Predictive Indexing with Reinforcement Learning

ABSTRACT

There has been considerable research on automated index tuning in database management systems (DBMSs). But the majority of these solutions tune the index configuration by retrospectively making computationally expensive physical design changes all at once. Such changes degrade the DBMS's performance during the process, and have reduced utility during subsequent query processing due to the delay between a workload shift and the associated change. A better approach is to generate small changes that tune the physical design over time, forecast the utility of these changes, and apply them ahead of time to maximize their impact.

This paper presents predictive indexing that continuously improves a database's physical design using lightweight physical design changes. It uses a machine learning model to forecast the utility of these changes, and continuously refines the index configuration of the database to handle evolving workloads. We introduce a lightweight hybrid scan operator with which a DBMS can make use of partially-built indexes for query processing. Our evaluation shows that predictive indexing improves the throughput of a DBMS by 3.5–5.2× compared to other state-of-the-art indexing approaches. We demonstrate that predictive indexing works seamlessly with other lightweight automated physical design tuning methods.

1 INTRODUCTION

The performance of modern data-driven applications is often constrained by that of the underlying DBMS. The physical design of the database plays a dominant role in determining the system's performance. Thus, it is important to tune it based on the application's query workload. One key component of the physical design problem is to determine a set of indexes that balances the trade-off of accelerating query execution and reducing index maintenance overhead. Database administrators (DBAs) address this problem using *index advisors* that are offered by most database vendors [15, 60]. These tools recommend indexes to build based on the current index configuration of the database and the query workload.

Index advisors require a DBA to provide a representative workload collected over some period of time (e.g., several weeks). But in modern hybrid transaction-analytical processing (HTAP) workloads, it is unlikely that an index is globally useful over the entire workload [12, 41, 42]. DBAs must, therefore, continuously monitor and re-tune the index configuration to ensure that indexes that are appropriate for the current query workload are available. Such an *offline* index tuning process involves performing large computationally expensive index configuration changes. DBAs are required to carefully schedule these index building operations during non-peak hours or offline maintenance breaks to minimize their impact on the application. With these offline approaches, the DBMS cannot react in time to workload changes.

Several indexing approaches have been proposed to address the limitations of offline index advisors, such as online indexing [12, 48], adaptive indexing [23, 31], self-managing indexing [56], and holistic indexing [41]. *Online* indexing obviates the need for manually scheduling index configuration changes by automatically monitoring the workload and refining the index configuration. It still, however, involves computationally expensive changes, such as immediately building an entire index, which degrade the performance of the DBMS while they are being applied. *Adaptive* and *self-managing* indexing schemes instead advocate an incremental approach towards index tuning, wherein the index configuration of the DBMS is evolved using smaller steps. They build indexes partially and incrementally during query processing, and amortize the overhead of constructing an ad-hoc index across several queries. The main limitation of these schemes is that they only refine the indexes during query processing, and do not leverage idle system resources when the DBMS is not processing any queries. *Holistic indexing* overcomes this limitation by enabling the DBMS to use idle system resources for optimistically evolving the index configuration.

All of these indexing approaches examine the recent query workload in hindsight to determine the set of indexes that must be refined next. This retrospective approach towards index tuning increases the delay between a workload shift and the associated physical design change, thereby reducing the utility of the indexes. Another limitation is that these approaches can still degrade the performance of the DBMS while applying the index configuration changes. For example, even with holistic indexing, the index tuner might start populating a substantial part of an index while processing a query to materialize all the index entries matching the query's predicate. This increases the query's latency. Such latency spikes may prevent the DBMS from honoring latency service-level agreements.

In this paper, we present *predictive indexing* that uses a machine learning (ML) model to predict the utility of indexes in future, and adapts the index configuration of the database *ahead of time* to increase the utility of indexes. We propose a simple lightweight *value-agnostic hybrid scan* operator that allows the DBMS to use partially-built indexes without incurring latency spikes. A summary of the differences between predictive indexing and other state-of-the-art indexing approaches is presented in Table 1.

To evaluate our approach, we implemented the predictive index tuner and the hybrid scan operator in DBMS-X, an in-memory HTAP DBMS that is designed for autonomous operation [28]. We examine the impact of predictive indexing on the DBMS's performance and its ability to handle evolving workloads. Our results show that predictive indexing improves DBMS-X's throughput by 3.5–5.2× compared to

SIGMOD'18, June 2018, Houston, Texas USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Indexing Approach	Workload Type	Index Type	Always On	Decision Logic	Hybrid Scan
Offline [15, 20]	Static	Full	×	×	×
Online [12, 48]	Dynamic	Full	✓	Retrospective	×
Adaptive [31], Self-managing [56]	Dynamic	Partial	×	Immediate	Value-based
Holistic [41]	Dynamic	Partial	✓	Immediate	Value-based
Predictive	Dynamic	Partial	✓	Predictive	Value-agnostic

Table 1: Comparison of Automated Indexing Approaches – Qualitative differences among the different indexing schemes with respect to the types of supported workloads and indexes built, the ability to always evolve the index configuration, and the types of decision logic and hybrid scan employed.

other state-of-the-art indexing approaches. In summary, this paper makes the following contributions:

- We present a predictive indexing approach that uses reinforcement learning to forecast the utility of the index configuration and tunes it ahead of time. We propose a lightweight value-agnostic hybrid scan operator with which the DBMS can make use of partially-built indexes.
- We implement an end-to-end index tuner that decides both when and how to adapt the index configuration. We integrate different indexing approaches in the same DBMS, including (1) online [12, 48], (2) adaptive [23, 31], (3) self-managing [56], (4) holistic [41], and (5) predictive. We also develop a new open-source benchmarking suite to perform a detailed comparison of these indexing approaches [28].
- We demonstrate that predictive indexing works seamlessly with other automated physical design tuners, and is a step forward towards DBMSs designed for autonomous operation. By working in tandem, the predictive index tuner and the storage layout tuner solve two key components of the physical design problem.

The remainder of this paper is structured as follows. We first discuss the benefits of predictive indexing, and then highlight the performance impact of the hybrid scan operator on HTAP workloads in Section 2. Next, in Section 3, we describe the design of the hybrid scan operator. We then present the predictive ML model that forms the underlying decision logic of the tuner in Section 4, followed by our index tuning benchmark in Section 5. We then present our experimental evaluation in Section 6. We conclude with a discussion of related work in Section 7.

2 MOTIVATION

An important aspect of modern database applications is that their query patterns evolve over time. These changes reflect the hourly, daily, or weekly processing cycles of an organization. For instance, consider a stock exchange application. During regular trading hours, it generates a write-intensive online transaction processing (OLTP) workload. After the trading hours are over, it mostly executes read-intensive online analytical processing (OLAP) queries that examine the data collected during the day. At the end of every week, it runs more complicated analytics to generate reports and summaries.

Several indexing approaches have been proposed to allow the DBMS to handle such dynamic workloads by automatically adapting the index configuration during different workload *phases* [12, 31, 41, 48, 56]. These approaches have two main limitations. First, they can degrade the performance of the DBMS while applying a physical design change. For example, the tuner might start populating a substantial part of an ad-hoc index while processing a query, thereby increasing the query's latency. Second, the utility of the physical design change is dampened due to the delay between a workload shift and the associated change. We refer to this delay as the tuner's *reaction time*. This reaction time is comprised of two components: (1) the *detection time* is how long it takes to detect a workload shift and pick a physical design change, and (2) the *adaptation time* is how long it takes to apply the change and leverage it during query processing. Longer reaction times decrease the utility of indexes that are dynamically built to accelerate query processing.

Predictive indexing shrinks the tuner's detection time by employing a ML model to forecast the utility of the changes so that they can be applied ahead of time. It incrementally builds indexes using lightweight changes to prevent latency spikes. The DBMS employs a hybrid scan operator to leverage these partially-built indexes during query processing, and thereby shrinks the adaptation time.

Target Workloads: Predictive indexing accelerates production HTAP workloads containing both *recurring* and *ad-hoc* queries. As the workload evolves over time with recurring queries being added and removed, the predictive index tuner learns to forecast the utility of the index configuration, and adapts it ahead of time using lightweight changes. This shrinks the DBMS's reaction time on evolving HTAP workloads, and obviates the need for periodic manual tuning. We next discuss how predictive indexing differs from other state-of-the-art approaches, and then highlight the performance impact of hybrid scan on HTAP workloads in Section 2.2.

2.1 Predictive Decision Logic

Every indexing approach is driven by a *decision logic* (DL) that determines the type of physical design changes made, and when these changes are applied. Online indexing schemes employ a *retrospective DL* that examines the history of the last k queries executed to determine the set of indexes that could have helped accelerate processing those queries [2, 12, 47]. The main limitation of retrospective DL is that it increases the detection time of the tuner as it needs to examine a longer window of queries. Longer reaction times of retrospective DL decrease the utility of the changes.

In contrast, adaptive, self-managing, and holistic approaches adopt an *immediate DL* ($k = 1$) [23, 31, 41, 56]. With immediate DL, the tuner only examines the most recent query while picking an index for construction, and thus has a shorter detection time. But it cannot guard against noisy workloads with one-off queries as it immediately starts building indexes to accelerate them. With a retrospective DL, the tuner would not immediately construct these indexes as it examines a longer window of queries to estimate their utility before building them.

Predictive DL addresses the limitations of these two approaches. It uses a statistical estimator that captures workload trend and seasonality to forecast the utility of indexes. For example, the predictive index tuner can detect that a workload shift occurs every day at 8am,

Figure 1: Hybrid Scan – Value-agnostic hybrid scan over the EMPLOYEE table and the partially-built ad-hoc index on its SALARY attribute.

and then build a candidate index ahead of time at 7am, so that the index is immediately available for query processing after the workload shift. Thus, this forecasting technique reduces the tuner's detection time. Like retrospective DL, it examines a longer window of queries before picking a physical design change. This increases its detection time when it encounters a one-off query for the first time, but once it captures the query's trend, it detects it ahead-of-time. In this manner, predictive DL can both reduce the tuner's detection time, and guard against noisy workloads with one-off queries. We next present how our hybrid scan operator helps shrink the tuner's adaptation time.

2.2 Hybrid Scan

The DBMS can make use of ad-hoc indexes constructed by an index tuner in three ways. We illustrate the differences between these approaches using an example. Consider the EMPLOYEE table shown in Figure 1; assume that it is not indexed. Suppose the application executes multiple analytical queries that compute the total salary of all employees whose salary falls within a range:

```
SELECT SUM(salary) FROM EMPLOYEE
 WHERE salary > t1 AND salary < t2
```

In this scenario, the tuner would observe that the predicate in these queries repeatedly accesses the salary attribute, and build an index to accelerate these queries. Among the three approaches for using this index, the first one is to start using it only after it is fully populated. This is referred to as the *full* scheme (FULL), and is employed by online indexing [14, 45, 53]. The main limitation of this approach is that it increases the adaptation time, thus shrinking the utility of the index during query processing. A better way is to use the index even when the DBMS has not finished populating it.

Holistic, adaptive, and self-managing indexing approaches adopt such a *value-based partial* scheme (VBP) [23, 31, 41, 56]. With VBP, the tuner adds index entries based on the values present in the predicate of the queries. For the EMPLOYEE table query, the tuner will add index entries for employees whose salary $\in (t_1, t_2)$. The DBMS can, then, efficiently handle subsequent queries accessing the same sub-domain of the salary attribute using the index. However, it can do so only after the sub-domain is completely indexed. Another limitation of VBP is that it can still degrade the DBMS's performance while building the index. When the number of employees whose salary $\in (t_1, t_2)$ is large, the tuner immediately adds index entries

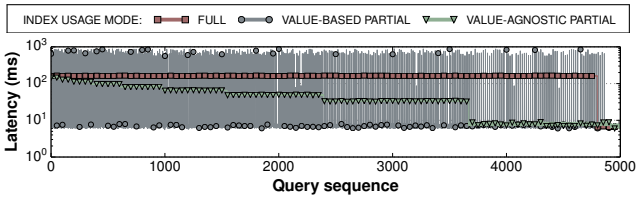


Figure 2: Ad-hoc Index Usage Schemes – This time series graph illustrates the benefits of using a partially-built index during query processing. While the query latency follows a bimodal distribution with VBP, it gradually drops over time with VAP.

for those employees while processing the query, thus causing latency spikes. Furthermore, the tuner must maintain an additional data structure (e.g., a covering tree [56]) for each index to keep track of the indexed sub-domains, so that the DBMS can determine if it can process the query using the index.

We propose a *value-agnostic partial* scheme (VAP), where the tuner only adds entries for tuples contained in a fixed number of pages at a time during each tuning cycle, irrespective of the value of the salary attribute in these tuples. With VAP, the DBMS can use an index to accelerate query processing without having to immediately populate the (t_1, t_2) sub-domain. This design ensures that the index construction overhead is independent of the value distribution of the salary attribute. The DBMS reads the partially-built index for the already indexed pages, and falls back to a table scan for the other pages. With VAP, the tuner does not need to maintain any additional data structures. Decoupling index construction from query processing ensures that the tuner only performs lightweight physical design changes, thereby preventing latency spikes. We defer a detailed description of this scheme to Section 3.

We present a motivating experiment to illustrate the performance implications of employing these three different schemes. We load 10m tuples into the EMPLOYEE table whose salary attribute is an integer value from a Zipf distribution in the range [1,1m] [1]. We configure the input parameters t_1 and t_2 in the query's predicate such that it selects 1% of the employees. We execute 5000 queries of the same query type with different input parameters.

The results are shown in the time series graph in Figure 2. We observe that with FULL, the query latency drops sharply after the tuner completely builds the index. Till then, the DBMS scans the table during query processing. With VBP, the query latency follows a bimodal distribution. The latency matches that of an index scan when the query accesses an already indexed sub-domain, but then degenerates to a table scan when it retrieves tuples from a sub-domain that has not been indexed yet. In the latter case, the tuner immediately populates entries associated with the sub-domain in the index while processing the query, thus causing latency spikes.

With VAP, the query latency gradually drops over time. This is because the tuner incrementally populates the index over several tuning cycles, and the hybrid scan operator increasingly leverages it to skip scanning parts of the table. A key observation from the time series graph is that VAP prevents latency spikes. After the tuner fully populates the index, the DBMS executes the query 10.1× faster than a table-scan. The cumulative time taken by the DBMS to execute this workload with VAP is 1.6× and 3.2× shorter than that

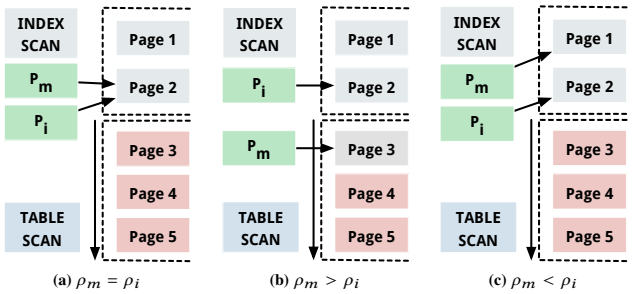


Figure 3: Hybrid Scan Operator – Different scenarios depicting the page from which the hybrid scan operator must start scanning the table in order to ensure that it returns each matching tuple once and exactly once.

taken with the VBP and FULL schemes, respectively. The tuner periodically pauses the tuning process to control the index construction overhead. The impact of the partial schemes (VAP and VBP) is more pronounced during such pauses, as they enable the DBMS to leverage the partially-built indexes.

The main issue with using a partially-built index is that since the tuner is dynamically populating the index, unless care is taken, the hybrid scan operator will return the same matching tuple twice. Moreover, it may not be able to find a matching tuple if that tuple has not yet been inserted into the index. In the next section, we describe how to solve these problems with our hybrid scan operator. We then present the predictive index tuner in Section 4.

3 HYBRID SCAN

The hybrid scan operator is a combination of an index and table scan. We illustrate it using the query on the EMPLOYEE table that contains nine tuples stored across three pages. Figure 1 shows the point when the tuner has finished indexing the tuples contained in the first two pages, and is in the process of indexing the third page. It has indexed only the first tuple in the third page, and suppose that this tuple satisfies the query's predicate. While processing this query, the hybrid scan reads the partially-built index to cover the first two pages and then resorts to a table scan over the third page. It then returns the tuples retrieved using these two operations.

This design ensures that the DBMS finds all the tuples satisfying the predicate because the operator covers every page at least once when scanning the index and table. It can, however, still return the same matching tuple twice, if it is retrieved by both operations. We next describe how hybrid scan removes such duplicate tuples.

The DBMS assigns each page a unique *page identifier* (ρ). While populating an ad-hoc index on a table, the index tuner traverses its pages in ascending order with respect to the page identifier. During query processing, the hybrid scan operator goes over the pages in the same order. While scanning the index, the operator keeps track of the identifiers of two pages: (1) the page with the largest identifier that contains a matching tuple (ρ_m) and (2) the page with the largest identifier that has already been completely indexed (ρ_i).

When the index scan finishes, the operator determines the page from where it should begin the table scan by computing $\max(\rho_m, \rho_i + 1)$. This is because, as shown in Figure 3, there are only three possibilities: (1) $\rho_m = \rho_i$, (2) $\rho_m > \rho_i$, or (3) $\rho_m < \rho_i$. In all of these, starting the table scan from $\max(\rho_m, \rho_i + 1)$ ensures that it returns all matching tuples at least once. Only when $\rho_m > \rho_i$, the operator additionally needs to remove duplicate tuples present in the *overlapping page*. The EMPLOYEE table example illustrates this scenario. After the index scan, the operator first populates the matching tuples contained in the overlapping third page in a sorted data structure. Then, during the subsequent table scan over the same page, it checks for duplicate tuples using the sorted data structure, and skips returning them twice.

Concurrency Control & Updates: DBMS-X employs the multi-version concurrency control (MVCC) protocol for scheduling transactions [39, 58]. The DBMS records the versioning meta-data alongside each tuple, and uses it to determine whether a tuple version is visible to a transaction. While handling updates, it appends the modified tuple versions in the table. Since the tuner will subsequently index these tuples, the DBMS does not propagate the changes to the ad-hoc indexes built by the tuner. The hybrid scan operator, therefore, reads updates made by concurrent transactions that are visible as per the MVCC protocol during its table scan [36]. This design allows the hybrid scan operator to work well on a wide variety of HTAP workloads, including those containing long-running scan queries.

Query Optimization: We now describe how we extended the query optimizer to use hybrid scan. DBMS-X's optimizer examines the query's predicates and the access paths available on its referenced tables, and estimates a cost for each plan [49, 51]. It uses a hybrid scan access path similar to how it employs an index scan access path. It picks a hybrid scan only for processing high selectivity queries, and switches to a table scan otherwise.

3.1 Value-Agnostic & Value-Based Scan

The value-agnostic hybrid scan operator differs from the value-based operator employed by prior indexing approaches [23, 31, 41, 56]. With the value-agnostic operator, the tuner can better control the index construction overhead. It incrementally populates the index by adding entries only for a fixed number of pages at a time. This decouples the indexing overhead from the value distributions of the attributes being indexed. The partially-built index only needs to keep track of the number of pages that have already been fully indexed, so that the operator can determine the page from which it must begin the table scan operation. The main limitation of the value-agnostic operator is that the query processing time drops linearly over time, unlike the logarithmic drop observed with the value-based operator [7, 31].

With the value-based operator, after the tuner completely populates a sub-domain of an index, it can skip the table scan operation while handling subsequent queries for the indexed sub-domain. This shrinks the query processing time but causes latency spikes when the size of the sub-domain accessed is large. As such, the tuner may not be able to react in time to future workload shifts. Further, the index must maintain meta-data about the sub-domains that have already been indexed, so that the operator can safely skip the table scan operation when scanning indexed sub-domains. To prevent latency spikes, we therefore employ the value-agnostic operator. We note that both hybrid scan operators help shrink the *adaptation time* of the DBMS in comparison to the FULL scheme employed by online indexing approaches [14, 53]. This is because they enable the DBMS to immediately make use of partially-built indexes.

Algorithm 1 Predictive Indexing Algorithm

Require: Recent queries $\mathcal{Q}$ ($\mathcal{R}$ and $\mathcal{W}$), Index configuration $\mathcal{I}$, Index storage budget $\mathcal{S}$

 # Executed once during every tuning cycle

function EVOLVE-INDEX-CONFIGURATION($\mathcal{Q}$, $\mathcal{I}$, $\mathcal{S}$)

 # Stage I : Workload classification

Label $\mathcal{L}$ = Workload-Classifier($\mathcal{Q}$)

 # Stage II : Action Generation

Candidate indexes $\mathcal{C}$ = Candidate-Indexes($\mathcal{Q}$)

Overall utility $\mathcal{O}$ = Overall-Utility($\mathcal{I}$, $\mathcal{C}$, $\mathcal{Q}$)

Index configuration $\mathcal{I}'$ = Index-Knapsack($\mathcal{L}$, $\mathcal{S}$, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{U}$, $\mathcal{O}$)

Apply lightweight changes to reach index configuration $\mathcal{I}'$

 # Stage III : Index Utility Forecasting

Forecasted utility $\mathcal{U}$ = Holt-Winters-Forecaster($\mathcal{U}$, $\mathcal{O}$)

end function

4 PREDICTIVE DECISION LOGIC

We next present how the tuner's predictive DL decreases its *detection time*. The complexity of the index tuning problem can be attributed to two factors [20]. First, the tuner must balance the performance gains of an index during query processing against the overhead of maintaining it while executing queries that update the index. The tuner must, therefore, adapt the index configuration based on the current workload mixture. On a read-intensive workload, it should build additional indexes for accelerating query processing. But when the workload shifts to write-intensive queries, it should drop a subset of these indexes to reduce the index maintenance costs.

Second, the tuner must take the storage space consumed by the index into consideration. This is an important constraint even when there are no updates. Indexes typically consume a significant portion of the total memory footprint of the DBMS, particularly on OLTP workloads. This is because OLTP applications often maintain several indexes per table to ensure that the queries execute quickly. For instance, a previous study found that indexes consume more than half of the storage space used by an in-memory DBMS for OLTP workloads [59]. The tuner must, therefore, ensure that the indexes fit within the available memory.

The predictive machine-learning (ML) model contains three principal components: (1) workload classifier, (2) action generator, and (3) index utility forecaster. These components perform the three steps that correspond to the *observe-react-learn* template that an agent typically follows in reinforcement-learning [33]. Algorithm 1 presents an overview of the predictive indexing algorithm.

During every index tuning cycle, the classifier examines the recent query workload that is tracked by a lightweight *workload monitor*, and determines the type of the workload [8, 17, 37]. Based on the workload classification, the recently executed queries, the utility of the indexes present in the current index configuration, and the recent query workload, the action generator performs a set of *actions*, that involve building and dropping indexes. These actions change the index configuration of the DBMS, and the utility of this *state transition* is fed back as input by the index utility forecaster to the action generator as the reinforcement signal [33]. Through this systematic trial-and-error, the tuner learns to choose actions that increase the utility of the index configuration, thereby improving the performance of the DBMS. We next describe these three model components in further detail.

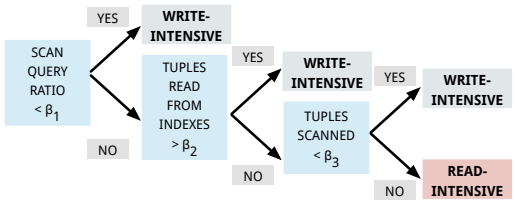


Figure 4: Workload Classifier – The pruned decision tree used by the tuner to determine the type of the query workload.

4.1 Workload Classification

The type of the DBMS's current query workload is an important consideration for the index tuning process. This is because the benefits of an index during the workload's *read-intensive* phases could be outweighed by the cost of maintaining it during the *write-intensive* phases. The tuner constructs an index only when it expects it to be beneficial for the current and near-future workloads.

We construct a workload classifier in the form of a *decision tree* using the classification and regression tree (CART) algorithm [10]. This algorithm constructs a binary tree, where each node contains a feature and threshold that yields the largest information gain. We train the classifier using *workload snapshots* collected by running a set of OLTP and OLAP benchmarks in DBMS-X [18]. Each snapshot contains meta-data about k recently executed queries, and is manually assigned a *classification label*. The collected meta-data contains a set of domain-specific *features*, such as ratio of the number of `SELECT` statements to that of `UPDATE`, `INSERT`, and `DELETE` statements [19]. We construct a training set of snapshots that are each assigned one of the following labels:

- **Write-Intensive:** The workload is characterized by a large number of short on-line transactions that mutate the database. The maintenance cost of an ad-hoc index might outweigh its benefits during query processing on this type of workload.
- **Read-Intensive:** The workload is characterized by a small number of complex read-only queries that often involve aggregations. The benefits of ad-hoc indexes is often greater than the index maintenance overhead on this type of workload.

We use a decision tree for the classifier because it is better suited for this classification task compared to other ML techniques by virtue of its *interpretability*; it is easier to understand the classification rules and explain the classifier's decisions. Based on our domain-specific knowledge, we configure the DBMS's monitor to collect the following features in every workload snapshot:

- **Ratio of scan queries to mutators:** The ratio of the number of `SELECT` to `UPDATE`, `INSERT`, and `DELETE` statements.
- **Ratio of tuples accessed using indexes:** The ratio of tuples that are accessed via indexes rather than directly from the table. It tends to be higher for write-intensive workloads.
- **Number of tuples scanned:** The average number of tuples scanned (higher for read-intensive workloads).

Figure 4 presents the pruned decision tree for workload classification. The higher-level nodes in the decision tree correspond to features that are more important in the classification process. The

ratio of queries to mutators is the crucial feature for workload classification. It is important to not take features that are dependent on the system utilization and database configuration into consideration, such as metrics like system throughput and cache hit ratio.

The tuner uses the classifier during each tuning cycle to determine the current query workload type. Based on this classification, it determines how to next evolve the index configuration. If it detects a write-intensive workload, it prunes the configuration by dropping ad-hoc indexes of low utility[1]. In contrast, if the classifier detects a read-intensive workload, then the tuner expands the configuration by building auxiliary indexes to accelerate the analytical queries. The classifier only emits classifications that are supported by a minimum threshold of query samples. This ensures that it is robust during periods of low system utilization. We next describe how the tuner generates actions for mutating the index configuration.

4.2 Action Generation

The action generator picks the actions that need to be taken to improve the performance of the DBMS. These actions could either involve building an index or dropping an existing index. We note that the following discussion is relevant for both *primary* and *secondary* indexes. The generator picks these actions based on the current workload classification, the recently executed queries, and the utility of the indexes present in the current index configuration. These actions mutate the index configuration of the DBMS, and the value of this state transition is later estimated by the forecaster.

The lightweight workload monitor tracks the attributes that are accessed by each query. The goal is to determine the set of attributes present in the statement predicates and clauses that should be indexed by an ad-hoc index. In particular, it keeps track of the attributes appearing in equal, range, and join predicates, the attributes present in `GROUP BY` and `ORDER BY` clauses, and the remaining attributes referenced in the SQL statement. The monitor stores this information for each individual table. The generator then uses this information to enumerate various combinations of these attribute sets, and determines the set of *candidate indexes* that are not present in the current index configuration [53]. The set of candidate indexes includes both *single-attribute* and *multi-attribute* indexes. We next describe our methodology for computing the utility of an index.

Query Processing Utility: The tuner uses the query optimizer to obtain the estimated processing cost of a given query in the presence and absence of a candidate index [13, 53]. We refer to the difference between the costs of executing a query in the presence and absence of an index as its *query processing utility* (QPU). The tuner uses this information to determine whether it should construct a particular candidate index or not. QPU of a candidate index $\mathcal{I}$ is derived by evaluating its impact on the set of scan queries $\mathcal{R}$ in the latest workload snapshot whose underlying table scans can be accelerated using $\mathcal{I}$. Let us denote the cost of processing a query r using only the existing indexes as $\eta(r)$, and let $\eta(r, \mathcal{I})$ represent the cost of processing the same query using $\mathcal{I}$ in addition to the existing indexes ($\eta(r) \geq \eta(r, \mathcal{I})$). Then, QPU of $\mathcal{I}$ is represented by:

$$\text{QPU}(\mathcal{I}, \mathcal{R}) = \sum_{r \in \mathcal{R}} (\eta(r) - \eta(r, I))$$

[1]The tuner assigns high utility values to indexes that are used for processing `UPDATE` statements even in a write-intensive workload, and will thus refrain from dropping them.

Index Maintenance Cost: The tuner next considers the overhead of maintaining the candidate and currently built indexes while executing statements that mutate the index. We refer to this overhead as the *index maintenance cost* (IMC). This is crucial for the DBMS to perform well on write-intensive workloads. For each index, the tuner keeps track of the queries $\mathcal{W}$ that update, insert, or delete entries in the index. Let us denote the cost of maintaining an index $\mathcal{I}$ while processing a query w by $\tau(w, \mathcal{I})$. The IMC of $\mathcal{I}$ is represented by:

$$\text{IMC}(\mathcal{I}, \mathcal{W}) = \sum_{w \in \mathcal{W}} \tau(w, \mathcal{I})$$

The *overall utility* of an index is obtained by discounting the index maintenance cost from the query processing utility of an index:

$$\text{Overall Utility}(\mathcal{I}, \mathcal{R}, \mathcal{W}) = \text{QPU}(\mathcal{I}, \mathcal{R}) - \text{IMC}(\mathcal{I}, \mathcal{W})$$

Index Knapsack Problem: After determining the overall utility of the candidate and current indexes, as shown in Algorithm 1, the tuner constructs a subset of the candidate indexes and drops a subset of the currently built indexes, after taking their storage footprint into consideration. We can formulate it as a 0-1 knapsack problem [38]. Let C represent the set of candidate and currently built indexes. The set of candidate indexes includes: (1) indexes that are expected to speed up queries in the near future by the index utility forecaster, and (2) indexes that accelerate newly encountered queries whose trends have not yet been captured by the forecaster. For the latter subset, we bootstrap the utility of these indexes in the forecaster's model with their overall utility, as illustrated in Algorithm 1. The tuner dampens the utility of redundant indexes with correlated attributes by characterizing their interactions [12].

By capturing the index utility patterns, the forecaster enables the tuner to look ahead into the future and optimistically build indexes. Let us assume that each index $c \in C$ has a utility $U_c \geq 0$ and an estimated storage footprint $S_c \geq 0$. For newly suggested indexes, U_c refers to the overall utility. In case of indexes that are either currently built or existed in the past, it represents the forecasted utility. Let B denote the storage space available for storing indexes. Then, the solution of the knapsack problem is a subset of indexes $C' \subseteq C$ with maximal utility:

$$\text{maximize} \sum_{c \in C'} U_c \text{ such that } \sum_{c \in C'} S_c \leq B$$

The tuner uses a fully polynomial time approximation scheme that delivers solutions of the knapsack problem in polynomial time [55]. The first step of this approximation scheme is a greedy algorithm that adds indexes to the knapsack in the decreasing order of utility per unit storage size (U_c/S_c) until the knapsack cannot fit any more indexes. It then uses dynamic programming to determine the desired subset of candidate indexes C'.

Index Configuration Transition: Based on the classification by the workload classifier presented in Section 4.1, the tuner determines the minimum utility (U_{min}) required for an index to exist. It scales up U_{min} while handling a write-intensive workload, and scales it down on a read-intensive workload. After determining the indexes to add and drop, the action generator applies those changes in the index configuration. We refer to this step as a state transition towards the desired final index configuration state. The tuner amortizes the

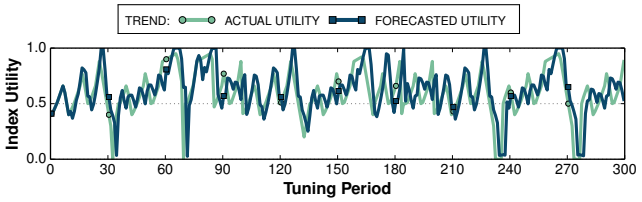


Figure 5: Learning Utility – Forecasting index utility using the Holt-Winters method with multiplicative seasonality.

overhead of this state transition over a set of tuning cycles to ensure that the DBMS's performance is predictable.

4.3 Index Utility Forecasting

The final component of the tuner is the index utility forecaster which predicts the utility of indexes in the near future. As shown in Algorithm 1, the forecasted utility of the index configuration after the state transition is fed back as input to the action generator during the next tuning cycle. Using this feedback, the tuner learns to pick actions that increase the long-run sum of the reinforcement signal, which is the overall utility of the index configuration.

Holt-Winters Seasonal Method: To forecast the utility of the indexes, the tuner uses a variant of *exponential smoothing*. This method predicts the index utility by computing a weighted average of past utility observations with the weights decaying exponentially as the observations get older. The *Holt-Winters seasonal method* extends the basic exponential smoothing technique to allow forecasting of data with a trend besides capturing seasonality [30, 57]. This method is comprised of the forecast equation along with three smoothing equations. The three smoothing equations estimate the *level* l_t, the *trend* b_t, and the *seasonal* component s_t of a time series.

$$\text{Forecast equation: } \hat{y}_{t+h|t} = (l_t + hb_t)s_{t-m+h_m}$$
$$\text{Level equation: } l_t = \alpha(y_t/s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1})$$
$$\text{Trend equation: } b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$
$$\text{Seasonal equation: } s_t = \gamma(y_t/(l_{t-1} + b_{t-1})) + (1-\gamma)s_{t-m}$$

Here, α, β, and γ are the smoothing parameters for the utility level, trend, and seasonality respectively, and α, β, $\gamma \in [0,1]$. m denotes the period of the seasonality.

Using the forecast equation, the tuner predicts the utility of indexes in the near future after factoring in the trend and seasonality aspects of the time series, as shown in Figure 5. The accuracy of the forecaster's predictions with respect to an index increases over time as it obtains more observations. The Holt-Winters method dampens the utility of indexes that are not beneficial in the recent past. Over time, the tuner drops the less useful indexes to make space for building new indexes that are more useful during query processing. Even after dropping an index, the tuner's forecaster retains the model meta-data associated with that index. This allows it to predict that index's utility in future when it is requested by the action generator.

We chose to employ the Holt-Winters method because it quickly generates reliable index utility forecasts with minimal storage requirements. The tuner forecasts the utility of an index configuration containing 1000 indexes in 0.3 µs. We note that other lightweight function approximators can be used for this task [21, 54].

Tuning Refinements: Although we designed the tuning algorithm to be lightweight, it still imposes some overhead on the performance of the DBMS. The tuner, therefore, uses the classifier to determine the nature of workload, and when the load on the database server is high or while processing a write-intensive workload, it automatically reduces its *tuning frequency* [11]. We defer the discussion of this parameter to Section 5.2. When the tuning process is temporarily suspended, the DBMS still uses the partially-built indexes with the value-agnostic hybrid scan operator during query processing. Later, when the tuning process resumes, the tuner finishes constructing these indexes. We next describe the design of the benchmark that we use for evaluating predictive indexing.

5 TUNER BENCHMARK

We developed a new open-source benchmarking suite to evaluate the efficacy of predictive indexing with respect to other online indexing approaches. The queries in this benchmark are derived from a HTAP workload in a bus tracking mobile application [28]. The TUNER benchmark extends prior physical design tuning benchmarking suites proposed in [5, 6, 24, 40, 47] in three ways. It examines the impact of the index tuner under both regular and heavy system loads, when the tuner throttles its tuning frequency to cope with workload spikes. Second, it evaluates the impact of automatically adapting both the storage layout and the index configuration of the database in tandem, thus focusing on two key components of the physical design problem. Lastly, it characterizes the predictive index tuner's ability to leverage idle system resources.

The TUNER database contains two tables: a narrow table and a wide table. Each table contains tuples with a timestamp (a_0) and p integer attributes ($a_1, \ldots, a_p$), each 4 bytes in size. The narrow table has $p = 20$ attributes, and the wide table has $p = 200$ attributes. Each attribute a_k is an integer value from a Zipf distribution in the range [1, 1m]. The benchmark can configure the selectivity of the predicates in a query while constructing multi-attribute *range* and *equality predicates* present in the mobile application's HTAP workload. We next describe our methodology for constructing these queries that filter and aggregate the dataset based on different attribute sets.

5.1 Query Generation

We construct workloads using two kinds of queries: (1) scan queries and (2) update queries.

Scan Queries: We use three `SELECT` templates to construct queries with different levels of workload complexity:

- **Low-Complexity Scan (LOW-S):** It computes aggregates over a single table after applying a *comparison predicate* defined on a single attribute. This query template represents the simplest scenario for an index tuner.

```
SELECT a1, a2 + a3, ..., SUM(ak) FROM R
 WHERE ai ≥ δ1 AND ai ≤ δ2
```

- **Moderate-Complexity Scan (MOD-S):** It is similar to the previous template except that the comparison predicate is now defined on a combination of attributes. The tuner must build multi-attribute indexes to accelerate such queries.

```
SELECT a1, a2 + a3, ..., SUM(ak) FROM R
 WHERE ai ≥ δ1 AND ai ≤ δ2 AND aj ≥ δ3 AND aj ≤ δ4
```

- **High-Complexity Scan (HIGH-S):** Unlike the previous query templates, this query contains an equi-join operation over two relations in addition to the multi-attribute comparison predicates. The query would, therefore, benefit from the presence of indexes defined on the attributes involved in the *join predicate*.

```
SELECT X.a1, ..., X.ak, Y.a1, ..., Y.ak
  FROM X, Y
 WHERE X.ai ≥ δ1 AND X.ai ≤ δ2 AND Y.aj ≥ δ3
   AND Y.aj ≤ δ4 AND X.al = Y.am
```

We vary the selectivity and projectivity of the predicates in the scan queries by altering δ_1, δ_2, and k.

Update Queries: We use three different query templates to construct updates and inserts that correspond to different levels of workload complexity. These include:

- **Low-Complexity Update (LOW-U):** It updates a random subset of attributes for tuples that satisfy a comparison predicate defined on a single attribute.

```
UPDATE R SET a1=v1, a2=v2, ..., ak=vk + 1
 WHERE ai ≥ δ1 AND ai ≤ δ2
```

- **High-Complexity Update (HIGH-U):** It is similar to the low-complexity update query except that the comparison predicate is now defined on a combination of attributes.

```
UPDATE R SET a1=v1, a2=v2, ..., ak=ak + 1
 WHERE ai ≥ δ1 AND ai ≤ δ2 AND aj ≥ δ3 AND aj ≤ δ4
```

- **Insert (INS):** It inserts a set of tuples into the table. Unlike the UPDATE statement that involves both table scan and modifications, INSERT statement only performs table modifications.

```
INSERT INTO R VALUES (a0, a1, ..., ap)
```

5.2 Workload Generation

We use the above queries to construct a set of workloads that exercise different aspects of predictive indexing. A workload is a sequence of queries with changing properties. To clearly delineate the impact of the index configuration on the different queries, we divide a sequence into *phases* consisting of a fixed number of queries that each correspond to a particular query type. This means that the DBMS executes the same query type (with different input parameters) in one phase, and then switches to another one in the next phase.

Tuning Frequency: The benchmark examines the tuner's impact when it only operates during certain time periods. The tuner usually runs frequently, and is likely to react quicker to short-lived workload shifts. In contrast, when the load is high, it runs less frequently and is likely to treat these ephemeral shifts as noise, and demonstrates gains only on longer phases. The benchmark examines the tuner's impact when it operates under four different frequencies: FAST, MOD, SLOW, and DIS. With the FAST, MOD, and SLOW configurations, the tuner runs on average once every 100, 1000, and 10000 ms. Index tuning is disabled with DIS, and serves as a baseline.

Shifting Workloads: We construct shifting workloads of different phase lengths to examine how swiftly can the tuner adapt to these shifts. If the total number of queries in a workload is t, and the phase length is l, then the workload is comprised of $\frac{t}{l}$ phases. The performance impact of tuning depends on the average phase length of the workload. Stable workloads with longer phases benefit more from tuning. This is because after the tuner observes a set of queries based on a query template, it builds an index to accelerate query processing. Subsequent queries of the same template would then benefit from this index.

Hybrid Workloads: To evaluate the ability of the tuner to handle both scan and update queries, we construct hybrid workloads wherein the tuner must factor in both the query processing utility and the maintenance cost of indexes. We construct four types of workload mixtures that vary the operations that the DBMS executes. These mixtures represent different ratios of scans and updates:

- **Read-Only:** 100% *scans*
- **Read-Heavy:** 90% *scans*, 10% *updates*
- **Balanced:** 50% *scans*, 50% *updates*
- **Write-Heavy:** 10% *scans*, 90% *updates*

On the read-intensive mixtures, the DBMS benefits from the construction of ad-hoc indexes that accelerate query processing. The benefits are not as pronounced on the write-intensive mixtures due to index maintenance costs. We construct the updates using the low-complexity and high-complexity UPDATE query templates, and configure the scans to be low-complexity queries.

6 EXPERIMENTAL EVALUATION

We now present an analysis of our predictive indexing approach using DBMS-X [29]. DBMS-X is an in-memory MVCC DBMS that supports HTAP workloads. We integrated our index tuner as a background thread that runs periodically depending on the tuning frequency. It decides both when and how to adapt the index configuration. During every tuning cycle, it classifies the workload, generates and applies physical design changes, and learns the utility of these changes. We extended the system's execution engine and query optimizer to support the value-agnostic hybrid scan operator. To perform a comparative evaluation, we integrated different indexing approaches in DBMS-X, including (1) online [12, 48], (2) adaptive [23, 31], (3) self-managing [56], and (4) holistic [41].

We deployed DBMS-X on a dual-socket Intel Xeon E5-4620 server running Ubuntu 14.04 (64-bit). Each socket contains eight 2.6 GHz cores. It has 128 GB of DRAM and 20 MB of L3 cache. For each experiment, we execute the workload five times and report an average of the metrics. All transactions are executed with the default snapshot isolation level. We disable logging to ensure that our measurements only reflect query processing time. Each table in the TUNER database contains 10m tuples, and the total size of the database is 8.8 GB.

We begin with an analysis of the ability of the tuner to recognize workload trends using its forecaster. We then investigate the benefits of using hybrid scan to leverage partially built indexes. Next, we compare our tuner against the holistic indexing approach on a HTAP

workload. We then demonstrate how the index tuner works seamlessly with the storage layout tuner in DBMS-X. Next, we present an analysis of the tuner's ability to adapt to shifting workloads of different phase lengths. We then show that the index tuner can converge to an optimal index configuration without human intervention. Lastly, we examine the impact of workload variability and index storage space constraints on the DBMS's performance.

6.1 Decision Logic

We use a HTAP workload to examine the ability of the tuner to recognize workload trends and adapt the index configuration ahead of time. This workload contains 5000 moderate complexity scan queries with a phase length of 500 queries. We throttle the client request throughput at the beginning of each phase to evaluate the tuner's ability to leverage idle system resources while it employs different types of decision logic: (1) retrospective DL, (2) predictive DL, and (3) immediate DL. We note that the index tuner must construct multi-attribute indexes to accelerate the processing of these scan queries. Most of the queries contain the same multi-attribute predicate, and will thus benefit from the same index. A small fraction (1%) of the queries are noisy queries containing other predicates. Ideally, the tuner should recognize that it is not useful to build indexes to accelerate such queries. We drop all the built indexes at the end of each phase. This is meant to model a diurnal workload where indexes have to be rebuilt every morning.

The key observation from Figure 6 is that with predictive DL, the tuner's forecaster captures the query pattern after observing 1500 queries. It then makes use of idle system resources at the beginning of each phase to build the index ahead of time, thus allowing the DBMS to start using the index earlier within the phase. With retrospective DL, the tuner only examines the recent k queries in the same phase, and does not proactively build indexes. This increases the tuner's detection time, and it finishes index construction only later in the phase. Retrospective DL still guards against noisy queries. In contrast, with immediate DL, the tuner constructs indexes to accelerate these noisy queries, thereby slowing down the overall index construction speed and increasing the adaptation time. The cumulative time taken by the DBMS to execute this workload with predictive DL is 5.2× and 3.5× shorter than that taken with the retrospective and immediate schemes, respectively. This experiment shows that predictive DL can both shrink the detection time of the tuner and guard against noisy queries.

6.2 Hybrid Scan

We next examine the performance impact of different approaches for leveraging a partially-built index: FULL, VAP, and VBP. We consider three workloads with varying *affinity levels* with respect to the index sub-domains accessed by the queries: (1) very high, (2) high, and (3) moderate levels. Queries in workloads with higher affinity levels target a smaller number of sub-domains. The workloads contain 5000 moderate complexity scan queries with a phase length of 500 queries. We configure the number of different sub-domains accessed by the queries to be 2, 5, and 10 to create workloads with varying affinity levels. We construct a variant of VBP that prevents latency spikes by decoupling index construction from query processing. With this scheme, the tuner incrementally populates entries associated with

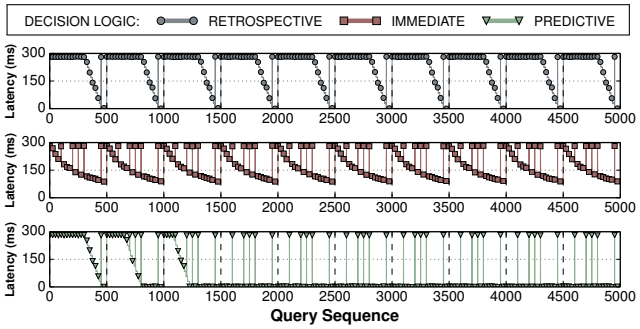


Figure 6: Decision Logic – Reaction times of the index tuner when using different types of decision logic for a HTAP workload.

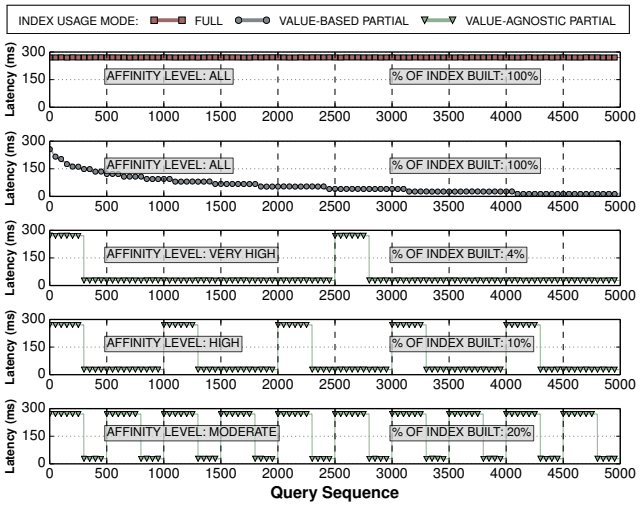


Figure 7: Hybrid Scan Operators – Comparison of the performance impact of the different approaches for leveraging partially-built indexes.

a sub-domain over several tuning cycles, instead of immediately adding all the entries while processing the query.

The results shown in Figure 7 indicate that unlike VBP, the behavior of VAP and FULL is unchanged on workloads with different affinity levels. This is because with VBP, the DBMS can leverage the partially-built index only when the sub-domain accessed by the query has been already indexed. But VAP enables the DBMS to make use of the partially-built index irrespective of the sub-domain accessed by the query.

The cumulative time taken to execute the very high affinity level workload with VAP is 1.05× longer than that taken with VBP. This is because the VBP tuner populates the frequently accessed sub-domains faster than with VAP. However, on workloads with high and moderate affinity levels, VAP outperforms VBP by 1.7× and 3.1×, respectively. This is because, unlike VBP, VAP can accelerate query processing even on workloads with lower affinity levels. We observe that with VBP and FULL, the index is fully populated near the end of the workload. In contrast, with VAP, only 4%, 10%, and 20% of the index is built depending on the number of different sub-domains accessed in the workload. The key observation from this experiment is that VAP allows the DBMS to leverage the index even when the sub-domain accessed by the query is not yet completely indexed.

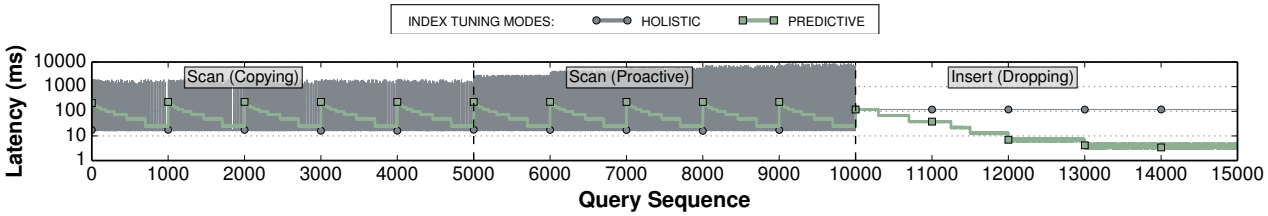


Figure 8: Holistic Indexing – Comparison of the performance impact of the holistic and the predictive indexing approaches on a HTAP workload.

6.3 Holistic Indexing

We now demonstrate how predictive indexing improves the performance of the DBMS in comparison to holistic indexing. Holistic indexing is an always-on VBP scheme where the tuner optimistically chooses which indexes to build next and then populates them when the DBMS is idle [41]. Although predictive indexing shares many of the goals of holistic indexing, it differs in three ways. It employs a value-agnostic hybrid scan operator instead of a value-based one. It adopts a predictive DL instead of an immediate DL. Lastly, it decouples index construction from query processing. For this experiment, we implemented holistic indexing with a random index selection strategy in DBMS-X and compare it against our approach [41]. We run a workload consisting of three *segments*, each of which contains 5000 queries based on multiple query templates. The first two segments consist of scan queries of moderate complexity, while the last one contains insert queries.

The results in Figure 8 show that on the first segment with holistic indexing, there are latency spikes that are as high as 4× the latency of a table scan. When the tuner encounters a query accessing a subdomain that has not yet been indexed, it immediately starts indexing it while processing the query [31]. Although this approach accelerates the execution of subsequent queries with the same template, it involves heavyweight physical design changes. In contrast, we do not observe such spikes with predictive indexing, since it amortizes the index construction overhead across several tuning cycles.

To leverage idle system resources, holistic indexing advocates a proactive approach towards building indexes even on attributes that have not been queried yet [41]. This can, however, increase the overall index construction overhead, and thereby cause latency spikes during subsequent query processing. This is illustrated during the second scan segment in Figure 8. Predictive DL mitigates this problem by populating indexes on attributes only after observing several queries that access them.

Lastly, on the third segment comprising of insert queries, the predictive tuner's classifier detects the workload shift, and its action generator periodically drops indexes of limited utility over time. This shrinks the insert query's latency, as the DBMS needs to update fewer indexes. The holistic tuner does not drop any indexes since (by design) it drops them only when they exceed its storage budget. The cumulative time taken to execute this workload with predictive indexing is 7.7× shorter than that taken with holistic indexing.

6.4 Storage Layout & Index Tuning

This experiment examines the ability of the index tuner to work with other physical design tuners. We investigate the impact of the index tuner when used in tandem with the storage layout tuner. This

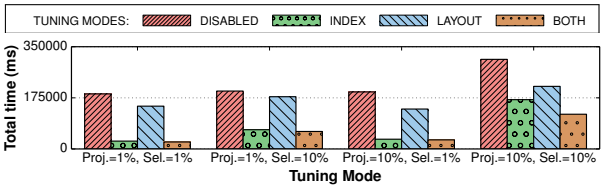


Figure 9: Storage Layout & Index Tuning – The impact of index tuning in tandem with storage layout tuning on the query processing performance.

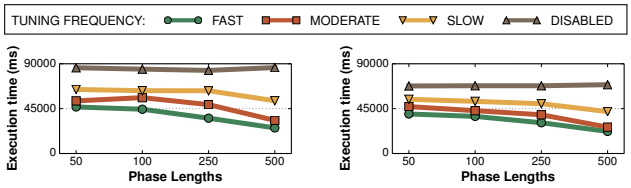
layout tuner accelerates query execution by co-locating the attributes accessed together in a query in memory [5, 6]. This enables the DBMS to make better use of memory bandwidth by fetching only the relevant attributes during query processing. We evaluate four tuning modes to measure the impact of index and layout tuning on the system's performance:

- **Disabled:** Both index and layout tuning are disabled.
- **Index:** Only index tuning is enabled.
- **Layout:** Only layout tuning is enabled.
- **Both:** Both index and layout tuning are enabled.

We consider the behavior of the layout and index tuners under different projectivity and selectivity settings on a read-only workload comprising of moderate-complexity scan queries on the wide table. We vary both the projectivity and selectivity of scan query from 1% to 10%. We measure the total time taken by the DBMS to execute the workload, including the time spent on tuning the layout and index configuration of the database.

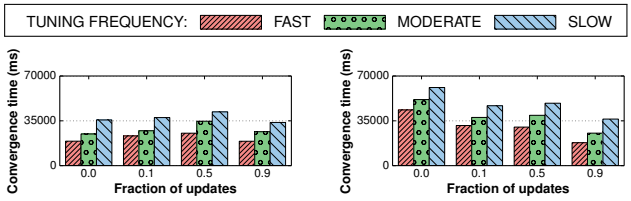
The results in Figure 9 show that the index and layout tuners together are able to reduce the workload's execution time more than that what they accomplish independently. Under high projectivity and selectivity settings, we observe that while index and layout tuning independently speed up query execution by 1.9× and 1.5×, respectively, they reduce query execution time by 2.7× when used in tandem. The layout tuner incrementally morphs the table to a hybrid storage that collocates the query's projection attributes. On average, the layout tuner takes 2.6 ms to transform the layout of a page containing 1000 tuples. Concurrently, the index tuner incrementally builds indexes to quickly retrieve the matching tuples. During one iteration of Algorithm 1, the index tuner takes on average 5.2 ms to populate a set of indexes with entries associated with a page.

The impact of the tuners is more prominent under low projectivity and selectivity settings, as shown in Figure 9. In this case, index and layout tuning together shrink the execution time by 7.8×. We attribute this to the higher gains from index and layout tuning when 1% of the attributes and 1% of the tuples are respectively projected



(a) Read Only (b) Write Heavy

Figure 10: Tuner Adaptability – The impact of the index tuning on the query processing time under different phase length settings. The execution engine runs different workload mixtures, each comprising of scan queries of varying complexity, on the narrow table.



(a) Low-Complexity Scan (b) High-Complexity Scan

Figure 11: Tuner Convergence – The impact of query complexity and workload mixture on the convergence time with different configurations.

and selected from the table. This highlights the importance of continuously improving the physical design of the DBMS using many small incremental steps.

6.5 Tuner Adaptability

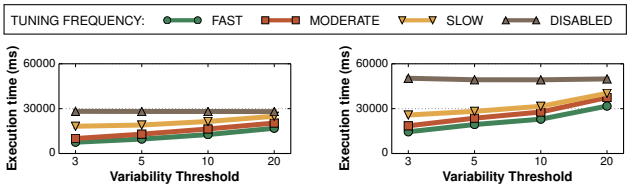
We next analyze how the tuner adapts to shifting HTAP workloads. We examine the impact of the tuner when operating under four different tuning frequencies: FAST, MOD, SLOW, and DIS. We consider two workload mixtures, each comprising of 5000 queries, and we vary the phase length of the workload from 50 to 500 queries.

Figures 10a and 10b show the results for the read-only and write-heavy workloads under different phase length settings. The most notable observation is that the benefits are greater on workloads with longer phases. When the phase length is 500 queries, FAST outperforms DIS by 3.4×. This is because after the tuner constructs the appropriate index in a phase, all the subsequent queries within the same phase benefit from the index. We observe that MOD and SLOW outperform DIS by 2.6× and 1.6×, respectively. The impact of the tuner is less prominent under these configurations because it takes more time to build the indexes due to lower tuning frequencies.

6.6 Tuner Convergence

This experiment evaluates how long it takes for the tuner to converge to an optimal index configuration for a stable workload whose queries belong to a finite set of query templates. We refer to the duration of time from when the DBMS starts executing the stable workload to when the tuner reaches an optimal index configuration as its *convergence time*. Our goal here is to understand the impact of the tuning frequency, the query complexity, and the workload mixture on the convergence time. We construct a stable workload by restricting the number of query templates to 10. This limits the number of useful ad-hoc indexes that the tuner must build.

Figure 11 shows the results for workload mixtures comprising of low and high-complexity scan queries. We found that the tuner



(a) Low-Complexity Scan (b) High-Complexity Scan

Figure 12: Workload Variability – The impact of workload variability and complexity on performance with different tuner configurations.

automatically converged to an optimal index configuration for all the stable workloads, thus obviating the need for manual tuning. The most notable observation is that the convergence time on the read-heavy workload comprising of high-complexity scan queries is 1.7× higher than that it takes on the workload containing low-complexity queries. This is because the tuner must additionally build indexes on the attributes present in the join predicate of the high-complexity queries. For the read-only workload in Figure 11a, we observe that the tuner converges 2.1× and 1.4× faster with FAST than with SLOW and MOD, respectively. We attribute this to the higher tuning frequency of the FAST configuration. We observe that the convergence time gap between the configurations is more prominent on the write-intensive workloads.

6.7 Workload Variability

We now analyze the impact of workload variability on the query processing performance with different tuner configurations. TUNER uses a set of query templates to construct the workload. We refer to the size of this set as the *variability threshold* (v). Larger values for v correspond to workloads that have a higher degree of variability. We vary v from 3 to 20 query templates. We expect that the challenge for the tuning algorithm increases on more varied workloads. In this experiment, we examine read-only workloads containing 5000 low- or high-complexity scan queries with a phase length of 250 queries.

Figures 12a and 12b show the time taken by the DBMS to execute the different workloads. We observe that when v is 20 in Figure 12a, FAST outperforms DIS by 1.6×. This gap expands to 3.7× when v is 3. This is because the tuner quickly converges to a stable configuration on workloads with smaller v. The impact of index tuning is, thus, more prominent on stabler workloads. Another notable observation is that on the high-complexity workload in Figure 12b, the gap between FAST and DIS is more significant across different v settings. This highlights the importance of index tuning for workloads comprising of high-complexity scan queries. The FAST configuration is more sensitive to workload variability than SLOW. When v is increased from 3 to 20, the execution speedup of FAST drops from 3.7× to 1.6×. However, SLOW only drops from 1.7× to 1.1×. This shows that the benefits of frequent tuning are more pronounced on stabler workloads due to quicker convergence.

6.8 Index Storage Space

In this experiment, we investigate how constraining the storage space available for maintaining indexes impacts the behavior of the FAST configuration. We construct a read-only workload comprising of 5000 moderate-complexity scan queries and configure the phase

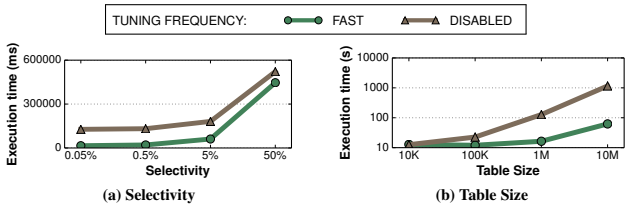


Figure 13: Query Selectivity & Table Size – The impact of query selectivity and table size on the query processing performance.

length to 250 queries. We restrict the size of the set of query templates to 10. We vary the storage budget from 2 GB to 6 GB. This limits the number of indexes that can be constructed at a time in memory. While processing the workload, the tuner periodically drops the indexes with lower utility in order to satisfy the index storage space constraint. We found that the total execution time, when the storage budget is 2 GB, shrinks by 1.3× and 2.1× when the budget is increased to 4 GB and 6 GB, respectively. This is because the tuner leverages the additional storage space available under higher storage budget settings to build indexes that accelerate query processing.

6.9 Query Selectivity & Table Size

Lastly, we examine the benefits of index tuning when varying the selectivity of the low-complexity scan query from 0.05% through 50% on a table containing 1M tuples. Figure 13a presents the execution time on the read-only workload with the FAST and DIS configurations. We observe that the gap between FAST and DIS expands from 1.2× to 8.5× when we decrease the selectivity from its highest setting to its lowest. This is because under lower selectivity settings, the performance of the index scan and table scan operators diverge.

We next configure the query selectivity to 0.5% and vary the size of the table from 10k to 10m tuples. The results in Figure 13b show that the gap between the FAST and DIS configurations expands from 7.9× to 18.6× when the table size is increased from 1m tuples to 10m tuples. This shows that the benefits of predictive index tuning are more prominent on larger tables.

7 RELATED WORK

The design of predictive indexing benefited from prior work on online index tuners, offline index advisors, partial indexes, self-managing indexing, holistic indexing, and storage layout tuning.

Online Index Tuners: The problem of automatically tuning the index configuration of a database to improve its performance on evolving query workloads has been studied for several decades [26]. More recently, Bruno and Chaudhuri present an online algorithm that dampens physical design oscillations and takes index interactions into consideration [12]. COLT is an online index-tuning framework that dynamically lowers its overhead when the system has converged to an optimal index configuration [48]. All of this work present tuners that leverage the indexes during query processing only after fully populating them. In contrast, DBMS-X uses the index with the hybrid scan operator even before it is fully built, thereby improving its reaction time.

Offline Index Advisors: There has been considerable research on *offline* techniques for tuning the index configuration in DBMSs [15, 20, 44, 46]. Offline index advisors can be classified into two types: those that use an explicit cost model to estimate the overall utility of an index configuration [16, 26], and those that rely on the optimizer's estimates [15]. This work focuses on online index tuning.

Partial Indexes: A *partial index* is an unclustered index built over a subset of a table, where the subset is defined by a conditional expression [52]. We focus on full indexes, that unlike partial indexes, cover all the tuples in the table. The query optimizer can use a partial index only when it can determine that queries access a strict subset of the index. However, there are no such constraints associated with performing a hybrid scan over a partially-built index.

Self-Managing Indexing: Self-managing indexing (SMIX) is a VBP scheme that can dynamically expand and shrink indexes based on the workload [56]. Unlike adaptive indexing that incrementally converges to a full index, SMIX also shrinks the index by dropping less frequently accessed entries. However, SMIX only refines indexes during query processing and adopts an immediate DL. This can cause latency spikes and increases the reaction time. Unlike predictive indexing, it does not support range queries. Smooth scan is an adaptive scan operator which dynamically morphs between the index and table scan operators at runtime [9]. This technique is orthogonal to our static hybrid scan operator.

Holistic Indexing: With holistic indexing, the tuner optimistically makes an educated guess about which indexes should be built next and populates them using idle system resources [22, 23, 31, 35, 41]. Although predictive indexing shares many of the goals of holistic indexing, it differs in three ways. It employs a value-agnostic hybrid scan operator instead of a value-based one. It adopts a predictive DL instead of an immediate DL. Lastly, it decouples index construction from query processing.

Storage Layout Tuning: H_2O is a hybrid system that dynamically adapts the storage layout [5]. It maintains the same data in different storage layouts to improve the performance of read-only workloads through multiple execution engines [3, 4, 25, 27, 32, 34, 43, 50]. It combines data reorganization with query processing. In contrast, DBMS-X uses a single execution engine to process data stored in different storage layouts. It performs data reorganization in the background to prevent query latency spikes, and stores a given page in only one storage layout to reduce the synchronization overhead on HTAP workloads.

8 CONCLUSION

This paper presented predictive indexing that uses reinforcement learning to predict the utility of indexes, and continuously refines the index configuration of the database to handle evolving HTAP workloads. We proposed a lightweight value-agnostic hybrid scan operator that allows the DBMS to leverage partially-built indexes during query processing. Our evaluation showed that the predictive index tuner learns over time to choose physical design changes that increase the overall utility of the index configuration. We demonstrated that the index and storage layout tuners in DBMS-X work in tandem to incrementally optimize two key components of the database's physical design without requiring any manual tuning. Our evaluation showed that predictive indexing improves the throughput of DBMS-X on HTAP workloads by 3.5–5.2× compared to other state-of-the-art indexing approaches.

REFERENCES

[1] Zipf Distribution. http://mathworld.wolfram.com/ZipfDistribution.html.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, 2006.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[4] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[5] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, 2014.

[6] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *SIGMOD'16*.

[7] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The rum conjecture.

[8] A. Bog, H. Plattner, and A. Zeier. A mixed transaction processing and operational reporting benchmark. *Information Systems Frontiers*, 13(3):321–335, July 2011.

[9] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth scan: Statistics-oblivious access paths. In *ICDE*, pages 315–326, 2015.

[10] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[11] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.

[12] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[13] S. Chaudhuri and V. Narasayya. AutoAdmin "what-if" index analysis utility. In *SIGMOD*, pages 367–378, 1998.

[14] S. Chaudhuri and V. Narasayya. Microsoft index turning wizard for SQL Server 7.0. In *SIGMOD*, pages 553–554, 1998.

[15] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[16] S. Choenni, H. M. Blanken, and T. Chang. On the automation of physical database design. In *Symposium on Applied Computing*, pages 358–367, 1993.

[17] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *SIGMOD*, pages 1923–1934, 2016.

[18] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *VLDB*, 7(4):277–288, 2013.

[19] S. Elnaffar, P. Martin, and R. Horman. Automatically classifying database workloads. In *CIKM*, pages 622–624, 2002.

[20] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM TODS*, pages 91–128, 1988.

[21] K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3), 1989.

[22] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *ICDEW*, pages 69–74. IEEE, 2010.

[23] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[24] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2010.

[25] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. In *VLDB*, pages 105–116, 2010.

[26] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, 1976.

[27] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, 2003.

[28] Hidden. Retracted due to double-blind constraints, .

[29] Hidden. Retracted due to double-blind constraints, .

[30] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5 – 10, 2004.

[31] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[32] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence*, Lecture Notes in Business Information Processing, 2012.

[33] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *J. Artif. Int. Res.*, pages 237–285, 1996.

[34] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[35] M. L. Kersten, S. Manegold, et al. Cracking the database store. In *CIDR*, volume 5, pages 4–7, 2005.

[36] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB*, 2011.

[37] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[38] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[39] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, 2015.

[40] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *VLDB*, 2017.

[41] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, pages 1153–1166, 2015.

[42] M. Pezzini, D. Feinberg, N. Rayner, and R. Edjlali. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/doc/2657815/, 2014.

[43] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 430–441, 2002.

[44] S. Rozen and D. Shasha. A framework for automating physical database design. In *VLDB*, pages 401–411, 1991.

[45] K.-U. Sattler, I. Geist, and E. Schallehn. Quiet: Continuous query-driven index tuning. In *VLDB*, pages 1129–1132, 2003.

[46] K. B. Schiefer and G. Valentin. DB2 universal database performance tuning. *IEEE Data Eng. Bull.*, 22:12–19, 1999.

[47] K. Schnaitter and N. Polyzotis. A benchmark for online index selection. In *ICDE*, pages 1701–1708, 2009.

[48] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous on-line tuning. In *SIGMOD*, pages 793–795, 2006.

[49] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[50] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *SIGMOD*, pages 731–742, 2012.

[51] M. A. Soliman et al. Orca: a modular query optimizer architecture for big data. In *SIGMOD*, pages 337–348, 2014.

[52] M. Stonebraker. The case for partial indexes. *ACM SIGMOD Record*, 18(4):4–11, 1989.

[53] G. Valentin, M. Zuliani, and D. C. Zilio. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[54] V. Vapnik, S. E. Golowich, and A. J. Smola. Support vector method for function approximation, regression estimation and signal processing. In *NIPS*, 1997.

[55] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., 2001.

[56] H. Voigt, T. Kissinger, and W. Lehner. SMIX: self-managing indexes for dynamic workloads. In *SSDBM*, pages 24:1–24:12, 2013.

[57] P. R. Winters. Forecasting sales by exponentially weighted moving averages. *Management Science*, 6(3):324–342, 1960.

[58] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10:781–792, March 2017.

[59] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *SIGMOD*, pages 1567–1581, 2016.

[60] 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 *VLDB*, pages 1087–1097, 2004.