Database dynamic real-time cardinality estimation

Background of the topic:

Cardinality estimation plays a key role in database systems. The execution of database queries usually involves the join and filter conditions of multiple tables. Cardinality estimation is used to predict the size of the result set generated by each step, thus helping the query optimizer to select the optimal execution plan. Inaccurate cardinality estimates can cause the Database Engine to choose an inefficient execution strategy, which can affect the overall performance of the query.

Selectivity is the percentage of rows that satisfy a particular predicate condition and can be calculated from a cardinality estimate:

$$Selectivity = \frac{Number\ of\ tuples\ after\ filtering}{Total\ number\ of\ tuples\ before\ filtering}$$

The selection rate affects the selection of a database execution plan, such as whether to perform an index scan, a full table scan, or a join algorithm. When the selection rate of the condition is high, it means that most of the data meets the condition, and the database may prefer a full table scan instead of an index. When the selection rate is low, the database prefers to use index scans to read only the rows that meet the conditions, reducing unnecessary I/Os. In the case of multiple table joins, cardinality estimation can affect the join order. For example, if a join condition produces a small result set, the optimizer may prioritize the join to reduce the overhead of subsequent operations.

Assume that there is a table `employees` whose structure is as follows:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    department_id INT,
    salary DECIMAL(10, 2),
    hire_date DATE
);
```

The data distribution in the table is as follows:

- The data in the `department_id` column is skewed. Most employees belong to `department_id = 1`, and the remaining few belong to other departments.
- The'salary' column is more evenly distributed, but most employees are paid less than 100000.

Query 1: High Selectivity Condition

```
SELECT * FROM employees WHERE department_id = 1;
```

Because the query condition `department_id = 1` satisfies most of the data rows, the database considers that the result set returned by the query is large (high selectivity). Therefore, **Full table scan** may be selected to read all rows. A typical execution plan is as follows:

```
Seq Scan on employees (cost=0.00..1000.00 rows=900 width=50)
  Filter: (department_id = 1)
```

In this plan, the database chooses to scan the entire table sequentially (Seq Scan) rather than using indexes because most of the estimated rows match the query criteria. The overhead of a full table scan may be lower in this case.

Query 2: Low Selectivity Condition

```
SELECT * FROM employees WHERE salary > 100000;
```

Assuming that the number of employees with `salary > 100000` is small (low selection rate), the database optimizer considers that the returned result set is small and may choose **Index Scan** to read only the rows that meet the conditions, rather than scan the entire table. A typical execution plan is as follows:

```
Index Scan using idx_salary on employees (cost=0.00..50.00 rows=10 width=50)
  Index Cond: (salary > 100000)
```

In this execution plan, instead of scanning the entire table sequentially, the database uses the index on the `salary` column (assuming the index `idx_salary` exists) to efficiently locate eligible rows.

These examples show that accurate cardinality estimation can calculate accurate selection rates and allow the database to select the optimal execution plan, thereby greatly improving query execution performance.

Cardinality Estimation Method

In a database, the primary goal of cardinality estimation is to predict the number of records returned by a query. These estimation methods typically rely on data distributions and statistics. Common cardinality estimation methods include:

1. **Histogram**: The histogram divides data values into different intervals (buckets) and records the frequency of data in each bucket. The database system uses this frequency information to estimate the number of rows of data that satisfy the predicate condition. For example, if a query requires `salary > 100000`, the optimizer can look at the buckets that contain the range and estimate the number of matching rows.
2. **Number of Distinct Values**: indicates the number of unique values in a record column (that is, the cardinality of the column). For equivalent queries (such as `WHERE department_id = 5`), the system can estimate the selection rate based on the number of unique values and the total number of records.
3. **Most Common Values**: frequency of recording specific values (especially values with extremely high or low frequency). When queries involve these frequent or rare values, the system can use these statistics for accurate cardinality estimates.
4. **Sampling**: Sampling is performed based on a subset of data to infer the cardinality of the entire table. Sampling methods are usually used to supplement cardinality estimation. When there is not enough statistics, sampling can provide a fast approximation method.

Difficulty in cardinality estimation in multi-column

1. **Statistics out of date**
    - In a real-time updated system, data is constantly inserted , deleted and modified , resulting in these statistics becoming less accurate, especially if the data changes frequently.
    - **Performance-accuracy trade-offs**: Maintaining high accuracy and low latency of statistics requires frequent running of cardinality estimation algorithms, but this

operation incurs additional system overhead and may affect the performance of other real-time operations (such as inserts, queries, updates, etc.). As a result, database systems often require a trade-off between performance and accuracy.

2. **Correlation between columns**:
    - Typically, the database assumes that different columns are **independent** , i.e. the distribution of `department_id` and `salary` is irrelevant. However, in actual data, there is a **correlation between columns** . For example, high-paid employees may be more likely to come from a particular department. If you ignore this correlation, you may underestimate or overestimate the cardinality of the query.
    - For example, when querying `WHERE department_id = 5 AND salary > 100000`, if employees with `department_id = 5` generally have lower salaries, the assumption of column independence can cause a significant skew in the cardinality estimate.
3. **Combination complexity of multi-column conditions**:
    - when the query contains more than one predicate (e.g. `WHERE department_id = 3 AND hire_date >'2022-01-01'`) The database system needs to integrate the distribution of multiple columns to estimate the number of records that meet the conditions. The complexity of this combination increases with the increase of the number of columns, which directly affects the accuracy of cardinality estimation.
    - Databases typically estimate multi-column cardinality using a method called **selectivity product model**, which multiplies the selectivity of each predicate to estimate the final result set size. However, if the columns are not independent of each other, this approach tends to produce significant errors.
4. **Skewness**:
    - Data skew is when some columns or values are extremely unevenly distributed. For example, the `department_id` column might record mostly `department_id = 1`, while other departments have very little data. This invalidates the database's cardinality estimation method based on the evenly distributed assumption, resulting in inefficient execution plan selection.
    - In the case of high data skew, the optimizer may not be able to accurately estimate the cardinality of low-frequency data values, and deviations are more likely to occur in multi-column conditions.

Problem Description:

You need to perform a real-time cardinality estimate for a table containing two columns of integers, and implement the specified interface based on the provided template project to support the interaction of the profiler. We assume that all tuples in the table are stored row-by-row in a contiguous space, and that each tuple has a unique index that determines its location on disk, numbered from 0. The profiler passes in the initial size of the dataset. Assuming that the table has `N` tuples, the table `T` can be expressed as:

$$T = \{t_0, t_1, t_2, \ldots, t_{N-1}\}, T_i = \{A_i, B_i\}$$

where `ti` is the tuple with index `i` in the table. The cardinality estimate returns the number of results in the table that match the filter criteria.

Assume that two constraints `A > 0` and `B = 100` are transferred. Only the union operation is considered. You need to find the tuple set that meets both `A > 0` and `B = 100`. Assume that:

$$S1 = \{t_i \in T \mid t_i.A > 0\}$$
$$S2 = \{t_i \in T \mid t_i.B = 100\}$$

The cardinality estimation result is the size of the intersection of `S1` and `S2`:

$$result = \mid S1 \cap S2 \mid$$

Data Description.

The structure of this table is as follows:

Column Name	Type
A	Int
B	Int

- Column `A` and column `B` are both integers.
- The data volume is very large. You need to estimate the number of elements that meet the filter criteria in the two integer columns `A` and `B` in real time under limited memory and time requirements.

Competitors need to implement the specified interface based on the provided template project to support the interaction of the evaluation program. We prepared a template of a CardinalityEstimation structure in CardinalityEstimation/include/CardinalityEstimation.h:

```
class CEEngine {
public:
    /**
     * Insert a tuple, indicating that the tuple is inserted and appended to the
end of the disk.
     * @param tuple Inserted tuple.
     */
    void insertTuple(const std::vector<int>& tuple);
    /**
     * Deletion function. Pass a tuple and tupleId, indicating that the tuple at
the tupleId position is deleted.
     * @param tuple Deleted tuple.
     * @param tupleId Location of the deleted tuple.
     */
    void deleteTuple(const std::vector<int>& tuple, int tupleId);
    /**
     * Query function, pass in expression, return estimated cardinality result.
     * @param quals expression.
     * @return return estimated cardinality result.
     */
    int query(const std::vector<CompareExpression>& quals);
    /**
     * Preprocessing function of the cardinality estimation algorithm. This
function is executed before each operation
     * is called.
     */
    void prepare();
    /**
     * The constructor function of cardinality estimation.
     * @param num Size of the initial data set.
     * @param dataExecuter Interfaces for datasets.
     */
    CEEngine(int num, DataExecuter *dataExecuter);
    ~CEEngine() = default;
```

```
private:
    DataExecuter *dataExecuter;
};
```

We assume that all tuples in the table are stored row by row in a contiguous space, and that each tuple has a unique index that determines its location on disk, numbered from 0. The assessment program transfers the initial data volume of the data set. The following types of services need to be processed:

- 1. Insert: Input a tuple `t`, indicating that the tuple is inserted to the end of the table `T`. No value is returned. After the insert operation, the table becomes:

$$T' = T \cup \{t\}。$$

- 2. Delete: Input a tuple `t` and an index `i`, the tuple whose index is `i` is deleted. No value is returned. After the delete operation, the table changes to:

$$T' = T \setminus \{t_i\}$$

- 3. Query: A maximum of two constraint conditions (one for each column) are input. (The ratio of single-column query to multi-column query is about 1:1.) Each condition can be a predicate greater than (`>`) or equal to (`=`). Only the **conjunctive** operation is considered.

  Each constraint condition is represented by the CompareExpression structure.

```
// CompareOp is an enumerated value representing the predicate condition
type
enum CompareOp {EQUAL = 0, GREATER = 1};

typedef struct CompareExpression {
    int columnIdx;  // Corresponding to columnIdx in the table, starting
from 0.
    CompareOp compareOp;
    int value;      //Comparison constant representing predicate
condition
} CompareExpression;
```

  For struct {columnIdx: 0, compareOp: 0, value: 12321}, the expression is "A=12321".

Assessment Process:

The overall operation process is summarized as follows:

1. The contestant downloads the code template provided by the competition group and implements the algorithm. It must be implemented in C/C++ language.
2. Submit the code. The evaluation program pulls the corresponding submitted code for evaluation.
3. The evaluation program updates the contestant's score.

Note: Third-party libraries are not recommended through links and multi-threading is prohibited. If third-party libraries are required, copy the source code to the project. If links are required, debug them by yourself. We provide a demo program for local debugging. See the README document of the code framework for details.

Assessment logic:

Because the algorithm needs to interact with the evaluation framework, the final resource usage is equal to the resource used by the algorithm submitted by the contestant plus the resource used by the evaluation framework.

- The initial data volume of each test case cannot exceed **50,000,000**, including **20,000,000** insert, delete, and query operations. The data value ranges from **1 to 20,000,000**.
- Limit the memory usage to 10 MB, (Since the evaluation framework itself consumes about 6MB, you can use up to about **4 MB** of memory.)
- The execution time of each test case is limited to 12s. (The execution time of the evaluation framework is 2s, so you can consume up to **10s**.)
- Provides a disk read interface to obtain tuple content in a specified range. Players can use this interface to obtain real data distributed on disks. It takes about **1s to read 10,000,000 pieces of data**.
- Ranked according to the error in ascending order (the more accurate the estimate is, the higher the ranking is). Assume that n is the number of queries. The error calculation formula is as follows:

$$score = \frac{\sum_{i=0}^{n} |\log(\frac{EstimatedValue_i + 1}{ActualValue_i + 1})|}{n}$$

Competition Description

The competition is divided into two stages:

- **Task 1: October 14, 2024 to November 27, 2024**. Using the Task 1 dataset for profiling during Task 1.
- **Task 2: November 27, 2024 to November 30, 2024.** Using the Task 2 dataset for profiling during Task 2.

Competition officials will compete for the final prizes and award the prizes according to the team's code, historical results and judges.

Scoring criteria for the finals: We take the average ranking of the contestants in Task 1 and Task 2 as the final ranking. When the average rankings are the same, we sort the contestants in ascending order according to the average error of Task 1 and Task 2. The higher the ranking, the lower the error, the better the final result.

In order to ensure the fairness and justice of the competition, the organizer has the right to disqualify the contestants if any of the following conditions is meet:

- The trustlist is hardcoded in the code as a part of the prediction result.
- Core code plagiarism exists.