Semantic requirements for databases in casual environments

J.H. ter Bekke
Faculty of Information Technology and Systems
Delft University of Technology
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands
Telephone: +31 15 2784402
Fax: +31 15 2786632
Email: j.h.terbekke@its.tudelft.nl

Abstract

Database access in a casual environment has properties of an unstable environment; it is open, unrestricted, dynamic, interactive and unpredictable. Present database modeling approaches do not fulfil all requirements for access in this environment because they demand hindering restrictions and substantial professional expertise in practice. In existing organizations these preconditions can be met. However, in casual environments many charming advantages of databases are then lost. This paper presents some simple semantic requirements and solutions for databases in casual environments. They lead to self-supporting database systems with reliable and efficient database processing. The kernel of the solutions lies in the application of extremely simple data semantics. The resulting solutions are also useful in present professional database environments.

Keywords: Internet databases, semantic data modeling, active database systems, extensible databases, metadata, schema evolution, end-user computing.

1. Introduction

Database access in casual environments such as the Internet can be characterized as dynamic and unpredictable. This environment is unprecedented and can in some respect be considered as chaotic: things that are relevant today are possibly irrelevant tomorrow. How can databases be constructed to anticipate such a chaotic environment?

The first solution for this chaotic environment could be restricting database access to some predefined applications. This implies that several potential advantages of databases will not be exploited, which means the end of further database developments in these new environments.

Another acceptable solution abandoning necessary restrictions of the past is in favour of new techniques. This is a direction also indicated in [4, 11]. The final solution should be extremely simple and intelligent software is needed which maintains itself instead of depending on periodical interrupts for structural reorganizations. The software system should be reliable, guarantee good performance (i.e. independent of an optimization program conform recommendations in [1]), support users and not allow users to block database access for other users by lengthy processing.

The proposed solution is considered as an alternative to SQL based solutions [3] and can be compared with a solution found for semi structured data. A certain inherent structure is required for both user and system. In [5] a method is proposed to derive this structure from large collections of unstructured data, resulting in a collection of hierarchically organized types. The solution in the present paper results in a similar collection using aggregation and generalization abstractions. This inherent structure is almost automatically defined without complex concepts. Since structure is under the system's control, it is possible to assist users with structural changes and query formulation.

Attention is paid to some general database aspects arising from database access in casual environments. Often these aspects are also useful under other circumstances. The most relevant property is complete freedom for all database users. No user requirement is proposed regarding definition and manipulation: nothing must be done, everything can be done as long as the integrity of the database is not endangered.

For illustration purposes one might think of an environment of scientific research where researchers develop a common database for their research results. Each researcher has database access, which can result in addition of new subject classes (i.e. extension/adaptation of the data model), addition of new results into the database (i.e. data manipulation) and derivation of information (i.e. formulation of complex queries). Researchers may use database structures, contents and derivations created by others.

No attention is paid to technical aspects of caching, communication, multi databases, data distribution, security, indexing, browsers and operating systems, which can also play an important role in this database environment. Attention is paid to some essential characteristics for data modeling and data manipulation, resulting in some fundamental system characteristics. This paper contains some examples and solutions using relational [2] and semantic data modeling concepts [6, 7].

2. Data modeling requirements

An important aspect of a casual environment is its interactive character. Clear-cut applications do not exist. This aspect has enormous consequences on data modeling concepts for such an environment. It requires that each consistent state of a data model (consisting of structures and functions) must be considered as a situation allowing further extensions at a later stage. To allow this, it is relevant to consider data modeling concepts like conceptual redundancy, conceptual ambiguity, view independence and orthogonality.

When redundancy is discussed, often operational (i.e. data) redundancy is emphasized. However, the concept of redundancy has a much broader meaning. Suppose, for example, the case of relational tables according SQL having relations and attributes. Each attribute contains the definition of its domain in each table. If several tables contain the same comparable attribute, then in SQL it leads to a situation in which several times the same domain is given for the attribute. So a form of conceptual redundancy occurs.

It is important that the domain of a type is specified only once, namely in the table where it occurs for the first time, i.e. in the table where it occurs as identification. This has nice but radical consequences for its use, as will be demonstrated later. It leads to a situation agreeing

with a semantic data model (notice that below the domains of base types like 'number' are specified once):

type description (*char* 25).
type number (*int* 4).
type day (*date*).
type item (*char* 4) = description, stock_number.
type sale (*char* 4) = item, number, day.

A graphical representation of this model is given in figure 1. No distinction is made between base types (without attributes) and composite types (with attributes): both types are indicated by rectangles. This approach gives us the required freedom in casual environments; it allows us to consider an existing base type in a later stage as being composite. This fits perfectly in a casual environment because there existing applications do not require any modification.

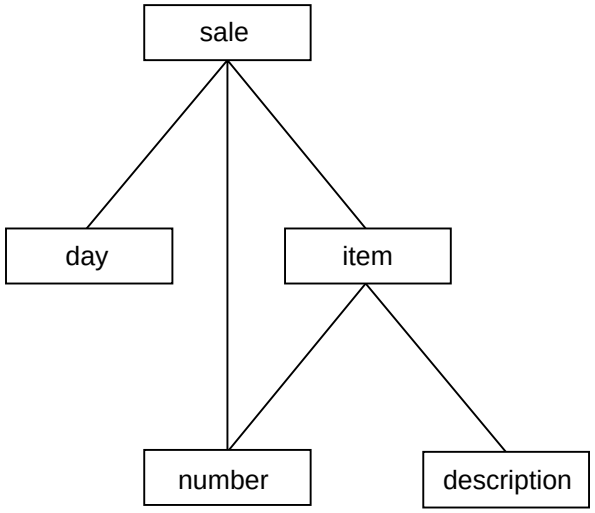

Figure 1: Aggregation

A second form of conceptual redundancy can be avoided by applying the generalization abstraction. The generalization of two or more types is defined by the intersection of attributes (i.e. the common attributes) of the participating types. So by definition each specialization contains all attributes of its generalization and therefore a reference to the corresponding generalization (below: *is item* between square brackets) plus the attributes not occurring in the intersection of attributes. This approach facilitates schema evolution. An example is the following overlapping extension to the earlier data model. For convenience the underlying base types and domains have been omitted:

type item = description, stock_number.
type electronic item = [item], voltage, current.
type mechanical item = [item], fuel, power.

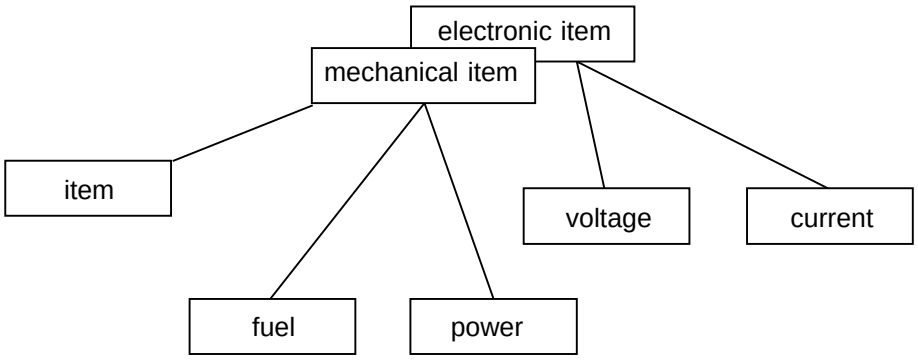

Figure 2: Generalization

Figure 2 contains the graphic representation of generalization abstractions. The semantic concepts allow us to extend the earlier given aggregation hierarchy with this generalization hierarchy. It implies that type item can be considered as: aggregation, attribute, type, generalization, etc. without adapting existing structures and applications already defined in the earlier aggregation part.

Although application of the generalization abstraction does not lead to elimination of operational redundancy, it has another advantage: NULL values are eliminated. Conceptual redundancy is also eliminated. Instead of including attribute 'description' in the type definitions of both the electronic item and the mechanical item, it is now by application of the generalization abstraction only included in the definition of type item. The attribute description can be obtained from the electronic item or the mechanical item by means of a special language construct: item *its* description. This meaningful language construct is always used for both aggregation and generalization relationships instead of the relational join for the relationship, thereby preventing misinterpretation and lengthy queries [8].

Advantages of databases without operational redundancy have been described extensively in literature. Conceptual redundancy was hardly discussed. Yet this form of redundancy has similar impact because all data of a data model are stored in a data dictionary. A data dictionary is a database and must therefore conform to requirements put on databases. The absence of redundancy in the data dictionary makes it easier to keep this database consistent. This is exemplified in [10]. Dependent aspects cannot be forgotten and there are no anomalies when this database is used. This improves the integrity of the complete database system.

In a perfect modeling environment there should be no limitation to the number of interpretations of a certain concept. This property is denoted by view independence. When language allows a certain interpretation, then this interpretation should not be prevented by the modeling approach. It is obvious that this is also relevant during database access. Each user may have his own view on the data in the database; each user may determine what part of the database is relevant. One user might require details of a concept, while another user might not want to consider these. The last user considers the concept as indivisible, while the other one considers the same as an aggregation consisting of a number of components. This property should not be limited in any way and no interpretation should be given preference. View independence is not limited to these two extremes, but can be extended to other interpretations, as: instance,

type, attribute, relationship, generalization, specialization, aggregation, constraint, function, etc. This flexibility implies that a database consists not only of operational data considered as attributes values; it can also contain functions, constraints etc.

View independence does not only influence the way operational data in a database can be used. It has also substantial influence on the possibilities for data definition and schema evolution. New interpretations arise from new relationships. This is an important aspect in database design. A data model for a complex application area is never developed in exactly one mental step. Parts of the model are developed and integrated in a later stage. For integration, none of the existing parts should require modification. On the contrary, integration should be realized solely by extension of existing interpretations with new ones as a consequence of new relationships between the parts.

An advantage of this new approach is that the final solution does not contain any information about the development trajectory of the designers. This makes it easier to use the final model: one may find one's own way and use one's own interpretations. Users are not dependent on information about the development trajectory. An illustration of such a trajectory is the development of a data model for version management [9].

Finally, there is an aspect that also plays a role in other disciplines. It has to do with a better recognition of similar problems. The combination of similar problems together with the same constructs will lead to recognition at the user's side. In such a situation not all problems have to be solved completely. Then one can re-use previous solutions for similar problems. An essential precondition for this is that the language in which solutions are expressed consists of orthogonal constructs without anomalies. Conceptual redundancy is not desired. The relevance of this aspect is not limited to data modeling. It plays also an important role in data manipulation and queries, as will be illustrated in the next section.

3. Query requirements

The interactive character of database access in casual environments makes special demands on available query facilities. In this respect we consider extensibility, orthogonality and consistency. These aspects are covered in this section using some well-known examples from publications on relational database systems.

By query extensibility is meant the facility to extend a given query to a more complex query without adapting the existing derivation and without any performance degradation in the existing derivation. This can be illustrated using well-known examples from relational database systems as expressed in SQL.

SQL offers solutions for individual problems, not solutions for classes of similar problems. For example take the supplier-item-supply data model from literature on relational databases. The following problems result (be aware of pitfalls [8]) in completely different statements:

- Give names of suppliers supplying no item.
- Give names of suppliers supplying at least one item.
- Give names of suppliers supplying less than ten items.
- Give names of suppliers supplying more than ten items.
- Give names of suppliers supplying all items.

Some of these queries could occur in one single interactive user session. Suppose that a user queries a database. The result of a query can initiate another more specific query. In practice this happens usually when the result is not informative enough (for example when the query results in an empty or nearly empty result).

Query extensibility makes it possible to compose queries incremental from other queries, in which it is irrelevant to know whether information is derived from operational data or whether the information is already available in the database. To be more specific, we give below the semantic data model for a supplier-item-supply database:

type supplier = name, city.
type item = description, stock_number.
type supply = supplier, item, day, quantity.

This structure is represented in figure 3. The new definitions could be added to definitions in the parts given in figure 1 and figure 2 without any modification.

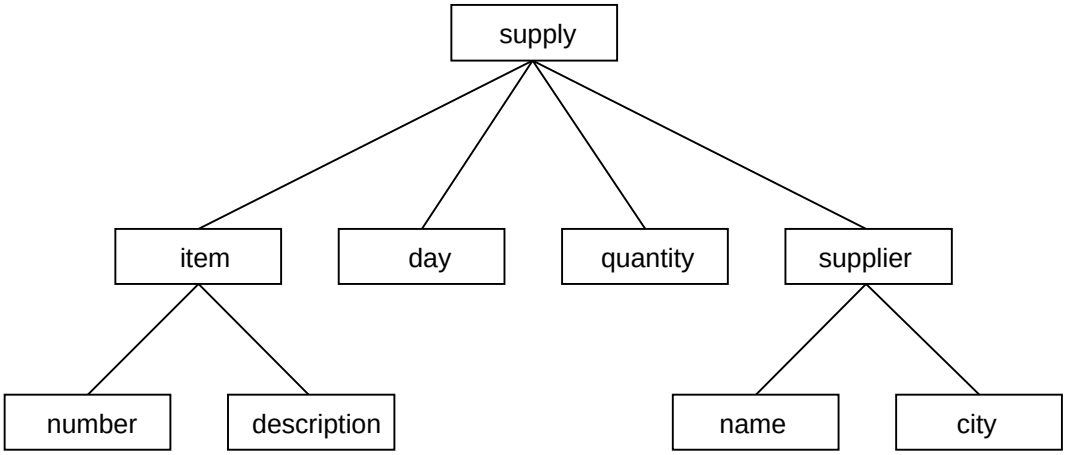

Figure 3: Supply aggregation

The above problems would benefit from a derived attribute *'number of items per supplier'* in the type definition of supplier. Then the first problem could be formulated simply by:

get supplier *where* number = 0.

Because this attribute is not included in the data model definition, the meaning of the extension *number* is defined for this purpose as follows:

extend supplier *with* number = *count* supply *its* item *per* supplier.

This definition illustrates another form of view independence: the attribute 'supplier *its* number' can be considered as a function. Notice that this solution differs completely from the facility to create views in relational databases. Views are not generic and result in data separated from and independent of operational data in the database.

The foregoing extension definition is beneficial for previous queries. They can be formulated with one simple *get*-command, with exception of the last problem. It is obvious to add the definition of the number of items for this query, as follows:

value variety = *count* item.

Now it is allowed to use one single command in which numbers (not the sets themselves!) are compared:

get supplier *where* number = variety.

Extensions must be considered as unambiguous semantic definitions of derived quantities, which are an intrinsic part of the database. The defined attribute plays a role as any other attribute. Suppose that suppliers with the minimum number of parts must be selected. The following semantic definition would help to do the job:

value minimum = *min* supplier *its* number.
get supplier *where* number = minimum.

In a perfect situation all extensions and values would have been defined beforehand. Then queries can be formulated using one single *get*-command. Because queries are uniform they can easily be adapted during a user session. Below is an example to illustrate this:

value limit = *input* (*int* 4).
get supplier *where* number < limit.

This query enables users to adapt the selection criterion during query execution. In case the result of an executed query is not satisfactory, the latter part can be executed using another parameter value. Earlier extensions are not derived again.

A second important aspect for querying is query orthogonality. To achieve this we must base concepts for data manipulation on the concepts for data definition. Foregoing examples already indicated that this is the case. Application of semantic concepts in a certain environment results in a unique data model. For semantic data manipulation only types and relationships of the data model can be used; arbitrary relationships cannot be enforced. Fixed constructs are used for these relationships. This results in a unique query solution for a certain problem.

Query orthogonality has many advantages. As shown above, it results in query extensibility. It offers also prospects of a good performance for both the user and the software system. Since problems have unique solutions in which the processing order is completely data driven, query optimizers have no effect. This is so because an optimizer is used (for example in relational databases) to compare calculated processing durations of different alternative solutions and to select from these the best alternative for execution. With one alternative the choice is already known. The software must give a good performance for this unique solution. In this case it is irrelevant whether the query is executed once or several times. This freedom fits perfectly in a

casual environment because here we do not know what is/becomes relevant and what is/becomes large.

Because only types and relationships from the data model can be used, the possibility that unintentional anomalies occur, is excluded. By query consistency is meant a query language in which misinterpretations at the user side as a consequence of ambiguous/incomplete semantics are excluded. For example, when a query formulation requires a certain type, the semantic data model makes clear what is meant because the domain of a type is defined only once. This place is needed in the query formulation. So the previous *'number of items per supplier'* must be considered as an extension of type supplier. This attribute does not belong to something that has no relationship with the database.

4. System requirements

In casual environments it is unknown who the users are and what they want. That is why such an environment would benefit from self-regulating declarative systems in which no place is needed for programming. Foregoing sections have shown that semantic concepts can give a solution for database access in these environments. That is why this section contains some system requirements which could arise from the semantic data modeling concepts.

Dynamic environments benefit from a software system in which self-regulating data structures play a dominant role. These structures must give a good and predictable performance under all circumstances. In this respect a number of aspects can be distinguished: an active data dictionary, access methods, transposed files and type-oriented operations.

A database with extensible structures and queries cannot exist without an active data dictionary. Because structural updates are elementary and without redundancy, these operations can be executed on an interactive basis (i.e. on the fly). This implies that an extension (for both definition and manipulation) can be used immediately after it has been processed.

With data structures it is evident that the concepts for data definition and manipulation play a dominant role. Indispensable are the inherent properties of type interrelationships. B+tree organizations are in this respect evident because of their self-organizing character and their possibilities for direct access.

Because extensions play an important role in semantic data models, it is obvious that not a record-oriented file structure must be chosen but a transposed file structure. This choice offers optimal flexibility and for databases on secondary storage medium a remarkable reduction in processing time. Only necessary parts of a database are then used. This technique is particularly suitable in an environment which is dynamic or even tends to become chaotic.

In a transposed file structure a user is not dependent of other users. Together with the earlier query facilities, only absolutely necessary parts of the database are involved in query processing. This also implies that the executions of query extensions are not influenced by requests of other users. This remains valid even when other users add new parts to the database structure.

A high performance can be guaranteed when all users use type-oriented operations and never use record-at-a-time programming statements. Important for a high performance is that only the software system determines the processing order. Blockades by mistakes of other users are then excluded.

The performance of database systems is also dependent on the concepts for data manipulation. It is a well-known fact that multitable SQL solutions often result in exponential performance characteristics. This is not allowed in our environment. One user could use all system resources and hence blockade database access for all other users.

High performance was a motive in the development of the language constructs for data manipulation. The resulting linear time/space complexity can be demonstrated with the following global syntax of the used query language constructs:

- *get* <maintype> *its* <simple expression> [*where* <simple condition>].
- *value* <variable> =
  <function> <maintype> [*its* <simple expression>] [*where* <simple condition>].
- *extend* <subtype> *with* <extension> =
  <function> <maintype> [*its* <simple expression>] [*where* <simple condition>]
  per <subtype>.

The <simple expression> and <simple condition> require direct access. All language constructs can therefore be implemented with a simple scan over <maintype>. This linearity is also confirmed by research where Xplain DBMS was used with complex queries on large databases [8]. Therefore the semantic concepts can be used safely. It is important that high performance is automatically achieved without query optimizers because we do not know anything about user queries; the database is too dynamic and database access is too random.

Conclusion

By encapsulating extremely simple semantics in the data model and the query language the database systems are capable of supporting the user. This results in suitable database systems for casual environments. Solutions are unique, which implies that each problem solution is always the best solution. It implies that users in this environment can easily reach the level of an expert user.

Acknowledgments

Several persons are acknowledged for frequent interesting discussions and their support concerning research on semantic data models, among them: Martin van Dinther of Data Distilleries, Peter Janse of the Ministry of Finance, Peter van den Hamer and Herman ter Horst of Philips Research.

References

[1] P. Bernstein et al., The Asilomar Report on Database Research, SIGMOD Record, Vol. 27, No. 4, December 1998.

[2] C.J. Date, An introduction to database systems, Addison-Wesley (1993).
[3] D. Florescu, A. Levy and A. Mendelzon, Database techniques for the world-wide web: a survey, SIGMOD Record, Vol. 27, 3, September 1998.
[4] J. Mylopoulos, Next generation database systems won't work without semantics, Panel ACM SIGMOD Conference, 1998.
[5] S. Nestorov, S. Abiteboul and R. Motwani, Inferring structure in semistructured data, SIGMOD Record, Vol. 26, 4, December 1997, 39-43.
[6] F.D. Rolland, The essence of databases, Prentice Hall (1998).
[7] J.H. ter Bekke, Semantic data modeling, Prentice Hall (1992).
[8] J.H ter Bekke, Can we rely on SQL?, Proceedings 8th International DEXA Workshop, Toulouse (1997), 378-383.
[9] J.H. ter Bekke, Semantic modeling of successive events applied to version management, Cooperative Databases and Applications (eds. Y. Kambayashi and K. Yokota), World Scientific, Singapore (1997), 440-447.
[10] J.H. ter Bekke, Advantages of a compact semantic meta model, Proceedings 2nd IEEE Metadata Conference, Silver Spring (1997), http://www.computer.org/conferen/proceed/meta97/papers/jterbekke/jterbekke.html.
[11] G. Witt, Is data modeling standing still?, Database Programming and Design, August 1997, 64-71.