EDSOs, implosion and explosion

Veldwijk, R.J.

1990

SERIE RESEARCH MEMORANDA

EDSOs, implosion and explosion: concepts to automate a part of application maintenance

R.J. Veldwijk
M. Boogaard
M.V. van Dijk
E.R.K. Spoor

Researchmemorandum 1990-25

Mei 1990

VRIJE UNIVERSITEIT
FACULTEIT DER ECONOMISCHE WETENSCHAPPEN
EN ECONOMETRIE
AMSTERDAM

EDSOs, implosion and explosion: concepts to automate a part of application maintenance

by R J VELDWIJK*, M BOOGAARD, M V VAN DIJK and E R K SPOOR

Abstract: The purpose of this article is to present a set of concepts to automate application maintenance which is necessary when structural changes in an implemented relational database occur. The kind of structural alterations in consideration are those that need no human interference with database structure and application program sources, and do not change the information contents of the database. These alterations only rearrange the description of the database tables and the way those tables are related to one another. The structural alteration process consists of the following three steps: implosion, Elementary Database Structure Operations (EDSOs) and explosion. Alteration of a database structure, however, requires changes in the DML used in application programs that make use of the database. This DML alteration process uses the same three steps mentioned above. The necessary information is obtained from an advanced data dictionary.

Key words: automated application maintenance, database, data dictionary, flexible applications, meta-data

* RAET N.V., Eendrachtlaan 10, Postbus 2382, 3500 GJ Utrecht, The Netherlands.

Introduction

A considerable part of the capacity of data processing departments is spent on maintenance of existing application programs. This phenomenon, known as the software crisis, is caused by an inherent inflexibility of information systems. Often in the long run the maintenance costs even exceed the implementation costs of these inflexible information systems. Because in most cases an initially stable application procedure turns out to be the subject of change. Quoting Martin the above fact must be considered a major principle of data processing, because 'you can never tell how the application of stored data will eventually evolve'[1]. Information systems designed on the basis of this principle use databases to store the data instead of hard-coding the data in the application programs. Storing data in a database allows restructuring of the data without affecting the application programs that make use of the database (i.e. physical data independence). With the help of database management systems, application programs become physically data independent and therefore immune for changes on the physical level. Where logical data independence is concerned the problem of inflexible application programs still remains. For instance, to date it would not be customary to hard-code a VAT-percentage. Instead this percentage is read from the database. VAT-changes therefore do not give rise to recoding. Obviously this is a trivial example of introducing flexibility, but the current practice does not go very far beyond this level. If, for instance, the VAT-percentage were to be changed from 18 to 17.5, every application program would need recoding ('picture 99.9') to keep it tailored to user requirements.

While trying to avoid hard-coding in application programs is an accepted practice to improve flexibility, the same does not apply to database design. It is common practice to tie the structure of a database as much as possible to the structure of the environment about which information is stored (i.e. the schema). For example, a database structure is designed for storing information about suppliers and parts. If a certain part can be delivered by only one supplier the normalization process leads to a one-to-many relationship between suppliers and parts. A database design in which parts can be delivered by more than one supplier would be unsound because it would permit 'wrong' database contents. In that case database integrity could only be maintained by programming extra screening routines.

Consequently, schema modification or structural database alteration results in rewriting the affected application programs. This causes a large part of the remaining maintenance problem besides data independence. Again quoting Martin: 'An application program can not be independent of all changes that could be made to the data it uses. ... any change which impacts the program's internal algorithm rather than merely the nature of the data will necessitate program rewriting'[1]. One may conclude that the structure of an information system is predominantly hard-coded in both application programs and database structure.

The current attempt to solve this problem is to increase programming productivity of the maintenance programmer. By means of structured programming techniques, detailed documentation as well as adoption of a 4GL-RDBMS, application maintenance can be done more efficiently. These "solutions" do not solve the maintenance problem as such, but only make it a more a bearable one. They result in more automation for the same amount of money, but do not change the development/maintenance ratio.
The only way to increase this ratio is to achieve a higher degree of flexibility of information systems. An expensive and impractical way is to try to predict future changes, and program the possible changes in advance. An alternative reply is the use of views. Flexibility is indeed increased if 'column-subset views' are used, nevertheless, problems arise if nonupdatable views are used[2]. Hence, using views only partial solves the matter. At least from a conceptual point of view the only way to achieve true flexibility is by avoiding hard-coding in both areas mentioned above. All information about application programs and database structure should be transferred to the contents of a database.
A database containing this kind of structural information is a data dictionary. In a comprehensive data dictionary all system specifications can be stored. Because the data dictionary itself is an information system too, its specifications can be stored in its own tables. A similar reasoning can be applied to the application database whose structure is stored in the data dictionary. It will be shown that it is possible to reduce any conceivable database structure to just one table without losing information and without introducing redundancy, i.e. 'implosion'. The affected DML statements in application programs that make use of the data dictionary must be rewritten for the imploded database. Because of the implosion both the database and the statements lose most of their structure. The central thesis of this article is that this dual loss of structure enables a formalized procedure for dealing with changes in an information system.

In this article we shall introduce an approach to implement changes in database structures and adapting the affected statements in application programs automatically*. The kind of structural changes that need no human interference are those that do not change the information contents of the database but only rearrange the description of the database tables and the way those tables are related to one another.
In the first section our proposed procedure to formalize (automate) the implementation of structural changes in a database is explained and illustrated by an exemplary database implementation. The next section states the requirements for a data dictionary that is to support the described procedure and introduces an 'imploded' or

---

* Although the procedure has been implemented in the described manner it should be noted that our aim is to present a conceptual framework. It is conceivable to implement the approach in a different manner using the framework presented in this article.

'ultimate' data dictionary. In the subsequent section the concept of imploding a database structure and its contents, and then rewriting the DML onto the ultimate data dictionary is demonstrated, again illustrated by the fictional example database of the first section. The next section demonstrates the effect of a database structural alteration on the contents of the ultimate data dictionary and a procedure for implementing the corresponding changes in the example DML statements is discussed. In the subsequent section the explosion of the new database and the rearranged statements is illuminated. The article concludes with some critical notes and conclusions on the subject.

Changing database structures

EDSOs

A Database Structure Operation (DSO) can be defined as an operation which alters the structure of an existing relational database without changing its information contents. Whenever a current database structure is no longer suitable a DSO enables us to change the database structure in accordance with the changes of user requirements.

There are three reasons for changing a database structure:
1 correcting wrong database design (normalization);
2 improving application performance (denormalization);
3 adapting a database structure to changes in user requirements*.

The specific features of the required DSO depend on the structure of the existing database and the desired database structure. Therefore, every DSO is different/unique and will have to be analyzed individually. However, it is possible to break up a DSO into a certain number of structural alteration types. By generalizing (and automating) the structural alteration types, they will be applicable for every database structure. With the aid of the generalized types, i.e. EDSOs (Elementary DSOs), a DSO has not to be analyzed individually anymore, instead it will be possible to decompose the DSO into a number of EDSOs**.

A relational database consists of tables filled with data. The structure of the database is established in the relationships between the tables and the description of the tables. Thus, EDSOs result in changes of the relationships between tables and/or the description of the tables. An EDSO may even introduce a new table. The only principal restriction is that the information contents of the database remain the same.

We shall discuss the problem of restructuring a database by means of a suppliers-and-parts database that consists of two tables:

SUPPLIERS (S#, SNAME, CITY)
PARTS (P#, S#, PNAME, QTY)

---

* Although changed requirements usually lead to changes in the information contents of a database, part of the changes can be seen as a DSO. Neglecting this DSO component often leads to systems that are very hard to maintain.

** Establishing a complete set of EDSOs will be the subject of 'Decomposition of structural database alterations'[3].

The tables hold the following data:

SUPPLIERS

S#	SNAME	CITY
S1	SMITH	LONDON
S2	JONES	PARIS
S3	BLAKE	PARIS

PARTS

P#	S#	PNAME	QTY
P1	S1	NUT	300
P2	S1	BOLT	200
P3	S2	SCREW	400

The two tables are related by the foreign key 'S#' of the table called PARTS. It indicates a one-to-many relationship between SUPPLIERS and PARTS which means that a supplier can deliver more than one part but that a part can not be delivered by more than one supplier.

We shall now assume that identical parts can be delivered by more than one supplier, for instance part 'P1' can be delivered by both supplier 'S1' and 'S2'. The original one-to-many relationship can not deal with the many-to-many situation described above. The current database structure would cause problems for stock keeping departments*. Therefore, the relationship between SUPPLIERS and PARTS must change into a many-to-many relationship. While the SUPPLIERS-table remains unchanged, the PARTS-table is modified and a new table, SP, is introduced:

PARTS (P#, PNAME, QTY)
SP (S#, P#)

The tables hold the following data:

PARTS

P#	PNAME	QTY
P1	NUT	300
P2	BOLT	200
P3	SCREW	400

SP

S#	P#
S1	P1
S1	P2
S2	P3

Note that the information contents have not changed but that the new database structure permits the combination 'P1 S2' in the SP-table. In our view this DSO is a basic one, i.e an EDSO. The inverse procedure would also be an EDSO**. Another example of an

---

* In such cases it is common practice to assign two part numbers to the same part, because changing the database structure and the application programs would take too much effort.

** The inverse EDSO can only take place if there are no parts that are supplied by more than one supplier.

EDSO would be the transfer of PNAME to the new SP-table, because different suppliers may use different names for the same part.

The effect on DML

Changing an implemented database structure affects the DML statements of application programs that make use of the database. For instance, it is possible that after the alteration of a database structure, attributes specified in the SELECT clause no longer refer to the table specified in the FROM clause. Consequently, an EDSO requires the adjustment of the DML statements to the new database structure.

Consider the following DML statements:

Statement A

```
SELECT S#, SNAME
FROM SUPPLIERS
WHERE S# IN
      (SELECT S#
       FROM PARTS
       WHERE QTY > 250)
```

Statement B

```
SELECT S#, SNAME
FROM SUPPLIERS
WHERE S# IN
      (SELECT S#
       FROM SP
       WHERE P# IN
             (SELECT P#
              FROM PARTS
              WHERE QTY > 250))
```

Both statements are equivalent. If statement A yields a result on the original database, statement B does the same for the modified structure. Automating changes in database structure also requires automating the restructuring of the DML statements that are affected by the structural alteration.

In conclusion, the definition of an EDSO can be extended to 'an elementary operation which alters the structure of an existing relational database and adjusts the affected DML statements in application programs of the database system to the altered database structure'.

The procedure

The implementation of the EDSO concept can be divided into three parts:

1 the selection of a single EDSO or a number of EDSOs for a given DSO;
2 the database structural alteration process;
3 the DML alteration process.

The relation between these parts can be visualized in a flowchart like figure 1.

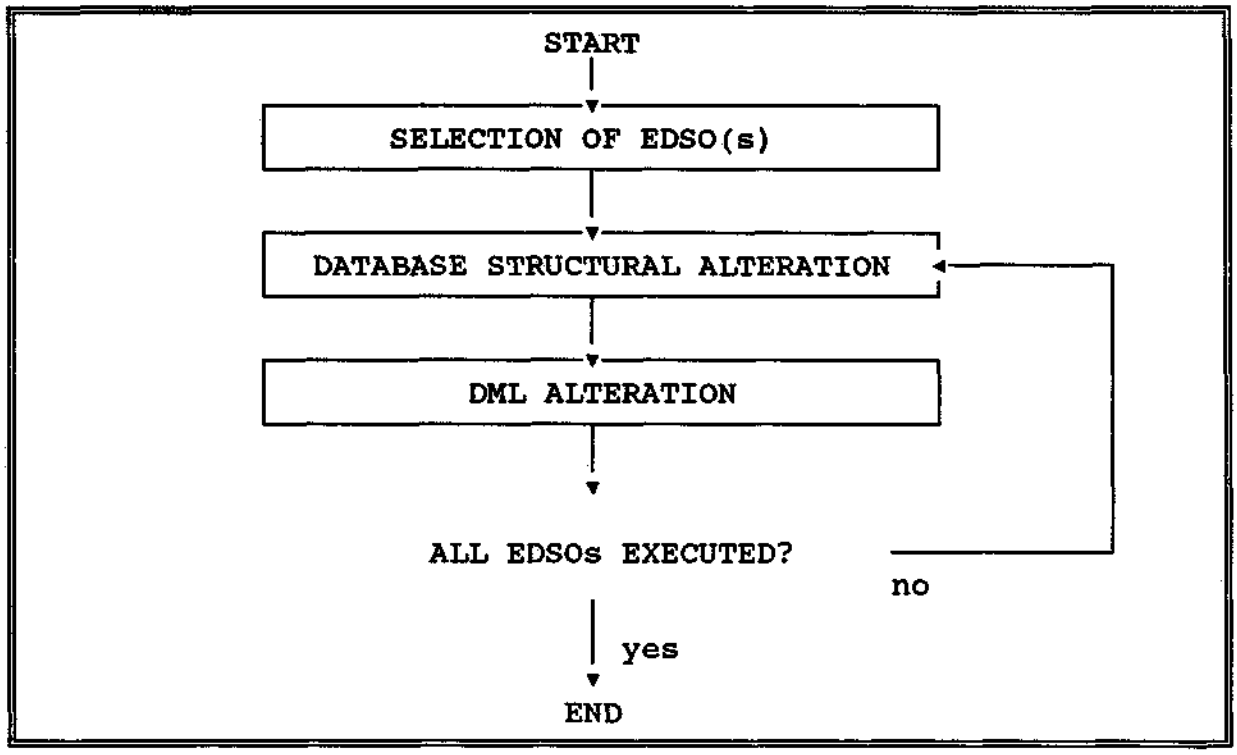


Figure 1. Flowchart of the EDSO concept

Selection.

Alteration of the database structure starts with the kind of database structure that is desired. Any kind of structural alteration can be described with a complete set of EDSOs. In our suppliers-and-parts example, we shall select just one EDSO that changes a one-to-many relationship between two tables into a many-to-many relationship. After the selection of the required EDSO(s), the actual alteration of the database structure starts. The structural alteration process is followed by the DML alteration process. These two processes must be executed for every selected EDSO.

Database.

The database structural alteration process consists of the following three steps (figure 2):

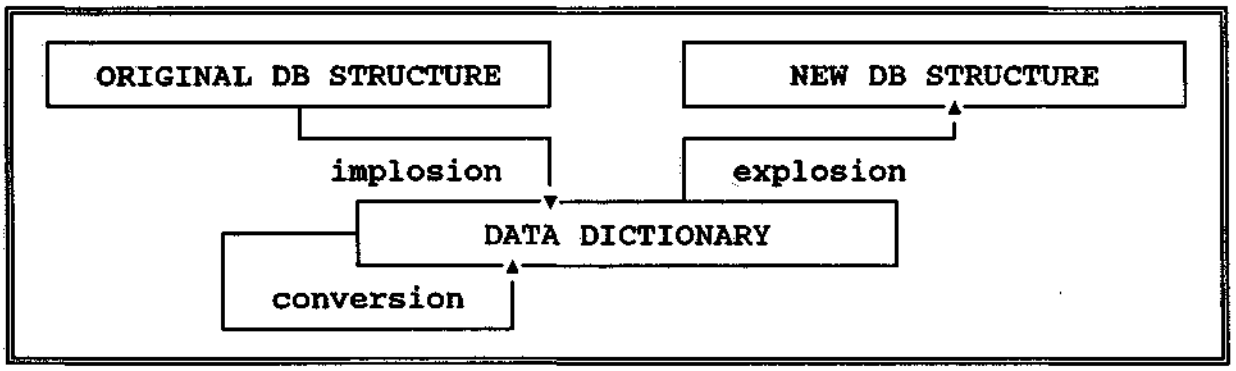


Figure 2. The database structural alteration process

The first step (IMPLOSION) is a transformation of the necessary database tables in the data dictionary. Because the SUPPLIERS-table does not change during the alteration process, it is only the PARTS-table that requires implosion into the data dictionary. This first step is the same for every EDSO.

The second step (CONVERSION) is specific for the chosen EDSO. After the transformation in the data dictionary, the imploded tables are removed (dropped) from the database. Because the structure of the original database is stored in the data dictionary, altering the structure can be realized by changing the specifications of the database tables in the data dictionary as well as the relationships between them with the aid of DML. After this, we have to adapt the contents of the original database to the new database structure, again using DML. The complete specification of the new database is now available in the data dictionary. The conversion ends with the creation of the necessary tables. For our suppliers-and-parts example this means the creation of the tables PARTS and SP.

The last step in the structural alteration process (EXPLOSION) is, like implosion, the same for every EDSO. So far the created tables are empty. In the explosion process the created tables are filled with the contents of the original database that is stored in the data dictionary.

DML statement.

The DML alteration process is quite similar to the structural alteration process. It is essential to determine which DML statement in which application program is affected by the EDSO, but apart from that, the process follows the same three-step-procedure

as the structural alteration process (figure 3).

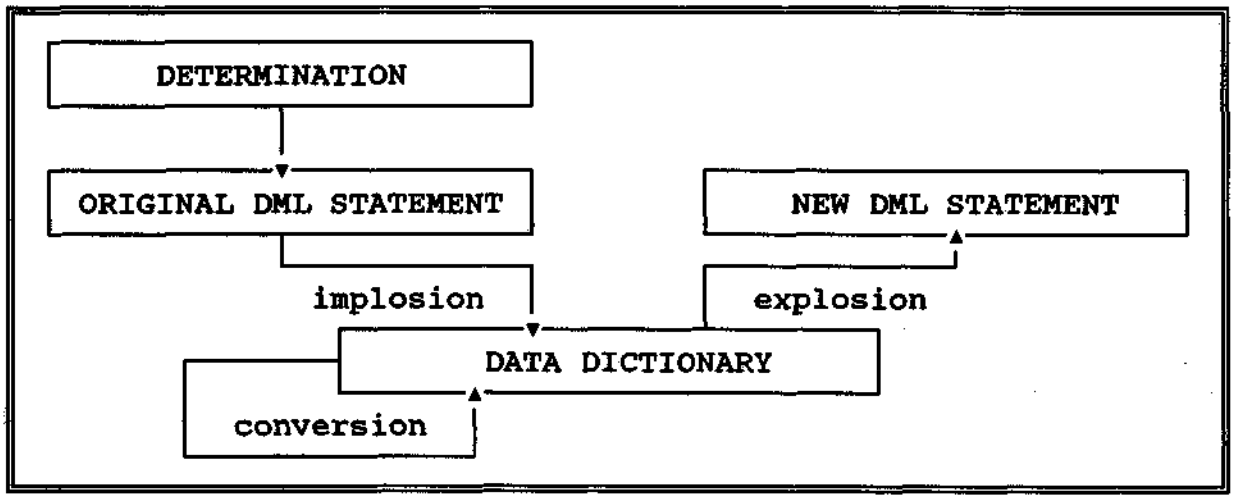


Figure 3. The DML alteration process

The first step in the DML alteration process (DETERMINATION) is the search for all DML statements in every application program affected by the alteration of the database structure. This step uses the data dictionary, since all the necessary information is stored in the data dictionary. Each affected statement and its required application program specifications (i.e. program-id and line number), are stored in a table.

The second step (IMPLOSION) is a translation of the above mentioned DML statements to the data dictionary that still contains the information that was stored in the (now removed) original tables. The result is a loss of statement structure, which enables formalized change of the statements.

These first two steps are the same for every EDSO. The third step (CONVERSION), however, is specific for the chosen EDSO. In this part of the process the imploded DML statements are altered according to the new database structure that is available in the data dictionary.

The last step in the DML alteration process (EXPLOSION) is, again, the same for every EDSO. The modified DML statements are transformed in the new database structure. The adapted statements are inserted into the original application program in their original place.

When all the selected EDSOs for the DSO are executed sequentially, the complete database system will have been adapted to the desired database structure.

Data dictionary requirements

The implementation of this concept requires an advanced data dictionary. Such a data dictionary is, in our opinion, the next stage in the current evolution of the use of data dictionaries. The stages of this evolution are summarized below. Note that every subsequent stage contains the features of the preceding stage, the last stage (i.e. six) describes the data dictionary essential to the EDSO concept.

The six stages of the evolution of the use of data dictionaries:

1 Storage data dictionary
The data dictionary is used as a central storage device in which the descriptions (meta-data) of, for instance, tables and columns are kept[2].

2 Semantic data dictionary
By adding constraints and domains, the data dictionary is used to assure the semantics of the database, like entity integrity[4,5].

3 Active data dictionary
The application programs in the database system use the data dictionary for their meta-data. This means that a change in the meta-data takes just one action in the data dictionary and affects the application programs without changing them[6,5].

4 Self actualizing data dictionary
The data dictionary contains data about its own structure. Quoting Ross this means that 'since data about the current configuration of the data dictionary are always readily available, nothing (or almost nothing) needs to be ... 'hard-coded' into the system'[7].

5 Self actualizing applications
The data dictionary not only contains the description of the data but also contains the description of the design of the application programs of the database system. This means that the data dictionary consists of meta-database-data and meta-application-data. Because the data dictionary treats these meta-data in the same way it is not necessary to make this conceptual distinction. Since the data dictionary contains all the necessary meta-data about the design of the application programs, the data dictionary is able to (re)generate application programs using these meta-data. This leads to generalized application programs suitable in specific environments. Ross referred to this stage as 'the next frontier'[7]. As far as we know no one has elaborated this self-actualizing-applications concept.

6 'Ultimate' data dictionary
In our view the last frontier is the 'ultimate' data dictionary. The proposed data dictionary is an active, self actualizing data dictionary containing meta-database-data as well as meta-

application-data. Up till this stage there were at least two levels: the database and the data dictionary*. The database includes operational data and a certain structure, the dictionary includes the structural description of the database. This redundancy can be eliminated by also storing (imploding) the operational data in the data dictionary. Consequently, only one level remains, the ultimate data dictionary[8].

The ultimate data dictionary consists of just one table, called VALUE. This table holds all information any database (or data dictionary) can contain**.

---

* There are two levels only if the data dictionary is considered to be fully self-referential. This means that all data dictionary specifications are stored within itself. If not, the specifications of the data dictionary are stored in a meta-data dictionary (and so on).

** Since it is not necessary to have a database consisting of more than one table, the described procedure can be simplified considerably because the implosion and explosion concepts are no longer required.

Implosion: changing structure into contents

In this section the suppliers-and-parts database and subsequently two DML SELECT statements will be imploded. Implosion of a database means that each value in the table along with its structural specifications is incorporated in the data dictionary table VALUE. The structural specifications of the table VALUE itself are:

VALUE (ENT_NM, ATT_NM, TUPLE#, VAL).

Where: ENT_NM is the name of the table (entity),
ATT_NM is the name of the column (attribute),
TUPLE# is used to relate the values occurring in the same initial tuple and
VAL contains the value belonging to an attribute.

Obviously the structural description yields hardly any information about the data dictionary or any other database structure; that information can be found in the contents of the table VALUE. Along these lines every database structure can be transformed into database contents*.

The values of the columns ENT_NM and ATT_NM in the table VALUE can be found in the original database structure, TUPLE#, however, is a newly introduced feature and needs some explanation. During the database implosion, the values of a tuple are disintegrated. The implicit relations between the values are made explicit in order to make up for the break up of structure. To prevent any loss of information, the column TUPLE# is introduced. All attribute occurrences in the original database that belong to one tuple are explicitly connected in VALUE because those attribute occurrences values have identical ENT_NMs and TUPLE#s.
The TUPLE# as it appears in VALUE, however, is not used to put the tuples in any order and therefore it does not contradict the relational theory. The result of the implosion of PARTS is depicted in figure 4. Every value of PARTS has become a single tuple in VALUE. The tuple 'P1 S1 NUT 300' in PARTS, for example, can be reconstructed in VALUE by selecting all tuples where the table name (ENT_NM) equals 'PARTS' and the TUPLE# equals 1.
The result of the implosion is that both structure and contents of the database are occurring as the contents of the data dictionary table**. In the original pre-implosion situation the alteration of the database structure would need Data Definition Language

---

* The structure of the data dictionary itself will not be discussed in this article. It is sufficient to note that it must contain all information about the structure of a database and that this information can be stored in the VALUE-table.

** Descriptions of the values (for instance data type and format of the value) are also stored in the VALUE-table. The values of the column VAL are variable length, alphanumeric fields.

statements to change the structure; moreover some complex additional operations would have to adjust the contents of the database to the new structure. If an imploded data dictionary is used instead, a similar alteration can be realized quite easily by directly executing DML statements in the imploded data dictionary[9].

ENT_NM	ATT_NM	TUPLE#	VAL
PARTS	P#	1	P1
PARTS	P#	2	P2
PARTS	P#	3	P3
PARTS	S#	1	S1
PARTS	S#	2	S1
PARTS	S#	3	S2
PARTS	PNAME	1	NUT
PARTS	PNAME	2	BOLT
PARTS	PNAME	3	SCREW
PARTS	QTY	1	300
PARTS	QTY	2	200
PARTS	QTY	3	400

Figure 4. The imploded PARTS-table

Due to the implosion of the database, the application programs that depend on the database for their data, are no longer operational because the database structure does not exist anymore. Hence it is necessary to convert the DML statements to an imploded form. The procedure for the DML statement implosion will be described on the basis of the following two SELECT statements.

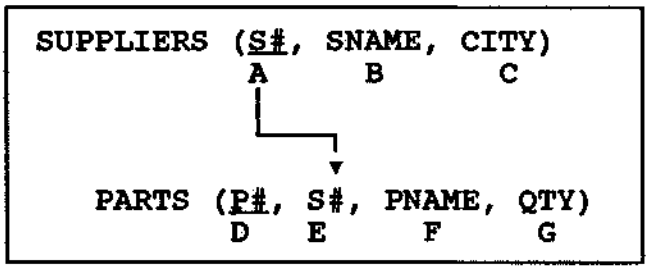
Statement 1

```
SELECT S#, SNAME
FROM SUPPLIERS
WHERE S# IN
      (SELECT S#
       FROM PARTS
       WHERE QTY > 250)
```

Statement 2

```
SELECT P#, PNAME, SNAME
FROM PARTS P, SUPPLIERS S
WHERE P.S# = S.S#
ORDER BY P.S#
```

The following aliases are assigned to the original attributes of the SUPPLIER and PARTS tables:



Aliases of an attribute in a pre-implosion statement do not affect the assignment of aliases mentioned above. The implosion of the DML statements result in statements that are quite alike as far as form is concerned. All statements have similar SELECT and FROM clauses. The WHERE clauses must at least include a structural expression. Tuple-joins are necessary if two or more attributes are mentioned in the statement. Both value-joins and normal expression are used in a common way. Implosion results in the subsequent rearranged equivalent SELECT statements:

Statement 1

```
SELECT A.VAL, B.VAL
FROM VALUE A, VALUE B
WHERE A.ENT_NM = 'SUPPLIERS'  ┐
  AND A.ATT_NM = 'S#'         │  structural
  AND B.ENT_NM = 'SUPPLIERS'  │     expressions
  AND B.ATT_NM = 'SNAME'      ┘
  AND A.TUPLE# = B.TUPLE#  ---------- tuple-join
  AND A.VAL IN  ┐
      (SELECT E.VAL  ---------- value-join
       FROM VALUE E, VALUE G
       WHERE E.ENT_NM = 'PARTS'  ┐
         AND E.ATT_NM = 'S#'     │  structural
         AND G.ENT_NM = 'PARTS'  │     expressions
         AND G.ATT_NM = 'QTY'    ┘
         AND E.TUPLE# = G.TUPLE#  ---------- tuple-join
         AND TO_NUMBER(G.VAL) > 250)  ---- normal expression
```

Statement 2

```
SELECT D.VAL, F.VAL, B.VAL
FROM VALUE A, VALUE B, VALUE D, VALUE E, VALUE F
WHERE A.ENT_NM = 'SUPPLIERS'  ┐
  AND A.ATT_NM = 'S#'         │
  AND B.ENT_NM = 'SUPPLIERS'  │
  AND B.ATT_NM = 'SNAME'      │
  AND D.ENT_NM = 'PARTS'      │  structural
  AND D.ATT_NM = 'P#'         │     expression
  AND E.ENT_NM = 'PARTS'      │
  AND E.ATT_NM = 'S#'         │
  AND F.ENT_NM = 'PARTS'      │
  AND F.ATT_NM = 'PNAME'      ┘
  AND A.TUPLE# = B.TUPLE#  ┐
  AND D.TUPLE# = E.TUPLE#  │  tuple-join
  AND E.TUPLE# = F.TUPLE#  ┘
  AND A.VAL = E.VAL  ---------- value-join
ORDER BY E.VAL  ---------- normal expression
```

Implosion of a database has three important effects on WHERE clauses in the DML statements.
Firstly, conversion from structure to contents leads to a WHERE clause (in the converted DML statements) that contains structural descriptions usually part of the SELECT and FROM clauses. Every tuple in VALUE contains a structural part, that is to say ENT_NM, ATT_NM and TUPLE#. For every column name used in the original statement two structural phrases must be added to the WHERE clause, one containing the ENT_NM , the other one the ATT_NM. In the first original SELECT clause, for example, the column name 'S#' will be replaced by 'A.VAL' in the new SELECT clause and the string

"A.ATT_NM = 'S#'" in the WHERE clause. The table name of the column is removed from the FROM clause and replaced by "A.ENT_NM = 'SUPPLIERS'" in the WHERE clause . When this 'structure shifting' is completed for all columns, the FROM clause will consist of the table name VALUE plus one alias for each column name mentioned in the original statement.

The second effect involves the implicit relation between the existing values of a tuple. As stated before, this implicit relation is lost when implosion takes place and must therefore be replaced by an explicit relation, which can be created by a so called tuple-join. A tuple-join is a join by TUPLE# of two columns in the same table of the original database. It is essential that all columns mentioned are joined by a tuple-join in order to obtain the same outcome as the pre-implosion statement. In the post-implosion statement the tuple-joins have the form 'A.TUPLE# = B.TUPLE#'. In our example the columns 'S#' and 'SNAME' in SUPPLIERS are related by the tuple-join 'A.TUPLE# = B.TUPLE#'. In PARTS the columns 'S#' and 'QTY' are related by the tuple-join 'E.TUPLE# = G.TUPLE#'.

Statement 2, however, is not complete yet. Because of the link between the tables SUPPLIERS and PARTS, realized by 'P.S# = S.S#', a so called value-join is required, constituting the third effect on the WHERE clause. A value-join is a join by VAL of one column appearing in two tables by means of 'A.VAL = B.VAL'. For statement 2 the value-join is 'A.VAL = E.VAL'.

Conversion

The next step in the procedure is the conversion of the imploded supplier-and-parts database and the conversion of the two imploded DML SELECT statements of the preceding section. The conversion of both the database and the statements is specific for the selected EDSOs. The problem of the suppliers-and-parts database system can be described in only one EDSO: 'change a one-to-many relationship into a many-to-many relationship'*. This EDSO creates a implicit many-to-many relationship between SUPPLIERS and PARTS by creating two one-to-many relationships and a new entity, SP (figure 5).

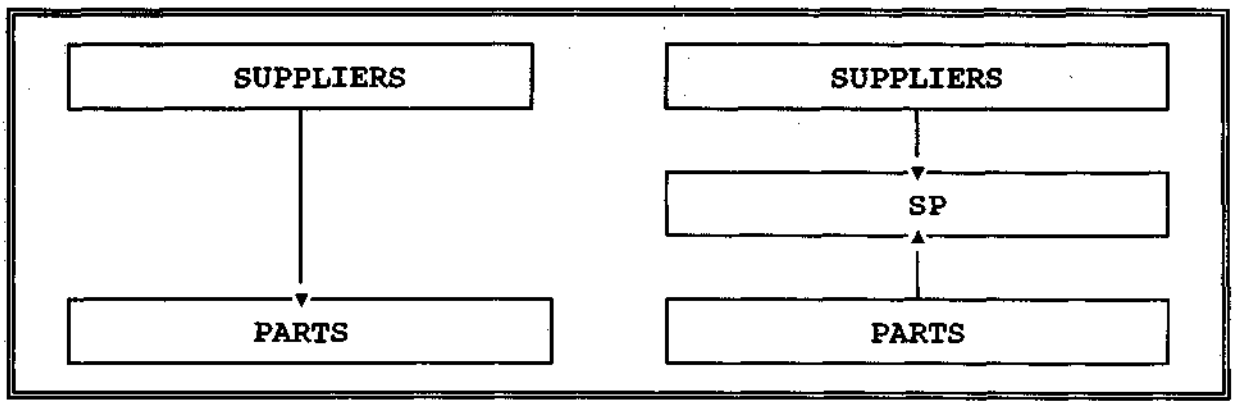


Figure 5. Result of the EDSO

Database conversion

The above EDSO requires the implosion of the table on the 'many-side' of the original one-to-many relationship (i.e. PARTS), because it is the description of this table that will need to be changed**. Conversion of the imploded database for the exemplary EDSO begins by establishing the attributes of the new entity, SP. These attributes are the PK attributes of the two original entities, i.e. 'S#' and 'P#', and become both PK and FK attributes in the SP-table. Then the original PARTS-table is dropped.

The next part of this step consists of the application of two procedures on the VALUE-table. First, a 'rename' procedure is needed in order to change the ENT_NM of tuples having "ATT_NM = 'S#'" into 'SP'. Secondly, a 'copy' procedure will change the ENT_NM of tuples having "ATT_NM = 'P#'" into 'SP'. The result is the VALUE-table below:

---

* This EDSO is also applicable to recursive one-to-many relationships, like, for instance, department hierarchies.

** Execution of the imploded statements on the imploded data dictionary also requires the implosion of the SUPPLIER-table.

ENT_NM	ATT_NM	TUPLE_CD	VAL	
PARTS	P#	1	P1	
PARTS	P#	2	P2	
PARTS	P#	3	P3	
PARTS	PNAME	1	NUT	
PARTS	PNAME	2	BOLT	
PARTS	PNAME	3	SCREW	
PARTS	QTY	1	300	
PARTS	QTY	2	200	
PARTS	QTY	3	400	
SP	S#	1	S1	rename procedure
SP	S#	2	S1	
SP	S#	3	S2	
SP	P#	1	P1	copy procedure
SP	P#	2	P2	
SP	P#	3	P3	

Figure 6. The converted VALUE-table

Conversion of the database ends with the creation of two tables, one with the original table name PARTS, and a completely new one called SP. These tables have the following description:

PARTS (P#, PNAME, QTY).
SP (S#, P#).

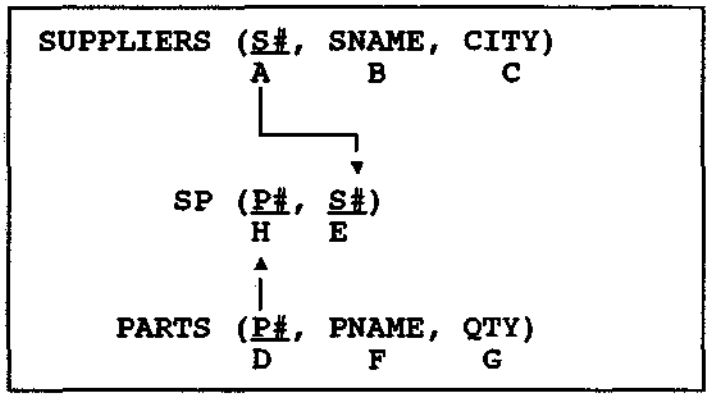
The new database structure is now in imploded form available in the data dictionary.

DML conversion

The exemplary EDSO requires the implosion of the DML statements that contain:

1 in the FROM clauses, the table name of the many-side entity (i.e. 'PARTS') and
2 in other clauses, PK attributes of the one-side entity that are FK in the many-side entity (i.e. 'S#').

The following aliases are assigned to the attributes of the three tables of the new database structure:



Conversion of DML statements for the exemplary EDSO, begins with changing 'PARTS' into 'SP' for structural expressions having "alias.ENT_NM = 'PARTS' AND alias.ATT_NM = 'S#'". Tuple-joins containing the alias of a changed structural expression (i.e. alias E) have to be adjusted by removing the tuple-join or, if the alias occurs in more than one tuple-join, by combining the other aliases of those tuple-joins into (a) new tuple-join(s). For statement 1 this means that "E.ENT_NM = 'PARTS'" becomes "E.ENT_NM = 'SP'" and that the tuple-join 'E.TUPLE# = G.TUPLE#' is removed from the statement.

Furthermore, for every PK attribute of the many-side entity ('PARTS.P#'), the following actions must be taken:

1. an alias must be added to the FROM clause;
2. a structural expression for the new entity and the PK attribute ("alias.ENT_NM = 'SP' AND alias.ATT_NM = 'P#'") and a tuple-join must be added to the WHERE clause;
3. if the structural expression for the many-side entity and the PK attribute already exists ("alias.ENT_NM = 'PARTS' AND alias.ATT_NM = 'P#'"), appending a value-join to the WHERE clause suffices.
   If not, another alias must be added to the FROM clause. Furthermore: a structural expression for the many-side entity and the PK attribute; and a tuple-join and finally a value-join have to be appended to the WHERE clause (if there is only one PK attribute on the many-side entity, this value-join is realized by adding a sub-SELECT statement by means of the logical operator 'IN' to the WHERE clause).

The addition of a value-join is the result of the creation of a new table during the execution of this exemplary EDSO. Because attribute 'S#' has been removed from PARTS and has subsequently been incorporated in SP, the original statements that use 'S#' of the PARTS-table now require a value-join between PARTS and SP in order to have the same effect.

For statement 1 the above actions mean the addition of:

```
1 'VALUE H'
2 "H.ENT_NM = 'SP'"
  "H.ATT_NM = 'P#'"
  'E.TUPLE# = H.TUPLE#'
3 'VALUE D'
  "D.ENT_NM = 'PARTS'"
  "D.ATT_NM = 'P#'"
  'D.TUPLE# = G.TUPLE#'
  'D.VAL = H.VAL'
```

The conversion of the two DML statements, given the chosen EDSO, leads to the following converted statements. The changes as compared to the imploded pre-conversion statements are presented in **bold** face.

Statement 1

```
SELECT A.VAL, B.VAL
FROM VALUE A, VALUE B
WHERE A.ENT_NM = 'SUPPLIERS'
  AND A.ATT_NM = 'S#'
  AND B.ENT_NM = 'SUPPLIERS'
  AND B.ATT_NM = 'SNAME'
  AND A.TUPLE# = B.TUPLE#
  AND A.VAL IN
      (SELECT E.VAL
       FROM VALUE E, VALUE H
       WHERE E.ENT_NM = 'SP'
         AND E.ATT_NM = 'S#'
         AND H.ENT_NM = 'SP'
         AND H.ATT_NM = 'P#'
         AND E.TUPLE# = H.TUPLE#
         AND H.VAL IN
             (SELECT D.VAL
              FROM VALUE D, VALUE G
              WHERE D.ENT_NM = 'PARTS'
                AND D.ATT_NM = 'P#'
                AND G.ENT_NM = 'PARTS'
                AND G.ATT_NM = 'QTY'
                AND D.TUPLE# = G.TUPLE#
                AND TO_NUMBER(G.VAL) > 250))
```

Statement 2

```
SELECT D.VAL, F.VAL, B.VAL
FROM VALUE A, VALUE B, VALUE D, VALUE E, VALUE F, VALUE H
WHERE A.ENT_NM = 'SUPPLIERS'
  AND A.ATT_NM = 'S#'
  AND B.ENT_NM = 'SUPPLIERS'
  AND B.ATT_NM = 'SNAME'
  AND D.ENT_NM = 'PARTS'
  AND D.ATT_NM = 'P#'
  AND F.ENT_NM = 'PARTS'
  AND F.ATT_NM = 'PNAME'
  AND E.ENT_NM = 'SP'
  AND E.ATT_NM = 'S#'
  AND H.ENT_NM = 'SP'
  AND H.ATT_NM = 'P#'
  AND A.TUPLE# = B.TUPLE#
  AND D.TUPLE# = F.TUPLE#
  AND E.TUPLE# = H.TUPLE#
  AND A.VAL = E.VAL
  AND D.VAL = H.VAL
ORDER BY E.VAL
```

Conversion of DML statements is the key issue in the whole structural alteration process. Actual conversion on the imploded form of a DML statement can be automated, because most of the structure of the imploded DML statements has been lost. This leads to statements of a similar form.

Explosion

The last step in the procedure is the explosion of the new database structure in physical database tables and the explosion of the converted DML statements in a 'normal' form.

Database explosion

So far, the newly created tables PARTS and SP are empty. They are to be filled with the tuples from the VALUE-table having ENT_NM 'PARTS' and 'SP' respectively, resulting in:

PARTS

P#	PNAME	QTY
P1	NUT	300
P2	BOLT	200
P3	SCREW	400

SP

S#	S#
S1	P1
S1	P2
S2	P3

The new database structure has now been physically realized. The new suppliers-and-parts database consists of three non-empty tables: SUPPLIERS, PARTS and SP. The relationship between SUPPLIERS and PARTS has been changed into a many-to-many relationship by means of two one-to-many relationships with SP (figure 5).

DML explosion

The last step of the whole alteration process is the explosion of the rearranged DML statements. The shape of the two statements is adjusted to the new physical database structure, resulting in:

Statement 1

```
SELECT S#, SNAME
FROM SUPPLIERS
WHERE S# IN
       (SELECT S#
        FROM SP
        WHERE P# IN
              (SELECT P#
               FROM PARTS
               WHERE QTY > 250))
```

Statement 2:

```
SELECT P.P#, PNAME, SNAME
FROM PARTS P, SP, SUPPLIERS S
WHERE SP.S# = S.S#
  AND SP.P# = P.P#
ORDER BY SP.S#
```

After repositioning the statements in the initial places in the original application program and recompiling the adapted application program, the whole structural alteration process is finished.

Some critical notes and conclusions

The approach described in this article is aimed at eliminating application maintenance due to changes in a database structure. It should be noted, however, that the EDSO procedure is a conceptual one. Although it has been implemented in the described manner for research purposes, implementations that take performance aspects into consideration can be developed using the concepts described in this article.
More important is the need to establish a complete set of EDSOs which make it possible to break up any conceivable DSO. For every EDSO a procedure needs to be developed to change the database structure and the affected DML statements.
So far the EDSO procedure has been limited to SELECT statements, however, it is conceivable to extend the procedure to other kinds of DML statements.

If a complete set of EDSOs can be established a tremendous step forward in improving systems flexibility will have been made. Not only will it be possible to increase the development/maintenance ratio considerably, the whole approach to systems design will change, too. The prototyping approach to system development, for instance, will cease to be a costly one (at least in the short run).

A totally different kind of challenge remains that calls for further research. It will be clear that while it is possible to eliminate all structure from a database by imploding it, more research is needed in order to do the same for DML statements. If it is possible to store the remaining structure of the imploded DML in the data dictionary an even higher degree of flexibility can be achieved*. Perhaps even the definition of a DSO that limits its scope to structural changes that do not alter the information contents of a database can be broadened.

---

* The extent to which the example DML statements has been imploded is the maximum SQL permits. A DML language that uses the information of the data dictionary that supports the EDSO procedure can be even more "structureless".

References

1 **Martin, J** *Computer data-base organization*, Prentice Hall, Englewood Cliffs (1975)

2 **Date, C J** *An introduction to Database systems*, fourth ed., Vol I, Addison-Wesley Publishing Company, Massachussets (1986)

3 **Veldwijk, R J, Boogaard, M and Van Dijk, M V** *Decomposition of structural database alterations (in preparation)*

4 **Noble, H and Abbod, T** 'Meta-rules and the semantic integrity constraints in the database', *Proceedings of the Fifth National Conference on Databases (BNCOD5)*, (1986) pp 126-140

5 **Salvatore, T M and Young-Gul, K** 'Information Resource Management: A metadata perspective', *Journal of management information systems*, Vol 5 No 3 (1989), pp 5-18

6 **Martin, J** *Managing the database environment*, Prentice Hall, Englewood Cliffs (1983)

7 **Ross, R G** *Data dictionaries and data administration: concepts and practice for data resource management*, Amacom, New York (1981)

8 **Nijsen, G M and Halpin, T A** *Conceptual schema and relational database design*, Prentice Hall, Sydney (1989)

9 **Lek, H van der and Buitendijk, R B** "Is 'CREATE TABLE' nodig ?", *Informatie*, No 10 (1989), pp 721-804