Dynamic Analysis of SQL Statements in Data-intensive Programs

Jean-Luc Hainaut and Anthony Cleve

March 14, 2008

Abstract

SQL statements control the bi-directional data flow between application programs and a database through a high-level, declarative and semantically rich data manipulation language. Analyzing these statements brings invaluable information that can be used in such applications as program understanding, database reverse engineering, intrusion detection, program behaviour analysis, program refactoring, traffic monitoring, performance analysis and tuning, to mention some of them. SQL APIs come in two variants, namely static and dynamic. While static SQL statements are fairly easy to process, dynamic SQL statements most often require dynamic analysis techniques that may prove more difficult to implement. The goal of the paper is to identify and evaluate the most effective techniques for dynamic SQL statement analysis in data-intensive application programs. First, it describes the SQL API variants from the program architecture point of view. Then, it discusses some of the most important software engineering applications to which SQL statement understanding can be a significant contribution. A large range of analysis and processing techniques are proposed and the properties of each of them are evaluated. Finally, the applicability of these techniques to the software engineering applications is established. Two practical applications are presented and discussed.

1 Introduction

Data-intensive programs, also called database programs, generally comprise a database (sometimes in the form of a set of files) and a collection of application programs in strong interaction with the former. They constitute critical assets in most entreprises, since they support business activities in all production and management domains. Data-intensive programs form most of the so-called legacy systems : they typically are one or more decade old, they are very large, heterogeneous and highly complex. Many of them *significantly resist modifications and change* [BS95] due to the use of ageing technologies and to inflexible architectures. Since they nevertheless are due to evolve, sophisticated techniques have been elaborated that allow programmers and developers to identify and understand the logic of the code fragments and of the data structures that are to be changed, despite the lack of precise and up to date documentation. Recovering the required knowledge and control of poorly documented software components is the main goal of software reverse engineering [CC90].

1.1 The database component of software systems

It is interesting to learn how the communities devoted to the database and programming paradigms perceive the database component, both from the scientist and practitioner points of view. These views are quite different but complementary.

According to database experts, the database must be developed independently of the program needs. The goal of the database is to collect all the relevant data about a definite application domain (that part of the world concerned by the software system). The *Magna Carta* of a database is its *conceptual schema*, that identifies and describes the domain entities, their properties and their associations in a technology-independent way. The implementation of the database produces data structures that organize the data according to the data model of the chosen technology (the Database Management System or DBMS) but in strict conformance with the conceptual schema. The DBMS-dependent schema is called the *logical schema* of the database. In short, the conceptual schema expresses formally the intended semantics of the logical schema. Once the logical schema has been produced and coded in the Data Description Language (DDL) of the DBMS, application programs can be developed. In the final architecture, database experts perceive the database as the system's central component, around which the application programs are built.

For software engineering experts, system development largely ignores the database component. The latter appears as an encapsulated subsystem acting as a reliable and efficient data server. The database is an appropriate data container that ensures persistence, limited consistency, smooth concurrency and accident resistance. When external data are necessary, the programs invoke data extraction services from the DBMS through some sort of data management language (DML). The same channel is used to send data to be stored in the database. The emphasis is less on the domain modeling aspect of the database than on convenience (such as storage transparency) and performance.

1.2 On the complexity of data-intensive software systems

According to the modern description of pure software systems, a typical database can be perceived as a software sub-system made up of several thousands of classes, comprising dozens of thousands of attributes, and connected through several thousands of inter-class associations. Each class can collect several millions of persistent instances, so that a typical database forms a semantic network comprising billions of nodes and edges. These instances are shared, possibly simultaneously, by thousands of programs that read, create, delete and update several thousands times per second. Any of these programs can include hundreds of database statements of arbitrary complexity.

Since the database is supposed to include all the pertinent data about all the static object types of the application domain and since each program is designed to translate a business activity relying on these objects, it should not come as a surprise that data and processing aspects are tighly intertwined in application programs.

1.3 Evolution, maintenance and reverse engineering

Database schemas, software architecture and source code are supposed to be fully documented in order to make further maintenance and evolution easy and reliable. Unfortunately, developement teams seldom have time to write and maintain a precise, complete and up to date documentation. Therefore, many complex software systems lack the documentation that would be necessary for maintenance and evolution. Faced with the necessity of frequently changing the program code and the database structure due to maintenance needs or to functional or technological evolution, developers perform local code analysis to try to understand how things work in these parts of the software and of the schema that should be modified.

The problem happens to be particularly complex for the database documentation due to prevalent developement practices. First of all, many databases have not been developed in a disciplined way, that is, from a preliminary conceptual schema. This was true for old systems, but loose empirical design approaches keep being widespread for modern databases due, notably, to time constraints, poor database education and the increasing use of object-oriented middleware that tend to consider the database as the mere implementation of program classes. Secondly, the logical (DBMS-dependent) schema, that is supposed to be derived from the conceptual schema and to translate all its semantics, generally misses several conceptual constructs. This is due to several reasons, among others the poor expressive power of DBMS models and the lazziness, awkwardness or illiteracy of some programmers [BP95]. From all this, it results that the logical model often is incomplete and that the DDL code that expresses the DBMS schema in physical constructs ignores important structures and properties of the data. The missing constructs are called *implicit*, in contrast with the *explicit constructs* that are declared in the DDL of the DBMS. Several field experiments and projects have shown that as much as half of the semantics of the data structures is implicit. Therefore, merely parsing the DDL code of the database, or, equivalently, extracting the physical schema from the system tables, sometimes provides barely half the actual data structures and integrity constraints.

Recovering the implicit constructs is a prerequisite to maintenance and evolution of both the database and the programs. It also proves fairly difficult.

Indeed, it relies on such complex techniques as data mining, source code analysis, graphical interface analysis, program trace analysis.

1.4 Code analysis in reverse engineering

Data-intensive systems exhibit an interesting symmetrical property due to the mutual dependency of the database and the programs. When no useful documentation is available, it appears that (1) understanding the database schema is necessary to understand the procedural code and, conversely, (2) understanding what the procedural code is doing on the data considerably helps in understanding the properties of the data.

Procedural code analysis has long been considered a complex but rich information source to redocument database schemas. Even in ancient programs based on standard file data managers, identifying and analysing the code sections devoted to the validation of data before storing them in a file allows developers to detect constructs as important as actual record decomposition, uniqueness constraints, referential integrity or enumerated field domains. In addition, navigation patterns in source code can help identify such important constructs as semantic associations between record types.

When, as has been common for more than two decades, data are managed by relational DBMSs, the database/program interactions are performed through the SQL language and protocols. Based on the relational algebra and the relational calculus, SQL is a high-level language that allows programmers to describe in a declarative way the properties of the data they instruct the DBMS to provide them with. By contrast, navigational DMLs (also called *one-record-at-a-time* DML) access the data through procedural code that specifies the successive operations necessary to get these data. Therefore, a single SQL statement can be the declarative equivalent of a procedural section of several hundreds of LoC. Understanding the semantics of an SQL statement is much easier than that of this procedural fragment. The analysis of SQL statements in application programs is a major program understanding technique in database reverse engineering [PKBT94, And98, ES01, WES04, CHH06].

1.5 SQL code analysis

We illustrate the importance of SQL statement analysis on a small but representative example based on the schema of Figure 1 made up of two tables describing customers and orders. This schema graphically translates the constructs of the DDL code. Query 1 obviously extracts the customer city and the ordered product for a definite order. It ask the DBMS to extract these data from the row built by joining tables `CUSTOMER` and `ORDERS` for that order. It exhibits the main features of the program/query interface: the program transmits an input value through host variable `ORDID` and the query transmits result values in host variables `CITY` and `PRODUCT`. The analysis of this query brings to light some important hidden information, two of which are essential for the understanding of the database schema.

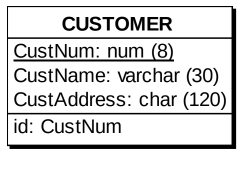

Figure 1: Two tables including implicit constructs

```
select substring(CustAddress from 61 for 30), Reference
into   :CITY, :PRODUCT
from   CUSTOMER C, ORDERS O
where  C.CustNum = O.Sender
and    OrdNum = :ORDID
```

Query 1. Extracting hidden City and Product information (predicative join)

1. The join is performed on columns `CustNum` and `Sender`. The former is the primary key (the main identifying column) of table `CUSTOMER` while the second one plays no role so far. Now, we know that most joins found in application programs are based on the matching of a foreign key (a column that is used to reference a row in another table) and a primary key. As a consequence, Query 1 strongly suggests that column `Sender` is a foreign key to `CUSTOMER`. Further analysis will confirm or reject this hypothesis.

2. The seemingly atomic column `CustAddress` appears to actually be a compound field, since its substring at positions 61 to 90 is extracted and stored into a variable named `CITY`.

Translating this new knowledge in the original schema leads to the more precise schema of Figure 2.

The SQL code of Query 1 explicitly exhibits the input and output variables through which the host statements interact with the query. These variables are important since, on the one hand, input variables define the variable parts of the query and, on the other hand, input and output variables potentially weave links with other SQL queries. The code fragment Query 2, that develops the same join as that of Query 1 in a procedural way, illustrates such an inter-query

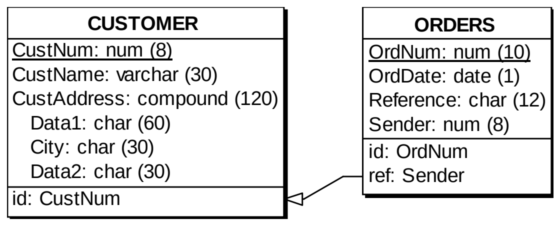

Figure 2: Two implicit constructs revealed by the analysis of Query 1

dependence. Naming *Q21* and *Q22* the two fragments, we observe that the

```
select Sender into :CUST
from   ORDERS
where  OrdNum = :ORDID
   <host statements>
select CustName, CustAddress
into :CNAME, :CADDRESS
from   CUSTOMER
where  C.CustNum = :CUST
```

Query 2. Extracting hidden City and Product information (procedural join)

host variable `CUST` is the output variable of query *Q21* and the input variable of the query *Q22*. If the intermediate host statements do not change the value of `CUST` (this invariance can be checked through a dependency analysis), then these queries can be considered as a global query. Such inter-query analysis has been suggested by [PKBT94] for example.

The analysis of SQL statements in a program can address each statement independently (Query 1) or can extend the understanding to chains of dependent statements (Query 2).

1.6 Static vs dynamic SQL

These introductory examples are expressed in *static SQL*, a variant of the language in which the SQL statements are hard-coded in the source program. There is another family of SQL interfaces, called *dynamic SQL*, with which the SQL statements are built at runtime and sent to the database server through a specific API. Typically, these programs *build* each query as a character string, then ask the DBMS to *prepare* the query (i.e., to compile it) and finally *execute* it. The only point in the program at which the actual query exists is at runtime, when, or just before, the query string is sent to the DBMS for compilation and/or execution. Query 3 is a simple example of the use of dynamic SQL to extract the name of customer 'C400'.

```
CNUM = "C400";
QUERY = "select CustName from CUSTOMER where CustNum = '" + CNUM + "'";
exec SQL prepare Q from :QUERY;
exec SQL execute Q into :NAME;
```

Query 3. First example of dynamic SQL

1.7 Static vs dynamic analysis of (dynamic) SQL statements

With some disciplined programming styles (Query 3 is an example), the mere examination of the code of the program provides enough information to infer the actual content of the query string before execution. However, the way this string is computed can be so complex and tricky that only runtime analysis can yield this value. Capturing, saving and processing the values of the query string at runtime resorts to *dynamic program analysis*.

1.8 Structure of the paper

The objective of this paper is to identify, study and apply dynamic analysis techniques of static and dynamic SQL statements in the context of software and database engineering. Section 2 describes in further detail the dynamic SQL interface as well as the scope of its static and dynamic analyses. Section 3 identifies important objectives and applications to which dynamic analysis of SQL can contribute significantly. Section 4 identifies and describes ten different techniques for capturing SQL statements traces, while Section 5 elaborates on the way such traces may be processed. These techniques are evaluated and their applicability is assessed in Section 6. Section 7 details two practical applications of SQL dynamic analysis, namely program reenginnering and database implicit constraints recovery.

2 Static VS Dynamic SQL

The introductory examples of Queries 1 and 2 are expressed in *static SQL*, which is a variant of the language in which the SQL statements are hard-coded in the source program. The code of static SQL explicitly shows the architecture of the query according to the SQL native syntax. It mentions the schema constructs concerned and identifies the input and output variables through which the host statements interact with the query. Static SQL will appear in standard languages under the name *embedded SQL* (ESQL) and in Java as SQLJ. Many proprietary database languages, such as InterBase, Oracle (PL/SQL) or Sybase and SQL Server (Transact SQL) are based on static SQL as well, though some of them provide some form of dynamicity.

2.1 Dynamic SQL

Dynamic SQL or call-level interface (CLI), that has been standardized in the eighties and implemented by most relational DBMS vendors, is illustrated in Queries 3 and 4. The most popular DBMS-independent APIs are ODBC, proposed by Microsoft, and JDBC, proposed by SUN. Dynamic SQL provides a high level of flexibility but the application programs that use it may be difficult to analyse and to understand. Most major DBMS, such as Oracle and DB2, include interfaces for both static and dynamic SQL.

The examples in Queries 3 and 4 are written in dynamic SQL for C/C++. The first one shows the build time injection of constants from variable `CNUM`, while Query 4 illustrates the binding of formal variable `v1` with the actual host variable `CNUM` at execution time.

```
QUERY = "select CustName from CUSTOMER where CustNum = :v1";
exec SQL prepare Q from :QUERY;
exec SQL execute Q into :NAME using :CNUM;
```

Query 4. Another common usage pattern of dynamic SQL

The ODBC and JDBC interfaces provide several query patterns, differing notably on the binding technique, that we refer to in this study. The most general form is illustrated in Query 5, where the iteration structure for obtaining results has been ignored for simplicity. Line 1 creates database connection `con`. Line 2 builds the SQL query in host variable `SQLquery`. This statement includes input placeholder `?` which will be bound to an actual value before execution. Line 3 creates and prepares statement `SQLstmt` from string `SQLquery`. This statement in completed in Line 4, by which the first (and unique) placeholder is replaced by value C400. The statement can then be executed (Line 5), which creates the set of rows `rset`. Method `next` of `rset` positions its cursor on the first row (Line 6) while Line 7 extracts the first (and unique) output value specified in the query select list and stores it in host variable `Num`. Line 8 deletes the result set.

```
    CNum = "C400";
1   Connection con = DriverManager.getConnection(DBurl, Login, Password);
2   String SQLquery = "select OrdNum from ORDERS where Sender = ?";
3   PreparedStatement SQLstmt = con.prepareStatement(SQLquery);
4   SQLstmt.setString(1, CNum);
5   ResultSet rset = SQLstmt.executeQuery();
6   rset.next();
7   Num = rset.getInt(1)
8   rs.close();
```

Query 5. Standard JDBC database interaction

When input binding is performed by build time value injection, the prepare and execute steps can be merged, as illustrated in Query 6, which is equivalent to Query 5 (operations of Lines 1 and 8 omitted).

```
String SQLquery = "select OrdNum from ORDERS where Sender = ’"+"C400"+"’";
Statement SQLstmt = con.createStatement();
ResultSet rset = SQLstmt.executeQuery(SQLquery);
rset.next();
Num = rset.getInt(1)
```

Query 6. Concise JDBC database interaction

2.2 Static vs dynamic analysis of dynamic SQL statements

In some disciplined programming styles, the building step is written as a sequence of explicit substring concatenations just before the prepare/execute section, so that significant fragments of the SQL query, if not the whole query itself, can be recovered through careful static analysis [HvdBV07] or symbolic execution [NT08]. However, some fragments of the query may be initialized long before, and far away from the execution point, or computed, or extracted from external sources (file, database, user interface, web pages, etc.) in such a way that discovering the intended query by code examination alone may prove impossible. The JDBC fragment of Query 7 illustrates the problem. There obviously is no realistic static analysis procedure that could provide us with the actual SQL queries that will be executed. The only way to know the actual query is to capture it at runtime, when the `executeQuery` method is executed.

```
String query, SQLv, SQLa, SQLs, SQLc;
SQLv = currentAction; SQLa = keyboard.readString();
SQLs = userDefaultTable; SQLc = getFilter(currentDate, filterNumber);
Connection con = DriverManager.getConnection(url, login, pwd);
Statement stmt = con.createStatement();
query = SQLv + SQLa + SQLs + SQLc;
ResultSet rset = SQLstmt.executeQuery(query);
```

Query 7. An example of dynamic SQL (JDBC)

2.3 Capturing results of SQL statement execution

Dynamic analysis has been considered so far as a means to get the actual SQL code at execution time in order to examine it. This technique can also be used to examine the results of SQL code execution. The information is captured after code execution. The format of the output data is defined by the form of the SQL query and the information of the logical schema. Result analysis often is the only technique to trace the link between two SQL queries through shared host variables when the intermediate host statements cannot be analysed statically (see Query 2 for example).

2.4 Dynamic SQL patterns

Any dynamic statement can be seen as a partially filled frame. Some parts are constant and will appear in the same place as all the instances of the statement while the other parts are drawn from variables or computed to complete the frame. There are two extreme cases: all the parts of the frame are variable and the whole frame is a pure constant. The most interesting cases are in between, depending on which SQL syntactic component is variable from one execution to the other. We will briefly describe five of them.

Constant variability. The only variable parts are the constant values of the query. They appear in the `where` clause of `select/update/delete` queries (as the constants with which column values, or any scalar expression value, are compared) and in `insert` queries (as the column values of the row to insert).

This form has been illustrated in Query 4. A dynamic query can only appear in three successive states: (1) *unbound static query*, in which the variable tokens still appear as placeholders or formal variable names (see Queries 4 and 5), (2) *bound static query*, in which the variable tokens have been replaced with actual host variable names, providing a pure ESQL query (Query 1) and (3) *instantiated static query*, in which the variable tokens have been replaced by constants.

Column variability. The select list, that is, the list of data items in each row of the result set, may vary from one execution to the other. This form encompasses also `insert` queries, in which the value list can be variable. It generally includes constant variability as well.

Table variability. The table(s) on which the statement operates may vary. All the tables must have the same structure. This form may include the first two variabilities.

Condition/order variability. The selection criteria specified in the `where` clause of `select`, `update` and `delete` queries, or the `order by` clause of a `select` query, may vary.

Action variability. The other parts of the SQL may vary. We can consider that the successive instantiations of the frame build quite different queries. This pattern will be found in highly interpretative applications, for instance in widespread interactive Java SQL client applications allowing users to submit any SQL statement to a database engine.

It is important to observe that a constant variability pattern can be losslessly replaced by a static SQL statement and that this property does not hold for the other patterns. For the latter, the building sequence dedicated to a given dynamic statement generates in fact a family of syntactically different static statements.

3 Applications of SQL Statement Analysis

The introduction has motivated the dynamic analysis of SQL statements as a contribution to program understanding, in particular for implicit data structure elicitation. In fact, these analysis techniques have a wider range of applications, some of which will be discussed in this section. We will develop the role of dynamic analysis in program understanding and in process control and monitoring.

3.1 Program understanding

SQL statement analysis contributes significantly to the more global process of program understanding. In addition, it is a major instrument to database structure understanding.

Dependency graph analysis. The dependency graph of the variables of a program is a popular way to understand the coupling of different components of this program, notably for change impact analysis. This graph comprises nodes that represent variables and constants and edges, each of which specifies that the state of a variable depends on the state of another variable or constant. While this analysis is quite common for assignment, comparison, computing and simple input/output statements, understanding variable dependencies from rich middleware API may prove more complex. In such a context, an SQL statement can be reduced to a simple function. If only host variable dependencies are considered, Query 1 can be reduced to the following peudo-code, in which `sql1` and `sql2` are syntactic functions merely expressing uninterpreted dependencies between variables: the values of `CITY` depend, among others, on the values of `ORDID`.

```
CITY = sql1(ORDID);
PRODUCT = sql2(ORDID);
```

This transformation has been used to build the dependency graphs of program slices in [HEH+98] and [CHH06]. It can be strongly improved by considering the properties of external data as expressed in the database schema. For instance, the schema of Figure 2 shows that (1) a functional dependency (FD) holds from column `OrdNum` to column `Sender` in table `ORDERS`, (2) an inclusion dependency holds between column `Sender` and column `CustNum` and (3) a FD holds from column `CustNum` to subcolumn `City` in table `CUSTOMER`. The transitivity rule implies that a FD also holds from column `OrdNum` to subcolumn `City` in the join `CUSTOMER*ORDER`. As a consequence, syntactical functions `sql1` and `sql2` are proved to be *mathematical* functions as well (the value of `CITY` depends on the value of `ORDID` only), so that we have proved that the result set of Query 1 will never include more than one row. This property should considerably enrich the program understanding process. As far as the authors know, this approach has not been explored yet.

Implicit database construct elicitation. The exploitation of Query 1 illustrates the contribution of the analysis of SQL queries to the elicitation of implicit constructs and contraints of a database [HMCM93, SLGC94, PTK95, YC99, LPT99, SLF+01, Hai02]. In addition, the enriched dependency graph that can be built by considering SQL statement interactions are used to propagate information on a node to other nodes, in particular database constructs [CHH06].

Reconstruction of the static code. The most obvious application of dynamic analysis is the reconstruction of the static equivalent of dynamic SQL queries. Static analysis of static SQL statements has long be studied and analytical techniques and tools have been developed and are now available, in particular in database reverse engineering, as illustrated in the introduction. Rebuilding the static code can be performed at two levels of scope, namely local and global. Local SQL code analysis consider each SQL statement independently of its environment, and in particular of the other SQL statements executed before and after it. In contrast, global SQL code analysis consider also the inter-relations between successive SQL statements, mainly through shared host variables. Query 2 is a simple example of dependencies that can hold between two distant SQL statements. The latter analysis is much more powerful than the former, but it must rely on complex program understanding techniques such as program slicing [Wei84]. This dynamic/static conversion can be used to reengineer complex programs in order to improve their readability and to ease their maintenability and evolvability.

Quality evaluation. Logging all the SQL statements issued by a program in a definite period provides a fairly comprehensive sample of all the SQL forms that the program actually uses. This information can be used to identify the SQL programming style and, in particular, *bad smells* [Man03] that should be reengineered.

Database usage matrix. Statement analysis provides partial information that can be used to redocument the programs. A simple though quite useful derived information is the usage matrix [DK98] that specifies which tables and which columns each program unit uses and modifies. Formal concept analysis can then be applied to identify potential program clustering or database schema partitioning.

3.2 Process control and monitoring

Analysing the statement flow and the data flow in specific critical program points can yield precise information on the behaviour of the program at execution time. Five applications are described below.

Statistics. Logging SQL statements produces a data collection that can be mined to extract aggregated information and statistics on the behaviour of the program as far as database interaction is concerned. According to the additional data recorded with these statements, useful information can be derived such as database failure rate at each program point (update rejected, access denied, empty result set, etc.), most frequently used SQL forms, complexity of SQL statements, programming idiosyncrasies, awkward and inefficient SQL patterns and non-standard syntax (inducing portability vulnerability). These derived data can be used to monitor the program behaviour but also for refactoring purpose, for instance to improve the quality and the portability of the source code. The analysis of selection criteria in select, delete and update statements can be used to define and tune the data access mechanisms such as indexes and clusters.

Accounting. The execution of a program can be charged information access cost. This cost can depend on the value of the information, on the volume of the data extracted from the database or on the time the database engine spent on executing the queries.

Performance analysis. Time recording before and after each SQL statement execution allows a precise evaluation of that part of program execution spent on data exchange with the database. Analysing these data can be used to fine tune both the application program and the database for better performance.

Transaction management. The DBMS takes in charge transaction management according to the policy defined by the database administrator. Ad hoc, customized, transaction management can be necessary in some critical situation. Capturing all the data modification SQL statements submitted to the DBMS at run time and recording them in a log make it possible to redo (and in some circumstances to undo) their effect when some adverse events or conditions occur.

Security. The main database application vulnerability is *SQL code injection* [MLA06]. It occurs when an external user is requested to provide data (typically its user ID and password) that are injected at building time into an incomplete SQL query. However, the user actually provides, instead of the expected data, spurious but syntactically valid data in such a way that a malware query is formed. This query is then executed with privileges that the user has not been granted. For instance, a user who logs in a system is requested to provide its password through a dialog box. When aquired, the value is injected into an SQL query to form a valid statement that checks the existence of this password in the authorization table. The instantiated statement generally looks like the following: `"select count(*) from USERS where PASSWORD = 'x1123bz';"`, and is expected to return a zero (access denied) or non-zero (access granted) value. It has been built by injecting the value `x1123bz` provided by the user according to the frame `"select count(*) from USERS where PASSWORD = '" + "x1123bz" + "';"`. If the user enters the string `"X' or 'A' = 'A"` instead of a valid password, the statement built becomes the unexpected but valid statement `"select count(*) from USERS where PASSWORD = 'X' or 'A' = 'A';"`. Since this query returns the number of rows in the table, the unauthorized user is given access to the system. Interestingly, the problem can be formalised as the illegal transformation of the constant variability pattern written by the programmer into an unexpected condition variability pattern.

Most common attack detection techniques rely on the analysis of the value provided by the user, but dynamic analysis of the actual (vs intended) SQL query may prove easier and more reliable. In the same domain, SQL code analysis can detect unauthorized queries and updates. For instance, a query including the fragment `"from REPORT where STATUS = 'classified'"` can be identified as suspect and blocked until official authorization notification. The analysis of the result set of SQL queries can also be used to identify the presence of sensitive information.

4 SQL Statement Capturing Techniques

At runtime, the SQL protocol relies on a dataflow that starts and ends at the host program point defined by an SQL statement. The successive steps define the flow points at which capture instruments can be installed. The following eleven steps are identified.

- Step d1. (Building step, client side) The SQL statement is formed by the concatenation of statement fragments. The resulting string generally includes constant placeholders.
- Step d2. (Preparation step, client side) The statement is sent to the database engine for preparation.
- Step d3. (Binding step, client side) The placeholders, if any, are bound with host variables so that the statement is now complete.
- Step d4. (Transmission step, client side) The host program transmits the SQL statement to the client API for execution. The program passes control to the API and suspends itself.
- Step d5. (Statement sending step, link side) The client API receives the SQL statement and sends it to the database engine.
- Step d6. (Statement receiving step, link side) The database engine receives the statement and writes it on its log.
- Step e1. (Recompilation step, server side) The database engine checks the validity of execution plan of the query by comparing the compilation date of the query with the last update time of the schema.
- Step e2. (Execution step, server side) The database engine executes the query.
- Step b1. (Result step, server side) The database engine sends the status messages to the client and, if needed, the result set is extracted from the database.
- Step b2. (Receiving step, link side) The client API receives the message and result set.
- Step b3. (Extraction step, client side) The client receives control from the API and extracts the results to store them in its host variables.

It is important to note that each API variant executes a subset only of these steps. For instance, in static SQL, steps d1, d2, d3 are performed at writing time and compile time; similarly, b3 is a compile time step. In dynamic SQL, step e1 is absent.

Now, we describe ten techniques to capture the SQL statements that are executed in a data-intensive application program. Seven techniques are intended to understand the behaviour of the client/server system at execution time while three of them resort to static analysis.

```
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(SQLstmt);
SQLlog.write("132;SQLexec;"+stmt.hashCode()+";"+SQLstmt);
rs.next();
vName = rs.getString(1);
SQLlog.write("133;SQLgetS1;"+rs.getStatement().hashCode()+";"+vName);
vSalary = rs.getInt(2);
SQLlog.write("134;SQLgetI2;"+rs.getStatement().hashCode()+";"+vSalary);
```

Figure 3: Insertion of tracing statements in the source code for logging SQL operations

Insertion of tracing statements. The capture of a dynamic SQL statement is performed by a dedicated code section inserted before the program point of this statement. Similarly, the result of an SQL statement will be captured by a code section inserted after it. This technique requires code analysis to identify and decode database API statements and entails source code modification and recompilation. It provides a temporal list of statement instances. In the example of Figure 3, the tracing statement writes in the log file the program point id (132), the event type (SQLexec), the statement object id (`stmt.hashCode()` or `rs.getStatement().hashCode()`) followed by the SQL statement (building time constant injection is assumed) or the output variable contents. According to the information that needs to be extracted from the trace, program id, process id and/or timestamp can be output as well. This technique may involve more complex source code restructuring when the SQL statement is built in the SQL prepareStatement argument (as in prepareStatement(A+B+C)). Indeed, this building can invoke functions with side effects such as acquiring statement fragments from external sequential sources. It should be noted that static SQL can be processed in the same way. Since the statement explicitly appears in the source code, only the values of input and output variables need to be captured. The advantage of dynamic analysis of static SQL is that it provides information on statement instances. However, it does not yield any information on statements that are not executed.

Static analysis of dynamic SQL statements. When SQL statements are built according to a disciplined and systematic procedure, static analysis techniques such as mere sequential parsing or program slicing can be used to understand this procedure and reconstruct the static query. This technique provides program points but cannot trace statement instances. The example of Query 3 provides a clear illustration of such a scenario, that has been studied in [HvdBV07]. As far as the authors know, this reference is the first one that describes this technique.

```
public aspect ExecuteQueryAspect {
  pointcut queryExecution() : call(ResultSet Statement.executeQuery(String));
  before(): queryExecution(){
    String className= thisJoinPoint.getSourceLocation().getFileName();
    int lineOfCode = thisJoinPoint.getSourceLocation().getLine();
    Statement s = (Statement) thisJoinPoint.getTarget();
    Object[] o = thisJoinPoint.getArgs();
    String query = (String) o[0];
    int statNumber = s.hashCode();
    SQLlog.write(className+";"+lineOfCode+";"+statNumber+";"+query);
  }
}
```

Figure 4: A tracing aspect capturing the execution of SQL statements

Tracing aspect. Aspect technology, if available for the programming language of interest, allows triggers to be added to an existing program without source code modification. The introduction of tracing aspects simply requires the programs to be recompiled. In the case of Java/JDBC applications, AspectJ pointcuts and advices can be defined in order to capture the successive events involved in dynamic database manipulation like query preparation, input value definition, query execution, result extration. As an example, we give in Figure 4 a simple tracing aspect that captures and records the execution of an SQL query (without statement preparation). Aspect-oriented support is now available for several programming languages among which Java, C and C++. Note that a prototype aspect language has been recently introduced for Cobol [LD05].

API overloading. The API overloading technique consists in encapsulating (part of) the client side API within dedicated classes which provide the same public methods but produce, in addition, the required SQL execution trace. For instance, we could write our own `Statement`, `PreparedStatement` and `ResultSet` classes which, in turn, make use of the JDBC corresponding classes. This technique requires a minimal program adaptation and, consequently, recompilation.

API substitution. If the source code of the client side API is available, which is the case for ODBC and JDBC drivers of Open Source RDBMSs, tracing statements can be inserted directly in this API. The latter is then recompiled and bound to the client applications. The client program need not be modified nor recompiled. This technique records the statement instances but ignores the program points.

DBMS log. Most database engines store the requests received from the client application programs in a specific file or table. For example, MySQL writes, in the order it received them, all the client queries in its general query log. Each record comprises the client process id, the time stamp the query is received and the text of the query as it was executed, in particularly with input variables replaced with their values. As compared to the trace obtained by the insertion

```
create trigger LOG_ORDER_UPDATE before update on ORDERS
for each row
begin
declare S varchar(500);
S := "update ORDERS ";
if new.ORDDATE <> old.ORDDATE then S := S + "set ORDDATE=" + new.ORDDATE;
if new.REFERENCE <> old.REFERENCE then S:= S + ", set REFERENCE=" + new.REFERENCE;
if new.SENDER <> old.SENDER then S:= S + ", set SENDER=" + new.SENDER;
S = S + " where ORDNUM=" + old.ORDNUM + ";";
end;
```

Figure 5: Reconstructing a fictitious update query using an update trigger

of tracing statement technique, DBMS log does not provide program points and can be processed off-line only. This technique does not require source code modification.

Tracing triggers. A tracing trigger is an SQL trigger created to capture and record data update activities on a definite table. The body of the trigger is executed before or after each data modification query. It is not provided with the original text of the query but its effect can be recovered by the comparison of the states of the data before and after query execution. By combining for-each-statement and for-each-row triggers, elementary fictitious queries can be reconstructed and recorded for further analysis. This technique is weaker than most others since it cannot capture the actual statements and since it ignores consultation queries, but it implies no source code modification. For instance, let us consider an update trigger executed for each update query instance. An equivalent fictitious query can be built as follows: for each column `C` for which `new.C <> old.C`, a set clause `"set C=" + new.C` is defined. A trigger monitoring update queries with invariant primary key for the table `ORDERS` of Figure 1 would be defined by the pseudocode shown in Figure 5.

Tracing SQL procedures. Another possible technique consists in replacing some or all SQL statements in client programs by the invocation of equivalent SQL procedures created in the database. The set of these procedures acts as an ad hoc API that can be augmented with tracing instructions that maintain a log of the instances of SQL statements that are executed. This technique can be considered in architectures that already rely on SQL procedures. When client programs include explicit SQL statements, it entails in-depth and complex code modification. However, since it replaces complex input and output variable binding by mere procedure arguments, this reengineering can provide a better code that will be easier to maintain and evolve.

Extraction of compiled queries from system tables. Programs relying on static SQL must be precompiled. Through this process, each SQL statement is analysed by a precompiler that stores both the source statement and its compiled equivalent in a DBMS system table (step e1 above). Generally, this table can be read off-line, so that all the SQL statements of a program can be examined. This procedure does not provide statement instances, host variable values nor program points. It does not apply to dynamic SQL statements, that are compiled for each execution of the client program.

Static analysis of static SQL statements. As Query 1 shows it, the identification of static SQL statements is immediate, all the more since these statements are signalled by specific meta-instructions such as `exec SQL` and `end-exec` in COBOL, `#SQL{...}` in SQLJ or equivalent in other languages. Their further analysis poses no hard problems since the statements are guaranteed to be correct. This technique provides statement program points but does not trace their instances. Global static analysis consists in identifying the dependency relationships between SQL statements. Query 2 illustrates the point by explicitly showing that output variable `CUST` of the first query is used as input variable in the second one. However, this identification may prove much more complex than local analysis when there is no simple and transparent path from the output variables of a statement and the input variable of a subsequent statement [WES04].

Dynamic analysis of static SQL statements. Dynamic analysis can be used to study static SQL statement structure and behaviour. In particular, it can help solve two problems exhibited by static analysis. First, it produces a statement instance trace that static analysis is unable to provide. Second, the dependency identification problem can be coped with, and often solved, by identifying statement constants that are shared by several statement instances. For instance, considering a variant of Query 2 in which the link between the queries has not been elicited (for instance the input value is computed from the output value through an unknown function), observing that the value of the input variable of the second query is always the same as that of the first (and only) output variable of the first query strongly suggests that both variables always have the same value.

5 SQL Trace Processing

Once one or several SQL traces have been produced from program executions, they need to be further analyzed and interpreted. The complexity of process obviously depends on its objective. Applications of dynamic analysis of dynamic SQL statements rely on lower-level, intermediate objectives that can be classified in two major categories, namely identification of the varying components in constant variability patterns and the reconstruction of static statements equivalent to dynamic patterns.

As discussed in [NT08], identifying all the variants of static statements that can be generated by the dynamic SQL statements of a program still is an open problem. In the framework developed in this paper, this means that, in general, as long as we are not able to generate the set of instantiated static statements that could be generated at a given program point, we cannot determine the category of variability patterns a dynamic statement generation code section belongs to. The following analysis considers trace examination and mining only. They could be refined by taking into account the contribution of (partial) static analysis. For instance, the category of variability can sometimes be determined by static analysis [HvdBV07, NT08], as illustrated by Query 3.

5.1 Constant/variable identification

The objective is to identify, in instantiated statements, the constants that may vary depending on the execution and to establish the link between these constants and the host variables. The trace of any SQL statement that is, or has been, submitted for execution is an instantiated static statement.

Variable constants in constant variability pattern. If a significantly large set of traces produced at the same program point vary on the constants only, they can be considered as generated by a constant variability pattern. A problem arises for slowly varying constants. This phenomenon concerns query parameters related to spatio-temporal aspects of program execution. If a query includes a condition on the current year or on the branch of the company, dynamic analysis of the trace generated from a given workstation during a given month cannot identify such constants as possibly varying. However, if the trace also includes variable binding statements, as in Query 5, matching the placeholder or formal variable and the binding couple allows variability to be explicitly identified.

Input and output host variables of a constant variability pattern. Once the constants have been identified as varying, it can be necessary to identify their data sources in the program code, that is, in most case, the host input variables. When static analysis fails, as in Query 7, data analysis can detect constant equality in the traces from different program points. If the first constant belongs to the result of query q_1, if the second constant is an input value of query q_2 and if both constants are equal throughout the traces, then the probability that both constants come from the same variable is high. Studying input and output constants may also reveal another kind of inter-query dependency, according to which the result of a query q_1 influences the fact that another query q_2 is executed or not. This is typically the case when a program first verifies an implicit integrity constraint (as a foreign key) before inserting or updating an SQL row.

5.2 Static statements reconstruction

Several major applications of SQL analysis require the reconstruction of the static statement(s) generated by a definite dynamic pattern. Therefore, techniques for deriving these statements from the trace are essential when static analysis cannot completely identify these statements from the source code.

Instantiated statements in constant variability pattern. As long as the execution statements are traced, all instantiated statements are included in the trace. All is needed is to check that there is no other variability than constant variability.

Bound statements in constant variability pattern. Replacing constants by their data sources yields the exact static equivalent of a dynamic SQL statement. This process, that leads to the identification of the program constructs that supply the constants, is based on both static analysis and trace processing. It can be particularly complex when a constant is built by an expression *in situ* (`CustNum = 'C' + :CODE`) instead of merely extracted from a variable (`CustNum = :CNUM`). Introducing an additional variable in the host code will solve the problem.

Unbound statements in constant variability pattern. Though less useful in program understanding, it can be used to standardize programming style. Aligning all dynamic patterns of a program on Query 5 or Query 6 styles is an example. If the unbound statements have not been traced, then they can easily be derived from their bound versions.

Family of constant variability patterns. When statement parts other than the constants are varying, then the dynamic pattern generates a family of static statements instead of a single statement. Considering a definite execution (or preparation) program point, the syntactic analysis of its instantiated (or bound/unbound) static statement produces a set of constant variability patterns. If each pattern is induced from a significantly large subset of traces, then each pattern can be considered, with a high probability, as a pertinent static statement. As already mentioned, the completeness problem of the set of constant variability patterns is still unsolved.

6 Evaluation and Applicability of SQL Analysis Techniques

The techniques identified in Section 4 are designed to contribute to the goals of SQL statement analysis described in Section 5. To this aim, they will be evaluated and compared against functional and operational criteria that we will study first. The evaluation criteria will be classified into four property families, namely the nature of the information captured, completeness, host languages and operational characteristics.

Nature of information captured. Each technique captures information that will be further processed. This information can be the static SQL statement, which comes in three variants, namely unbound (with variable placeholders), bound (with actual host variable names) and instantiated (with constants).

The technique can also return the result of the execution of the statement as data values and/or execution status. The program point of the statement can be returned as well. It appears that some techniques cannot cope with all the SQL, the class of statement that can be captured will be specified: DDL (`create`, `drop`, `alter`), data extraction (`select`), data modification (`insert`, `update` and `delete`), control (`grant`, `revoke`, etc.) and binding statements that connect host variables to the unbound input and output placeholders.

Completeness. The question is, can all the SQL statements of the client program be captured by the technique. Due to its complexity in the context of dynamic SQL, this question will only be mentioned for static SQL statements. We will distinguish the identification of statements and that of statement instances. In [NT08], the authors define an automatic procedure to identify as many as possible SQL statement instances (they call them concrete statements) through static analysis techniques based, notably on symbolic execution of program paths that include database interactions. Actual case studies show that about 80% of instantiated statements can be identified. None of the techniques described in this paper address this problem, so that we will discuss the completeness characteristic for static statements only.

Host languages. A technique is applicable to some host languages and not to others. For instance, aspect technologies have so far been developed for popular OO languages only. We will examine whether each technique is applicable to COBOL, C, C++ and Java.

Operational characteristics. Some techniques impose more or less strong requirements on the following aspects:

- *Database schema*: does the technique require schema *examination*, schema *modification*?
- *Client source code*: does the technique require code *examination*, code *modification*, code *recompiling*?
- *Cost*: each technique induces various kinds of additional cost. We distinguish preparation cost, client runtime cost and server runtime cost. In each category, a coarse-grain score of 0 (no to low cost), 1 (medium cost) and 2 (high cost) will be assigned.
- *Processing time*: the information captured allow real-time processing (RT) or requires deferred processing (D). This property is crucial for such applications as security control.

Table 1 synthetizes the characteristics of each capture technique according to these properties (top part of the table). The bottom part of the table specifies the applicability or these techniques to the Constant/variable identification and Static statements reconstruction procedures described in Section 5.

Table 1: The capture techniques and their characteristics

	Stat. anal. of static SQL	Stat. anal. of dynamic SQL	Tracing statements	Tracing aspects	API overloading	API substitution	DBMS log	Compile sys. tables	SQL triggers	SQL proc.
Static/dynamic SQL	S	D	SD	D	D	D	SD	S	SD	SD
Static/dynamic technique	S	S	D	D	D	D	D	S	D	D
Information captured										
unbound static stmt		±X	X	X	X	X*		Xs		X
bound static stmt	Xs	±X	X*	X*	X*	X*				
instantiated static stmt		±X	X*	X*	X*	X				X
extracted data/status			X	X	X	X	X			X
program point	Xs	±X	X	X						
DDL stmt	Xs	±X	X	X	X	X	X	Xs	±X	X
extraction stmt	Xs	±X	X	X	X	X	X	Xs		X
modification stmt	Xs	±X	X	X	X	X	X	Xs	±X	X
control stmt	Xs	±X	X	X	X	X	X	Xs		X
binding stmt		±X	X	X	X	X				
Completeness (static stmt)	Xs							Xs		
Host language										
COBOL	Xs	X	X	(X)		X	X	Xs	X	X
C	Xs	X	X			X	X	Xs	X	X
C++	Xs	X	X	X	X	X	X	Xs	X	X
Java	Xs	X	X	X	X	X	X	Xs	X	X
Operational charact.										
schema examination	X	X						X	X	X
schema modification									X	X
code examination	X	X	X							
code modification			X		X					
code recompilation			X	X	X	X				
preparation cost	2	2	2	1	1	2	0	0	2	2
client runtime cost	0	0	1	1	1	1	0	0	0	0
server runtime cost	0	0	0	0	0	0	0	0	1	1
processing time	na	na	RT	RT	RT	RT	D	D	RT	RT
Variable constants in CV	X	±X	X*	X*	X*	X*	X*	X	±X	±X
I/O host variables of a CV	X	±X	±X	±X	±X	±X	-	-	-	-
Instantiated stmts in CV	-	-	X	X	X	X	X	-	±X	±X
Bound stmts in CV	X	±X	±X	±X	±X	±X	-	-	-	-
Unbound stmts in CV	X*	±X	X*	X*	X*	X*	X*	x	±X	±X
Family of CV equiv. to a ¬CV	na	±X	±X	±X	±X	±X	±X	na	na	±X

Conventions: **X**: available/applicable. **(X)**: studied or prototyped but not commercially available. **X***: available, possibly through post-processing. **±X**: may be available/applicable in certain conditions; in SQL triggers, possibly degenerated statements. **Xs**: available in static SQL only. na: not applicable. **RT**: real-time. **D**: deferred time. **CV**: constant variability pattern. **¬CV**: column/table/condition/action variability pattern.

A JDBC fragment ...

```
12   query = "select CustAddress from CUSTOMER where CustNum = ?";
13   SQLstmt = connection.prepareStatement(query);
14   SQLstmt.setInt(1, vCNUM);
15   rset = SQLstmt.executeQuery();
16   rset.next();
17   vADDRESS = rset.getString(1);
```

A corresponding execution trace...

```
 Event        File      Line#  Statement#   Details
SQLprep;  MyClass.java;  13;   28064776;  select CustAddress from CUSTOMER where CustNum = ?
SQLset ;  MyClass.java;  14;   28064776;  setInt(1, "C400")
SQLexec;  MyClass.java;  15;   28064776;  executeQuery()
SQLget ;  MyClass.java;  17;   28064776;  getString(1) = '10 Downing Street, London, SW1A 2AA'
```

Figure 6: A JDBC code fragment together with a corresponding execution trace

7 Applications

In this section, we show how dynamic analysis of SQL statements may contribute to the reengineering of data-intensive program and to database reverse engineering. For each application, we illustrate the use of trace analysis and we elaborate on the way the trace analysis results may be interpreted according to the objective.

7.1 Dynamic-to-static SQL reengineering of a Java program

SQLJ was designed to compensate for the poor readability of JDBC program patterns. For example, the dynamic code section of Query 5 could be rewritten as the following static SQLJ statement:
`#sql{Select OrdNum into :Num from ORDERS where Sender = :CNum}`
Statement preparation, input variable binding, execution and result extraction statements are merged into a single statement that explicitly shows the query architecture as well as the input and output host variables. The resulting program is much more readable and far easier to maintain.

Figure 6 shows a JDBC code fragment together with a corresponding execution trace obtained with a tracing aspect. Each entry of the trace contains the object id of the `Statement` involved in the query execution (such an id is obtained via `SQLstmt.hashCode()`). This allows the correct recovery of all the steps involved in a given query execution at a specific program point. By combining and exploiting the information provided by each inter-related event, both the query instance and, as a second step, the static query can be reconstructed. We illustrate this process by analyzing the trace of Figure 6. We first notice that a query was executed (`SQLexec` event). Based on the statement number, we can go backward in the trace to understand the way the executed query was constructed. The `SQLprep` trace entry provides us with a query string including an input value placeholder, while the `SQLset` entry reveals the source code location where this placeholder is replaced by an actual input value using a `setInt` method call. At this point, we are able to produce the query instance which was actually executed at line 15: `select CustAddress from CUSTOMER where CustNum = 'C400'`. Concerning the static form of the query, we can derive from the trace that method `executeQuery()` of line 15 actually executes an SQL statement of the form `select CustAddress from CUSTOMER where CustNum =` v, where v is the variable/constant used as second argument of method `setInt` of line 14. This line shows that v actually is host variable (`vCNUM`). Similarly, the `into` clause of the static query can be obtained by retrieving the variable to which the result of method `getString(1)` of line 17 is assigned (`vADDRESS`). We now obtain the following complete static SQLJ query:

```
#sql {select CustAddress into :vADDRESS from CUSTOMER where CustNum = :vCNUM}
```

7.2 Recovering Database Implicit Constraints

As illustrated in the introduction, analyzing SQL queries can help eliciting implicit database schema constructs and constraints. Inter-query dependencies can also be used to recover constraints like undeclared foreign keys, identifiers and functional dependencies [PTK95, LPT99, TLG02, TZ03]. Below, we will illustrate the use of inter-query, intra-execution trace analysis as a basis for formulating hypotheses on the existence of a foreign key. These hypotheses will still need to be validated afterwards (e.g., via data analysis).

We can distinguish two different approaches to detecting implicit referential constraints between columns of distinct tables (i.e., foreign keys). We can either observe the way such referential constraints are *used* or the way they are *verified.*

- Referential constraint usage (RCU): within the same execution, an output value o_1 of an SQL statement s_1 querying table T_1 is used as an input value of another SQL statement s_2 accessing table T_2. This suggests the existence of an implicit foreign key between tables T_1 and T_2.

- Referential constraint checking (RCC): before modifying the content of a table T_2 (by an `insert` or `update` statement s_2), the program always executes a verification query q_1 on table T_1. According to the result of q_1, s_2 is executed or not. When both q_1 and s_2 are executed, they contain at least one common input value. Conversely, before deleting a row of a table $T1$ using a `delete` statement d_2, the program always deletes a possibly empty set of rows of another table T_2 via a previous statement d_1 (procedural delete cascade).

As an example, let us consider the execution trace given in Figure 7. This trace strongly suggests the existence of an implicit foreign key between column `Sender` of table `ORDERS` and column `CustNum` of table `CUSTOMER`. Indeed, each row insertion on table `ORDERS` is preceded by the execution of a query that:

1. counts the number of rows of table `CUSTOMER` having c as value of column `CustNum`, where c corresponds to the value of column `Sender` of the

```
SQLexec; 21504776; select count(*) from customer where CustNum = 128
SQLget ; 21504776; getInt(1) = 1
SQLprep; 22356780; insert into orders(OrdNum, OrdDate, Reference, Sender) values(?,?,?,?)
SQLset ; 22356780; setInt(1, 456)
SQLset ; 22356780; setDate(2, '2007-12-21')
SQLset ; 22356780; setString(3,'An order reference')
SQLset ; 22356780; setInt(4, 128)
SQLexec; 22356780; executeQuery()
...
SQLexec; 25150611; select count(*) from customer where CustNum = 152
SQLget ; 25150611; getInt(1) = 0
...
SQLexec; 27232456; select count(*) from customer where CustNum = 251
SQLget ; 27232456; getInt(1) = 1
SQLprep; 27366704; insert into orders(OrdNum, OrdDate, Reference, Sender) values(?,?,?,?)
SQLset ; 27366704; setInt(1, 457)
SQLset ; 27366704; setDate(2, '2007-12-21')
SQLset ; 27366704; setString(3,'Another order reference')
SQLset ; 27366704; setInt(4, 251)
SQLexec; 27366704; executeQuery()
```

Figure 7: An execution trace that may reveal the existence of an implicit foreign key (cfr. Figure 1)

inserted `ORDERS` row;

2. returns 1 as a result.

8 Conclusions and Perspectives

SQL statements appear to be a particularly rich source of information in program and data structure understanding and therefore their processing must improve such essential processes as program and database maintenance, evolution and reengineering. Though some encouraging results have been obtained in the 90's, particularly to support database reverse engineering, systematically exploring and mastering this information source still is a largely unexplored research and technical domain. The goal of this paper is, quite modestly, to mark this engineering domain out by identifying and discussing its basic concepts, its specific techniques and some of its representative applications. It particularly explores the role of dynamic analysis in SQL statement understanding. This study leads to three important conclusions on the relationships between dynamic and static analyses:

- dynamic SQL analysis can be used to complete and refine incomplete results provided by static analysis (e.g., to detect variables shared by two SQL queries);
- in many cases, both static and dynamic SQL analysis must be used in an interleaved way (e.g., to identify host variable names when reconstructing bound static SQL);
- dynamic SQL analysis can provide information that is out of scope of static analysis, specially when processing dynamic SQL.

Concerning the capture techniques, some of them are quite light though highly informative, such as DBMS log examination, that is available for free. For pure dynamic SQL, the four tracing techniques (tracing statement injection, tracing aspects and API transformations) appear quite powerful. Among them, the development of a tracing aspect in AspectJ has proved particularly easy and flexible, allowing rapid adjustement of the tracing process. Its second advantage is its isolation from the source code as compared with statement injection. Finally it requires no internal knowledge on the DBMS drivers as in the last two tracing techniques.

This paper basically is exploratory. While the authors have the feeling that the framework they have designed describes adequately this software analysis domain, much remains to be done to validate, evaluate and instrument the techniques they propose. First case studies have shown the feasability of all the techniques on academic data-intensive applications. Five paths of further research have been identified and will be explored in the future.

1. Testing the techniques on real world applications to validate the framework. In particular to evaluate the runtime penalty of the capture techniques and to measure the scalability of the techniques.

2. Evaluation of the completeness of the constant variability patterns derived from the column/table/condition/action variability patterns.

3. Evaluating the precision of the capture and processing techniques and refining them.

4. Estimating the development effort in both time and skill.

5. Developing supporting tools and integrating them in CASE environments.

References

[And98] Martin Andersson. Searching for semantics in cobol legacy applications. In Stefano Spaccapietra and Fred J. Maryanski, editors, *Data Mining and Reverse Engineering: Searching for Semantics, IFIP TC2/WG2.6 Seventh Conference on Database Semantics (DS-7)*, volume 124 of *IFIP Conference Proceedings*, pages 162–183. Chapman & Hall, 1998.

[BP95] M. R. Blaha and W. J. Premerlani. Observed idiosyncracies of relational database designs. In *Proc. of the Second Working Conference on Reverse Engineering (WCRE'95)*, page 116, Washington, DC, USA, 1995. IEEE Computer Society.

[BS95] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems. Gateways, Interfaces, and the Incremental Approach.* Morgan Kaufmann Publishers, 1995.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CHH06] Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[DK98] A. Van Deursen and T. Kuipers. Rapid system understanding: Two cobol case studies. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.

[ES01] Suzanne M. Embury and Jianhua Shao. Assisting the comprehension of legacy transactions. In *Proc. of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 345, Washington, DC, USA, 2001. IEEE Computer Society.

[Hai02] J.-L. Hainaut. Introduction to database reverse engineering. LIBD Publish., 2002. http://www.info.fundp.ac.be/ dbm/publication/2002/DBRE-2002.pdf.

[HEH+98] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, and J.-L. Hainaut. Program understanding in databases reverse engineering. In G. Quirchmayr, E. Schweighofer, and T. Bench-Capon, editors, *Proc. of 9th Int. Conf. on Database and Expert Systems Applications (DEXA'98)*, volume 1460 of *Lecture Notes in Computer Science*, pages 70–79. Springer, 1998.

[HMCM93] J.-L. Hainaut, Chandelon M., Tonneau C., and Joris M. 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.

[HvdBV07] Rob van der Leek Huib van den Brink and Joost Visser. Quality assessment for embedded sql. In Michael Godfrey and Bogdan Korel, editors, *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE Computer Society, 2007.

[LD05] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *Proceedings of Aspect-Oriented Software Development (AOSD 2005)*. ACM Press, March 2005. 12 pages.

[LPT99] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovery of "interesting" data dependencies from a workload of sql statements. In *Proc. of the Third European Conference on Principles of*

Data Mining and Knowledge Discovery (PKDD'99), pages 430–435, London, UK, 1999. Springer-Verlag.

[Man03] Mika Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study.* PhD thesis, Helsinki University of Technology, 2003.

[MLA06] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. Insider and outsider threat-sensitive sql injection vulnerability analysis in php. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 147–156, Washington, DC, USA, 2006. IEEE Computer Society.

[NT08] Minh Ngoc Ngo and Hee Beng Kuan Tan. Applying static analysis for automated extraction of database interactions in web applications. *Inf. Softw. Technol.*, 50(3):160–175, 2008. to appear.

[PKBT94] Jean-Marc Petit, Jacques Kouloumdjian, Jean-Francois Boulicaut, and Farouk Toumani. Using queries to improve database reverse engineering. In *Proc. of the 13th International Conference on the Entity-Relationship Approach (ER'94)*, pages 369–386, London, UK, 1994. Springer-Verlag.

[Pön95] Richard Pönighaus. 'favourite' sql-statements - an empirical analysis of sql-usage in commercial applications. In *Proc. of the sixth International Conference on Information Systems and Management of Data*, volume 1006 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 1995.

[PTK95] Jean-Marc Petit, Farouk Toumani, and Jacques Kouloumdjian. Relational database reverse engineering: A method based on query analysis. *Int. J. Cooperative Inf. Syst.*, 4(2-3):287–316, 1995.

[SLF$^+$01] Jianhua Shao, Xingkun Liu, G. Fu, Suzanne M. Embury, and W. A. Gray. Querying data-intensive programs for data design. In *Proc. of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, pages 203–218, London, UK, 2001. Springer-Verlag.

[SLGC94] Oreste Signore, Mario Loffredo, Mauro Gregori, and Marco Cima. Reconstruction of er schema from database applications: a cognitive approach. In *Proc. of the13th International Conference on the Entity-Relationship Approach (ER'94)*, pages 387–402, London, UK, 1994. Springer-Verlag.

[TLG02] Hee Beng Kuan Tan, Tok Wang Ling, and Cheng Hian Goh. Exploring into programs for the recovery of data dependencies designed. *IEEE Trans. Knowl. Data Eng.*, 14(4):825–835, 2002.

[TZ03] Hee Beng Kuan Tan and Yuan Zhao. Automated elicitation of inclusion dependencies from the source code for database transactions. *Journal of Software Maintenance*, 15(6):379–392, 2003.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[WES04] David Willmor, Suzanne M. Embury, and Jianhua Shao. Program slicing in the presence of a database state. In *Proc. of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 448–452, Washington, DC, USA, 2004. IEEE Computer Society.

[YC99] Hongji Yang and William C. Chu. Acquisition of entity relationship models for maintenance-dealing with data intensive programs in a transformation system. *J. Inf. Sci. Eng.*, 15(2):173–198, 1999.