CERIA Research Report 2006-05-27, May 2006

Structured Relational Views Using Multiquery Group By

Witold LITWIN

CERIA, U. Paris Dauphine

(Position Paper)

27 May 2006

1. Introduction

There is a large trend these days to use structured data. This idea seems the motivating one for the XML galaxy. In its background there is an apparent insufficiency of the relational database systems in the handling of such data. The result of a relational query is indeed a (flat) relation. Such result may be not satisfactory for more and more users. The issue seems rather ignored by the relational database system designers. Unsatisfied users brought in response a rather revolutionary revision of the acquired database knowledge. This one has several unfortunate side-effects. A benign one is the renaming of several SQL operations, e.g., **Order By** becoming **Sort** in XQuery. More concerning is the return of old ideas proven not best for the database management, e.g., in Cobol and Codasyl databases. For instance, since these ideas bring back the stored data redundancy and the insert and update anomalies. On the top, it turns out that, at least for now, the best practice to store the structured data, e.g., XML, is still in a relational database. This generates a modern version of impedance mismatch, clear for every database folk approaching XML.

A scrutiny of the relational model principles shows however that the view of relational data as structured instances whenever needed, and not only as (flat) relations as by the general approach today, was a preoccupation since the inception of the model. One may see such a view as some composition of the (flat) relations[1]. Reversing into structured form, the decomposition into flat relations in xNF, of the structured data being the typical input to such a decomposition.

In other words, a structured view should simply show the tuples produced by a collection of SQL queries. The tuples from different tables should be however composed further in a way typical for the structured data. Again, at a closer look, one may see this composition as a simple grouping of tuples forming conceptually together an instance of the structured view, e.g., a document in XML vocabulary. The grouping was already an intention of **Group By** operator applied, however, up to now to a single SQL query. Here, its meaning should be extended to the result produced by multiple queries. The extension should aim on the non-redundancy of data, characteristic of the relational model, but applied in practice up to now only to the stored data storage. In this sense, the grouping principles should be the 2nd "leg" of the relational model, hinted to but cached in practice till now. We propose such an extension of the usual **Group By** and call it *multiquery* or *multirelational* **Group By**. This is almost the whole proposal that follows. We also propose a new command termed **Create Structured View**. This one names and makes persistent the collection of queries subject of the *multiquery* **Group By**.

[1] The "Composition" section in the very first report by E. F. Codd, Aug. 19, 1969.

Our approach is free of the above discussed disadvantages. It is evolutionary, keeping in place SQL and the acquired advantages of the relational technology. It is free of the impedance mismatch problem. Finally, as it will appear, it could be more powerful than structures manipulated by XML tools today. One should be able to generate for instance structured views over the transitive closure.

We now back up our claims with motivating example. Details, including the implementation issues will make later papers.

2. Structured Views by Example

Example 1. Consider the database with the following tables**.** Table **P(<u>P#</u>, Name)** contains some persons. Tables **H (<u>P#</u>, <u>Hobby</u>), F (<u>P#</u>, <u>Friend</u>)** and **R (<u>P#</u>, <u>Rest</u>)** indicate respectively the hobbies, friends and preferred restaurants of each person**.** There are the referential integrity constraints between **P** and each other table. One may be interested in producing for each person a document with all the related information. The classical approach would be the SQL query:

(Q1) Select P.P#, Name, Hobby, Friend, Rest from P, H, F, R where P.P# = H.P# and P.P# = F.P# and P.P# = R.P# ;

Consider further, for instance, that a person in **P** has on the average 10 hobbies, 10 friends and 10 restaurants. Unfortunately, the query would produce then for, let us say a thousand folks, 1M tuples under our assumptions. Obviously, the vast majority, i.e., more than 99% of attribute values, would be the highly redundant data, repeated many times among the tuples. Next, the sheer number of tuples would blur the utility of the result. In a word, the result would be useless in practice. Surprisingly, this annoying aspect of the relational model seems largely ignored of the literature. Being, perhaps intuitively the rationale for its above discussed rejection by a large crowd.

Another approach could be simply the set of four queries:

(Q2)

Select P.P#, Name, from P ;
Select P.P#, Hobby from H ;
Select P.P#, Friend, from F ;
Select P.P#, Rest, from R ;

This result also contains all the data required. Its average result size is $N + 30N$ tuples for N tuples in **P**. Hence, only 31K tuples, in our case, most of them also with only 2 values per tuple only (instead of five above). This is, the outcome is now more than 30 times more compact. Unfortunately, it has the drawback which is that the tuples are badly grouped for our need. Two tuples of the same person can be identified by **P#** value, but are separated in our case by a thousand of other tuples. Query **(Q**2) is thus as useless as **(Q**1) for an average user.

Nevertheless, **(Q**2) remains obviously a better basis for a useful outcome than **(Q**1). It contains only little more than the minimal information needed, we show below. What remains is a fix to the grouping problem. The goal is to have the outcome with the tuples of the same person grouped together. This is the intended semantics of the final following query. It is almost the same as of **(Q**2**).** Except for the final **Group By** operator, and of dropping of **P#** in the **select** lists in the subsequent queries. It is a beneficial side effect of the better grouping as explained in detail below.

(Q3)

Select P.P#, Name, from P ;
Select Hobby from H ;
Select Friend, from F ;
Select Rest, from R ;
Group By ;

The final **Group By** in (Q3) has a new semantics. It forms a group for each tuple produced by the 1st query, i.e., each tuple of **P** in our case. The indication that it refers to this query and table are implicit for the user convenience. The fuller and full syntax could be **Group By [1]** and **Group By [1].P**. Each selected tuple of **P** becomes the *root* of one group. The other elements are the tuples from H, F and R that share with the root the P# value, i.e., were subject of implicit joins of the referential integrity constraint. If we wanted to make these joins explicit (what should be done for the query evaluation internally anyhow) our *multiquery* would become:

Select P.P#, Name from P;
Select Hobby from H where H.P# = P.P#;
Select Friend, from F where F.P# = P.P#;
Select Rest, from R where F.P# = P.P#;
Group By ;

The default order in each group is that of the queries in **(Q3)**. The final outcome presents the tuples grouped accordingly. We thus would have in each group one person, followed by all the hobbies, followed by all the friends of the person, with all the restaurants at the end. Hence something looking like the text at Fig. 1.

Notice that the outcome is not a relation, as the tuples are not union compatible. The outcome is a multi-relation, in the sense of MSQL language. A form editor, like under, e.g., MsAccess could obviously present it in a nicer way. For instance, as a form with three linked subforms. Obvious additions to the **select** list of each query could on the other hand decorate this result with XML or HTML like tags. Finally, observe that the outcome at Fig. 2 is free of any data redundancy. Like the stored normalized data in our database. This is the illustration of our comment about our grouping being the missing 2nd leg of the relational model.

```
P
P1 Witold
  H
  Tennis
  Ski
   …
   Databases

  F
  Alexis
 ….
  Alfred
  R
  Miyake
  Louis 13
P
P2, Elisabeth
  Cooking
   ….
```

Fig. 1: Structured view defined by query (Q3) with multiquery Group By

Query **(Q3)** is particularly simple. The grouping specifications are entirely implicit, by default. Also there is no condition on the selection of the root tuples. Simply every tuple of **P** is a root. Next example shows more complex needs and the structured views responding to.

Example 2. Consider the classical Supplier Part database with the tables:

S (<u>S#</u>, SName, City, Status), P ((<u>P#</u>, PName, Color, Weight), SP (<u>S#</u>, <u>P#</u>, Qty)

A user wishes now to know all the suppliers in Paris and, for each, the quantity of each supply together with all the data about the part supplied. This structured view would be defined by the simple grouping as follows:

(Q4)

Select * from S where City = « Paris » ;
Select Qty, P.* from SP, P where SP.P# = P.P# ;
Group By;

As before, the grouping calculus explores here the referential integrity constraints, between **S** and **SP** in this case. Notice that the implicit joint that results from, restricts then the meaning of the 2nd query with respect to the usual one it would have alone. Only the supplies with a supplier in Paris would appear in the groups. We leave the reader to trial out the **(Q4)** internal representation with all the explicit joins and the outcome. If you're in trouble, wait till what follows.

Consider now further that **SName** is a candidate key. The user wishes the name of each supplier in the same city as some part. For the selected supplier one needs further (i) the total quantity of all the parts it supplies, and (ii) the total quantity and the number of parts it supplies with the individual quantity of the supply smaller than 100. For the latter we also need the id, the name and the actual quantity of each part concerned. The resulting structured view could be defined by **(Q5)** below.

(Q5)

With S1 as Select S#, Sname From S, P where S.City = P.City;
Select SName from S1;
Select Sum(Qty) As Sum1 from SP, S1 where S1.S# = SP.S# Group By S# ;
With SP1 as Select S#, Sum(Qty), count (*) from SP, S1 where S1.S# = SP.S# and
Qty < 100 Group By S# ;
Select Sum(Qty) As Sum2, count (*) From SP1 ;
Select P#, PName, Qty from P, SP, SP1 where SP.P# = P.P# and SP1.S# = SP.S#;
Group By;

Here, the 1st query creates a temporary view **S1** that disappears after the execution of the multiquery . To create S1 here is not a necessity, merely a convenience for easier expression of the structured view. The **With** statement itself comes from the recursive query formulation in DB2. Notice, although we do not discuss it further here, that this clause paves potentially the way towards the structured views with the transitive closure as well (the capability seemingly yet unknown to XML). The view contains **S#** attribute for the join necessary to calculate the grouping within the result of 3rd query. We did not want however this attribute in the outcome. That is why the 2nd query filters it out. It does it somehow like the **Return** clause of XQuery. The 4th query uses again **Create Temp View** statement, for the similar purpose. The next **Select** filters out **S#**, since we do not want it to appear. The last **Select**, provides the details of the supplies aggregated by the previous **Select**. Finally the **Group By** produces again the multi-table groups. The production of these groups goes here as follows, for the outcome somehow like at Fig. 2. The figure also shows the schema of the structured view and of the group, including the join clauses that the grouping calculus below evaluates.

The roots are the tuples of **S1**, projected by the 1st select above. These tuples are some but not all tuples of **S**. The root choice is again implicit here. The **Group By** statement addresses by default the table produced by the 1st query, i.e. **S1** here. In other situations, easy to imagine, the statement may need to be however more explicit. Once the root is chosen, next element of the group is the tuple from the 2nd **Select,** matching the join clause. There is no further query that would produce a tuple matching a join clause with the tuple just selected. Hence, next element of the group is picked up in the view derived from **SP1** temporary view. These are all the tuples matching again the root through the join clause. Actually, there is only one such tuple. The evaluation of the group continues through the last query. This one brings to the group again the tuples matching the currently selected **SP1** tuple. These show the required details of the aggregate values above them. There is no more queries to evaluate, hence we move to the next root, if any.

```
SName
S1
  Sum1
   300
  Sum2   Count
   200      3
      P#   PName   Qty
      P1  screw      90
      P2  Cog         50
      P4   bolt       60
SName
S2
…..
```

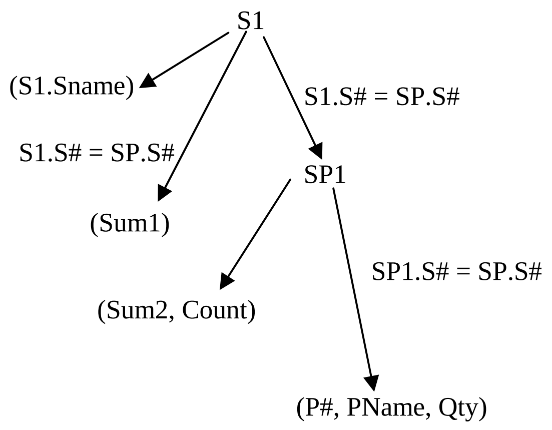

Fig. 2 A detail and the schema of the structured view with join clauses defined by (Q5).

As one sees, the outcome is rather clear. The figure also shows the schema of the structured view and of the group, including the join clauses that the grouping calculus evaluates. The multiquery **Group By** evaluation for each group corresponds to the preorder traversal of the schema. The indentation of the text at both figures follows the *node level*, reflecting the number of join clauses traversed from the root. Notice that the **Select** commands immediately following the **With** commands at the figure, are therefore at the same level as their temporary views. The outcome could be obviously again made easily better decorated, e.g., through a form editor or into an XML document.

Notice that using a little more sophisticated approach to the implicit joins though the referential integrity constraints, one could formulate **(Q5)** less procedurally as follows. The additional rule is that the join(s) through this constraint is expanded with any restrictions on the root.

Select Sname From S, P where S.City = P.City;
Select Sum(Qty) As Sum1 from SP Group By S# ;
Select Sum(Qty) As Sum2,, count (*) from SP, S where S.S# = SP.S# and Qty < 100 Group By S# ;
Select P#, PName, Qty from P, SP where SP.P# = P.P# ;
Group By;

The additional rule is that the join(s) through this constraint is expanded with any restrictions on the root. Hence the 2nd query for instance would become internally:

Select Sum(Qty) As Sum1 from SP, S, P where S.S# = SP.S# and S.City = P.City
Group By S# ;

Observe finally that **(Q5)** shows the need for some formal work around the structured views. Especially, it seems to make sense to talk about a *well-formed* multiquery, defining a connected schema. Finally, observe that one may wish to save **(Q5)** for future use. It appears then useful to add to SQL the statement **Create Structured_View** <structured_view_name> **as** <multiquery >, with the syntax modelled upon that of **Create View**. Here, <multiquery > would stand thus respectively for either **(Q2**), **(Q5) or (Q5)**.

3. Conclusion

Structured views of relational databases seem to be in growing demand. Relational database designers seem slow in responding. The situation left place for the proposals of quite radically new types of databases. Unfortunately the current proposals seem to often revive old daemons, put away by the relational systems. Especially, - of the redundant data storage and (subsequent) presence of insert, or update anomalies, problems solved for a relational database by the normalization into xNF. Our examples show that a revolutionary approach is not the only way, since the creation of structured views over relational data may be surprisingly easy. The key is the adequate grouping of tuples of multiple SQL queries. The groups are the structured instances that the others call differently, e.g., documents. Our examples show that to define the groups, rather minimal extensions to SQL syntax and semantics may suffice. It is also easy to see that the grouping principles as they appear from above, allow to satisfy many other practical needs. Thin about books, wine databases, restaurants, hotels…

One should notice here a conceptual difference between our approach and those manipulating the structured data. E.g., using XPath or XQuery. We produce the structured views from the (flat) relational tables only. We do not attempt to transform structured views into other views. Every new view is produced entirely from the relations by a new multiple query. Nevertheless, there is also the room for a language directly manipulating a structured view, e.g., XQuery or XUpdate like, from, perhaps, a tagged mapping tree on the view.

Finally, the examples clearly only hint how the proposed approach should work. The general formulation of the grouping algorithm remains nevertheless to be done. Another challenging hole is of course the efficient implementation of the multiquery **Group By**.