On Universal Database Design

Erki EESSAAR, Marek SOOBIK
Department of Informatics, Tallinn University of Technology, Estonia

Abstract. Requirements to software systems, including databases, often change. It is tempting to design a database according to very abstracts concepts. The generic design, which is called "universal database design", allows us to record all possible facts in terms of object types, objects, attributes, attribute values, and relationships. This kind of design arguably simplifies the recording of new kinds of facts. However, the flexibility comes with a price. Existing studies point mostly to the performance problems and complexity of queries. We found twelve different problems with this kind of design. In this paper, we present the analysis of the advantages and disadvantages of the use of the universal design.

Keywords. Database design, data management, universal design, SQL

Introduction

Two important properties of the system development methodologies of the *Internet time* are constant time pressure to the developers and vague requirements that often change [1]. Database designers must also consider these factors because *requirements* to a database evolve. One tempting solution seems to be the use of a highly generic database design that has different names: "Universal Data Model" [2], "The entity-attribute-value representation with classes and relationships (EAV/CR)" [3], "Generic data model" [4], and "vertical design" [5]. However, ease of evolution is not the only aspect that we must consider in case of selecting a suitable database design.

The *goal* of the paper is to present an analysis of the advantages and disadvantages of the use of the *universal design*. This analysis is based on the *experiments* and the results of a literature review. In our view this kind of analysis is currently missing in the research literature. Existing studies consider only *few* disadvantages of the universal design and provide few experimental results. In this paper, we extend the work that was started in [6]. We use the concepts of SQL [7] in the paper because existing literature about the universal design uses these concepts.

The rest of the paper is organized as follows. Firstly, we explain the principles of the universal design. Secondly, we analyze the advantages and disadvantages of the use of this design. Finally, we draw conclusions and point to the future work.

1. The Universal Design

The part a) of Figure 1 is pictorial representation of the universal design. The part b) of Figure 1 presents an example of the *regular design*. It is also possible to use a combination of these designs. The conceptual data model, which illustrates the regular

design, is a fragment of the model that is presented in the specification of the Transaction Processing Performance Council (TPC) benchmark C (TPC-C) [8].

The specification of the universal design uses the modeling principle, according to which a model should be explicitly divided into operational and knowledge levels [9]. "The knowledge level objects define legal configuration of operational level objects." [9] Data about an object system is recorded at the operational level in terms of the objects, their attributes, and relationships. Entity type *Attribute_value* has a set of attributes, the specification of which has the general form: *<<data_type_name>>_value data_type_name*. These attributes allow us to record values that have different types. The amount of these attributes and their data types depend on a database system (DBMS) where this database is created. Data at the knowledge level determines the legal values that can be associated with an object at the operational level. The knowledge level contains data about object types and their attributes. Some of the *variations* of the universal design are:

- The word "object" can be replaced with the words "entity" or "thing".
- Hay [2] proposes a many-to-many relationship between *Attribute* and *Entity_type* and specifies it by using entity type *Attribute_assignment*.
- *Attribute* or *Attribute_assignment* could have associated entity type *Legal_value* that allows us to specify the legal values of the attributes [2].
- *Attribute* could have an attribute or even associated entity type *Format* in order to permit recording of a format for the values of an attribute [2].
- Each supported data type should have exactly one corresponding table for recording attribute values with this type according to the EAV/CR approach [3]. This is different from the design a) in Figure 1. There is one generic entity type *Attribute_value* that has an attribute for each supported data type.
- Hay [2] proposes to record the type of each relationship. In addition, it is possible to record permissible relationship types between object types at the knowledge level. This data determines permitted relationships between objects at the operational level.

At first glance, the universal design seems like an easy way to achieve quick success. However, it also has many serious problems.

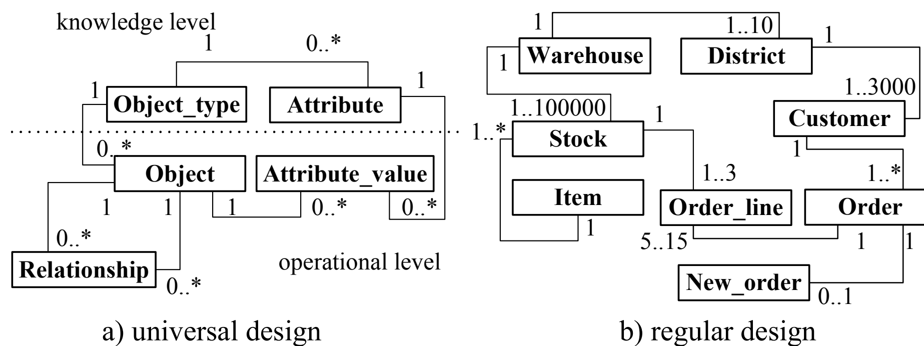

Figure 1. Examples of the database designs

Some authors have pointed to the problems with query complexity [4, 10] and query speed [4, 5]. These are not the only problems. It seems that there is a lack of *consensus* about this design and no *comprehensive* discussion of its shortcomings. Developers have tried to use it repeatedly in order to achieve maximum *flexibility*.

Systems that use some form of the universal design have to manage large amounts of data. Some bioinformatics systems use a database that is designed according to EAV/CR approach: (a) Subsystem of system net-TRIAL that helps to manage procedures and laboratory results of the clinical trials [11]; (b) SenseLab database for recording neuroscience data [12]; (c) System PhD for web-based management of phenotype data [13]. System GenMapper that helps scientists to integrate heterogeneous molecular-biological annotation data [10] uses database that is designed according to *Generic Annotation Model* that is a variation of the universal data model. It contains the source level (knowledge level) and the object level (operational level).

Some *software engineering systems* also use the universal design. Bernstein et al. [14] describe the Microsoft repository that uses a SQL-based relational DBMS in order to provide persistent storage for the different software tools. This database contains generic tables *Object* and *Relationship* among others. Habela [15] proposes a flattened metamodel that resembles the universal data model. Habela [15] envisages that the schema of a metadata database could be designed based on this metamodel. Bednárek et al. [16] describe the data integration system DataPile that records data in a repository that follows the rules of the universal design.

2. The Analysis

We decided to perform experiments based on databases that are created based on the specification of the TPC-C [8]. Why we decided to use this kind of simulation domain? Firstly, it is specified in a well-known benchmark and has therefore probably been carefully evaluated. Secondly, the specification provides requirements to the test data. Thirdly, the specification is available to public and therefore it is easier to repeat the experiments. Fourthly, it is in our view reasonably complex in terms of the amount of tables, data types, and integrity constraints. Finally, it is quite *generic* and represents "any industry which must manage, sell, or distribute a product or service" [8].

We used the database system (DBMS) PostgreSQL v8.1 [17]. The DBMS was installed in the server with the Intel Xeon CPU 2.40GHz processor (with Hyper-Threading technology) and 1 GB RAM.

We created tables based on the specifications in Figure 1. We assume that each entity type in the conceptual data model (see Figure 1) has a corresponding base table (table for short) in a SQL database that is created based on this model. The names of the tables are the same as the names of the entity types. For each table of the regular design the set of columns and their data types match with those presented in the TPC-C. Both sets of tables must contain data about the same objects and relationships.

For generating data, we created in a database user-defined functions that use system-defined generators of random values. The data was generated based on the requirements to the test data that is specified in the TPC-C. We generated data for the tables of the regular design (in the round brackets is the amount of rows): *Warehouse* (2), *District* (20), *Customer* (60000), *Item* (10000), *Stock* (20000), *Order* (60000), *Order_line* (599394), and *New_order* (18000). The amount of rows corresponds mostly to the requirements of the TPC-C. The only difference is that tables *Item* and *Stock*

contain 10 times less rows than is advised in the benchmark specification, due to restrictions to storage space.

We loaded all the data from these tables to the tables of the universal design (in the round brackets is the amount of rows): *Object* (767416), *Relationship* (1376808), and *Attribute_value* (4657166). Tables *Object_type* (8) and *Attribute* (63), which are at the knowledge level, were populated based on the specification of the regular design.

2.1. The Advantages of the Universal Design

It is possible to extend a database without executing DDL (Data Definition Language) statements. Instead, a user has to modify data at the knowledge level and the system has to execute DML (Data Manipulation Language) statements. Wang et al. [5] claim that it is possible to define new attributes (data elements) "without the additional programming". Still, experts (and not end users) must perform extension of the database because "incorrect metadata will yield a malfunctioning application." [12] Question remains – why is this approach better compared to the generation and execution of DDL statements by a system based on the instructions of a user?

These changes do not require corresponding changes in the user interface of an application, if there is one to one mapping between the columns in the tables and fields in the forms.

A query for finding all the data about an object has to access only one table (*Attribute_value*). If the attributes of an object type change, then the query does not need reprogramming [18]. If an object has attributes with the different types, then more than one table has to be accessed in case of the EAV/CR approach.

If the value of an attribute is missing, then we do not have to use NULL because we do not record a row in table *Attribute_value*. However, there are many reasons why the value of an attribute could be missing [19]. It is a useful data that could be recorded in a database. Figure 2 presents a conceptual data model of a possible solution to this problem in case of the universal design. The *external predicate* of a table is an *informal* construct that specifies what the data in the table means to a user [19]. The *parameters* of the predicate correspond to the columns. We write the parameters in *italics*. For example, table *Missing_value* has the simplified external predicate Eq. (1).

The following rules, which increase the complexity of the system, also have to be enforced: (1) If a row *r* in *Object* has an associated row in *Attribute_value*, then *r* cannot have the associated row in *Missing_value*. (2) If a row in *Relationship* represents a relationship between objects *o1* and *o2*, then there cannot be a row in *Missing_relationship* that represents the missing relationship between *o1* and *o2*. (3) The registration of a value of an attribute or a relationship must cause the deletion of the corresponding data about missing data.

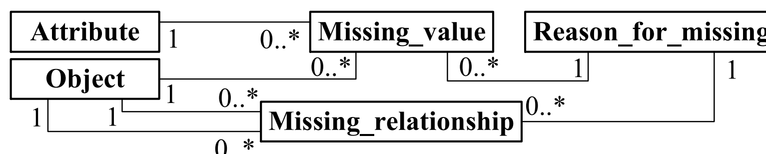

Figure 2. A possible solution for recording reasons of missing data

Value of the attribute *attribute_id* of the object *object_id* with the type *object_type_id* is missing because of the reason *reason_for_missing_id*. (1)

2.2. The Problems of the Universal Design

We evaluated the designs that are specified in Figure 1 by using some *database design metrics*. Piattini et al. [20] write: "the number of foreign keys in a relational database schema is a solid indicator of its complexity". The number of foreign keys is 6 and 8 in case of the universal design and the regular design, respectively. It shows that the *complexity* of the universal design is somewhat smaller compared to the complexity of the regular design. Piattini et al. [21] write: "the table size metric (TS) is a good indicator for the maintainability of a table". In our case all the columns of the tables are "simple columns" and hence the table size is the amount of columns in a table. The median TS is 3 in case of the universal design and 9.5 in case of the regular design. According to this metric the *maintainability* of the tables of the universal design is better compared to the tables of the regular design. Unfortunately, as we see in the next sections, these metrics do not take into account all the aspects of database design.

2.2.1. Database Schema Evolution

A database that is created according to the universal design may still need schema changes in the future because of the data types that are usable in a DBMS. Each data type could have a corresponding column in table *Attribute_value* or even a separate table in case of the EAV/CR approach. The set of predefined data types in a DBMS may change from release to release. Some of these changes are caused by the changes in standards. For example, SQL:1999 introduced the predefined type BOOLEAN [22]. SQL:2003 deleted the data types BIT and BIT VARYING [7]. DBMSs, which follow the prescriptions of SQL:2003, provide data type constructors and allow database designers to create user-defined types. Therefore, a large amount of data types could be used in a database. If new requirements stress the need for having an attribute with a data type that has no corresponding column in table *Attribute_value* or no corresponding separate table, then the database structure has to be changed. It seems reasonable to use the most popular predefined data types at the beginning and gradually add support to the data types. The result of the application of this kind of design could be the use of the limited amount of simple data types (for example, VARCHAR and INTEGER) as column types [18]. For example, the median of the amount of data types in case of the examples in [3, 10, 13, 15, 16] is 4. This, on the other hand, limits and complicates operations with the data values. An application that uses this database must perform type conversions.

All the data values that otherwise would be part of different tables are now in table *Attribute_value*. A DBMS usually locks a table exclusively in case of changing its structure. If someone changes the structure of *Attribute_value*, then it locks a very large portion of a database. Therefore, all the schema changes have to be done at the times, when the use of the *entire database* is as minimal as possible. Corruption of a database table or its indexes has far greater consequences compared to the regular design.

2.2.2. Expressiveness of a Database Schema

External predicates of tables do not give any information about the object system, the data of which is recorded in a database. For example, table *Attribute_value* could have the following external predicate Eq. (2). The exact predicate depends on the used types.

> The object *object_id* with the type *object_type_id* has an associated value of the attribute *attribute_id,* which is either an integer value *int_value*, or string value with the length between 0 and 2 characters *string_value_2*, or string value with the length of more than 2 characters *text_value*, or timestamp value *timestamp_value,* or Boolean value *boolean_value*. (2)

We need a special tool in order to present database conceptual schema based on the data at the knowledge level [12].

2.2.3. Integrity Constraints

It is more difficult to enforce constraints to the data values than in case of the regular design [18]. For example, the data at the knowledge level could state that an object type *ot* has a mandatory attribute *a* with the multiplicity "exactly 1". Attribute *a* has the data type *d*. Therefore, each object *o*, the type of which is *ot*, must have exactly one associated attribute value *v* (with type *d*) that is associated with the attribute *a*.

In case of the regular design of a SQL database we can enforce this constraint by declaring that a column of a table has NOT NULL constraint. It is more difficult in case of the universal design. The SQL standard permits the creation of assertions and the use of subqueries in the table CHECK constraints in order to create *declarative* constraints. However, most DBMSs do not support these features [23]. Therefore, trigger *procedures* (triggers) have to be created in order to enforce these rules at the database level. These triggers must react to the creation, modification, and deletion of a row of *Attribute_value*. If each data type has a corresponding separate table like in case of the EAV/CR approach, then each of these tables must have these triggers. The code of trigger procedures that implement a constraint *c* must ensure that concurrently running transactions, which cause the execution of a validation query of *c*, are *serialized* [24]. In case of the universal design it means locking of the entire table *Attribute_value*. It has bigger impact than in case of the regular design because *Attribute_value* contains data about objects with different types.

If data changes at the knowledge level, then triggers have to be created/altered/dropped as well. This means that the system has to generate and execute DDL statements after all. In addition, the system has to check whether the existing data violates new rules and in case of violation prohibit the changes at the knowledge level.

The primary key of table *Object* is (*object_id*, *object_type_id*). Fields that correspond to the column *object_id* of *Object* contain system-generated identifiers of objects. Values of the primary key do not prevent duplication of data about objects. It is difficult, if not impossible, to declare that a set of attributes of an object type must have unique values in case of the universal design. For example, let us assume that we want to enforce the rule that each warehouse must have a unique name. The constraint Eq. (3) of *Attribute_value* does not give the desired result because column *text_value* contains values of many different attributes – for instance, names of districts and last names of customers. Last names of customers do not have uniqueness constraint. A district and a warehouse could have the same name.

CONSTRAINT unique_warehouse_name UNIQUE (text_value) (3)

Sometimes it is possible to use *proprietary* solutions in order to solve this problem. For example, in PostgreSQL we could use the statement Eq. (4) in order to declare the key that consists of one attribute. In this case, *val1* is the identifier of attribute *w_name* of object type *Warehouse* and *val2* is the identifier of object type *Warehouse*.

CREATE UNIQUE INDEX idx_unique_w_name ON Attribute_value
(text_value) WHERE attribute_id=*val1* AND object_type_id=*val2*; (4)

We note that the SQL standard [7] does not specify indexes and therefore this solution is not universal. In this case, we do not declare database constraints that belong to the *conceptual* level of a database, but indexes that are constructs of the database internal level. If someone changes the identifiers *val1* or *val2* (in tables *Attribute* or *Object_type*), then this index enforces an incorrect rule. The primary keys of 6 tables, which are created based on the example of the regular design (part b of Figure 1), involve two or more columns. It is not possible to enforce these keys by using a unique index.

A database is "a collection of true propositions" [19]. A DBMS cannot enforce truth, but as an approximation, it can check that all the data values are consistent (i.e., conform to the integrity constraints) [19]. Not all the consistent propositions are correct, but all the correct propositions must be consistent. It is possible that the complexity of defining constraints leads to a database with few constraints. Constraint checking, if any, is done by an application that uses a database. It is likely that many constraints are not checked at all because they need complex queries (see 2.2.6).

2.2.4. Compensating Actions

A DBMS can sometimes resolve constraint violations as they arrive by executing a compensating action. We have to implement some compensating actions by using triggers. For example, if we wish that deletion of an object with the type *ot1* (for example *Order*) should cause cascading deletion of all the related objects (see *Relationship* in Figure 1) with the type *ot2* (for example, *Order_line*), then the use of "ON DELETE CASCADE" option in the declaration of a foreign key is not enough and we have to create a trigger.

2.2.5. Default Values

SQL permits the declaration of zero or one default value for a column of a base table. This feature is not always usable in case of the universal design. For example, attributes *ol_quantity* of object type *Order_line* and *c_payment_cnt* of *Customer* could have the default values 1 and 0, respectively. The values of these attributes are in the same column *int_value* of table *Attribute_value* in case of the universal design. Therefore, we have to use the triggers in order to use the default values. For example, if we decide to create one trigger, then it must contain a set of if-then statements, each of which specifies the default value of an attribute (see Eq. (5)). Values *val3* and *val4* are the identifiers of attribute *ol_quantity* and object type *Order_line*, respectively.

IF NEW.attribute_id = *val3* AND NEW.object_type_id=*val4* THEN
NEW.int_value:=1; END IF; (5)

An alternative is to create a separate trigger for each default value. If someone specifies new attributes or modifies the existing ones, then the system may have to generate and execute DDL statements for creating, replacing, or removing triggers.

2.2.6. Query Complexity

Next, we present an informal specification of four queries. *(Q1)*: Find the amount of customers whose last name starts with the letter "B". Table 1 presents the SQL code of this query in case of the different designs. (*Q2)*: Find the amount of customers whose last name starts with the letter "B" and who live in the state of Nebraska. *(Q3)*: Find the amount of customers whose last name starts with the letter "B", who live in the state of Vermont, and who belong to district, the location of which is also Vermont. *(Q4)*: Find the amount of order lines where the price of the associated stock is bigger than 99.

We selected the queries based on their complexity. *Q1* and *Q2* use data about one type of objects. *Q3* and *Q4* use data about two and three types of objects, respectively. In case of the regular design, the restriction condition of *Q1* is a simple predicate that involves no connectives and the restriction condition of *Q2* is a compound predicate that involves one connective.

Intuitively, we see that the SQL code is more complex in case of the universal design. How can we show it formally? Tow [25] proposes a method of tuning SQL queries, the part of which is the creation of a *query diagram*. It is a directed graph that represents a query. In this graph the nodes are *table aliases* and arcs represent *join* and *semijoin* operations. We can use the metrics of complexity of a graph [26] in order to measure the complexity of a query. We constructed the query diagrams and in case of each diagram counted the amount of nodes (N) and the amount of arcs (A). From the set of complexity metrics that are introduced by Latva-Koivisto [26] we calculated the *Coefficient of Network Complexity*: CNC=(A*A)/N. The bigger the value is the more complex is the graph (query). It is possible to use other complexity metrics as well.

Based on Table 2 we can conclude that the resolution of *the presented problems* requires more complex queries in case of the universal design than in case of the regular design. It is possible to simplify the query-writing task by creating operators [10] or viewed tables. However, after performing the view resolution, a DBMS still has to execute a complex query even in case of the simple problems. Chen et al. [3] propose to use combinations of simpler queries and temporary tables in order to speed up the queries. In this case, a user of a database looses an advantage of a DBMS according to which a user can make a (complex) query and a DBMS decides how to execute it. In this case, a query designer has to describe a *procedure* how to retrieve the desired results.

Table 1. Example of queries in case of different database designs.

Universal design	Regular design
SELECT Count(*) AS amt FROM Attribute_value AS AV INNER JOIN (Attribute AS A INNER JOIN Object_type AS OT ON A.object_type_id = OT.object_type_id) ON (AV.object_type_id = A.object_type_id) AND (AV.attribute_id = A.attribute_id) WHERE OT.name = 'CUSTOMER' AND A.name = 'C_LAST' AND AV.text_value LIKE 'B%';	SELECT Count(*) AS amt FROM Customer WHERE c_last LIKE 'B%';

Table 2. Query complexity.

	Universal design			Regular design		
Query ID	N	A	CNC	N	A	CNC
Q1	3	2	1.33	1	0	0.00
Q2	6	5	4.17	1	0	0.00
Q3	10	9	8.10	2	1	0.50
Q4	9	8	7.11	3	2	1.33

2.2.7. Size of Data

Chen et al. [3] write: "The EAV/CR representation consumed approximately four times the storage of our conventional schema." Conventional schema is created according to the regular design.

Our findings support this conclusion. We analyzed the tables that are specified in Section 1 (see Table 3). PostgreSQL provides the system-defined functions *pg_relation_size* that can be used in order to find the disk space (in bytes) used by a table [17]. PostgreSQL provides the system-defined function *pg_total_relation_size* that can be used in order to find the total size of a table together with its associated indexes and toasted data (in bytes) [17]. Before calculating the size of tables we collected statistics and reclaimed storage that was occupied by deleted rows.

The only indexes that the tables had during the calculation of data size were the B-tree indexes that were created automatically due to the primary key constraints. Each table had the primary key. The primary keys of the tables of the regular design are specified in the document of TPC-C [8]. In average, both the tables of the universal design and the regular design had 2.4 columns in a primary key. Each index contains the values of indexed columns. Therefore, due to large indexes, the difference between the *sum of table size* and the *sum of total table size* is relatively big.

Data in the column "Sum of table size" is calculated by summarizing the sizes of tables that are found by using *pg_relation_size.* The sum of sizes of tables of the universal design is approximately 4.75 times bigger than the sum of sizes of tables of the regular design. Data in the column "Sum of total table size" is calculated by summarizing the sizes of tables that are found by using *pg_total_relation_size.* The sum of total sizes of tables of the universal design is approximately 6.01 times bigger than the sum of total sizes of tables of the regular design.

"It is true that EAV/CR is more space-efficient for sparse data." [3] Our test data is not *sparse*. Only 2.4 percent of columns of the tables of the regular design (Figure 1. part b) are optional (permit NULLs).

Table 3. The size of tables that are created according to the different designs.

	Sum of table size (in bytes)	Sum of total table size (in bytes)
Universal design	687415296	1083359232
Regular design	144728064	180404224

2.2.8. Performance of Database Operations

Next, we present the results of measurements of *the performance* of database operations in case of the different designs. We performed five times each query (Q1-Q4) that was specified in Section 2.2.6. In addition, we measured the performance of a transaction (RW): Create new order where the amount of order lines is between 5 and 15. The amount of order lines is randomly selected.

Each time we measured the time (in milliseconds) that a DBMS needed in order to perform the operation and present the results. For each operation, we calculated the *average* time in millisecond (see Table 4) based on these five measurements.

In case of *Q4* the performance was firstly better in case of the universal design. However, the performance of the same operation was better in case of the regular design (*Q4**) after we created a composite index that covers the columns *ol_i_id* and *ol_supply_w_id* of table *Order_line*.

2.2.9. Dependencies Between Database Objects

Database objects like triggers, declarative constraints, and conditional indexes depend on the specifications at the knowledge level in case of the universal design. Data changes at this level can cause creation, modification, or removal of these database objects. Dependencies between the database objects are automatically recorded in a database catalog by a DBMS. A database developer has to explicitly design and implement tables for recording the dependencies in case of the universal design.

2.2.10. Access Control

SQL provides statements for granting privileges that allow us to perform a given action on a specified table or column. Let us assume that: (1) a database contains data about objects with types *ot1* and *ot2*; (2) user *u1* has right to select and update data that correspond to *ot1* and does not have rights to use the data that corresponds to *ot2*.

It is unreasonable to grant *u1* direct access to tables *Object* and *Attribute_value.* These tables contain data about objects with type *ot1* as well as data about objects with type *ot2*. We could create two viewed tables that present data about objects with types *ot1* and *ot2*, respectively. Then we can give *u1* rights to use these tables in order to see or modify data. In some DBMSs (like PostgreSQL 8.1) it is not possible to modify data in base tables through viewed tables without further programming [17].

Table 4. Performance test results.

Operation ID	Regular design (ms)	Universal design (ms)
Q1	259,213	877,700
Q2	260,305	1865,397
Q3	26,252	654, 364
Q4	1581,838	599,413
Q4 *	396,471	599,413
RW	207,682	327,591

The result might be that the systems, which use a database that follows the universal design, do not use the security mechanisms of a DBMS, in order to restrict access to the data.

2.2.11. Concurrency Control

Locking is a widely used mechanism for concurrency control by DBMSs. There are situations that need special care in case of the universal design. Modification of data about an object (its attribute values and relationships) should restrict concurrent modification of data about the same object. For example, two users could change data of a district concurrently so that one modifies the state and another modifies the city. The result could be an incorrect specification of the district.

In addition, modifications at the knowledge level that influence the operational level should restrict concurrent data changes at the operational level.

If we use the regular design, then it is usually sufficient to rely on locking that is *automatically* performed by a DBMS. On the other hand, in case of the universal design we have to use some statements of a database programming language in order to *explicitly* lock the data. Let us assume that we use DBMS PostgreSQL:

Regular design: If we modify the structure of table *District*, then a DBMS does not allow concurrent modifications (insertions, updates, deletions) of data about districts.

Universal design: If we modify the data in table *Attribute*, then the unstandardized LOCK statement has to be used. It blocks modifications of data about districts in table *Attribute_values*. However, it blocks modification of attribute values of *all* the objects.

2.2.12. User Interface Design

Marenco et al. [12] write that data in a database that follows the universal design "must be transiently converted (‘‘pivoted’’) into a conventional representation through fairly elaborate metadata-driven code." [12] It requires "considerable front-end programming" [18]. Conventional representation means that each data element is presented in the separate field with the meaningful label.

3. Conclusions

We identified *four* advantages and *twelve* problems of the universal design. Some studies suggest that it is easy to extend a database, which is created based on the universal design. However, we found that it is difficult to extend this kind of database in terms of the use of data types, integrity constraints, and default values. In addition, experiments showed us that the use of the universal design causes problems in terms of query complexity and performance. Whether or not to use the universal design depends on the importance of the different database design aspects in a particular project. The results also illustrate the need to extend the existing database design metrics and create new metrics because, for instance, they do not consider all the integrity constraints.

In conclusion, the universal design advocates building a DBMS on top of a DBMS. The knowledge level is actually a database catalog – an addition to the one that is automatically created by a DBMS. Designers have to work out many *ad hoc* solutions and do redundant work instead of relying on the system-defined features of DBMSs. Many features that are present in a DBMS have to be duplicated in the applications.

The future work must include more experiments based on databases from different domains. There is also a need to investigate alternative database designs that can be used in order to cope with changing requirements.

References

[1] R. Baskerville, J. Pries-Heje, Racing the e-bomb: how the Internet is redefining information system development methodology, In: Realigning Research and Practice in Information System Development, Proceedings of the IFIP TC8/WG8.2 Working Conference, (2001) 49-68.

[2] D.C. Hay, *Data model patterns: conventions of thought*, Dorset House Pub, New York, 1996.

[3] R.S. Chen, P. Nadkarni, L. Marenco, F. Levin, J. Erdos, P.L. Miller, Exploring Performance Issues for a Clinical Database Organized Using an Entity-Attribute-Value Representation, *Journal Of The American Medical Informatics Association*, 7 (2000), 475-487.

[4] T. Kyte, *Effective Oracle by Design*, Oracle Press, McGraw-Hill/Osborne, 2003.

[5] S. A. Wang, F. Yang, C. Huey, F. Pecjak, B. Upender, A. Frazin, R. Lingam, S. Chintala, G. Wang, M. Kellog, R.L. Martino, C.A. Johnson, Performance of using Oracle XMLDB in the evaluation of CDISC ODM for a clinical study informatics system, In: Proceedings of the 17th IEEE Symposium on Computer-Based Medical Systems. (2004) 594- 599.

[6] E. Eessaar, Relational and Object-Relational Database Management Systems as Platforms for Managing Software Engineering Artifacts, Ph.D. thesis, Tallinn University of Technology, Estonia, 2006.

[7] J. Melton, ISO/IEC 9075-2:2003 (E) Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation), August 2003. Retrieved December 26, 2004, from http://www.wiscorp.com/SQLStandards.html

[8] TPC BENCHMARK C Standard Specification Revision 5.8.0, 2006. Retrieved December 29, 2007, from http://www.tpc.org/tpcc/spec/tpcc_current.doc

[9] M. Fowler, *Analysis Patterns: Reusable Object Models,* Addison Wesley Professional, 1997.

[10] H.H Do, E. Rahm, Flexible Integration of Molecular-Biological Annotation Data: The GenMapper Approach, In: Proceedings of the 9th International Conference on Extending Database Technology, LNCS Vol. 2992/2004. Germany: Springer Berlin, (2004) 811 – 822.

[11] D. Hageman, D.M. Reeves, net-Trials TM Clinical Trials Information System, In: Proceedings of 14th IEEE Symposium on Computer-Based Medical Systems, (2001) 141-145.

[12] L. Marenco, N. Tosches, C. Crasto, G. Shepherd, P.L. Miller, P.M. Nadkarni, Achieving Evolvable Web-Database Bioscience Applications Using the EAV/CR Framework: Recent Advances, *Journal of American Medical Informatics Association*, 10 (2003), 444–453.

[13] J.L. Li, M. X. Li, H.Y. Deng, P.E. Duffy, H.W. Deng, PhD: a web database application for phenotype data management, *Bioinformatics*, 21 (2005) 3443-3444.

[14] P. A. Bernstein, B. Harry, P. Sanders, D. Shutt, J. Zander, The Microsoft Repository, In: Proceedings of the 23rd International Conference on Very Large Data Bases, Morgan Kaufmann, (1997) 3-12.

[15] P. Habela, *Metamodel for Object-Oriented Database Management Systems*, Ph.D. Thesis, Polish Academy of Sciences, Warsaw, Poland, 2002.

[16] D. Bednárek, D. Obdržálek, J. Yaghob, F. Zavoral, Data Integration Using DataPile Structure, In: Proceedings of ADBIS 2005. Tallinn: Institute of Cybernetics at Tallinn University of Technology, (2005) 178-188.

[17] PostgreSQL 8.1.10 Documentation. Retrieved December 23, 2007, from http://www.postgresql.org/docs/8.1/interactive/index.html

[18] J. Ahnøj, Generic Design of Web-Based Clinical Databases, *Journal of Medical Internet Research*, 5 (2003).

[19] C.J. Date, *An Introduction to Database Systems*, 8th edn, Pearson/Addison Wesley, 2003.

[20] M. Piattini, C. Calero, M. Genero, Table Oriented Metrics for Relational Databases, *Software Quality Journal*, 9 (2001) 79–97.

[21] M. Piattini, C. Calero, H. Sahraoui, H. Lounis, Object-Relational Database Metrics. *L'Object*, March 2001.

[22] P. Gulutzan, T. Pelzer, *SQL-99 Complete, Really*, CMP Books, 1999.

[23] C. Türker, M. Gertz, Semantic integrity support in SQL:1999 and commercial (object-) relational database management systems, *The VLDB Journal*, 10 (2001) 241–269.

[24] L. de Haan, T. Koppelaars, *Apllied Mathematics for Database Professionals*, Apress, USA, 2007.

[25] D. Tow, *SQL Tuning*, O'Reilly, 2003.

[26] A. Latva-Koivisto, Finding a complexity measure for business process models, Research Report, 2001.