An Introduction to TiDB's Query Optimizer

hanfei@pingcap.com

September 23, 2017

What is TiDB ?

TiDB is a distributed SQL database. Inspired by the design of Google F1 and Google Spanner, TiDB supports the best features of both traditional RDBMS and NoSQL.

- Horizontal Scalability
- Asynchronous Schema Changes
- Consistent Distributed Transaction
- Compatible with MySQL Protocol
- NewSQL over TiKV

Overview

- Logical Optimization
  - Column Pruning
  - Projection Elimination
  - Correlated Subquery Unnested
  - Predicate Pushing Down
  - Eager Aggregation (Experimental)
  - TopN Pushing Down
- Physical Optimization
  - Statistics
  - Dynamic Program Based on Interesting Order
  - TopN Query Support

Correlated Subquery Unnested

In SQL92, There are three types of subqueries:

- Scalar Valued. A relational query outputs a single-column table.
- Existential Test. An arbitrary query is enclosed as exists(Q) and its result is a boolean.
- Quantified Comparison. Check a scalar expression against a set of values returned by a single-column table. The form is <Expr> <cmp> ALL|SOME Q.

Correlated Subquery Unnested

In SQL92, There are three types of subqueries:

- Scalar Valued. A relational query outputs a single-column table.
- Existential Test. An arbitrary query is enclosed as exists(Q) and its result is a boolean.
- Quantified Comparison. Check a scalar expression against a set of values returned by a single-column table. The form is <Expr> <cmp> ALL|SOME Q.

A recommended paper:
Orthogonal Optimization of Subqueries and Aggregation

Correlated Subquery Unnested

A useful tool: Represent parameterized execution algebraically

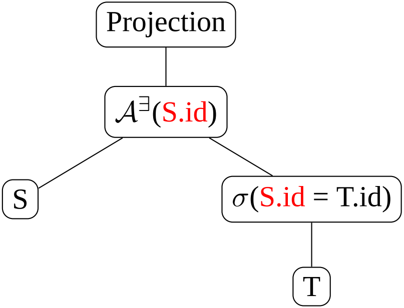
$$R\,\mathcal{A}^{\otimes}\,E = \bigcup_{r \in R} (\{r\} \bigotimes E(r))$$

Apply takes a relational input R and a parameterized expression $E(r)$; it evaluates expression E for each row $r \in R$, and collects the results. $\otimes$ is either left outer join, left semi-join or left anti-join, besides, TiDB introduces left outer semi-join to process SQL like CASE WHEN ... in (...).

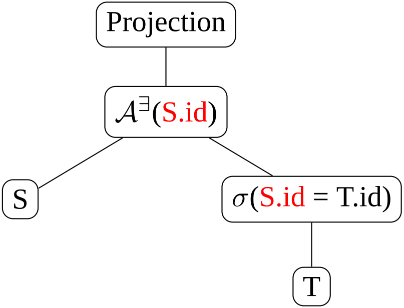
Rules to Remove Correlation

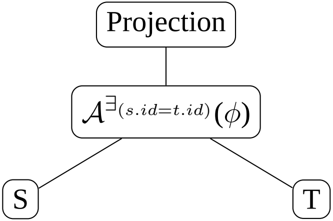
$$
\begin{aligned}
R\,\mathcal{A}^{\otimes}\,E &= R \otimes_{\text{true}} E && (1)\\
& \quad \text{if no parameters in } E \text{ resolved from } R \\
R\,\mathcal{A}^{\otimes}\,(\sigma_p E) &= R \otimes_p E && (2)\\
& \quad \text{if no parameters in } E \text{ resolved from } R \\
R\,\mathcal{A}^{\times}\,(\sigma_p E) &= \sigma_p(R\,\mathcal{A}^{\times}\,E) && (3)\\
R\,\mathcal{A}^{\times}\,(\pi_v E) &= \pi_{v \bigcup \text{cols}(R)}(R\,\mathcal{A}^{\times}\,E) && (4)\\
R\,\mathcal{A}^{\times}\,(E_1 \bigcup E_2) &= (R\,\mathcal{A}^{\times}\,E_1) \bigcup (R\,\mathcal{A}^{\times}\,E_2) && (5)\\
R\,\mathcal{A}^{\times}\,(E_1 - E_2) &= (R\,\mathcal{A}^{\times}\,E_1) - (R\,\mathcal{A}^{\times}\,E_2) && (6)\\
R\,\mathcal{A}^{\times}\,(E_1 \times E_2) &= (R\,\mathcal{A}^{\times}\,E_1) \Join_{R.key} (R\,\mathcal{A}^{\times}\,E_2) && (7)\\
R\,\mathcal{A}^{\times}\,(\mathcal{G}_{A,F} E) &= \mathcal{G}_{A \bigcup R.key, F}(R\,\mathcal{A}^{\times}\,E) && (8)\\
R\,\mathcal{A}^{\times}\,(\mathcal{G}^1_F E) &= \mathcal{G}_{R.key, F'}(R\,A^{LOJ}\,E) && (9)
\end{aligned}
$$

SELECT * FROM S WHERE
EXISTS(SELECT * FROM T WHERE T.id = S.id)

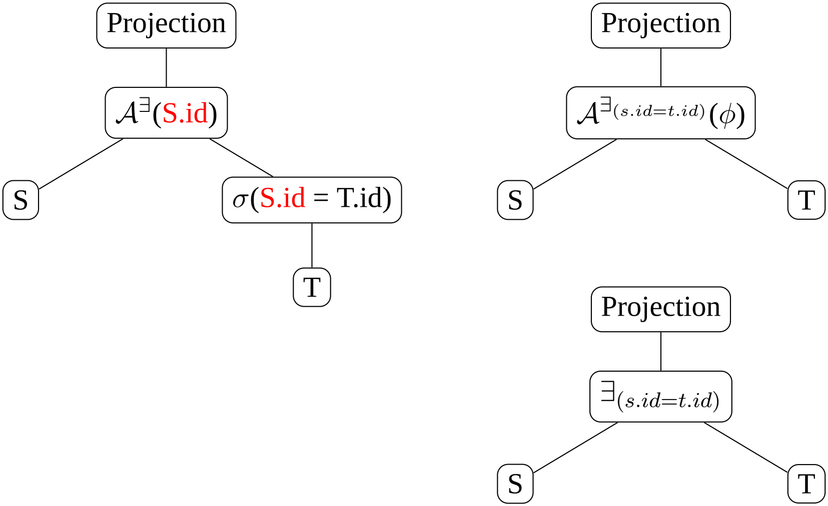


SELECT * FROM S WHERE
EXISTS(SELECT * FROM T WHERE T.id = S.id)





SELECT * FROM S WHERE
EXISTS(SELECT * FROM T WHERE T.id = S.id)



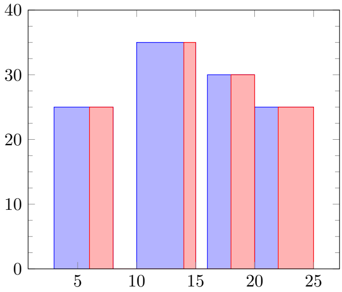
Histograms

Key Issues for Histograms:

- Bucketing Scheme
  - equi-width
  - equi-depth
  - V-Optimal
- Estimation Scheme
  - Continuous-Value Assumption
  - Uniform-Spread Assumption
  - Four-Level Trees
- Efficiency
  Here efficiency refers to the space and time requirements for constructing the histogram.
- Incremental Maintenance

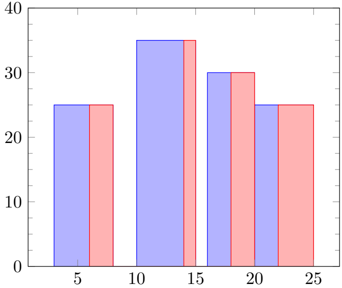
Histogram in TiDB

TiDB maintains lower bound, upper bound, repeats, count for every bucket and ndv, null value for every histogram



Histogram in TiDB

TiDB maintains lower bound, upper bound, repeats, count for every bucket and ndv, null value for every histogram



Repeats can help us find out hot data!

How to Build Histograms

There are two types of data stored in TiDB, either of them can build histograms distributedly

- Index Data
  Index data is aligned orderly, so we can build histograms by an ordered row stream.

How to Build Histograms

There are two types of data stored in TiDB, either of them can build histograms distributedly

- Index Data
  Index data is aligned orderly, so we can build histograms by an ordered row stream.
- Row Data
  Row data is aligned arbitrarily, so we sample firstly, then build histograms for every column.

How to Build Histograms

There are two types of data stored in TiDB, either of them can build histograms distributedly

- Index Data
  Index data is aligned orderly, so we can build histograms by an ordered row stream.
- Row Data
  Row data is aligned arbitrarily, so we sample firstly, then build histograms for every column.
  We use the reservoir algorithm which is the most well known SRSWoR algorithm. The algorithm selects the first n rows from the input stream and puts them into reservoir. Subsequently, when the ith row in the stream arrives ($i > n$), the row is accepted into the reservoir with probability n/i, replacing a randomly chosen reservoir element.

Sketches for Distinct Value Queries

The Flajolet-Martin sketch is a method for approximating the distinct count in small space. During the execution of the algorithm, an integer variable l records the current level of the sampling. Each item in the input is hashed using a function h which obeys

$$Pr[h(i) = j] = 2^{-j}$$

Initially, $l = 1$ and each item is sampled. When the sample is full (i.e., it contains more than k distinct items), the level is increased by 1. The sample is then pruned: all items in the sample whose hash value is less than the current value of l are rejected. Note that when l increases by 1, the effective sampling rate halves, and so we expect the sample to decrease in size to approximately $k/2$. At any moment, the current number of distinct items in the whole sequence so far can be estimated as $s * 2^l$, where s denotes the current number of items in the sample.

Stats Maintenance

There are many items to maintain:

- Table Level
  - Count
- Column/Index Level
  - NullCount
  - Number of Distinct Values
  - Histograms

Stats Maintenance

There are many items to maintain:

- Table Level
  - Count
- Column/Index Level
  - NullCount
  - Number of Distinct Values
  - Histograms

How to notify each TiDB node after updating ?

Stats Maintenance

There are many items to maintain:

- Table Level
  - Count
- Column/Index Level
  - NullCount
  - Number of Distinct Values
  - Histograms

How to notify each TiDB node after updating ?

Does every column need histograms ?

Stats Meta

Version	Table	Count
1	t1	600
2	t3	999
3	t2	150

Stats Meta

Version	Table	Count
1	t1	600
2	t3	999
3	t2	150

Stats Histogram

Version	Table	Column	NDV	NullCount
1	t2	c1	340	10
2	t2	c2	55	0
3	t2	idx1	10	0

Stats Meta

Version	Table	Count
1	t1	600
2	t3	999
3	t2	150

Stats Histogram

Version	Table	Column	NDV	NullCount
1	t2	c1	340	10
2	t2	c2	55	0
3	t2	idx1	10	0

Stats Buckets

Table	Column	Buckets ID	Lower	Upper	Count	Repeats
t2	c1	0	10	20	100	99
t2	c1	1	20	30	80	30
t2	c1	2	40	50	90	70

Estimation

- For $\sigma_{\theta_1 \wedge \theta_2}(r)$, the result should be

$$n_r * s_1 * s_2$$

- For $\sigma_{\theta_1 \vee \theta_2}(r)$, the result should be

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r})$$

- For $r \bowtie s$, the result should be

$$\frac{n_r * n_s}{V(A, s) * V(A, r)} * min(V(A, s), V(A, r))$$

- For $\mathcal{G}_{A,F}(r)$, the result should be $V(A, r)$

Interesting Order

For a classic optimizing framework like System R, every operator cares about some particular orders. We call it "interesting order". For different physical algorithm, the interesting orders are also different.

- For $\mathcal{G}_{A,F}$, its interesting order is $\{A\}$(For Streamed Aggregation) and $\emptyset$ (For Hash Aggregation)
- For Sort(r.A), its interesting order is $\{A\}$ and $\emptyset$
- For $s \bowtie_{\sigma_{s.A=r.A \wedge s.B=r.B}} r$, its interesting order is $\{s.A, s.B\}, \{r.A, r.B\}$(sort merge join) and $\emptyset$ (hash join)

Interesting Order

For a classic optimizing framework like System R, every operator cares about some particular orders. We call it "interesting order". For different physical algorithm, the interesting orders are also different.

- For $\mathcal{G}_{A,F}$, its interesting order is $\{A\}$(For Streamed Aggregation) and $\emptyset$ (For Hash Aggregation)
- For Sort(r.A), its interesting order is $\{A\}$ and $\emptyset$
- For $s \bowtie_{\sigma_{s.A=r.A \wedge s.B=r.B}} r$, its interesting order is $\{s.A, s.B\}, \{r.A, r.B\}$(sort merge join) and $\emptyset$ (hash join)

For the 3rd case, can $\{s.B, s.A\}, \{r.B, r.A\}$ be an interesting order?

Interesting Order

For a classic optimizing framework like System R, every operator cares about some particular orders. We call it "interesting order". For different physical algorithm, the interesting orders are also different.

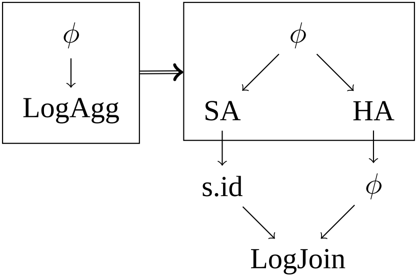
- For $\mathcal{G}_{A,F}$, its interesting order is $\{A\}$(For Streamed Aggregation) and $\emptyset$ (For Hash Aggregation)
- For Sort(r.A), its interesting order is $\{A\}$ and $\emptyset$
- For $s \bowtie_{\sigma_{s.A=r.A \wedge s.B=r.B}} r$, its interesting order is $\{s.A, s.B\}, \{r.A, r.B\}$(sort merge join) and $\emptyset$ (hash join)

For the 3rd case, can $\{s.B, s.A\}, \{r.B, r.A\}$ be an interesting order? So if we have n equal conditions, then we can get $n!$ different interesting orders! The solution is to determine all the possible interesting orders before CBO.

A running example

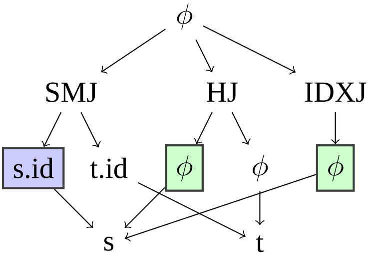
SELECT * FROM S JOIN T ON S.id = T.id AND S.c1 < 5 GROUP BY S.id

At first, the logical aggregation meets interesting order with ϕ:

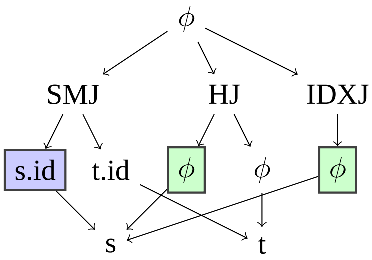


Then we consider stream aggregation with interesting order s.id:

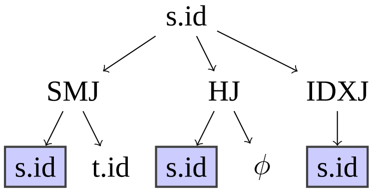
The enforced branch(We should enforced a sort operator upon Join):



The enforced branch(We should enforced a sort operator upon Join):



The inherited branch:



There’re some sub-problems that can be memorized.

There’re two ways to read s:

$$\begin{array}{c} \sigma_{(c1<5)} \\ \downarrow \\ \text{FullScan(s)} \end{array} \qquad \begin{array}{c} \text{Sort(id)} \\ \downarrow \\ \text{RangeScan(s}(-\infty, 5)\text{)} \end{array}$$

There're two ways to read s:

$$\sigma_{(c1<5)} \qquad\qquad \text{Sort(id)}$$
$$\downarrow \qquad\qquad\qquad \downarrow$$
$$\text{FullScan(s)} \qquad\qquad \text{RangeScan(s}(-\infty, 5))$$

What if there is a limit ? Say
SELECT * FROM s WHERE s.c1 < 5 ORDER BY id LIMIT 1

There're two ways to read s:

$$\begin{array}{c} \sigma_{(c1<5)} \\ \downarrow \\ \text{FullScan(s)} \end{array} \qquad \begin{array}{c} \text{Sort(id)} \\ \downarrow \\ \text{RangeScan(s}(-\infty, 5)\text{)} \end{array}$$

What if there is a limit ? Say
SELECT * FROM s WHERE s.c1 < 5 ORDER BY id LIMIT 1
We can still produce two plans:

$$\begin{array}{c} \text{Limit(1)} \\ \downarrow \\ \sigma_{(c1<5)} \\ \downarrow \\ \text{FullScan(s)} \end{array} \qquad \begin{array}{c} \text{TopN(id,1)} \\ \downarrow \\ \text{RangeScan(s}(-\infty, 5)\text{)} \end{array}$$

Is it really a full scan ?

There're two ways to read s:

$$\sigma_{(c1<5)} \quad\quad\quad\quad \text{Sort(id)}$$
$$\downarrow \quad\quad\quad\quad \downarrow$$
$$\text{FullScan(s)} \quad\quad\quad\quad \text{RangeScan(s}(-\infty, 5))$$

What if there is a limit ? Say
SELECT * FROM s WHERE s.c1 < 5 ORDER BY id LIMIT 1
We can still produce two plans:

$$\text{Limit(1)}$$
$$\downarrow$$
$$\sigma_{(c1<5)} \quad\quad\quad\quad \text{TopN(id,1)}$$
$$\downarrow \quad\quad\quad\quad \downarrow$$
$$\text{FullScan(s)} \quad\quad\quad\quad \text{RangeScan(s}(-\infty, 5))$$

Is it really a full scan ?
The scanning count expects to be $min(n(s), 1/f(\sigma_{(c1<5)}))$