Index Estimation on column-oriented database

Supervised by SHI Jiachen, CONG Gao

Clément Rouvroy

September 9, 2025

ENS-PSL (Paris,France), NTU (Singapore, Singapore)

Background and motivation

Why column-oriented storage

- Traditional models store information row by row on disk.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

Why column-oriented storage

- Traditional models store information row by row on disk.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk. Not optimized to read one column.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk. Not optimized to read one column.
- Column-oriented models store information column by column on disk.

Emp Id	**Sell Id**	**Amount**
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk. Not optimized to read one column.
- Column-oriented models store information column by column on disk.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk. Not optimized to read one column.
- Column-oriented models store information column by column on disk. Optimize to read one column.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Why column-oriented storage

- Traditional models store information row by row on disk. Not optimized to read one column.
- Column-oriented models store information column by column on disk. Optimize to read one column.

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT SUM(Amount) FROM Sell
```

Hence, column-oriented databases are used mostly in analytical workloads (finance, e-commerce, data analysis, ...).

An inefficient query on column-oriented

Worst patterns for columnar systems are point accesses. For example:

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

An inefficient query on column-oriented

Worst patterns for columnar systems are point accesses. For example:

seek

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

1. Read `SellId` columns to find the value 7. Remember Row id 3

An inefficient query on column-oriented

Worst patterns for columnar systems are point accesses. For example:

seek

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

1. Read `SellId` columns to find the value 7. Remember Row id 3
2. Read `EmpId` and keep row 3 value.

An inefficient query on column-oriented

Worst patterns for columnar systems are point accesses. For example:

seek

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

1. Read `SellId` columns to find the value 7. Remember Row id 3
2. Read `EmpId` and keep row 3 value.
3. Read `Amount` and keep row 3 value.

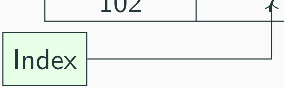
Hash index and seekable columns in MemSQL

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

Hash index and seekable columns in MemSQL

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750




```
SELECT * FROM Sells WHERE SellId = 7
```

1. Use `Hash Index` on `SellId` to find row ids with value 7.

Hash index and seekable columns in MemSQL

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

1. Use `Hash Index` on `SellId` to find row ids with value 7.
2. Access `EmpId` directly on each saved row ids using seekable encoding.

Hash index and seekable columns in MemSQL

Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

1. Use `Hash Index` on `SellId` to find row ids with value 7.
2. Access `EmpId` directly on each saved row ids using seekable encoding.
3. Access `Amount` directly on each saved row ids using seekable encoding.

Hash index and seekable columns in MemSQL

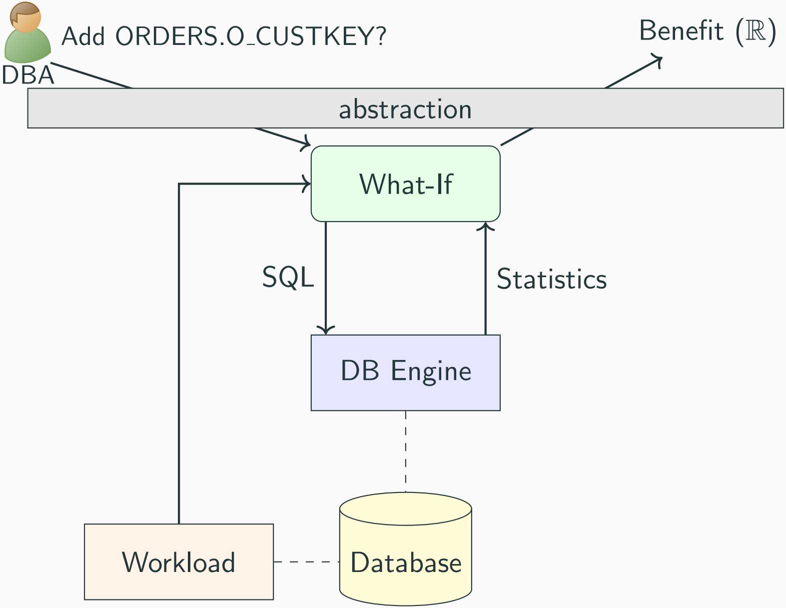
Emp Id	Sell Id	Amount
101	23	1200
102	12	950
102	7	1750

```
SELECT * FROM Sells WHERE SellId = 7
```

1. Use `Hash Index` on `SellId` to find row ids with value 7.
2. Access `EmpId` directly on each saved row ids using seekable encoding.
3. Access `Amount` directly on each saved row ids using seekable encoding.

How to know if we should build an index ?

What-If Hypothetical Index Estimation



Problem Statement

Problem: On a column-oriented database. Given an analytical workload $\mathcal{W} = \{q_1, \ldots, q_w\}$ and a configuration $c = \{I_1, \ldots, I_k\}$,

Problem Statement

Problem: On a column-oriented database. Given an analytical workload $\mathcal{W} = \{q_1, \ldots, q_w\}$ and a configuration $c = \{I_1, \ldots, I_k\}$,

- Build an hypothetical index estimator that estimates the benefit of $c \cup I_{k+1}$ over c on $\mathcal{W}$.

Problem Statement

Problem: On a column-oriented database. Given an analytical workload $\mathcal{W} = \{q_1, \ldots, q_w\}$ and a configuration $c = \{I_1, \ldots, I_k\}$,

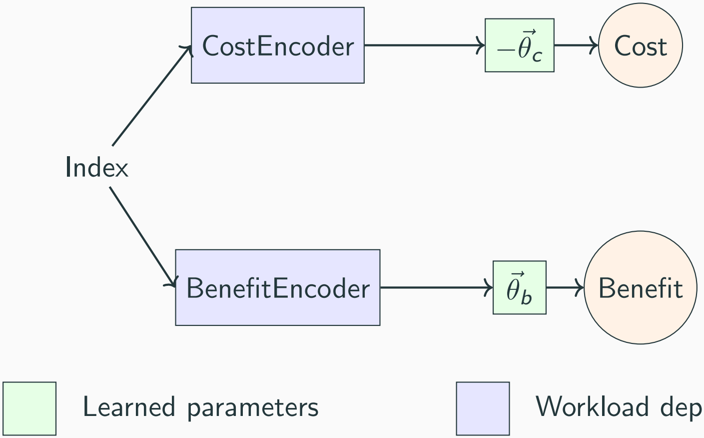
- Build an hypothetical index estimator that estimates the benefit of $c \cup I_{k+1}$ over c on $\mathcal{W}$.

As this is the first work for column-oriented database, we want to provide a foundation that is: heuristics-based, extendable and tunable.

Hypothetical Index Benefit Estimation - Overview

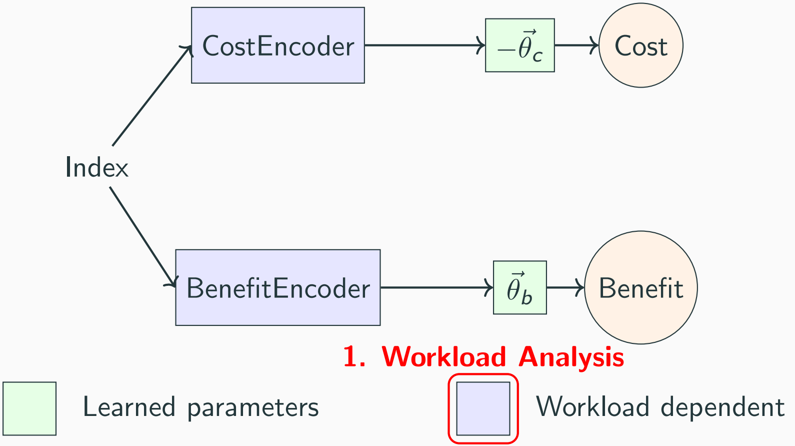
QHICS benefit model

Quantile **H**ypothetical **I**ndex for **C**olumn-oriented **S**torage



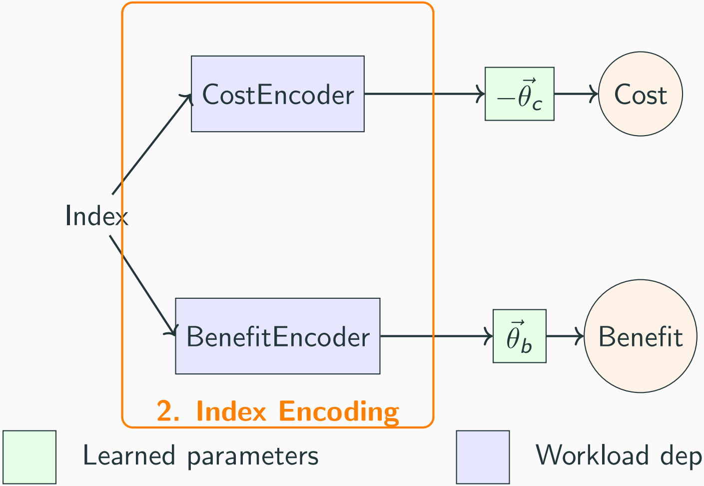
QHICS benefit model

Quantile **H**ypothetical **I**ndex for **C**olumn-oriented **S**torage



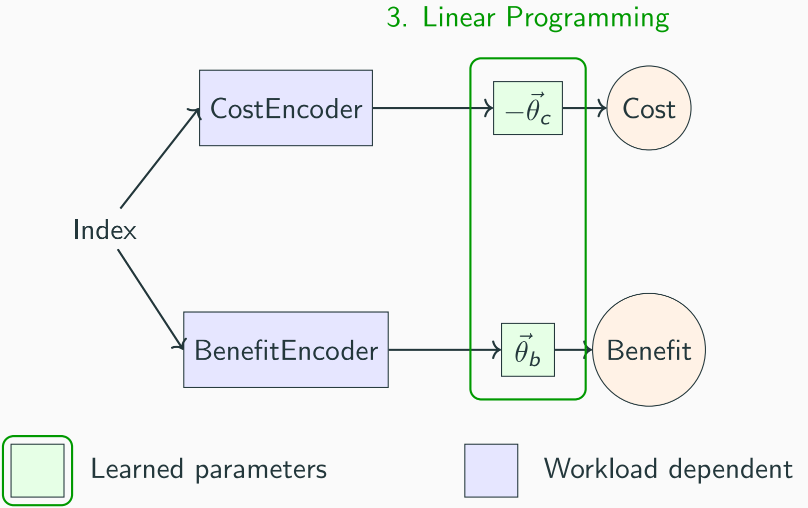
QHICS benefit model

Quantile **H**ypothetical **I**ndex for **C**olumn-oriented **S**torage



QHICS benefit model

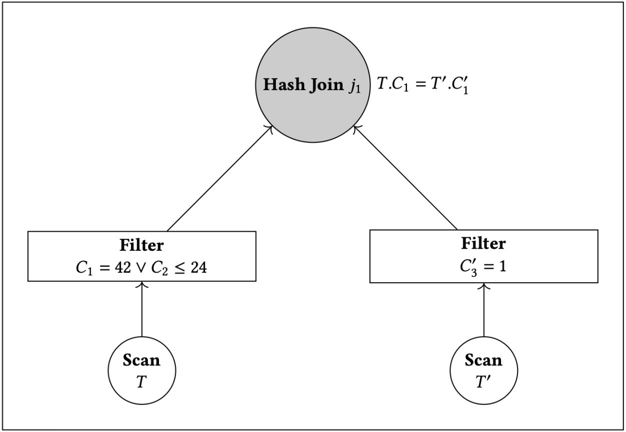
Quantile **H**ypothetical **I**ndex for **C**olumn-oriented **S**torage



Workload Analysis

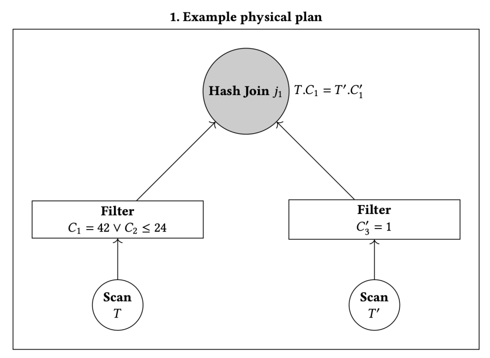
Given a query plan, analyze it and return objects that can be used to estimate the benefit of an index.

1. Example physical plan



What can be accelerated ?

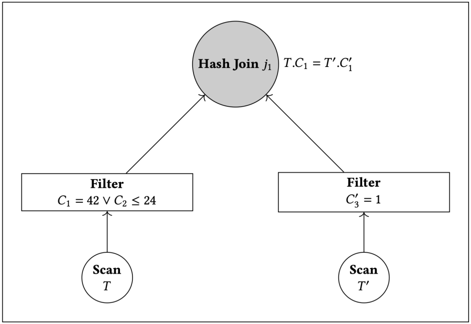
Hash index are used to accelerate predicate checking (e.g. equality).



What can be accelerated ?

Hash index are used to accelerate predicate checking (e.g. equality).

1. Example physical plan



Hence, we estimate the difference of resource consumptions in access paths with and without new index.

Scan without index

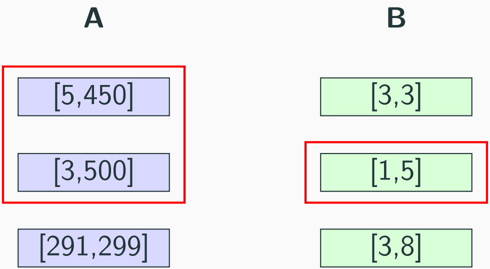
Each column is cut in segments, each columnar segment stores min/max metadata.

A	B
[5,450]	[3,3]
[3,500]	[1,5]
[291,299]	[3,8]

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

Scan without index

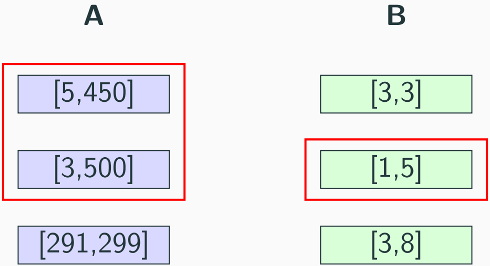
Each column is cut in segments, each columnar segment stores min/max metadata. This is used to skip data.




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

Scan without index

Each column is cut in segments, each columnar segment stores min/max metadata. This is used to skip data.

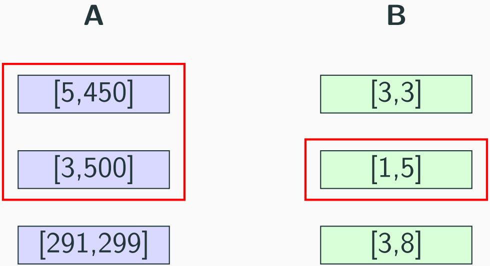



```
SELECT * FROM T WHERE A <= 270 AND B = 2
```

You only need to open the second segment of each column.

Scan without index

Each column is cut in segments, each columnar segment stores min/max metadata. This is used to skip data.




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

You only need to open the second segment of each column.

To estimate the scan cost, we need an estimation of the hit factor.

Hit Factor Approximation

Let $h_C(v)$ for $v \in \texttt{Dom}(C)$ be the percentage of segments needed to get all rows with value v.

Let $h(C)$ be the hit factor of C, defined as $E_{v \in \texttt{Dom}(C)}(h_C(v))$.

Hit Factor Approximation

Let $h_C(v)$ for $v \in \texttt{Dom}(C)$ be the percentage of segments needed to get all rows with value v.

Let $h(C)$ be the hit factor of C, defined as $E_{v \in \texttt{Dom}(C)}(h_C(v))$.

The hit factor of a condition operator that filters k unique values is defined as $1 - (1 - h(C))^k$.

Hit Factor Approximation

Let $h_C(v)$ for $v \in \texttt{Dom}(C)$ be the percentage of segments needed to get all rows with value v.

Let $h(C)$ be the hit factor of C, defined as $E_{v \in \texttt{Dom}(C)}(h_C(v))$.

The hit factor of a condition operator that filters k unique values is defined as $1 - (1 - h(C))^k$.

Now we can use F-algebra:

- For two predicates p_1, p_2, $h(p_1 \wedge p_2) = h(p_1) \times h(p_2)$
- For two predicates p_1, p_2, $h(p_1 \vee p_2) = h(p_1) + h(p_2) - h(p_1) \times h(p_2)$
- For a predicate p_1, $h(\neg p_1) = 1 - h(p_1)$.

Scan processing

We compute a map

$$\texttt{table} \mapsto \mathit{list}\left[\overbrace{\overbrace{\left(\texttt{col} \mapsto \texttt{list(Scan)}\right)}^{\text{Per operator}}, h}^{\text{Per Scan}}\right]$$

Scan processing

We compute a map

$$\texttt{table} \mapsto \mathit{list}\left[\overbrace{\overbrace{\left(\texttt{col} \mapsto \texttt{list(Scan)}\right)}^{\text{Per operator}}, h}^{\text{Per Scan}}\right]$$

Let the query $\sigma_{C_1=17 \vee C_2<=35}(A) \bowtie_C \sigma_{C_3=90}(B)$:

- $\texttt{A} \rightarrow [(C_1 \rightarrow [=17]; C_2 \rightarrow [\leq 35])\,, 0.3]$
- $\texttt{B} \rightarrow [(C_3 \rightarrow [=90])\,, 0.1]$

Scan processing

We compute a map

$$\texttt{table} \mapsto \mathit{list}\left[\overbrace{\overbrace{\left(\texttt{col} \mapsto \texttt{list(Scan)}\right)}^{\text{Per operator}}, h}^{\text{Per Scan}}\right]$$

Let the query $\sigma_{C_1=17 \vee C_2<=35}(A) \bowtie_C \sigma_{C_3=90}(B)$:

- $\texttt{A} \rightarrow [(C_1 \rightarrow [=17]; C_2 \rightarrow [\leq 35]), 0.3]$
- $\texttt{B} \rightarrow [(C_3 \rightarrow [=90]), 0.1]$

It is sufficient to estimate benefit as:

- Without index one needs the hit factor of each column.

Scan processing

We compute a map

$$\texttt{table} \mapsto \textit{list}\left[\overbrace{\overbrace{(\texttt{col} \mapsto \texttt{list(Scan)})}^{\text{Per operator}}, h}^{\text{Per Scan}}\right]$$

Let the query $\sigma_{C_1=17 \vee C_2<=35}(A) \bowtie_C \sigma_{C_3=90}(B)$:

- $\texttt{A} \rightarrow [(C_1 \rightarrow [= 17]; C_2 \rightarrow [\leq 35]), 0.3]$
- $\texttt{B} \rightarrow [(C_3 \rightarrow [= 90]), 0.1]$

It is sufficient to estimate benefit as:

- Without index one needs the hit factor of each column.
- With index one needs estimation of metrics on each condition.

Scan processing

We compute a map

$$\texttt{table} \mapsto \textit{list}\left[\overbrace{\left(\overbrace{\texttt{col} \mapsto \texttt{list(Scan)}}^{\text{Per operator}}, h\right)}^{\text{Per Scan}}\right]$$

Let the query $\sigma_{C_1=17 \vee C_2<=35}(A) \bowtie_C \sigma_{C_3=90}(B)$:

- $\texttt{A} \rightarrow [(C_1 \rightarrow [=17]; C_2 \rightarrow [\leq 35]), 0.3]$
- $\texttt{B} \rightarrow [(C_3 \rightarrow [=90]), 0.1]$

It is sufficient to estimate benefit as:

- Without index one needs the hit factor of each column.
- With index one needs estimation of metrics on each condition.

We also capture Highly Selective Joins (example in appendix).

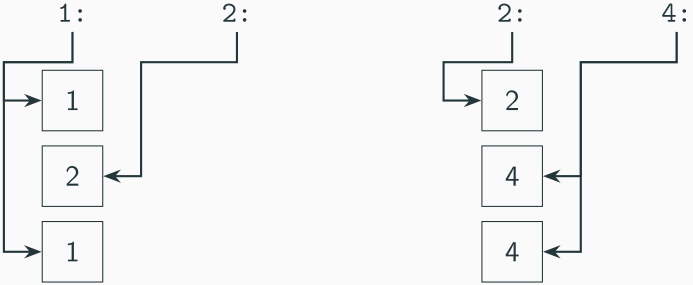
Index Encoding

Hybrid LSM index format

- For each segment, an inverted index (a dict) is built mapping from column values to offset.

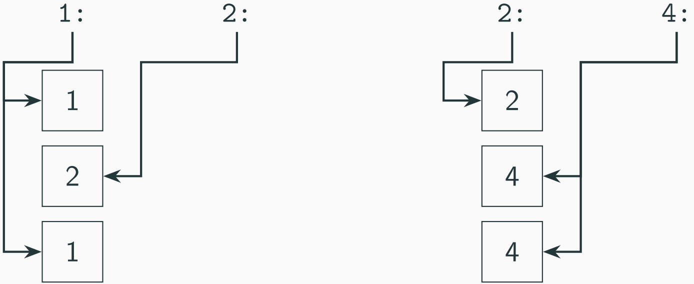
Hybrid LSM index format

- For each segment, an inverted index (a dict) is built mapping from column values to offset.



Hybrid LSM index format

- For each segment, an inverted index (a dict) is built mapping from column values to offset.



- A global index (LSM-based hash tables) is built to map from value to a list of inverted index positions. Here, 2 maps to `[seg:1,offset:2; seg:2, offset:0]`

Encoding creation

We encode an index into a vector (disk IO, CPU, MEM).

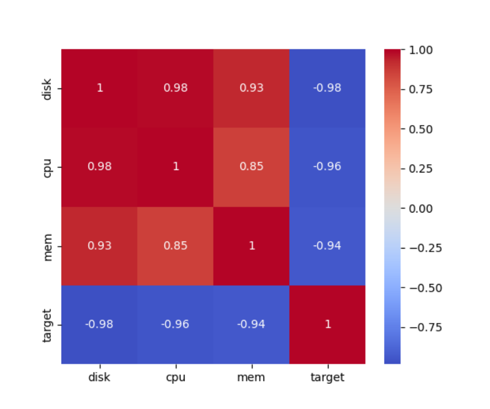
Encoding creation

We encode an index into a vector (disk IO, CPU, MEM).

- Reading compressed data and writing inverted index on disk.
- Using CPU to decompress and hash each value.
- Storing uncompressed data in memory.

Encoding creation

We encode an index into a vector (disk IO, CPU, MEM).



- Reading compressed data and writing inverted index on disk.
- Using CPU to decompress and hash each value.
- Storing uncompressed data in memory.

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

- Read the global hash index

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

- Read the global hash index
- Open inverted indexes.

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

- Read the global hash index
- Open inverted indexes.
- Open matching values.

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

- Read the global hash index
- Open inverted indexes.
- Open matching values.

Without Index

For each scan concerning C:

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

- Read the global hash index
- Open inverted indexes.
- Open matching values.

Without Index

For each scan concerning C:

- Read h percentage of column data.

Encoding benefit

To encode the benefit, we encode the access pathes and joins difference. Let C be the columns in the new index that are not already indexed.

With Index

For each scan operation concerning C:

- Read the global hash index
- Open inverted indexes.
- Open matching values.

Without Index

For each scan concerning C:

- Read h percentage of column data.
- Check each value.

Tuning QHICS

From Encodings to Execution Time

What we have: Resource vectors for cost and benefit.

$\vec{e}_{cost} = (c_{disk}, c_{cpu}, c_{mem})$ — $\vec{e}_{benefit} = (\Delta c_{disk}, \Delta c_{cpu}, \Delta c_{mem})$

From Encodings to Execution Time

What we have: Resource vectors for cost and benefit.

$\vec{e}_{cost} = (c_{disk}, c_{cpu}, c_{mem})$ — $\vec{e}_{benefit} = (\Delta c_{disk}, \Delta c_{cpu}, \Delta c_{mem})$

What we want: A model to predict execution time.

- We use a linear model with a learned weight vector $\vec{\theta}$.

$$\text{Time} \approx \vec{e} \cdot \vec{\theta}$$

From Encodings to Execution Time

What we have: Resource vectors for cost and benefit.

$$\vec{e}_{cost} = (c_{disk}, c_{cpu}, c_{mem}) \quad — \quad \vec{e}_{benefit} = (\Delta c_{disk}, \Delta c_{cpu}, \Delta c_{mem})$$

What we want: A model to predict execution time.

- We use a linear model with a learned weight vector $\vec{\theta}$.

$$\text{Time} \approx \vec{e} \cdot \vec{\theta}$$

We learn two separate vectors for our two objectives:

- Creation Cost Time $\approx \vec{e}_{cost} \cdot \vec{\theta}_c$
- Query Benefit Time $\approx \vec{e}_{benefit} \cdot \vec{\theta}_b$

From Encodings to Execution Time

What we have: Resource vectors for cost and benefit.

$$\vec{e}_{cost} = (c_{disk}, c_{cpu}, c_{mem}) \quad — \quad \vec{e}_{benefit} = (\Delta c_{disk}, \Delta c_{cpu}, \Delta c_{mem})$$

What we want: A model to predict execution time.

- We use a linear model with a learned weight vector $\vec{\theta}$.

$$\text{Time} \approx \vec{e} \cdot \vec{\theta}$$

We learn two separate vectors for our two objectives:

- Creation Cost Time $\approx \vec{e}_{cost} \cdot \vec{\theta}_c$
- Query Benefit Time $\approx \vec{e}_{benefit} \cdot \vec{\theta}_b$

How to learn the optimal weights $\vec{\theta}_c$ and $\vec{\theta}_b$?

What is optimal ?

We can minimize average error. Overestimation can reduce database performance.

What is optimal ?

We can minimize average error. Overestimation can reduce database performance. We want to trade-off prediction precision and overestimations.

What is optimal ?

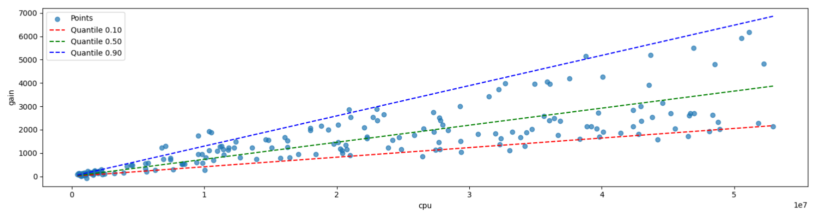
We can minimize average error. Overestimation can reduce database performance. We want to trade-off prediction precision and overestimations.

We propose to use Quantile Regression. It minimizes a loss where overestimation has weight $q-1$ and underestimation weight q.

What is optimal ?

We can minimize average error. Overestimation can reduce database performance. We want to trade-off prediction precision and overestimations.

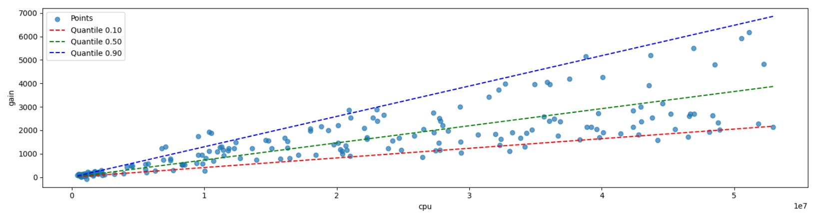
We propose to use Quantile Regression. It minimizes a loss where overestimation has weight $q - 1$ and underestimation weight q.



What is optimal ?

We can minimize average error. Overestimation can reduce database performance. We want to trade-off prediction precision and overestimations.

We propose to use Quantile Regression. It minimizes a loss where overestimation has weight $q - 1$ and underestimation weight q.



We restrict $\vec{\theta}$ to positive values $\Rightarrow$ Linear Programming.

Results

Context

We have implemented a toy QHICS to recommend indexes on Singlestore. It is limited to:

- `=,<=,>=,<,>,RANGE` condition predicates.
- `AND,OR,NOT` logical predicates.
- No grouping or ordering operators.

Context

We have implemented a toy QHICS to recommend indexes on Singlestore. It is limited to:

- `=,<=,>=,<,>,RANGE` condition predicates.
- `AND,OR,NOT` logical predicates.
- No grouping or ordering operators.

We use workloads over schema of TPC-H, using custom queries containing:

- Point accesses,
- Random multi-column conditions queries.
- Queries with up to 2 joins.

Context

We have implemented a toy QHICS to recommend indexes on Singlestore. It is limited to:

- `=,<=,>=,<,>,RANGE` condition predicates.
- `AND,OR,NOT` logical predicates.
- No grouping or ordering operators.

We use workloads over schema of TPC-H, using custom queries containing:

- Point accesses,
- Random multi-column conditions queries.
- Queries with up to 2 joins.

We also used another schema (TPC-DS) to test transferability.

Results

Configuration	Average error	Ranking	Underestimation
Lot	9%	97%	91%
Few	34%	92%	96%
None	Don't	81%	Don't

Table 1: Range of QHICS depending on the number of points

Results

Configuration	Average error	Ranking	Underestimation
Lot	9%	97%	91%
Few	34%	92%	96%
None	Don't	81%	Don't

Table 1: Range of QHICS depending on the number of points

Quantile allows to mitigate the needs of a huge starting dataset, and to fine tune over time.

Conclusion

In this work we have proposed:

- Heuristics for the number of segments needed for a query.
- Hypothetical Index estimation for column-oriented storage.
- The first use of Quantile Regression for risk-gain trade-off in WhatIf. Demonstrating that quantiles can be used to give early estimations while the system is being tuned on runtime information.

Appendix

Highly selective joins

A (unfiltered)		**B** (filtered)
Card=100,000 NDV(C) = 10,000	$\bowtie_C$	Card=10,000 NDV(C) = 50

Highly selective joins

A (unfiltered)

Card=100,000
NDV(C) = 10,000

$\bowtie_C$

B (filtered)

Card=10,000
NDV(C) = 50

Once hash table on B is built:

- Without index, filter all A and probe H for each remaining tuples.

Highly selective joins

A (unfiltered)		**B** (filtered)
Card=100,000 NDV(C) = 10,000	$\overset{C}{\bowtie}$	Card=10,000 NDV(C) = 50

Once hash table on B is built:

- Without index, filter all A and probe H for each remaining tuples.
- With index, probe A 50 times and filter matched tuples.

Highly selective joins

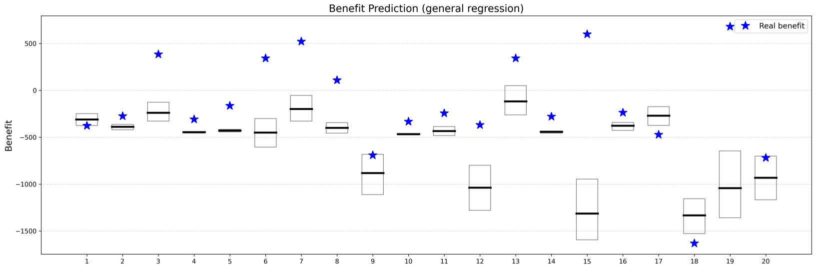
A (unfiltered)	C	**B** (filtered)
Card=100,000 NDV(C) = 10,000	$\bowtie$	Card=10,000 NDV(C) = 50

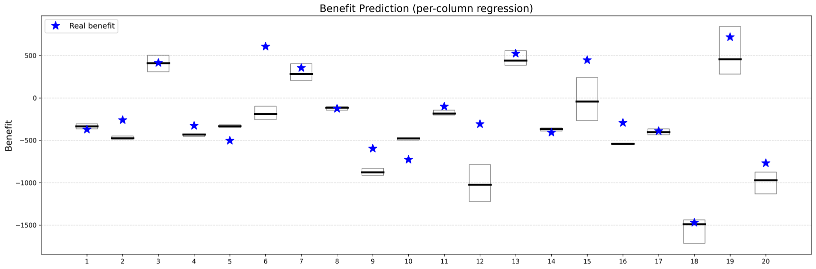
Once hash table on B is built:

- Without index, filter all A and probe H for each remaining tuples.
- With index, probe A 50 times and filter matched tuples.

QHICS captures this in its `JOIN` processing algorithm.

Visual example





Syntax of QHICS - creatung an instance

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

whatif = Qhics(db_wrapper,db_utilities)
```

Syntax of QHICS - Workload

```python
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)
```

Syntax of QHICS - Configuration

```

new_indexes = [Index("CUSTOMER",["C_NATIONKEY"],["int"],"Hash")]
whatif.add_to_configuration(new_indexes)
whatif.remove_from_configuration(new_indexes)
```

Syntax of QHICS - Estimating

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

Positive Quantile Regression

We only accept positive coefficient for quantile regression, as we are modeling system costs. Hence, we need to write it as a Linear Programming problem:

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

Appendix - Heuristics

Notations

- $S_{comp}(T.C)$ is the compressed size of the column.
- $S_{uncomp}(T.C)$ is the uncompressed size of the column.
- N_T is the number of tuples of the table.
- f_{op} is the time needed for one *op*.
- h is the hit factor.
- S_{offset} is the size of an offset in an inverted index.
- N_{res} is the number of resulting rows of a query.
- $ndv(T.C)$ is the number of distinct values of the column.

Creation Encoding

- $c_{disk} := S_{compressed} + \text{seg} \times S_{IV}$

Creation Encoding

- $c_{disk} := S_{compressed} + \text{seg} \times S_{IV}$
- $c_{cpu} := S_{compressed} \times c_{decompress} + (N_{tuple} + \text{ndv}) \times c_{op}$

Creation Encoding

- $c_{disk} := S_{compressed} + \text{seg} \times S_{IV}$
- $c_{cpu} := S_{compressed} \times c_{decompress} + (N_{tuple} + \text{ndv}) \times c_{op}$
- $c_{mem} := S_{uncompressed}$

Creation Encoding

- $c_{disk} := S_{compressed} + \text{seg} \times S_{IV}$
- $c_{cpu} := S_{compressed} \times c_{decompress} + (N_{tuple} + \text{ndv}) \times c_{op}$
- $c_{mem} := S_{uncompressed}$
- For multi-column indexes we sum uni-column costs.

Gain encoding (Without Index)

For an index over $T.C$. If a scan reads h percentage of segments:

- Read $c_{disk}{:=}h \times S_{\text{comp}}(T.C)$ on the disk.
- Store $c_{mem} h \times S_{\text{uncomp}}(T.C)$ uncompressed data on the memory.
- Scan and check value using $c^1_{cpu}{:=}N_T \times h \times (f_{colscan} + f_{op})$.
- Uncompress data using $c^2_{cpu}{:=}S_{\text{comp}}(T.C) \times h \times f_{dec}$

Gain encoding (With Index) (1/3)

Assumptions:

- Inverted index are on disk, but a portion r_{meta} is cached in memory,
- The global index is read in memory *but this can be changed easily with a fixed parameter*,
- The database does not reverify that values have the one we are searching for (we trust the index).
- Leveraging seekable encoding adds a s_f seek factor to data needed.

Gain encoding (With Index) (2/3)

For an index over $T.C$, for each condition over $T.C$.

- The Inverted Index is estimated to $S_{iv}{:=}N \times S_{offset}$.
- At each level of the Global Index, we need to read each needed offsets: $S^1_{global}{:=}\log_k(N_{seg}(T)) \times \text{ndv}(T) \times S_{offset}$
- We need to read one offset per segments that contains the searched value: $S^2_{global}{:=}\text{ndv}(T) \times N_{seg}(T) \times h \times S_{offset}$

Gain encoding (With Index) (3/3)

For an index over $T.C$, for each equality condition over $T.C$.

- Read inverted index using the disk $c_{disk}^{1}{:=}(1 - r_{meta})S_{iv}$
- Read the remaining inverted index part using memory $c_{mem}^{1}{:=}r_{meta} \times S_{iv}$
- Read the global index using memory $c_{mem}^{2}{:=}S_{global}$
- Read needed data using the disk $c_{disk}^{2}{:=}s_f \frac{N_{res}}{N(T)} \times S_{\mathrm{comp}}(T.C)$
- Store all read data on memory $c_{mem}^{3}{:=}\frac{N_{res}}{N(T)} \times S_{\mathrm{uncomp}}(T.C)$
- Use the CPU to probe hash index $c_{cpu}^{1}{:=}\mathrm{ndv} \times \log_k(N_{seg}(T)) \times f_{op}$
- Use the CPU to decompress results $c_{cpu}^{2}{:=}S_{\mathrm{comp}}(T.C) \times \frac{N_{res}}{N(T)} \times f_{dec}$