Orthogonal Optimization Of Subqueries And Aggregationn

February 14, 2025

1 Introduction

In this paper, we present subquery and aggregation techniques implemented in Microsoft SQL Server.

1.1 Standard subquery execution strategies

Before describing subquery strategies in detail, it is important to clarify the two forms of aggregation in SQL, whose behavior diverges on an empty input.

"Vector" aggregation specifies grouping columns as well as aggregates to compute.

```
1 select o_orderdate, sum(o_totalprice)
2 from orders
3 group by o_orderdate
```

And there are querys that *always returns exactly one row*:

```
1 select sum(o_totalprice) from orders
```

In algebraic expressions we denote vector aggregate as $\mathcal{G}_{A,F}$, where A are the grouping columns and F are the aggregates to compute; and denote scalar aggregate as $\mathcal{G}^1_F$

We review standard subquery execution strategies using the following SQL query, which finds customers who have ordered more than $1,000,000.

```
-- Q1
select c_custkey
from customer
where 100000 <
      (select sum(o_totalprice)
       from orders
       where o_custkey = c_custkey)
```

Outerjoin, then aggregate:

```
select c_custkey
from customer left outer join
     orders on o_custkey = c_custkey
group by c_custkey
having 1000000 < sum(o_totalprice)
```

Aggregate, then join:

```
select c_custkey
from customer,
     (select o_custkey from orders
      group by c_custkey
      having 1000000 < sum(o_totalprice))
     as aggresult
where o_custkey = c_custkey
```

1.2 Our technique: Use primitive, orthogonal pieces

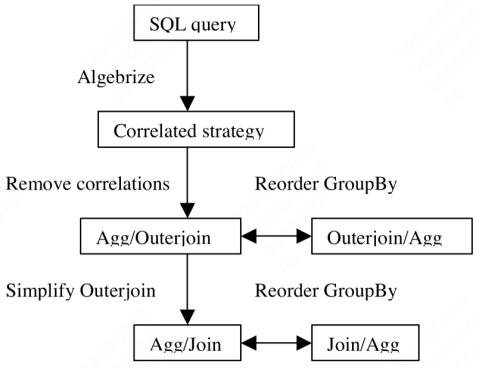

Figure 1: Primitives connecting different execution strategies

By implementing all these orthogonal techniques, the query processor should then pro- duce the same efficient execution plan for the various equiv- alent SQL formulations we have listed above, achieving a degree of syntax-independence.

- **Algebrize into initial operator tree**
- **Remove correlations**
- **Simplify outerjoin**
- **Reorder GroupBy**

1.3 A useful tool: Represent parameterized execution algebraically

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. Formally,

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

where $\otimes$ is either cross product, left outerjoin, left semijoin, or left antijoin.

The most primitive form is $\mathcal{A}^{\times}$, and cross product is assumed if no join variant is specified.

All operators used in this paper are bag-oriented, and we assume no automatic removal of duplicates. In particular, the union operator above is `UNION ALL`. Duplicates are removed explicitly using `DISTINCT`.

Now Q1 is like

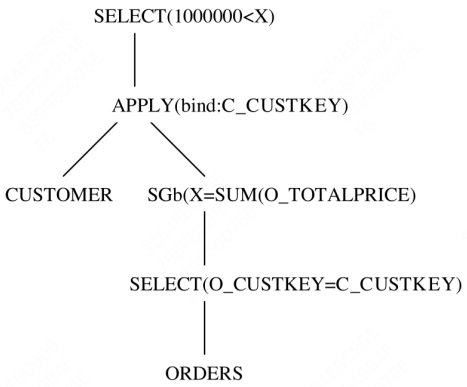

Figure 2: Subquery execution using Apply

Apply works on expressions that take scalar (or row-valued) parameters. A second useful construct is **SegmentApply**, which deals with expressions using *table*-valued parameters. It takes a relation input R, a parameterized expression $E(S)$, and a set of segmenting columns A from R. It creates segments of R using columns A, much like GroupBy, and for each such segment S it executes $E(S)$. Formally,

$$R\,\mathcal{SA}_A\,E = \bigcup_a (\{a\} \times E(\sigma_{A=a}R))$$

where a takes all values in the domain of A.

2 Representing and normalizing subqueries

2.1 Direct algebraic representation with mutual recursion

2.2 Algebraic representation with Apply

2.3 Removal of Apply

$$R\mathcal{A}^{\otimes}E = R \otimes_{\text{true}} E \tag{1}$$

if no parameters in E resolved from R

$$R\mathcal{A}^{\otimes}(\sigma_p E) = R \otimes_p E \tag{2}$$

if no parameters in E resolved from R

$$R\mathcal{A}^{\times}(\sigma_p E) = \sigma_p(R\mathcal{A}^{\times}E) \tag{3}$$

$$R\mathcal{A}^{\times}(\pi_v E) = \pi_{v \cup \text{columns}(R)}(R\mathcal{A}^{\times}E) \tag{4}$$

$$R\mathcal{A}^{\times}(E_1 \cup E_2) = (R\mathcal{A}^{\times}E_1) \cup (R\mathcal{A}^{\times}E_2) \tag{5}$$

$$R\mathcal{A}^{\times}(E_1 - E_2) = (R\mathcal{A}^{\times}E_1) - (R\mathcal{A}^{\times}E_2) \tag{6}$$

$$R\mathcal{A}^{\times}(E_1 \times E_2) = (R\mathcal{A}^{\times}E_1) \bowtie_{R.key} (R\mathcal{A}^{\times}E_2) \tag{7}$$

$$R\mathcal{A}^{\times}(\mathcal{G}_{A,F}E) = \mathcal{G}_{A \cup \text{columns}(R),F}(R\mathcal{A}^{\times}E) \tag{8}$$

$$R\mathcal{A}^{\times}(\mathcal{G}^1_F E) = \mathcal{G}_{\text{columns}(R),F'}(R\mathcal{A}^{LOJ}E) \tag{9}$$

In (9), F' contains aggregates in F expressed over a single-column - for example, if F is `COUNT(*)`, then F' is `COUNT(C)` for some not-nullable column C from E. It is valid for all aggregates s.t. $agg(\emptyset) = agg(\{null\})$, which is true for SQL aggregates.

LOJ is left outerjoin

The proof is in [GL00]

2.4 All SQL subqueries

For boolean-valued subqueries, i.e., `EXISTS`, `NOT EXISTS`, `IN` subquery, and quantified comparisons, the subquery can be rewritten as a scalar `COUNT` aggregate. From the utilization context of the aggregate result, either equal to zero or greater than zero, it is possible for the aggregate operator to stop requesting rows as soon as one has bee found, since additional rows do not affect the result of the comparison.

A common case that is further optimized is when a relational select has an existential subquery as its only predicate. In this case, the complete select operator is turned into Apply-semijoin for *exists*, or Apply-antisemijoin for *not exists*. Such Apply is then converted into a non-correlated expression, if possible, using Identify (2).

There are two scenarios where normalization into standard relational algebra operators is hindered in a fundamental way. We call those **exception subqueries** and they require scalar-specific features. Consider the following query.

```
1  -- Q2
2  select c_name,
3         (select o_orderkey from orders
4          where o_custkey = c_custkey)
5  from customer
```

For every customer, output the customer name, and the result of a subquery that retrieves an oderkey. There are three cases: If exactly one row is returned from the subquery, then such value is used in the scalar expression; if no rows are returned, then null is used; finally, if more than one row is returned, then a run-time error is generated. We call such operator `Max1row`.

2.5 Subquery classes

2.5.1 Class 1. Subqueries that can be removed with no additional common subexpressions

In general, removing Apply requires introduction of additional common subexpressions. E.g., Identity (5) introduces two copies of R.

The common case of subqueries that are formed by a simple select/project/join/aggregate block are easy to handle.

2.5.2 Class 2. Subqueries that are removed by introducing common subexpressions

It is hard to formulate a short, meaningful query that fits in this class, using the TPCH schema.

2.5.3 Class 3. Exception subqueries

3 Comprehensive optimization of aggregation

3.1 Reordering GroupBy

We will denote GroupBy as $\mathcal{G}_{A,F}$, where A is the set of grouping columns and F are the aggregate functions.

We can move a filter around a GroupBy iff all the columns used in the filter are functionally determined by the grouping columns in the input relation.

A GroupBy can be pushed below a join if the grouping columns, the aggregate calculations and the join predicate each satisfy certain conditions. Suppose we have a GroupBy above a join of two relations, i.e., $\mathcal{G}_{A,F}(S \bowtie_p R)$, and we want to push the GroupBy below the join so that the relation R is aggregated before it is joined, i.e.,

$$S \bowtie_p (\mathcal{G}_{A \cup \text{columns}(p) - \text{columns}(S), F} R)$$

Assuming S and R have no common columns, and p has columns from S, so we need to filter them out. This is feasible iff the following conditions are met:

1. If a column used in the join predicate p is defined by the relation R then it is part of the grouping columns
2. The key of the relation R typo in the paper is part of the grouping columns
3. The aggregate expressions only use columns defined by the relation R

Pulling a GroupBy above a join is a lot easier. All that is required is that the relation being joined has a key and that the join predicate does not use the results of the aggregate functions.

$$S \bowtie_p (\mathcal{G}_{A,F} R) = \mathcal{G}_{A \cup \text{columns}(S), F}(S \bowtie_p R)$$

One can think of semijoins and antisemijoins as filters since they include or exclude rows of a relation based on the column values. The conditions necessary to reorder these operators around

3.2 Moving GroupBy around an outerjoin

Removing correlations for scalar valued subqueries results in an outerjoin followed by a GroupBy.

4 Problems

5 References

References

[GL00] Cesar Galindo-Legaria. Parameterized queries and nesting equivalencies. Technical Report MSR-TR-2000-31, April 2000.