Proceedings of the IASTED International Conference
DATABASES AND APPLICATIONS
February 17-19, 2004, Innsbruck, Austria

EXPLICIT AND IMPLICIT LIST AGGREGATE FUNCTION FOR RELATIONAL DATABASES

Witold Litwin

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

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 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 simple to implement. The relational systems already provide most of the capabilities it requires to the existing aggregate functions.

Keywords

Relational data, aggregate function, SQL, OLAP

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), [1], [2], [3], [4], [5], [6]. 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), [7].

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 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, [3]. Both options are beyond SQL. They require additional capabilities from the user and the RDBS does not optimise them, unlike an SQL query, [8].

```
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" ;
```

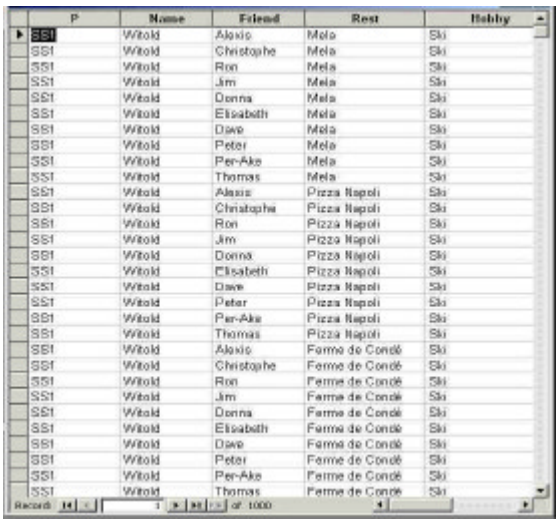

Figure 2 Result of MSAccess SQL query requesting the name, friends, preferred restaurants and hobbies of person identified with 'SS1'.

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 has several hobbies, friends, and preferred restaurants. The good relational database scheme would separate these data adequately in 4th NF tables, [9], [7], [8]. These 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, [9], [7], [8]. 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[1]. 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.

[1] 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.

We proposed the LIST function in [10]. In what follows, we argue further for it through an expansion of its capabilities. We start by recalling some motivating examples from [10], 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 presents the explicit LIST. Section 3 describes the implicit LIST, and motivates it through the analysis of the recalled examples. Section 4 discusses the implementation of LIST and the related work. We conclude in Section 5.

2 The LIST Aggregate Function

We first analyse the capabilities the function should provide through additional motivating examples. We then propose its basic syntax and semantics. We use the MsAccess SQL dialect as the basis. For the intended queries with LIST, *we consider only the capabilities it already provides to the existing aggregate functions.* We finally elaborate the capabilities 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:

(Q1) **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.

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. We welcome any hints.

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, [7]), 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[2], 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. If LIST function should only reuse the current implementation of DISTINCT, a way around today is to use the nested FROM clause, [10]:

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

[2] The output was simulated using the Min aggregate function instead of List in (Q2)

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 (Q4) would limit the output to parts supplied by 's4' among other suppliers, i.e., to lines 2,4,5 only in Figure 5:

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

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

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)

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 ***A*** is an attribute, perhaps composite, i.e., ***A*** = (A_1,…,A_k), then LIST (***A***) produces for each group ***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 ***G*** on ***A***, i.e.; $T = t_1$ &…& 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 (***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[3]. The variant: TOP *N* percent should apply as well. One should also be able to invoke the scalar functions and value expressions within LIST, as for the other aggregates accepted by the RDBS (see the motivating examples in [10], providing results impossible with any current SQL dialect). 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 ' ;'. 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, [11]. Again, the reason for this semantics is that the other aggregate functions are already in use in this way.

We do not elaborate the formal definition of the LIST function grammar here. It does not seem necessary and would vary 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

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

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:

(Q7) **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 ;

Here the attributes **Friend, Rest, Hobby** belong to each to a different table. Hence, each is under a separate implicit LIST. In contrast to (Q2), (Q1) 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]:

(Q8) **select P#, SUM (Qty) as [Total Qty], S#, Qty from SP group by P#;**

(Q9) **select P#, SUM (Qty) as [Total Qty], S#, Qty from SP group by P# having S# like '*s4***'*;

Here **S#,** and **Qty** belong to the same table. Hence they end up under the same implicit LIST and as a single column, like at Figure 5.

To couple the implicit LIST with the use of implicit joins and of implicit FROM clause, further enhances the non-procedurality of SQL queries. The *implicit joins*, [12]12], also called for some systems *auto* joins, usually avoid the explicit writing of some joins in the WHERE or FROM clauses. One avoids especially the equijoins (inner or outer) along the primary-foreign key structural constraints. Major RDBSs offer this capability. The multirelational queries with the implicit LIST, become substantially less procedural, e.g., (Q2) and (Q5).

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 the implicit joins and FROM capabilities, our sample queries may become almost ideally non-procedural. Thus (Q2) and (Q7) lead to their possibly simplest expression:

(Q10) **select SS#, Name, Friend, Rest, Hobby**
where SS# ="ss1"
group by SS# ;

Likewise (Q3) without **[Per supplier]** and (Q8) lead to :

(Q11) **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 (Q1):

(Q12) **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 avoid the design anomalies. *The latter can change without affecting the query formulation*. For instance, when a single valued property becomes a multivalued one. This gain is in the line with the fundamental goal of non-procedurality the relational data model [7] and makes the relational model somehow naturally more object-oriented. See [10] for more discussion.

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. Usually, theses lists result from a two pass hash algorithm, e.g., the linear hash LKRHash algorithm, [13], [14], [15], 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. See the example for SQL Server in [10].

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, [16], [17], [18][4]. See [10] for more on this subject.

The analysis of the related work showed further that major RDBs do not offer the function offered yet, e.g., [4], [6], [2], [5], [6]. The less known SQL Anywhere Studio 9 does offer the single attribute explicit LIST, [21]. We have also spotted one explicit user request for LIST in SQL Server on Feb., 16, 2002, in DbForums [10]. 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, e.g., the TRANSFORM and PIVOT clauses in MsAccess or COMPUTE in SQL Server or Sybase. These are quite awkward to use with respect to LIST as proposed. See [10] for deeper analysis.

Besides, the basic capabilities for the manipulation of lists were proposed for the object-oriented OQL language intended for an OODBS, e.g., for AMOS-II, [19]. Research proposals were consequently formulated for object relational systems. List manipulation capabilities should also characterize XML oriented systems, DBS especially, [19], [20]. We discuss these proposals extensively in [10]. Having the LIST function within RDBS should facilitate all these goals as well.

5 Conclusion

The LIST aggregate function is simple and should be useful. It creates a framework for queries to both aggregated and individual data values. These are hard 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 awkward to deal with in practice at present. As a simple solution to this problem, LIST function appears surprisingly overdue. By twenty five years or so with respect to the 4 NF invention, [9].

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.

[4] Located by Jim Gray

Acknowledgements

We thank Ron Fagin for suggestions to the motivating examples. We are grateful to Jim Gray for the help with the analysis of LIST implementation under SQL Server and the pointers to the "gray" literature. We thank Tore Risch for the comments to the related work. This work was partly supported by the grants from Microsoft Research, and EEC ICONS project, no. IST-2001-32429.

References

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

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

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

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

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

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

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

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

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

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

[11] Melton, J. Advanced SQL:1999 *Understanding Object-Relational and Other Advanced Features.* Morgan Kaufmann, 2002).

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

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

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

[15] 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.).

[16] 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

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

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

[19] 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.

[20] *Berkeley DB XML*. http://www.sleepycat.com/products/

[21] SQL Anywhere Studio 9.0.1. iAnywhere Solutions.