Orthogonal Optimization Of Subqueries And Aggregation

February 13, 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.

```sql
select o_orderdate, sum(o_totalprice)
from orders
group by o_orderdate
```

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

```sql
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.

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

Outerjoin, then aggregate:

```sql
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:

```sql
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

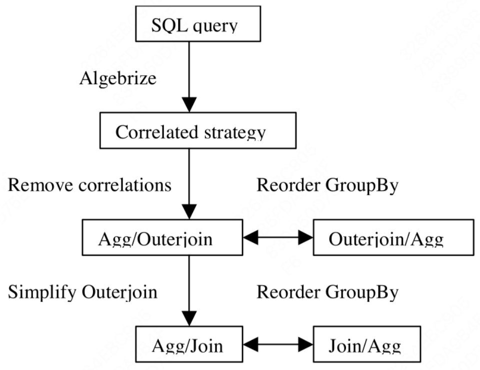

Figure 1: Primitives connecting different execution strategies

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

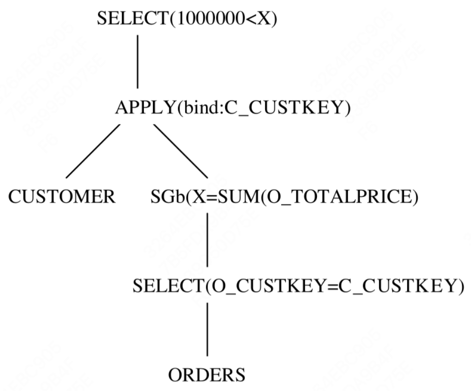

Figure 2: Subquery execution using Apply

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

$$R\mathcal{A}^{\otimes}E = R \otimes_{\text{true}} E \quad (1)$$

if no parameters in E resolved from R

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

if no parameters in E resolved from R

3 Problems

4 References

References