 4 prevents significant regressions in the performance of individual queries.

The reason why databases need to undergo continuous tuning is because w_q and $cost(q, \mathcal{X})$, for a given query q and configuration $\mathcal{X}$, are functions of time as the workload and underlying data distributions change.

III. Algorithm

A. Terminology

A detailed description of the algorithms requires defining some frequently used concepts for ease of understanding.

1) Normalized query: A normalized (or parameterized) form of a query is obtained by replacing parameters with placeholders. This mechanism is used to group together queries with similar structure. Note the ? used to represent a parameter in the following example.

```
SELECT id, name FROM students WHERE score > ?
```

2) Discarded data ratio (ddr): For every execution of a query, we compute the amount of data that was read during its execution and the amount of data that was returned by it. The disparity between these two metrics per execution of a normalized query is computed as the ratio of data sent to data read averaged across executions of a normalized query.

3) Partial order of index columns: Potential index candidates are denoted as strict partial orders [15] of columns on a single table such as $(X_t, \prec)$ where X_t is the set of columns of table t and the relation $\prec$ denotes that a column precedes another in an index. Consider the following partial order where its ordered partitions are listed.

```
<{col1, col2}, {col3}, {col5, col6, col7}>
```

It represents all non-unique secondary indexes where columns $col1$ and $col2$ occupy the first two places (order immaterial), followed by column $col3$ in third place. $col3$ is, in turn, followed by any permutation of $col5, col6$, and $col7$ in the last three places.

4) Dataless indexes: Dataless indexes are secondary indexes that are created without actual data. They only have data distribution statistics and are used for estimating query execution costs by the optimizer[1]. Dataless indexes are similar in implementation to "what-if" indexes described by Chaudhuri and Narasayya [4].

5) Covering index: A covering index is a secondary index that constitutes all the columns of a table that are referenced in a specific query. Therefore, the clustered primary key (base table) need not be accessed in order to compute the result of the query. Covering indexes usually reduce the number of random disk seek operations.

B. Overview

The high-level flow of the AIM procedure is depicted in Figure 1. The algorithm can be thought of as having two distinct phases; both of which follow the same general outline presented in algorithm 1. Initially, if multiple queries are executing inefficiently, it is not easy to predict their relative frequency in steady state after adding the necessary supporting indexes. Therefore, AIM tries to add indexes of smaller width to support each inefficient query in the observed workload. In a subsequent phase, it identifies queries that might benefit from covering indexes. Usually, these queries execute extremely frequently to offset the extra storage overhead of creating wider indexes.

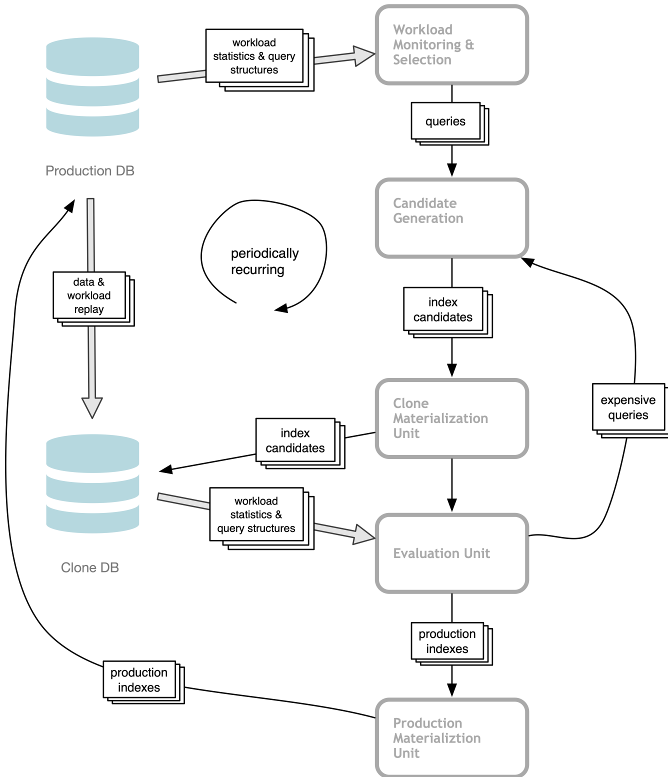


Fig. 1: AIM flowchart

[1]The estimated execution costs might be a bit off since index dives cannot be performed

---

Algorithm 1: Automatic Index Manager

Input: $database$: name of the database, j: join parameter
Output: $production_indexes$: Set of index to be created in production on $database$

1 $\mathcal{W} \leftarrow WorkloadSelection(database)$;
2 $candidates \leftarrow GenerateCandidates(\mathcal{W}, j)$;
3 Materialize $candidates$ on the clone $database$, ordering them in descending order of perceived benefits from optimizing queries in $\mathcal{W}$ until storage budget is exhausted;
4 $production_indexes \leftarrow RankSelectedIndexes(candidates)$

Line 1 of algorithm 1 denotes the representative workload selection step which identifies the set of queries that need tuning by monitoring their execution statistics. For each query that is identified by the previous step, a set of candidate indexes are generated (line 2). The join parameter j determines which tables (out of the many) present in a complex join query are exhaustively evaluated for all possible relative join orders (explained in subsection IV-C).

The set of candidate indexes is materialized on a clone of the database to determine if the query optimizer is able to use the indexes efficiently during execution (line 3). This is necessary to guarantee that none of the queries regress in production (more details in subsection VII-B). The set of production indexes is identified by ranking the usefulness of the candidate indexes (line 4) which are then materialized on the production database.

The algorithm works for both static and continuous index tuning. It can be run once to bootstrap the initial set of indexes when the database is created and periodically thereafter for continuous tuning to adapt to changes in the workload. The following sub-sections provide detailed description of each of the steps outlined in the high level algorithm.

C. Representative workload selection

The goal of this step is to identify the set of queries that require index tuning. The workload monitor analyzes execution statistics to identify queries that are not being executed efficiently. Query execution statistics include important information such as CPU cost, rows read, rows sent and number of executions corresponding to each normalized query. The utility of this data is in selecting / ordering queries with maximum expected benefits from adding potentially useful indexes.

Based on the query execution statistics, a representative sample of the workload is selected by identifying the most expensive queries that need tuning. The selection of the representative workload for a given database is fully automated and done periodically at the end of a configurable interval of time. The selection process takes three main attributes into

account for each query; execution frequency, average CPU consumption per execution and the discarded data ratio.

The threshold on the execution frequency is only used to weed out spurious executions of queries from ad hoc sources. The combination of average CPU consumption and discarded data ratio (ddr_{avg}) is used to assign an optimistic expected benefit (B) from optimizing a normalized query q executed with configuration $\mathcal{X}$.

$$B(q, \mathcal{X}, \Delta t) \coloneqq (1 - ddr_{avg}(q, \mathcal{X}, \Delta t)) \cdot cpu_{avg}(q, \mathcal{X}, \Delta t) \quad (5)$$

$cpu_{avg}(q, \mathcal{X}, \Delta t)$ represents the average CPU utilization by normalized query q over the time span Δt. It is measured in CPU seconds and includes cycles spent on CPU_IOWAIT events. Equation 5 assumes that all the I/O done by the query which isn't returned in the result set could have been avoided by proper index structures and thereby represents the maximum potential benefit from optimizing the query[2]. A threshold on B (e.g. 1/20 of a CPU core) is used to select the queries in the workload targeted for optimization.

D. Generate Candidate Indexes

The candidate generation process takes into account the structure of the query and is *efficiently* able to generate effective composite indexes. Statistics around data distribution are indirectly taken into account in our implementation via dataless indexes. The key lies in defining transformation functions from column usage metadata to potential index candidates represented by a partial order of columns. The basic transformation functions are more or less the same across most popular SQL database servers and this approach reduces the number of candidates considerably.

Algorithm 2: GenerateCandidates

```
   Input: W: workload, j: join parameter
   Output: IP: Set of candidate indexes
1  PO ← φ;
2  foreach Q ∈ W do
3  |  mode ← TryCoveringIndex(Q);
4  |  PO ← PO ∪
   |    GenerateCandidatesForSelection(Q, j, mode) ∪
   |    GenerateCandidatesForGroupBy(Q, j, mode) ∪
   |    GenerateCandidatesForOrderBy(Q, j, mode);
5  end
6  PO_final ← MergePartialOrders(PO);
7  return GenerateCandidateIndexPerPO(PO_final);
```

Algorithm 2 shows the high level algorithm for the generation of candidate indexes. The algorithm takes as input the workload and the join parameter j. For each query $\mathcal{Q}$ in the workload, the algorithm checks if a covering index would be beneficial for it (line 3). The $TryCoveringIndex$ procedure makes this determination based on the indexes utilized for the execution of $\mathcal{Q}$ in the current configuration. A covering index is tried if it is not possible to improve selectivity any further and the addition of a covering index is expected to significantly bring down the number of additional seeks from the primary key. Taking the example of Q4 in subsection IV-D, a covering index is not tried until an index with the prefix <col2, col3> is already being utilized. Furthermore, if such an index is already being used, the number of disk seeks should be high enough to offset the cost of paying the extra storage cost of including one more column $col1$ to the index. This threshold is high for fast storage media such as SSDs. Line 4 of the algorithm generates a set of partial orders of index columns by taking into consideration alternative query execution strategies. The execution strategy is based on the primary operation (out of selection, grouping or ordering) that is optimal for the query. The primary operation is the one that precedes all other operations. For instance, one execution strategy of query $\mathcal{Q}$ might involve sorting rows in group order before selection of rows, whereas another strategy might prefer selection before sorting in group order.

The set of partial orders generated in the previous steps are merged to get the final set of partial orders (line 6). The procedure is described in detail in subsection III-E. Corresponding to each partial order, one index candidate is generated by arbitrarily choosing a total ordering of index columns which satisfies the partial order. This is done by the $GenerateCandidateIndexPerPO$ procedure (line 7).

[2]This assumption is not always true for certain cases such as queries with a GROUP BY operation

E. Merge Partial Orders

We first define a function $MergeCandidatesPairwise$ which takes two strict partial orders $(P, \prec_P)$ and $(Q, \prec_Q)$ to produce another strict partial order $(R, \prec_R)$ when the prerequisite condition C_{merge} is met.

$$C_{merge} \coloneqq P \subseteq Q \bigwedge (\nexists\, a, b \in P : a \prec_P b \wedge b \prec_Q a)$$

$$(R, \prec_R) \coloneqq \begin{cases} (P, \prec_P) \bigoplus (Q, \prec_Q), & \text{if } C_{merge} \\ (\phi, \prec_R), & \text{otherwise} \end{cases}$$

P, Q and R are subsets of columns of the same table. $a \prec_P b$, $a \prec_Q b$ and $a \prec_R b$ denote relations; all of which semantically translate to the fact that column a precedes column b in the index represented by these partial orders. $\bigoplus$ denotes the ordinal sum [16] operator.

The function $MergePartialOrders$ (from algorithm 2) takes a set of partial orders of index columns ($\mathcal{PO}$) and recursively merges the constituent partial orders in pairs (using $MergeCandidatesPairwise$) until no new ones are produced. The following equation give a more formal definition.

$$\mathcal{PO}_{n+1} \coloneqq \{MergeCandidatesPairwise(X, Y) \mid X, Y \in PO_n\} \quad (6)$$

$MergePartialOrders(\mathcal{PO})$ starts with $\mathcal{PO}_0 \coloneqq \mathcal{PO}$ and returns $\mathcal{PO}_m$ using Equation 6 such that $\mathcal{PO}_m = \mathcal{PO}_{m+1}$. Intuitively, merging of the partial orders of index columns helps in picking the right order of index keys that will maximize the benefit of the index created. For example, consider the following two partial orders: <{col1, col2, col3}> & <{col2, col3}>.

The first partial order denotes that some query might benefit from the creation of an index with any permutation of $col1, col2$ and $col3$. The second partial order denotes that some query might benefit from the creation of an index with any permutation of $col2$ and $col3$. Merging the partial orders leads to creation of a new partial order: <{col2, col3}, {col1}>.

The merged partial order puts a constraint on the index columns' ordering to be limited to <col2, col3, col1> and <col3, col2, col1>. Either candidate satisfies the merged partial orders and can individually be beneficial to queries for which the base partial orders were merged.

F. Ranking and selection of candidates

The utility of an index is computed by summing up the benefits observed by individual queries that use it and discounting the write amplification resulting from index maintenance. The cost of a SQL query q can be decomposed into two main components expresssed as $cost(q, \mathcal{X}) = cost_r(q, \mathcal{X}) + \sum_{i \in \mathcal{X}} cost_u(q, i)$ where $cost_r(q, \mathcal{X})$ represents the cost of locating the records by using indexes in configuration $\mathcal{X}$ and $cost_u(q, i)$ represents the overhead of updating index i in $\mathcal{X}$. $cost_u(q, i)$ is non-zero only for DML statements. The logic for estimating (components of) costs is dependent on the storage engine [17] and can utilize the statistics offered by dataless indexes.

The two main ways in which indexes help queries are by reducing the amount of data that needs to be read and preventing repeated sorting operations. In order to quantify this benefit, every query q is treated in isolation at first. This gain (U_+) can be estimated by Equation 7 where $\mathcal{I}$ is the set of candidates generated to benefit query q. U_+ is distributed amongst the indexes (in $\mathcal{I}$) which are used during q's execution. The share $s_{i,q}$ of U_+ attributed to an index i is directly proportional to the reduction in I/O due to i (estimated by the optimizer).

$$U_+(q, \mathcal{I}) := \frac{cost(q, \phi) - cost(q, \mathcal{I})}{cost(q, \phi)} \cdot cpu_{avg}(q, \phi, \Delta t) \quad (7)$$

Similarly, the overhead of maintenance arises from index updates and storage space required. The overhead of index maintenance can be computed for an individual index i using Equation 8. Higher I/O operations lead to higher CPU utilization because the cpu_{avg} metric includes CPU_IOWAIT events.

$$u_-(i) := \sum_{q \in \mathcal{W}} \frac{cost_u(q, i)}{cost(q, \phi)} \cdot cpu_{avg}(q, \phi, \Delta t) \quad (8)$$

The overall utility of an index is represented by $u(i) = s_{i,q} \cdot U_+(q, \mathcal{I}) + u_-(i)$. The accounting for index interactions is limited to the merging of representative partial orders of the index candidates. When index candidates are merged, the benefits corresponding to individual queries gets added up and write amplification overhead is the same as that of the wider candidate being merged. Index selection can then be modeled as a knapsack problem [18] where index candidates are evaluated in the order of their overall utility per unit storage overhead while not violating the budget allocated for indexes. We implemented several other heuristics for a more complete accounting of index interactions but comparison of their relative effectiveness is deferred to future work.

IV. CANDIDATE GENERATION

Candidate generation takes into account the structure of the query and transforms structural metadata to partial orders representing index candidates. The structural metadata collected by the workload monitor includes information about operations corresponding to each column, edges in the table join graph and factors in the selection predicate. Examples of this metadata are listed in Table I. This information is particularly useful in generating candidate indexes for common SQL operations as is described in the following subsections.

A. Projection

The projection operation is used to select a subset of the attributes (columns) from a relation (table). Let us assume the existence of a table t_1 with five columns, namely, $col1, col2, col3, col4$ and $col5$. A very simple example demonstrating projection would be the following.

```
Q1: SELECT col2, col3 FROM t1 WHERE col5 < 2
```

In this query, instead of requesting all columns from t_1 for rows that satisfy the WHERE clause, only $col2$ and $col3$ are requested in the output. The following partial order represents the set of indexes that would optimize Q1 in the example above: <{col5}, {col2, col3}>.

The consideration of these secondary indexes which prevent lookups to the primary key can be useful when using slow storage media. It should also be noted that addition of *both* $col2$ and $col3$ at the end is necessary to avoid primary key lookups. Most greedy algorithms which add one column at a time may not be able to discover covering indexes for common queries.

B. Selection (Filter operation)

The WHERE clause may contain predicates of the following form connected by logical operators (e.g. AND, OR)

```
column_name op expression
```

These predicates which aid in the selection of rows from a single table instance are termed as filter operations. For each distinct table instance, we construct partial orders of columns that participate in filter operations. For e.g., a predicate like

```
E1: WHERE col1 = 5 AND col2 = 'ABC'
      AND col3 IN (5, 9, 11)
```

would result in the following partial order of columns: <{col1, col2, col3}>.

TABLE I: Query structure

Column Usage Metadata	Operation	FILTER, TABLE JOIN, GROUP BY, ORDER BY, PROJECTION etc.
	Operator	GREATER THAN, EQUAL, SORT ASCENDING etc.
Structural metadata	Edges in the table join graph, grouping of predicates in AND-OR chains etc.	

1) Complex AND-OR predicates: The selection predicate might employ complex AND-OR chains and generation of candidates may differ slightly depending on the factorization technique implemented by the database server. This technique is denoted by the $FactorizeIndexPredicates$ routine in algorithm 5. The algorithm employed by MySQL is simple but other database engines may use more complex techniques of factorization [19]. In our implementation, we simply use the disjunctive normal form (DNF) for complex predicates and it works well with MySQL. Each factor in the DNF results in a separate partial order. For e.g., the following predicate results in two partial orders; $<\{$col1, col2, col3$\}>$ and $<\{$col2, col4$\}>$.

```
E2: WHERE (
        col1 = 5 AND col2 = 'ABC'
            AND col3 IN (5, 9, 11)
    ) OR ( col2 = 'CDE' AND col4 = 8 )
```

2) Index prefix predicates: Depending on the database server, the operator plays an important role as well. Filtered upon columns associated with operators like equal (=), nullsafe equal (<=>), set membership etc. can be chained together to construct multi-column indexes wherein each participating column would typically enhance the selectivity (or the effectiveness) of the index. However, if columns associated with operators such as greater than (>), less than or equal to (<=), $BETWEEN$ etc. were concatenated together to form a multi-column secondary index, each subsequent column after the first one may not have a strict *additive* benefit on selectivity [20]. This is best explained with an example.

```
E3: WHERE col1 = 5 AND col2 = 'ABC'
        AND col3 > 5 AND col4 < 2.0
```

If a multi-column secondary index on <col1, col2, col3, col4> is added, all rows with $col1 = 5$, $col2 = ABC$ and $col3 > 5$ might have to be read and the evaluation of $col4 < 2.0$ would be done in a subsequent step. This subsequent step maybe performed by reading the qualifying rows from primary key or by using the index condition pushdown optimization [21]. Therefore, we can construct the following partial order for the expression E3: $<\{$col1, col2$\}$, $\{$col3, col4$\}>$.

This partial order represents a set of 4 multi-column secondary indexes where $col1$ and $col2$ precede $col3$ and $col4$. The order of $col1$ with respect to $col2$ is immaterial. Similarly, the order of $col3$ with respect to $col4$ would have been immaterial as well but it depends on the overall selectivity of the individual predicates $col3 > 5$ and $col4 < 2.0$. Putting the column corresponding to the more selective atomic predicate first would be more efficient and can be done by using dataless indexes. Therefore, the subpredicates $col1 = 5$ and $col2 =$'ABC' decide the prefix of the candidate index. This process is described by the procedure $GenerateCandidateIndexPredicates$ in algorithm 5.

More formally, an index prefix predicate is an atomic subpredicate of the filter clause which can be resolved by an index scan of rows that share a constant non-empty prefix. This constant prefix can be determined from the atomic subpredicate itself. For e.g. the atomic predicate $col1 = 5$ can be resolved using an index on $col1$ by scanning all the rows that have the prefix 5 and $col2 =$'ABC' can be resolved using an index on $col2$ by scanning all the rows with prefix 'ABC'. Contrary to this, the atomic predicate $col3 > 5$ can also be resolved using an index on $col3$ by performing a range scan. However, all the matching rows need *not* have a common prefix. Therefore, $col3 > 5$ is not an index prefix predicate.

C. Joins

Table join operations compute a subset of the cartesian product of two or more tables (relations). Therefore, it can be thought of as selection operation (described in subsection IV-B) on a cross-product of two or more tables.

For most database execution engines, only two tables are joined together at a time. The order in which tables are joined together plays an important role in determining the performance of a join query[3]. Determining the most efficient join order for a query is in itself a NP hard problem [23]. There is a circular dependency between the problem of selecting the most efficient indexes for facilitating a join query and determining the best join order for it. Most relational database engines employ heuristics to predict an efficient join order rather than computing the most efficient one. The goal is to prevent query optimization from becoming more expensive than actual execution. Therefore, only a small number of join orders are even considered by the optimizer [24].

Our approach can generate candidate indexes for join queries which works well for transactional workloads. Column usage metadata includes identifiers for table instances that participate in a join condition with any given column of another table instance. This helps us construct the join graph where nodes are table instances and edges between them represent a predicate in the join condition involving columns from the participating tables instances (nodes). Consider the following query Q2 with the join graph shown in Figure 2.

```
Q2: SELECT t1.col1, t2.col2, t3.col3
    FROM t1, t2, t3
    WHERE t1.col2 = t3.col2
        AND t2.col4 = t3.col7
```

AIM accepts a positive integer j to limit its search for join orders. All table instances with columns participating in the

[3]The order of some join types is predetermined (e.g. straight join [22])

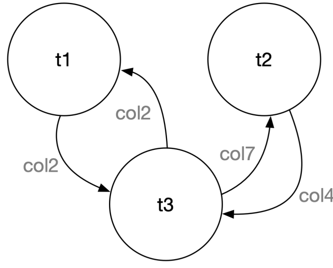


Fig. 2: Join graph for Q2

join predicates with at most j other table instances are selected. The candidate indexes for the selected table instances include all possibilities of join orders with respect to that table. The upper bound is exponential in j, since every participating table could either precede or succeed the selected table instance. Therefore, the value of j is usually a small positive integer. Although, we do not try to exhaustively facilitate all join orders for table instances that share a join predicate with more than j tables, we have not seen any incremental benefit of exploring $j > 3$ for any production workload so far. The impact of choosing j is demonstrated in Figure 6.

If the join order is predetermined, the join predicates can be merged with the selection predicates to generate partial orders representing candidate indexes benefiting both the join and selection operations. Intuitively, if we assume that the join order for the query Q2 above is [$t3$, $t1$, $t2$], it makes sense to try out an index on $t2.col4$ since $t3$ precedes it and we perform a lookup into table $t2$ for every value of $t3.col7$. However, if the join order were [$t1$, $t2$, $t3$], an index on $t2.col4$ would not reduce the number of records considered from $t2$.

Our approach considers candidate indexes to facilitate several possible join orders and lets the query optimizer pick the best one. The pseudocode for generation of index candidates catering to filter and join predicates is provided in algorithm 4. The procedure $ReferencedColumns$ is an overloaded helper which returns the columns referenced in an entity. The entity could be a table in a query or a partial order.

It should again be noted that the greedy approach (utilized by most modern index selection algorithms) would not be able to discover optimal indexes for complex join queries because their exploration logic doesn't consider co-ordinated exploration of candidate indexes across multiple tables.

Algorithm 3: JoinedTablesPowerset

Input: $\mathcal{Q}$: query, t: table, j: join parameter
Output: $\mathcal{T}_j$: Power set of tables which have join predicates with t

1 $\mathcal{T} \leftarrow$ Set of tables with join predicates with t in $\mathcal{Q}$;
2 **if** $|\mathcal{T}| > j$ **then**
3 $\quad \mathcal{T} \leftarrow \phi$
4 **end**
5 return $2^{\mathcal{T}}$

Algorithm 4: GenerateCandidatesForSelection

Input: $\mathcal{Q}$: query, j: join parameter, $mode$: covering / non-covering
Output: $\mathcal{IP}$: Set of partial orders of index columns

1 $\mathcal{IP} \leftarrow \phi$;
2 $\mathcal{T} \leftarrow$ tables queried in $\mathcal{Q}$;
3 **foreach** $t \in \mathcal{T}$ **do**
4 $\quad \mathcal{C}_F \leftarrow$ Set of columns in t that feature in filter predicates of $\mathcal{Q}$;
5 $\quad$ **foreach** $S \in JoinedTablesPowerset(\mathcal{Q}, t, j)$ **do**
6 $\quad\quad \mathcal{C}_J \leftarrow$ Set of columns in t that feature in join predicates (of $\mathcal{Q}$) with any table in S;
7 $\quad\quad candidates \leftarrow GenerateCandidateIndexPredicates(\mathcal{Q}, \mathcal{C}_F \cup \mathcal{C}_J)$;
8 $\quad\quad$ **if** $mode = covering$ **then**
9 $\quad\quad\quad candidates \leftarrow \{c.append(ReferencedColumns(\mathcal{Q}, t) \setminus ReferencedColumns(c)) \mid c \in candidates\}$;
10 $\quad\quad$ **end**
11 $\quad\quad \mathcal{IP} \leftarrow \mathcal{IP} \cup candidates$;
12 $\quad$ **end**
13 **end**
14 return $\mathcal{IP}$

Algorithm 5: GenerateCandidateIndexPredicates

Input: $\mathcal{Q}$: query, $\mathcal{C}$: columns used for selection
Output: $\mathcal{IP}$: Set of partial orders of index columns

1 $\mathcal{IP} \leftarrow \phi$;
2 $\mathcal{G}_C \leftarrow FactorizeIndexPredicates(\mathcal{Q}, \mathcal{C})$;
3 **foreach** $\mathcal{G} \in \mathcal{G}_C$ **do**
4 $\quad \mathcal{C}_{IPP} \leftarrow \{c \mid c \in \mathcal{G}$ and column c features in an index prefix predicate (IPP)$\}$;
5 $\quad \mathcal{C}_{RSP} \leftarrow \mathcal{G} \setminus \mathcal{C}_{IPP}$;
6 $\quad last_col \leftarrow \text{argmin}_{c \in \mathcal{C}_{RSP}}\, dataless_index_cost(\mathcal{Q}, < \mathcal{C}_{IPP}, \{c\} >)$;
7 $\quad \mathcal{IP} \leftarrow \mathcal{IP} \cup \{< \mathcal{C}_{IPP}, \{last_col\} >\}$;
8 **end**
9 return $\mathcal{IP}$

D. Group By

In case of a group by operation on a table instance, a secondary index can come in handy when reading rows in grouping order and computing any expression involving aggregation for each group. An example is as follows.

```
Q3: SELECT col3, COUNT(*) FROM t1
    GROUP BY col3

Q4: SELECT col3, SUM(col1)
    FROM t1 WHERE col2 = 5 GROUP BY col3
```

For query Q3, a secondary index on $col3$ might come in handy when evaluating the aggregate values per group (defined by unique values of $col3$). For query Q4, however, the follwoing partial order would be more efficient since it forms a covering index while facilitating selection and grouping operations simultaneously: $<\{\text{col2}\}, \{\text{col3}\}, \{\text{col1}\}>$.

Notice that $col2$ precedes the grouping column ($col3$) since it features in an index prefix predicate defined above.

Candidate generation for group by clause is described in algorithm 6.

Algorithm 6: GenerateCandidatesForGroupBy

```
    Input: Q: query, j: join parameter, mode: covering /
           non-covering
    Output: IP: Set of partial orders of index columns
 1  IP ← φ;
 2  T ← tables queried in Q ;
 3  foreach t ∈ T do
 4      C_G ← Set of columns in t that feature in the group-by
          clause within Q;
 5      if mode = non − covering then
 6          IP ← IP ∪ {< C_G >};
 7      else
 8          C_F ← Set of columns in t that feature in filter
              predicates of Q;
 9          foreach S ∈ JoinedTablesPowerset(Q, t, j) do
10              candidate ←<>;
11              C_J ← Set of columns in t that feature in join
                  predicates (of Q) with any table in S;
12              G_C ←
                  FactorizeIndexPredicates(Q, C_F ∪ C_J);
13              foreach G ∈ G_C do
14                  C_IPP ← {c | c ∈ G and column c features
                      in an index prefix predicate (IPP)};
15                  candidate ← candidate.append(C_IPP);
16                  candidate ← candidate.append(C_G);
17                  remaining_columns ←
                      ReferencedColumns(Q, t)\(C_IPP ∪ C_G)
                      ;
18                  candidate ←
                      candidate.append(remaining_columns);

19                  IP ← IP ∪ {candidate};
20              end
21          end
22      end
23  end
24  return IP
```

E. Order By

A SQL query might request only a few rows in the output which are sorted in a specific order. Consider the following query.

```
Q5: SELECT t1.col13, t1.col14,
                t2.col25, t1.col17
    FROM t1 LEFT JOIN t2
        ON t1.col11 = t2.col21
    WHERE t1.col12 IN ('ABC', 'DEF')
    ORDER BY t1.col13 LIMIT 2
```

It is possible that the number of rows in t_1 matching the predicate t1.col12 IN ('ABC', 'DEF') is too high and a more efficient execution plan might read rows of t_1 in ascending order of $col13$ and check if the predicate t1.col12 IN ('ABC', 'DEF') is satisfied. In such a scenario, a secondary index on $col13$ would be more beneficial than one on $col12$. Candidate generation for order by clause is described by algorithm 7.

Algorithm 7: GenerateCandidatesForOrderBy

```
    Input: Q: query, j: join parameter, mode: covering /
           non-covering
    Output: IP: Set of partial orders of index columns
 1  IP ← φ;
 2  T ← tables queried in Q ;
 3  foreach t ∈ T do
 4      C_O ← Sequence of columns in t that feature in the
          order-by clause within Q;
 5      if mode = non − covering then
 6          IP ← IP ∪ {C_O};
 7      else
 8          C_F ← Set of columns in t that feature in filter
              predicates of Q;
 9          foreach S ∈ JoinedTablesPowerset(Q, t, j) do
10              candidate ←<>;
11              C_J ← Set of columns in t that feature in join
                  predicates (of Q) with any table in S;
12              G_C ←
                  FactorizeIndexPredicates(Q, C_F ∪ C_J);
13              foreach G ∈ G_C do
14                  C_IPP ← {c | c ∈ G and column c features
                      in an index prefix predicate (IPP)};
15                  candidate ← candidate.append(C_IPP);
16                  candidate ← candidate.append(C_O);
17                  remaining_columns ←
                      ReferencedColumns(Q, t) \ (C_IPP ∪
                      elems C_O);
18                  candidate ←
                      candidate.append(remaining_columns);

19              end
20              IP ← IP ∪ {candidate};
21          end
22      end
23  end
24  return IP
```

V. Salient Features

A. Solution Granularity

As described above, AIM compromises on the solution granularity to offer lower runtimes. Utilization of the query structure to severely limit our search space prevents the algorithm from exploring too many index configurations. It can support the following levels of solution granularity.

- Only a subset of the queries can be chosen to be optimized. Anecdotally, only the top few most expensive queries account for most of the CPU utilization.
- At the query level, each table instance in a multi-table query has three options. It can either be unindexed, may have a non-covering index or a covering index.
- Relaxation / reduction of the number of sub-predicates in the index prefix predicates (IPP) can also lead to significant cost reductions especially when the additive selectivity falls below a certain threshold.

B. Role of dataless indexes

Dataless indexes (section III-A4) are only utilized when determining the join order (when a lot of tables are involved as described in subsection IV-C), the most selective atomic

predicate that is not an index prefix predicate (algorithm 5) and picking out the primary operation for each distinct query (subsection III-D).

VI. Experimental Study

A. Comparison with manual tuning

AIM has been designed for transactional workloads served by MySQL which powers a variety of real-time applications across the industry. It supports both storage engines; InnoDB (B+ trees) and RocksDB (LSM trees). The workload generated by these applications is diverse and most databases receive dynamic ad-hoc updates. In order to compare AIM's performance against manual tuning, we conducted several tests on representative production workloads where all secondary indexes were removed and AIM was allowed to add them from scratch after analyzing the workload. Not only did AIM achieve performance comparable to manual tuning, it did so using fewer indexes in most cases. Metadata about these representative workloads is provided in Table II. The Jaccard similarity index between the sets of indexes created by DBAs and AIM is also provided. AIM was able to achieve performance at par with manually tuned databases, even for highly optimized databases with dedicated DBAs. This performance is demonstrated in Figure 3 for Product A, B & C listed in Table II. Each graph shows the CPU utilization on the machines and the observed throughput as we drop all secondary indexes and initiate AIM which recreates them from scratch on the test setup (shown in red). The control and test setups are hosted on separate machines with the exact same hardware capabilities, data and workload. The control setup (shown in blue) is left untouched with indexes created manually by DBAs.

B. Comparison with other state-of-the-art algorithms

We also benchmarked AIM against the state-of-the-art algorithms from industry (DTA [12]) and academia (Extend [11]) using the open-source framework developed by Kossmann et al [13] on PostgreSQL v12.11. The framework supports eight algorithms (AutoAdmin [3], CoPhy [25], DB2Advis [7], DTA [12], Dexter [26], Drop [27], Extend [11] and Relaxation [28]) but we chose only two to compare against for clarity in our graphs. DTA and Extend were reported as the best performing algorithms in industry and academia, respectively. The framework utilizes HypoPG [29] built for PostgreSQL and the optimizer does not account for index maintenance costs which is why purely analytical benchmarks are chosen to compare the index selection prowess of different algorithms.

The main challenge which hindered a proper comparison with DTA was its evaluation strategy, which became prohibitively expensive [13] when considering indexes of width $>$ 3 for complex workloads (TPC-DS and JOB). Therefore, we limited the setup to only consider indexes of maximum width 4 for TPC-H and width 3 for JOB. This experiment is presented in Figure 4 where the optimizer *estimated* cost of processing a workload relative to unindexed processing cost (y-axis) is plotted against different values of storage budgets (x-axis) for the TPC-H and JOB benchmarks in Figure 4a & 4c. The algorithm runtimes (y-axis) are also plotted against storage budgets (x-axis) in Figure 4b & 4d. Graphs from TPC-DS benchmark followed the same trend and are not included to save space.

The compromise implemented by AIM is evident from the benchmark results. It trades solution granularity for faster convergence. For lower storage budgets, AIM's solution is sometimes not as good as DTA or Extend since it doesn't explore the search space in a more granular fashion. However, as soon as the budget constraints are reasonably relaxed, it's solution quality is better or at par with the state-of-the-art algorithms. The main thing to notice here is the *relatively cheap and stable runtime* offered by AIM since it can quickly detect the most efficient indexes for a query by utilizing its structural information. Most state-of-the-art algorithms explore configurations at a fine granularity; often greedily adding one column at a time. This leads to prohibitively large runtimes and missing out on optimal solutions where intermediate configurations are rejected for lack of incremental gain.

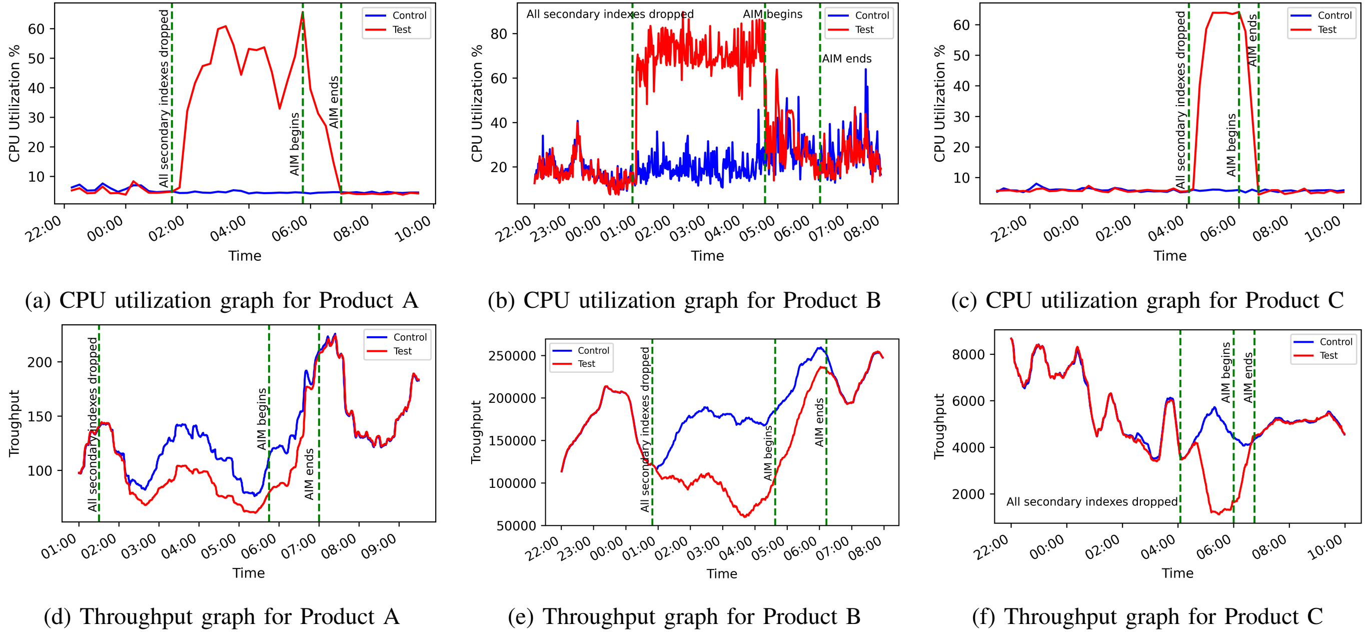
Figure 5a and Figure 5b show the performance of the solutions chosen by different algorithms across individual queries in the TPC-H benchmark (scale factor 10) with a budget constraint of 15 GB. Relatively expensive query costs are shown in log scale. It can be seen that the performance for individual queries is pretty similar across all algorithms, except for Q21. AIM chose a covering index for Q21 and PostgreSQL query optimizer assigned it a higher estimated cost. However, the actual query execution costs were similar.

C. Impact of the join parameter

Another interesting aspect of the AIM algorithm is its treatment of join queries. The only way to ensure that the most optimal join order gets picked for each query is to have indexes that support all potentially efficient join orders. On top of that, the optimizer should be able to select the optimal join order in the presence of these indexes in an efficient manner. Neither of these prerequisites can be fulfilled in practice due to various constraints [10], [30]. Therefore, AIM uses the join parameter j and limits the number of explored configurations as described in subsection IV-C. In this experiment, all secondary indexes were removed from identical databases on two separate machines. A constant perioidically repeating workload is replayed on both machines. For this experiment, we chose a transactional workload with many complex join queries to demonstrate AIM's impact. AIM progressively creates secondary indexes with increasing values of the join parameter j on one machine and a greedy incremental algorithm (Extend) creates them on the other. The difference in performance between the two solutions is visible from Figure 6. AIM's solution achieves 27% better throughput compared to the greedy algorithm (labeled GIA) and results in 4.8% lower CPU utilization. This is because the quantum of configuration exploration is limited to one column at every step and the incremental benefit might not be justified by the cost function. Consider a join clause between

TABLE II: Performance comparison between DBAs and AIM on production workloads

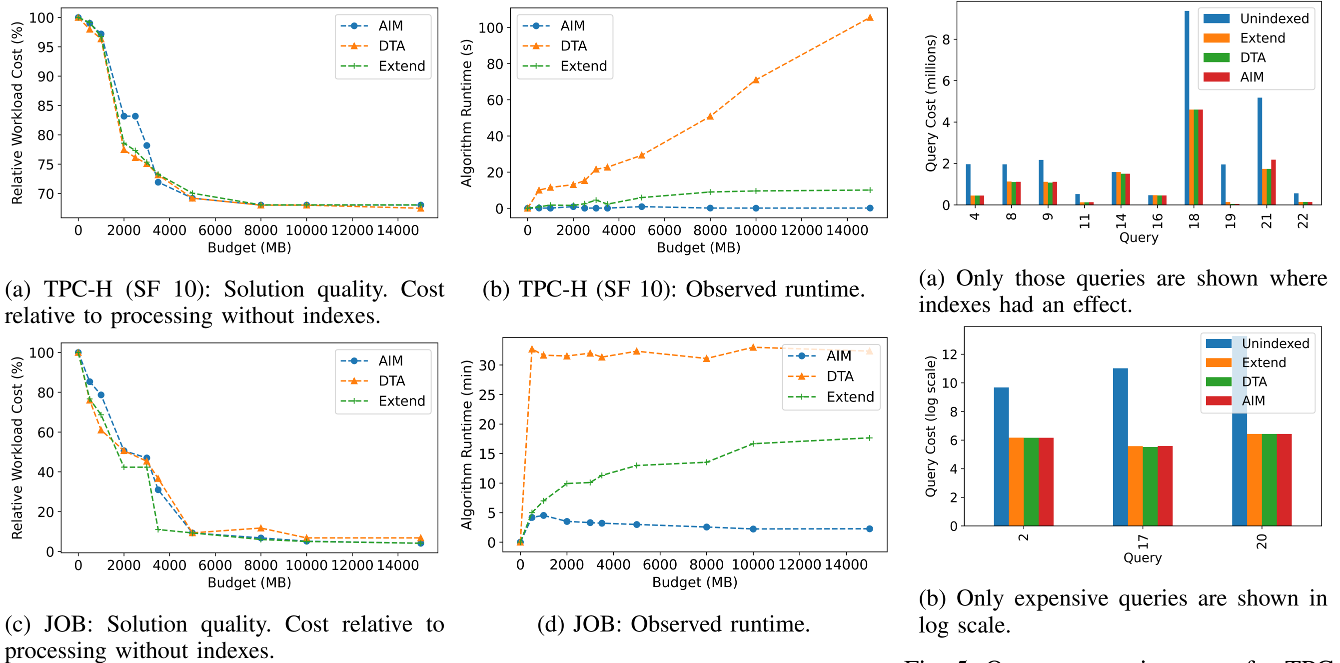
Product	Tables	Join Queries	Workload Type	Index Count		Total Indexes Size		Jaccard Similarity
				DBA	AIM	DBA	AIM	
Product A	147	67	Write Heavy	248	217	149.76 GiB	119.49 GiB	0.72
Product B	184	733	Read Heavy	287	291	7.69 GiB	9.14 GiB	0.81
Product C	42	25	Balanced	51	49	26.23 GiB	23.11 GiB	0.89
Product D	16	18	Write Heavy	56	52	10.27 GiB	8.42 GiB	0.61
Product E	51	41	Read Heavy	109	82	688.16 GiB	517.01 GiB	0.68
Product F	5	10	Read Heavy	33	34	193.83 MiB	102.52 MiB	0.97
Product G	79	386	Balanced	232	220	868.64 GiB	815.33 GiB	0.76



(a) CPU utilization graph for Product A (b) CPU utilization graph for Product B (c) CPU utilization graph for Product C

(d) Throughput graph for Product A (e) Throughput graph for Product B (f) Throughput graph for Product C

Fig. 3: CPU utilization & throughput profiles before and after AIM execution



(a) TPC-H (SF 10): Solution quality. Cost relative to processing without indexes.

(b) TPC-H (SF 10): Observed runtime.

(c) JOB: Solution quality. Cost relative to processing without indexes.

(d) JOB: Observed runtime.

Fig. 4: Estimated workload processing costs & algorithm runtimes

(a) Only those queries are shown where indexes had an effect.

(b) Only expensive queries are shown in log scale.

Fig. 5: Query processing costs for TPC-H (scale factor 10) on PostgreSQL.

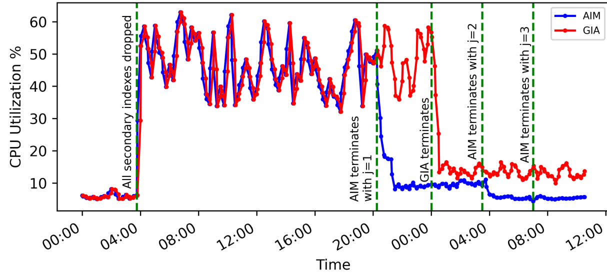


Fig. 6: Effect of join parameter

a pair of tables with three sub-predicates. It is possible that any combination of two sub-predicates is not selective enough but a combination of all three is highly selective. In such cases, greedy algorithms would not be able to detect efficient join orders for the query. Some machine learning algorithms [31] try to explore suboptimal intermediate configurations in a probabilistic fashion but utilizing the query structures for identifying changes to the existing configuration yields definitive results.

Another important aspect demonstrated by the experiment is the change in performance with respect to increasing values of j. The solution with $j = 2$ results in 16% better throughput than the one AIM arrived at with $j = 1$. The gain from increasing j from 2 to 3 was insignificant. In theory, we can construct queries on top of synthetic data distributions that would require $j > 3$ to achieve optimal performance but we haven't seen such cases in production so far.

Please note that indexes were created incrementally with sleeps in between in order to clearly observe the impact on the graphs presented in Figure 3 & Figure 6.

D. Continuous index tuning

AIM is not only good at bootstrapping secondary index creation, it also does particularly well at detecting indexes when the workload changes significantly. Most of the times, expensive queries result from new code pushes where developers forget to create supporting secondary indexes beforehand. It is evident from Figure 4 that AIM's runtime is significantly lower compared to other state-of-the-art algorithms. Therefore, we naïvely achieve continuous tuning by running AIM periodically. Continuous tuning has resulted in savings of approximately 2% CPU capacity required for serving OLTP workloads with roughly 31% of the improved queries resulting in at least an order of magnitude improvement. Adopting features like controllable overhead and phased index profiling from advanced continuous tuning algorithms like Colt [32] will be addressed by future work.

VII. Supporting Components

A. Continuous statistics export

The fault tolerant MySQL offering at Meta [33] replicates databases across multiple machines. Read queries are served by one of the many available replicas. Therefore, statistics need to be gathered from all of these machines and aggregated to get a holistic view of the workload for a given database. This is achieved by a daemon process which periodically queries every single machine running on the fleet and exports it to Meta's internal data warehouse. The export is achieved by a pub-sub system (similar to Apache Kafka [34]) and complex analytics can be run almost instantaneously.

B. MyShadow

MyShadow framework is a test environment provider which creates a temporary logical copy of the database (clone) and can replay traffic from production database instances onto test instances. This framework plays an important role in catching significant regressions that are only possible to detect in a production-like environment. It helps us make sure that there are no side-effects of running AIM on sensitive production workloads. The MyShadow framework was used to setup the experiments presented in section VI. It has the ability to sample the data and workload being replayed. Therefore, it is suitable for providing economical test beds.

C. Continuous Regression Detector

The continuous regression detector is an independent process that looks for regressions in performance of queries across the fleet due to query optimizer plan changes. It relies on the average CPU time consumption by a normalized query over a period of time. If a regression is detected due to an index added by automation, it is flagged for removal.

VIII. Operational Experience and Lessons

a) Query optimizer nuances: Our work on AIM gave us a lot of insight into the tradeoffs that database engines make when executing queries. The work of Chaudhuri and Narasayya [3] pioneered the concept of consulting the query optimizer for index selection. It was a huge step forward but optimizers (even in mature databases) make lots of mistakes [35]. There are three interesting aspects to consider. First, the optimizer might pick a sub-optimal plan which can prevent usage of beneficial indexes. We have seen several cases in production where AIM's index suggestions which offered significant reduction in query processing cost were rejected by MySQL's query optimizer. Second, optimizer switches are often used to influence the query optimizer plan selection. The number of candidates generated can be significantly reduced if candidate generation takes the value of these switches into account. Features like index skip scan [36], index merge intersections [37] etc. maybe switched off for a subset of databases due to correctness and performance bugs [38], [39]. Making the index candidate generation aware of their values improves the efficiency of the algorithm. Third, most modern algorithms do not scale well with increasingly complex workloads as a huge fraction of the runtime is spent making optimizer calls [14], [25]. In fact, while benchmarking AIM against state-of-the-art algorithms (subsection VI-B), we had to set a really high timeout for DTA when exploring candidates of width greater than or equal to 3 for TPC-DS and JOB benchmarks.

b) Economics of sharding: Databases that cannot be served by the resources on a single machine are partitioned horizontally. Each constituent partition which consists of only a subset of the data is called a shard. Since our deployment mandates common physical design across all shards of a database, the economics of the index selection problem were adjusted for heavily sharded databases. This is because the improved queries might not frequently execute on all consituent shards but each shard has to pay for the storage and maintenance of all indexes. Similarly, query regressions which occur on a subset of the shards are difficult to detect in real-time without comprehensive validation across all shards. We provide configurable parameters for performance sensitive databases to perform comprehensive validation and rely on the continuous regression detection to revert unwanted changes to the physical design.

c) Compute placement and regression management: The AIM process does not run on individual database hosts and a centralized coordinator kicks off the tuning process for a database if it detects inefficient queries. This setup prevents compute wastage on the entire fleet. The continuous regression detector is also an off-host centralized process that serves as an indispensable protection mechanism because some portions of the workload may repeat after a very long duration (e.g. monthly report generation). Therefore, reverting any regressions quickly is of paramount importance and algorithms which take hours to converge cannot be relied upon.

IX. Related & Future Work

The problem of automatically adding secondary indexes has been studied since 1970 [40]. A variety of algorithm classes have been developed over time with varying degrees of production readiness. Based on the methodology used for candidate selection, these algorithms can be broadly classified into two classes; imperative and declarative [13]. The imperative class of algorithms either start with an empty configuration and iteratively add indexes or start with a very large configuration which is then iteratively reduced. Most of these algorithms rely heavily on usage of "what-if" indexes [4]. Papadomanolakis et al. show that index selection algorithms spend 90% of their runtime in the optimizer, on average [14]. There is also a significant body of work that tries to reduce the number of "what-if" calls to the optimizer [14], [41], [42]. The problem is complicated by the fact that optimizer decisions are sometimes unreliable [10], [35] and can lead to severe plan regressions [43], [44].

The occasional shortcomings of the optimizer brings the focus to the declarative class of algorithms which either employ machine learning or linear programming. The linear programming models [25], [45], [46] solve optimization problems by specifying the optimization goal and constraints as linear equations. These formulations do not scale well with the size of the problem [11] and often restrict the solution space (e.g. restricting index width).

More recently, machine learning algorithms have been utilized for index management [42], [47]–[52]. There are two common themes across these algorithms. They either use deep reinforcement learning (DRL) [42], [48], [50], [51] for the index selection problem or craft a deep learning (DL) model [52], [53] to improve query cost prediction as compared to the optimizer.

The problem with DRL algorithms is that they are computationally expensive [31]. They usually rely on a lot of training data and are not well suited for dynamic workloads due to their higher runtimes. For e.g., the implementation of Sharma et al. [48] required 8 hours of training on CPUs for a small workload like TPC-H [13]. The DL models on the other hand operate at the query level and are better suited for analytical queries. Most of these methods do not model the index maintenance overheads. However, there are many useful ideas that can be borrowed from machine learning techniques to AIM, such as training a classifier to predict comparative performance of execution plans rather than relying on the optimizer as described by Ding et al [52]. In fact, the technique described by UDO [51] can be used to build environment specific benchmarks based on actual query performance since it generates *guaranteed* optimal plans via an expensive process. Furthermore, several avenues to improve cost models of open source query optimizers based on the utilization of interesting sort orders [54] have been identified.

AIM's candidate index selection falls under the imperative class but it tries to limit its reliance on the query optimizer by utilizing the structural properties of queries. AIM compromises on the granularity of the solution in favour of converging really fast. Since the importance of a query is a subjective attribute, it is not possible to ascertain it automatically. Therefore, our approach does not try to explore configurations that are suboptimal for constituent normalized queries but optimal for the overall performance. To the best of our knowledge, relaxation [28] is the only other modern algorithm which utilizes the query structure to a significant extent but its approach of iteratively pruning indexes after starting from a very large candidate set gives it a prohibitively expensive runtime. Unlike Relaxation, AIM does *not* need any extra instrumentation to the query optimizer.

X. Concluding remarks

In this paper, we presented AIM which is an autonomous index management solution for SQL databases. It works well for transactional workloads in production and can scale with increasingly complex workloads. A detailed discussion of the AIM algorithm proposes a novel way of exploring the search space of candidate indexes and presents a systematic treatment of complex join queries for the first time. The compromise between fast convergence and reduced solution exploration granularity has been explained thoroughly. The challenges and lessons learned from the production deployment of AIM have been shared with the hope that engineers looking to automate index management on their SQL databases can utilize this information in their implementations. Experiments described in the paper compare AIM against DBAs and other modern algorithms.

References

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database tuning advisor for Microsoft SQL Server 2005," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 930–932, 2005.

[2] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *2007 IEEE 23rd International Conference on Data Engineering (ICDE)*, pp. 826–835, IEEE, 2007.

[3] S. Chaudhuri and V. R. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, p. 146–155, 1997.

[4] S. Chaudhuri and V. Narasayya, "AutoAdmin "What-If" Index Analysis Utility," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, p. 367–378, 1998.

[5] S. Chaudhuri and V. Narasayya, "Self-Tuning Database Systems: A Decade of Progress," in *Proceedings of the 33rd International Conference on Very Large Databases (VLDB)*, pp. 3–14, 2007.

[6] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic SQL tuning in Oracle 10g," in *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pp. 1098–1109, 2004.

[7] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley, "Db2 advisor: An optimizer smart enough to recommend its own indexes," in *2000 IEEE 16th International Conference on Data Engineering (ICDE)*, pp. 101–110, IEEE, 2000.

[8] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated Automatic Physical Database Design," in *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pp. 1087–1097, 2004.

[9] S. Chaudhuri, M. Datar, and V. Narasayya, "Index selection for databases: A hardness study and a principled heuristic solution," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, no. 11, pp. 1313–1323, 2004.

[10] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?," *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[11] R. Schlosser, J. Kossmann, and M. Boissier, "Efficient scalable multi-attribute index selection using recursive strategies," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1238–1249, IEEE, 2019.

[12] S. Chaudhuri and V. Narasayya, *Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server*, October 2022. https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/.

[13] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2382–2395, 2020.

[14] S. Papadomanolakis, D. Dash, and A. Ailamaki, "Efficient use of the query optimizer for automated physical design," in *Proceedings of the 33rd International Conference on Very Large Databases (VLDB)*, pp. 1093–1104, 2007.

[15] B. Dushnik and E. W. Miller, "Partially ordered sets," *American journal of mathematics*, vol. 63, no. 3, pp. 600–610, 1941.

[16] H. Appelgate, M. Barr, J. Beck, F. Lawvere, F. Linton, E. Manes, M. Tierney, F. Ulmer, and F. W. Lawvere, "Ordinal sums and equational doctrines," in *Seminar on Triples and Categorical Homology Theory: ETH 1966/67*, pp. 141–155, Springer, 1969.

[17] MySQL Reference Manual 8.0, *Estimating Query Performance*, January 2023. https://dev.mysql.com/doc/refman/8.0/en/estimating-performance.html.

[18] S. Martello and P. Toth, "Algorithms for knapsack problems," *North-Holland Mathematics Studies*, vol. 132, pp. 213–257, 1987.

[19] S. Chaudhuri, P. Ganesan, and S. Sarawagi, "Factorizing Complex Predicates in Queries to Exploit Indexes," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 361–372, 2003.

[20] MySQL Reference Manual 8.0, *Range Access Method for Multiple-Part Indexes*, October 2022. https://dev.mysql.com/doc/refman/8.0/en/range-optimization.html#range-access-multi-part.

[21] MySQL Reference Manual 8.0, *Index Condition Pushdown Optimization*, October 2022. https://dev.mysql.com/doc/refman/8.0/en/index-condition-pushdown-optimization.htm.

[22] MySQL Reference Manual 8.0, *Join Clause*, October 2022. https://dev.mysql.com/doc/refman/8.0/en/join.html.

[23] I. Toshihide and K. Tiko, "On the optimal nesting order for computing n-relational joins," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 3, pp. 482–502, 1984.

[24] N. Reddy and J. R. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers," in *Proceedings of the 31st International Conference on Very Large Databases (VLDB)*, p. 1228–1239, 2005.

[25] D. Dash, N. Polyzotis, and A. Ailamaki, "Cophy: A scalable, portable, and interactive index advisor for large workloads," *arXiv preprint arXiv:1104.3214*, 2011.

[26] Dexter, *The automatic indexer for Postgres*, October 2022. https://github.com/ankane/dexter.

[27] K. Y. Whang, "Index selection in relational databases," *Foundations of Data Organization*, pp. 487–500, 1987.

[28] N. Bruno and S. Chaudhuri, "Automatic physical database tuning: A relaxation-based approach," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 227–238, 2005.

[29] HypoPG, *Hypothetical Indexes for PostgreSQL*, October 2022. https://github.com/HypoPG/hypopg.

[30] MySQL Reference Manual 8.0, *InnoDB Limits*, October 2022. https://dev.mysql.com/doc/refman/8.0/en/innodb-limits.html.

[31] X. Zhou, L. Liu, W. Li, L. Jin, S. Li, T. Wang, and J. Feng, "AutoIndex: An Incremental Index Management System for Dynamic Workloads," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 2196–2208, IEEE, 2022.

[32] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-line index selection for shifting workloads," in *2007 IEEE 23rd International Conference on Data Engineering Workshop (ICDEW)*, pp. 459–468, IEEE, 2007.

[33] R. Yadav and A. Rahut, "FlexiRaft: Flexible Quorums with Raft," *The Conference on Innovative Data Systems Research (CIDR)*, 2023.

[34] K. M. M. Thein, "Apache Kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[35] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton, "Predicting query execution time: Are optimizer cost models really unusable?," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 1081–1092, IEEE, 2013.

[36] MySQL Reference Manual 8.0, *Skip Scan Range Access Method*, October 2022. https://dev.mysql.com/doc/refman/8.0/en/range-optimization.html#range-access-skip-scan.

[37] MySQL Reference Manual 8.0, *Index Merge Optimization*, October 2022. https://dev.mysql.com/doc/refman/8.0/en/index-merge-optimization.html.

[38] MySQL Bug, *Skip Scan retrieves incorrect Result*, October 2022. https://bugs.mysql.com/bug.php?id=100253.

[39] MySQL Bug, *Clustered primary key included in index merge may cause higher execution times*, October 2022. https://bugs.mysql.com/bug.php?id=80390.

[40] F. Palermo, "A quantitative approach to the selection of secondary indexes," *IBM Research, RJ*, vol. 730, 1970.

[41] N. Bruno and R. V. Nehme, "Configuration-parametric query optimization for physical design tuning," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 941–952, 2008.

[42] W. Wu, C. Wang, T. Siddiqui, J. Wang, V. Narasayya, S. Chaudhuri, and P. A. Bernstein, "Budget-aware Index Tuning with Reinforcement Learning," in *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*, pp. 1528–1541, 2022.

[43] R. Borovica, I. Alagiannis, and A. Ailamaki, "Automated physical designers: what you see is (not) what you get," in *Proceedings of the Fifth International Workshop on Testing Database Systems*, pp. 1–6, 2012.

[44] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri, "Automatically indexing millions of databases in Microsoft Azure SQL database," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pp. 666–679, 2019.

[45] A. Caprara, M. Fischetti, and D. Maio, "Exact and approximate algorithms for the index selection problem in physical database design,"

IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 7, no. 6, pp. 955–967, 1995.

[46] A. Caprara and J. González, "A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem," *Top*, vol. 4, no. 1, pp. 135–163, 1996.

[47] D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan, "Regularized cost-model oblivious database tuning with reinforcement learning," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII*, pp. 96–132, Springer, 2016.

[48] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "The case for automatic database administration using deep reinforcement learning," *arXiv preprint arXiv:1801.05643*, 2018.

[49] Z. Sadri, L. Gruenwald, and E. Leal, "Online index selection using deep reinforcement learning for a cluster database," in *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pp. 158–161, IEEE, 2020.

[50] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic, "DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 600–611, IEEE, 2021.

[51] J. Wang, I. Trummer, and D. Basu, "UDO: universal database optimization using reinforcement learning," *arXiv preprint arXiv:2104.01744*, 2021.

[52] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "AI meets AI: Leveraging query executions to improve index recommendations," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pp. 1241–1258, 2019.

[53] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das, "Deep learning models for selectivity estimation of multi-attribute queries," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1035–1050, 2020.

[54] R. Guravannavar, S. Sudarshan, A. A. Diwan, and C. S. Babu, "Which Sort Orders Are Interesting?," *The VLDB Journal*, vol. 21, no. 1, pp. 145—165, 2012.