An Experimental Study of Online Database Index Selection using Metrical Task Systems

Tanzid Sultan

Project Supervisors:

Dr. William Umboh
Dr. Junhao Gan
Dr. Renata Borovica-Gajic

COMP90055: Research Project

October 2024

Abstract

The emergence of *Big Data* has led to larger and more complex databases for which human administration is untenable. Indexes play a crucial role in the efficient operation of such databases, making automated index selection tools a necessary component of the database management system. In this research project, we leverage an existing approach called **WFIT**, based on *Metrical Task Systems* (MTS) and offering a strong theoretical performance guarantee, to solve the online index selection problem. Our main contribution includes extending the original WFIT algorithm by replacing it's use of the database system's *internal cost-model* with an efficient and accurate *external cost-model*, thus addressing a major weak-point. We also carry out a set of benchmarking experiments demonstrating the validity of our approach and confirming the improvements brought on by our extension of WFIT. Additionally, we also demonstrate the extended WFIT algorithm's ablility to strongly compete against a current state-of-the-art online index selection technique based on *Multi-Armed Bandits*.

Declaration

I certify that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- Where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School.
- The thesis is 9394 words in length (excluding text in images, table, bibliographies and appendices).

Tanzid Sultan

October 28, 2024

Acknowledgments

I am deeply grateful to my supervisors William Umboh, Junhao Gan and Renata Borovica-Gajic for their continued support and guidance throughout this project and for everything they have taught me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Need for Online Index Selection

In today's era of *Big Data*, organizations across various sectors - such as finance, healthcare, social media and e-commerce - rely on vast amounts of data to carry out their daily operations and for strategic planning and decision-making. A Database Management System (DBMS) is a software system designed to robustly store and manage such collections of data. A DBMS also provides an interface that allows end-users to query and update the data. The speed and efficiency with which a DBMS is able to process user queries/updates depends critically on the *physical design structures* (PDS) available within the system. These are data structures which specify how the data is stored on disk, retrieved and updated. We will restrict our attention to databases which follow the *relational model* [37] in which the data is organized across multiple *tables*[1] related to each other through *keys*.

A PDS on each table primarily consist of *heap files*, in which the data is unsorted, and *clustered indexes* in which the data is sorted according to a *primary key* which uniquely identifies the data records. These PDS do not efficiently support most queries which are based on columns other than the primary key. This is where *secondary/nonclustered indexes*[2] come into play . A database index serves a similar purpose as an index at the end of a book,

[1] the words *table* and *relation* are often used interchangeably

[2] Secondary indexes come in various types, such as B^+-tree index, hash index, etc. We will restrict our attention to only B^+-tree indexes due to their efficiency, versatility and prevalence in most applications.

i.e. it allows us to quickly retrieve the specific rows we are interested in without performing a *full-table scan*. Because the main performance bottleneck in large-scale relational databases is disk I/O, an intelligently designed set of secondary indexes, also referred to as a *configuration*, can significantly reduce the amount of disk I/O needed to process a query, resulting in dramatic speedup.

Most DBMS do not automatically create secondary indexes. Instead, either an expert database administrator (DBA) needs to manually select and materialize a set of secondary indexes or a non-expert user can invoke *offline* index recommendation tools provided by the DBMS, such as the *Database Tuning Advisor* in Microsoft SQL Server [1]. Such offline selection/tuning methods require a representative workload to be provided based on which index recommendations are generated. This approach poses several challenges, particularly when workload patterns change unpredictably over time due to varying user demands or seasonal trends, or when the scale of the database changes due to data growth or increasing number of users. Such dynamic environments would necessitate periodically invoking the offline tuning process from scratch, each time with a new representative workload. These offline methods also don't take into consideration, overheads associated with generating recommendations and materializing the recommended indexes. This can lead to a situation where those overheads can cause significant delays, outweighing potential improvements in future query processing costs.

Given these challenges with manual/offline tuning, there is a strong motivation for developing **automated online index selection algorithms** which can continuously monitor evolving database workloads (and system environments) and adaptively adjust the index configuration based on observed patterns. More formally, we can frame the algorithm as solving a combinatorial optimization program, where the objective is to make a sequence of decisions, each decision corresponding to materializing and/or dropping a set of secondary indexes (i.e. changing configuration), such that the total running time of the entire workload and configuration changes is minimized over an indefinitely long horizon, subject to memory and/or time constraints.

1.2 Related Works

Substantial efforts have been made to address the need for online index tuning, resulting in a number of notable algorithms over the last few decades. A key component which is common across all of these different approaches is the use of a **cost-model** to estimate query execution time in any *hypothetical* index configuration[3]. This cost estimation is a necessary step because, at a high level, each of these approaches perform an *exploration* of some part of the space of all possible configurations to find an optimal configuration according to some specified criterion. Beyond the commonality, these algorithms can be broadly divided into **two categories**, based on the nature of the cost-model.

In the **first category**, we have algorithms that rely on the **internal cost-model**[4] of the database system's query optimizer, such as [5, 23, 31, 33, 35, 43]. In addition to using this internal cost-model, some of these algorithms, such as [5], are also tightly coupled to the DBMS query optimizer and are *invasive* in the sense that they intercept the query plan at various stages of query optimization and make local plan transformations to enumerate and explore a large number of alternative hypothetical configurations and their corresponding estimated costs. Among all approaches in the first category, the WFIT algorithm of [33], which uses a variant of the *work function algorithm* from *metrical task systems* (MTS) [3], is most notable due to its theoretical performance guarantee in the form of a bounded *competitive ratio*. To our knowledge, [23] and [33] are the only two works which have applied MTS to the online index selection problem and provided proof of bounded competitive ratio[5], with the latter providing a substantially tighter bound.

In recent years, there has been a shift towards using learning-based approaches for index selection such as [2, 8, 11, 26, 28, 27, 36, 47]. This brings us to the **second category** of algorithms which use an *external cost-model*, which learns directly from observed data instead of relying on the DBMS what-if interface. [2] and [11] take a reinforcement learning (RL) approach, modeling queries and updates as a Markov decision process (MDP) with states represented by the index configurations, actions represented by configuration

[3] a hypothetical configuration is one that has not been materialized

[4] the internal cost-model can be accessed via a *what-if* interface [7]

[5] [5] also proves a constant competitive ratio for the special case of a single-index configuration space, however their analysis does not extend to the general multi-index case

changes and rewards represented by query execution and configuration change costs. [2] uses least-squares policy iteration while [11] applies deep Q-learning to solve the MDP. On the other hand, [26, 27, 28] uses a different kind of RL approach in the form of stateless Multi-Arm Bandits (MAB) where bandit actions/arms are represented by configuration changes. Among all the algorithms in the second category, the MAB particularly stands out due to its theoretical performance guarantee in the form of a sub-linear *regret* bound and has been shown in experiments to have superior performance compared to a state-of-the-art offline tuning tool provided by a commercial DBMS.

1.3 The Need for Theoretical Performance Guarantees

The complexity of performing cost-based search over an exponentially large space of candidate configurations contributes to the NP-hardness of the index selection problem, even in the offline setting [29]. In the online setting, we have the added difficulty where decisions have to be made given only past observations of the workload and system environment and without knowledge of the future. Due to these difficulties, there is a need for algorithms which offer **robust theoretical (worst-case) performance guarantees**, because such guarantees would protect an algorithm against arbitrary performance degradation and also ensure good performance under a wide range of scenarios.

In the study of online optimization problems, two very different performance metrics are typically used, each of them leading to very different algorithm design methodologies [41]:

- **Cumulative Regret** which is defined as the additive performance loss incurred by the online algorithm compared to an optimal offline algorithm[6]. It is desirable for an algorithm to have cumulative regret which is bounded by a function that grows sub-linearly with the number of decision rounds. So that on average, the algorithm's decision making ability improves gradually and it's performance approaches that of the optimal offline strategy over time.
- **Competitive Ratio** which is defined as the ratio of the online algorithm's worst-case performance to that of an optimal offline algorithm.

[6] An optimal offline algorithm has complete foresight of the input sequence.

It is desirable for the competitive ratio to be bounded and as close to 1 as possible which would indicate that the online algorithm is performing almost as well as the optimal offline algorithm, for arbitrary input sequences. The competitive ratio often tends to be a pessimistic measure of performance because it strictly considers worst-case scenarios, however this property also contributes to the overall robustness of the metric.

Performance guarantees are especially important for online index selection algorithms, for safeguarding against disastrous consequences brought on by selection of unfavorable indexes which may cause query response times to degrade by several orders of magnitude, which is untenable in a production environment. Surprisingly enough, our survey of the research literature revealed that only a small handful of existing online index selection techniques make any direct effort towards providing theoretical performance guarantees!

1.4 Research Goals and Contributions

The original impetus for this research project came from an idea for adopting existing techniques from a general framework for online algorithms known as *Metrical Task Systems* (MTS) [3] for the index selection task. As will be explained in Chapter 2, there exists a one-to-one correspondence between the online index selection problem and MTS. Moreover, a powerful deterministic algorithm possessing a bounded competitive ratio exists for solving the MTS problem, known as the *work function algorithm* (WFA) [3].

Indeed, this idea has been explored previously by [33] who contributed extensions to the existing WFA to make it computationally tractable for online index selection, culminating in their **WFIT** algorithm. Among the online index selection algorithms belonging to the category which relies on the database system's internal cost-model (following our categorization defined in Section 1.2), the WFIT algorithm is regarded as the state-of-the-art (SOTA), a position established through a range of benchmarking tests [18].

Even though WFIT has demonstrated strong performance, we have identified a major **weak-point** in the algorithm: it's **reliance on an internal cost-model**. It is widely known that the database query optimizer's internal

cost-model suffers from errors due to it's use of weak heuristics and incorrect assumptions about data distributions [21, 22]. This in turn could significantly degrade the quality of index recommendations made by the WFIT algorithm, specially for large and complex real-world datasets containing highly skewed data distributions. Additionally, invocation of the internal cost-model via the *what-if interface* is inefficient, which dramatically slows down the recommendation speed of the WFIT algorithm, as pointed out in benchmarking tests from [18].

Our research project aims to leverage the WFIT algorithm and make extensions targeted at alleviating its weak-points, with the hope of ultimately producing an online index selection algorithm which can rival the current SOTA MAB algorithm of [26, 28]. To achieve this overarching objective, we have set out two main goals for this project:

1. **Extend the WFIT algorithm by replacing the what-if interface with a lightweight external cost-model. This is our primary contribution.**
2. **Run benchmarking experiments to evaluate the performance of the extended WFIT against the SOTA MAB algorithm.**

The external cost-model can be specifically designed to avoid using weak heuristics and incorrect assumptions leading to greater accuracy in estimated costs, and can also be endowed with an ability to adapt online based on actual observations. Moreover, making the model lightweight can significantly accelerate the speed at which the WFIT algorithm generates index recommendations, improving its overall efficiency.

Both MAB and WFIT have individually demonstrated SOTA performance in prior benchmarking tests, however a direct comparison between the two has never been attempted. Given that the original incarnation of WFIT uses an internal cost-model, whereas MAB uses an external cost-model, our extended WFIT will put them on a more equal footing, and enable a direct comparison. It is also interesting to note that the MAB and WFIT algorithms each offer very different theoretical performance guarantees, the former with it's bounded cumulative regret and the latter with a bounded competitive ratio. It will be interesting to see how these differences may manifest across their performance aspects during our benchmarking experiments.

Chapter 2

Background

In this chapter, we start by formulating the online index selection problem, introducing relevant concepts, definitions and notation along the way. We then provide a brief overview of the WFA and original WFIT algorithm from [33], setting the stage for our extension of this algorithm, to be covered in Chapter 3.

2.1 Formulation of Online Index Selection Problem

We define a *workload* as the following sequence, $W = (q_1, q_2, ..., q_n)$, where each q_i is a SQL query[1]. Let I be the set of all secondary indexes which can be created on the database schema such that $\mathcal{S} = \{C \in 2^I \mid mem(C) \leq B\}$ is the set of all possible index configurations which satisfy a memory budget constraint. Starting from an initial configuration $s_0 \in \mathcal{S}$, the goal of an online index selection algorithm is to propose a *configuration sequence* $S = (s_0, s_1, ..., s_n)$, where $s_i \in \mathcal{S}$ is the configuration chosen for round i, which minimizes the *total workload cost*, defined as follows:

$$C_{tot}(W, S) = \sum_{i=1}^{n} C_{exe}(q_i, s_i) + C_{tr}(s_{i-1}, s_i) \tag{2.1}$$

where $C_{exe}(q_i, s_i)$ denotes the cost[2] of **executing** query q_i at round i, given that indexes in configuration s_i are materialized, and $C_{tr}(s_{i-1}, s_i)$ denotes the

[1] the workload can also defined as a sequence of *mini-workloads*, where each mini-workload is a batch of queries

[2] the word cost can be used interchangeably with elapsed time

cost of **transitioning** from configuration s_{i-1} to s_i. Thus, at the beginning of round i, the algorithm gets to observe the incoming query q_i, then generate a recommendation for transitioning into configuration s_i, and after completing the transition (which may involve materializing some new indexes and/or dropping existing ones), the query is executed. Note that at each round, a recommendation is made based solely on the workload observed up to that point in time and without any knowledge of the future workload.

2.2 Reduction to Metrical Task Systems

The total workload cost, as defined in Equation 2.2, suggests a reduction of the online index selection problem into a *metrical task system* (MTS), which is a general framework for online problems first introduced in [3].

In the MTS framework, a system can be in any **state** from a set of possible states $\mathcal{S}$. There is a transition cost associated with switching from state x to y which is given by a function $d(x, y)$ such that the pair $(\mathcal{S}, d)$ defines a *metric space* [25] (i.e. d is a *metric*). Then, given a sequence of *tasks* $T = (t_1, t_2, ..., t_n)$ and given the system is initially in state $s_0 \in \mathcal{S}$, the objective is to propose a sequence of states $S = (s_0, s_1, ..., s_n)$ which minimizes the *total cost of* T, defined as follows:

$$C_{tot}(T, S) = \sum_{i=1}^{n} t_i(s_i) + d(s_{i-1}, s_i) \tag{2.2}$$

where $t_i(s_i)$ denotes the cost of serving task t_i in state s_i. The tasks are revealed sequentially, and at each step, the algorithm must propose a state in which to serve the task, without any knowledge of future tasks. Thus, we see a one-to-one correspondence between MTS tasks-states and queries-configurations from the index selection problem.

An interesting fact is that any deterministic algorithm for solving the MTS problem has a competitive ratio which is lower bounded by $2|\mathcal{S}| - 1$. In other words, for any online algorithm $\mathcal{A}$ which solves the MTS problem by proposing a sequence of states $S_{\mathcal{A}}$ and an optimal offline algorithm **OPT** which gets to see the full sequence of tasks and therefore proposes the sequence of states $S_{\mathbf{OPT}}$ which achieves the true minimum total cost, we have the following

bound on the competitive ratio $\mu_{\mathcal{A}}$ of algorithm $\mathcal{A}$:

$$\mu_{\mathcal{A}} = \frac{C_{tot}(T, S_{\mathcal{A}})}{C_{tot}(T, S_{\mathbf{OPT}})} \geq 2|\mathcal{S}| - 1 \tag{2.3}$$

Moreover, there exists a simple deterministic online algorithm for solving the MTS problem, called the *work function algorithm* (WFA) which attains the optimal competitive ratio, i.e. the tightest possible bound $\mu_{\text{WFA}} \leq 2|\mathcal{S}| - 1$. These results have been proven in [3], however, the proof relies on function d being a metric. In the case of the online index selection problem, we see an "almost" direct correspondence with an MTS, with the only exception that the transition cost function C_{tr} is not a metric due to its asymmetric nature, i.e. $C_{tr}(s_{i-1}, s_i) \neq C_{tr}(s_i, s_{i-1})$, because materializing an index has a much higher cost than dropping it. Despite this exception, [33] proves[3] that the WFA, with an appropriate modification, is able to handle the case of an asymmetric transition cost function while still maintaining the optimal competitive ratio.

Thus, given the simplicity of WFA, its strong performance guarantee in the form of a bounded competitive ratio, and the fact that it is an optimal deterministic online algorithm for an MTS, even with asymmetric transition costs, it is highly desirable to leverage this algorithm for solving the online index selection problem.

2.3 The Work Function Algorithm

Intuitively, the WFA tries to *follow* an optimal offline algorithm, given only limited information. It does so by maintaining an internal state known as the *work function*. To make the intuition more concrete, we need to first define this work function. Given the initial state s_0, let $T_i = (t_1, t_2, ..., t_i)$ be the prefix of T containing the first i tasks. Then for each state $s \in \mathcal{S}$, the work function $w_i(s)$ is defined as the **minimum offline total cost of T_i starting from state s_0 and ending up in state s**. Then clearly the optimal offline total cost of T_i is given by $\min_{s \in \mathcal{S}} w_i(s)$ and an *optimal substructure* is present

[3] this proof rests on the fact that C_{tr} still satisfies the triangle inequality despite being asymmetric

in the problem, leading to the following recurrence relation:

$$\begin{aligned} w_i(s) &= \min_{x \in \mathcal{S}} \{w_{i-1}(s) + t_i(x) + d(x, s)\} \\ w_0(s) &= d(s_0, s) \end{aligned} \tag{2.4}$$

The WFA leverages on this recurrence relation to make decisions at each step, i.e. each time a new task arrives, deciding whether to stay in the same state or perform a transition, before executing the task. Supposing that the algorithm is in state s_{i-1} after processing T_{i-1}, then after arrival of task t_i, the algorithm moves to the state s_i using the following decision rule:

$$\begin{aligned} s_i = \arg\min_{x \in \mathcal{S}} &\{w_i(x) + d(s_{i-1}, x)\} \\ \text{s.t.} \quad & w_i(s_i) = w_{i-1}(s_i) + t_i(s_i) \end{aligned} \tag{2.5}$$

We can see how WFA tries to mimic the optimal offline algorithm in the following way: we know that the optimal offline algorithm will end up in some state $s_i^* = \arg\min_{x \in \mathcal{S}} w_i(x)$, whereas WFA ends up in the state $s_i = \arg\min_{x \in \mathcal{S}} \{w_i(x) + d(s_{i-1}, x)\}$, note the only difference being the added transition cost. So, even if WFA makes decisions which deviate from those of the optimal offline algorithm, its decision rule still ensures that it ends up in the best possible state in light of those sub-optimal decisions made so far.

The decision rule in Equation 2.5 applies only to the case where d is a metric. For the case of asymmetric transition costs, [33] shows that a slight modification is required where the transition cost term $d(s_{i-1}, x)$ is replaced with $d(x, s_{i-1})$, leading to the following:

$$\begin{aligned} s_i = \arg\min_{x \in \mathcal{S}} &\{w_i(x) + d(x, s_{i-1})\} \\ \text{s.t.} \quad & w_i(s_i) = w_{i-1}(s_i) + t_i(s_i) \end{aligned} \tag{2.6}$$

2.4 WFIT: Divide and Conquer via Stable Partitions

As shown in Algorithm 1, WFA requires evaluating the work function over an exponentially large configuration space $\mathcal{S} = 2^I$, making the computations intractable. To overcome this problem, [33] developed a scheme for partitioning the set of indexes I into smaller clusters $\{I^{(1)}, I^{(2)}, .., I^{(m)}\}$ called *stable par-*

Algorithm 1: WFA for Online Index Selection

Input: configuration set $\mathcal{S}$, work function array $\boldsymbol{w}$, next query in workload q_i, previous recommendation S_{i-1}

Output: recommended next configuration S_i

Function `WFA`($\mathcal{S}$, $\boldsymbol{w}$, q_i, S_{i-1})**:**
  Initialization: create arrays $\boldsymbol{w'}$ and $\boldsymbol{p}$;
  for *each configuration* $S \in \mathcal{S}$ **do**
    $\boldsymbol{w'}[S] \leftarrow \min_{X \in \mathcal{S}}\{\boldsymbol{w}[X] + C_{exe}(q_i, X) + C_{tr}(X, S)\}$;
    $\boldsymbol{p}[S] \leftarrow \{X \in \mathcal{S} \mid \boldsymbol{w'}[S] = \boldsymbol{w}[X] + C_{exe}(q_i, X) + C_{tr}(X, S)\}$;
  $\boldsymbol{w} \leftarrow \boldsymbol{w'}$;
  $S_i \leftarrow \arg\min_{S \in \boldsymbol{p}[S]}\{\boldsymbol{w}[S] + C_{tr}(S, S_{i-1})\}$;
  return S_i;

titions. Separate WFA instances can be solved independently on each stable partition $I^{(k)}$ with it's corresponding configuration space given by $\mathcal{S}^{(k)} = 2^{I^{(k)}}$. The recommended index configuration is then obtained by taking the union of the sets of indexes recommended by the separate WFA instances.

In addition to making the computations tractable, this divide and conquer approach to applying WFA also has the added advantage of significantly lowering the competitive ratio bound, from $O(2^{|I|})$ down to $O(2^{I_{max}})$ where $I_{max} = \max_k\{|I^{(k)}|\}$. For a more detailed description of the stable partitioning procedure, we refer the reader to [33, 34].

Algorithm 2: WFIT for Online Index Selection

Data: stable partitions $\{I^{(1)}, I^{(2)}, ..., I^{(m)}\}$,
configuration sets for stable partitions $\{\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, ..., \mathcal{S}^{(m)}\}$,
work function arrays for WFA instances $\{\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, ..., \boldsymbol{w}^{(m)}\}$

Input: next query in workload q_i, previous recommendations for WFA instances $S_{i-1} = \{S_{i-1}^{(1)}, S_{i-1}^{(2)}, ..., S_{i-1}^{(m)}\}$

Output: recommended next configuration for WFA instances $S_i = \{S_i^{(1)}, S_i^{(2)}, ..., S_i^{(m)}\}$

Function `WFIT`(q_i, S_{i-1})**:**
  for $k \leftarrow 1$ **to** m **do**
    $S_i^{(k)} \leftarrow$ `WFA`($\mathcal{S}^{(k)}$, $\boldsymbol{w}^{(k)}$, q_i, $S_{i-1}^{(k)}$);
  $S_i \leftarrow \{S_i^{(1)}, S_i^{(2)}, ..., S_i^{(m)}\}$;
  return S_i;

Chapter 3

The Extended WFIT Algorithm

The original formulation of WFIT [33] utilizes an *internal* cost-model, via the database system's what-if interface, and is therefore susceptible to significant inaccuracy and inefficiency. In this chapter, we describe our efforts towards addressing the first goal of our research project, which aims to extend the WFIT algorithm by replacing it's use of the what-if interface with an external cost-model, thus overcoming the aforementioned weakpoints of what-if.

3.1 Limitations of an Internal Cost-Model

At a high level, an online index selection algorithm generally arrives at a recommendation using some form of cost-based search, which involves exploring a large space of candidate configurations, then identifying a configuration which is expected to provide the most "benefit" in terms of minimizing the total workload cost. In essence, the benefit of each configuration is derived from the cost of executing the latest query given that the configuration is materialized (along with other information pertaining to previously observed queries, such as their execution costs).

Of course, the true cost of executing a given query can only be determined by actually executing that query and observing the cost, similarly the true cost of materializing an index can only be determined through observation. Clearly, it is then infeasible to determine the benefit of each candidate configuration based on true costs, since that would require materializing a very large number of configurations and executing the query for each. Thus, the only practical

alternative is to utilize a **cost-model** which can efficiently **estimate** the true cost for any hypothetical query and index configuration, without actually executing the query or materializing any indexes.

In section 1.2, we discussed how cost-models generally come in two flavors, internal vs. external, the former being the cost-model used by the database system's query optimizer accessed via a *what-if* interface, while the latter is a separate model independent from the query optimizer. While it is often convenient for most online index selection algorithms to make use of the internal cost-model, and indeed this has been the common practice, we also noted that over the years, there has been a gradual shift towards approaches relying on external cost-models (often learning-based). This transition has been motivated by some significant drawbacks of an internal cost-model.

The **query optimizer's cost estimates can be highly error-prone**. This is mainly due to the use of inaccurate or outdated table statistics, weak heuristics and incorrect assumptions about data distributions which can result in selectivity/cardinality estimation errors [21, 22]. This becomes especially apparent when data distributions are skewed, as is often the case in most realistic scenarios, where estimations can be off by orders of magnitude. Such errors can then lead to large discrepancies between estimated and actual costs and adversely affect automated index selection tools [4]. The use of arbitrary units for query optimizer cost estimates also makes it difficult, if not impossible, to directly compare these estimates with actual observed costs. There is also no direct way to obtain an index creation cost estimate using the what-if interface in some DBMS[1]. Moreover, **invoking what-if calls tends to be very slow** in practice and there can be significant variability in its running-time depending on complexities of the hypothetical query and configuration.

External-cost-models have the potential to mitigate much of these problems associated with the query optimizers internal cost-model. This is because they have the freedom to directly utilize information gained from actual performance observations and adapt accordingly, rather than having to rely on a fixed set of heuristics or assumptions which lead to the same mistakes being made over and over again. Additionally, an external cost-model also offers freedom in terms of flexibility in design choices. In particular, the model can

[1] e.g. Microsoft SQL Server does not provide the functionality for estimating hypothetical index creation cost

be specifically designed to operate within the narrow context of index selection and, therefore, can afford to be **lightweight** and heavily optimized for that task.

3.2 An External Cost-Model

Recent advances in machine learning (ML), particularly deep-learning, have spurred a broad movement known as *ML for Systems* [42] which encompasses many different areas of computer systems. In the context of DBMS, this line of research attempts to enhance or replace different components of database systems with machine learning models [9], such as index structures [19], physical operators [20] and even query optimizers [46]. Traditionally, the design of core components of a DBMS has relied heavily on expert domain knowledge and carefully hand-crafted heuristics. The one-size-fits-all nature of these heuristics often leads to poor performance, e.g. heuristics employed in traditional cost-based query optimizers are based on assumptions which are often violated. At a high level, ML models are capable of directly leveraging hidden/underlying patterns in the data and system, potentially learning to perform tasks better than the best available hand-crafted heuristics and adapting to the particular situation. This strongly motivates their use in database systems.

Not surprisingly, there have been increasing efforts into replacing traditional heuristic-based internal cost-models with ML based external cost-models, targeted at improving query optimizer performance. These efforts have mainly focused on developing *end-to-end* ML models which map an encoded query plan-tree along with an appropriately encoded index configuration into a scalar query execution cost [17, 39, 45].

End-to-end models for predicting query execution cost pose several challenges. The complexity in encoding the input query plan and configuration often necessitates the use of models with more complex architectures and larger number of parameters. Such models tend to be data inefficient[2] making offline training unavoidable. Training entirely online may require a long time before

[2]data inefficiency means that a model requires large amounts of training data to converge meaningfully

the model has converged sufficiently and is therefore impractical[3]. Bootstrapping such models by initially training them offline followed by a further online refinement stage (e.g. [45]) may help to mitigate some of these problems, however models with high complexity may still be too slow in practice. Thus, end-to-end ML-based external cost-models are unwieldy/unsuitable for use in the online index selection task.

3.2.1 A Lightweight Model with Access Path Prediction

Due to the challenges faced by complex end-to-end ML models, a different approach is needed for developing an external cost-model which is **lightweight** and specifically targeted towards the index selection task. To motivate our approach, we first make the following important observation. In large disk-based database systems serving typical workloads, the query processing cost tends to be dominated by disk I/O. This is because data for each relation is stored on disk, usually contained in a large heap file and across a primary index and one or more secondary indexes. And reading from or writing to these sources far exceeds the cost of any CPU operations performed during query processing. Given this fact, we can conclude that the most costly physical operators in a query execution plan will be the *access path* operators[4], as illustrated by the example in Figure 3.1. Leveraging this observation, we can design a lightweight external cost-model which focuses solely on predicting the most likely access paths that will be chosen by the query optimizer for a given query, while safely ignoring the effects of non-access operators.

When generating an execution plan, the query optimizer tries to minimize the total disk I/O cost by selecting the cheapest access path operator(s) for each relation involved in the query result[5]. Usually, a query will contain some number of *filter predicates* under the WHERE clause. The *selectivity* of a predicate, on a certain attribute of a relation, is defined as the fraction of tuples in the relation for which the attribute value satisfies that predicate. If an index is available whose *leading column* intersects with that attribute, then given a low enough selectivity, the query optimizer will favor an index

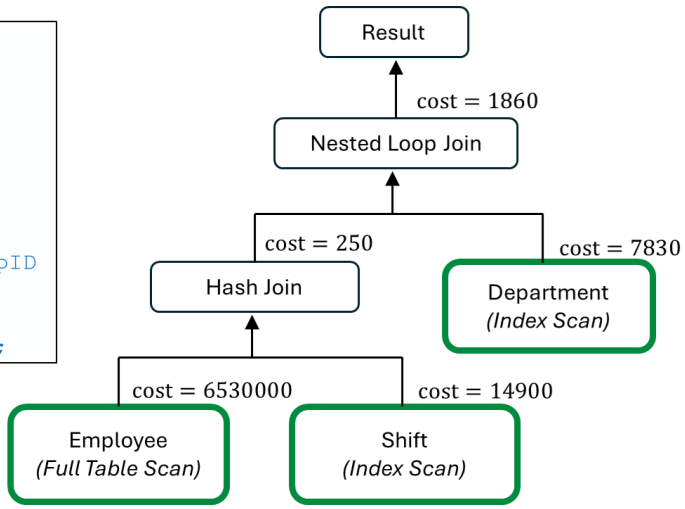
[3]There is high variability in actual query processing times. The difference in execution times between a fast and slow query can often be many orders of magnitude.

[4]access path just means the method used for retrieving data for a relation/table, either through a full table scan or some form of index scan(s) if indexes are available.

[5]In some cases, multiple indexes may be utilized to access data from a relation.

```
SELECT
   e.EmployeeID,
   e.EmployeeName
FROM
   Employee e
JOIN
   Shift s ON e.EmpID = s.EmpID
JOIN
   Department d ON e.DepID = d.DepID
WHERE
   s.ShiftStartTime = '08:30'
   AND d.DepartmentName = 'Sales';
```

(a)



(b)

Figure 3.1: (a) SQL query for retrieving the ID and names of all employees from the Sales department whose shift starts at 8:30 am. (b) Execution plan tree with access paths highlighted in green. Observe that these access-path operators have significantly larger costs.

scan over a full table scan because the relevant tuples can be identified and retrieved using the index with fewer disk reads. Conversely, a full table scan will be preferred for high selectivity as the additional overhead of an index scan would provide no advantage over a full table scan. Thus the choice between a full table scan and an index scan depends critically on selectivities of query predicates and some effort is required for their estimation.

Before going into the details of selectivity estimation, we present our approach for a **simple external cost-model** which is based on the ideas that we've just discussed. At a high level, our algorithm attempts to predict the access paths to be selected by the query optimizer by directly mimicking it's selection approach. Given a hypothetical query and configuration, the algorithm considers the relations from which data needs to be retrieved, then for each relation enumerates all possible access paths (including full table scan and possible index scans based on available indexes in the hypothetical configuration). The disk I/O cost is then estimated for each of those candidate access paths and the cheapest access path is identified and selected. The estimated query execution cost is then defined as the sum over the disk I/O costs of the cheapest access path for each relation. The cost due to all other non-access

operators in the execution plan is assumed to be negligible.

A more formal description of our approach is provided in Algorithm 3. Note that the overhead is mainly from the estimation of disk I/O costs for candidate access paths. We measure the disk I/O cost in terms of the **number of pages** which need to be read from disk. For a full table scan, this quantity is readily available from the database catalog, or can be computed using a simple empirical formula (based on size of each tuple in the relation and the number of tuples per disk page). However, estimating the I/O cost for an index scan is more challenging and involves first estimating the selectivity of the associated filter predicates, then using that selectivity to determine the number of disk pages containing the filtered data. In the next few sections, we will discuss different strategies that we have utilized for selectivity estimation.

Algorithm 3: Simple Cost-Model with Access Path Prediction

Input: hypothetical query q, hypothetical configuration S
Output: execution cost C_{exe}, predicted access paths P

```
1  Function simpleCost(q, S):
2      Initialization: C_exe ← 0, P ← dict();
3      tables ← list of relations which need to be accessed;
4      for each table in tables do
5          paths ← all possible access paths for table given S;
6          cost ← dict();
7          for each p in paths do
8              cost[p] ← estimated disk IO cost of p;
9          C_exe ← C_exe + min_{p∈paths} cost[p];
10         P[table] ← arg min_{p∈paths} cost[p];
11     return C_exe, P;
```

3.2.2 Heuristic-Based Selectivity Estimation

Estimating the disk I/O cost of index scans requires knowledge of the associated predicate selectivity. We start by adopting a **heuristic-based** selectivity estimation approach, commonly used by query optimizers across most modern commercial and open-source database systems, which we will describe by way of an example. Figure 3.2 shows a table called *Employee* and a SQL query which retrieves all tuples satisfying equality predicates on the *City* and

Occupation attributes. The true selectivity for this *multi-attribute predicate* is given by the following joint probability:

$$\text{selectivity} = P(\text{City} = x, \text{Occupation} = y) \tag{3.1}$$

We will now invoke the first heuristic which is to assume that attributes of a relation are **statistically independent**. Applying this heuristic to our example results in the joint probability decomposing into a product of marginal probabilities: $P(\text{City} = x, \text{Occupation} = y) \approx P(\text{City} = x)P(\text{Occupation} = y)$. In order to proceed with this calculation, we require the single-attribute probability distributions. A similar method of calculation also applies for other types of predicates, e.g. *range* predicates.

```
SELECT
    *
FROM
    Employee e
WHERE
    e.City = 'x'
    AND e.Occupation = 'y';
```

Employee Table

ID	Name	Age	City	Occupation
1	Alice	30	Gotham	Data Analyst
2	Bob	22	Metropolis	Grave Digger
3	Charlie	51	Gotham	Product Manager
...	...	...	...	...

Figure 3.2: A SQL query for retrieving data from the *Employees* table

Most modern database systems maintain a collection of tables known as the *system catalog*, in which metadata is stored for every table, index and view. This metadata typically contains information about the data statistics, e.g. the estimated min, max and number of distinct values of each attribute. Additionally, the system catalog may also contain *equiwidth* or *equidepth* **histograms** which approximate each single-attribute distribution[6]. If such histograms are available in the system catalog, then we use them as the single-attribute distributions in our selectivity estimation. If a histogram is unavailable, we instead fall back on a second heuristic which is to assume that the data distribution is **uniform** such that each distinct value of the attribute occurs across the same number of tuples.

[6]Most DBMS prefer the use of equidepth histograms because bins with more frequent values have smaller range leading to better precision compared to an equiwidth histogram with the same number of bins.

The accuracy of the selectivity estimates depends critically on the quality of the system catalog statistics and the validity of our heuristics regarding attribute value independence and uniformity of distributions. It is quite common for a relation to contain attributes which are highly correlated (e.g. [22]), thus the attribute value independence assumption is often violated. Likewise, assuming uniform distributions for attributes is also unrealistic as data distributions tend to be skewed. Moreover, histograms are also prone to inaccuracy for skewed data and catalog statistics have to be updated manually by the user which means the information can often be outdated. In the next section, we will explore some approaches for remedying these problems to obtain more accurate selectivity estimates.

Given that our heuristic-based selectivity estimation approach borrows ideas from the database system's query optimizer, it is not surprising to find that our simple external cost-model with access path prediction is **comparable in accuracy** to the what-if interface. However, our model is also very **lightweight** and therefore **significantly faster**.

3.3 Towards Better Selectivity Estimation via ML

Selectivity estimation is a core aspect of query optimization in database systems. As mentioned in the previous section, assumptions such as attribute value independence and uniform data distribution are often violated and can therefore lead to significant errors in heuristic-based selectivity estimation. Development of techniques which directly alleviate these problems has been an active area of research.

Early approaches focused on incorporating adaptive multi-dimensional histograms into the catalog statistics, e.g. [6, 38]. Other approaches model multi-attribute joint distributions using different density estimation techniques such as statistical moments [14], kernel density estimation [15], cosine series expansion [44] and probabilistic relational models [12]. Selectivity estimates are subsequently derived from these distribution models. The commonality between all these different approaches is that they employ techniques for modeling joint data distributions, which eliminates the need for both attribute independence and uniform distribution assumptions. More recent approaches have started utilizing deep neural networks and tree-based ensemble models, trained using

supervised machine learning (ML), to directly map predicates to selectivity, e.g. [10, 16]. These ML based techniques can also be interpreted as indirectly learning the underlying joint data distributions.

Inspired by some of these aforementioned research efforts, we have made some preliminary investigations into developing our own approach for obtaining improved selectivity estimates in an efficient manner. We have identified two different simple and lightweight techniques which may be promising. Due to the limited scope of this research project, we are unable to carry out a full detailed investigation of these approaches or incorporate them into our experimental benchmarking tests. Nonetheless, we briefly mention some of our ideas below and leave a more detailed investigation for future efforts.

3.3.1 Selectivity Prediction using Online Linear Regression

The first technique that we propose involves selectivity prediction via supervised machine learning. Our goal is to design a learned prediction model which not only has good accuracy but is also sufficiently lightweight. To that end, we believe an **online linear regression** model [32] with *L2-regularization* is suitable as it provides a good balance between model complexity and computational overhead and can also be trained incrementally.

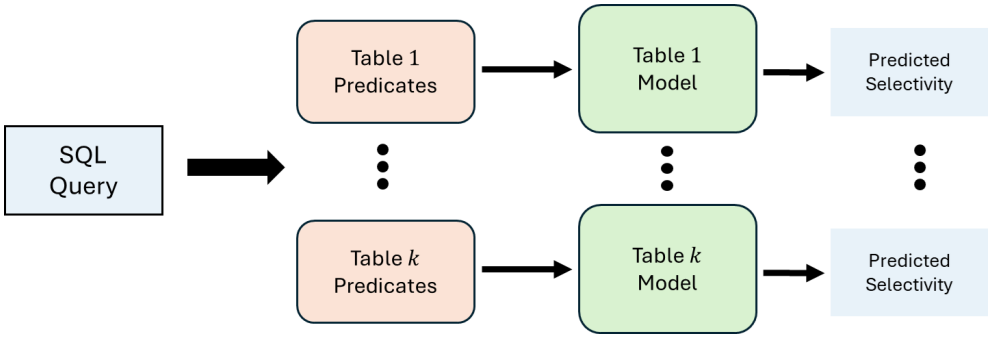


Figure 3.3: Schematic of a selectivity prediction model.

In order to support prediction over multi-attribute predicates, we would need to train a separate model on each database relation/table. Given a SQL query, the first step would be to extract all filter predicates. This is followed by an aggregation of the predicates based on their associated table. Then aggregated predicates for each table can be jointly encoded and fed into the model for the corresponding table to yield a scalar selectivity value as the output.

This general scheme is described in Figure 3.3. A simple encoding format could be used for the filter predicate values, particularly if the values are numeric. Training the model could be carried out in two-separate stages, with initial bootstrapping from predictions made by a heuristic-based selectivity estimator followed by further online training for model refinement. Of course, the training data would consist of pairs of predicate and observed selectivities obtained from actual query execution results[7].

3.3.2 Selectivity Prediction using Learned CDFs

The second technique that we propose follows a similar vein as the different density estimation techniques from the research literature which we mentioned previously. This idea involves directly learning isotonic regression models [13] for the *cumulative distribution function* (CDF) of table attribute values. Selectivity predictions for range (and also equality) filter predicates can then be derived from such a CDF model.

Isotonic regression is a non-parametric model and light-weight algorithms are available for solving this problem. More importantly, this model also has very high sample efficiency and therefore requires relatively few data points to obtain a good fit, making it suitable for online training. Multidimensional isotonic regression can also be used to learn joint CDFs between multiple attributes allowing the model to predict joint-attribute selectivities.

[7] e.g. most DBMS provide the option of viewing the exact cardinalities of each physical operator used in executing the query plan, including access path operators, from which the selectivities can be derived.

Chapter 4

Experimental Evaluation

In this chapter, we address the second goal of our research project by presenting an empirical evaluation of the extended WFIT algorithm, equipped with our simple external cost-model with heuristic-based selectivity estimation. We benchmark the performance of the extended WFIT against not only the vanilla WFIT, which uses the database system's internal cost-model, but also another competitor online index selection technique which is currently considered to be the state-of-the-art.

4.1 Experimental Setup

4.1.1 A State-Of-The-Art Competitor

As mentioned in Section 1.2, the online index selection algorithm of [26, 28], based on Multi-Armed Bandits (MAB), offers a robust theoretical performance guarantee in the form of a sub-linear regret bound. Moreover, this algorithm has been demonstrated to achieve superior performance, both against a state-of-the-art offline index selection tool and other recently developed RL-based algorithms, over a broad range of benchmarking tests. We have therefore chosen to adopt and implement this MAB algorithm as the competitor for our extended WFIT.

The MAB algorithm formulates the online index selection problem in terms of a contextual combinatorial multi-armed bandit [30], where each bandit arm represents an index and an online linear/ridge regression model is used to predict the expected reward for each arm. Before executing each new mini-

workload, the algorithm uses the UCB strategy to balance exploration with exploitation along with a sub-modular maximization oracle to select a subset of arms constituting the recommended index configuration to be materialized. The learned prediction model for expected arm rewards is essentially analogous to the external cost-model of our extended WFIT, and therefore avoids the problems associated with a database system's internal cost-model.

4.1.2 Datasets and Workloads

We make use of two publicly available datasets for our experiments: **TPC-H** [40] and **SSB** [24], both of which are widely used and recognized for their relevance in database performance benchmarking.

The TPC-H schema is a benchmark that simulates a real-world decision support system which is used for assessing performance under a variety of complex workloads characterized by both ad-hoc business-oriented queries and updates. The SSB schema is designed to evaluate the performance of database systems in a data warehousing context, where read-only queries are processed and therefore the data remains static. SSB is also a simplified version of TPC-H, focusing on star schema queries which are also common in decision support systems. These datasets test a database system's ability to efficiently handle large volumes of data and process complex queries, making them suitable as a test bed for online index selection algorithms.

We use a scale factor (SF) of 10 so that each database consists of approximately 10GB of base-table data on disk. We also consider two different types of workloads for our experiments, each simulating a different kind of real-world use-case so that different aspects of the online index selection algorithm can be assessed:

- *Static*: In a static workload, a fixed sequence of query templates appear in a periodic manner, representing a situation where interactions with the database follow a repetitive pattern. A common use-case would be a front-end application interface for a banking or accounting system, involving routine tasks such as account balance inquiries and automated generation of reports/statements. In such scenarios, a single static configuration would provide the most benefit and, ideally, an online index selection algorithm's ability to quickly converge to a single good index

configuration would be tested.

- *Dynamic Shifting*: In a dynamic shifting workload, the sequence of query templates observed can abruptly shift at different points in time. This kind of variability simulates a situation where query interests can fluctuate, requiring the database system to quickly adapt to the new conditions. A common use-case would be an e-commerce platform which experiences seasonal trends and offers promotions during peak shopping seasons, causing abrupt changes in workload patterns for inventory/order tracking and customer service inquiries. In this scenario, the online index selection algorithm would be expected to quickly adapt the index configuration in response to the changing patterns.

For each dataset, we have created a set of custom query templates (12 for SSB and 10 for TPC-H), designed specifically for testing the efficacy of index selection algorithms. Details about these templates can be found in Appendix A. For static workloads, instances of the same set of templates appear on every round. For dynamic shifting workloads, we use a simple design such that the workload is a concatenation of two separate static workloads, each constituting a *phase*. On the first phase, we have instances from exactly half of the query templates appearing on every round in the same sequence. At the end of the first phase, the workload *shifts* into the second phase in which only instances from the remaining half of the query templates appear.

We also note that due to the limited scope of this project, we only consider OLAP (online analytical processing) workloads, which contain SQL queries exclusively, i.e. read-only statements, and no updates. Thus the data distributions remain static.

4.1.3 Operating Environment

Our experiments have been conducted on a personal laptop running a **PostgreSQL** DBMS server inside a Linux Ubuntu operating system. Table 4.1 provides some pertinent information about the system hardware.

Our prototype implementations of the extended WFIT and MAB competitor are both written in python. The python programs running the online index selection algorithms interface with PostgreSQL via the *psycopg2* adapter

Hardware Specs	
Processor	Intel i7-13700HX
Physical Cores	16
Frequency Range	2.1 - 5.0 GHz
Physical Memory (RAM)	16 GB
RAM speed	4800 MHz
Disk (SSD)	1 TB

Table 4.1: CPU and memory specifications

(available as a python package), allowing the programs to directly issue SQL statements to the database server. Since PostgreSQL does not possess a built-in what-if interface, we use a third-party what-if tool available via the *HypoPG* extension. We also ensure a "cold cache" prior to each query execution, which is achieved by restarting the PostgreSQL server to clear out the database buffer cache and invoking appropriate Linux system commands to clear out the OS buffer cache.

4.1.4 Performance Metrics

Our main performance metric is the total workload time, defined in Equation 2.2. Given that the objective of an online index selection algorithm is to minimize this quantity, it gives us a clear and direct measure for comparing and ranking the superiority of different algorithms. We also report the total time per round (sum of mini-workload execution and transition costs), which can be used for analyzing convergence properties, i.e. the ability of the algorithm to converge quickly to a good index configuration. Additionally, we report the average time to generate a recommendation per mini-workload. This quantity is useful because it allows us to gauge the overhead of running the online index selection algorithm and analyze the improvement in efficiency due to the external cost-model over the what-if interface.

To measure the improvements in query processing times brought on by the online index selection algorithm, we use a "naked" baseline called **No Index**, for which we process the workload without any indexes present (i.e. an empty configuration). Finally, a memory budget of 20GB is allotted for the index configuration, i.e. the index selection algorithms are allowed to recommend only feasible configurations which satisfy this budget.

4.2 Results and Analysis

4.2.1 Static Workloads

As mentioned previously, our static workload for each of the two datasets consists of instances of the same set of query templates appearing on every round[1]. Our workloads for both SSB and TPC-H datasets contain 20 rounds each[2].

At the beginning of each round, the online algorithm receives the mini-workload for that round and recommends an index configuration which is materialized before the queries are executed. Given that the data distributions remain static, such a repetitive workload pattern primarily tests the index selection algorithm's ability to **converge** towards a good index configuration over time. Once the algorithms has found a good configuration, it is expected to maintain that same configuration for the remainder of the workload.

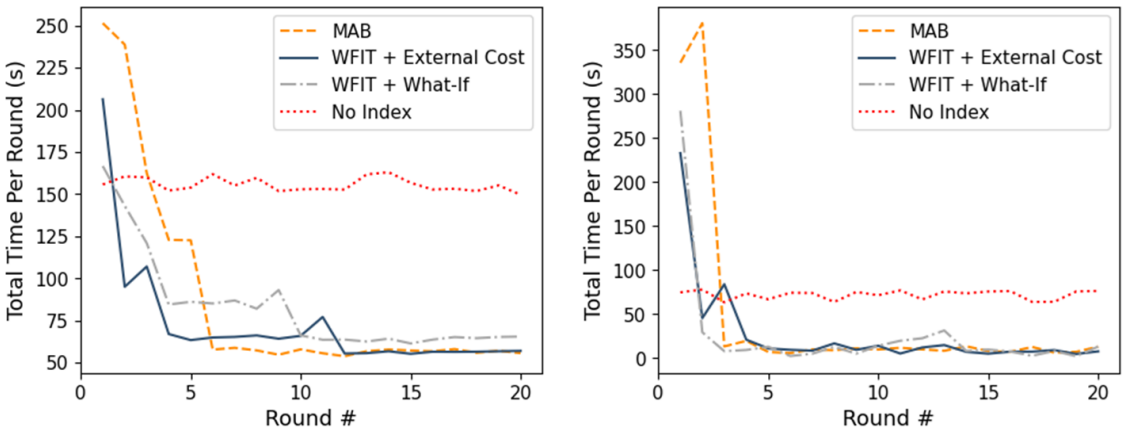


Figure 4.1: Convergence for static workloads: (a) SSB, (b) TPC-H

Figure 4.1 shows convergence plots comparing the total time (i.e. sum of the query execution and index materialization times) per round for the WFIT variants, MAB competitor and No Index baseline. We see that all three index selection algorithms exhibit convergence, outperforming the No Index baseline

[1]Note that instances for the same template across different rounds differ only in the arguments of the query filter predicates.

[2]We deemed 20 rounds to be sufficient since our online index selection algorithms are able to converge within that timescale.

within the first 5 rounds. We note that the extended WFIT with external cost-model is able to converge to a slightly better index configuration than the vanilla WFIT in the case of the SSB workload, as evident from the lower total time per round near the end of the workload. For the TPC-H workload, all three algorithms are able to converge to equally good configurations and at a similar pace.

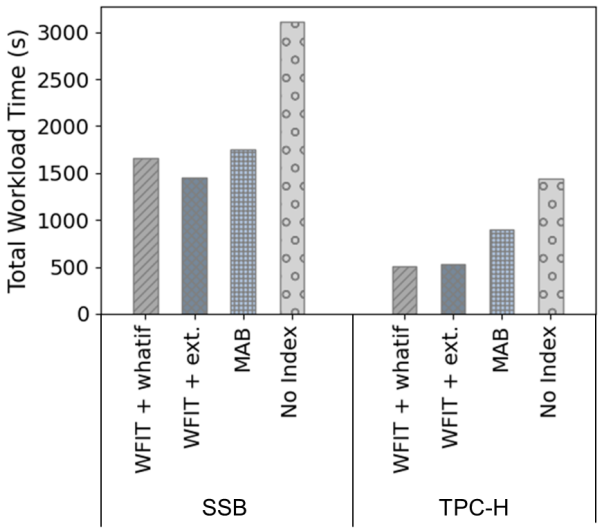


Figure 4.2: Total workload time for static workloads.

The MAB competitor is also able to match the extended WFIT by converging to a very similar configuration, although showing faster convergence on the SSB workload, with MAB converging by round 6 while the extended WFIT materializes it's last index on round 11 to achieve convergence (indicated by a spike on the total time curve in Figure 4.1a). This behavior is not a coincidence and stems from fundamental properties of the work function algorithm, causing WFIT to materializes indexes in a more conservative and incremental manner, growing the index configuration gradually over time[3]. This is in sharp contrast to the way MAB performs exploration to converge to a good configuration. MAB materializes a large number of indexes during the early exploration phase which is why we see larger total costs per round initially. Over time, MAB adds and drops indexes more gradually to converge towards a good configuration, while always maintaining a large configuration throughout this process.

[3]In all our experiments, we initialize both MAB and WFIT with the empty configuration, i.e. no indexes are present at the beginning of the experiment.

The difference in behavior between WFIT and MAB can also be understood from the aspect of how their algorithm design is fundamentally connected to their theoretical performance guarantee. The overly pessimistic nature of the competitive ratio metric manifests in WFIT's reluctance towards committing resources towards exploration of new indexes, leading to more conservative choices during it's decision making for configuration changes. Whereas, the cumulative regret bound of MAB is primarily concerned with improvements in decision making over the long run, and in an average sense. Which is why we see the MAB making more risky and liberal decisions during exploration.

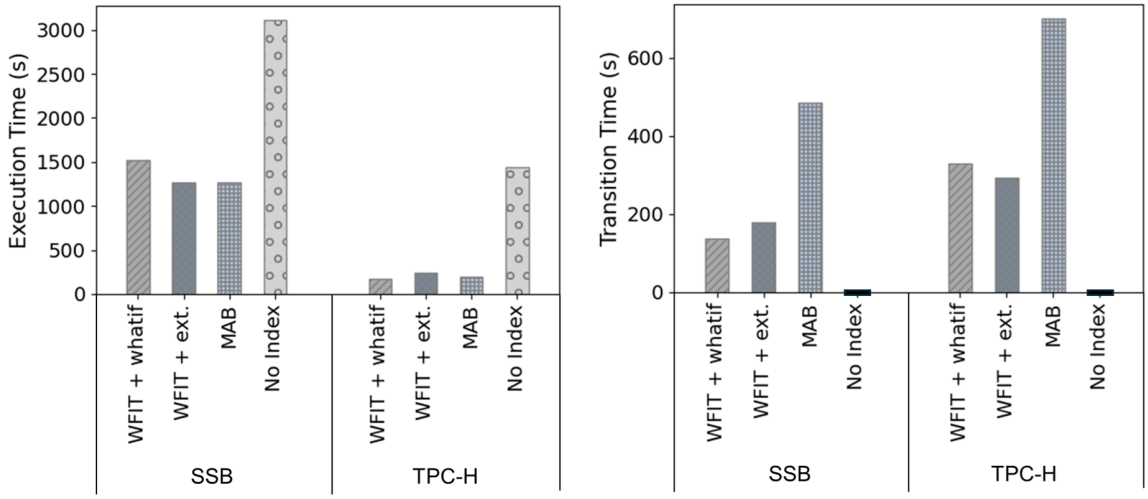


Figure 4.3: Execution and transition time breakdown for static workloads.

Figure 4.2 shows that the total workload times are comparable for the three algorithms, and also substantially better than the No Index baseline. Interestingly, the extended WFIT exhibits superior performance for both SSB and TPC-H workloads, by achieving the lowest total workload time compared to both the vanilla WFIT and MAB competitor. This further lends support to the efficacy of using even a simple external cost-model to replace the reliance on a what-of interface. Also, as noted earlier, WFIT materializes indexes more conservatively, thus it "wastes" less time materializing indexes which are deemed not to be beneficial, whereas the MAB algorithm performs a more liberal exploration of indexes that may not be beneficial, thus incurring very large transition costs, as evident from the breakdown of total workload times shown in Figure 4.3.

In order to highlight the substantial improvement in efficiency of the extended

WFIT over the vanilla WFIT which relies on the what-of interface, we report in Table 4.2, the average time per round taken for each algorithm to perform its computations for generating a recommendation. We see that the extended WFIT is over 500x and 300x faster than the vanilla WFIT, for the SSB and TPC-H workloads respectively, while also generating index recommendations which are at least as good in quality as that of the vanilla WFIT.

Workload	**WFIT + whatif**	**WFIT + ext.**	**MAB**
SSB	78.24	0.15	0.07
TPC-H	65.56	0.21	0.08

Table 4.2: Average Recommendation Time per round (sec) for static workloads.

4.2.2 Dynamic Workloads

Our dynamic workload consists of two phases. Each phase can be thought of as a static workload with it's own unique set of query templates. A shift from one phase to the next brings about a new set of query templates which requires the online index selection algorithm to adapt to the new pattern by appropriately changing the index configuration. Thus, experiments with the dynamic shifting workload primarily test the online algorithms **adaptation ability**. Our workloads for both SSB and TPC-H datasets contain 40 rounds each, with the 21st round marking the shift between phases.

In these set of experiments, we want to focus mainly on what happens near the phase shift event. From the convergence plots in Figure 4.4 showing total time per round, we can see large spikes occuring on the 21st round for all three algorithms, indicating that the algorithms are adapting to the changing workload pattern by materializing new indexes. For both SSB and TPC-H workloads, we note that the extended WFIT is able to very quickly adapt to the workload shift while also achieving the lowest total time per round in the second phase and also the lowest total workload time overall.

For the SSB workload, we observe an odd behavior being exhibited by the MAB competitor. After the phase shift occurs, performance of the MAB algorithm degrades significantly due to an *index oscillation* phenomenon, where

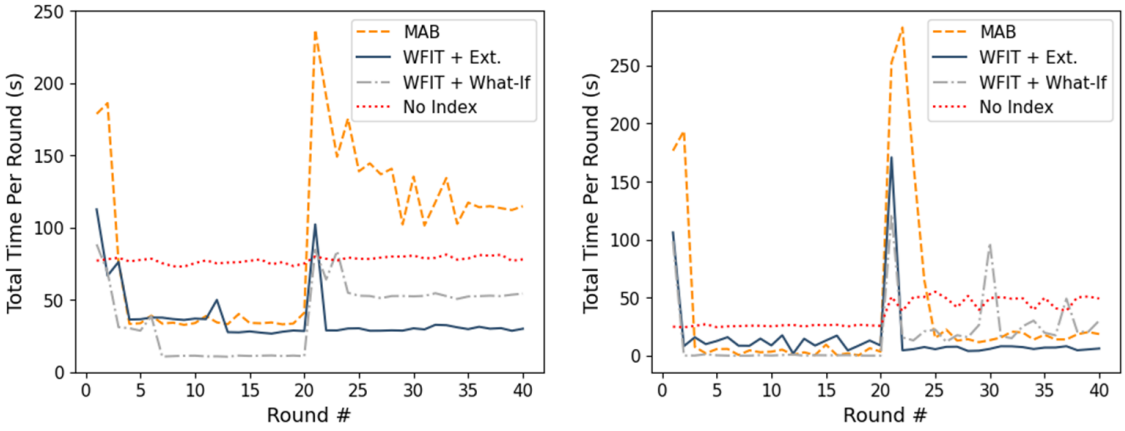


Figure 4.4: Convergence for dynamic workloads: (a) SSB, (b) TPC-H

the algorithm prematurely responds to an abrupt change in the workload, and consequentially gets stuck in a loop of creating and dropping the same bad index repeatedly (see e.g. [5] for more details about this effect). The bad index causes a large cost to be incurred for executing the 3rd query in the mini-workload sequence (substantially larger than the execution cost without that index present) and re-creating this index multiple times incurs a large transition cost. Ultimately, this setback causes the MAB competitor's total workload time to be beaten by the No Index baseline, as highlighted by Figure 4.5.

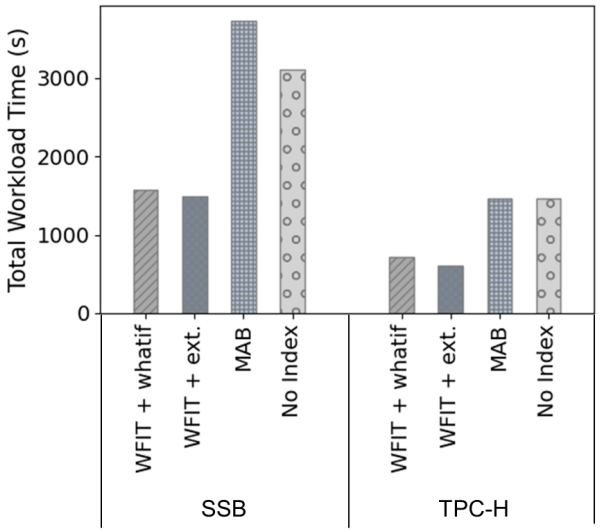


Figure 4.5: Total workload time for dynamic workloads.

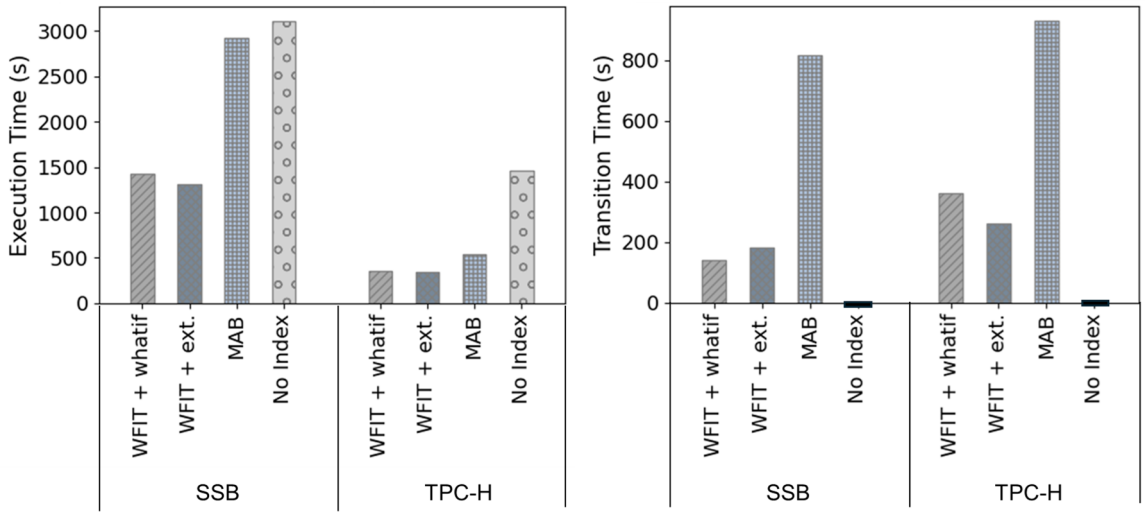


Figure 4.6: Execution and transition time breakdown for dynamic workloads.

Similar to our previous experiments with the static workloads, we see generally much higher total transition cost for the MAB competitor compared to WFIT (see Figure 4.6), highlighting the tendency of MAB to explore a wider range of configurations during the adaptation process. The extended WFIT is also able to achieve lower total execution cost than vanilla WFIT, due to it's slightly better index recommendations thanks to its external cost-model. Once again, average recommendation times per round are also significantly lower for the extended WFIT compared to vanilla WFIT.

Workload	**WFIT + whatif**	**WFIT + ext.**	**MAB**
SSB	84.00	0.14	0.05
TPC-H	30.65	0.07	0.04

Table 4.3: Average Recommendation Time per round (sec) for dynamic workloads.

Chapter 5

Conclusion

Index selection plays a crucial role in the efficient performance of database systems, by directly and profoundly impacting query processing times. Given the need for robust performance guarantees, this research project investigates the use of the work function algorithm (WFA) from the general framework of metrical task systems (MTS), which offers a strong competitive ratio bound.

In particular, we adopt the existing WFIT algorithm from [33] and extend it via incorporating an external cost-model, with the aim of alleviating major weak-points associated with the use of the database system's internal cost-model. Additionally, we also carry out a set of benchmarking experiments to confirm the improvements brought on by our extension of WFIT and also compare its performance against another state-of-the-art (SOTA) online index selection technique [26, 28], based on Multi-Armed Bandits, which offers a different kind of theoretical performance guarantee (bounded cumulative regret).

5.1 Summary of Our Achievements and Lessons Learned

Chapter 3 documents our efforts towards designing the external cost-model. Motivated by the observation that query processing costs for large disk-based databases are often dominated by disk I/O, we reduce the problem of cost prediction to one of **access path prediction**, which avoids much of the intricacies of query processing in general and therefore enables simpler light-weight cost-models to be employed. Because the index selection task is also

intimately linked to data access paths[1], our approach is not only expedient in terms of its simplicity, but is also more natural/suitable for the online index selection task.

To perform access path prediction, we employ a heuristic-based approach similar to that used by a typical DBMS query optimizer, whose overall essence is encapsulated by Algorithm 3. An important component of access path prediction is the problem of selectivity estimation, whereby we estimate the fraction of tuples in any given relation satisfying certain filter predicates. We investigate standard techniques used for selectivity estimation, which include the use of assumptions about the data distributions (such as attribute value independence and uniformity) and database catalog statistics, and highlight some of the perils with these techniques. This leads us into considering alternative machine learning-based techniques for enhancing the accuracy of selectivity estimation. Due to limitations in project scope, we are unable to venture further into this exciting territory, and therefore do not utilize these techniques in our final implementation of the external cost-model.

In Chapter 4, we carry out benchmarking experiments, pitting our extended WFIT against the vanilla WFIT and a state-of-the-art MAB competitor. We use the publicly available and widely used SSB [24] and TPC-H [40] datasets and run experiments over contrived workload patterns (static and dynamic shifting) representing situations that may be typically encountered in real-world use-cases.

Our benchmarking experiments revealed that the extended WFIT, equipped with our simple heuristic-based external cost-model, is able to generate index recommendations which are at least as good as that of the vanilla WFIT, confirming that our access-path driven approach is indeed a valid and powerful idea. Furthermore, we demonstrate that our external cost-model significantly improves the efficiency of WFIT (by up to over 500x). Our experiments also demonstrate that both the vanilla and extended WFIT algorithms hold up well against the current SOTA MAB online index selection algorithm, in some cases outperforming MAB due to WFIT being less susceptible to encountering index oscillations thanks to its conservative and risk-averse decision-making style.

[1] because of the obvious reason that an index is a data access path

However, we must also note that our benchmarking experiments are limited due to certain properties of the datasets we have employed. The data distributions in the SSB and TPC-H datasets are approximately uniform, and attribute values of all relations are only weakly correlated. Consequently, the heuristic-based selectivity estimation approach which we employed is indeed highly accurate (due to the data distributions conforming to our heuristics/assumptions and the database catalog statistics being accurate), which could have contributed towards both the vanilla and extended WFIT being able to strongly compete with the SOTA MAB algorithm. Furthermore, we only used OLAP (i.e. read-only) SQL queries in our workloads, which means that data distributions do not change over time, eliminating the possibility of the database catalog statistics ever becoming outdated.

In a more realistic scenario where data distributions are skewed, attributes are highly correlated and data distributions are evolving over time, we speculate that the heuristic-based selectivity estimation will suffer greatly, leading to a degradation in the quality of WFIT's index recommendations. Whereas the MAB, which relies on an online learning-based cost-model, may be able to adapt more gracefully in those scenarios. Indeed, this scenario motivated us to consider employing online machine learning techniques to enhance the selectivity estimation for our external cost-model.

5.2 Future Directions

As we have already highlighted, a major aspect in which our external cost-model can be improved is via the incorporation of a machine learning-based selectivity estimator. In Section 3.3, we identified and proposed two different possible directions for doing this which may be promising. The first being a light-weight machine learning model (such as online linear/ridge regression) to directly learn to predict selectivities of range or equality predicates. Our second idea is the use of learned models (such as multi-dimensional monotonic regression) for representing the cumulative distribution functions of table attributes. Both of these models are capable of jointly predicting the selectivity over multiple attributes, thus eliminating the need for invoking any heuristics or making incorrect assumptions about the data distributions.

Additionally, we also need to design benchmarking experiments utilizing larger

and more complex datasets with skewed data distributions and strong correlations among values of different table attributes. This kind of dataset would provide an ideal opportunity for testing the ability of learned models to predict join-attribute selectivities and to handle non-uniform data distributions. There is also a need for carrying out experiments using OLTP (online transaction processing) and HTAP (hybrid transactional and analytical processing) workloads, which contain both read-only and update statements, so that data distributions are dynamic. This would provide an opportunity for comparing the learned selectivity models against the use of database catalog statistics which can become easily outdated and therefore highly inaccurate.

In terms of the implementation of the WFIT algorithm, there also remains some opportunities for optimization. Certain sections of the algorithm require the computation of nested loops which are "embarrassingly parallel" in nature and therefore could be significantly accelerated via multi-thread or multi-process parallelization. Also, more carefully designed heuristics/strategies could be employed to identify and prune out non-beneficial candidate indexes to lower the size of the configurations spaces that the work function algorithm instances have to deal with.

Appendix A

Workload SQL Templates

The following 12 query templates were used in constructing the static and dynamic shifting workloads for the SSB dataset. Note that the curly braces represent placeholders for filter predicate arguments which are filled with randomly selected values whenever a template is instantiated.

```
1
2  -- Template 1
3  SELECT lo_discount
4  FROM lineorder, dwdate
5  WHERE lo_orderdate = d_datekey
6  AND d_year = {year}
7  AND lo_discount BETWEEN {discount_low} AND {discount_high}
8  AND lo_quantity BETWEEN {quantity_low} AND {quantity_high};
9
10
11 -- Template 2
12 SELECT lo_discount
13 FROM lineorder, dwdate
14 WHERE lo_orderdate = d_datekey
15 AND d_weeknuminyear = {weeknuminyear}
16 AND d_year = {year}
17 AND lo_discount BETWEEN {discount_low} AND {discount_high}
18 AND lo_quantity BETWEEN {quantity_low} AND {quantity_high};
19
20 -- Template 3
21 SELECT lo_revenue, d_year, p_brand
22 FROM lineorder, dwdate, part, supplier
23 WHERE lo_orderdate = d_datekey
24 AND lo_partkey = p_partkey
25 AND lo_suppkey = s_suppkey
26 AND p_category = '{category}'
27 AND s_region = '{sregion}';
28
29 -- Template 4
```

```
SELECT lo_revenue, d_year, p_brand
FROM lineorder, dwdate, part, supplier
WHERE lo_orderdate = d_datekey
AND lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND p_brand = '{brand1}'
AND s_region = '{sregion}';

-- Template 5
SELECT c_nation, s_nation, d_year, lo_revenue
FROM customer, lineorder, supplier, dwdate
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdate = d_datekey
AND c_region = '{region}'
AND s_region = '{region}'
AND d_year >= {year_low} AND d_year <= {year_high};

-- Template 6
SELECT c_city, s_city, d_year, SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, dwdate
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdate = d_datekey
AND c_nation = '{region}'
AND s_nation = '{region}'
AND d_year >= {year_low} AND d_year <= {year_high}
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC;

-- Template 7
SELECT c_city, s_city, d_year, SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, dwdate
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdate = d_datekey
AND (c_city = '{city_1}' OR c_city = '{city_2}')
AND (s_city = '{city_1}' OR s_city = '{city_2}')
AND d_year >= {year_low} AND d_year <= {year_high}
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC;


-- Template 8
SELECT d_year, s_city, p_brand, lo_revenue
FROM dwdate, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_partkey = p_partkey
AND lo_orderdate = d_datekey
AND c_region = '{region}'
AND s_nation = '{nation}'
AND (d_year = {year_1} OR d_year = {year_2})
AND p_category = '{category}';
```

```
84
85 -- Template 9
86 SELECT lo_linenumber, lo_quantity
87 FROM lineorder
88 WHERE lo_linenumber >= {linenumber_low} AND lo_linenumber <= {linenumber_high
       }
89 AND lo_quantity = {quantity};
90
91 -- Template 10
92 SELECT lo_quantity, lo_orderdate
93 FROM lineorder
94 WHERE lo_quantity = {quantity};
95
96 -- Template 11
97 SELECT lo_linenumber, lo_extendedprice
98 FROM lineorder
99 WHERE lo_extendedprice = {extendedprice};
100
101 -- Template 12
102 SELECT lo_commitdate
103 FROM lineorder
104 WHERE lo_commitdate BETWEEN DATE '{start_date}' AND DATE '{end_date}';
```

Listing A.1: SSB Query Templates

The following 10 query templates were used in constructing the static and dynamic shifting workkoads for the TPC-H dataset.

```
1
2 -- Template 1
3 SELECT
4     c.c_custkey
5 FROM
6     customer c
7 JOIN
8     lineitem l ON c.c_custkey = l.l_orderkey
9 JOIN
10    nation n ON c.c_nationkey = n.n_nationkey
11 JOIN
12    region r ON n.n_regionkey = r.r_regionkey
13 WHERE
14    r.r_name = '{region}'
15    AND c.c_mktsegment = '{mktsegment}'
16    AND l.l_shipdate BETWEEN DATE '{start_date}' AND DATE '{end_date}';
17
18 -- Template 2
19 SELECT
20    s.s_name,
21    p.p_type
22 FROM
23    supplier s
```

```
JOIN
    partsupp ps ON s.s_suppkey = ps.ps_suppkey
JOIN
    part p ON ps.ps_partkey = p.p_partkey
WHERE
    s.s_acctbal > {acctbal_threshold}
    AND p.p_size < {max_size}
    AND p.p_type = '{type}'
    AND ps.ps_availqty > {availqty_threshold}
    AND s_name = '{supplier_name}';

-- Template 3
SELECT
    c.c_custkey,
    c.c_name,
    o.o_orderdate,
    o.o_totalprice
FROM
    customer c
JOIN
    orders o ON c.c_custkey = o.o_custkey
WHERE
    o.o_totalprice BETWEEN {min_price} AND {max_price}
    AND o.o_orderdate BETWEEN DATE '{start_date}' AND DATE '{end_date}'
    AND c.c_name = '{customer_name}'
    AND c.c_nationkey = '{nation_key}';

-- Template 4
SELECT
    l.l_shipdate,
    l.l_extendedprice,
    l.l_discount,
    l.l_quantity
FROM
    lineitem l
WHERE
    l.l_shipdate BETWEEN DATE '{start_shipdate}' AND DATE '{end_shipdate}'
    AND l.l_extendedprice > {price_threshold}
    AND l.l_discount < {discount_threshold}
ORDER BY
    l.l_shipdate;

-- Template 5
SELECT
    o_orderpriority
FROM
    orders
WHERE
    o_totalprice BETWEEN {min_price} AND {max_price}
    AND o_orderstatus = '{status}'
    AND o_orderdate BETWEEN DATE '{start_date}' AND DATE '{end_date}'
    AND o_orderpriority = '{order_priority}'
GROUP BY
    o_orderpriority;
```

```
78
79 -- Template 6
80 SELECT
81     p_partkey,
82     p_name,
83     l_shipmode
84 FROM
85     part
86 JOIN
87     lineitem ON p_partkey = l_partkey
88 WHERE
89     p_retailprice BETWEEN {min_price} AND {max_price}
90     AND p_brand = '{brand}'
91     AND l_shipdate BETWEEN DATE '{start_date}' AND DATE '{end_date}';
92
93 -- Template 7
94 SELECT
95     l.l_orderkey,
96     o.o_orderdate,
97     o.o_shippriority
98 FROM
99     lineitem l
100 JOIN
101     orders o ON l.l_orderkey = o.o_orderkey
102 WHERE
103     o.o_orderdate BETWEEN DATE '{start_date}' AND DATE '{end_date}'
104     AND l.l_discount BETWEEN {min_discount} AND {max_discount}
105     AND l.l_quantity BETWEEN {min_quantity} AND {max_quantity}
106 ORDER BY
107     o.o_orderdate;
108
109 -- Template 8
110 SELECT
111     l.l_extendedprice,
112     s.s_suppkey,
113     s.s_name
114 FROM
115     lineitem l
116 JOIN
117     supplier s ON l.l_suppkey = s.s_suppkey
118 WHERE
119     l.l_shipdate BETWEEN  DATE '{start_shipdate}' AND DATE '{end_shipdate}'
120     AND s.s_name = '{supplier_name}'
121 ORDER BY
122     l.l_shipdate;
123
124 -- Template 9
125 SELECT
126     s.s_name,
127     ps.ps_availqty,
128     o.o_orderdate
129 FROM
130     supplier s
131 JOIN
```

```
    partsupp ps ON s.s_suppkey = ps.ps_suppkey
JOIN
    lineitem l ON ps.ps_partkey = l.l_partkey AND ps.ps_suppkey = l.l_suppkey
JOIN
    orders o ON l.l_orderkey = o.o_orderkey
WHERE
    o.o_orderdate BETWEEN DATE '{start_date}' AND DATE '{end_date}'
    AND s.s_name = '{supplier_name}'
ORDER BY
    o.o_orderdate;

-- Template 10
SELECT
    ps.ps_partkey,
    ps.ps_suppkey,
    ps.ps_availqty,
    ps.ps_supplycost
FROM
    partsupp ps
WHERE
    ps.ps_availqty BETWEEN {min_quantity} AND {max_quantity}
    AND ps.ps_supplycost BETWEEN {min_cost} AND {max_cost}
ORDER BY
    ps.ps_supplycost;
```

Listing A.2: TPC-H Query Templates

Bibliography

[1] Sanjay Agrawal et al. “Database Tuning Advisor for Microsoft SQL Server 2005”. In: *VLDB*. Very Large Data Bases Endowment Inc., 2004.

[2] Debabrota Basu et al. “Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning”. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXVIII - Volume 9940*. Berlin, Heidelberg: Springer-Verlag, 2016, 96–132.

[3] Allan Borodin, Nathan Linial, and Michael E. Saks. “An optimal on-line algorithm for metrical task system”. In: *J. ACM* 39.4 (1992), 745–763.

[4] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. “Automated physical designers: what you see is (not) what you get”. In: *Proceedings of the Fifth International Workshop on Testing Database Systems*. DBTest ’12. Scottsdale, Arizona: Association for Computing Machinery, 2012.

[5] Nicolas Bruno and Surajit Chaudhuri. “An Online Approach to Physical Design Tuning”. In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pp. 826–835.

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. “STHoles: a multidimensional workload-aware histogram”. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’01. Santa Barbara, California, USA: Association for Computing Machinery, 2001, 211–222.

[7] Surajit Chaudhuri and Vivek Narasayya. “AutoAdmin “what-if” index analysis utility”. In: *SIGMOD Rec.* 27.2 (1998), 367–378.

[8] Bailu Ding et al. “AI Meets AI: Leveraging Query Executions to Improve Index Recommendations”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1241–1258.

[9] Jialin Ding et al. “SageDB: An Instance-Optimized Data Analytics System”. In: *Proc. VLDB Endow.* 15.13 (2022), 4062–4078.

[10] Anshuman Dutt et al. “Selectivity estimation for range predicates using lightweight models”. In: *Proc. VLDB Endow.* 12.9 (May 2019), 1044–1057.

[11] Jianling Gao et al. “Automatic index selection with learned cost estimator”. In: *Information Sciences* 612 (2022), pp. 706–723.

[12] Lise Getoor, Benjamin Taskar, and Daphne Koller. “Selectivity estimation using probabilistic models”. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’01. Santa Barbara, California, USA: Association for Computing Machinery, 2001, 461–472.

[13] Charles J. Geyer. *Isotonic Regression*. https://www.stat.umn.edu/geyer/8054/notes/isotonic.pdf. [Lecture Notes for Stat 8054 at UMN]. 2023.

[14] Goetz Graefe. *Selectivity estimation using moments and density functions*. Tech. rep. Oregon Graduate Center, 1987.

[15] Dimitrios Gunopulos et al. “Selectivity estimators for multidimensional range queries over real attributes”. en. In: *The VLDB Journal* 14.2 (2005), pp. 137–154.

[16] Shohedul Hasan et al. “Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, 1035–1050.

[17] Benjamin Hilprecht and Carsten Binnig. “Zero-shot cost models for out-of-the-box learned cost prediction”. In: *Proc. VLDB Endow.* 15.11 (2022), 2361–2374.

[18] Ivo Jimenez et al. “Benchmarking Online Index-Tuning Algorithms”. In: *IEEE Data Eng. Bull.* 34.4 (2011), pp. 28–35.

[19] Tim Kraska et al. “The Case for Learned Index Structures”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: Association for Computing Machinery, 2018, 489–504.

[20] Ani Kristo et al. “The Case for a Learned Sorting Algorithm”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Man-*

agement of Data. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 1001–1016.

[21] Viktor Leis et al. "How good are query optimizers, really?" In: *Proc. VLDB Endow.* 9.3 (2015), 204–215.

[22] Guy Lohman. "Is Query Optimization a "Solved" Problem?" In: *ACM SIGMOD Blog* (2014). [https://wp.sigmod.org/?p=1075].

[23] Tanu Malik et al. "Adaptive Physical Design for Curated Archives". In: *Scientific and Statistical Database Management*. Ed. by Marianne Winslett. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 148–166.

[24] Pat O'Neil, Betty O'Neil, and Xuedong Chen. "Star Schema Benchmark". Available at: https://www.cs.umb.edu/~poneil/StarSchemaB.PDF. 2024.

[25] M. O'Searcoid. *Metric Spaces*. Springer Undergraduate Mathematics Series. Springer London, 2006.

[26] R. Malinga Perera et al. "DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 600–611.

[27] R. Malinga Perera et al. "HMAB: self-driving hierarchy of bandits for integrated physical database design tuning". In: *Proc. VLDB Endow.* 16.2 (2022), 216–229.

[28] R. Malinga Perera et al. "No DBA? No Regret! Multi-Armed Bandits for Index Tuning of Analytical and HTAP Workloads With Provable Guarantees". In: *IEEE Transactions on Knowledge amp; Data Engineering* 35.12 (2023), pp. 12855–12872.

[29] Gregory Piatetsky-Shapiro. "The optimal selection of secondary indices is NP-complete". In: *SIGMOD Rec.* 13.2 (Jan. 1983), 72–75.

[30] Lijing Qin, Shouyuan Chen, and Xiaoyan Zhu. "Contextual Combinatorial Bandit and its Application on Diversified Online Recommendation". In: *Proceedings of the 2014 SIAM International Conference on Data Mining*. SIAM, 2014, pp. 461–469.

[31] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. "QUIET: continuous query-driven index tuning". In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, 1129–1132.

[32] Rob Schapire and Maryam Bahrani. *Online Linear Regression*. https://www.cs.princeton.edu/courses/archive/spring18/cos511/scribe_notes/0411.pdf. [Lecture Notes for COS 511: Theoretical Machine Learning]. 2018.

[33] Karl Schnaitter and Neoklis Polyzotis. "Semi-automatic index tuning: keeping DBAs in the loop". In: *Proc. VLDB Endow.* 5.5 (2012), 478–489.

[34] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. "Index interactions in physical design tuning: modeling, analysis, and applications". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), 1234–1245.

[35] Karl Schnaitter et al. "On-Line Index Selection for Shifting Workloads". In: *2007 IEEE 23rd International Conference on Data Engineering Workshop*. 2007, pp. 459–468.

[36] Tarique Siddiqui et al. "DISTILL: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning". In: *Proc. VLDB Endow.* 15.10 (2022), 2019–2031.

[37] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 6th ed. New York: McGraw-Hill, 2010. Chap. 2.

[38] Michael Stillger et al. "LEO - DB2's LEarning Optimizer". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, 19–28.

[39] Ji Sun and Guoliang Li. "An end-to-end learning-based cost estimator". In: *Proc. VLDB Endow.* 13.3 (2019), 307–319.

[40] *TPC-H Benchmark*. [https://www.tpc.org/tpch/].

[41] Adam Wierman. "A tale of two metrics: competitive ratio and regret". In: (2014). [https://rigorandrelevance.wordpress.com/2014/10/13/a-tale-of-two-metrics-competitive-ratio-and-regret/].

[42] *Workshop on ML for Systems at NeurIPS*. [https://mlforsystems.org/].

[43] Wentao Wu et al. "Budget-aware Index Tuning with Reinforcement Learning". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 1528–1541.

[44] Feng Yan et al. "Selectivity estimation of range queries based on data density approximation via cosine series". In: *Data Knowledge Engineer-*

ing 63.3 (2007). 25th International Conference on Conceptual Modeling (ER 2006), pp. 855–878.

[45] Jiani Yang et al. “Rethinking Learned Cost Models: Why Start from Scratch?” In: *Proc. ACM Manag. Data* 1.4 (2023).

[46] Zongheng Yang et al. “Balsa: Learning a Query Optimizer Without Expert Demonstrations”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 931–944.

[47] Tao Yu et al. “Refactoring Index Tuning Process with Benefit Estimation”. In: *Proc. VLDB Endow.* 17.7 (2024), 1528–1541.