repairs that are obtained via direct updates of attribute values (as opposed to deletions followed by insertions, which might not represent a minimal change). In this case, the number of those local changes could be minimized. A different, more general aggregation function of the local changes could be minimized instead (cf. [2,3] for surveys).

Cross-references

► Consistent Query Answering
► Inconsistent Databases

Recommended Reading

1. Arenas M., Bertossi L., and Chomicki J. Consistent query answers in inconsistent databases. In Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1999, pp. 68–79.
2. Bertossi L. Consistent query answering in databases. ACM SIGMOD Rec., 35(2):68–76, 2006.
3. Chomicki J. Consistent query answering: five easy pieces. In Proc. 11th Int. Conf. on Database Theory, 2007, pp. 1–17.

Database Replication

► Data Replication
► Replica Control

Database Reverse Engineering

Jean-Luc Hainaut, Jean Henrard, Vincent Englebert, Didier Roland, Jean-Marc Hick
University of Namur, Namur, Belgium

Synonyms

Database redocumentation; Database design recovery

Definition

Database reverse engineering is the process through which the logical and conceptual schemas of a legacy database, or of a set of files, are reconstructed from various information sources such as DDL code, data dictionary contents, database contents or the source code of application programs that use the database.

Basically, database reverse engineering comprises three processes, namely physical schema extraction, logical schema reconstruction, and schema conceptualization. The first process consists in parsing the DDL code or the contents of an active data dictionary in order to extract the physical schema of the database. Reconstructing the logical schema implies analyzing additional sources such as the data and the source code of the application programs to discover implicit constraints and data structures, that is, constructs that have not been declared but that are managed by the information system or by its environment. The conceptualization process aims at recovering the conceptual schema that the logical schema implements.

Database reverse engineering is often the first step in information system maintenance, evolution, migration and integration.

Historical Background

Database reverse engineering has been recognized to be a specific problem for more than three decades, but has been formally studied since the 1980's, notably in [3,6,12]. The first approaches were based on simple rules, that work nicely with databases designed in a clean and disciplined way. A second generation of methodologies coped with physical schemas resulting from empirical design in which practitioners tend to apply non standard and undisciplined techniques. More complex design rules were identified and interpreted [2], structured and comprehensive approaches were developed [11,7] and the first industrial tools appeared (e.g., Bachman's Reengineering Tool). Many contributions were published in the 1990's, addressing practically all the legacy technologies and exploiting such sources of information as application source code, database contents or application user interfaces. Among synthesis publications, it is important to mention [5], the first tentative history of this discipline.

These second generation approaches were faced with two kinds of problems induced by empirical design [8]. The first problem is the recovery of *implicit constructs*, that is, structures and constraints that have not been explicitly declared in the DDL code. The second problem is that of the *semantic interpretation* of logical schemas that include non standard data structures.

Foundations

The ultimate goal of reverse engineering a piece of software is to recover its functional and technical specifications, starting mainly from the source code of the

programs [4]. The problem is particularly complex with old and ill-designed applications. In this case, there is no documentation to rely on; moreover, the lack of systematic methodologies for designing and maintaining them have led to tricky and obscure code. Therefore, reverse engineering has long been regarded as a complex, painful and failure-prone activity, in such a way that it is simply not undertaken most of the time, leaving huge amounts of invaluable knowledge buried in legacy software, lost for all practical purposes.

In most software engineering cases, analysts have to content themselves with the extraction of abstract and/or partial information, such as call graphs, dependency graphs or program slices in order to ease the maintenance and evolution of the software. The result of reverse engineering a database is more satisfying, in that reconstructing the logical and conceptual schemas of an undocumented database is achievable with reasonable effort.

Database Design Revisited

To understand the problems, challenges and techniques specific to database reverse engineering, it is necessary to reexamine the way databases are developed, both in theory and in practice.

Standard Database Design Methodology Standard database design comprises four formal processes, namely conceptual analysis, logical design, physical design and coding.

Conceptual analysis produces the conceptual schema of the database, that is, an abstract description of the concepts that structure the application domain, of the relationships between these concepts and of the information to be collected and kept about theses classes and relationships. This schema is independent of the application programs that will use the database and is expressed in an abstract formalism such as some variant of the Entity-relationship model. It must be readable, maintainable, normalized and independent of any implementation technology.

Logical design translates the conceptual schema into data structures compliant with the data model of a family of DBMSs. This process is best described by a transformation plan, according to which the constructs (or components) of the conceptual schema that cannot be directly translated into the target DDL are first *transformed* into constructs of the DBMS model. For instance, a single-valued atomic attribute is directly translated into a column. On the contrary, a N:N relationship type cannot be expressed in the relational DDL. Therefore, it is first transformed into a relationship entity type and two N:1 relationship types, which in turn are translated into a relationship table and two foreign keys. The resulting logical schema is the basis for program development. It must be clear, simple and devoid of any performance concern. Denoting the conceptual and logical schemas respectively by `CS` and `LS`, this process can be synthesized by the functional expression `LS = logical-design(CS)`, that states that the logical schema results from the transformation of the conceptual schema.

Physical design enriches and potentially reshapes the logical schema to make it meet technical and performance requirements according to a specific technology (DBMS). Physical design can be expressed by `PS = physical-design(LS)`, where `PS` denotes the physical schema.

Coding expresses the physical schema in the DDL of the DBMS. Some of the data structures and integrity constraints can be translated into explicit DDL statements. Such is the case, in relational databases, for elementary data domains, unique keys, foreign keys and mandatory columns. However, the developer must resort to other techniques to express all the other constructs. Most relational DBMSs offer check and trigger mechanisms to control integrity, but other servers do not include such facilities, so that many constraints have to be coped with by procedural code distributed and duplicated in the application programs. The derivation of the code can be expressed by `code = coding(PS)`. The code itself can be decomposed into the DDL code in which some constructs are explicitly expressed and the external code that controls and manages all the other constructs: $\texttt{code} = \texttt{code}_{\texttt{ddl}} \cup \texttt{code}_{\texttt{ext}}$. Similarly, the coding function can be developed into a sequence of two processes $(\texttt{coding}_{\texttt{ddl}}\texttt{(PS)}\,;\, \texttt{coding}_{\texttt{ext}}\texttt{(PS)})$.

The production of the database code from the conceptual schema (forward engineering or `FE`) can be written as `code = FE(CS)`, where function `FE` is the composition `coding` o `physical-design` o `logical-design`.

Empirical Database Design Actual database design and maintenance do not always follow a disciplined approach such as that recalled above. Many databases

have been built incrementally to meet the evolving needs of application programs. Empirical design relies on the experience of self-taught database designers, who often ignore the basic database design theories and best practices. This does not mean that all these databases are badly designed, but they may include many non-standard patterns, awkward constructs and idiosyncrasies that make them difficult to understand [2]. Since no disciplined approach was adopted, such databases often include only a logical schema that integrates conceptual, logical, physical and optimization constructs. Quite often too, no up-to-date documentation, if any, is available. An important property of the functional model of database design evoked in previous section is that it is still valid for empirical design. Indeed, if empirical design rules of the designer are sorted according to the criteria of the three processes, functions `logical-design'`, `physical-design'` and `coding'` can be reconstructed into an idealized design that was never performed, but that yields the same result as the empirical design.

Database Reverse Engineering Processes

Broadly speaking, reverse engineering can be seen as the reverse of forward engineering [1], that is, considering the function $RE = FE^{-1}$, $CS = RE(code)$. Since most forward engineering processes consist of *schema transformations* [9], their reverse counterparts should be easily derivable by inverting the forward transformations.

Unfortunately, forward engineering is basically a lossy process as far as conceptual specifications are concerned. On the one hand, it is not unusual to discard bits of specifications, notably when they prove difficult to implement. On the other hand, the three processes are seldom injective functions in actual situations. Indeed, there is more than one way to transform a definite construct and several distinct constructs can be transformed into the same target construct. For instance, there are several ways to transform an is-a hierarchy into relational structures, including the use of primary-foreign keys (forward engineering). However, a primary-foreign key can also be interpreted as the implementation of a 1:1 relationship type, as the trace of entity type splitting or as the translation of an is-a relation (reverse engineering). Clearly, the transformational interpretation of these processes must be refined.

Nevertheless it is important to study and to model the reverse engineering as the inverse of FE, at least to identify and describe the pertinent reverse processes. Decomposing the initial relation $CS = RE(code)$, one obtains:

$$CS = conceptualization(LS)$$
$$LS = logical\text{-}reconstruction(PS, code_{ext})$$
$$PS = physical\text{-}extraction(code_{ddl})$$
$$RE = conceptualization \circ logical\text{-}reconstruction \circ physical\text{-}extraction$$

where

$$Conceptualization = logical\text{-}design^{-1}$$
$$Logical\text{-}reconstruction = physical\text{-}design^{-1} \;||\; coding_{ext}{}^{-1}$$
$$Physical\text{-}extraction = coding_{ddl}{}^{-1}$$

This model emphasizes the role of program code as a major source of information. As explained below, other sources will be used as well.

Physical Schema Extraction

This process recovers the physical schema of the database by parsing its DDL code ($code_{ddl}$) or, equivalently, by analyzing the contents of its active data dictionary, such as the system tables in most relational systems. This extraction makes visible the *explicit constructs* of the schema, that is, the data structures and constraints that have been explicitly declared through DDL statements and clauses. Such is the case for primary keys, unique constraints, foreign keys and mandatory fields. Generally, this process is fairly straightforward. However, the analysis of sub-schemas (e.g., relational views, CODASYL sub-schemas or IMS PCBs) can be more intricate. Indeed, each sub-schema brings a partial, and often refined view of the global schema. In addition, some data managers, such as standard file managers, ignore the concept of global schema. A COBOL file for instance, is only described in the source code of the programs that use it. Each of them can perceive its data differently. Recovering the global physical schema of a COBOL file requires a potentially complex schema integration process.

Logical Schema Reconstruction

This process addresses the discovery of the *implicit constructs* of the schema. Many logical constructs have not been declared by explicit DDL statements and clauses. In some favorable situations, they have been translated into programmed database components such as SQL checks, triggers and stored procedures.

However, most of them have been translated into application program fragments (nearly) duplicated and scattered throughout millions of lines of code ($\texttt{code}_{\texttt{ext}}$). For instance, a popular way to check a referential constraint consists in accessing the target record before storing the source record in its file. Recovering these *implicit constructs*, in contrast with *explicit constructs*, which have been expressed in DDL, requires a precise analysis of various pieces of procedural code. Though program source code is the richest information source, the database contents (the *data*), screen layout, report structure, program execution, users interview and, of course, (possibly obsolete) documentation will be analyzed as well. As a final step of the reconstruction, physical components are discarded inasmuch as they are no longer useful to discover logical constructs.

Implicit Constructs All the structures and constraints that cannot be expressed in the DDL are implicit by nature. However, many database schemas include implicit constructs that could have been declared at design time but that were not, for such reasons as convenience, standardization, inheritance from older technology or simply by ignorance or bad design. Two popular examples can be mentioned. In network and hierarchical databases, some links between record types are translated into implicit foreign keys despite the fact that relationship types could have been explicitly declared through set types or parent-child relationship types. In many legacy relational databases, most foreign keys are not declared through `foreign key` clauses, but are managed by an appropriate set of triggers. The most important implicit constructs are the following [8].

Exact field and record structure. Compound and multivalued fields are often represented by the concatenation of their elementary values. Screen layout and program analysis are major techniques to discover these structures.

Unique keys of record types and multivalued fields. This property is particularly important in strongly structured record types and in sequential files.

Foreign keys. Each value of a field is processed as a reference to a record in another file. This property can be discovered by data analysis and program analysis.

Functional dependencies. The values of a field can depend on the values of other fields that have not been declared or elicited as a candidate key. This pattern is frequent in older databases and file systems for performance reasons.

Value domains. A more precise definition of the domain of a field can be discovered by data and program analysis. Identifying enumerated domains is particularly important.

Meaningful names. Proprietary naming standards (or, worse, the absence thereof) may lead to cryptic component names. However, the examination of program variables and electronic form fields in/from which field values are moved can suggest more significant names.

Sources and Techniques Analytical techniques applied to various sources can all contribute to a better knowledge of the implicit components and properties of a database schema.

Schema analysis. Spotting similarities in names, value domains and representative patterns may help identify implicit constructs such as foreign keys.

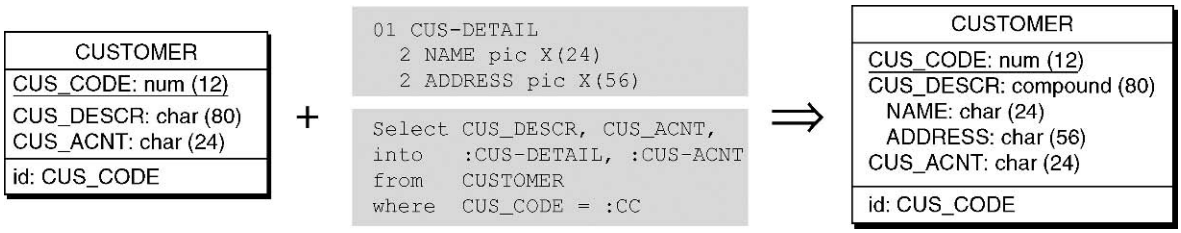
Data analysis. Mining the database contents can be used in two ways. First, to discover implicit properties, such as functional dependencies and foreign keys. Second, to check hypothetic constructs that have been suggested by other techniques. Considering the combinatorial explosion that threaten the first approach, data analysis is most often applied to check the existence of formerly identified patterns.

Program analysis. Understanding how programs use the data provides crucial information on properties of these data. Even simple analysis, such as dataflow graphs, can bring valuable information on field structure (Fig. 1) and meaningful names. More sophisticated techniques such as dependency analysis and program slicing can be used to identify complex constraint checking or foreign keys.

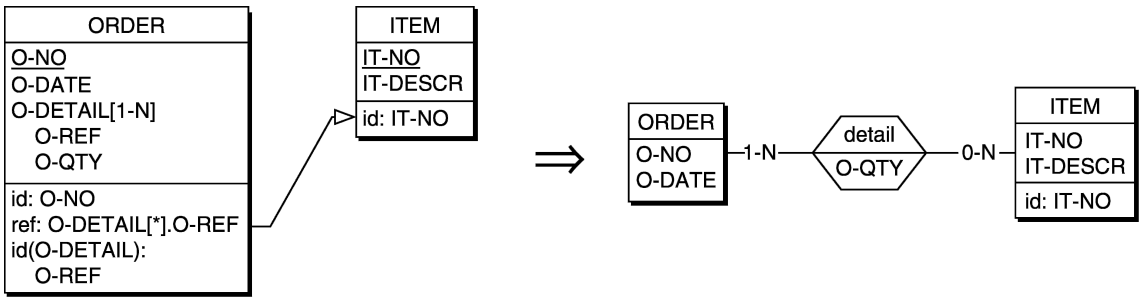
Screen/report layout analysis. Forms, reports and dialog boxes are user-oriented views on the database. They exhibit spatial structures (e.g., data aggregates), meaningful names, explicit usage guidelines and, at run time, data population that, combined with dataflow analysis, provide much information on implicit data structures and properties.

Schema Conceptualization

The goal of this process is to interpret the logical schema semantically by extracting a conceptual schema that represents its intended meaning. It mainly relies on transformational techniques that undo the effect of the logical design process. This complex process is decomposed in three subprocesses, namely



Database Reverse Engineering. Figure 1. Illustration of the physical schema extraction and logical schema reconstruction processes.



Database Reverse Engineering. Figure 2. Conceptualization of a complex field.

untranslation, de-optimization and conceptual normalization. The *untranslation* process consists in reversing the transformations that have been used to draw the logical schema from the conceptual schema. For instance, each foreign key is interpreted as the implementation of a N:1 relationship type. This process relies on a solid knowledge of the rules and heuristics that have been used to design the database. Those rules can be standard, which makes the process fairly straightforward, but they can also be specific to the company or even to the developer in charge of the database (who may have left the company), in which case reverse engineering can be quite tricky. The main constructs that have to be recovered are relationship types (Fig. 2), super-type/subtype hierarchies, multivalued attributes, compound attributes and optional attributes. The *de-optimization* process removes the trace of all the optimization techniques that have been used to improve the performance of the database. Redundancies must be identified and discarded, unnormalized data structures must be decomposed and horizontal and vertical partitioning must be identified and undone. Finally, *conceptual normalization* improves the expressiveness, the simplicity, the readability and the extendability of the conceptual schema. It has the same goals and uses the same techniques as the corresponding process in Conceptual analysis.

Tools

Reverse engineering requires the precise analysis of huge documents such as programs of several millions of lines of code and schemas that include thousands of files and hundreds of thousands of fields. It also requires repeatedly applying complex rules on thousands of patterns. In addition, many reverse processes and techniques are common with those of forward engineering, such as transformations, validation and normalization. Finally, reverse engineering is only a step in larger projects, hence the need for integrated environments that combine forward and reverse tools and techniques [11].

Examples

Figure 1 illustrates the respective roles of the physical schema extraction and logical schema reconstruction processes. Parsing the DDL code identifies column CUS_DESCR as a large atomic field in the physical schema (left). Further dataflow analysis allows this column to be refined as a compound field (right).

The conceptualization of a compound field as a complex relationship type is illustrated in Figure 2. The multivalued field O-DETAIL has a component (O-REF) that serves both as an identifier for its values (the values of O-DETAIL in an ORDER record have distinct values of O-REF) and as a reference to an

ITEM record. This construct is interpreted as an N:N relationship type between ORDER and ITEM.

Key Applications

Database reverse engineering is most often the first step in information system maintenance, evolution [10], migration and integration. Indeed, such complex projects cannot be carried out when no complete, precise and up-to-date documentation of the database of the information system is available.

The scope of data reverse engineering progressively extends to other kinds of information such as web sites, electronic forms, XML data structures and any kind of semi-structured data. Though most techniques and tools specific to database reverse engineering remain valid, additional approaches are required, such as linguistic analysis and ontology alignment.

Cross-references

- ▶ Database Design
- ▶ Entity-Relationship Model
- ▶ Hierarchical Data Model
- ▶ Network Data Model
- ▶ Relational Model

Recommended Reading

1. Baxter I. and Mehlich M. Reverse engineering is reverse forward engineering. Sci. Comput. Programming, 36:131–147, 2000.
2. Blaha M.R. and Premerlani W.J. Observed idiosyncrasies of relational database designs. In Proc. 2nd IEEE Working Conf. on Reverse Engineering, 1995, p. 116.
3. Casanova M.A. and Amaral de Sa J.E. Mapping uninterpreted schemes into entity-relationship diagrams: two applications to conceptual schema design. IBM J. Res. Develop., 28(1):82–94, 1984.
4. Chikofsky E.J. and Cross J.H. Reverse engineering and design recovery: a taxonomy. IEEE Softw., 7(1):13–17, 1990.
5. Davis K.H. and Aiken P.H. Data reverse engineering: a historical view. In Proc. 7th Working Conf. on Reverse Engineering, 2000, pp. 70–78.
6. Davis K.H. and Arora A.K. A methodology for translating a conventional file system into an entity-relationship model. In Proc. 4th Int. Conf. on Entity-Relationship Approach, 1985, p. 148–159.
7. Edwards H.M. and Munro M. Deriving a logical model for a system using recast method. In Proc. 2nd IEEE Working Conf. on Reverse Engineering, 1995, pp. 126–135.
8. Hainaut J.-L. Introduction to database reverse engineering, LIBD lecture notes, Pub. University of Namur, Belgium, 2002, p. 160. Retrieved Oct. 2007 from http://www.info.fundp.ac.be/~dbm/publication/2002/DBRE-2002.pdf.
9. Hainaut J.-L. The transformational approach to database engineering. In Generative and Transformational Techniques in Software Engineering, R. Lämmel, J. Saraiva, J. Visser (eds.), LNCS 4143. Springer-Verlag, 2006, pp. 89–138.
10. Hainaut J.-L., Clève A., Henrard J., and Hick J.-M. Migration of Legacy Information Systems. In Software Evolution, T. Mens, S. Demeyer (eds.). Springer-Verlag, 2007, pp. 107–138.
11. Hainaut J.-L., Roland D., Hick J-M., Henrard J., and Englebert V. Database reverse engineering: from requirements to CARE tools. J. Automated Softw. Eng., 3(1/2):9–45, 1996.
12. Navathe S.B. and Awong A. Abstracting relational and hierarchical data with a semantic data model. In Proc. Entity-Relationship Approach: a Bridge to the User. North-Holland, 1987, pp. 305–333.

Database Scheduling

▶ Database Middleware

Database Security

Elena Ferrari
University of Insubria, Varese, Italy

Synonyms

Database protection

Definition

Database security is a discipline that seeks to protect data stored into a DBMS from intrusions, improper modifications, theft, and unauthorized disclosures. This is realized through a set of *security services*, which meet the security requirements of both the system and the data sources. Security services are implemented through particular processes, which are called *security mechanisms*.

Historical Background

Research in database security has its root in operating system security [6], whereas its developments follow those in DBMSs. Database security has many branches, whose main historical developments are summarized in what follows:

Access control. In the 1970s, as part of the research on System R at IBM Almaden Research Center, there was a lot of work on access control for relational DBMSs [3]. About the same time, some early work on Multilevel Secure Database Management Systems (MLS/DBMSs) was reported, whereas much of the