Advanced Query Optimization

Antonio Badia
University of Louisville, USA

INTRODUCTION

Query optimization has been an active area of research ever since the first relational systems were implemented. In the last few years, research in the area has experienced renewed impulse, thanks to new developments like data warehousing. In this article, we overview some of the recent advances made in complex query optimization. This article assumes knowledge of SQL and relational algebra, as well as the basics of query processing; in particular, the user is assumed to understand cost optimization of basic SQL blocks (Select-Project-Join queries). After explaining the basic unnesting approach to provide some background, we overview three complementary techniques: source and algebraic transformations (in particular, moving outerjoins and pushing down aggregates), query rewrite (materialized views), new indexing techniques (bitmap and join indices), and different methods to build the answer (online aggregation and sampling). Throughout this article, we will use subquery to refer to a nested SQL query and outer query to refer to an SQL query that contains a nested query. The TPCH benchmark database (TPC, n.d.) is used as a source of examples. This database includes (in ascending size order) tables Nation, Customer (with a foreign key for Nation), Order (with a foreign key for Customer), and Lineitem (with a foreign key for Order). All attributes from a table are prefixed with the initial of the table’s name (“c_” for Customer, and so on).

BACKGROUND

One of the most powerful features of SQL is its ability to express complex conditions using nested queries, which can be non-correlated or correlated. Traditional execution of such queries by the tuple iteration paradigm is known to be inefficient, especially for correlated subqueries. The seminal idea to improve this evaluation method was developed by Won Kim (1982) who showed that it is possible to transform many queries with nested subqueries in equivalent queries without subqueries. Kim divided subqueries into four different classes: non-correlated, aggregated subqueries (type-A); non-correlated, not aggregated subqueries (type-N); correlated, not aggregated subqueries (type-J); and correlated, aggregated subqueries (type-JA). Nothing can be done for type-A queries, at least in a traditional relational framework. Type-N queries correspond to those using IN, NOT IN, EXISTS, NOT EXISTS SQL predicates. Kim’s observation was that queries that use IN can be rewritten transforming the IN predicate into a join (however, what is truly needed is a semijoin). Type-J queries are treated essentially as type-N.

Type-JA is the most interesting type. For one, it is a very common query in SQL. Moreover, other queries can be written as type-JA. For instance, a query with EXISTS can be rewritten as follows: a condition like EXISTS (SELECT attr...) (where attr is an attribute) is transformed into 0 > (SELECT COUNT(*) (the change to “*” is needed to deal with null values). To deal with type-JA queries, we first transform the subquery into a query with aggregation; the result is left in a temporary table, which is then joined with the main query. For instance, the query:

Select c_custkey
From Customer
Where c_acctbal > 10,000 and c_acctbal > (Select sum(o_totalprice) From Order
Where o_custkey = c_custkey))

is executed as:

Create Temp as
Select o_custkey, sum(o_totalprice) as sumprice
From Order
Group by o_custkey

Select c_custkey
From Customer, Temp
Where o_custkey = c_custkey and c_acctbal > 10,000
and c_acctbal > sumprice

However, this approach fails on several counts: first, non-matching customers (customers without Order) are lost in the rewriting (they will fail to qualify in the final join) although they would have made it into the original query (if their account balances were appropriate); this is sometimes called the zero-count bug. Second, the approach is incorrect when the correlation uses a predicate other than equality. To solve both problems at once, several authors suggested a new strategy: first, outerjoin the relations involved in the correlation (the
outerjoin will keep values with no match), then, compute the aggregation. Thus, the example above would be computed as follows:

```
Create Table Temp(c_key, sumprice) as
  (Select c_custkey, sum(o_totalprice)
   From Customer Left Outer Join Order on o_custkey = c_custkey
   Group by c_custkey)
```

```
Select c_custkey
From Customer, Temp
Where c_custkey = c_key and c_acctbal > 10,000 and c_acctbal > sumprice
```

This approach still has two major drawbacks in terms of efficiency. First of all, we note that we are only interested in certain customers (those with an account balance over a given amount); however, all customers are considered in the outerjoin. The Magic Sets approach (Seshadri et al., 1996) takes care of this problem by computing first the values that we are interested in. For instance, in the example above, we proceed as follows:

```
Create table C_mag as
  (Select c_custkey, c_acctbal From Customer
   Where c_acctbal > 10,000);
Create table Magic as
  (Select distinct c_custkey as key From C_mag);
/* this is the Magic Set */
Create table Correl (custkey, sumprice) as
  (Select c_custkey, sum(o_totalprice)
   From Customer outer join Magic on key = c_custkey
   Group by c_custkey)
Select c_custkey
from C_mag, Correl
where c_custkey = custkey and c_acctbal > sumprice
```

### QUERY OPTIMIZATION

#### Query Transformations

Although the approaches discussed are an improvement over the naïve approach, they introduce another issue that must be taken care of: the introduction of outerjoins in the query plan is problematic, since outerjoins, unlike joins, do not commute among themselves and with other operators like (regular) joins and selections. This is a problem because query optimizers work, in large part, by deciding in which order to carry out operations in a query, using the fact that traditional relational algebra operators can commute, therefore, can be executed in a variety of Order. To deal with this issue, Galindo-Legaria and Rosenthal (1997) give conditions under which an outerjoin can be optimized. This work is expanded in Rao et al. Intuitively, this line of work is based on two ideas: sometimes, outerjoins can be substituted by regular joins (for instance, if a selection is to be applied to the result of the outerjoin and the selection mentions any of the padded attributes, then all padded tuples that the outerjoin adds over a regular join will be eliminated anyway, since they contain null values and null values do not pass any condition, except the IS NULL predicate); and sometimes, outerjoins can be moved around (the generalized outerjoin proposed in the work above keeps some extra attributes so that interactions with joins are neutralized).

Finally, one last note on unnesting to point out that most approaches do not deal properly with operators involving negation or, equivalently, universal quantification, like operators involving the ALL comparison. It has been suggested that operators using ALL could be rewritten as antijoins; a query like:

```
Select c_custkey
From Customer
Where c_acctbal > ALL (Select o_totalprice From Order where c_custkey = o_custkey)
```

can be decided by outerjoining Customer and Order on condition $(c_{custkey} = o_{custkey} \text{ AND } c_{acctbal} <= o_{totalprice})$ (since a tuple in Customer would be present in the outerjoin only if the $c_{acctbal}$ was never less than or equal to a total price, that is, if the $c_{acctbal}$ was greater than all total prices); unfortunately, this reasoning does not hold when there are nulls present in either attributes. A different approach dealing with these operators is that of Aiken and Bahlen (2003). This work introduces a multidimensional join operator (MD), in which two relations can be joined on several conditions. The MD operator can be annotated with aggregate operations, each one computed in the result of a different join condition. Then, the above query would be computed by an MD-join between Customer and Order, where (a) a grouping by $c_{custkey}$ and a count(*) are computed over the join via condition $c_{custkey} = o_{custkey}$ and (b) a grouping by $c_{custkey}$ and a count(*) are computed over the join via condition $c_{custkey} = o_{custkey}$ and $c_{acctbal} > o_{totalprice}$. The tuples where the two counts are the same are then picked up by a selection. Intuitively, we count all tuples for a given value that could possibly fulfill the ALL operator and all tuples that actually fulfill it; if both numbers are the same, then