Can We Rely on SQL?

J.H. ter Bekke
Department of Information Systems
Delft University of Technology
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands
E-mail: j.h.terbekke@is.twi.tudelft.nl

Abstract

It is important for any data language that it enables many people to derive correct information from a databases in a simple, effective way with predictable performance. In an analysis is shown that SQL cannot fulfill these essential preconditions. It is caused by insufficient structural semantics of the relational model and the related operations in SQL. This follows from a comparison with results obtained from application of the semantic Xplain data language. The consequences of these shortcomings are an extreme performance degradation and a growing uncertainty among SQL users.

1. Introduction

The facility to express queries in a non-procedural way is an essential property of SQL. It gives the relational model its unique (and for modern applications indispensable) position in comparison with many other data models. In general, it is assumed that always logically predictable results are produced. In this respect SQL has certain shortcomings.

Suppose that a new vehicle is designed such that we need one hour for a certain distance, yet for another comparable distance more than a week. Even when the vehicle has a beautiful color, wonderful and comfortable chairs including a user friendly dashboard, the vehicle would remain unused because of the previous transporting properties. A time difference of a few hours (for example caused by traffic jam) would be acceptable under certain circumstances, but a time difference of this order (hour versus week) would not be acceptable.

In the database area there is a vehicle with foregoing properties that is accepted by everybody. It is almost superfluous to say that the standard data language for databases agrees with the foregoing description. In the database area this is accepted because a better alternative is unknown.

How is it possible that such a situation can continue? The most important reason seems to be the fact that the performance of computer hardware and software has increased more than one hundred times since the early eighties (the beginning of data languages in practice): if in the beginning a day was needed to solve a certain problem, now less than ten minutes are needed. Nobody feels any pain. There remain opportunities to tackle new more complex problems. We possess enough creativity to accept new challenges, so that is what we do. The time is ripe to start with problems like data mining and data warehousing. Then suddenly we discover that modern software does not perform so well. Database users must be warned constantly for the limits of the data language to avoid serious performance problems.

Even now, nothing would be wrong if computer capacities could grow again one hundred times. However this is not expected, so everybody tries to find a solution again in brute force. In the past hardware has solved so many problems, why not try again. So we try massive parallel hardware. Hundreds of processors in parallel promise an enormous improvement. If we succeed, we can continue on the same basis for another decade.

This paper is about a new elegant conceptual (not a brute force) approach for the previous problems. Many subjects are not covered in this paper. We do not pay attention to existing SQL-products which are close to each other and are based on the present database standard. These systems are often good implementations of this standard. We do not pay attention to specific software by which improvements can be achieved. In this area a lot has been achieved and there is no illusion now to add any contribution to this field. Finally hardware is out of reach. Leading companies have created the present high technical level, we have no illusion to contribute anything whatsoever. The only (but important) thing that remains is the introduction of a new revolutionary conceptual approach that creates extreme performance improvements.

This new conceptual approach is no longer only a range of ideas. Meanwhile we have also developed an extensive and reliable prototype of a software system (see [3] and [4]) that has been used in several practical circumstances. Although the realized software has been designed primarily for correctness and not for performance, the differences with existing systems are gigantic. To give an idea: in complex situations we might think of a factor of one hundred (or much more) faster: it seems that long duration queries have disappeared.

The new possibilities would be restricted if only highly specialized professionals could use the approach. Experiences in education [1] and in practice show that this is certainly not the case. Reliable results are now easily obtainable for a much larger group than in the past.

What is the essence of this new approach, why can it be so simple, fast, predictable and reliable? In fact, the kernel of this approach was the main goal many years ago: try to create a theory for databases based on one single concept similar to mathematics which is also based on one single concept: the set concept. In the seventies this was the goal for researchers who worked on the relational database theory. They based their theory on the set concept and they failed. They came up with a data model based on one single data structure.

If existing mathematical concepts cannot help us, then it seems that for databases a different view on reality is needed. We must try to create for this view our own conceptual framework. Lessons from the past from the application of the set concept must not be forgotten. Possibly we can develop a new, but in a certain sense analogous theory with similar properties.

It is well known that in an abstraction (which plays a role in both disciplines) we emphasize certain aspects and ignore others. Possibly we must ignore in databases things which are essential in mathematics and emphasize things that are completely irrelevant in mathematics. In that case a similar situation can arise. We do not ignore the existing, we add something similar to it for our own purpose.

This paper contains some simple problems which will be solved using SQL. Further analysis shows where problems occur and how these can be solved in practice. These adaptations require the level of an advanced user. Besides that, these adaptations lead to an extreme performance degradation of one hundred times (or even more than one thousand times). Later the same problems are solved using the semantic Xplain approach. These solutions require only a beginner's level. It is evident that using this approach the requirements regarding flexibility, performance and reliability are met in all respects.

2. A simple SQL database

Many examples can be given of problems which can occur in using SQL. This paper will analyze only one single aspect using a few simple examples. They are characteristic for a large class of database queries. Assume a simplified database with items and sales. The relational model for this consists of the following relations:

- **items**, for items are relevant: item identification, description, quantity in stock and price. These properties are stored in the ITEMS table.
- **sales**, for each sales transaction are relevant: sales identification, item number, date (week and day), number and amount. These properties are stored in the SALES table.

This leads to the following table definitions in SQL. Notice that domains and referential constraints are part of these definitions.

```
CREATE TABLE ITEMS (
ITEM#           CHAR (4) NOT NULL,
DESCRIPTION     CHAR (13) NOT NULL,
STOCK           NUMERIC (4) NOT NULL,
PRICE           NUMERIC (4,2) NOT NULL,
PRIMARY KEY (ITEM#) );
CREATE TABLE SALES (
SALE#           CHAR (4) NOT NULL,
WEEK            NUMERIC (2) NOT NULL,
DAY             CHAR (3) NOT NULL,
ITEM#           CHAR (4) NOT NULL,
NUMBER          NUMERIC (4) NOT NULL,
AMOUNT          NUMERIC (4,2) NOT NULL,
CHECK (DAY IN 'Mon','Tue','Wed','Thu','Fri','Sat'),
PRIMARY KEY (SALE#),
FOREIGN KEY (ITEM#) REFERENCES ITEMS );
```

We consider the following contents with sales data. These data enable us to verify query results. It seems superfluous, but it is essential to realize that for each database the closed world assumption holds. For the sales table it means that only sales are stored in this table, not the non-sales. Also for the items table holds: only existing items are stored in this table, not the non-items.

ITEM#	DESCRIPTION	STOCK	PRICE
I1	Table	20	234.00
I2	Chair	50	114.00
I3	Lamp	15	378.00

SALE#	WEEK	DAY	ITEM#	NO.	AMOUNT
S1	1	Mon	I2	1	114.00
S2	1	Tue	I1	2	468.00
S3	1	Wed	I3	1	378.00
S4	2	Mon	I1	1	234.00
S5	2	Sat	I2	4	456.00

We specify now two simple queries for which similar formulations can be found in any textbook on relational databases (see: [5] page 155 ff., [6] page 416 ff., [7] page 234 ff., [8] page 217 ff., [9] page 207 ff.]).

S1: Determine the turnover per item.

```
SELECT ITEM#, TURNOVER = SUM (AMOUNT)
FROM SALES
GROUP BY ITEM#;
```

ITEM#	TURNOVER
I1	468.00
I2	702.00
I3	378.00

S2: Determine items for which the turnover decreased in week 2 compared with week 1.

```
SELECT I.ITEM#, DESCRIPTION
FROM ITEMS I, SALES S2
WHERE I.ITEM# = S2.ITEM#
AND S2.WEEK = 2
GROUP BY I.ITEM#, DESCRIPTION
HAVING SUM(S2.AMOUNT) <
        (SELECT SUM(S1.AMOUNT)
        FROM SALES S1
        WHERE S1.ITEM# = I.ITEM#
        AND S1.WEEK = 1);
```

ITEM#	DESCRIPTION
I1	Table

Well-trained SQL programmers know that we must start with a calculation of turnovers in one week. In this expression an inner block is needed to compare sales of week 2 with sales of the same item in week 1 (that is the reason of a join between sales from the second SELECT with items from the first SELECT).

The result consists of only item I1. This is incorrect. Item I3 has been sold in week 1 and not in week 2 and must therefore be included in the result. The first SELECT however selects only items sold in week 2 (ITEM#s I1 and I2). For these items only sales in week 1 are compared (ITEM# I1 has lower sales). In the result ITEM# I3 is missing, because it was not sold in week 2. This problem can be solved by adding items sold in week 1 and not sold in week 2 as follows:

```
SELECT I.ITEM#, DESCRIPTION
FROM ITEMS I, SALES S2
WHERE I.ITEM# = S2.ITEM#
AND S2.WEEK = 2
GROUP BY I.ITEM#, DESCRIPTION
HAVING SUM(S2.AMOUNT) <
        (SELECT SUM(S1.AMOUNT)
        FROM SALES S1
        WHERE S1.ITEM# = I.ITEM#
        AND S1.WEEK = 1);
UNION
SELECT I.ITEM#, DESCRIPTION
FROM ITEMS I, SALES S
WHERE I.ITEM# = S.ITEM#
AND WEEK = 1
AND I.ITEM# NOT IN
        (SELECT ITEM#
        FROM SALES
        WHERE WEEK = 2);
```

Characteristic for this solution is the fact that the addition is not generic; the repair is dependent on the required relationship (< or >) and the order in which week 1 and week 2 occur in the solution. Now that this is a known fact, let us now also inspect the other query.

Query S1 is conceptually wrong but the query produces accidentally the correct result. However, unsold items are not selected. The following addition is therefore required:

```
SELECT ITEM#, TURNOVER = SUM (AMOUNT)
FROM SALES
GROUP BY ITEM#
UNION
SELECT ITEM#, TURNOVER = 0
FROM ITEMS
WHERE ITEM# NOT IN
        (SELECT ITEM#
        FROM SALES);
```

It is obvious to say that these examples have shown that SQL statements are valid for only one single problem and cannot be used for a class of similar problems.

These problems demonstrated an essential problem in SQL. In SQL a set function (such as SUM) must always be used together with one single (eventually derived) table. No use is made of the relationships in the database. Semantics as expressed in the database by means of PRIMARY KEY and FOREIGN KEY specifications are completely missing.

Additions to queries, as shown before, result in a gigantic performance problem. Even if the original wrong formulation can be executed in a reasonable time, the extended query requires a multitude of this time. In a large database this can be easily one hundred times or even much more. Presumably this is mainly caused by building and scanning temporary tables over and over again.

3. Semantics in databases

A set is completely defined by a certain collection of elements into a unit. This **set-element** membership relationship is fundamental (it occurs in the mathematical axioms) and plays a role in all set operations.

In computer science set elements are often transient. For example, when a database contains data about clients, individual occurrences of clients can be considered as a coincident (they may come and go without changing the type client). So, types occurring in a database (such as client) can impossibly be defined by these occurrences. This implies that we cannot base the theory on the set concept. The concept of a relation from relational databases is therefore inapplicable in the database area. Individuals do not play any role in the definitions, they are also completely irrelevant in operations on types. We ignore them here completely.

The **properties** of type instances play an essential role in databases. In database terminology: attributes define a type; attributes make the meaning of a type clear. This is expressed in the concept of aggregation: a collection into a type of certain attributes. So in databases the **type-attribute** membership relationship is fundamental, this relationship defines the semantics of a type. This is why we arrive at two analogous concepts. In mathematics an element can appear somewhere else as a set, in databases an attribute can appear somewhere else as a type. In mathematics we have also operations on sets: intersection and union. These operations can also be defined in the database area; we call them generalization and specialization.

If an attribute is somewhere else considered as a type, then it means that the attribute is related to that type. This property does not have to be defined separately; it is indissolubly connected with these definitions. The same is true for the mathematical equivalents. Such a property is called inherent.

Besides structural definitions also certain language constructs by which information can be derived from a database are needed. It implies usage of the inherent relationships, in other words semantics is essential. This is illustrated with the following simple examples.

4. The semantic alternative

According to the semantic Xplain model [2] the sales example consists of two interrelated types. The referential property is expressed by the used type names and attribute names. Besides that, base type day is defined by means of an enumeration. Only the types day, item and sale have a corresponding data structure. The complete definition of the previous sales example is as follows:

base day (A3) ("Mon","Tue","Wed","Thu","Fri","Sat").
base week (I2).
base amount (R4,2).
base description (A13).
base stock (I4).
base price (R4,2).
base number (I4).
type item (A4) = description, stock, price.
type sale (A4) = week, day, item, number, amount.

We consider now exactly the same database contents with sales data. In this case we don't see any difference on the implementation level between the relational database and the semantic database of Xplain. The only difference is its interpretation.

item	description	stock	price
i1	table	20	234.00
i2	chair	50	114.00
i3	lamp	15	378.00

sale	week	day	item	number	amount
s1	1	Mon	i2	1	114.00
s2	1	Thu	i1	1	234.00
s3	1	Wed	i3	1	378.00
s4	2	Mon	i1	1	234.00
s5	2	Sat	i2	4	456.00

X1: Determine the turnover per item.

This query requires the two related types. Determination of the total quantity is formulated by using this relationship in the extend-statement. The extend-statement results in a derived attribute value for each stored item (note that the domain of type item is given in only one place, indicating the type we need for this extend). The next step is to select the derived information using the get-statement. This results in the following:

extend item *with* turnover =
    total sale *its* amount
    per item.
get item *its* turnover.

item	turnover
i1	468.00
i2	702.00
i3	378.00

X2: Determine items for which the turnover decreased in week 2 compared with week 1.

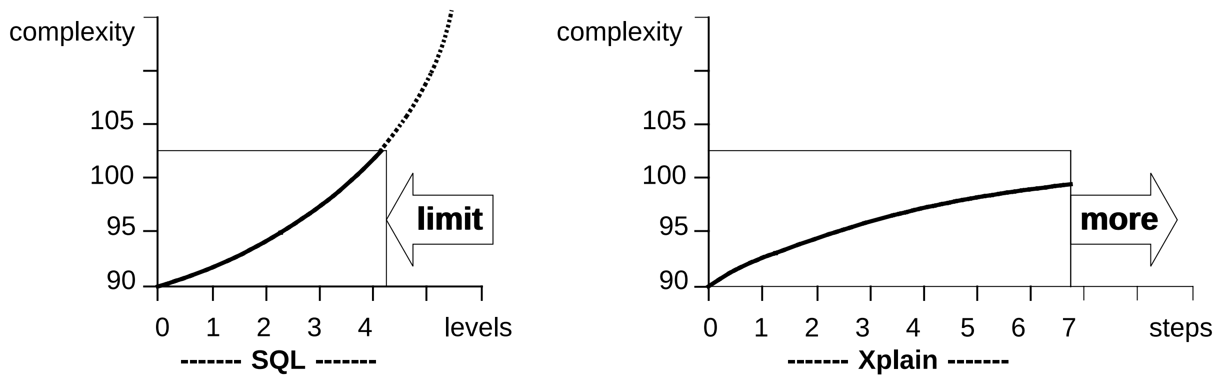

Figure 1: Query complexity in SQL and Xplain.

```
*extend* item *with* turnover1 =
    *total* sale *its* amount
    *where* week = 1
    *per* item.
*extend* item *with* turnover2 =
    *total* sale *its* amount
    *where* week = 2
    *per* item.
*get* item *its* description
    *where* turnover1 > turnover2.
```

item	description
i1 i3	table lamp

All Xplain solutions are without anomalies. Experiences show that execution time for these solutions is always linear. This is significantly less than in case of SQL. Tests in practice have shown ratios of 100:1 or more. In stead of waiting a few seconds in Xplain one must wait several minutes in SQL. This is no problem if one has the time to wait: it is unacceptable when the information has lost its relevance in that period. Xplain-solutions allow query adaptation to solve similar problems and interactive database use. In SQL both are not allowed.

The above experiences have created a difficult situation. If one has to wait for a long period to get an answer but a better alternative is unknown, then one can leave it at that. However when it is known that the waiting period is completely unnecessary, even stronger: the process can be executed more than one hundred times faster without complex repairs or programming tricks, one gets frustrated and uncertain. The only thing that's needed is another interpretation and the recognition of certain quantities (such as turnover) which can also be used in other problems.

5. Performance analysis

First remarkable difference between the approaches is that derivation of information in the semantic model always uses the semantics as expressed in the definition of the data model. The structure of the extend-statement denotes that for each instance of a type certain information must be derived from another type. In SQL a set function (such as SUM) must always be applied on one and only one (eventually derived) table. This implies that the attribute plays the role of domain. It leads to the mentioned conceptual problems. Repairs that result from this have two main drawbacks:

- For each non-trivial query the users must be aware of a sudden occurrence of this phenomenon. User can become uncertain. It is in a way comparable with the situation occurring with divide-by-zero in the application of mathematics. In computer programming sometimes such a mistake occurs. The difference with the divide-by-zero problem is that the computer system will often give a warning in case of a divide error (or worse: a core dump) or that an occurrence of the problem can be derived from the result. In the SQL problem there is no warning and in the result this is not visible.
- The necessary repairs lead often to queries which are very complex and hard to maintain. This has enormous consequences for the costs for maintenance. Besides that, repairs lead to a dramatic performance degradation. Tests in practice with real problems from the area of executive information systems have shown inexecutable queries with an execution time of more than one week on a modern SUN Sparc Station with 96 Mb internal memory. Another big difference is the formulation in one complex step versus a formulation in a few simple steps. This results in different complexities. This can be illustrated with figure 1.

execution time SQL	execution time Xplain	table sizes in number of rows	computer system
126 sec.	32 sec.	26 and 10 000	486DX PC
30 310 sec.	115 sec.	25 000 and 10 000	486DX PC
> 1 week (> 600 000 sec.)	410 sec.	40 000 and 500 000	SUN Sparc Station + 96 Mb intern

Table 1: Execution times with SQL and Xplain

In SQL one is often inclined to formulate one single statement for a problem. In Xplain one needs in general only a few simple statements. This difference has two important consequences:

- The Xplain formulation is unique for a certain problem and can therefore easily be extended to the solution of a more complex problem. On the other hand, in SQL one encounters very soon the limits of what humans can comprehend. For example, experiences from programming languages have learned that the number of abstraction levels humans can understand at once is limited to about four levels. Above this number one must split up the problem into smaller problems. At that moment a standard splitting procedure is missing and the advantages of an optimizer are lost.
- The complex SQL formulation results in an execution time of higher order. On the contrary, in Xplain all steps lead to maximal linear order. This implies that for large databases SQL often leads to the maximal processing capacity, while this limit is never reached in Xplain. This conclusion can also be drawn from several practical test cases. Without further explanation three results from the area of executive information systems are given in table 1. The times needed to produce the wrong results are obviously not given.

Conclusion

It is too bad that developments of SQL where solely driven by the goal to create a data language for untrained users. Fundamental properties are therefore pushed too much into the back. This has resulted in a language that due to its bad habits is not very useful for humans (both untrained and professional users). Professional users would appreciate a simple language with fundamental properties which enable them to derive reliable information from a database. The semantic data language Xplain fulfills this. Remarkable is that none of the given semantic formulations needed any repair.

Another essential problem is the following. Often users of other data modeling techniques assume that a translation into a relational data model is enough. The applications can then safely be implemented in a relational DBMS. In other words: implementation is of no concern, these problems have been solved by others. This paper has shown that this is a wrong conclusion. The foregoing has shown that those who follow other data modeling approaches cannot shelter behind relational implementations; in that case a sound foundation is lacking and systems are built on quicksand. Everybody has to indicate how a reliable implementation can be achieved. The Xplain DBMS has shown how such a situation can be achieved.

The problems in this paper may look trivial at first instance. Experiences has shown that these situations often occur unexpectedly in complex situations. The given critics are therefore more fundamental: it is almost impossible to realize complex systems using a data language with anomalies.

Acknowledgments

I thank my colleagues Dolf van der Ende and Bert Bakker for discussions on SQL pitfalls. I thank Ranjan Tewarie for his efforts in performance analyses between relational and semantic databases in the area of executive information systems.

References

[1] J.H. ter Bekke, *Comparative study of four data modeling approaches*, Proc. 2nd EMMSAD, Barcelona (1997).

[2] J.H. ter Bekke, *Semantic data modeling*, Prentice Hall, Hemel Hempstead UK (1992).

[3] J.H. ter Bekke, *Complex values in databases*, Proceedings DKSME 1994, Hong Kong (1994), pp. 449-455.

[4] J.H. ter Bekke, *Meta modeling for end user computing*, Proc. DEXA Workshop 95, London (1995), pp. 267-273.

[5] C.J. Date, *An introduction to database systems*, Addison Wesley, Reading Mass. (1990).

[6] T. Connolly, C. Begg and A. Strachan, *Database systems: a practical approach to design, implementation and management*, Addison-Wesley, Wokingham UK (1995).

[7] B.C. Desai, *An introduction to database systems*, West Publ. Company, St. Paul MN (1990).

[8] J.D. Ullman, *Database and knowledge base systems Vol. 1*, Computer Science Press, Rockville MD (1988).

[9] R. Elmasri and S.B. Navathe, *Fundamentals of database systems*, The Benjamin/Cummings Publ. Company, Redwood City CA (1994).