Case for Explicit and Implicit LIST Aggregate Function for Relational Databases

Witold Litwin[1]

Abstract

We argue for a new aggregate function we termed the LIST function. It aggregates a set of values of one or more attributes into a single value that is internally a list of these values, perhaps ordered. The principle may seem a formal twist, but should be useful in practice. It overcomes important limitations of the current relational systems, due to the use of relations in first normal form, and the separation between the aggregate and the individual data values in the standard SQL. LIST function can be made often implicit, making its use even less procedural. The function should be basically simple to implement. The relational systems already provide most of the capabilities it requires to the existing aggregate functions.

1 Introduction

A relational database system (RDBS), e.g., MsAccess, SQL Server, DB2 or Oracle basically uses today relations in 1st normal form (1 NF), [K0], [IBM], [LGG2], [MS], [S], [LGG2]. The attribute values are supposed atomic. An aggregate function in an RDBS takes a selected set of values and produces a single one, e.g., the sum. In the classical example of Supplier-Part database S-P, described in many books, one calculates in this way, using the GROUP BY clause, the sum of quantities per supplier S# from the table SP (S#, P#, QTY), Figure 1, [D2].

S#	P#	Qty
s1	p1	300
s1	p2	200
s1	p3	400
s1	p4	200
s1	p5	100
s1	p6	100
s2	p1	300
s2	p2	400
s3	p2	200
s4	p2	200
s4	p4	300
s4	p5	400

```
SELECT SP.[S#], Sum(SP.Qty) AS [Total Qty]
FROM SP
GROUP BY SP.[S#];
```

S#	Total Qty
s1	1300
s2	700
s3	200
s4	900

Figure 1 The classical (i) SP table from the Supplier-Part relational database and (ii) query with GROUP BY clause calculating the total quantity of parts per supplier

In the era of data mining, an application may also often need the individual quantities contributing to the sum for each supplier. The way to do it in standard SQL is to issue a separate query SELECT * FROM SP. One cannot indeed mix this result with the aggregated one in a single standard SQL query, although SQL dialects in some commercial RDBSs offer non-standard extensions for it, as we discuss later on. The result repeats S# value in each tuple of the same supplier as many times as there are parts P# it provides. For instance, six time for supplier S1 in S-P. The repetition results from the 1st NF relational calculus. Both constraints: the need for two queries and the S# redundancy in the result may be annoying for applications

[1] Université Paris 9 Dauphine, mailto:Witold.litwin@dauphine.fr

and found awkward by users, despite the wide acceptance of the 1st NF for the base table SP. The typical solution at present is to either use a 4GL, e.g., the forms for MsAccess, or a programming language, [LGG2]. Both options are beyond SQL. They require additional capabilities from the user and the RDBS does not optimise them, unlike an SQL query, [GUW2].

```
SELECT  P.[SS#], P.Name, F.Friend, R.Rest, H.Hobby
FROM ((P INNER JOIN F ON P.[SS#] = F.[SS#])
INNER JOIN H ON P.[SS#] = H.[SS#])
INNER JOIN R ON P.[SS#] = R.[SS#]
WHERE P.[SS#] ="ss1" ;
```

P	Name	Friend	Rest	Hobby
SS1	Witold	Alexis	Mela	Ski
SS1	Witold	Christophe	Mela	Ski
SS1	Witold	Ron	Mela	Ski
SS1	Witold	Jim	Mela	Ski
SS1	Witold	Donna	Mela	Ski
SS1	Witold	Elisabeth	Mela	Ski
SS1	Witold	Dave	Mela	Ski
SS1	Witold	Peter	Mela	Ski
SS1	Witold	Per-Ake	Mela	Ski
SS1	Witold	Thomas	Mela	Ski
SS1	Witold	Alexis	Pizza Napoli	Ski
SS1	Witold	Christophe	Pizza Napoli	Ski
SS1	Witold	Ron	Pizza Napoli	Ski
SS1	Witold	Jim	Pizza Napoli	Ski
SS1	Witold	Donna	Pizza Napoli	Ski
SS1	Witold	Elisabeth	Pizza Napoli	Ski
SS1	Witold	Dave	Pizza Napoli	Ski
SS1	Witold	Peter	Pizza Napoli	Ski
SS1	Witold	Per-Ake	Pizza Napoli	Ski
SS1	Witold	Thomas	Pizza Napoli	Ski
SS1	Witold	Alexis	Ferme de Condé	Ski
SS1	Witold	Christophe	Ferme de Condé	Ski
SS1	Witold	Ron	Ferme de Condé	Ski
SS1	Witold	Jim	Ferme de Condé	Ski
SS1	Witold	Donna	Ferme de Condé	Ski
SS1	Witold	Elisabeth	Ferme de Condé	Ski
SS1	Witold	Dave	Ferme de Condé	Ski
SS1	Witold	Peter	Ferme de Condé	Ski
SS1	Witold	Per-Ake	Ferme de Condé	Ski
SS1	Witold	Thomas	Ferme de Condé	Ski

Record: 1 of 1000

Figure 2 Result of MSAccess SQL query requesting the name, friends, preferred restaurants and hobbies of person identified with 'SS1'.

Similar situation occurs for other needs. We will show some through the motivating examples in next section. At present, notice only that the result is especially awkward if data to store present the multivalued dependencies, as very often. For instance, consider a person identified with SS# who may have several hobbies, friends, and preferred restaurants. The good relational database scheme would separate these data adequately in 4th NF tables, [F77] , [BB79], [D2], [GUW2]. They could be four tables: P (SS#, Name), H (SS#, Hobby), F (SS#, Friend) and R (SS#, Rest). Ten tuples in each table H, F, and R for a person, e.g., (SS1, Witold), would lead to the total of 31 tuples for Witold. However, the application may still need all the data together for SS1, including the name 'Witold'. The SQL query would lead to 1000-tuple relation. Figure 2 shows the query and about top 30 tuples, i.e., 3 % of the result produced by MSAccess. It appears hardly useful for anyone.

The fundamental reason is that any current RDBS, the MSAccess used here being just one example, would create, according to the relational calculus rules in use, all the tuples with all the combinations of a hobby, a friend and of a restaurant. It would also repeat 1000 times that the person's name is Witold. Basically, the query output would be a denormalized relation fragment of the 4th NF relations stored, with the well-known anomalies characterizing a non 4th NF relations, [F77], [D2], [GUW2]. The only solutions at present are basically to either issue four separate SQL queries, missing thus the goal of all the desired data together, or, again, to use a 4GL interface, or a programming language.

In the above examples, one may observe that the problem disappears if one aggregates the values non functionally dependent on others in the query output. This aggregation cannot be done to a single value in the classical sense for an RDBS, such as an integer or real or a few byte long character string. However, one can still aggregate into a **single** value being a list. Internally, the list may be multi-valued, or include a value expression, or a DISTINCT or TOP predicate, or refer to an aggregate function computed elsewhere in the query, or include a scalar function… One can nevertheless assimilate it to a character string. The string can be

possibly longer than a classical one for an RDBS, but it is still a **single** value for the RDBS[2]. Hence the table remains flat, i.e., in 1 NF at least. This is precisely the intention in the *LIST aggregate function* we will discuss here.

In our 1st example, the QTY values should be aggregated in that way into the single list of six values. Only one tuple per supplier will result from. Likewise, in 2nd example, one should be able to have only one tuple for our person to show, with its SS# and name once only, and three comparatively short lists of ten elements each. This, instead of the 1000 tuples in Figure 2.

We proposed the LIST function in [L3]. In what follows, we argue further for it through an expansion of its capabilities. We start by recalling the motivating examples from [L3], and the features of the LIST function they implied there. On this basis, we extend this features with the *implicit* LIST we introduce here. We refer backward to the core form of LIST as *explicit*. We show that a query can mix both forms.

Section 2 recalls the explicit LIST. Section 3 describes the implicit LIST, and motivates it through the analysis of the recalled examples. We show in particular the utility of coupling this capability with that of the implicit equijoins and the implicit FROM clause we propose as well.

Section 4 briefly discusses the implementation of LIST that appears rather easy, and the related work. We conclude in Section 5.

2 The LIST Aggregate Function

We first analyse the need for the function and the specific capabilities it should provide through additional motivating examples. We then propose its basic syntax and semantics. We finally build upon the capability of the implicit LIST.

2.1. Examples

Example 1

Consider again the SP relation. The LIST function should be invoked similarly to the classical query calculating the total quantity per supplier in Figure 1. Thus the query for the total quantity and contributing individual ones together could be:

(Q 1) **SELECT S#, SUM (QTY) AS [TOTAL QTY], LIST (Qty) AS Histogram FROM SP GROUP BY S#;**

S#	Total Qty	Histogram
s1	1300	300, 200, 400,200;100, 100
s2	700	300, 400
s3	200	200
s4	900	200, 300, 400

Figure 3 The expected result of (Q1) with LIST aggregate function requesting together the total quantity and the histogram of parts supplied per supplier.

The expected result would be the table in Figure 3. *There is one tuple per S# with the 4th column of character string type with lists*, e.g. of six values for S1. The lists are presented here horizontally. Longer lists could appear at the screen as a combo boxes, as usual today for MSAccess.

[2] Notice that RDBSs routinely manage longer text attributes, e.g. even the "small" MsAccess accepts 255-byte long strings. This is more than enough for any motivating example below. See also Section 4.

Incidentally, we did not find *any way* to formulate this query as a single one in SQL dialect of MSAccess, even considering the non-standard extension, e.g., the Pivot and Transform clauses. Any hint is welcome.

Example 2

In our 2nd example above discussed, the LIST function should serve as usual in an SQL query:

(Q2) **Select P.SS#, Name, LIST (DISTINCT (Friend)), LIST (DISTINCT (Rest)),**
LIST (DISTINCT (Hobby))
from P, F, R, H
where P.SS# = F.SS# and F.SS# = R.SS# and R.SS# = H.SS# and P.SS# ="ss1"
group by P.SS#, Name ;

The output should be similar, e.g. one tuple with three lists of ten elements each for our example person, Figure 4. Compare this output to the usual one at present of 1000 tuples in Figure 2. Although the table above could appear visually as 0 NF (unnormalized relation with non-atomic attributes, [D2]), it is not. In fact, again, each list is an **atomic** attribute of character string type as any other such attribute in a currently used RDBS. Hence, this table is also in 1 NF at least. We stay in the usual framework of the relational calculus. The presentation of the string is supposed chosen by RDBS here. At Figure 4a, it uses the text boxes for a printout fitting best the available width of the paper sheet. In Figure 4b, it was intended for a screen, each box being a combo box. As usual for MsAccess, only the 1st few values of each list would appear, one in our case[3], till one click into the box, opening it completely.

(a)

P	Name	Friend	Rest	Hobby
SS1	Witold	Alexis, Christopher, Ron, Jim, Donna, Elisabeth, Dave, Peter, Per-Ake, Thomas	Bengal, Cantine Paris 9, Chef Wu, Ferme de Condé, Miyake, Louis XIII, Mela, North Beach Pizza, Pizza Napoli, Sushi Etoile	Bike, Classical Music, Good food, Hike, Movie, Science Fiction, Ski, Swim, Tennis, Wine

(b)

P	Name	Friend	Rest	Hobby
SS1	Witold	Alexis	Bengal	Bike

Figure 4 Intended result of (Q2) with three LIST functions, to compare with the result in Figure 2, presented with text boxes (a) and with combo boxes for a screen (b)

Some SQL dialects, e.g., MsAccess, do not offer the DISTINCT predicate within an aggregate function. One way around today is to use the nested FROM clause. If LIST function should only reuse the current implementation of DISTINCT, the following query to MsAccess gives the (Q2) result[4], as in Figure 4:

```
SELECT P.[SS#], Name, Fr as Friend, Re as Rest, Hb as Hobby From Pers as P,
  (SELECT F.[SS#], List (F.Friend) AS Fr, Re, Hb  from F,
    (SELECT R.[SS#],  List (R.rest) AS Re, Hb  from R,
      (SELECT H.[SS#],  List (H.Hobby) AS Hb FROM H  where [ss#] = 'ss1' GROUP BY H.[SS#])
    where R.[ss#] = H.[ss#]  group by R.[SS#], Hb)
  where F.[ss#] = R.[ss#]  group by F.[SS#], Re, Hb)
```

[3] The output was simulated using the Min aggregate function instead of List in (Q2)
[4] The query was simulated using the Min aggregate function instead of List in (Q2), producing the output in Figure 4b. We recall that MsAccess SQL requires [ ] around attributes with spaces or special characters like #.

where P.[SS#] = F.[SS#] ;

Example 3

In above examples, one needed to list values of a single attribute only. This example motivates the multi-attribute LIST function.

a. A user wishes the ID and the total quantity of each part in the warehouse and a 2-d histogram with quantities per contributing supplier. One can satisfy the need as:

(Q3) **Select P#, SUM (Qty) as [TOTAL QTY], LIST (S#, Qty) as [Per supplier] from SP group by P#;**

The result of (Q3) is in Figure 5. Each element of each list is now constituted internally from two values. Each element is presented on a new line. However, as before, the whole list remains for the RDBS an **atomic** character string. In particular the use of LIKE clause remains legitimate. For instance, the following query would limit the output to parts supplied by 's4' among other suppliers, i.e., to lines 2,4,5 only in Figure 5:

(Q4) **Select P#, SUM (Qty) as [TOTAL QTY], LIST (S#, Qty) as [Per supplier] from SP group by P# having [Per supplier] like '*s4*';**

P#	Total Qty	Per supplier
p1	600	s1 300 s2 300
p2	1000	s1 200 s2 400 s3 200 s4 200
p3	400	s1 400
p4	500	s1 200 s4 300
p5	500	s1 100 s4 400
p6	100	s1 100

Figure 5 Intended result of (Q3) with the multi-attribute LIST function

b. Consider that S-P user wishes to see for each supplier S its data S (S#, SNAME, STATUS, CITY) and all its supplies. While most users of an RDBS are convinced that 1 NF is a great idea for the stored form of data, it is Polishinel's secret that most of them are also annoyed with the traditional 1 NF output of:

Select S.*, P#, Qty From S, SP where S.S# = SP.S# ;

The reason is that all supplier's data are uselessly repeated in each tuple of the supply, e.g., again, six time for S1. The LIST function responds to the need simply as follows:

(Q5) **Select S.*, List (P#, Qty) From S, SP**
where S.S# = SP.S#
group by S#, SNAME, STATUS, CITY;

The intended result is in Figure 6.

Observe interestingly in (Q5) that GROUP BY clause enumerates all the attributes of S. The enumeration of all but **S#** is in fact useless here as they are all functionally dependent on **S#**. Since the enumeration is a quite long list besides, it should be typically be annoying to the user. The constraint steams from the general property *P* that (i) in SQL at present any attribute

in SELECT clause that is not aggregated has to be a grouping one, and (ii) SQL does not accept at present '*' in the GROUP BY clause.

A clever use of LIST function may avoid the constraint. One needs to formulate the query so that every attribute *A*, single or composite, included '*', not aggregated by any other function, is declared as LIST (A) in SELECT clause. The query would respect the property *P* and it is no more necessary to declare *A* as the grouping attribute. The obvious reason is that in this case LIST (*A*) = *A*. For (Q5), the alternative would be as follows :

(Q6) **Select List (S.*), List (P#, Qty) From S, SP**
where S.S# = SP.S#
group by S#;

S#	SName	Status	City	p#	Qty
s1	Smith	200	London	P1	300
				p2	200
				p3	400
				p4	200
				p5	100
				p6	100
s2	Jones	10	Paris	p1	300
				p2	400
s3	Blake	30	Paris	p2	200
s4	Clark	20	London	p2	200
				p4	300
s4	Clark	20	London	p4	300
				p5	400

Figure 6 Intended output table of query (Q5)

c. We continue with the idea in (b), but switch to the context perhaps more common to the real life than the Supplier-Part database. We will also illustrate the use of the ORDER BY clause with the LIST function. Consider the following DB fragment similar to tables in PUB database provided with SQL Server:

Book (ISBN#, Title, Publisher, Year)
Author (ISBN#, Name, First Name, Rank)

The application needs to show 2003 books. We can respond to the need with :

(Q7) **Select B.*, List (First Name, Name) from Book B, Author A**
where B.ISBN = A.ISBN and Year = 2003
Group By ISBN, Title, Publisher, Year
Order by Title, Rank ;

The result should be one tuple per book listed with the list of the authors. Without our function, using the standard SQL, all the book attributes would be repeated with each author, definitively surprising any real-life user. The tuples produced by (Q7) should be in ASC order by title. Each list should also be in ASC order by the rank of the author. This intended behavior models that of MsAccess, in its (non-standard SQL) crosstab queries. Finally, like for (Q4.1), one may shorten the GROUP BY clause to ISBN only, by in turn aggregating B.* to LIST (B.*). Here the alternative does not have much importance. In real life however it might. A book scheme typically has many more attributes.

Example 4

We now illustrates the potential new capabilities of LIST function applying value expressions and scalar functions. We use only the possibilities that current SQL dialects provide already to the other aggregate functions, e.g., in MsAcces dialect, most used by (very) far. Consider again Supplier-Part database and the user who wishes for each part its total quantity on hand, and its distribution into quantities supplied by different suppliers, as above in Example 1. In addition the user wishes to know (i) the integer average quantity per supplier, and for each supply (ii) the fraction in % that it represents of the total quantity, and (iii) its positive or negative deviation from the average. Finally, the user wishes to order the result so that larger total quantities appear first, as well as larger contributing supplies.

The first formulation of this query one may think about in MsAccess SQL dialect could be:

(Q8)
```
SELECT SP.[p#], Sum(Qty) AS [total Qty], int(Avg(Qty)) AS [Avg Qty],
List (qty AS Distribution, Int(qty / [total Qty] * 100) AS [% of Total],
(qty - [Avg Qty]) AS [Deviation from Avg]
FROM SP
GROUP BY SP.[p#]
ORDER BY [total Qty] DESC, qty DESC;
```

Unfortunately, some popular dialects, e.g., MsAccess, and perhaps all at present, do not accept the reference to a dynamic attribute, e.g., **[total Qty],** in an aggregate function in the same Select list, nor in the Order By clause of the expression. The general way out is again the nested FROM clauses. This would lead in MsAccess SQL dialect to the following query[5]:

(Q9)
```
SELECT SP.[p#], Sum(Qty) AS [total Qty], int(Avg(Qty)) AS [Avg Qty],
LIST (qty AS Distribution, Int(qty / t1 * 100)) AS [% of Total],
(qty - t2) AS [Deviation from Avg])
FROM SP,
        (select sum(qty) as [t1], [p#] as p1, int(Avg(Qty)) AS t2 from sp group by [p#])
WHERE sp.[p#] = p1
GROUP BY SP.[p#]
ORDER BY Sum(SP.Qty) DESC, qty DESC;
```

The expected result, showing only the first line of each assumed combo box, would appear on MsAccess perhaps like in Figure 7. We do not know about any way to obtain a similar result using current SQL dialects.

2.2. Core Syntax and Semantics of LIST function

The motivating examples should make the intended syntax and semantics of the LIST function clear enough. If $\boldsymbol{A}$ is an attribute, perhaps composite, i.e., $\boldsymbol{A} = (A_1,\ldots,A_k)$, then LIST ($\boldsymbol{A}$) produces for each group $\boldsymbol{G}$ of m tuples, resulting from the GROUP BY and possibly HAVING clauses, a character string T formed by concatenating tuples t from the projection of $\boldsymbol{G}$ on $\boldsymbol{A}$, i.e.; $T = t_1 \,\&\ldots\&\, t_m$. The tuples may be ordered according to ORDER BY clause. The projection is the SQL one, i.e., is the k-d bag with the duplicates, and, perhaps, nulls. The RDBS may allow for the DISTINCT predicate in an aggregate function, as discussed for (Q2) above. One should be able to invoke then the LIST (DISTINCT ($\boldsymbol{A}$)), eliminating the duplicates.

Likewise, one should be able to invoke the popular TOP n predicate, limiting accordingly each T to at most the min (n, m) top concatenated tuples with respect to the ORDER BY clause[6]. The variant: TOP N percent should apply as well. One should also be able to invoke

[5] Simulated for List clause using Max aggregate function

[6] Unless, as usual, the tuples n, n+1… are duplicates with respect to the values of attributes invoked in ORDER BY.

the scalar functions and value expressions within LIST, as for the other aggregates accepted by the RDBS. The default separator between the concatenating values is ' '. In practice, a more elaborated syntax for LIST than used above could easily allow for the definition of other separators. For instance, following upon the related actual syntax of SQL Server and of MsAccess dialects, the expression:

LIST (A1 & ', ' & A2 & ', ' & A3 & ' ;'),

could mean that ', ' separates each t_1, t_2 and t_3 and that each list terminates with ' ;'.

p#	total Qty	Avg Qty	Distribution	% of Total	Deviation from Avg
p2	1000	250	400	40	150
p1	600	300	300	50	0
p5	500	250	400	80	150
p4	500	250	300	60	50
p3	400	400	400	100	0
p6	100	100	100	100	0

Figure 7 Expected result of (Q9) assumed displayed as combo boxes for the result of the LIST function.

The result of LIST of a single value, i.e., for $m = 1$, reduces simply to that value. The concatenation of a tuple with a null value within, keeps the null in *T*. Likewise, the concatenation should keep a null tuple, if the SQL dialect of the RDBS used has chosen to generally do it as well. By the same token, the currently used aggregate functions nest in a subquery in WHERE or FROM clauses. Hence LIST should as well. Finally, one should be able to refer to LIST in ROLLUP and CUBE clauses. We recall that these well-known clauses generalize, the GROUP BY in new dialects, [M99]. Again, the reason for this semantics is that the other aggregate functions are already in use in this way. We discuss more in the section below that it should thus be rather simple to reuse the capabilities existing in an RDBS for those functions for LIST as well.

We do not elaborate the formal definition of the LIST function grammar here. It does not seem necessary and would vary anyhow with the SQL dialect intended to support it.

3 Implicit LIST

Observe that in an SQL query at present, any attribute in SELECT clause should be either aggregated or a grouping one, referred to in GROUP BY. One can explore this property to enhance the SQL non-procedurality. The idea is to allow for non-aggregated and not grouping attributes referred to in the query, but to consider that some LIST implicitly aggregates any of them. More precisely, the following rule for the implicit LIST appears the most useful at present:

- Let ***A*** be an attribute, perhaps composite, grouping all the attributes from the same base table or view, referred to in SELECT clause and neither (explicitly) aggregated nor a grouping one. Then, any such *A* is considered as aggregated by the *implicit LIST* defined as LIST (DISTINCT *A*).

We call *implicit* any such LIST. The query where every implicit LIST is made explicit becomes conform to the present SQL syntax, hence acceptable to the RDBSs. The introduction of DISTINCT that may surprise at first glance, steams from the wish to apply the implicit LIST idea to (Q2). This application and similar ones, seem more practical than those of interest otherwise, i.e., if implicit LIST was defined so to preserve the duplicates. The idea

also means that the implicit LIST capability naturally targets in the first place an RDBS, accepting DISTINCT in an aggregate function.

One also needs some convention on the default attribute naming with respect to the result of an implicit LIST. Below, we consider that any atomic *A* simply keeps its name. The name generated for a composite A is a concatenation of the names of atomic attributes of *A*, with the space as separator. We also consider that other clauses that may syntactically refer to the attribute created by an implicit LIST, e.g., the HAVING clause, may still refer to the original attribute names within composite *A*.

To review our examples, observe first that the implicit LIST, nicely simplifies query (Q2) to more familiar:

(Q10) **Select P.SS#, Name, Friend, Rest, Hobby**
from P, F, R, H
where P.SS# = F.SS# and F.SS# = R.SS# and R.SS# = H.SS# and P.SS# ="ss1"
group by P.SS#, Name ;

In contrast, (Q 1) should remain the same. An implicit list would remove indeed the duplicates of **QTY**. This could lead to a different result, probably typically unintended.

Queries (Q3) and (Q4) would get respectively the familiar formulation, provided we do not care about the [Per supplier]:

(Q11) **Select P#, SUM (Qty) as [TOTAL QTY], S#, Qty from SP group by P#;**

(Q12) **Select P#, SUM (Qty) as [TOTAL QTY], S#, Qty from SP group by P# having S# like '*s4*'**;

Query (Q7) simplifies as well, in both the SELECT and GROUP BY clauses. But implicit LIST is of no use for (Q8) and (Q9), obviously.

To couple the implicit LIST with the use of implicit joins and of implicit FROM clause, further enhances the non-procedurality of SQL queries. We recall that major RDBSs offer the former capability, as we discuss more in Section 4 below. The implicit FROM is not yet in practical use, as far as we know. The basic idea is however well-known through the research on the universal relation interface. To apply this idea to our needs, we consider simply that FROM clause may contain an *implicit* table name *T* for any attribute *T.A* in the query that either (i) is uniquely qualified with its proper name *A*, or (ii) is referred to in an implicit or explicit inner equijoin clause in WHERE or FROM clause, or (iii) has already another attribute referred to in the query. In the latter cases, *T* can be any of tables with *A*. The result will remain unaffected.

With these capabilities available, our sample queries may become almost ideally non-procedural. Thus (Q2) and (Q10) lead to even simpler:

(Q13) **Select SS#, Name, Friend, Rest, Hobby where SS# ="ss1"**
group by SS# ;

Likewise (Q3) without **[Per supplier]** and (Q11) lead to :

(Q14) **Select P#, SUM (Qty) as [TOTAL QTY], S#, Qty group by P#;**

It may also be useful and quite non-procedural to apply both forms of LIST in the same query, e.g. the following one, expanding (Q 1):

(Q 15) **SELECT S#, SNAME, SUM (QTY) AS [TOTAL QTY], LIST (Qty) AS Histogram GROUP BY S#;**

And so on. The overall result is the conceptual separation between the high-level query formulation, and the actual decomposition of the relational schema to best avoids the design anomalies. *The latter can change without affecting the query formulation*. For instance, when a single valued property becomes a multivalued one. A popular case is that of users starting

having multiple phone numbers (mobile, home, work). Many similar often occurring needs are well-known. The end result is that not only the interactive user becomes happier, but one may also avoid the related nightmare changes to the application programs.

As the bemol, notice that the implicit LIST and the other bells and whistles discussed, help basically with the non-procedurality of typical queries. Even only this gain however is in the line with the fundamental goal of non-procedurality the relational data model [D2]. Notice also a new conceptually interesting role of GROUP BY. It was intended as a dynamic aggregator of tuples for a computation of some function. Here it servers also as a dynamic constructor of objects identified by the values of the grouping attribute(s) that becomes the OIDs.

The construction makes the relational model somehow naturally more object-oriented. A practical consequences is that the distinction between single-valued and multi-valued attributes necessary at the relational database schema level, becomes transparent for the user formulating the query. All this shows that introducing the LIST into RDBSs as fully as discussed here should reveal highly useful.

4 Implementation Issues and Related Work

The motivating examples have shown that the use of LIST function is intended to basically reuse the capabilities an RDBS already offers for other known aggregate functions. Hence, the implementation of LIST largely exists. Any SQL query processor creates the single-attribute lists for the GROUP BY based computation of, e.g., the SUM function, [GUW2], [YM98]. Usually, theses lists result from a two pass hash algorithm, e.g., the linear hash LKRHash algorithm, [LKR99], [L88], [L81], largely in use in MS products, including the SQL Server. The 1st pass creates in each bucket the list of all the selected tuples sharing the values of the grouping attribute(s). This is in fact an invisible core implementation of the LIST function already. The 2nd pass explores the list to compute the requested function(s). One has to enhance this processing with the list casting as a single character string, This should be a rather fast task for an experienced programmer [L3].

Another facet is the necessary extension to the SQL parser grammar to make it accepting the LIST verb with its implicit form. Although, the specific LIST function grammar depends on the SQL dialect used, both, its definition and efficient implementation seems also a rather routine task for a skilled folk.

Nonetheless, the "good" implementation of LIST function is an open research problem at present. The interface for the user-defined aggregates in an RDBS with this capability, e.g., Oracle 8i or 9i, or DB2 7.2, may perhaps help. There are proposals in the 'gray" on-line literature for the developers, for codes of user-defined aggregates that could be the basis for at least the simplest single attribute LIST, [T1], [T2], [B3][7]. See [L3] for more on this subject.

The analysis of the related work showed of course that existing RDBs do not offer the function offered yet, e.g., [MS], [S], [IBM], [O], [S]. It also showed one explicit user request, from Bonny Junior on Feb., 16, 2002, in DbForums [S]. We did not find any reply listed. We cannot say of course also whether our proposal really matches his question.

The RDBSS offer at present different tools, dealing less or more specifically with some but not all needs we have discussed. These are 4GL forms, and limited non-standard extensions to SQL. The latter are quite awkward to use with respect to LIST as proposed. See [L3] for deeper analysis.

[7] Located by Jim Gray (Microsoft Research, BARC)

As we mentioned already in Section 3, the LIST function becomes even more attractive if combined with the *implicit joins*, also called for some systems *auto* joins [LWS91]. We recall that these usually avoid the need to explicitly write some joins in the WHERE or FROM clauses. One avoids especially the equijoins (inner or outer) along the primary-foreign key structural constraints. The multirelational queries, e.g., (Q2), (Q5), (Q7) and the related ones with the implicit LIST, become substantially less procedural.

Implicit joins are now available in popular RDBSs. One can invoke the capability in DB2 and SQL Server through its visual interface to SQL. They are also credited for contributing to the mammoth popularity of MsAccess through its generalized QBE interface[8]. This one is mapped internally to SQL, although, as for SQL Server, one can invoke the SQL interface in MsAccess also directly. Notice incidentally that while not all SQL MsAccess capabilities are expressible in MsAccess QBE, the aggregate functions are. There should not be any major trouble thus to add LIST to this QBE as well.

Besides, the basic capabilities for the manipulation of lists were proposed for the object-oriented OQL language intended for an OODBS, [YM98]. Research proposals were consequently formulated for the object relational systems. These proposals concerned new data models, or substantial extensions to the relational model at least, with all interesting but heavy implications of any such approach. Among active outcomes, notice the popular AMOSII mediator system, [RJK3], supporting through its object-functional approach the vector data type intended for 1-d ordered bags. Notice also the *sorted* relations, and the related algebra, enhancing consequently SQL to the "*Sorted Relational Query Language*, *or SRQL*", [R&al98]. There is also a sequel to SRQL which is the concept of *arrables* in [LS3].

The bulk of this work will perhaps influence future dialects of SQL, may be steaming from SQL:1999 proposal, [M99], If so, it will affect the internals of an RDBS, with respect to the GROUP BY, ORDER BY, and selected new clauses proposed by these languages. It may then impact the use, the implementation, or the performance of the LIST aggregate function as well.

Finally, list manipulation capabilities should also characterize XML oriented systems, DBS especially, [LRK2]. Having the LIST function within RDBS should facilitate these goals as well. Again look into [L3] for more on all the mentioned issues.

5 Conclusion

The LIST aggregate function is simple and should be highly useful. It creates an integrated framework for queries to both aggregated and individual data values. These are harder to formulate or yet inexistent in an RDBS at present, although potentially highly useful for the popular data mining. The user may also naturally present and manipulate data normalized to 4 NF. These are particularly awkward to deal with in practice at present.

The implicit LIST should often simplify the query with respect to that with the explicit one only. It is further desirable to couple it with the implicit joins and the implicit FROM clause. The overall capabilities of LIST that result from alleviate long standing wishes of the relational database users.

We backed the semantics of the LIST function with the choice of the details, so to make the implementation of LIST function technically easy. The future work should focus on the experimental proof of this claim, by prototyping the implementation in the first place.

[8] See the shelves with the database books in the nearest tech. bookstore.

Acknowledgements

For the help with early ideas on the LIST function, we thank Ron Fagin for suggestions with respect to the motivating examples, Jim Gray for the example code for the SQL Server's ***Northwind DB,*** *and the related pointers to the gray literature, as well as Tore Risch for the comments on the related work. This work was partly supported by the research grants from Microsoft Research, and from the European Commission project ICONS project no. IST-2001-32429.*

References

[B3] Bowden, B. Increase code reuse with Oracle user-defined aggregate functions. Builder.Com. 2003. http://builder.com.com/5100-6388-1058914.html

[BB79] Beeri, C., Berstein, P. Computation Problems Related to the Design of Normal Form Relational Database Schemes. ACM Trans. On Database Systems, ACM-TODS 4(1), 30-59.

[D2] Date, C., J. An Introduction to Database Systems. Addison-Wesley, 2002.

[F77] Fagin, R. Multivalued Dependencies and a New Normal Form for Relational Databases. ACM Trans. On Database Systems, ACM-TODS 2(3), 262-278.

[GUW2] Garcia-Molina, H. Ullman, J., D., Widom, J. Database Systems: the Complete Book. Prentice Hall, 2002.

[K0] Kreines, D., C. Oracle SQL: The Essential Reference. O'Reilly, 2000.

[IBM] IBM Manual for DB2. ibm.com/software/data/db2/library.

[L88] Larson, P.-Å. Dynamic hash tables, *Communications of the ACM*, Vol. 31, No 4, 1988, 446–457.

[LKR99], Larson, P-Å., Krishnan M., and Reilly, V., G. LKRhash: Scaleable Hash Tables. Res. Rep., 1999 http://www.microsoft.com/

[LGG2] Litwin, P., Getz, K, Gilbert, M. Access 2000 Developpers Handbook. Volume 1 & 2. Sybex, 2000.

[LRK2] Lin, H., Risch, T. Katchanounov, T. Adaptive data mediation over XML data. Special Issue on *Web Information Systems Applications* of *Journal of Applied System Studies (JASS)*, Cambridge Intl. Science Publ., 3(2), 2002.

[L81] Litwin, W. Linear Hashing : a new tool for file and tables addressing. Reprint from VLDB-81. Readings in Databases. 2-nd ed. Morgan Kaufmann Publishers, Inc., 1994. Stonebraker , M.(Ed.).

[L3] Litwin, The LIST Aggregate Function for Relational Databases. CERIA Research Report 2003-06-09, 2003, http://ceria.dauphine.fr/.

[LS3] Lerner, A., Shasha, D. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. Intl. Conf. on Very Large Databases, VLDB 2003.

[LWS91] Litwin, W., Wiederhold, G., Suk Lee, B. Implicit Joins in the Structural Data Model. IEEE-COMPSAC, Kyoto, (Sep. 1991).

[M99] Melton, J. Advanced SQL:1999 Understanding Object-Relational and Other Advanced Features. Morgan Kaufmann (publ.), (Sep., 2002).

[MS] Microsoft SQL Server Home Page. http://www.microsoft.com/sql/

[RJK3] Risch, T. Josifovski, V., Katchaounov, T. Functional Data Integration in a Distributed Mediator System. In P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003

[O] Oracle SQL* Plus. http://technet.oracle.com/tech/sql_plus/content.html

[R&al98] Ramakrishnan, R. Donjerkovic, D., Ranganathan, A. Beyer, K., Krishnaprasad, M. SRQL: Sorted Relational Query Language. Intl. Conf. on Scientific and Statistical Database Management, SSDBM98.

[S] Sybase Transact-SQL User's Guide. http://manuals.sybase.com/onlinebooks/

[S3] Berkeley DB XML. Sleepycat Software. http://www.sleepycat.com/products/xml.shtml

[SLR94] Seshadri, P., Livny, M., and Ramakrishnan, R. Sequence query processing. ACM SIGMOD Conference on Management of Data, 1994, 430–441.

[T1] Tropashko, V. Program Your Own Aggregate Functions. Tip for Week of May 20, 2001. Oracle Publishing Document. http://www.oracle.com/oramag/code/tips2001/index.html?052001.html

[T2] Tropashko, V. Matrix Transposition in SQL. Dbazine.com, 2002. http://www.dbazine.com/tropashko2.html