Reihe Informatik
04 / 2002

Compiling Away Set Containment and Intersection Joins

Sven Helmer Guido Moerkotte

Compiling Away Set Containment and Intersection Joins

Sven Helmer and Guido Moerkotte

April 22, 2002

Abstract

We investigate the effect of query rewriting on joins involving set-valued attributes in object-relational database management systems. We show that by unnesting set-valued attributes (that are stored in an internal nested representation) prior to the actual set containment or intersection join we can improve the performance of query evaluation by an order of magnitude. By giving example query evaluation plans we show the increased possibilities for the query optimizer.

1 Introduction

The growing importance of object-relational database systems (ORDBMS) [8] has kindled a renewed interest in the efficient processing of set-valued attributes. One particular problem in this area is the joining of two relations on set-valued attributes [4, 5, 7]. Recent studies have shown that finding optimal join algorithms with set-containment predicates is very hard [1]. Nevertheless, a certain level of efficiency for joins on set-valued attributes is indispensable in practice.

Obviously, brute force evaluation via a nested-loop join is not going to be very efficient. An alternative is the introduction of special operators on the physical level of a DBMS [5, 7]. Integration of new algorithms and data structures on the physical level is problematic, however. On one hand this approach will surely result in tremendous speed-ups, but on the other hand this efficiency is purchased dearly. It is very costly to implement and integrate new algorithms robustly and reliably.

We consider an alternative approach to support set-containment and non-empty intersection join queries by compiling these join predicates away. The main idea is to unnest the set-valued attributes prior to the join. Thereby, we assume a nested internal representation [6]. This is also the underlying representation for the specific join algorithms proposed so far [5, 7]. Whereas [7] concentrates on set-containment joins, we also consider joins based on non-empty intersections. Ramasamy et al. also present a query rewrite for containment queries in [7], but on an unnested external representation, which (as shown there) exhibits very poor performance. Further, the special case of empty sets was not dealt with.

The goal of our paper is to show that by rewriting queries we can compile away the original set-containment or intersection join. As our experiments with DB2 show, our

rewrite results in speed-up factors that grow linearly in the size of the input relations as compared to quadratic growth for brute-force nested-loop evaluation. The advantage of this approach—as compared to [5, 7]—is that no new join algorithms have to be added to the database system.

The paper is organized as follows. In Section 2 we briefly describe some preliminaries. We list the treated query types along with their respective rewrites in Section 3. In Section 4 we give a description of the environment in which the experiments took place. After that we present the results of the experiments in Section 5. Finally, Section 6 concludes our paper.

2 Preliminaries

In this section we give an overview of the definition of the set type. Due to the deferral of set types to SQL-4 [3], we use a syntax similar to that of Informix [1]. A possible example declaration of a table with a set-valued attribute is:

```
create table ngrams (
   setID   integer not null primary key,
   content set<char(3)>
);
```

`setID` is the key of the relation, whereas `content` stores the actual set. The components of a set can be any built-in or user-defined type. In our case we used `set<char(3)>`, because we wanted to store 3-grams (see also Section 4). We further assume that on set-valued attributes the standard set operations and comparison operators are available.

Our rewriting method is based on unnesting the internal nested representation. The following view defining the unnested version of the above table keeps our representation more concise:

```
create view view_ngrams(setID, d, card) as (
   (select ngrams.setID, d.value, count(ngrams.content)
    from ngrams, table(unnest<char(3)>(ngrams.content)) d)
   union all
   (select ngrams.setID, NULL, 0)
    from ngrams
    where count(ngrams.content) = 0)
);
```

where `setID` identifies the corresponding set, `d` takes on the different values in `content` and `card` is the cardinality of the set. We also need `unnest<char(3)>`, a table function that returns a set in the form of a relation. As `unnest<char(3)>` returns an empty relation for an empty set, we have to consider this special case in the second subquery of the union statement, inserting a tuple containing a dummy value.

[1] http://www.informix.com/documentation/

3 The Queries

We are now ready to describe the queries we used to compare the nested and unnested approach. We concentrate on joins based on subset-equal and non-empty intersection predicates, because these are the difficult cases as shown in [1]. We have skipped joins involving predicates based on equality, because the efficient evaluation of these predicates is much simpler and can be done in a straightforward fashion (see [5]).

3.1 Checking Subset Equal Relation

Here is a query template for a join based on a subset-equal predicate:

```
select n_1.setID, n_2.setID
from   ngrams n_1, ngrams n_2
where  is_subseteq(n_1.content, n_2.content) <> 0;
```

(The comparison with 0 is only needed for DB2, which does not understand the type bool.)

This query can be rewritten as follows. The basic idea is to join the unnested version of the table based on the set elements, group the tuples by their set identifiers, count the number of elements for every set identifier and compare this number with the original counts. The filter predicate `vn1.card <= vn2.card` discards some sets that cannot be in the result of the set-containment join. We also consider the case of empty sets in the second part of the query. Summarizing the rewritten query we get

```
(select vn1.setID, vn2.setID
 from   view_ngrams vn1, view_ngrams vn2
 where  vn1.d = vn2.d
 and    vn1.card <= vn2.card
 group by vn1.setID, vn1.card, vn2.setID, vn2.card
 having   count(*) = vn1.card)
union all
(select vn1.setID, vn2.setID
 from   view_ngrams vn1, view_ngrams vn2
 where  vn1.card = 0);
```

3.2 Checking Non-empty Intersection

Our query template for joins based on non-empty intersections looks as follows.

```
select n_1.setID, n_2.setID
from   ngrams n_1, ngrams n_2
where  intersects(n_1.content, n_2.content) <> 0;
```

The formulation of the unnested query is much simpler than the unnested query in Section 3.1. Due to our view definition, not much rewriting is necessary. We just have to take care of empty sets again, although this time in a different, simpler way.

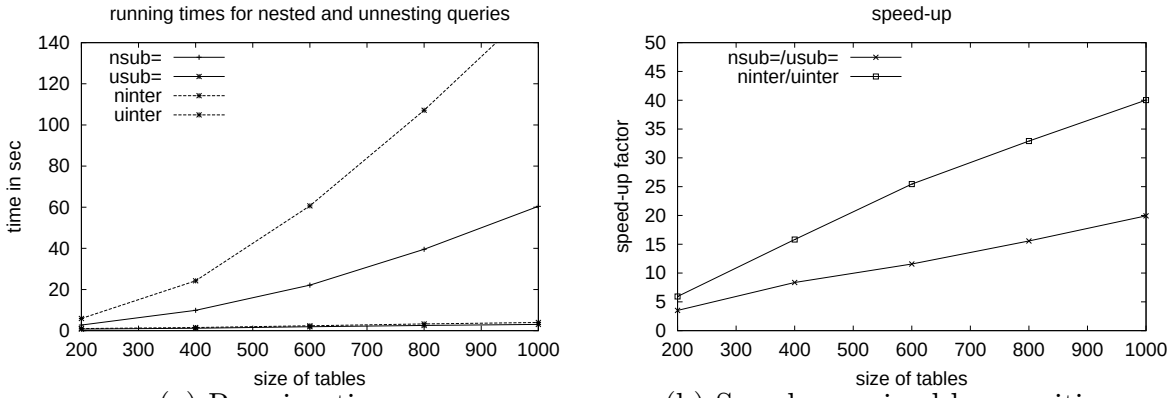
```
select distinct vn1.setID, vn2.setID
from   view_ngrams vn1, view_ngrams vn2
where  vn1.d = vn2.d
and    vn1.card > 0;
```

4 Experimental Environment

The experiments were run on a lightly loaded Sun Enterprise Server E450 under Solaris 2.6 with two 300 MHz Ultra Sparc processors and 512 MByte of main memory. For the database we used UDB version 6.1 from IBM. The user-defined type and functions were implemented in C++ using the GNU C++ Compiler 2.95.2. For performance reasons all user-defined functions were run unfenced.

We used real data to load the tables by storing words taken from a dictionary containing all different words from the King James Bible. However, we did not insert the words directly into sets, but generated a set of 3-grams for each word (e.g., "along" has the set {alo, lon, ong}) and assigned to each set the set of 3-grams of that word. N-grams are usually used for queries with partially specified terms [9]. After loading the tables we updated the statistics of the database systems by running a runstat command in order to improve the results of the query optimizer. The level of the optimizer was set to 5 (which is the default value). Raising it to higher levels did not improve the query plan, it only resulted in a longer total running time, because the optimizer needed more time.

5 Results



(a) Running times (b) Speed-up gained by rewriting

Figure 1: Results of experiments

In Figure 1(a) the results of our experiments are depicted (`n` stands for nested, `u` for unnesting queries). We measured the query performance in terms of running time (the unit of measurement on all y-axes is in seconds). The resulting tuples of the queries were not sent to output, as we started all queries with the command `db2batch -o r 0`. For each point in Figure 1 we averaged the evaluation times of ten different executions of a particular query. The rewritten queries perform much better (up to 40 times better for

intersection predicates and still up to 20 times better for subset equal predicates). The observed performance improvement factors are similar to those achieved applying special join algorithms in the main memory case [5].

The performance gap between the nested and unnested version increases steadily as can be seen in Figure 1(b), where the ratio between the running times of nested and unnested queries is plotted. While the nested queries have a running time quadratic in |`ngrams`| (where |`ngrams`| is the cardinality of the relation `ngrams`), due to the nested-loop algorithm, the rewritten queries have a sub-quadratic running time.

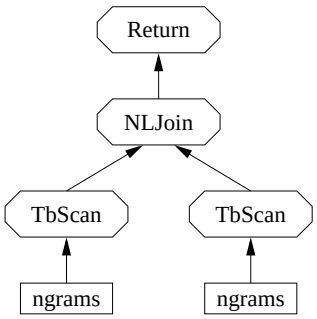


Figure 2: Execution plan for nested query

What are the reasons for this behavior? We took a closer look at the query plans generated by the optimizer via Visual Explain provided by the Control Center of DB2. The plan generated for the nested queries looks very simple. A schematic version is shown in Figure 2. As the join predicate involves a user-defined function, DB2 is forced to use a nested-loop join operator. This is very expensive in terms of evaluation costs, because the set comparison operator has to be called $|\texttt{ngrams}|^2$ times.

The query plans for the unnested queries look much more complex (see Figure 3 for a schematic version of the plan for the query $u_{\subseteq}$). In the lower part of the plan the view definition is resolved. We have two major branches from here on. In the branch on the right hand side we handle the special case for empty sets and in the branch on the left hand side we evaluate the comparison of all other sets. The results of both branches are then combined by a union operator. During the evaluation of this query a faster sort-merge join algorithm can be used.

For the query $u_{\cap}$ the lower part of the plan is identical to that of Figure 3. Only one branch containing a simple sort-merge join and a filter is needed after resolving the view, i.e., we have no separate branch to handle empty sets.

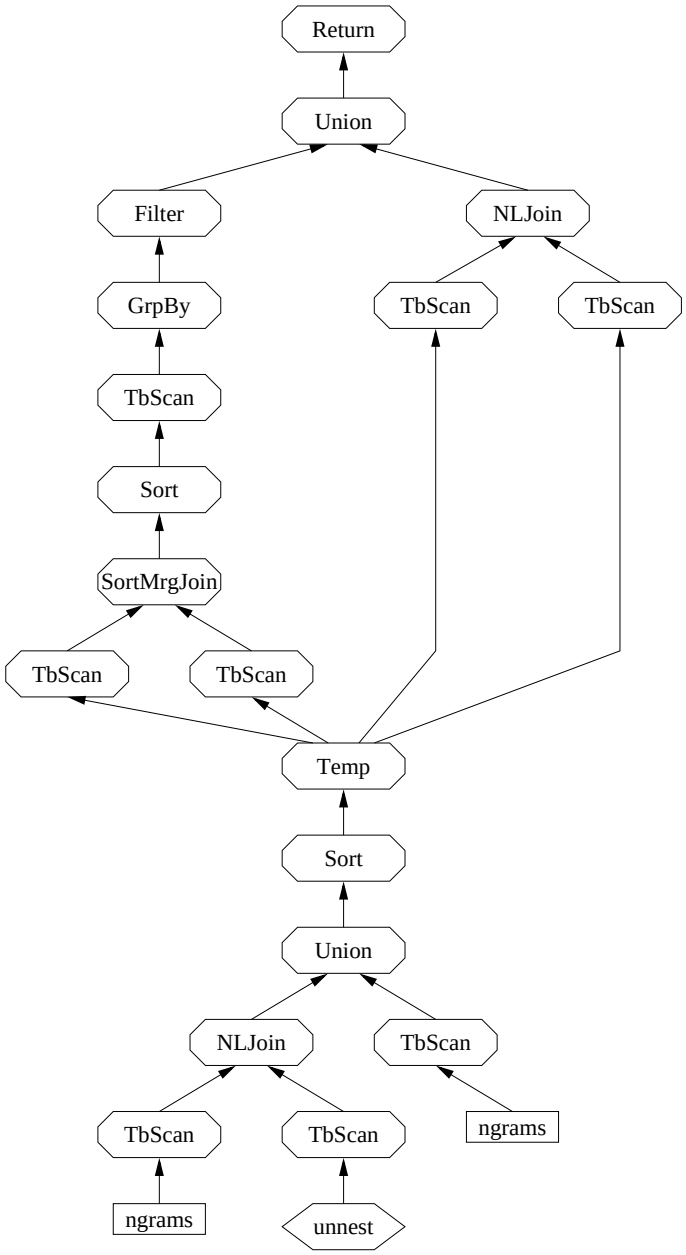


Figure 3: Execution plan for unnested query ($u_{\subseteq}$)

6 Conclusion

We compared the execution of queries involving joins on set-valued attributes, all of which were stored in a nested representation. We tested two different variants of each query: a nested version, in which most of the join predicate was evaluated in a user-defined function and a version where the set-valued attributes were unnested before the actual join took place. Additionally, in the unnested version the whole join predicate was formulated in plain SQL. In our experiments the unnested evaluation of the queries was the obvious winner. For relation sizes of 1000 tuples the unnested variants of the queries were 20 to 40 times faster than their nested counterparts. For larger relations we expect the gap to widen even more, because of the asymptotic running times of the different algorithms. The reason for this behavior is clearly the inability of the query optimizer to deal with complex (user-defined) types and functions efficiently.

Existing ORDBMS only support access to and indexing of complex (user-defined) objects in a rather primitive way. There are efforts to improve this situation, e.g. by allowing user-defined access methods and tightly coupling these with the database engine [2, 8]. This approach, however, is not an option as long as it is still in its infancy and not widely available in commercial DBMS. The consequence for joins of relations on set-valued attributes (and also probably for many other user-defined types and functions) must be to grant the query optimizer access to the functionality of the complex objects instead of hiding it in user-defined functions. Only by this means can we attain acceptable performance without having to modify the database engine itself.

References

[1] J.-Y. Cai, V.T. Chakaravarthy, R. Kaushik, and J.F. Naughton. On the complexity of join predicates. In *Proceedings of the Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.

[2] W. Chen, J.-H. Chow, J. Grandbois Y.-C. Fuh, M. Jou, N. Mattos, B. Tran, and Y. Wang. High level indexing of user-defined types. In *Proc. of the 25th VLDB Conference*, pages 554–564, Edinburgh,Scotland, 1999.

[3] P. Fortier. *SQL-3, Implementing the SQL Foundation Standard*. McGraw-Hill, 1999.

[4] G. Garani and R. Johnson. Joining nested relations and subrelations. *Information Systems*, 25(4):287–307, 2000.

[5] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with subset join predicates. In *Proc. of the 23rd VLDB Conference*, pages 386–395, Athens, August 1997.

[6] K. Ramasamy, J. Naughton, and D.Maier. High performance implementation techniques for set valued attributes. Technical report, Computer Sciences Department, University of Wisconsin, Madison, 2000.

[7] K. Ramasamy, J.M. Patel, J.F. Naughton, and R.Kaushik. Set containment joins: The good, the bad, and the ugly. In *Proc. of the 26th VLDB Conference*, Cairo, Egypt, August 2000.

[8] M. Stonebraker and P. Brown. *Object-Relational DBMSs, Tracking the Next Great Wave*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

[9] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Morgan Kaufmann, San Francisco, 1999.