Hypothetical Index Benefit Estimation on Column-oriented Databases using Quantiles

Clément Rouvroy
DIENS, ENS Paris
Paris, France
CCDS, Nanyang Technological University
Singapore, Singapore
rouvroy@phare.normalesup.org

Abstract

Columnar databases start to leverage indexes to enhance bottleneck queries efficiency. To decide if an index may be useful or not in row-oriented systems, a commonly used tool is a "What-If" API. It returns an estimation of the benefit of an index without constructing it, saving time and resources at the cost of possible mistakes in the prediction. Moreover, to have a complete auto-tuning database, recent researches leverage Machine Learning to have an automatic database administrator that tunes the database configuration using workload knowledge and "What-If" calls. As column-oriented databases are just starting to leverage indexes, there is yet no such tool specifically designed for these systems. In this work, we propose three contributions: heuristics for segment-aware access paths costs estimations in column-oriented databases, a hypothetical index benefit estimation designed for column-oriented databases (limited to hash indexes), and the use of quantile regression to trade-off precision and risk.

Keywords

Database, Reinforcement Learning, Query optimization

1 Introduction

Index Selection. To compute solutions to a query, Relational Database Management Systems (RDBMS) need to scan data from disk before processing them. To accelerate these scans, one can use alternative access to data with different properties (e.g. a different sorting order, or a different data structure), called secondary indexes. The most studied problem associated is *Index Selection Problem*: given a set of queries and a budget, select the optimal index configuration. This problem is known to be NP-Complete [20], hence it is mainly tackled by heuristics [8] or Machine-Learning based methods [5, 25].

Known methods. Recent progress in index selection involves Machine Learning [5, 13, 19]. Most methods rely on "What-If" API calls [4] (an API that estimates the benefit of an index without losing time constructing it), either to get a reward for their models using hypothetical index estimation [13] or to get query plans for hypothetical configuration [5]. However, What-If are known to be imprecise due to their heuristics and use of biased cardinality estimations, often assuming uniformity of distributions and ignoring data skew [15]. Hence, new methods [25] traded What-If for attention layer [29] to learn how the database reacts to changes, which seems to better capture index interaction [22].

Column-oriented databases. Traditionally, secondary indexes were used in row-oriented [27] databases, as column-oriented [3] databases use different architectures that rely on compression and immutable data structures efficient enough to trade-off indexes for projections [1]. However, recent works on hybrid workload databases [32] demonstrated that column-oriented database with the addition of secondary hash index can be used to make a trade-off between analytical and transactional workload. The first work to add secondary hash index support to column-oriented databases is Singlestore [21]. They use it to balance the worst query pattern for column-oriented system, the point access (a query that returns all columns of a single row).

Objective. Though What-If are less precise than trained Machine Learning model, and can take up to 90% of index selection time [18], they are a commonly used, interpretable, and lightweight tools that is still desirable to have for any RDBMS. To the best of our knowledge, there is yet no column-oriented database that support What-If API calls, hence our first objective is to build an API that estimates the benefit of a hypothetical index for column-oriented secondary hash index. Building this API implies giving heuristics for the cost of column-oriented plan operation in database system, which has not been done before. To test the reliability of our estimator, we used it in different algorithms to select index in Singlestore. In particular, we want to provide a solution that is:

- **Extendable**. As indexes are emerging in column-oriented databases, we want to provide an architecture that can be extended easily to capture new index types or operators without having to train Machine Learning models from zero.
- **Risk-aware**. As estimations are prone to errors, we want to allow the DBA to trade-off opportunity and risk in the benefit estimation.

To achieve these objectives, we propose a **Quantile-based Hypothetical Index for Column-Store** (QHICS). More precisely, QHICS first encodes an index into two vectors:

- A *benefit* vector representing the difference in estimated access path [24] cost in three metrics. We propose the first access path estimation designed for column-oriented databases. In particular, it is aware of segment skipping.
- A *cost* vector representing the cost of building LSM-based index, also in three metrics.

Then, QHICS leverages quantile regression [10] twice to predict creation cost and workload benefit. The use of cost and benefit quantiles allows QHICS to mostly overestimate creation cost and underestimate benefit. This usage of quantiles can be used to trade

off the minimization of prediction error and the number of overestimation of benefits. *To the best of our knowledge, this is the first time a hypothetical index estimation is built for column-oriented database systems, and the first time quantile regression is used for hypothetical index estimation.*

There is yet no method that satisfies all our objectives as there is no hypothetical index estimation built for column-oriented systems. There exists estimations that are agnostic to the architecture, hence the orientation([25]), but they are based on transformers, failing to be interpretable and easily extendable.

To test our model, we have implemented QHICS over SingleStore (former MemSQL)[1] to estimate benefit of secondary hash indexes.

In summary, our main contributions can be summarized as:

- Propose formulas to estimate access paths cost in column-oriented databases.
- Leverage quantile regression to estimate index benefit with a trade-off between estimation precision and number of benefit overestimation
- Implement our model in Python to test its efficiency.

Our implementation (over a subset of SQL) demonstrated that:

- Quantiles allow to use a What-If with default parameters to perform index comparison with satisfying results (81% of ranking score on zero-shot).
- Once enough index has been created, one can automatically tune the cost model for its database, getting much more satisfying results: 92% ranking score, 34% average benefit estimation error.
- Once a lot of data for a specific index has been accumulated, one can fine tune a model for this index, getting almost perfect results: 97% of ranking score, 9% average benefit estimation error.

The rest of this work is organized as follows:

- section 2 introduces the background on column-oriented databases and on index selection, providing key insights about the architecture of these systems and their indexes.
- section 3 introduces the problems of Index Benefit Estimation and Index Selection formally.
- section 4 presents an overview of QHICS, with its main components. For a more in-depth presentation of QHICS, section 5 presents which features we extract from the workload, section 6 presents how we use these features to encode an index into vectors, and section 7 presents how we associate a metric to a vector.
- section 8 presents the index selection algorithms we use and section 9 presents our result.

2 Background and related work

In this section we resume important advances in column-oriented databases and index selection. We also motivate our research problematic.

[1]Open-source on GitHub at https://github.com/CRouvroy/QHICS

2.1 Column-oriented databases strength and weakness

Column-oriented databases [3] are storing data on disk per columns and not per rows, hence:

- the access to all values in one column is a sequential scan (called *column scan*),
- the access to all values of one row needs one sequential scan per column to find the needed row ID in *each column*.

2.1.1 Column-oriented analytical benefit. Columnar scans can accelerate many query processing. fig. 1 contains an example where one read 50% less data on disk in a column-oriented database compared to row-oriented database. Moreover, column-oriented databases can leverage three mechanisms to enhance query efficiency:

- **Per-column compression**. In row-oriented databases, data are compressed per tuples, limiting the possibilities. In columnar, the database engine can use a different encoding for each column [1]. This can dramatically reduce the size of columns to read on disk, at the cost of extra CPU time for decompression. Vertica [12] can divide by almost 15 the size of certain columns.
- **Segment skipping**. Data are stored per columns, but tables are also horizontally split in different regions of configurable sizes (around 1,000,000 tuples), each region is called a *row segment*, and each row segment contains one *columnar segment* per column in the table. Apart from enabling to store the same tables on different disks or clusters, this also enable *segment skipping*. Each columnar segment stores a metadata containing its minimum and maximum value, hence given access to a column with a known range, one can efficiently skip segments by checking if the range intersects the segment's metadata. An example of segment skipping is given in fig. 2. This mechanism can have different names but is implemented in most major column-oriented databases [1, 9, 14, 23, 28].
- **Projections**. To accelerate queries that need a sorted order, column-oriented systems allow maintaining duplicate of some column subsets of a table, maintained in a precise order. This can be seen as an ordered index, but this is a whole physical copy of columns data hence is not as efficient as a structure using references.

2.1.2 Column-oriented immutable architecture and consequences. Most column-oriented systems are using an architecture where segments are *immutable* to store data using an LSM tree [17], this leverage two main properties:

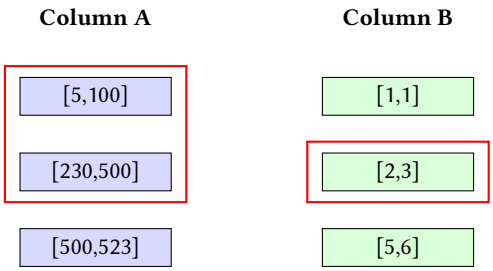
- The background merger of LSM-tree is used to maintain sorted runs that allow meaningful segment skipping [28].
- The first level of an LSM-Tree is stored in memory, and higher levels are on disk. Insertions are firstly performed in memory before being merged into higher levels, hence the first cost paid for update is low [28].

However, they are still less optimal than row-oriented databases for updates, as data newly written maybe merged once per level of the LSM-tree and there is a background process consuming

EmpId	SellId	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM SELLS
```

Figure 1: In row-oriented databases, one need to read every row and project on `Amount`, reading `EmpId` and `SellId` for nothing. In column-oriented database, one can ready only `Amount`, reading a third of what a row-oriented database would have read.




```
SELECT * FROM T WHERE A <= 270 AND B = 2
```

Figure 2: In this query, the database only needs to open the second columnar segment of each column. As the first segment of `A` contains to the same rows as the one of `B`, and as the condition is an `AND`, there can't be any solution in the first columnar segment.

resources to merge levels (inducing an overhead logarithmic in the number of segments).

As a solution, most Hybrid Transactional and Analytical Systems (HTAP) are pairing column-oriented components (to handle OLAP) and row-oriented components (to handle OLTP) to have databases capable of handling both kind of workloads [32].

2.1.3 Column-oriented point access flaws. All optimizations presented explain why column-oriented databases are mainly used in data warehouses, where the workload is analytical (OLAP) [11]. Analytical workloads require many aggregate functions over a subset of columns, hence there are few queries that ask for a precise element. However, when such a *point access* query appears, most column-oriented systems are less efficient than row-oriented systems because once the needed row ID has been computed, the column-oriented database need to search for this value in every column.

2.1.4 Seekable encoding. To mitigate bottleneck queries, SingleStore [21] restricted the encoding techniques to a subset that can are seekable: to access a precise row ID in a column, they can leverage seekable encoding to decompress only a small portion of data around the needed row ID to access its value, removing the need of reading every value until the desired row ID is reached. This optimization is powerful enough for SingleStore to run HTAP workloads without on-disk row-oriented storage.

2.2 Indexing in column-oriented databases

Column-oriented databases may benefit from indexes since the addition of seekable encoding, SingleStore [21] is the first system to leverage secondary hash index for column-oriented databases by allowing the DBA to create secondary Hash Index.

They use an index structure that has two components:

- An *inverted index* per segments, mapping from value to the list of offsets inside the segment corresponding to this value. This can be paired with the seekable encoding to efficiently retrieves row IDs with a precise value inside a column.
- A *global index* which is an LSM-based hash table that maps a hash value to a list of segment ID paired with the offset in the inverted index where offsets inside the segment are stored.

This structure also *support multi-columns index*. When the user wants to create an index over a vector of columns $\vec{C}$, an index of one column is created over each column in $\vec{C}$, and then another global index (but no new inverted index) is created. This global index takes the hash of a vector of values (one value from each column in $\vec{C}$) and returns a list of length $|\vec{C}|$, where each element in the list contains the segment and offset for each indexed column. For every combination of values across the columns in $\vec{C}$, the global index allows retrieval of, for each column, the segment and the offset in the inverted index where the row offsets for those values are stored. This structure optimizes the search for tuples matching a specific combination of values across multiple columns, while avoiding redundant inverted indexes for every possible value combination. As a drawback, the multi-column global index can not be leverage for prefix of $\vec{C}$ compared to some existing index structures.

This architecture allows efficient update. At the level 0 of the global index (in memory) there is one hash table per segment, and a background process merges hash tables together on the run. When a segment gets added it is merged with others in the background, and when a segment is deleted, it is suppressed the next time it is merged.

Recent system [26] proposed to use an ordered LSM-based data structure to replace the hash-based global index. This allows secondary ordered index in column-oriented databases, however it is still in production hence this work limits itself to secondary hash index. Our heuristics for access pathes should still apply for ordered indexes, but one need to think about adding ordering to the encoding.

As modern analytical databases [23] and modern hybrid databases [9, 21] are using column-oriented storage, and as this storage is starting to leverage indexes, we want to work towards automatic column-oriented index tuning. The first step, we think, is the creation of a hypothetical index estimation and a demonstration of its capacities in different algorithms to select indexes.

2.3 Quantile Regression & Linear Programming

For $(n, p) \in \mathbb{N}^2$, let $\mathcal{X} \in \mathbb{R}^{n \times p}$ be a matrix of n inputs and $y \in \mathbb{R}^n$ be the objective vector. The usual regression task in Machine Learning,

called *Linear Regression*, asks to find a vector $\vec{\theta}^* \in \mathbb{R}^m$ such that $\|y - \mathcal{X}\vec{\theta}^*\|_2^2 = \min_{\theta \in \mathbb{R}^m} \|y - \mathcal{X}\vec{\theta}\|_2^2$. It is well-known that one can use $\vec{\theta}^* = \left(\mathcal{X}^T\mathcal{X}\right)^{-1}\mathcal{X}^T y$ as a solution. This should minimize the *average error of the benefit estimation.*

However, the error weight is the same for overestimation and underestimation, though one might prefer to punish more overestimation than underestimation: missing an opportunity is more acceptable than recommending a bad index. Hence, we want to put more weights on overestimation errors than underestimation errors. For this, given a quantile $q \in]0;1[$, we leverage quantile regression [10], which learns to estimate the q-th quantile by minimizing $\sum_{i=1}^{n} \rho_q(y_i - \mathcal{X}_i\vec{\theta})$, where

$$\rho_q(r_i) = \begin{cases} qr_i & \text{if } r_i \geq 0 \\ (q-1)r_i & \text{if } r_i < 0 \end{cases}$$

is a function that give (in absolute value) a weight q to underestimation and $1 - q$ to overestimation.

There exists many methods to solve Quantile Regression problems, however in our model we want our learned coefficients to represent system times, which should be positive. Hence, we need to find the best solution to Quantile Regression under the constraint that learned parameters are positive. For this we leverage Linear Programming.

2.4 Related Work

There is no hypothetical index estimator for column-oriented system yet, however there are decades of study on hypothetical index benefit estimation. In this part we review why different methods do not fully fit our problematic and what key takeaway we can take from them.

2.4.1 What-If. What-If [4] uses database statistics to estimate the benefit of a (hypothetical) index on a given workload. They are mostly based on heuristics and are known to be vulnerable to data skews and wrong cardinality estimations [15]. A common way of building a What-If is to estimate the difference in query plan execution time between a plan with the hypothetical index included in the configuration, and a plan without the index. This implies giving heuristics to estimate access path cost [24]. All works on heuristics for access path selection are designed for traditional database architecture (B+-tree) and not for LSM-based databases. The only work giving an approximation for columnar scans is TiDB [9], however their heuristics do not capture *segment skipping* hence is not sufficient for this work. Moreover, no existing work provides a way to estimate the number of segments needed to open to answer a query, though it would be desirable to estimate the cost of a columnar scan.

2.4.2 Index Benefit Estimation. What-If is an API used to estimate the benefit of index using statistics, but there are other ways to estimate the benefit of an index. Some works leverage Machine Learning methods [30], with different degrees of *transferability*, i.e. up to which points we can transfer the algorithm to estimate index on different architectures. Though Machine Learning methods are showing better results, we want to keep a heuristics-based method to estimate the index benefit as there is a lack of What-If for column-oriented databases and this is the first step to have one. Moreover, Machine Learning methods need to be re-trained when changing architectures, and may lack of interpretability or extendability.

2.4.3 Index Selection using benefit estimation. Reinforcement Learning is used to learn to select indexes on a fixed workload by using What-If estimation as a reward and one-hot encoding of already built index as an input [13]. The goal of this method is to better capture index interaction [22] by using the agent to explore different interaction between indexes. Once the agent is trained, it is used to recommend index and really build the index.

3 Problem

3.1 Definitions

We start by presenting key definitions and notations used through this work. Then, we present our research problem statement.

Analytical Workload. An analytical workload $\mathcal{W} = \{q_1, \ldots q_n\}$ with $n \in \mathbb{N}$ is a set of n queries on one or multiple tables in a relational database, such that no query are creating, deleting, or updating data. A query is a string following the rules of one SQL language.

Secondary index. According to the architecture presented in section 2.2, we define a secondary index as a secondary hash index that uses two components, one inverted index per segments and a global hash table mapping values to a list of segments associated with their offset. A secondary index can be on one or multiple columns, but this work restricts indexed fields to integer and decimal fields.

Index Configuration. An index configuration c_i is a set of secondary indexes.

Index Candidates. Index candidates $C = \{c_1, \ldots, c_k\}$ if a set of k possible index configurations.

Definition 3.1 (Index Benefits). Given an index configuration c_i and a query q_j, the index benefits of c_i on q_j is defined as the *query cost reductions* induced by the materialization of c_i, It is formulated as:

$$cr_{i,j} := \text{Cost}(q_j \mid \emptyset) - \text{Cost}(q_j \mid c_i) \tag{1}$$

where Cost is the execution cost of a query under a given configuration and $\emptyset$ is the configuration with no secondary index.

One may also estimate the benefit of adding a configuration to an existing one, this is:

Definition 3.2 (Index Benefits over a configuration). Given an index configuration c_i, query q_j, and a configuration c, the index benefits of c_i on q_j over c is defined as:

$$cr_{i,j|c} := \text{Cost}(q_j \mid c) - \text{Cost}(q_j \mid c_i \cup c) \tag{2}$$

3.2 Problem statement

The goal of a hypothetical index estimator is to return a value that can be used to know if an index may be created or not. Some What-If returns a value that has no unit and should be used to compare different index possibilities, in this work we decided to return a value that should be as closed as possible from the real benefit in milliseconds. Hence, our goal will be to estimate the reduction in cost between executing the workload with and without a given index in the configuration. Formally, this gives the following problem definition:

Index Benefit Estimation. Given an analytical workload $\mathcal{W}$ and a configuration c, *Index Benefit Estimation* asks to estimate the benefit on $\mathcal{W}$ of a configuration c_i when there is already a configuration c. This is estimating $\sum_{q_j \in \mathcal{W}} cr_{i,j|c}$.

Once we are able to estimate the benefit of an index, we want to take decision based on this estimation. The main decision process we study is the index selection, this is trying to estimate the best indexes to create over some constraints (constructing time, space taken, number of indexes materialized, CPU pressure, ...). This can be formalized as:

Index Selection. Given an analytical workload $\mathcal{W}$, index candidates C, and some budget B, *Index Selection* asks to select the index configurations that maximizes the index benefit while satisfying the budget.

In this work, we present QHICS as an algorithm to tackle Index Benefit Estimation for column-oriented databases, and we use it in different known index selection algorithms to test its efficiency. As this is the first time the problem is studied for column-oriented database system, we have no other tools to compare it to, though we will compare it to schema-agnostic machine learning tools in the future.

4 QHICS overview

The hypothetical index benefit estimator presented in this work, named QHICS, is composed of three components:

- The **Feature extractor** takes a workload and extracts meaningful information. These features will be used to encode vectors. QHICS extracts each scan operations along with its filters, and also captures highly selective joins. It can be extended to capture more informations, but for hash index (which are the only secondary index structure available yet), we have found these features to be sufficient (for reasons detailled in section 5.1).
- The **Index encoder** can encode an index into two vectors, one for the benefit on the workload and one for the creation cost. Each vector is composed of three dimensions, the estimated Disk IO usage, the estimated CPU usage, and the estimated memory usage. We decided to use index encoding because each system has different components speed, hence we want to leverage regression to learn how each metrics impact the total benefit. This can be seen as a way of finding constants representing the time per cpu operation, disk operation, and memory operation.
- The **Cost model** takes an encoded benefit (resp. cost) vector and multiply it by a vector of parameters $\vec{\theta_{benef}}$ (resp. $\vec{\theta_{cost}}$) to get the estimated benefit (resp. cost). This vector is of dimension three and is tuned using positive quantile regressions on a dataset of encoded indices along with their real benefit (resp. creation cost). Quantiles of QHICS can be changed at any moment at the cost of solving a linear programming problem on the training dataset.

One can see how these different components communicate together in fig. 3. For a precise description of each component, section 5 explains why and how we extract the features we have found to be relevant, section 6 explains how we encode an index into two vectors given the analysed workload, and section 7 explains the cost model and how we tune parameters.

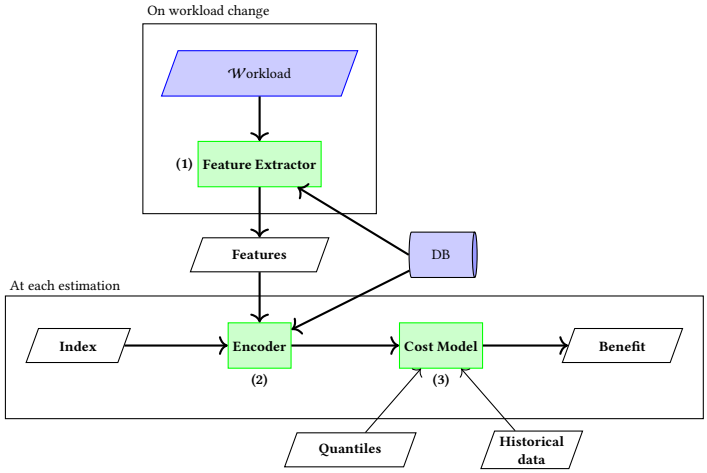


Figure 3: (1) represents the feature extractor, which is called once per workload change. It communicates with the databases to get statistics, schema information and physical plans. (2) represents the encoder, it uses the features extracted before and database statistics to represent the index into a vector of fixed dimension. (3) represents the tuned cost model, it uses two quantiles to have a risk and benefit trade-off, it is tuned using positive quantile regression over historical data. The cost model can be re-tuned with one call to QHICS, but it will not change automatically except if quantiles change.

The learned parameters are representing the cost per unit of processing disk, CPU, and memory, hence it needs to be fine-tuned per databases and hardwares. However, section 9 demonstrates that the ranking score (comparing two vectors to estimate the most beneficial) is great even in zero-shot.

5 Column-oriented feature extraction

The first major component needed for an hypothetical index estimator is a tool to transform a query, represented by a `string`, to data structures we can use to encode an index. We call *features* these data structures instances and the goal of this section is to explain how we decided to extract relevant features to estimate column-oriented index benefit.

The entry of QHICS workload analysis is a *physical query plan* for each query, which we get by calling `EXPLAIN`s to the database. Hence, the work of pushing down and optimizing predicates is let to the database. In the following subsections we explain how we compute our features given a physical query plan. We explain why we capture `SCAN` and `JOIN` operations in section 5.1, explain how we encode a scan in section 5.2, explain how we estimate the number of segments to open in section 5.3, and explain which joins are interesting and what we keep from them in section 5.4.

5.1 Relevant operations

A query plan is composed of many operations that all impact the query execution time: data accesses, grouping operations, distributed computations operations (gather, partition), filters, aggregators, etc. An example of query plan is given in fig. 4. Depending on the considered index types, different operations became relevant:

- Hash indexes allow to accelerate some bottleneck queries by providing direct access to precise values. This mostly speed up scan operations, but can also speed up some special joins.
- Ordered indexes can be leverage to speed up scans, but also for partitioning, gathering or grouping operations.

This work focuses on hash index, because no system supports ordered secondary index in column-oriented databases ([26] is not yet implemented), hence the main operations we need to look at are scans and joins.

5.2 Scan feature extraction

A *scan* operation on column `C` of table `T` with predicate `p`, denoted $\sigma_{\texttt{p}}(\texttt{T.C})$ asks to get all values of `T.C` corresponding to rows where `p` is valid. This is the only operation that may read its data from the main memory.

Given a scan operation, we need metrics that are relevant to the possible benefits of an hash index. As accesses are per columns, we need to measure for each column if it is accessed during the query, and if so, with which features (predicates, ordering, ...).

The scanning system is different in column-oriented databases compared to row-oriented ones. The main difference being that accesses are now per columns and not per rows. Hence, we will capture metrics in a dictionary that maps `(table, column)` tuples to a list of scan features.

When there is no index to leverage, the database engine needs to scan relevant rows on every needed columns. A column is needed if it is in the predicate, or it is the one we project resulting row IDs on. However, note that once one has identified needed row IDs by processing the predicate, accessing projected columns value is not different if you have a hash index or not: in both case you can either leverage seekable hash or just read all needed segments if needed rows are dense in the segments. Hence, we need to **capture metrics on columns used in the scan predicate**, as this is where indexes can increase query processing. In this work, a *predicate* is either:

- A condition operator (leaf), in this work we have implemented `=, RANGE, <, >, <=, >=`.
- A logical operator (node), in this work we have implemented `OR, AND, NOT`.

We make the following assumptions on the query processing:

- Before accessing data, the database first uses the predicate to compute segments needed to be open, i.e. we assume that the database leverages segment skipping.
- When no index is available for a column, the database fully read segments not skipped once, if it needs a value again, it reads it from memory.
- In a query plan, the same column can be accessed using different strategies (index scan or columnar scan) in different scans. However, within a single scan, only one access method is used per column.
- The database is clever and always choose the optimal access path for each column. In particular, it can leverage indexes for some columns and columnar scans for others.

Looking at assumptions made for the accesses, we will capture estimations of:

- The number of unique values filtered by each condition operators along with the cardinality. This will be used to estimate the cost of hash index scan.
- The number of segments that can not be skipped. This will be used to estimate the cost of columnar scan.
- The compressed and decompressed sizes of each needed column. This will be used for both estimations.

Getting exact values for each of these features would be time-consuming, to get cardinality and number of distinct values estimations, we rely on existing works on row-oriented databases [6]. However, there is no work that provides an estimation of the needed number of segments to answer a query, hence we propose our approximation in the following section.

5.3 Estimating needed segments

Segment skipping (see section 2.1.1) allows to skip some block of data (containing around 1,000,000 tuples) based on the predicates of access. Existing works that estimate the cost of a columnar scan [9] make the assumption that all segments need to be open to scan a column, though an efficient systems can optimize the number of segments needed to open: according to [28], given a column C and a value $v \in \texttt{Dom}(D)$ if the system maintains R sorted runs (group of segments without min-max intersections) and that at most k segments have for interval exactly $[v, v]$, the system only needs to open $R + k$ segments to find all necessary rows. Hence, a system that have an efficient merger reducing R and k can optimize the number of segments to open, significantly reducing the complexity of the columnar scan.

As column-oriented databases can have petabytes of data (and then a lot of segments), we don't want to compute the exact number of segments for each query. We will get some information for each column, and derive formula for all predicates using these samples.

Definition 5.1 (Hit factor). We define the *hit factor* (h factor) of a query as the minimum percentage of segments that need to be open to process its answers.

5.3.1 Column hit factor. Given a column `T.C` in the database, we define its hit factor, $h_{\texttt{T.C}}$, has the average on $k \in \texttt{Dom}(T.C)$ of the probability that a segment of `T.C` needs to be open to solve the query `SELECT * FROM T WHERE C = k`. As this value depends on the optimizations of the used database, we get this value *by sampling on each column*, approximating the real mean by an empirical one.

5.3.2 Condition operator hit factor. Given a column `T.C` in the database, and an operator o, to estimate the probability that a given segment need to be open, we first get o estimated number of distinct values k and the hit factor of `T.C`, $h_{\texttt{T.C}}$. To get the number of distinct values, one can rely on existing methods for row-oriented databases,

such as HyperLogLog [7]. Now, the hit factor of o, denoted $h_{\mathtt{T.C}}(o)$, is estimated to $1-(1-h_{\mathtt{T.C}})^k$. To understand this estimation, given a value $v \in \mathtt{Dom(T.C)}$, it is clear that the probability that the segment will not be open by searching for it will approximately be $1-h_{\mathtt{T.C}}$. Given a set of k distinct values of $\mathtt{Dom(T.C)}$, the probability that the segment will not be open by searching for all k values can be estimated to $(1-h_{\mathtt{T.C}})^k$. Finally, the average probability that the segment will need to be open to search for k different values can then be estimated to $1-(1-h_{\mathtt{T.C}})^k$.

5.3.3 Logical operator hit factor. Now that we have a way to estimate the hit factor of each condition operators, to be able to estimate the hit factor of a predicate we need to give estimations for logical operators. For this we use the F-algebra from [24]. This algebra estimates the selectivity for logical operators in a B+-tree row-oriented database:

- **And**. For a logical operator $p_a \wedge p_b$, where p_a and p_b are two predicates over the same table T, we estimate the hit factor of the AND to $h_T(p_a \wedge p_b) := h_T(p_a) \times h_T(p_b)$. The intuition is that we compute the intersection of segments, hence approximately the product of the percentages.
- **Or**. For a logical operator $p_a \vee p_b$, where p_a and p_b are two predicates over the same table T, we estimate the hit factor of the OR to $h_T(p_a \vee p_b) := h_T(p_a) + h_T(p_b) - h_T(p_a) \times h_T(p_b)$. The intuition is that we compute the union by summing the probability and removing the intersection (which was counted twice).
- **Not**. For a logical operator $\neg p_a$, where p_a is a predicate over a table T, we estimate the hit factor of the NOT to $h_T(\neg p_a) := 1 - h_T(p_a)$, as a segment is open by a NOT if and only if it was not open in the input.

Using these approximations, we can now use sampling to get the column hit factor for each columns, and then derive the estimation of hit factor for any predicates at a low cost.

5.4 Highly Selective Joins

Hash indexes can be used to accelerate scans, but they can also be leveraged for special joins. Consider a join $S_1 \bowtie_p S_2$ where S_1 (resp. S_2) is a scan operation on T_1 (resp. the result of another join or a scan on another table), and p is the join predicate. If:

(1) The join predicate is sargable for hash indexes. In our implementation, this is using only composed of equality conditions and $\wedge$ logical operators. This can be extended depending on how the target database system is implemented.
(2) The optimizer performs the join using a hash join, where the hash table is built over S_2.
(3) The number of distinct values stored inside the hash table is low (this is lower than an optimizer threshold).

then the join may be accelerated by an hash index over T_1, on the columns used in p. Here is how we assume the query optimizer to take a decision (it is the decision process of SingleStore [2]):

(1) Compute result of S_2 and build the hash tables over needed columns.
(2) If the NDV is sufficiently small, for each unique values $\vec{v}$, get all lines of T_1 with values $\vec{v}$, and keep those that pass the filter of S_1.
(3) If the NDV is not small enough, perform a normal hash join.

To estimate efficiently the number of distinct values, one can either rely on optimizer's estimates if available, or implement the method of its choice, there is plenty, from heuristics to language models [31]. To detect highly selective joins, QHICS uses a fixed threshold that can be configured. For each highly selective joins, we store cardinality of S_2 and S_1, along with number of distinct values estimation of S_2.

6 Encoding an index

Given a workload $\mathcal{W}$, we can now get a map of scans and a list of joins, with needed metrics. In this section we explain how, given a configuration c of already materialized indexes, we estimate the benefit and the cost of a new index I.

Section 6.1 explains the idea behind encoding the difference directly along with the encoding dimensions, section 6.2 explains how we encode scans on unary index, section 6.3 explains how we extend our scan encoding to multi-column index, section 6.4 explains how we encode joins, and section 6.5 explains how we encode the cost of an index.

6.1 Encoding meaning

Usual works in row-oriented databases use two costs to estimate a plan execution time: the CPU costs and the number of pages needed to be open. This can be seen as a CPU cost and a disk IO cost. However, column-oriented databases rely a lot on memory, and data on disk are compressed where they are not on memory, hence we propose to add an estimation of the memory usage to the encoding, leading to a vector of three dimensions, estimated disk IO, estimated CPU usage, and estimated memory usage.

Traditional What-If takes an index, creates two physical plans (one without and one with index), and returns the difference in estimated cost of each plan. But cost estimations are known to be imprecise, hence we sum two imprecise metrics, leading to possibly more errors. We propose to directly encode the difference into one vector, and to learn to predict the benefit using this encoding vector.

Hence, we encode an index benefit into a three dimensions vector representing the estimated disk IO, CPU, and memory resources difference between without and with the new index. For the cost, we will simply estimate the resources needed to create it (as without index there is nothing to create).

Before moving to our encoding, we define some key notations:

- S_{comp} (resp. S_{decomp}) represents the total compressed (resp. decompressed) size of a column.
- c_{disk} (resp. c_{cpu}, c_{mem}) represents the estimated disk (resp. CPU, memory) usage.
- N_{seg} represents the number of segments of a table (all columns in a table have the same number of segments).
- N represents the number of tuples in a table,
- S_{offset} represents the size of an offset on disk (it is approximately $\lceil \log_2(N/N_{seg}) \rceil + 1$).
- $ndv(\vec{C})$ represents the number of distinct values of $\vec{C}$ in the database.
- $ndv_{res}(\vec{C})$ represents the number of distinct values of $\vec{C}$ that are in the resulting tuples of a query.

- f_{op} (resp. f_{dec}) represents the CPU time needed to do one operation (resp. decompress one value).

As we are working with an already materialized configuration, we need to know which indexes are already materialized. We say that an *index is already present* in a configuration c if it is either in c, or is on one column and this column is part of a multi-column index in c (see section 2.2 for the second condition).

6.2 One column Scan benefit encoding

Suppose that $I = T.C$. If it is already present in c we simply encode its benefit to $\vec{0}$. Otherwise, as discussed in 5.2, we only look at the conditions that concern the column `T.C`. The other accesses are taken in consideration in the scan hit factor approximation.

6.2.1 Encoding of a scan without index. Without index, the database needs to open every matched segments, hence it is reading h percentage of data. We model the disk IO of one scan as c_{disk}:=$h \times S_{\text{comp}}(T.C)$, as it needs to read the compressed data of all segments that need to be open. The memory used estimation is c_{mem}:=$h \times S_{\text{decomp}}(T.C)$, as data read from disk need to be stored decompressed on memory. Finally, the CPU is used twice: it needs to scan data and check for each value if it needs to keep it, costing c^1_{cpu}:=$N_T \times h \times (f_{colscan} + f_{op})$, and it needs to decompress data, costing c^2_{cpu}:=$S_{\text{comp}}(T.C) \times h \times f_{dec}$. We define $E_\emptyset$ as a function that takes an index and a scan, and return the vector composed of $\langle c_{disk}; c^1_{cpu} + c^2_{cpu}; c_{mem}\rangle$.

6.2.2 Encoding of a scan with index. With index, the database can read only needed lines thanks to seekable encoding, however some implementation might multiply the number of lines to read by a seek factor $s_f \geq 1$ (some compression might require more than one value to decompress one precise tuple). Though in general we scan less tuples, there is still some queries (either not selective or selective but returning almost only whole segments) that will lose more times with a hash index compared to without. Moreover, the time to scan one value is longer with a hash index because the scan is not continuous anymore. For these reasons, if a scan has a bigger metric in one of three dimensions of encoding compared to the scan without index, the encoding with index fallback to the one without index.

A scan with index requires first to read the index, and then to read needed data. We assume that:

- Inverted indexes are on disk, but a fixed ratio r_{meta} is cached in memory,
- The global index is read in memory,
- The database does not recheck that rows have the value we are searching for (we trust the index).

Given a scan over $T.C$, we will encode the cost of each conditions over $T.C$, as hash index can only verify them one by one. Given one condition, let ndv_{res} be the number of distinct values of $T.C$ in resulting rows (one can rely on traditional estimations to get this value [6]).

We first need to estimate the size of an inverted index, for this we propose $S_{iv}(T.C)$:=$N_T \times S_{offset}$. We also need to estimate the size of global index that we need to read. Assuming that each level of the global index compress k hash tables into one, the height of the LSM-tree is approximately $\log_k(N_{seg}(T))$ ([28]). At each level we need to read the offset of the hash table to read for each distinct values, consuming S^1_{global}:= $\log_k(N_{seg}(T)) \times ndv_{res} \times S_{offset}$. Then, one need to read one offset per segment that contains the searched value, if h is the hit factor of the condition, this gives S^2_{global}:=$ndv_{res} \times N_{seg}(T) \times h \times S_{offset}$.

The disk will be used to read the inverted index, using c^1_{disk}:=$(1 - r_{meta})S_{iv}$, and to read needed data: c^2_{disk}:=$s_f \frac{N_{res}}{N(T)} \times S_{\text{comp}}(T.C)$. The memory will be used to read the global index, costing c^1_{mem}:=S_{global}, to read cached inverted index, costing c^2_{mem}:=$r_{meta} \times S_{iv}$, and to store decompressed read data, costing c^3_{mem}:=$\frac{N_{res}}{N(T)} \times S_{\text{decomp}}(T.C)$. The CPU is used twice, it needs to probe the hash index for each unique, costing c^1_{cpu}:=$ndv_{res} \times \log_k(N_{seg}(T)) \times f_{op}$, and it needs to decompress data, costing c^2_{cpu} :=$S_{\text{comp}}(T.C) \times \frac{N_{res}}{N(T)} \times f_{dec}$.

At the end, the encoding of the scan with index is the sum over all conditions of the encoding of the condition. However, if this sum is higher than the cost without an index of the scan, we fall back to the case without index. We define E_I as a function that takes an index and a scan, and return the vector composed of the sum over each conditions of $\langle c^1_{disk} + c^2_{disk}; c^1_{cpu} + c^2_{cpu}; c^1_{mem} + c^2_{mem} + c^3_{mem}\rangle$ (or the fallback). The fig. 4 represents a scan encoding.

6.3 Multi-column indexes scan

We consider that multi-column hash indexes over $\vec{C}$ are composed of an index on each column in $\vec{C}$, and a multi-column global hash index exactly on $\vec{C}$. This is the architecture used in [21]. The multi-column global hash index does not require any inverted index apart from the individual ones (see section 2.2 for architecture).

The benefit of a multi-column index is estimated as the sum of all benefits over one column (if one column is already materialized it will bring $\vec{0}$), to which we add the estimated benefit of the multi-column hash index.

A scan can leverage a multi-column hash index if and only if it contains only equality predicates and each column in $\vec{C}$ appears exactly once inside the condition. If multi-column hash index can be leveraged, we model the whole scan as only one index access (for one column index it is one per condition) where:

- The cardinality is estimated to be the minimum of each column access cardinality (it is an upper bound of the real cardinality).
- The number of distinct values is 1 (there is exactly one tuple of $\vec{C}$ that matches the equality condition).
- The hit factor is the hit factor of the scan.

6.4 Join benefit encoding

Given a high selective joins between a scan on T filtered with predicate p and a join tree (or a scan) J that can benefit from an index on $T.\vec{C}$, we estimate the difference between 1) scanning all values of T while filtering and then probing the hash table built on J, and 2) using the hash table on J to query the index over $T.\vec{C}$, filtering all matched rows. As we are encoding joins as a difference, we do not estimate the cost of to process the join J and of building the hash table over it.

6.4.1 Estimating join without index. We simply estimate the cost of the join without index to be the sum of scanning the table T with

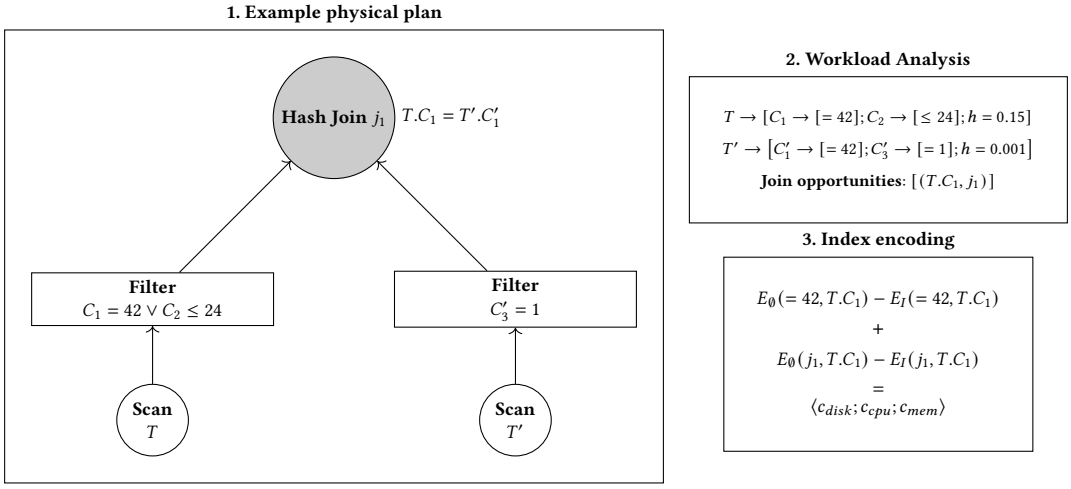


Figure 4: Example of index encoding for a given query. Given a query (see part 1), it is analyzed into two kinds of features: a scan map and a join opportunities list (see part 2). To estimate the benefit of the index $T.C_1$, QHICS gets every related features and sum the delta encoding (see part 3).

predicate p without index, and of probing the hash table over J using matched lines. The first point is evaluated directly using the scan formula, for the second one, we use $c_{cpu} := N_{res}(T) \times f_{op}$, where $N_{res}(T)$ is the number of tuples of T that are valid for predicate p.

6.4.2 Estimating join with index. For each value in the hash table over J, we will seek the hash index to read some data, we will then need to filter each resulting tuples according to the predicate on T. For the first part, let $r := ndv_{res}(J.\vec{C})/ndv(J.\vec{C})$ be an approximation of the ratio of tuples of T that we will match. We model the first part as a scan with index where the cardinality is $r \times N(T.\vec{C})$, and the number of unique values is $r \times ndv(T.\vec{C})$. Then we consider the cost for probing the hash table, this is $c^1_{cpu} := ndv_{res}(J, \vec{C}) \times f_{op}$ and the time to filter each resulting lines of T to be sure it is valid: $c^2_{cpu} := r \times N(T) \times f_{op}$.

6.5 Creation cost encoding

Estimating the benefit of an index is important to know which will increase the more our workload speed. However, some indexes might bring a lot, but they can also cost a lot to create. In this subsection we explain how we encode index creation into a vector. As we are doing hypothetical index benefit estimation, we only estimate the cost to pay at the creation, and we do not take in consideration the maintenance cost. This cost can be taken in consideration in index selection methods.

6.5.1 Cost of creating an index on one column. Let $T.C$ be an index that we would like to create over a configuration c. If $T.C$ is already present in c, it will cost nothing hence its encoding is $\vec{0}$. Otherwise, we will need to read the whole compressed data of $T.C$ on disk, leading to $c^1_{disk} := S_{\text{comp}}(T.C)$. We will also need to write on disk the inverted index, that we already have estimated to $c^2_{disk} := N_T \times S_{offset}$. We need to store in memory the decompressed data of $T.C$, costing $c_{mem} := S_{\text{decomp}}(T.C)$. Finally, the CPU is used three times, it needs to decompress data: $c^1_{cpu} := S_{\text{comp}}(T.C) \times f_{dec}$, to hash every unique values: $c^2_{cpu} := ndv(T.C) \times f_{op}$, and to add each offset to the inverted index: $c^2_{cpu} := N(T) \times f_{op}$. The resulting encoding is $\langle c^1_{disk} + c^2_{disk}; c^1_{cpu} + c^2_{cpu} + c^3_{cpu}; c_{mem} \rangle$.

6.5.2 Cost of creating an index on multiple columns. Let $T.\vec{C}$ be an index that we would like to create over a configuration c. For every column $C \in \vec{C}$:

- If C is not present in c, we add its creation encoding to the cost of $T.\vec{C}$.
- If C is present in c, we only add the size of its global hash index to the memory estimation of the encoding, because we don't have to recreate the whole index again, but we need to read its global hash index to create the multi-column one.

Our estimated encoding can be obtained by summing the encoding of all columns.

7 Cost Model

Using our encoding, we are able to transform a `string` representing an index to a vector of three dimensions capturing essential features and metrics to quantify the index benefit. In this part, we explain how we associate a cost and a benefit to a vector.

7.1 Positive Quantile Regression

As discussed in section 2.3, we leverage Quantile Regression to tune our parameters. More precisely, given a quantile for benefit (resp. for cost) $q_g \in]0; 1[$ (resp. $q_c \in]0; 1[$), we tune a vector $\vec{\theta_b} \in \mathbb{R}^3$ (resp. $\vec{\theta_c} \in \mathbb{R}^3$) on a training dataset.

There are many tools that can be used to tune parameters, however each dimension represents a system resources consumption difference, and negative parameters are not relevant here. Though it can give better solutions, it is fundamentally impossible to have a system that uses a negative time to process a unit of storage. Hence,

we want to solve quantile regression with a positive constraint on $\vec{\theta}$. For this, we model the problem using Linear Programming. Given a matrix of n inputs $X \in \mathbb{R}^{n\times 3}$ and objectives (benefit or cost) $y \in \mathbb{R}^n$, if we seek to optimize $\vec{\theta}$ over quantile q (whether it is a benefit or a cost), we get a $\vec{\theta}$ that is part of a solution of:

$$\begin{aligned} \min_{\vec{\theta},\vec{u},\vec{v}} \quad & \sum_{i=1}^{n} [q\,u_i + (1-q)\,v_i] \\ \text{s.t.} \quad & y_i - X_i\vec{\theta} = u_i - v_i \quad \forall i \\ & \theta_j \geq 0 \quad \forall j \\ & u_i \geq 0, \quad v_i \geq 0 \quad \forall i \end{aligned}$$

It is clear that it gives a solution to quantile regression from the definition of ρ_q given in section 2.3: $\vec{u}$ is used to model the positive r_i and $\vec{v}$ to model the negative r_i. The constraint of positivity over $\vec{\theta}$ implies that we are returning the best solution that has non-negative parameters.

7.2 Implementation

The quantiles can be changed online at the cost of resolving this linear programming problem, which is fast.

A model is trained over all dataset values, however if enough values are available for a table, a model is trained on this table, same goes for every possible indexes. One can configure the minimum number of points to fit a model on a subpart of training data. The more fine-grained model there is, the more precise predictions are.

8 Index selection algorithms

One can now estimate the benefit on an hash index in column-oriented databases using QHICS. We want to use this estimation to select index for a workload. In what follows, we will present two methods to select index in column-oriented databases: greedy and Reinforcement Learning.

8.1 Greedy algorithm

Since the earliest usage of What-If utilities in [4], greedy are used to select index. Given a configuration c and a list of possible new indexes $I_1, \ldots, I_k$, greedy selects the one that has the most estimated benefit using What-If calls.

We have implemented this algorithm using QHICS to select index in a column-oriented database.

8.2 Reinforcement Learning algorithm

8.2.1 Known method. The first method is efficient for fast strategy and can recommend the best choice for one index, but when we want to create multiple indexes, we need to take in consideration Index Interaction [22]. As a solution, [13] proposed to leverage a Deep Q-Network Reinforcement Learning agent [16] to learn to recommend index on an offline workload. The state is a one-hot encoding of index already created, and the action is selecting one index to create. The reward used is the What-If estimation of the created index benefit. This model has a fast training because the reward is the What-If estimated benefit and not the real benefit, but one needs to train one model per choices which can take a long time.

8.2.2 Quantile-aware agent. Due to time restriction, we did not have the time to implement Reinforcement Learning algorithm hence we let this to a future work. However, we propose a new model that we would like to test in the future: a Reinforcement Learning agent that can tune quantiles. This model would need one (long) training but could be re-used while the historical data does not shift from training one. Given a set of possible indexes $\{I_1, \ldots, I_k\}$ and quantiles $\vec{q}$, we propose to model index selection problem given a maximum number of indexes to choose from $n \in \mathbb{N}$ has:

- **State**. For $1 \leq i \leq k$, let $\vec{e_i} \in \mathbb{R}^2$ be the estimations of QHICS of vector I_i. The state is the concatenation of all $\vec{e_i}$, padded with $\vec{0}$ to get a vector of size $n \times 2$, to which we concatenate $\vec{q}$. Hence, the agent is aware of the estimations of QHICS among with the quantiles used.
- **Actions**. Agent can either do nothing (ending the episode), increase/decrease one of the quantile by a fixed ratio, or recommend an index.
- **Reward**. If the agent recommend an index the reward is the real benefit, except if the constraint is broken in which case it is an important penalty. If the agent does not recommend an index the reward is 0 and the episode is stop. If the agent changes quantile the reward is 0 (or a little penalty if one wants to limit the number of computations).

This simple model could reflect interesting trade-off permitted by quantiles, and is let to future work.

9 Implementation & result

To study the performance of QHICS, we propose four tasks that are increasing in difficulty and can correspond to different levels of confidence in the hypothetical index estimations:

- The **ranking score**: the first task that we want is to compare indexes to return the one with the maximum estimation (some What-If even returns a benefit that has no unit and is just used for comparison). Hence, the first metric that we will get is the ranking score. If B^* is the real benefit and $\hat{B}$ the predicted ones, the ranking score is:

  $$rs(B^*, \hat{B}) := \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \delta_{(B_i^* \leq B_j^*) == (\hat{B}_i \leq \hat{B}_j)}$$

  This ranking is useful to recommend the index with the most benefit, though it can not be used in algorithm that needs a precise balance of benefit and cost. This is the first level of hypothetical index that we accept.
- The **underestimation score**: once one starts to have some information but not enough to get precise estimations, it might want to be able to do some recommendations without too much risk. For this we define the underestimation score has the percentage of underestimations with a quantile 0.1, along with the average percentage of overestimation when there is one.
- The **benefit error**: once one has gathered enough data points from runtime, it may want to have an estimation that can be used to get an estimation in seconds of the real increase of efficiency, hence we look at is the average error in benefit estimation (in percentage).

- The **budget recommendation error**: Selecting the best index is hard but only requires to estimate the gain. An important feature of a What-If is to estimate the creation cost of an index. For example, in online index recommendation or offline with budget, it is important to be able to distinguish indexes that are bringing more benefit than they cost from others. If B^* is the real benefit and $\hat{B}$ the predicted ones, and if C^* is the real creation cost and $\hat{C}$ is the predicted ones, we define the budget recommendation error as a tuple containing the percentage of overestimation and:

$$brs := \frac{1}{n} \sum_{i=1}^{n} \delta_{(B_i^* - C_i^* \geq 0) == (\hat{B}_i - \hat{C}_i \geq 0)}$$

As hypothetical index estimation may be leveraged with a budget on the creation time, we will also compute the same scores for the difference between benefit and creation cost, that we call *effective benefit*. We can accept to have imprecise difference, but we want the ranking and underestimation score to be as high as possible.

We will evaluate the precision of QHICS for each of these tasks in three scenarios:

(1) A QHICS well-tuned, with a cost model tuned for each index to recommend (on TPC-H).
(2) A QHICS tuned with the same amount of data but with only one model for all indexes, simulating a situation where QHICS was already running and recommended some indexes but not enough to fine-tune (on TPC-H).
(3) A QHICS tuned over a different schema (TPC-H) and tested on one where it has no data (TPC-DS).

9.1 Configuration and Workload

We have implemented QHICS in Python communicating with a SingleStore cluster running with 16 CPUs AMD Ryzen Threadripper PRO 5965WX 24-Cores and 12GB of ram per node. When a GPU is needed, we use a RTX A5000. The results are over two schemas: TPC-H with scale factor 5, and TPC-DS with scale factor 5. As we have only implemented a toy QHICS, it can not analyze all queries in the TPC benchmarks, hence we have limited our experimentation to workload composed of point-access queries, multi-column conditions scans, and join queries with up to 2 joins. There is no grouping or ordering operations. We have tuned QHICS with 500 points for 27 possible indexes on TPC-H schema, up to 2 columns= per index. TPC-DS is used for the third scenario.

9.2 One model per index result

In this subsection, we consider a QHICS that has enough data to leverage one cost model per index to estimate. This scenario can happen if either the database maintained a history before launching QHICS, or if QHICS has already recommended many indexes. Note that if only some index have enough data to have a regression on them, QHICS can use a general model for indexes without enough data and a precise model for those that have enough (the threshold of minimum data to tune is configurable).

One can see the benefit estimation plot in fig. 5. The ranking score using the 0.5 quantile is really high (97%). The underestimation error on this run is perfect with zero overestimation for quantile 0.1. The benefit error is only of 9% in average, with a maximum of 51%. Finally, the budget recommendation error for quantiles $(0.1, 0.9)$ is the best with 91% of underestimation, and 100% of difference positivity correct estimation. One can see the benefit - creation cost estimation plot in fig. 6.

9.3 One general model result

In this part, we consider a QHICS that has enough data to tune a general model over all possible indexes, but that has not enough points to have a cost model for each index.

One can see the results in fig. 7. The ranking score is still nice with over 92% with quantile 0.5. With quantile 0.1 there is 96% of underestimation and an average overestimation of 8% when one appears. Estimation error however are more present, with an average error of 34% and a maximum error of 48%, though this is still acceptable compared to other heuristics-based What-If. Finally, the budget recommendation error for quantiles $(0.1, 0.9)$ is the best with no overestimation, there is only 67% of difference positivity correct estimation. However, as all errors are missed opportunities and wrong opportunity, we find this acceptable, hence a general model is, we think, enough to be leveraged in index recommendation with budget. One can see the benefit - creation cost estimation plot in fig. 8.

9.4 Default model result (zero-shot)

In this part, we consider a QHICS that has parameters tuned on TPC-H schemas, and that is used to estimate indexes on TPC-DS schemas.

Results can be seen in fig. 9. The ranking score is 81% which is acceptable, especially as there is no training data involved in this score. However, the recommendation is more risky with 58% of overestimation, even with 0.1 quantile used on the training dataset. The average estimation error is of 138%, hence it is clear that one should not use the default value, apart for ranking.

This result is logical as here this is a zero-shot model and the time per operation depends on the database schema (the more data there is, the more the background merger needs to work), hence having a precise estimation depends on the distance between the default parameters conditions and the actual condition.

9.5 Greedy Index Selection

According to previous subsections, we can estimate with an acceptable precision best indexes. To confirm this we have implemented greedy index selection using QHICS benefits.

One can see the results in fig. 10. Though it does not tell more than already showed in previous subsections, it motivates future works in index selections for column-oriented with a reduction of almost 30% of workload processing time.

9.6 QHICS Result

Looking at the presented results, we think that QHICS, thanks to the integration of quantiles, can be deployed in a progressive way.

On a database with no historical data, one should use QHICS with default parameters to compare indexes and create index based on this ranking, though it can not have a valid estimation of the

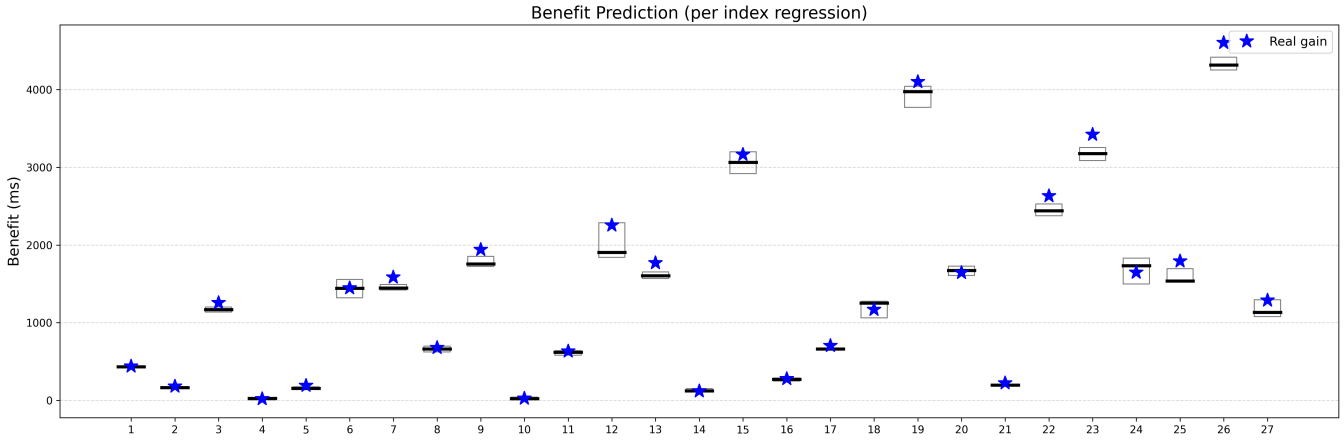


Figure 5: Precision of QHICS when estimating index with an important amount of historical data. The bottom of a box is the 0.1 quantile, the top 0.5 quantile, and middle line 0.3 quantile.

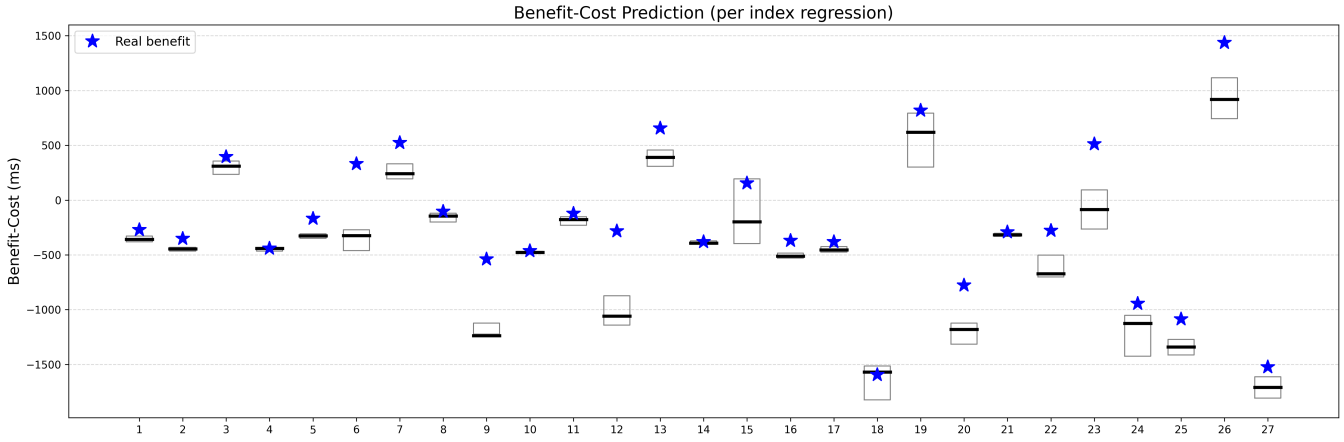


Figure 6: Precision of QHICS when estimating the difference between index benefit and creation cost, with an important amount of historical data. The bottom of a box corresponds to $(0.1, 0.9)$ quantiles, the top $(0.5, 0.5)$ quantiles, and middle line $(0.3, 0.7)$ quantiles.

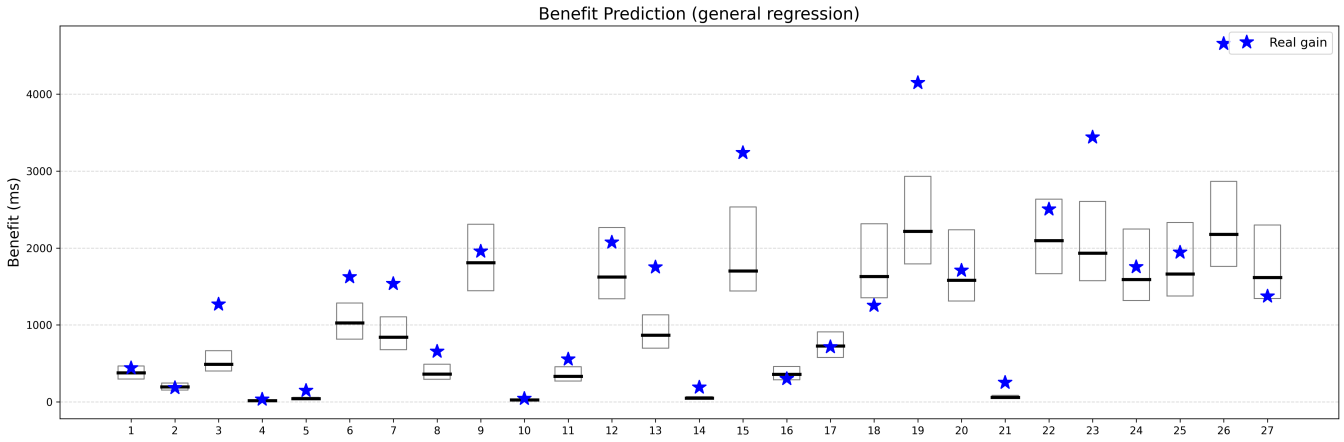


Figure 7: Precision of QHICS when estimating index with one model covering all indexes. The bottom of a box is the 0.1 quantile, the top 0.5 quantile, and middle line 0.3 quantile.

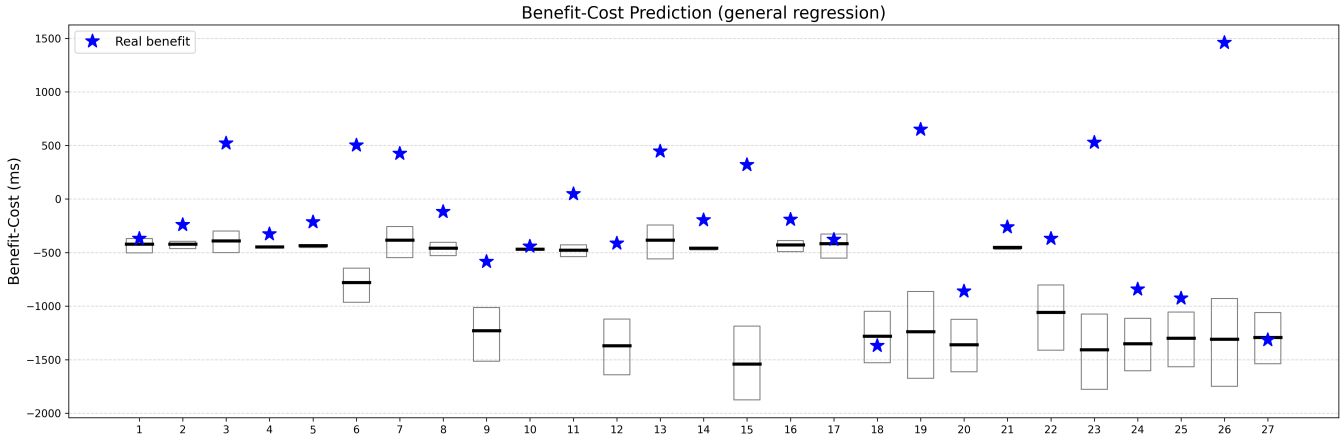


Figure 8: Precision of QHICS when estimating the difference between index benefit and creation cost, with an important amount of historical data. The bottom of a box corresponds to $(0.1, 0.9)$ quantiles, the top $(0.5, 0.5)$ quantiles, and middle line $(0.3, 0.7)$ quantiles.

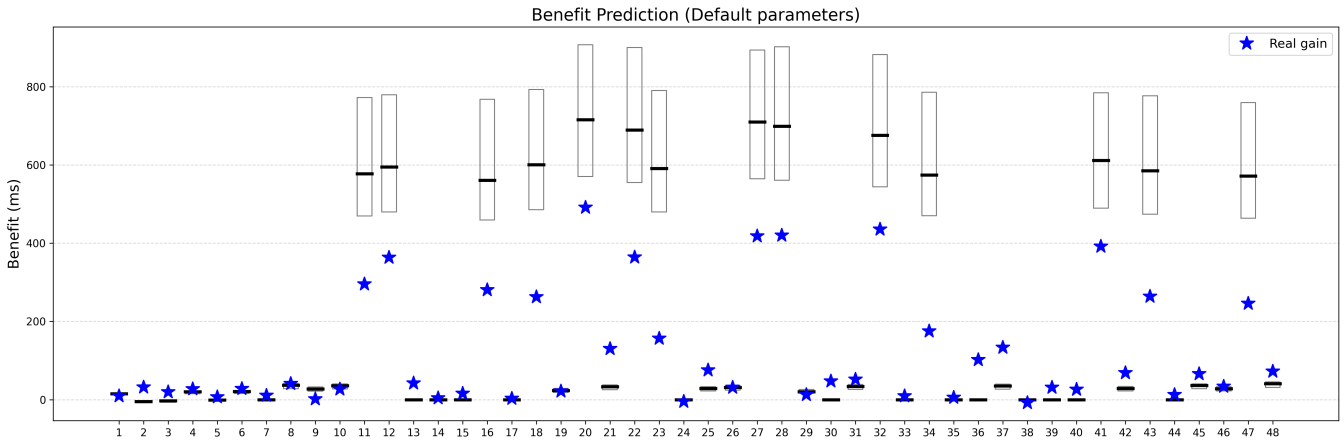


Figure 9: Precision of QHICS when estimating index with one model that is using the default parameters (not related to the database schema). The bottom of a box is the 0.1 quantile, the top 0.5 quantile, and middle line 0.3 quantile.

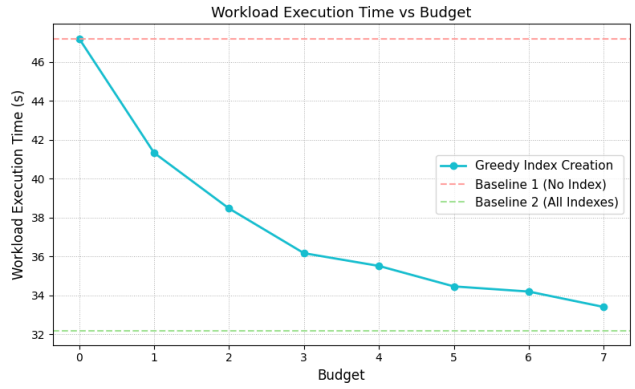


Figure 10: Greedy Index Selection using QHICS benefit estimation.

real benefit, it can use the lowest possible quantiles to reduce the overestimation amplitude.

Once some indexes have been created, one can switch to a general model tuned for its database, using quantiles $(0.5, 0.5)$ if it needs a precise estimation, or lower risk combination (such as $(0.1, 0.9)$) to minimize the risk in the decision.

On a database with a lot of historical data, one can tune QHICS per indexes and then have precise recommendations regardless of the quantiles.

10 Conclusion and limitations

In this work, we provide a first step towards column-oriented database automatic tuning. We propose heuristics for access paths cost estimations, especially providing a way to efficiently get an estimation of the number of segments that are needed to answer a given query. Though index architectures might change and depend on

```
db_wrapper = DbWrapper(...)
db_utilities = DbUtilities(db_wrapper)

whatif = Qhics(db_wrapper,db_utilities)
```

Listing 1: Creating a QHICS instance

the database, we propose foundations that future work could build upon to enhance or adapt to new architectures.

Moreover, we propose to leverage (positive) quantile regression to trade off the precision of the estimation and the number of overestimation, allowing to use our hypothetical index benefit for ranking even if it is using the default parameters.

Even though results of our experiments are satisfying, our implementation has been limited to a subset of SQL, and to hash indexes only, we look toward both extending it to support more index types (once systems capable of having more secondary index types will be developed) and comparing it to orientation-agnostic methods.

Acknowledgements

I want to thank PhD. SHI Jiachen and Prof. CONG Gao for providing the research direction along with precious helps and reviews throughout the research process. I also want to thanks ENS-PSL for funding this internship, and NTU CCDS and Computation Intelligence Lab for giving me access to the resources needed to test QHICS.

A Appendix: How to use QHICS

This appendix provides a tiny example of using QHICS to recommend indexes. The full code is available on GitHub (either the skeleton to extend to any system on the main branch, or the working example on Singlestore in the singlestore branch). In this appendix, we take the example of a E-commerce website under the TPC-H workload.

A.1 Creating a QHICS instance

To create a QHICS instance, you will need to connect your code to your database using two objects, and then create the instance. You can see an example in listing 1.

A.2 Configuring QHICS

Once QHICS is connected, it can not immediatly recommend indexes. You will need to set the workload to use (which can be derived using other tools), and then create the encoder along with the parameters for the cost model. You can do it easily using the syntax in listing 2. If the `fit` parameter is set to `True`, the parameters of the cost model are tuned using historical data, otherwise it will use default parameters.

QHICS can also estimate indexes regarding an existing index configuration, the syntax for editing the configuration is presented in listing 3

```
known_workload = [
    "SELECT c_nationkey FROM CUSTOMER WHERE c_acctbal > 150",
    "SELECT o_orderstatus, o_totalprice, o_shippriority FROM
    ↪  ORDERS WHERE o_orderdate >= '2004-02-04'",
    "SELECT l_shipinstruct FROM LINEITEM WHERE L_ORDERKEY =
    ↪  190209"
]
whatif.set_workload(known_workload)

whatif.create_encoder()
whatif.create_cost_model(fit=True)
```

Listing 2: Getting QHICS ready

```
# Erasing previous configuration
# and writing the new one
existing_indexes =
↪  [Index("LINEITEM",["l_orderkey"],["int"],"Hash")]
whatif.set_configuration(existing_indexes)

# Adding new indexes
# For some sale in France the site
# has created a special index.
new_indexes =
↪  [Index("CUSTOMER",["C_NATIONKEY"],["int"],"Hash")]
whatif.add_to_configuration(new_indexes)

# Removing indexes
# Now that the sales are over, the site can
# remove the special index to save resources.
whatif.remove_from_configuration(new_indexes)
```

Listing 3: Changing index configuration

```
candidate1 = Index("LINEITEM",["L_QUANTITY"],["decimal"])
candidate2 = Index("LINEITEM",["L_LINENUMBER"],["integer"])
whatif.estimate_benefit(candidate1)
whatif.estimate_benefit(candidate2)
```

Listing 4: Recommending using QHICS

A.3 Comparing indexes

Now that QHICS is operational, you can use it to estimate hypothetical indexes. The syntax can be seen in listing 4, note that it will always estimate regarding the workload and configuration in memory.

A.4 Online workload editing

QHICS supports adding (resp. removing) queries from the workload without re-analyzing the whole workload, an example is given in listing 5

References

[1] 2018. *C-Store: A Column-Oriented DBMS* (1 ed.). Association for Computing Machinery, 491–518. doi:10.1145/3226595.3226638

```
# Some sales are happening in France,
# there is new queries related to it:
new_queries = [
    "SELECT n_regionkey FROM NATION WHERE n_name = 'France'",
    "SELECT c_nationkey FROM CUSTOMER WHERE c_nationkey = 5",
    "SELECT O_ORDERKEY FROM ORDERS JOIN CUSTOMER ON
    ↪  c_nationkey = 5 AND O_CUSTKEY = C_CUSTKEY"
]
whatif.add_to_workload(new_queries)

# Now that the sales are over,
# these queries are not index relevant anymore.
whatif.remove_from_workload(new_queries)
```

Listing 5: Online workload edition

[2] 2025. *Highly Selective Joins · SingleStore Helios Documentation.* https://docs.singlestore.com/cloud/create-a-database/columnstore/highly-selective-joins/

[3] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column-Oriented Database Systems. *Proceedings of the VLDB Endowment* 2, 2 (Aug. 2009), 1664–1665. doi:10.14778/1687553.1687625

[4] Surajit Chaudhuri and Vivek Narasayya. [n. d.]. AutoAdmin "What-if Index Analysis Utility. ([n. d.]).

[5] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1241–1258. doi:10.1145/3299869.3324957

[6] Bailu Ding, Vivek Narasayya, and Surajit Chaudhuri. 2024. Extensible Query Optimizers in Practice. *Foundations and Trends® in Databases* 14, 3-4 (2024), 186–402. doi:10.1561/1900000077

[7] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: The Analysis of a near-Optimal Cardinality Estimation Algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH,..., Proceedings (Jan. 2007). doi:10.46298/dmtcs.3545

[8] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. 1997. Index Selection for OLAP. In *Proceedings 13th International Conference on Data Engineering*. IEEE Comput. Soc. Press, Birmingham, UK, 208–219. doi:10.1109/ICDE.1997.581755

[9] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 3072–3084. doi:10.14778/3415478.3415535

[10] Roger Koenker and Kevin F Hallock. [n. d.]. QUANTILE REGRESSION AN INTRODUCTION. ([n. d.]).

[11] Petr Kurapov and Areg Melik-Adamyan. 2023. Analytical Queries: A Comprehensive Survey. arXiv:2311.15730 [cs] doi:10.48550/arXiv.2311.15730

[12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1790–1801. doi:10.14778/2367502.2367518

[13] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. ACM, Virtual Event Ireland, 2105–2108. doi:10.1145/3340531.3412106

[14] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, Athens Greece, 1177–1184. doi:10.1145/1989323.1989448

[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs] doi:10.48550/arXiv.1312.5602

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[18] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. [n. d.]. Efficient Use of the Query Optimizer for Automated Physical Design. ([n. d.]).

[19] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2023. No DBA? No Regret! Multi-Armed Bandits for Index Tuning of Analytical and HTAP Workloads With Provable Guarantees. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (Dec. 2023), 12855–12872. doi:10.1109/TKDE.2023.3271664

[20] Gregory Piatetsky-Shapiro. 1983. The Optimal Selection of Secondary Indices Is NP-complete. *ACM SIGMOD Record* 13, 2 (Jan. 1983), 72–75. doi:10.1145/984523.984530

[21] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 2340–2352. doi:10.1145/3514221.3526055

[22] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. [n. d.]. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. ([n. d.]).

[23] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment* 17, 12 (Aug. 2024), 3731–3744. doi:10.14778/3685800.3685802

[24] P Griffiths Selinger, M M Astrahan, D D Chamberlin, R A Lorie, and T G Price. [n. d.]. Access Path Selection in a Relational Database Management System. ([n. d.]).

[25] Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *Proceedings of the VLDB Endowment* 15, 13 (Sept. 2022), 3950–3962. doi:10.14778/3565838.3565848

[26] Jiachen Shi, Jingyi Yang, Gao Cong, and Xiaoli Li. 2025. NEXT: A New Secondary Index Framework for LSM-based Data Storage. *Proceedings of the ACM on Management of Data* 3, 3 (June 2025), 1–25. doi:10.1145/3725330

[27] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2011. *Database System Concepts* (6th ed ed.). McGraw-Hill, New York, NY.

[28] Alex Skidanov, Anders J. Papito, and Adam Prout. 2016. A Column Store Engine for Real-Time Streaming Analytics. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, Helsinki, Finland, 1287–1297. doi:10.1109/ICDE.2016.7498332

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs] doi:10.48550/arXiv.1706.03762

[30] Yang Wu, Xuanhe Zhou, Yong Zhang, and Guoliang Li. 2024. Automatic Index Tuning: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (Dec. 2024), 7657–7676. doi:10.1109/TKDE.2024.3422006

[31] Xianghong Xu, Xiao He, Tieying Zhang, Lei Zhang, Rui Shi, and Jianjun Chen. 2025. PLM4NDV: Minimizing Data Access for Number of Distinct Values Estimation with Pre-trained Language Models. arXiv:2504.00608 [cs] doi:10.1145/3725336

[32] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (Nov. 2024), 6410–6429. arXiv:2404.15670 [cs] doi:10.1109/TKDE.2024.3389693