International Journal of Computer Applications (0975 – 8887)
Volume 149 – No.2, September 2016

Late-Materialization using Sort-merge Join Algorithm

Nachiket Deo
Department Of
Information Technology,
Savitribai Phule
Pune University.

ABSTRACT
In this paper, we study the use of Late-materialization for sort-merge join algorithm. We study various effects of using this strategy and also compare it with other techniques like pipelining.

General Terms
Databases, Query processing, Join algorithm.

Keywords
Sort-Merge Join, Materialization.

1. INTRODUCTION
Query processing is very important part in databases. Several queries are fired on the database during an application life-cycle. In this paper we will be dealing with Read-Only queries. This includes queries including SELECT statement, Aggregation queries and Joins. As the title suggests, we will be mainly dealing with Joins. Application developer mainly needs to understand the business goal and then create various queries to satisfy the various that goal. From developer's perspective the joins that are used are INNER, OUTER joins and there sub-types. The aim of this paper is not to study these joins. The internal mechanisms that query processor uses are as follows:

1) Nested-Loop Join.
2) Hash-Join.
3) Sort-Merge Join.

2. LATE MATERIALIZATION
In a particular query plan, various tables are accessed. The process of accessing several columns to create a record is known as tuple creation. Creation of tuples based on the information present in the query plan is known as *materialization*. In a query plan tuple is created if the join condition is satisfied. The process where the tuples (where join condition is satisfied) are stitched together as soon as possible during a query plan is known as *Early materialization.* The process where the tuples aren't created until some part of the query plan is processed (apart from the checking of the join condition) is termed as *Late-materialization* [1]. It generally is the operation specified in the WHERE condition of a query. This process is useful because it saves the CPU from performing operation of joining the unnecessary tuples [3].

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
v1	v2	v3	v4	v5	v6	v7	v8	v9	v10

Figure 1: Column-Store

The underlying database system which is under consideration is column-store. As shown in above figure for a particular relation data is stored in form of columns in contrast to row-store. So here the information about a particular attribute is placed in single column and hence, any query that needs to access specific column only needs to access that particular column rather than whole table [2][4].

3. SORT-MERGE JOIN

3.1 Normal Sort-Merge Join Algorithm
Let R and S be the join attributes. Let's assume that the attributes are sorted in ascending order.
Let pr be the address of the first tuple of r.

Let ps be the address of the first tuple of s.
WHILE (ps!= NULL AND pr != NULL) do
BEGIN
ts be the tuple to which ps points .
WHILE (ps!= null)
Let ts' points to the next tuple of s.
IF ts[attribute]=ts'[attribute] then
BEGIN
S:-(S U ts')
Let ts point to tuple next to ts'
END END.
Let tr point to the tuple which pr points.
WHILE (*pr* = null **and** *tr* [*JoinAttrs*] < *ts* [*JoinAttrs*])
DO
BEGIN
set *pr* to point to next tuple of *r*;
tr := tuple to which *pr* points;
END

WHILE (*pr* = null **and** *tr* [*JoinAttrs*] = *ts* [*JoinAttrs*])
DO
BEGIN

FOR EACH *ts* **IN** *Ss*
DO
BEGIN
add *ts* _ *tr* to result;
END
set *pr* to point to next tuple of *r*;
tr := tuple to which *pr* points;
END

3.2 Partial Late Materialization

Executing the following query which includes two table T,R
Query 1:SELECT T.A,T.B,R.B'
FROM T JOIN R ON (T.A=R.A')
WHERE A>5 AND B<50 AND C'<35

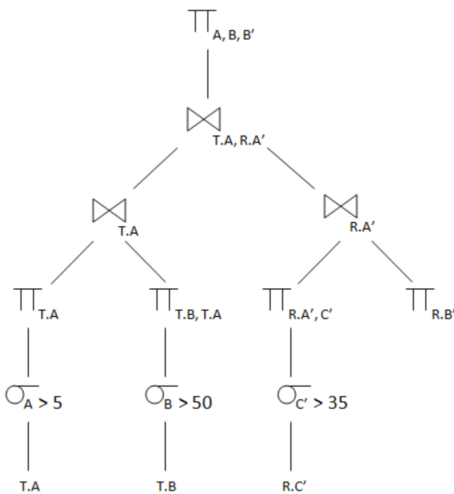

Figure 2 Query Tree for LM-Join (Query 1)

In Query 1, the query first applies the predicates to the columns as specified in the query plan.(Refer to figure 2) In the first leaf of the left sub tree the predicate is applied on the A columns of the T table. Similarly other predicates are applied to B. Similarly the same procedure is performed for the right sub tree. On upper level of the sub tree values are projected. As T.A is our join attribute and also required by the user in final output it is projected. Similarly B is projected. However the T.A' is also projected along with C' in the right sub tree, because it is the join attribute for the R table. In the next step the T.A and T.B columns in left sub tree are joined. For right sub tree something different happens. The R.B' column is not joined similar fashion. As you can see that R.A' column is projected after applying the condition c<35.So for the tuples which satisfy that conditions the respective values of R.B' column are retrieved. This process is called *anti-projection or tuple reconstruction*. Above process is partially late-materialized because the tuples are formed before the joining the two tables but these intermediate tuples are formed after we have applied the predicates. This is a technique is where we can employ Late-materialization without any re access of columns and also the internal join algorithm that can be used is sort-merge join. Now, as you can see that when the join over T.A and T.B columns is performed the value of T.A is also projected. This T.A column is used as a join attribute for sort-merge join algorithm. Using this technique there are lesser number of tuples or only the relevant tuples are joined during the final join operation. In the figure 2 at first level of the query tree (that is the position where join over T.A attribute is performed) there are tuples which satisfy the predicates. This is helpful when joining the two tables on the upper level because only relevant tuples are joined and the cost involving the unnecessary joining of tuples is avoided. If the memory is big enough to hold relations while performing the join the disk access is avoided. This makes usage of sort-merge join a good choice.

3.3 Normal SELECT queries

In above section the query under consideration includes joining two tables. This section there's query that involves only single table. Similar to above section, the query will be SELECT statement and won't involve Insertion or updating the records. Table under consideration is Order which has order_id as primary key.

Query 2:

SELECT amount, ship date,item FROM Order WHERE Amount>5000 AND ship date <CONST.

In above query ship date is some constant. The underlying system is column-store so it will contain four columns, Amount, ship date, order_id ,item which are necessary for this particular query. Other columns might be there but as they are unnecessary for this query so column-store avoids the access to these columns.

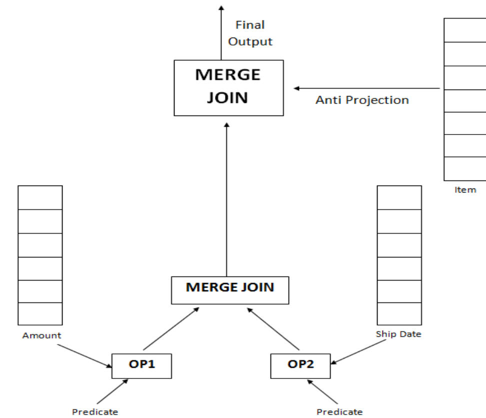

Figure 3 Query plan for Query 2

In column-store, relations contain many different columns which are stored at separate memory locations and that too contiguously. So for a particular query, it has to access more than one column in a table. The above query has to apply predicate on two columns and then project the third column. Here primary key also plays crucial part. For each record there will be order_id and it can be used to point to a particular record. So after applying predicate the order_id is projected as well. Using order_id sort-merge join is performed and result is the relation which satisfies both predicates. After that step query needs item column. So again *anti-projection* is performed and for the tuples that satisfy the predicates we query the respective values from item column. Due to column-store structure it's possible to join various columns from a table internally [2] [3].

3.4 Full Late Materialization

The most widely used way to implement late-materialization is by using position list. It's highly compressible data structure which stores the positions of the records which satisfy the predicates. Furthermore, position lists are created for other columns as well. The bitwise AND (or any other Boolean operation specified in the Query plan) is performed on the position lists. The final position list which satisfies all the predicates in used to re access the original columns and the final joining operation is performed. Once such algorithm is invisible join [2]. In this type the intermediate tuples are not formed because the predicates are applied on the position-lists rather than actual columns [1].

3.5 Applications in Row-Store

The above mentioned approaches are designed for column-store. Late-Materialization can have significant benefits for row-store in few cases. During a Join operation partial late-

materialization can be useful because fewer tuples will be joined during the final join operation. However, late materialization cannot be implemented for normal SELECT queries due to the internal layout of row-store.

4. CONCLUSION

1. Partial Late Materialization can be used to avoid re accessing the columns.
2. If columns need to be accessed from the disk sort-merge join algorithm is better because it only accesses the columns once.
3. Partial late-materialization can increase the number of join operation performed in a query plan.
4. For column-store late-materialization can give better performance for SELECT queries

5. FUTURE SCOPE

There's need to develop a query processor that provides the functionality of late-materialization. Test the use of late-materialization for Row-store. Testing whether Late-materialization is useful for all types of queries.

6. ACKNOWLEDGMENTS

We thank Mr. Prashant Patil for helping in editing work.

7. REFERENCES

[1] Daniel J. Abadi ,Daniel S. Myers, David J. DeWitt, Samuel R. Madden. Materialization Strategies in a Column-Oriented DBMS. Proceedings of ICDE 2007, Istanbul, Turkey.

[2] Daniel J. Abadi, Samuel R. Madden, Nabil hachem. Column-Store vs Row-store How different are they really? SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada

[3] Daniel Abadi,PeterBoncz,Stavros Harizopoulos, Stratos Idreos, Samuel Madden The Design and implementation of modern column-oriented database.

[4] A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database Hasso Plattner Hasso Plattner Institute for IT Systems EngineeringUniversity of Potsdam