International Conference on Internet Studies
August 16-17,2014, Singapore

CORRELATED TABLES FOR NESTED QUERY EVALUATION

Li-Yan Yuan

Department of Computing Science
University of Alberta
Edmonton Canada
`yuan@cs.ualberta.ca`

ABSTRACT

In this paper, we first define the correlated table of an SQL query without nested subqueries which extends the result set of an SQL query to a set of result sets, one for each distinct binding tuple. We then present an algorithm to convert any SQL query with nested subqueries to a correlated query without nested subqueries. The proposed algorithm provides a simple but very effective method unnest nested queries and thus can be used to efficiently evaluate nested queries in distributed database systems.
Keyword: query evaluation, distributed databases, nested queries

1 Introduction

We have developed a big database management system , named `RubatoDB`, for both OLTP and OLAP applications that uses a collection of commodity servers and support both ACID and BASE properties. The extensive experimental tests of RubatoDB shows that it outperforms several major database products and no SQL systems [1].

However, RubatoDB faces the challenges of evaluating nested queries for OLAP applications, and consequently, we propose and implement a new framework for nested query evaluation. In this paper, we specify the new framework and demonstrate its advantages over the existing technologies for nested query evaluation.

Traditionally, two approaches have been used to evaluate nested queries: the decorrelation (unnesting) strategy [2, 3] and the nested iteration with optimization [4, 5, 6].

A naive iteration plan for nested queries can be very inefficient as the nested subquery is evaluated for every distinct binding of the correlation attributes in the outer query. De-correlation techniques have been extensively studied and applied to enable traditional optimizers generate more efficient set oriented plans for nested queries. However, de-correlation is not always applicable, and even if applicable may not be the best choice in all situations since decorrelation carries a materialization overhead [6].

The nested iteration strategy has been greatly improved by two simple but very effective technologies, that is, (1) caching the results of the inner query [5] and (2) sorting the result tuples of the outer block [4]. As argued by Guravannavar, Ramanujam and Sudarshan that it is these two techniques that make the nested iteration a valid choice [6].

The query optimization algorithm presented in [6] is used to choose either the de-correlation or the nested iteration algorithms for efficient evaluation of nested queries.

Evaluation of nested queries in distributed (grid) database systems, like RubatoDB, and many other NoSQL systems, represents a new challenge in that the aforementioned two strategies cannot be used directly, mainly because the nested iteration suffers a great deal if either one or both the outer-inner blocks are distributed over different grid nodes. An efficient evaluation algorithm based on the de-correlation has been reported [7], but it does not work if the de-correlation is not available.

In this paper, we first define the `correlated table` of an SQL query without nested subqueries which extends the result set of an SQL query to a set of result sets, one for each distinct binding tuple. Assume Q is a query that (1) has no nested subqueries, and (2) contains a set of correlated (free) variables simulating the correlated variables of the outer query block. Given a set of distinct binding tuple values T_{in} for the correlated variable, the correlated table is defined as a set of result set, one for each binding tuple $tinT_{in}$.

The correlated result set is specified based on the following simple but very effective ideas.

- The correlated table utilizes the two key technologies in the optimized nested iteration: Sorting and Caching [5, 4].
- The correlated table can be constructed using any query evaluation algorithms for distributed database systems, as the underline query has no nested subqueries.

We then present an algorithm to convert any SQL query Q into a query Q_{ct} without nested subqueries using the correlated tables, and show that Q and Q_{ct} are equivalent in that both generated the same result set.

The proposed framework enjoys the following advantages over the existing technologies for nested query evaluation.

1. Similar to the unnesting technique, the correlated queries can be evaluated using existing optimization techniques developed for distributed databases.
2. Unlike the unnesting technology, the proposed proposed algorithm can convert any query into a query without nested subqueries.

3. The conversion is straightforward and is universally applicable, regardless different types of nested queries.

4. The correlated table utilizes the key optimization techniques of the nested iteration.

This paper reports a preliminary result and we are currently conduct experimental tests on RubatoDB.

The paper is organized as follows. Section 2 reviews the existing nested query evaluation algorithms. Section 3 defines a the correlated table and the algorithm used to convert any nested queries into the correlated queries without nested subqueries. The query evaluation algorithm is discussed in Section 4, and the conclusion is offered in Section 5.

2 Related Works

The nested query evaluation uses two different approaches, that is, the de-correlation (unnesting) strategy [2, 3] and the nested iteration with optimization [4, 5, 6].

De-correlation techniques have been extensively studied, but it is generally agreed that it is very difficult, if not impossible, to unnest queries with multiple and/or nested blocks [8, 6].

The nested iteration strategy has been greatly improved by two simple but very effective technologies, that is, (1) caching the results of the inner query [5] and (2) sorting the result tuples of the outer block [4]. As argued by Guravannavar, Ramanujam and Sudarshan that it is these two techniques that make the nested iteration a valid choice [6].

The query optimization algorithm presented in [6] is used to choose either the de-correlation or the nested iteration algorithms for efficient evaluation of nested queries.

Example 2.1 *Consider the query $Q_{2.1}$ below [6].*

```
SELECT  o_orderkey
FROM    ORDERS
WHERE   o_orderdate NOT IN (  SELECT  l_shipdate
                              FROM    LINEITEM
                              WHERE   l_orderkey = o_orderkey );
```

Suppose the LINEITEM table is stored sorted on l_orderkey column and the plan for the outer query-block guarantees to produce the bindings for the correlation variable, o_orderkey, in sorted order. The clustered table scan can stop as soon as a value greater than the value of the correlation variable is found and restart from this point on the next invocation, thus retaining state.

We believe that the restartable segment scan is a very efficient technique for general query evaluation, not necessarily for nested queries. Another example in [6] demonstrates that the restartable segment scan may not as efficient as the de-correlation if the number of rows to be scanned is small. However, this is due to inappropriate uses of the technique, not the problem associated with the nested iteration with optimization.

3 Correlated tables

In this section, we first specify the correlated table which extends the result set into a set of results, one for each distinct binding value. First a motivation example below.
Example 3.1 *Consider the query $Q_{2.1}$ in Example 2.1 again. The restartable segment scan is used to evaluate the inner query block with a given set of* `o_orderkey` *efficiently. That is, the evaluation of the inner block can be described logically as the evaluation of the query, denoted as Q_1 below:*

```
SELECT l_shipdate
FROM   LINEITEM
WHERE  l_orderkey = o_orderkey;
```

with a set of `o_orderkey` *values. The output of the evaluation is then a set of sets of* `l_shipdate` *values. Let T_1 denotes a set of* `o_orderkey`*, then, the evaluation of this query block can be described as*

$$cr(\{\texttt{l\_orderkey}\}_{T_1}, Q_1) \tag{1}$$

which returns a result set of two columns [1] *$\{o\_orderkey, \{l\_orderkey\}\}$, one for each distinct value in T_1. If T_1 is empty, it will then return a null value.*

The outer block of $Q_{2.1}$ is then simplified as Q_2 below.

```
SELECT o_orderkey
FROM   ORDERS
```

Then the evaluation of the Q_2 is then defined as $cr(\emptyset, Q_2)$ which returns a result set of `o_orderkey` *as usual. This is simply because the Q_2 has no correlated variables. For simplicity we use $cr(Q_2)$ to denote $cr(\emptyset, Q_2)$.*

The nested query $Q_{2.1}$ can then be evaluated by nested-join the outer block with the inner block, that is, by evaluating the modified (de-correlated) query $Q_{cr2.1}$ below.

SELECT o_orderkey
FROM ORDERS, $cr(\{o\_orderkey\}_{cs(Q_2)}, Q_1)$ as r_{cr}
WHERE o_orderkey $=_{set} r_{cr}$

where $=_{set}$ denotes the set-equation, that is, `o_orderkey` *value satisfies the condition if and only if there exists a tuple $< t, s > \in r_{cr}$ such that $o\_orderkey = t$ and $o\_orderkey \in s$.*

The correlation table is specified similar to the temporary table used in the de-correlation operation but with the following three distinct features.

1. it is constructed regardless the various types of nested queries,

2. it can be applied to any type of nested queries, including queries with nested store procedures.

[1]It consists of the two group of columns in general.

3. it needs not consider the query optimization strategy as the optimization will be considered by the evaluation of the correlation table. This is especially important if the tables of the block are distributed over different grid nodes.

The idea of the correlation table is based on the caching and sorting techniques proposed in [5, 4], as it is evaluated by taking in a set of distinct values (to be sorted if needed) and store each corresponding result set as its 2nd part in the correlation table. However, it can also be evaluated using other optimization techniques. Now we are in the position to formally defined the basic operators for nested query evaluation.

Let Q be an SQL query. A variable in Q is a symbol representing either a column of a base table, called a *column variable*, Q or a value to be specified, called a *free variable*. For example, `l_orderkey` and `o_orderkey` in Q_1 of Example 3.1 are a column variable and a free variable respectively. Q is called a `quantified query` if it has no free variable, and a `free query` if otherwise. All SQL queries to be evaluated are quantified queries while a subquery in a nested block with correlation variables is a free query.

Assume $v_1, \ldots, v_n$, $n \geq 0$, are the set of all the free variables in Q. A `binding tuple` is a tuple t of values, one for each free variable in Q. Given a binding tuple t, the evaluation of Q will return a set of tuples, $RSET_t(Q)$, called the `result set` with binding T.

Definition 3.1 *Let Q be a query with the set V of free variables, and T be a set of distinct binding tuples for V in Q. Then the correlation table, denoted as $cr(V_T, Q)$, is defined as*

$$cr(V_T, Q) = \{\langle t, RSET_t Q\rangle | t \in T\} \tag{2}$$

The evaluation of the correlation table represents a working horse of nested query evaluation, and it can be implemented using various optimization techniques, such as key search, index search, cost-based optimization, and restart segment scan, etc. Since the correlation table has no nested sub-queries, any optimization algorithms for distributed database systems can be readily extended to evaluate correlation tables.

Let Q be a quantified SQL query (i.e., a query without free variables). Then the result set of Q can be specified by $cr(null, Q)$, or $cr(Q)$ for short.

Further, the correlation table can be easily used to unnest any subqueries without semantic problems, as demonstrated below.

Example 3.2 *Consider the following query $Q_{3.2}$ in the well-known count bug below:*

```
SELECT pnum
FROM   parts
WHERE  qoh = ( SELECT  count(shipdate)
               FROM    supply
               WHERE   supply.pnum = parts.pnum AND
                       shipdate < to_date('1980-01-01')
             );
```

Now we demonstrate how the correlation table avoids such trouble. The following unnested query fails to return a correct result set if the table contains duplicated values on column pnum [SIGMOD87], and thus a more sophisticated unnest query is needed.

```
SELECT p1.pnum
FROM   parts p1,
       (SELECT p2.pnum s.pnum,count(s.shipdate) ct
        FROM   parts p2 left join supply s on p2.pnum=s.pnum
        WHERE  s.shipdate < to_date('1980-01-01')
        GROUP BY p2.pnum
       ) s2
WHERE  p1.qoh = s2.ct AND
       p1.pnum = s2.spnum;
```

Let Q_3 and Q_4 represent the two queries below.

```
SELECT DISTINCT pnum
FROM   parts;
```

and

```
SELECT count(shipdate)
FROM   supply
WHERE  supply.pnum = p_pnum AND
       shipdate < to_date('1980-01-01')
```

where `p_pnum` *is a free variable in Q_4. Using the correlation table, the unnesting can be easily accomplished, as shown below.*

SELECT p1.pnum
FROM parts p1, $cr(\{p\_pnum\}_{cr(Q_3)}, Q_4)$ as $r_{3.2}$
WHERE $p1.qoh =_{set} r_{3.2}$

It is not difficult to see how the correlation table resolves the count bug:

> For each pnum as a binding $p \in cr(Q_3)$, if p has no `shipdate` in `supply` that satisfies the where clause of Q_4 the corresponding `count(shipdate)` is 0, that is, $< p, \{0\} > \in r_{3.2}$ which effectively eliminate the count bug.

Now, we define the set operator used in our algorithm to be presented later. An SQL set comparison is a comparison operator of the form

$$\exists, < any, < all, =, > any, > all, =< any, =< all, >= any, >= all$$

Definition 3.2 *Let t be a binding tuple, R be a subquery, cr be the result set of the correlation table, and op be a set comparison. Then for any value v corresponding to the binging tuple t,*

$$v_t \quad op_{set}\ r \quad \text{if and only if} \quad \langle t, s\rangle \in r \wedge \quad v\ op \quad s \tag{3}$$

v is absent if op is $\exists$.

As a matter of fact, it is not difficult to see that $v\ op\ R$ represents the same condition of $v_t\ op_{set}\ r$. Consequently, the new set comparison op_{set} can be readily used to transform any set comparison with the newly specified correlation table. The operator $=_{set}$, given Examples 3.1 and 3.2, is a sample of such a set operator.

Using these two definitions, we are ready to convert any SQL queries with nested subquery to the one without nested subquery.

The first such an example is given Example 3.2, and the example below shows how to convert a query with three blocks, two of which are nested.

Example 3.3 *Consider a three-block nested query given in [8]*

$Q_{3.3}$
```
SELECT  R.a
FROM    R
WHERE   R.b=any {  SELECT  COUNT(S.*)
                   FROM    S
                   WHERE   R.c=S.c AND
                           S.d =any {  SELECT  COUNT(T.*)
                                       FROM    T
                                       WHERE   S.e = T.e
                                               R.f = T.f
```

First, consider the outer block, and let Q_5 be the query below:

Q_5
```
SELECT  R.c
FROM    R
```

The correlation table of Q_5 is $cr(Q_5)$ as it has no free variables. One may also see that each tuple in $cr(Q_5)$ provides two binding values, i.e., $R.c$ and $R.f$ correlated with the nested two blocks in the query.

Now consider the middle block. Note that the middle block has not just one free variable `R.c` *provided by the outer block Q_5, but also provide a binding value $S.e$ to be consumed by the inner block. For convenience we assume that all the binding values used by the nested block will be accessible at the correlation table. Hence, the middle block can be represented by the following query Q_6.*

Q_6
```
SELECT  COUNT(S.*)
FROM    S
WHERE   R.c = S.c
```

where `R.c` *is a free variable, and* `S.e` *representing a binding value to be consumed by the inner block. Therefore, it can be described by $cr(\{R.c\}_{cr(R_5)}, R_6)$.*

The inner block can be represented by the following query Q_7.

Q_7
```
SELECT  COUNT(T.*)
FROM    T
WHERE   S.e = T.e
        R.f = T.f
```

As all the binding tuples are generated by the query of the join of all previous (correlation) tables, consequently, we will omit the notation from now on.

Then the query $Q_{3.3}$ can be represented below.

```
Q_cr3.3  SELECT  R.a
         FROM    R, cr({R.c}, Q_6) as r_6, cr({R.f, S.e}, R_7) as r_7
         WHERE   R.b =_any r_6 AND r_6.d = any r_7.
```

Note that $r_6.d$ represents $S.d$ within r_6.

Now we present the algorithm used to represent any SQL query with nested subqueries in the form of the aforementioned correlation tables and set comparisons.

Algorithm 3.1 *Let Q be an SQL query. We assume that all the where clauses of subqueries are in conjunctive normal form. Further for simplicity, we do not consider the having clause in this paper, though the results can easily be extended to queries with the having clause.*

A query Q_{cr} is the query using the correlation tables constructed from Q by the following two modifications.

- **Step 1**. *For each block with the nested subquery Q_i, let V_i be the list of free variables in Q_i. Further, let Q'_i is constructed from Q_i by removing all sub-formulas (i.e. clauses in the CNF) containing nested subqueries.*

  Add one correlation table $cr(V_i, Q'_i)$ in the the from clause of the main (outer most) block.

- **Step 2**.*Replace each nested operator v op Q_j with the set comparison v op_{set} r_i, where r_i is the corresponding correlation name for $cr(V_i, Q'_j)$.*

- **Step 3**. *Remove all the sub-formulas (i.e. clauses in the CNF) that used in a correlation table.*

Obviously, Q_{cr} is a new query using correlated tables but without any nested subqueries.

The query $Q_{cr3.3}$ in Example 3.3 is constructed from $Q_{3.3}$ using the algorithm. The following example presents another sample for using the algorithm.

Example 3.4 *Consider the following query $Q_{3.4}$ with the group by clause and a disjunction in the where clause.*

```
SELECT    d.budget
FROM      dept d
WHERE     d.budget > {  SELECT  SUM(e.salary)
                        FROM    emp e
                        WHERE   d.dno = eno } OR
          d.e_cap > {   SELECT  count(*)
                        FROM    emp e
                        WHERE   d.dno = eno }
GROUP BY  d.budget
```

Let Q_8 and Q_9 be the following two queries:

`SELECT SUM(salary) FROM emp` *and*
`SELECT COUNT(*) FROM emp`.
The correlation query $Q_{cr3.4}$ is given below.

```
SELECT     d.budget
FROM       dept d, cr({d.no}, Q8) as r8, cr({d.no}, Q9) as r9
WHERE      d.budget >set r8 OR d.e_cap >set r9
GROUP BY   d.budget
```

Now we present the main result of this section, that is, the constructed Q_{cr} is equivalent to the given query Q.

Proposition 3.1 *The query constructed from Algorithm represents the same result set as the given query with nested subqueries.*

The proof of the proposition follows the specification of the correlation tables and the algorithm.

4 Evaluation strategies of Query Evaluation

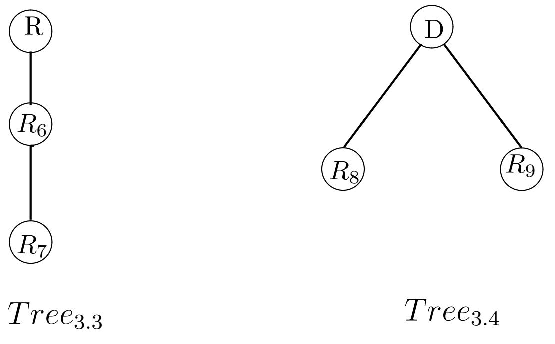
Since a correlation table cannot be evaluated before all its binding values are available, the converted query Q_{cr} must respect the calculation of the binding values.

Let Q_{cr} be the correlation query, and its from clause contains the list of correlation tables $R_1, R_2, \ldots, R_n$, then the evaluation tree is a tree with n nodes such that

- each node is labeled with exactly one relation R_i, and
- if a free variable v of R_i refers to a column variable in R_j then R_j is an ancestor node of R_i.

Obviously, the query Q_{cr} must be evaluated top down in a evaluation tree.

Example 4.1 *The following two evaluation trees are for $Q_{cr3.3}$ and $Q_{cr3.4}$ respectively.*



We are currently implementing the proposed framework in RubatoDB, and consider the following following optimization strategies.

- Key and Index search
- Sorting of the output bindings and caching of the correlated tables.
- Restart segment scan

5 Conclusion and Future works

We have proposed the correlation table used for nested query evaluation and present an algorithm to convert any query to the correlation queries without nested subqueries. Our work provides a much needed approach to evaluate nested queries for distributed database systems.

We are going to adapt various query optimization techniques and to conduct experimental tests to further verify the proposed framework.

References

[1] L. Yuan, L. Wu, J. You, and C. Yan, "Rubato db: A highly scalable staged grid database system for oltp and big data applications," in *Submitted for publication*, 2014.

[2] W. Kim, "On optimizing an sql-like nested query," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, 1982.

[3] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *ICDE*, pp. 450–458, 1996.

[4] G. Graefe, "Executing nested queries," in *BTW*, pp. 58–77, 2003.

[5] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD Conference*, pp. 23–34, 1979.

[6] R. Guravannavar, H. S. Ramanujam, and S. Sudarshan, "Optimizing nested queries with parameter sort orders," in *VLDB*, pp. 481–492, 2005.

[7] Y.-J. Kang, C.-H. Choi, K.-E. Yang, H.-G. Kim, and W.-S. Cho, "An efficient nested query processing for distributed database systems," in *ICHIT (2)*, pp. 669–676, 2011.

[8] M. Muralikrishna, "Improved unnesting algorithms for join aggregate sql queries," in *VLDB*, pp. 91–102, 1992.