Knowledge Transfer in Database Reverse Engineering
A Supporting Case Study

J-L. Hainaut, J-M. Hick, J. Henrard, D. Roland, V. Englebert
Institut d'Informatique, University of Namur, rue Grandgagnage, 21 - B-5000 Namur
jlh@info.fundp.ac.be

Abstract. *While the database reverse engineering problems and solving processes are getting more and more mastered, the academic community is facing the complex problem of knowledge transfer, both in university and industrial contexts. This paper addresses one efficient support of this transfer, namely academic case studies, i.e., small, clean, self-contained applications exhibiting representative problems and appropriate solutions that can be mastered in a limited time. First it recalls the basics of a generic methodology for database reverse engineering comprising two main steps, namely data structure extraction and data structure conceptualization. Then it describes a small academic case study which has been used for several years both as an illustration of the principles of DBRE and as an exercise aimed at academic and industrial students.*

Keywords: database, reverse engineering, methodology, knowledge transfer, education

1. Introduction

Database reverse engineering is a software engineering process through which the analyst tries to understand and redocument the files and/or the database of an application. More precisely, this process is to yield the complete logical schema and the conceptual schema of the database. The problem is particularly complex when old, ill-designed and poorly (if at all) documented applications are processed. The most frequent problems have been identified [1, 2, 6, 12, 13] and can be classified as follows.

- *Weakness of the DBMS models*. The technical model provided by the data management systems, such as CODASYL-like systems, standard file managers and IMS DBMS, can express only a small subset of the structures and constraints of the intended conceptual schema. In favorable situations, these discarded constructs are managed in procedural components of the application: programs, dialog procedures, triggers, etc., and can be recovered through procedural analysis.
- *Implicit structures*. Such constructs have *intentionally* not been explicitly declared in the DDL (i.e., Data Definition Language) specification of the database.
- *Optimized structures*. For technical reasons, such as time and/or space optimization, many database structures include non semantic constructs. In addition, redundant and unnormalized constructs are added to try to get better response time.
- *Awkward design*. Not all databases were built by experienced designers. Novice and untrained developers, generally unaware of database theory and database methodology, can produce poor or even wrong structures.
- *Obsolete constructs*. Some parts of a database can be abandoned, and ignored by the current programs.
- *Cross-model influence*. The cultural background of designers can lead to very peculiar results. For instance, some relational databases are actually straightforward translations of IMS databases, of COBOL files or of spreadsheets [2].

The Database Research Group of the University of Namur has proposed a general methodology for tackling these problem [5], and has developed a generic CASE tool to support reverse engineering processes [10].

The problem of training developers in reverse engineering techniques is a complex one [14]. Indeed, while numerous books and case studies have been published on software engineering methodologies, including information systems and databases, practically nothing exists for reverse engineering, despite the strategic importance for today's and future large system evolution. As we experienced it, and beyond the problems due to the bad image of reverse engineering (cost, low priority, unattractive languages, etc.), the difficulty is twofold: first to make practitioners aware of the complexity and (though) of the tractability of the problem, and secondly, when they have been *convinced*, to train them in reverse engineering techniques and methods.

It quickly appears that actual materials (see [10] for instance), i.e., raw reports on real reverse engineering projects, are inappropriate for training unless they have been deeply reworked, cleaned, condensed and sometimes enriched. In some cases, we spent much more time to develop usable training material (in the form of annotated *case studies*), than we spent to solve the actual industrial problem this material derives from.

One of the working groups organized during the 1996 WCRE was dedicated to reverse engineering education.

This paper can be considered as a formal contribution to this concern. It presents a (very) small study, inspired by some aspects found in real projects, on which we base our introductory and advanced training modules. Despite its small size (less than 300 COBOL LOC), it provides good support (1) for convincing actions (*DBRE is not as simple as advocated in CASE tool advertisements*, **but** *is solvable*),

(2) for illustrating specific reasonings and tools and (3) as a non trivial application exercise intended for novice *reverse engineers*. It is interesting to note that the case will be solved in one hour with the first objective in mind, two hours for the second one and four hours for the last one.

In this paper, we first recall the main processes of the methodology, then we develop the small case study which it illustrates. This work is a part of the DB-MAIN[1] project, dedicated to Database Application Evolution and Maintenance [7].

2. A Generic Methodology for Database Reverse Engineering

The problems that arise when one tries to recover the documentation of the data naturally fall into two categories that are addressed by the two major processes in DBRE, namely *data structure* extraction and *data structure conceptualization*. By *naturally*, we mean that these problems relate to the recovery of two different schemas, and that they require quite different concepts, reasonings and tools. In addition, each of these processes appears as the reverse of a standard database design process (resp. physical and logical design). Since this methodology has been described in earlier papers [5, 10], we will only recall some of its processes and the problems they try to solve. Its general architecture is outlined in Fig 1.

2.1 The Data Structure Extraction Process

This phase consists in recovering the complete DMS[2] schema, including all the implicit and explicit structures and constraints. True database systems generally supply, in some readable and processable form, a description of this schema (data dictionary contents, DDL texts, etc.). Though essential information may be missing, this first-cut schema is a rich starting point that can be refined through further analysis of the other components of the application (views, subschemas, screen and report layouts, procedures, fragments of documentation, database content, program execution, etc.).

The problem is much more complex for standard files, for which no computerized description of their structure exists in most cases. The analysis of each source program provides a partial view of the file and record structures only. For most real-world applications, this analysis must go well beyond the mere detection of the record structures declared in the programs, as will be shown in the case study (and as discussed in e.g., [1, 2, 9, 12]).

The main processes of DATA STRUCTURE EXTRACTION are the following:

- DMS-DDL text ANALYSIS. This rather straightforward process consists in analyzing the data structure declaration statements (in the specific DDL) included in the schema scripts and application programs. It produces a first-cut logical schema.
- SCHEMA REFINEMENT. Non declarative sources of information are analyzed in order to elicit implicit constructs and constraints. This process has been particularly studied in [9].
  - PROGRAM ANALYSIS. It consists in analyzing the procedural sections of the application programs in order to detect evidences of additional data structures and integrity constraints. The first-cut schema can therefore be refined through the detection of hidden, non declared structures.
  - DATA ANALYSIS. This refinement process examines the contents of the files and databases in order (1) to detect data structures and properties (e.g., to find the unique fields or the functional dependencies in a file), and (2) to test hypotheses (e.g., "could this field be a foreign key to this file?").
  - ANALYSIS of other sources (screens, reports, etc.).

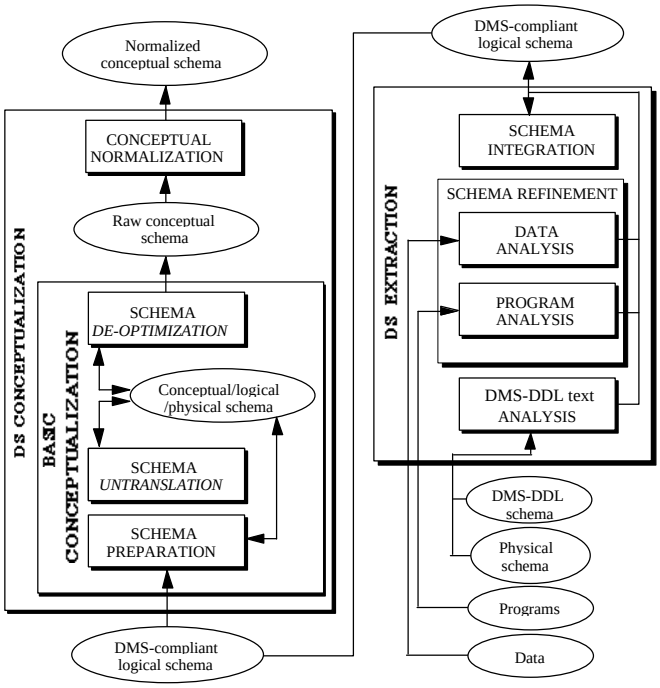

Fig. 1 - Main processes of the generic DBRE methodology.

- SCHEMA INTEGRATION. When more than one information source has been processed, the analyst is provided with several, generally different, extracted (and possibly refined) schemas. The final logical schema must include the specifications of all these partial views, through a *schema integration* process.

1 This research is a part of the DB-MAIN project, which is partly supported by a consortium comprising ACEC-OSI (Be), ARIANE-II (Be), Banque UCL (Lux), BBL (Be), Cap Gemini (Lux), Centre de recherche public H. Tudor (Lux), Cliniques Univ. St-Luc (Be), CGER (Be), Cockerill-Sambre (Be), CONCIS (Fr), D'Ieteren (Be), DIGITAL (Be), EDF (Fr), EPFL (CH), GEDIS (Be), Groupe S (Be), IBM (Be), OBLOG Software (Port), ORIGIN (Be), TEC Charleroi (Be), Ville de Namur (Be), Winterthur (Be), 3 Suisses (Be). The DB-PROCESS subproject is supported by the *Communauté Française de Belgique*.

2 A Data Management System (DMS) is either a File Management System (FMS) or a Database Management System (DBMS).

The end product of this phase is the (hopefully) complete logical schema. This schema is expressed according to the specific model of the DMS, and still includes possible optimized constructs.

2.2 The Data Structure Conceptualization Process

This second phase addresses the conceptual interpretation of the DMS schema. It consists, among others, in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimization and DMS-dependent constructs. It consists of two sub-processes, namely *Basic conceptualization* and *Conceptual normalization*.

- BASIC CONCEPTUALIZATION. The main objective of this process is to extract all the relevant semantic concepts underlying the logical schema. Two different problems, requiring different reasonings and methods, have to be solved: *schema untranslation* and *schema de-optimization*. However, before tackling these problems, we often have to *prepare* the schema by cleaning it.
  - *SCHEMA PREPARATION*. The schema still includes some constructs, such as files and access keys, which may have been useful in the Data Structure Extraction phase, but can now be discarded. In addition, translating names to make them more meaningful, and restructuring some parts of the schema can prove useful before trying to interpret them.
  - *SCHEMA UNTRANSLATION*[3]. The logical schema is the technical translation of the conceptual constructs to be retrieved. Through this process, the analyst identifies the traces of such translations, and replaces them by their original conceptual constructs.
  - *SCHEMA DE-OPTIMIZATION*. The logical schema is searched for traces of constructs designed for optimization purposes.
- CONCEPTUAL NORMALIZATION. This process restructures the basic conceptual schema in order to give it the desired qualities one expects from any final conceptual schema, e.g., expressiveness, simplicity, minimality, readability, genericity, extensibility, compliance with corporate standards. For instance, some entity types are replaced by relationship types or by attributes, is-a relations are made explicit, names are standardized, etc.

3. Introduction to the case study

The next section describes an academic case study which is presented as a part of a migration project. A set of COBOL file structures is converted into relational table structures in a systematic way. The source data structures appear in a small COBOL program which uses three files. The objective of the exercise is to produce a relational schema which translates as faithfully as possible the semantics of those source files. This can be done in two steps: first we elaborate a conceptual schema of the three files, then we translate this schema into relational structures. Due to space limit, we will simplify the relational translation, that is now considered standard [16, 17, 18]. For the same reason, we will develop the first main process (Data Structure Extraction) of the reverse engineering step in more detail than the second one (Data Structure Conceptualization), which has been more extensively treated in the literature [1, 3, 4, 6, 11, 13].

As expected for a small case study, not all the problems and reasonings will be illustrated. Hence the need for a library of case studies.

4. The DATA STRUCTURE EXTRACTION process

The only source of information that will be considered is the COBOL program listed in Appendix 1. Though the problem can be solved with the help of the DB-MAIN CASE tool [10], we will mainly emphase the reasonings rather than tool usage.

4.1 DMS-DDL text ANALYSIS

This operation is carried out by a *COBOL parser* which extracts the file and record types descriptions, and expresses them as a *first cut schema* in the repository. The resulting schema is given in Fig. 2.

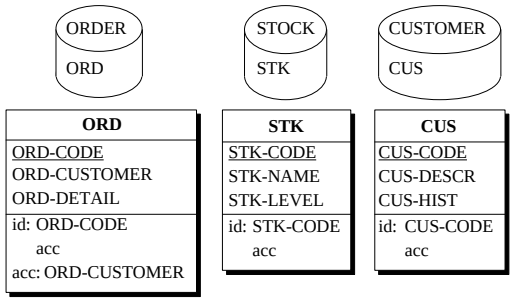

Fig. 2 - The first cut file and record schema.

The textual view shows the type and length of the fields:

```
Schema CUST-ORD/1stCut-Logical

collection CUSTOMER
 CUS
collection ORDER
 ORD
collection STOCK
 STK

entity type CUS
     in CUSTOMER
  CUS-CODE char(12)
  CUS-DESCR char(80)
  CUS-HIST char(1000)
  id:CUS-CODE
     access key

entity type ORD
     in ORDER
  ORD-CODE num(5)
  ORD-CUSTOMER char(12)
  ORD-DETAIL char(200)
  id:ORD-CODE
     access key
  access key:ORD-CUSTOMER

entity type STK
     in STOCK
  STK-CODE num(5)
  STK-NAME char(100)
  STK-LEVEL num(5)
  id:STK-CODE
     access key
```

Each record type is represented by a *physical entity type*, and each field by a *physical attribute*. Record keys are represented by *identifiers* when they specify uniqueness constraints and by *access keys* when they specify indexes.

[3] Any suggestion for a more correct term is welcome!

Files are represented as *physical entity collections* (cylinder icons).

4.2 SCHEMA REFINEMENT

The schema will be refined through an in-depth inspection of the ways in which the program uses and manages the data. Through this process, we will detect additional structures and constraints which were not explicitly declared in the file/record declaration sections, but which were expressed in the procedural code and in local variables. We will consider four important processes, namely *Field refinement*, *Foreign key elicitation*, *Attribute identifier elicitation*, and *Field cardinality refinement*. The baselines of these processes have been developed in [9].

4.2.1 Field refinement

Observation: some fields are unusually long (CUS-DESCR, CUS-HIST, ORD-DETAIL, STK-NAME). Could they be further refined? Let us consider CUS-DESCR first. We build the variable dep*endency graph*, which summarizes the dataflow concerning CUS-DESCR (statements <1> and <1'>):

CUS.CUS-DESCR —<1'>→ DESCRIPTION
CUS.CUS-DESCR ←<1>— DESCRIPTION

This graph clearly suggests that CUS-DESCR and DESCRIPTION share the same values, and should have the same structure as well, i.e.:

```
01 DESCRIPTION.
   02 NAME PIC X(20).
   02 ADDRESS PIC X(40).
   02 FUNCTION PIC X(10).
   02 REC-DATE PIC X(10).
```

This structure is associated with the field CUS-DESCR in the logical schema. We proceed in the same way for CUS-HIST, ORD-DETAIL and STK-NAME (Fig. 3). The analysis shows that the last one need not be refined.

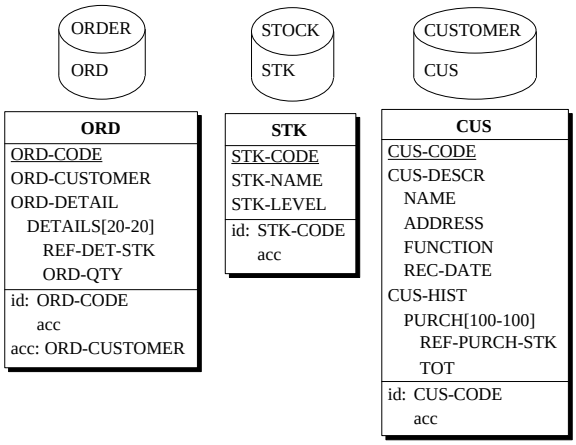

Fig. 3 - Result of the field refinement process.

4.2.2 Foreign key elicitation

There should exist reference links among these record types. Let us examine the field ORD-CUSTOMER for instance. We observe that:

- its name includes the name of a file (CUSTOMER);
- it has the same type and length as the record key of CUSTOMER;
- it is supported by an access key (i.e., an index);
- its dependency graph shows that it receives its values from the record key of CUSTOMER:

CUS.CUS-CODE —<4>→ ORD.ORD-CUSTOMER

- its usage pattern shows (through a *program slice* [15]) that, before moving it to the ORD record to be stored, the program verifies that ORD-CUSTOMER value identifies a stored CUS record

```
NEW-ORD.
  ...
  MOVE 1 TO END-FILE.
  PERFORM READ-CUS-CODE UNTIL END-FILE = 0.
  ...
  MOVE CUS-CODE TO ORD-CUSTOMER.
  ...
  WRITE ORD INVALID KEY DISPLAY "ERROR".

READ-CUS-CODE.
  ACCEPT CUS-CODE.
  MOVE 0 TO END-FILE.
  READ CUSTOMER INVALID KEY
    DISPLAY "NO SUCH CUSTOMER"
    MOVE 1 TO END-FILE
  END-READ.
```

These are five positive evidences contributing to asserting that ORD-CUSTOMER *certainly is* a foreign key.

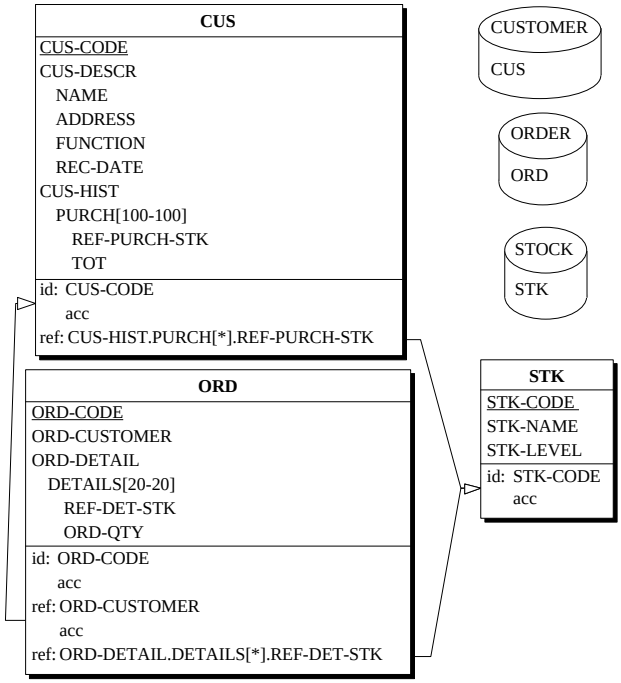

Fig. 4 - The foreign keys are made explicit.

Data analysis could have added additional information. We decide to confirm the hypothesis. In the same way, we conclude that:

- ORD-DETAIL.DETAILS.REF-DET-STK is a multi-valued foreign key to STOCK. Here the REF part of the name suggests a referential function of the field.
- CUS-HIST.PURCH.REF-PURCH-STK is a multivalued foreign key to STOCK.

Now the schema looks like that in Fig. 4.

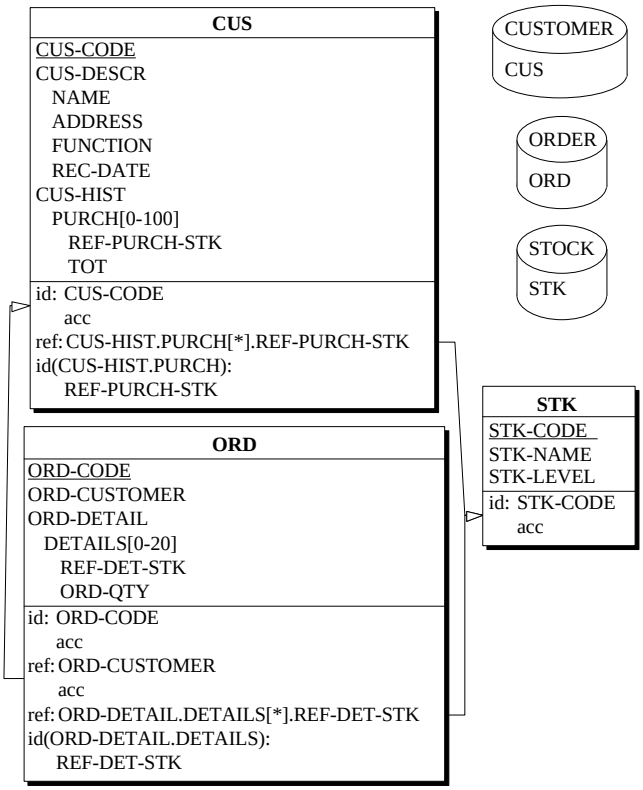

Fig. 5 - The (refined) COBOL-compliant logical schema.

4.2.3 Elicitation of identifiers of multivalued fields

Compound multivalued fields in COBOL records often have an implicit identifier that makes their values unique. The schema includes two candidate multivalued fields: ORD-DETAIL.DETAILS and CUS-HIST.PURCH.

By examining the way in which these fields are searched and managed, we isolate the following pattern (or *program slice* [15]):

```
  SET IND-DET TO 1.
UPDATE-ORD-DETAIL.
  MOVE 1 TO NEXT-DET.
  ...
  PERFORM UNTIL
       REF-DET-STK(NEXT-DET) = PROD-CODE
       OR IND-DET = NEXT-DET
    ADD 1 TO NEXT-DET
  END-PERFORM.
  IF IND-DET = NEXT-DET
    MOVE PROD-CODE TO REF-DET-STK(IND-DET)
    PERFORM UPDATE-CUS-HIST
    SET IND-DET UP BY 1
  ELSE
    DISPLAY "ERROR: ALREADY ORDERED".
```

It derives from this code section that the LIST-DETAIL.DETAILS array will never include twice the same REF-DET-STK value. Therefore, this field is the local identifier of this array, and of ORD-DETAIL.DETAILS as well. Through the same reasoning, we are suggested that REF-PURCH-STK is the identifier of LIST-PURCHASE.PURCH array. These findings are shown in Fig. 5.

4.2.4 Refinement of the cardinality of multivalued attributes

The multivalued fields have been given cardinality constraints derived from the `occurs` clause. The latter gives the maximum cardinality, but says nothing about the minimum cardinality.

Storing a new CUS record generally implies initializing each field, including CUS-HIST.PURCH. This is done through the INIT-HIST paragraph (line <13>), in which the REF-DET-STK is set to 0. Furthermore, the scanning of this list stops when 0 is encountered (line <7>). The conclusion is clear: there are from 0 to 100 elements in the list. A similar analysis leads to refine the cardinality of ORD-DETAIL.DETAILS). Hence the final schema of Fig. 5.

5. The DATA STRUCTURE CONCEPTUALIZATION process

5.1 Schema preparation

The schema obtained so far describes the complete COBOL data structures. Before trying to recover the conceptual schema, we will clean the current schema (Fig. 6).

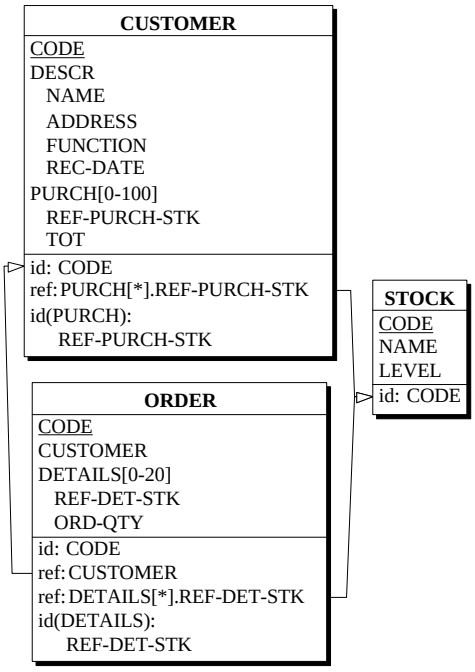

Fig. 6 - The (cleaned) COBOL-compliant logical schema.

Name processing. The files have more meaningful names than their record types: we give the latter the name of their files. The fields of each record type are prefixed with a common short-name identifying their record type. This is a common programming trick that provides shorter names but adds no semantics. We trim them out.

Unneeded compound fields. Compound fields CUS.CUS-HIST and ORD.ORD-DETAIL have one component only, and can be disaggregated without structural or semantic loss.

Physical cleaning. The physical constructs, namely files and access keys, are no longer useful, and are removed.

5.2 Schema de-optimization

The attributes CUSTOMER.PURCH and ORDER.DETAILS have a complex structure: they are compound, they are multivalued, they have a local identifier and they include a foreign key. They obviously suggest a typical COBOL trick to represent *dependent entity types*. This very efficient technique consists in representing such entity types by embedded multivalued fields. We transform the latter into entity types as in Fig. 7.

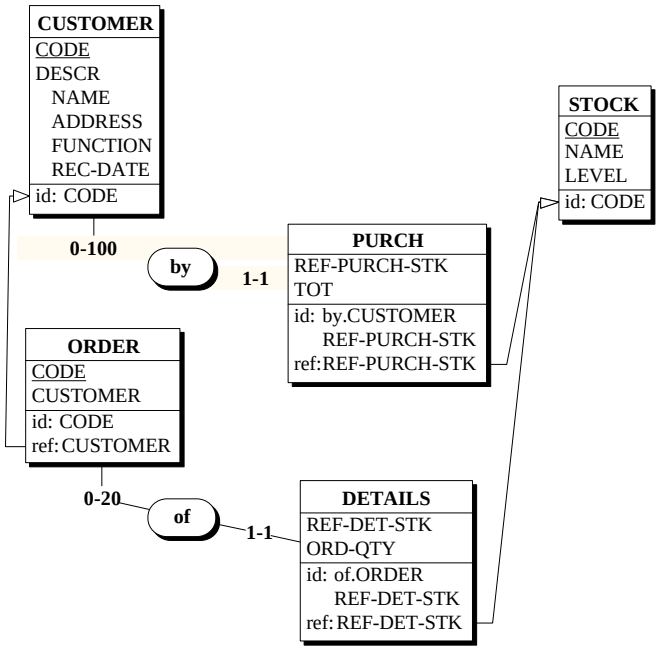

Fig. 7 - Making dependent entity types explicit.

5.3 Schema untranslation

The foreign keys are the most obvious traces of the ER/COBOL translation. We express them as *one-to-many* relationship types (Fig. 8).

5.4 Conceptual normalization

We will only mention four elementary problems to illustrate the process (Fig. 9).

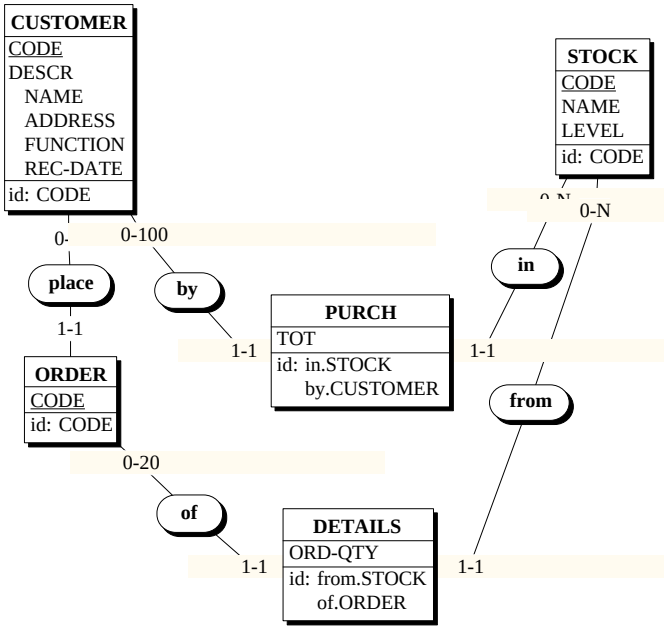

Fig. 8 - Untranslating foreign keys into relationship types.

Maximal cardinalities Are the maximum cardinalities 100 and 20 of real semantic value, or do they simply describe obsolete technical limits from the legacy system? Considering their origin these constraints are relaxed, and replaced with "N", meaning *no-limit*.

Rel-type entity types. PURCH and DETAILS could be perceived as mere relationships, and are transformed accordingly.

Names. Now the semantics of the data structures have been elicited, and better names can be given to some of them. For instance, PURCH is given the full-name ***purchase***.

Unneeded aggregation. Attribute DESCR in CUSTOMER is an artificial aggregate which can be dismantled without loss of semantics.

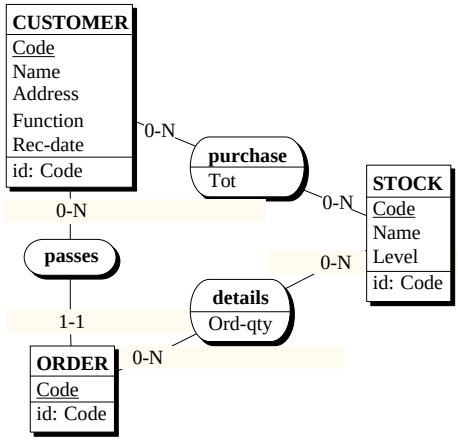

Fig. 9 - A normalized version of the conceptual schema.

6. Relational database design

To complete the exercice, let us develop a new relational database schema from the conceptual specifications. The process is fairly standard, and includes the Logical design and the Physic*al design* phases. Due to the size of the problem, they are treated in a rather symbolic way.

6.1 Logical design

Transforming this schema into relational structures is fairly easy: we express the complex relationship types detail and purchase into entity types, then we translate the one-to-many relationship types into foreign keys. The resulting schema comprises flat entity types, identifiers and foreign keys. It can be considered a logical relational schema (Fig. 10).

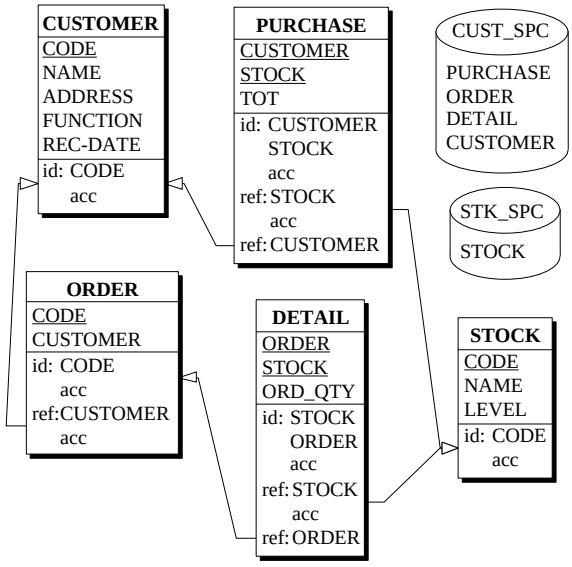

Fig. 10 -The physical relational schema.

6.2 Physical design

We reduce this phase to processing the names according to SQL standard (e.g., all the names in uppercase, no "-", no reserved words, etc.) and defining the physical spaces and the access keys (indexes) which support identifiers and foreign keys (Fig. 10). As a symbolic touch of optimization, we remove all the indexes which are a prefix of another index (i.e., no index on PURCHASE.CUSTOMER and on DETAILS.STOCK). An SQL script that encodes this schema is proposed in Appendix 2.

7. Conclusions

This case study was a toy application only, designed for training purposes. A typical, medium size, real-world project could included, for instance:

- 250 program units, with a size ranging from 100 to 20,000 lines;
- 1,000 files, 50 of which being relevant;
- these 50 files include 100 record types and 3,000 fields;
- more than one DMS (for instance COBOL files + an IMS database);
- more sophisticated constructs such as: alternate field structures, overlapping foreign keys, transitive and embedded foreign keys, conditional identifiers and foreign keys, field redundancies, explicit NEXT pointers [2,13];
- conflicting structures and views
- usage of data dictionaries, CASE tools, generators.

It is important to notice that we have translated the data structures only. We have to mention two additional problems: converting the COBOL data into relational data, and converting the COBOL programs into COBOL/SQL programs. These problems have been addressed in [7, 8].

The small reverse enginering project presented in this paper was used as a support to Database Engineering lectures given in the University of Namur (and in some foreign universities as well). It was also used during industrial training seminars, ranging from mere information to intensive technical formation. In each case, it proved quite accessible, and representative of actual problems (except for the size). It is generally complemented by reports on real size applications which illustrate the scale effect of large projects. We intend to develop other such case studies to cover the other aspects, not only of reverse engineering, but also of re-engineering as a whole.

While we did not gather enough formal feedback information on the approach, the use of a CASE tool has proved a motivating aspect of educational case studies. However, except for very simple applications, such a tool requires its own training. Since some functions are rather complex (e.g., text pattern-matching and program slicing), we had to limit their use to simple aspects only.

Note. An *Education version* of the DB-MAIN CASE tool and this case study are available for non-profit organizations at `http://www.info.fundp.ac.be/~dbm`

8. References

[1] Andersson, M. 1994. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[2] Blaha, M.R., Premerlani, W., J. 1995. Observed Idiosyncracies of Relational Database designs, in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press

[3] Casanova, M., A., Amaral De Sa. 1984. Mapping uninterpreted Schemes into Entity-Relationship diagrams: two applications to conceptual schema design, in *IBM J. Res. & Develop*., Vol. 28, No 1

[4] Davis, K., H., Arora, A., K. 1985. A Methodology for Translating a Conventional File System into an Entity-Relationship Model, in *Proc. of ERA*, IEEE/North-Holland

[5] Hainaut, J-L., Chandelon M., Tonneau C., Joris M. 1993a. Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press

[6] Hainaut, J-L, Chandelon M., Tonneau C., Joris M. 1993b. Transformational techniques for database reverse engineering,

in *Proc. of the 12th Int. Conf. on ER Approach, Arlington-Dallas*, E/R Institute and Springer-Verlag, LNCS

[7] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D. 1994. Evolution of database Applications: the DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[8] Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. 1996b. Database design recovery, *in Proc. of CAiSE·96*, Springer-Verlag, 1996

[9] Hainaut, J-L, Henrard, J., Roland, D.., Englebert, V., Hick J-M, 1996. Structure Elicitation in Database Reverse Engineering, *Proc. of the IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press

[10] Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. 1996c. Database Reverse Engineering: from Requirements to CARE tools, *Journal of Automated Software Engineering*, Vol. 3, No. 1 (1996).

[11] Nilsson,E., G. 1985. The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema, in *Proc. of ERA Conference*, IEEE/North-Holland,

[12] Petit, J-M., Kouloumdjian, J., Bouliaut, J-F., Toumani, F. 1994. Using Queries to Improve Database Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[13] Premerlani, W., J., Blaha, M.R. 1993. An Approach for Reverse Engineering of Relational Databases, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, IEEE Computer Society Press

[14] Selfridge, P., G., Waters, R., C., Chikofsky, E., J. 1993. Challenges to the Field of Reverse Engineering, in *Proc. of the 1st WC on Reverse Engineering*, pp.144-150, IEEE Computer Society Press

[15] Weiser, M. 1984. Program Slicing, *IEEE TSE*, Vol. 10, pp 352-357

[16] Batini, C., Ceri, S., Navathe, S., B. 1992. *Conceptual Database Design*, Benjamin/ Cummings

[17] Elmasri, R., Navathe, S. 1994/1997. *Fundamentals of Database Systems*, Benjamin-Cummings

[18] Teorey, T. J. 1994. *Database Modeling and Design: the Fundamental Principles*, Morgan Kaufmann

Appendix 1. The COBOL source text

```
IDENTIFICATION DIVISION.
PROGRAM-ID. C-ORD.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CUSTOMER ASSIGN
         TO "CUSTOMER.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS CUS-CODE.
  SELECT ORDER ASSIGN TO "ORDER.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS ORD-CODE
    ALTERNATE RECORD KEY
          IS ORD-CUSTOMER
    WITH DUPLICATES.
  SELECT STOCK ASSIGN TO "STOCK.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS STK-CODE.

DATA DIVISION.
FILE SECTION.
FD CUSTOMER.
01 CUS.
  02 CUS-CODE PIC X(12).
  02 CUS-DESCR PIC X(80).
  02 CUS-HIST PIC X(1000).
FD ORDER.
01 ORD.
  02 ORD-CODE PIC 9(10).
  02 ORD-CUSTOMER PIC X(12).
  02 ORD-DETAIL PIC X(200).
FD STOCK.
01 STK.
  02 STK-CODE PIC 9(5).
  02 STK-NAME PIC X(100).
  02 STK-LEVEL PIC 9(5).

WORKING-STORAGE SECTION.
01 DESCRIPTION.
  02 NAME PIC X(20).
  02 ADDRESS PIC X(40).
  02 FUNCTION PIC X(10).
  02 REC-DATE PIC X(10).

01 LIST-PURCHASE.
  02 PURCH OCCURS 100 TIMES
     INDEXED BY IND.
    03 REF-PURCH-STK PIC 9(5).
    03 TOT PIC 9(5).

01 LIST-DETAIL.
  02 DETAILS OCCURS 20 TIMES
     INDEXED BY IND-DET.
    03 REF-DET-STK PIC 9(5).
    03 ORD-QTY PIC 9(5).

01 CHOICE PIC X.
01 END-FILE PIC 9.
01 END-DETAIL PIC 9.
01 EXIST-PROD PIC 9.
01 PROD-CODE PIC 9(5).

01 TOT-COMP PIC 9(5) COMP.
01 QTY PIC 9(5) COMP.
01 NEXT-DET PIC 99.

PROCEDURE DIVISION.
MAIN.
  PERFORM INIT.
  PERFORM PROCESS UNTIL CHOICE = 0.
  PERFORM CLOSING.
  STOP RUN.
INIT.
  OPEN I-O CUSTOMER.
  OPEN I-O ORDER.
  OPEN I-O STOCK.
```

```
PROCESS.
  DISPLAY "1 NEW CUSTOMER".
  DISPLAY "2 NEW STOCK".
  DISPLAY "3 NEW ORDER".
  DISPLAY "4 LIST OF CUSTOMERS".
  DISPLAY "5 LIST OF STOCKS".
  DISPLAY "6 LIST OF ORDERS".
  DISPLAY "0 END".
  ACCEPT CHOICE.
  IF CHOICE = 1
    PERFORM NEW-CUS.
  IF CHOICE = 2
    PERFORM NEW-STK.
  IF CHOICE = 3
    PERFORM NEW-ORD.
  IF CHOICE = 4
    PERFORM LIST-CUS.
  IF CHOICE = 5
    PERFORM LIST-STK.
  IF CHOICE = 6
    PERFORM LIST-ORD.

CLOSING.
  CLOSE CUSTOMER.
  CLOSE ORDER.
  CLOSE STOCK.

NEW-CUS.
  DISPLAY "NEW CUSTOMER:".
  DISPLAY "CUSTOMER CODE?"
    WITH NO ADVANCING.
  ACCEPT CUS-CODE.

  DISPLAY "NAME DU CUSTOMER: "
    WITH NO ADVANCING.
  ACCEPT NAME.
  DISPLAY "ADDRESS OF CUSTOMER: "
    WITH NO ADVANCING.
  ACCEPT ADDRESS.
  DISPLAY "FUNCTION OF CUSTOMER: "
    WITH NO ADVANCING.
  ACCEPT FUNCTION.
  DISPLAY "DATE: " WITH NO ADVANCING.
  ACCEPT REC-DATE.
  MOVE DESCRIPTION TO CUS-DESCR.      <1>
  PERFORM INIT-HIST.
  WRITE CLI INVALID KEY DISPLAY "ERROR".


LIST-CUS.
  DISPLAY "LISTE DES CUSTOMERS".
  CLOSE CUSTOMER.
  OPEN I-O CUSTOMER.
  MOVE 1 TO END-FILE.
  PERFORM READ-CUS UNTIL END-FILE = 0.

READ-CUS.
  READ CUSTOMER NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY CUS-CODE
      DISPLAY CUS-DESCR
      DISPLAY CUS-HISTORY.

NEW-STK.
  DISPLAY "NEW STOCK".
  DISPLAY "PRODUCT NUMBER: "
    WITH NO ADVANCING.
  ACCEPT STK-CODE.

  DISPLAY "NAME: " WITH NO ADVANCING.
  ACCEPT STK-NAME.

DISPLAY "LEVEL: " WITH NO ADVANCING.
  ACCEPT STK-LEVEL.

WRITE STK INVALID KEY DISPLAY "ERROR ".

LIST-STK.
  DISPLAY "LIST OF STOCKS ".

  CLOSE STOCK.
  OPEN I-O STOCK.

  MOVE 1 TO END-FILE.
  PERFORM READ-STK UNTIL END-FILE = 0.

READ-STK.
  READ STOCK NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY STK-CODE
      DISPLAY STK-NAME
      DISPLAY STK-LEVEL.

NEW-ORD.
  DISPLAY "NEW ORDER".
  DISPLAY "ORDER NUMBER: "
    WITH NO ADVANCING.
  ACCEPT ORD-CODE.

  MOVE 1 TO END-FILE.
  PERFORM READ-CUS-CODE
          UNTIL END-FILE = 0.
  MOVE CUS-DESCR TO DESCRIPTION.     <1'>
  DISPLAY NAME.
  MOVE CUS-CODE TO ORD-CUSTOMER.     <4>
  MOVE CUS-HISTORY TO LIST-PURCHASE.

  SET IND-DET TO 1.
  MOVE 1 TO END-FILE.
  PERFORM READ-DETAIL
    UNTIL END-FILE = 0 OR IND-DET = 21.
  MOVE LIST-DETAIL TO ORD-DETAIL.    <2>

  WRITE COM INVALID KEY DISPLAY "ERROR".

  MOVE LIST-PURCHASE
    TO CUS-HISTORY.    <3>
  REWRITE CLI
    INVALID KEY DISPLAY "ERROR CUS".
READ-CUS-CODE.
  DISPLAY "CUSTOMER NUMBER: "
    WITH NO ADVANCING.
  ACCEPT CUS-CODE.
  MOVE 0 TO END-FILE.
  READ CUSTOMER INVALID KEY
    DISPLAY "NO SUCH CUSTOMER"
    MOVE 1 TO END-FILE
  END-READ.

READ-DETAIL.
  DISPLAY "PRODUCT CODE (0 = END): ".
  ACCEPT PROD-CODE.
  IF PROD-CODE = "0"
    MOVE 0
      TO REF-DET-STK(IND-DET) <12>
    MOVE 0 TO END-FILE
  ELSE
    PERFORM READ-PROD-CODE.

READ-PROD-CODE.
  MOVE 1 TO EXIST-PROD.
  MOVE PROD-CODE TO STK-CODE. <5>
  READ STOCK INVALID KEY
    MOVE 0 TO EXIST-PROD.
  IF EXIST-PROD = 0
    DISPLAY "NO SUCH PRODUCT"
  ELSE
     PERFORM UPDATE-ORD-DETAIL.

UPDATE-ORD-DETAIL.
  MOVE 1 TO NEXT-DET.
  DISPLAY "QUANTITY ORDERED: "
    WITH NO ADVANCING
```

```
  ACCEPT ORD-QTY(IND-DET).
  PERFORM UNTIL
    REF-DET-STK(NEXT-DET)
      = PROD-CODE      [9]
    OR IND-DET = NEXT-DET
    ADD 1 TO NEXT-DET
  END-PERFORM.
  IF IND-DET = NEXT-DET       <10>
    MOVE PROD-CODE
    TO REF-DET-STK(IND-DET)   <6>
    PERFORM UPDATE-CUS-HISTO
    SET IND-DET UP BY 1
  ELSE
    DISPLAY "ERROR: ALREADY ORDERED".

UPDATE-CUS-HISTO.
  SET IND TO 1.
  PERFORM UNTIL
    REF-PURCH-STK(IND) = PROD-CODE
    OR REF-PURCH-STK(IND) = 0
    OR IND = 101       <7>
      SET IND UP BY 1
  END-PERFORM.

  IF IND = 101
    DISPLAY "ERR: HISTORY OVERFLOW"
    EXIT.
  IF REF-PURCH-STK(IND)
     = PROD-CODE      <11>
    ADD ORD-QTY(IND-DET) TO TOT(IND)
  ELSE
    MOVE PROD-CODE
    TO REF-PURCH-STK(IND)     <8]
    MOVE ORD-QTY(IND-DET) TO TOT(IND).

LIST-ORD.
  DISPLAY "LIST OF ORDERS ".
  CLOSE ORDER.
  OPEN I-O ORDER.
  MOVE 1 TO END-FILE.
  PERFORM READ-ORD UNTIL END-FILE = 0.

READ-ORD.
  READ ORDER NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY "ORD-CODE "
        WITH NO ADVANCING
      DISPLAY ORD-CODE
      DISPLAY "ORD-CUSTOMER "
        WITH NO ADVANCING
      DISPLAY ORD-CUSTOMER
      DISPLAY "ORD-DETAIL "
      MOVE ORD-DETAIL TO LIST-DETAIL
      SET IND-DET TO 1
      MOVE 1 TO END-DETAIL
      PERFORM DISPLAY-DETAIL.

INIT-HIST.            <13>
  SET IND TO 1.
  PERFORM UNTIL IND = 100
    MOVE 0 TO REF-PURCH-STK(IND)
    MOVE 0 TO TOT(IND)
    SET IND UP BY 1
  END-PERFORM.
  MOVE LIST-PURCHASE TO CUS-HISTORY.

DISPLAY-DETAIL.
  IF IND-DET = 21
    MOVE 0 TO END-DETAIL
    EXIT.
  IF REF-DET-STK(IND-DET) = 0
    MOVE 0 TO END-DETAIL
  ELSE
    DISPLAY REF-DET-STK(IND-DET)
    DISPLAY ORD-QTY(IND-DET)
    SET IND-DET UP BY 1.
```

Appendix 2. The SQL DDL text

```
create database CUS-ORD;
create dbspace CUST_SPC;
create dbspace STK_SPC;
create table CUSTOMER (
  CUS_CODE char(12) not null ,
  CUS_NAME char(20) not null ,
  ADDRESS char(40) not null ,
  FUNCTION char(10) not null ,
  REC-DATE char(10) not null ,
  primary key (CUS_CODE))
     in CUST_SPC;
create table DETAILS (
  ORD_CODE numeric(10) not null,
  STK_CODE numeric(5) not null,
  ORD-QTY numeric(5) not null,
  primary key (STK_CODE,ORD_CODE))
    in CUST_SPC;
create table ORDER (
  ORD_CODE numeric(10) not null,
  CUS_CODE char(12) not null ,
  primary key (ORD_CODE))
   in CUST_SPC;
create table PURCHASE (
  CUS_CODE char(12) not null ,
  STK_CODE numeric(5) not null,
  TOT numeric(5) not null ,
 primary key(STK_CODE,CUS_CODE))
   in CUST_SPC;
create table STOCK (
  STK_CODE numeric(5) not null ,
  STK_NAME char(100) not null ,
  LEVEL numeric(5) not null ,
  primary key (STK_CODE))
   in STK_SPC;
alter table DETAILS add constraint FKDET_STO
    foreign key (STK_CODE) references STOCK;
alter table DETAILS add constraint FKDET_ORD
    foreign key (ORD_CODE) references ORDER;
  alter table ORDER add constraint FKO_C
    foreign key (CUS_CODE) references CUSTOMER;
  alter table PURCH add constraint FKPUR_STO
    foreign key (STK_CODE) references STOCK;
  alter table PURCH add constraint FKPUR_CUS
    foreign key (CUS_CODE) references CUSTOMER;
create unique index CUS-CODE
  on CUSTOMER (CUS_CODE);
create unique index IDDETAILS
  on DETAILS (STK_CODE,ORD_CODE);
create index FKDET_ORD
  on DETAILS (ORD_CODE);
create unique index ORD-CODE
  on ORDER (ORD_CODE);
create index FKO_C
  on ORDER (CUS_CODE);
create unique index IDPURCH
  on PURCHASE (STK_CODE,CUS_CODE);
create index FKPUR_CUS
  on PURCHASE (CUS_CODE);
create unique index STK-CODE
  on STOCK (STK_CODE);
```