Faults and Failures in SQL-based Data Manipulation Programming *

PLÍNIO DE SÁ LEITÃO JÚNIOR[1]
PLÍNIO ROBERTO SOUZA VILELA[2]
MARIO JINO[3]

[1] DCC-UFLA
Federal University of Lavras
Cx Postal 3037 - CEP 37200-000
Lavras, MG, Brazil
psleitao@dcc.ufla.br

[2] Methodist Univ. of Piracicaba
Piracicaba, SP, Brazil
prvilela@unimep.br

[3] DCA-FEEC-Unicamp
State University of Campinas
Campinas, SP, Brazil
jino@dca.fee.unicamp.br

Abstract. Database applications, including SQL-based applications, have received little attention directed towards improving the knowledge of their possible faults. This paper deals with issues related to software faults and failures aiming at understanding what types of faults occur in SQL manipulation commands, and how they are propagated to the output of command execution. SQL manipulation commands are studied and their structure is organized into structural items, a step towards understanding and grouping fault types. A list of manipulation fault types is determined and presented with SQL command examples. Failure dimensions are discussed along with query and state changing operations. An experiment to abstract the types of manipulation faults for SQL was carried out and the results are presented. The experiment built databases and faulty commands to promote failure in command execution. A database was built and a set of faulty SQL commands used to map fault types and failure dimensions. The analysis of data mapping indicates: i) there is a many-to-many mapping between faults and failures; ii) failure dimensions are dependent on fault type, faulty command, and the database itself; and iii) manipulation fault knowledge is crucial for SQL programming and testing of database applications. This work represents an initial step for testing SQL programming.

Keywords: Software Testing; Fault Enumeration; SQL-based Applications; Database Application Testing; Software Failures.

(Received August 06, 2007 / Accepted November 12, 2007)

1 Introduction

Software faults are present in software production. Faults are introduced in any phase of the software development process. One of the reasons is that humans carry out most of this process frequently using human made tools. Humans make mistakes that are reflected in latent software faults. When those faults are exercised they yield inconsistent software states, which may flourish as failures. That occurs when the inconsistent state make the software output deviate from its specification.

Faults may exist causing no harm if never reached. When a software input causes the program execution to reach the software fault a failure may occur. The same fault may yield different types of failure, depending on the particular software input used. The type of the fault, plus the particular software input used, influences the type of failure. Failures cause damage, frustration and monetary losses, so they must be kept to a minimum in deployed software. To minimize failures, faults ought to be identified and removed.

Fault details are dependent on the underlying programming language. To reach the desired software quality it is imperative to understand the relationship between faults and failures in a specific programming language. That is intrinsically associated to the knowledge of the language's syntactical construction and its common usage to achieve the desired semantics.

Testing is the process of executing a program to make it fail. It is the search for program inputs that contradict the assertion that the program is correct. When a failure is perceived, there is a chance to look for and remove its causing fault (debugging), which contributes to the quality improvement of the software. The knowledge of faults and failures is pertinent to promote program development with lesser faults, guide program

*This work was partially supported by a CNPq research grant.

debugging and nourish the development of testing approaches, towards quality improvement and cost reduction.

In recent years, a wide range of traditional software testing techniques have been proposed, implemented, and evaluated. However, relatively little effort has been made to develop systematic techniques towards the quality improvement of the database application programs [2, 8]. A database application is a program whose environment always contains one or more databases [8]. The main motivation of this work is the quality improvement of SQL-based applications. An SQL-based application is one that interacts with the databases using SQL (*Structured Query Language*), a well-known language that is extensively used by the database community [7, 6].

Differently from what happens with imperative languages such as C and $Java$, SQL have received little attention from the academic community. It is a paradox since there is a large amount of source code written in SQL that needs systematic testing methods. SQL is used in database applications that deal with real problems, remaining the most accepted and implemented interface language for relational database systems [5].

A data manipulation operation modifies the database state or queries the database to retrieve and format information. Such direct interactions between the program and the database are materialized by specific SQL commands, also mentioned as data manipulation commands. The testing terminology is extended to single SQL manipulation commands. A fault or likewise a defect is commonly defined as a single item in an SQL command that is incorrect. A faulty command is an SQL command, which has at least one fault. An error is an inconsistent execution state caused by the execution of a faulty command. A failure occurs when an error propagates to the outside of the faulty SQL command resulting in behavior that deviated from the expected one [1]. Failure dimensions abstract the ways where failures are manifested in faulty manipulation commands. To conclude, faults are the causes, failures are the symptoms, and failure dimensions are symptom types.

This paper discusses issues arising in data manipulation faults and failures, and presents the results of an investigation aiming at understanding the relationship between faults and failures. The structure of SQL manipulation commands are extracted from their basic constructions and organized in structural items. These items represent a step towards understanding and grouping faults to evaluate their propagation to outside of the command. The concept of manipulation failure is examined and two data sets are introduced to capture the notion of command output. Failure dimensions are discussed along with query and state changing operations. A list of manipulation fault types is presented by SQL command examples in a self-explanatory fashion. A set of faulty SQL commands has been used in an experiment to build a mapping between fault types and failure dimensions. The mapping data are discussed and the analysis results indicate that: i) there is a many-to-many mapping between faults and failures; ii) failure dimensions are dependent on fault type, faulty command, and the database itself; and iii) manipulation fault knowledge is crucial for SQL programming and testing of database applications.

The remainder of the paper is organized as follows. Section 2 describes related work dedicated to enumerate software faults. SQL manipulation commands are presented in Section 3, with special attention to structural items extracted from the language constructions. Section 4 presents manipulation fault types in a self-explanatory fashion. Section 5 focuses on data manipulation failure dimensions, defining two data sets related to the command output, and characterizing incorrect command behavior. Section 6 explores the building of a mapping aiming at analyzing the relationship between faults and failures. Section 7 highlights the lessons learned from building the mapping and extracts knowledge of the relationship between manipulation faults and failures. Section 8 concludes this work, showing the contributions and mentioning future work.

2 Related Work

Software fault enumeration, or fault scheme, is built according to specific purposes, such as identifying causes of errors, making decisions during software development, developing profiles of software development methodologies, and guiding the testing activity. These schemes are either simple or more elaborate. For instance, a simple scheme groups faults as major or minor, and a more detailed scheme refines the fault description by establishing successive levels such that the level above is divided into finer categories.

Chillarege [3] presented a classification scheme, financed by IBM, intending to provide analysis and feedback of the software process that deal with fault detection, correction and prevention. The goal is to provide a measurement paradigm to extract information from faults and use that information to assess some parts of a software development process to provide corrective actions for that process. The original proposal defines an orthogonal defect classification (ODC) that involves

[1] The notion of expected command behavior is extended from the expected software functionality in the software specification

identifying a trigger, a type, and a qualifier, for each fault. The trigger indicates the event that prompted the fault discovery. The type identifies the fault category, such as assignment, interface and algorithm. The qualifier involves adjectives such as missing or incorrect. IBM has been improving this scheme by including new fault dimensions in addition to the three original ones, while software techniques have evolved. Some dimensions are identified at the time of fault discovery and others after the fault has been fixed.

Based on Chillarege [3], Kelly and Shepard [9] developed a fault classification scheme to evaluate and compare the effectiveness of software inspection techniques. According to the authors, the IBM ODC was not adequate for inspection purposes, since the fault types have to reflect the problem at the same time the inspector perceives it in the code: the defect type must relate to the code rather than the fixing activity. Related to the IBM ODC, the number of fault types is substantially expanded. In addition, other fault dimensions are modified to be more adequate for the program inspection demands than IBM ODC. For instance, new fault qualifiers were added, such as superfluous and obscure. The fault scheme results from the analysis of findings from the inspection experiments.

In spite of the subjectivity removing intention, both schemes above are defined using subjective judgment. The former tries to answer the question "what were you thinking about when you discovered the defect?"; the later is related to the question "what task were you carrying out when you discovered the defect?". They are based on the different levels of understanding that finding represents.

A more complete enumeration of faults was proposed in [1]. The major categories encompass functional faults, passing by integration and internal interfaces, up to test definition or execution faults. Each major category is detailed in levels aiming at properly including sufficient description for all possible faults. This scheme serves as a starting point and a point of reference for building new fault schemes.

According to Beizer [1], there is no universally correct way to enumerate and categorize faults. Faults are difficult to enumerate, and a given fault can be put into one or another category depending on its history and tester viewpoint. Furthermore, the inherent aspects of programming languages determine the fault list and characterize the programming fault propensity. More important than adopting the "right" fault enumeration and classification is using any fault scheme on which to base testing strategies.

Even through there is research related to fault enumeration, up to the date of writing this article no software testing publication focused on enumeration of data manipulation faults and failures in SQL-based applications. This paper does not intend to build the framework of fault and failure enumeration, but a resource to help understand the causes and consequences of incorrectness in SQL manipulation commands.

3 Structure of Data Manipulation Commands

The general structure of the SQL manipulation commands is presented in this section and was inspired by a theoretical study of the manipulation commands based on SQL foundations, such as mentioned by Fortier [7] and Elmasri and Navathe [6]. One can validate this structure by exploring the essence of the manipulation commands regardless of syntactical details.

Data manipulation is implemented by *insert*, *delete*, *update* and *select* commands, to respectively manipulate databases by inserting, excluding, modifying and retrieving operations. The *select* command is mentioned as query database command (or only query command) and the other ones as state database changing commands (or only state changing commands).

The general structure of *select* command is made up of the $[s_1]$ to $[s_6]$ items as shown in Table 1. The simplest form of the *select* command is made up of the $[s_1]$ and $[s_2]$ items; the other items are used according to the query semantics.

To illustrate the *select* command semantics, consider the command example in Table 1.The query uses the *empl* table as a data source (item $[s_2]$). The selection mechanism chooses employees whose salary, together with their bonus, is greater than 1050 (item $[s_3]$). These employees are grouped based on their salary; in other words, the employees of each group have the same *salary* attribute value (item $[s_4]$). The groups that have more than one tuple are then selected (item $[s_5]$). For the remaining employee groups, the salary, tuple quantity, and sum of bonus are abstracted (item $[s_1]$). The abstracted data are sorted by salary in descending order (item $[s_6]$).

The structure of the commands, whose execution can modify the database state, is presented in Table 2. This table also includes an example for each command, aiming at illustrating the structural items. Note that the *insert* command has two possible constructions: one that explicitly mentions the data values and another where data values are obtained from the execution of a subquery.

The structural items based on SQL foundations are in no way complete or detailed, the purpose is to capture the command semantics with no regard to syntactical

Table 1: Structure of the select command.

Structural item	Structural item description	Command example
[s_1]	An ordered list of expressions used to compute the value of returned attributes	select salary, count(*), sum(bonus)
[s_2]	A list of table names used as data source	from empl
[s_3]	A predicate used for tuple selecting	where (salary + bonus) $>$ 1050
[s_4]	A list of expressions used for data grouping	group by salary
[s_5]	A predicate used for data group selecting	having count(*) $>$ 1
[s_6]	An ordered list of arguments used for data sorting	order by salary desc

Table 2: Structures of the insert, delete and update commands.

Command	Structural item	Structural item description	Command example
insert (1)	[i_1]	A table name	insert into empl
	[i_2]	An ordered list of attributes	(emplno, name, salary, bonus)
	[i_3]	An ordered list of attribute values	values (1234 , 'ana' , 1060 , 35)
insert (2)	[i_1]	A table name	insert into empl
	[i_2]	An ordered list of attributes	(emplno, name, salary, bonus)
	[i_4]	A subquery	select custno, name, salary, 0 from customer
delete	[d_1]	A table name	delete from empl
	[d_2]	A predicate used for tuple selecting	where (salary + bonus) $>$ 1050
update	[u_1]	A table name	update empl
	[u_2]	An attribute value assignment list	set salary = salary * 1.01, bonus = bonus * 1.10
	[u_3]	A predicate used for tuple selecting	where (salary + bonus) $>$ 1050

details and alternatives. Some clauses, such as union and intersect, are not covered by the proposed data manipulation structure since the major interest is exploring faults in basic constructions for each manipulation command. The subqueries are queries implicitly or explicitly embedded in any manipulation command, their structure is the same as presented in Table 1.

4 Fault Type List

Software fault enumeration is the basis to study fault-proneness and fault distribution. Enumerating faults requires knowledge on constructions of the corresponding programming language. This section introduces the fault type list for SQL manipulation commands. Based on command structures listed in Tables 1 and 2, a set of faulty commands is built for each structural item. This set attempts to cover all possible mistakes for each structural item, by exercising different syntactical constructions.

Tables 3, 4, 5, and 6 correspond to the fault type lists of the *select*, *insert*, *delete* and *update* commands, respectively. In these tables, each fault type is identified by *[x_m]-n*, where: x_m refers to the structural item as shown in Tables 1 and 2; and *n* is a sequential number for the fault types related to the same structural item. For each fault type, an identifier, a description, and an example are presented in these tables. The example illustrates a correct and an incorrect version of the manipulation command. One can understand the fault type essence by observing the fault description and comparing the two versions of the command example. The schema of the database used in the examples is very intuitive and is omitted from this discussion.

A qualifier could be associated with each fault type, such as *missing*, *inconsistent* and *wrong*. For example, fault types *[s_1]-1* and *[s_1]-4* concern mistakes related to the order list of expressions used to compute the value of the returned attributes; nevertheless, they have different qualifiers, *missing* and *wrong*, respectively. However, extending the fault type description by including the qualifier semantics is a better way of capturing the fault essence and provides a better comprehension of its semantics.

One pertinent question is how to check whether failures were produced in response to the execution of faulty manipulation commands. The tester must evaluate the data returned from the execution of each query command. For state changing commands, no data are returned by the command execution and the tester must pay attention to the state of the database after the command execution, specifically the state of the underlying relations.

The *select* command has a more complex and extensive structure than the *insert*, *delete* and *update* commands, as one can see from Tables 1 and 2; thus, it re-

Table 3: Fault type list for the *select* command.

Fault ID	*Description*	*Correct command example*	*Incorrect command example*
[s_1]-1	An expression is missing from the ordered list of expressions used to compute the value of returned attributes.	select emplno, name, salary from empl	select emplno, salary from empl
[s_1]-2	An expression is unduly present in the ordered list of expressions used to compute the value of returned attributes.	select emplno, name, salary from empl	select emplno, name, salary, salary + bonus from empl
[s_1]-3	The order of expressions in the ordered list of expressions used to compute the value of returned attributes is wrong.	select emplno, name, salary, salary + bonus from empl	select emplno, name, salary + bonus, salary from empl
[s_1]-4	A wrong expression is in the ordered list of expressions used to compute the value of returned attributes.	select emplno, name, salary from empl	select emplno, name, bonus * 0.5 from empl
[s_2]-1	A table name is missing from the list of table names used as a data source.	select e.name, e.salary from empl e, dept d	select e.name, e.salary from empl e
[s_2]-2	A table name is unduly present in the list of table names used as a data source.	select e.name, e.salary from empl e	select e.name, e.salary from empl e, dept d
[s_2]-3	A wrong table name is in the list of table names used as a data source.	select e.name, d.salary from empl e, dept d	select e.name, d.salary from empl e, depn d
[s_3]-1	The predicate used for tuple selecting is missing.	select emplno, salary from empl where salary $>$ 1050	select emplno, salary from empl
[s_3]-2	The predicate used for tuple selecting is unduly present.	select emplno, salary from empl	select emplno, salary from empl where salary $>$ 1050
[s_3]-3	The predicate used for tuple selecting is wrong.	select emplno, salary from empl where salary $>$ 1050	select emplno, salary from empl where salary $<$ 1200
[s_4]-1	The list of expressions used for data grouping is missing.	select salary from empl group by salary	select salary from empl
[s_4]-2	The list of expressions used for data grouping is unduly present.	select salary from empl	select salary from empl group by salary
[s_4]-3	An expression is missing from the list of expressions used for data grouping.	select salary, count(*) from empl group by salary, bonus	select salary, count(*) from empl group by salary
[s_4]-4	An expression is unduly present in the list of expressions used for data grouping.	select salary, count(*) from empl group by salary	select salary, count(*) from empl group by salary, bonus
[s_4]-5	A wrong expression is in the list of expressions used for data grouping.	select salary, count(*) from empl group by salary, bonus	select salary, count(*) from empl group by salary, salary - bonus
[s_5]-1	The predicate used for data group selecting is missing.	select salary, count(*) from empl group by salary having count(bonus) $>$ 1	select salary, count(*) from empl group by salary
[s_5]-2	The predicate used for data group selecting is unduly present.	select salary, count(*) from empl group by salary	select salary, count(*) from empl group by salary having count(bonus) $>$ 1
[s_5]-3	The predicate used for data group selecting is wrong.	select salary, count(*) from empl group by salary having count(bonus) $>$ 1	select salary, count(*) from empl group by salary having count(salary) $>$ 1
[s_6]-1	The ordered list of arguments used for data sorting is missing.	select emplno, name, salary from empl order by salary	select emplno, name, salary from empl
[s_6]-2	An ordered list of arguments used for data sorting is unduly present.	select emplno, name, salary from empl	select emplno, name, salary from empl order by salary
[s_6]-3	The order of the arguments in the ordered list of arguments used for data sorting is wrong.	select emplno, name, salary from empl order by salary, name	select emplno, name, salary from empl order by name, salary
[s_6]-4	A wrong argument is in the ordered list of arguments used for data sorting.	select emplno, name, salary from empl order by salary, name	select emplno, name, salary from empl order by salary desc, name

Table 4: Fault type list for the *insert* command.

Fault ID	*Description*	*Correct command example*	*Incorrect command example*
[i_1]-1	The table name is incorrect.	insert into empl (emplno, name, salary) values (8888, 'mary smith', 1050)	insert into depn (emplno, name, salary) values (8888, 'mary smith', 1050)
[i_2]-1	An attribute is missing from the ordered list of attributes.	insert into empl (emplno, name, salary, bonus) values (8888, 'mary smith', 1050, 12)	insert into empl (emplno, name, salary) values (8888, 'mary smith', 1050)
[i_2]-2	An attribute is unduly present in the ordered list of attributes.	insert into empl (emplno, name, salary) values (8888, 'mary smith', 1050)	insert into empl (emplno, name, salary, bonus) values (8888, 'mary smith', 1050, 12)
[i_3]-1	A wrong value is in the ordered list of attribute values.	insert into empl (emplno, name, salary, bonus) values (8888, 'mary smith', 1050, null)	insert into empl (emplno, name, salary, bonus) values (8888, 'mary smith', 1050, 22)
[i_3]-2	The order of the values in the ordered list of attribute values. is wrong.	insert into empl (emplno, name, salary, bonus) values (8888, 'mary smith', 1050, 22)	insert into empl (emplno, name, salary, bonus) values (8888, 'mary smith', 22, 1050)
[i_4]-1	There is a fault in the subquery.	insert into empl (emplno, name, salary) select 9999, name, salary from depn where emplno = 1111	insert into empl (emplno, name, salary) select 9998, name, salary from depn where emplno = 1111

Table 5: Fault type list for the *delete* command.

Fault ID	*Description*	*Correct command example*	*Incorrect command example*
[d_1]-1	The table name is incorrect.	delete from empl where name like 'ann%'	delete from depn where name like 'ann%'
[d_2]-1	The predicate used for tuple selecting is missing.	delete from empl where name like 'ann%'	delete from empl
[d_2]-2	The predicate used for tuple selecting is unduly present.	delete from empl	delete from empl where name like 'ann%'
[d_2]-3	The predicate used for tuple selecting is wrong.	delete from empl where name like 'ann%'	delete from empl where salary between 800 and 1060

Table 6: Fault type list for the *update* command.

Fault ID	*Description*	*Correct command example*	*Incorrect command example*
[u_1]-1	The table name is incorrect.	update empl set salary = salary * 1.1 where salary $<$ 550	update depn set salary = salary * 1.1 where salary $<$ 550
[u_2]-1	An assignment is missing from the attribute value assignment list.	update empl set salary = salary * 1.1, bonus = bonus * 1.1 where salary $<$ 550	update empl set salary = salary * 1.1 where salary $<$ 550
[u_2]-2	An assignment is unduly present in the attribute value assignment list.	update empl set salary = salary * 1.1 where salary $<$ 550	update empl set salary = salary * 1.1, bonus = bonus * 1.1 where salary $<$ 550
[u_2]-3	A wrong expression is in the right side of an assignment in the attribute value assignment list.	update empl set salary = salary * 1.1 where salary $<$ 550	update empl set salary = 1.1 where salary $<$ 550
[u_2]-4	A wrong attribute is in the left side of an assignment in the attribute value assignment list.	update empl set salary = salary * 1.1 where salary $<$ 550	update empl set bonus = salary * 1.1 where salary $<$ 550
[u_3]-1	The predicate used for tuple selecting is missing.	update empl set bonus = bonus * 1.1 where salary $<$ 550	update empl set bonus = bonus * 1.1
[u_3]-2	The predicate used for tuple selecting is unduly present.	update empl set salary = salary * 1.1	update empl set salary = salary * 1.1 where salary $<$ 550
[u_3]-3	The predicate used for tuple selecting is wrong.	update empl set salary = salary * 1.1 where salary $<$ 550	update empl set salary = salary * 1.1 where salary $>$ 1200

sults in a numerous fault type list. State changing commands which embed at least one subquery inherits the fault type list from the query commands, as the structure of subqueries are the same of the *select* command.

An initially assumed fault type for the *select* command was the wrong order of expressions in the list used for data grouping in the select command. It was observed that these expressions are used together as if they were an expression concatenation and this order is not significant to compute the value of returned attributes.

Fault types related to the structural item *[s_6]*, *[s_6]-1* to *[s_6]-4*, consider no implicit sorting mechanism. For instance, some implementations automatically order data by expressions in the ordered list used for data grouping. However, if a data ordering is expected, the command must have an explicit sorting clause.

The structural items represent a way towards knowledge improvement and grouping fault types. The fault type list is the first element of a comprehensive study towards manipulation faults and their implications in SQL-based applications. The following sections present an investigation to define failure dimensions and mapping them to fault types is in progress, aiming at understanding how an error caused by a specific fault type is propagated as the command output.

5 Manipulation Failure Dimensions

To understand the effects of the execution of faulty manipulation commands, two sets are defined in this section, aiming at capturing the notion of the command output. They are two-dimensional tabular sets, but are not relations according to the relational theory [4], as may have duplicate lines. Nonetheless, they are described using relational terminology, such as tuple and attribute.

Each manipulation command execution is associated to two set versions: the expected and the obtained. If the obtained one is distinct from the expected one in some dimension, then the fault was propagated to the command output and a failure has manifested. The dimensions where the obtained version differs from the expected version are called *failure dimensions*. Failure dimensions are related to the following aspects of a data set. The *attribute list* describes the attributes for the manipulation command output; this dimension is related to the attribute domain and the attribute semantics. The *attribute order in the attribute list* is significant when the underlined order of attributes characterizes the expected tuples. The *number of tuples* defines the cardinality of the expected set of tuples. The *attribute value list* establishes for all tuples the expected value of each attribute without considering the attribute value order. The *tuple order* defines unambiguously the expected order of tuples, when it is relevant.

The execution of a *select* command results in a set of tuples called *returned tuple set*. It is described by an ordered returned attribute list, *(A_{r1}, ..., A_{rp})*, resulting from the command execution, where A_{ri}, $1 \leq i \leq p$, is mentioned as a *returned attribute*. The returned tuple set resulting from a command execution is incorrect if one or more of the above dimensions are not the expected ones.

Defined tuple set is the denomination of the whole set of tuples related to the data that was actually inserted, deleted or updated by a state changing command execution. It is described by a defined attribute list, *(A_{d1}, ..., A_{dq})*, where A_{dj}, $1 \leq j \leq q$, is mentioned as a *defined attribute*. Consider *C* to be a state changing command, the execution of which can change the state of the relation *r*. If *C* is an *insert* command, the defined attribute list is the same that describes *r*. If *C* is a *delete* command, the defined attribute list is made up of the primary key attributes of *r*. If *C* is an *update* command, of which the defined tuple set is related to the database image after the command execution, the defined attribute list is determined by the attribute value assignment list in the command together with the primary key attributes of *r*. The defined tuple set resulting from a command execution is incorrect if one or more of the following dimensions are not the expected one: *the attribute list*; *the number of tuples*; and *the attribute value list*.

6 A Mapping Between Faults and Failures

This section explores the building of a mapping aiming at analyzing the relationship between faults and failures. As a result, it is possible to know how data manipulation faults are manifested in command output, supporting the identification of common causes for failures in order to determine corrective actions to be taken [10].

The mapping is made in three steps: the first one is to understand the fault semantics based on the knowledge extracted from SQL documents, having in mind the fault effect projection on the command output; the second one is to make a mapping between faults and their potential failures, resulting in a useful resource for the testing and debugging activities; the latter is to execute faulty commands to preliminarily validate and evolve the mapping. The authors' former experience with SQL was beneficial as it provided skills in fault discovery and fixing. The main idea was to build a unique mapping for all manipulation faults, despite the

manipulation command.

6.1 Experiment

Based on faults listed in Tables 3 to 6, a set of faulty commands was elaborated for each fault type. This set was intended to cover all mistake possibilities and different syntactical constructions for the fault type.

The commands are executed aiming at observing the failure dimensions manifested. Database states were prepared to cover all failure dimensions caused by the execution of faulty commands related to the same fault type. If any failure dimension was not exercised for the fault type at that point a new database state was created by inserting new tuples until all failure dimensions are covered, or it was assumed that it was not possible to exercise the remaining failure dimensions. The same reasoning is applied if any failure dimension were not exercised for that fault type, thus a new list of faulty commands is built. Once a failure dimension is manifested to a fault type, no additional effort is spent for that failure dimension. As a result, over 130 faulty commands were executed for all fault types using an Oracle database. Fault type is unique in each faulty command and no attempt was made to minimize the number of faulty commands used in each fault type. The database was composed of 3 relations, which had 2 to 6 tuples. This database is restored before starting the evaluation of a new fault type. As a consequence, the mapping evolved to the content of Table 7. The first column lists failure dimensions related to a returned tuple set or a defined tuple set if the command is a query command or a state changing command, respectively. The second column presents, for each failure dimension, the list of fault types that cause such a failure.

In the state changing operations, the faulty commands that violate any of the integrity constraints of the database, such as primary and foreign keys, are discarded. These commands are replaced by new ones that do not violate the constraints, otherwise all fault types of state changing operations are related to the number of tuples failure dimension. If it is expected that a command execution violates database constraints, aspects related to the exception raising must be observed in the application context.

The compilation of SQL commands by the database management system eliminates a sort of fault commands, thus reducing the number of possible manipulation faults. For instance, consider the fault type [i3]-2, where the order of the values in the ordered list of attribute values is wrong. If this incorrect order results in conflicts of data types and no implicit conversion of data type could be carried out, an error message is returned by the static command checking.

Null values required special attention in command execution. A null value is treated as an unknown value and any computation using a null value yields other unknown value. A problem arises when tuples have null values for attributes that are used in *join* operations. Another problem is how to count when aggregate functions such as *count* and *sum* are applied. In some cases, the fault is the absence of a dedicated function that prevents the possibility of null values for some attributes.

7 Analysis of Data Mapping

Fault propagation is used in the command context, since the analysis focuses on failure dimensions observed in the command output. The intention is not to comment on all the mapping data, but highlight the lessons learned by building the mapping and present the results of data mapping analysis.

Even though there is a unique mapping, some aspects of query commands are distinct from state changing commands. For instance, in database state changing commands, the attributes in the attribute list failure dimension are qualified by the database relation name in addition to the attribute name itself. Fault type [i1]-1 changes the state of an incorrect relation, thus applying the state changing operation to an incorrect list of attributes. The same reasoning is extended to fault types [d1]-1 and [u1]-1. Also, faults in the attribute value assignment list of update command could change the attribute list of the defined tuple set, thus resulting in this failure dimension. In query commands, this failure dimension is related to mistakes in the ordered list of expressions used to compute the value of returned attributes.

The attribute order in the attribute list dimension is caused by a unique fault, [s1]-3. This fault is difficult to discover when the domain order of the attribute list is not changed. One may argue that it is not relevant to spend a great deal of testing efforts to reveal such a fault, because it is a unique fault causing dimension. However, the correct argument is that any fault can propagate to the command output and the subsequent damage is dependent on the fault context. Furthermore, a program under test is observed with several faults and the use of testing criteria surpasses a unique fault type discovery.

The tuple order dimension considers no implicit sorting mechanism. For instance, some implementations automatically order the returned tuple set by expressions in the ordered list used for data grouping. However, if a tuple order is expected in the returned tuple set, the command must have an explicit sorting

Table 7: The mapping between faults and failures for manipulation commands.

Failure dimension	Fault type list
Attribute list	[s_1]-1, [s_1]-2, [s_1]-4, [i_1]-1, [d_1]-1, [u_1]-1, [u_2]-1, [u_2]-2, [u_2]-4
Attribute order in the attribute list	[s_1]-3
Number of tuples	[s_2]-1, [s_2]-2, [s_2]-3, [s_3]-1, [s_3]-2, [s_3]-3, [s_4]-1, [s_4]-2, [s_4]-3, [s_4]-4, [s_4]-5, [s_5]-1, [s_5]-2, [s_5]-3, [d_2]-1, [d_2]-2, [d_2]-3, [i_4]-1, [u_3]-1, [u_3]-2, [u_3]-3
Attribute value list	[s_2]-1, [s_2]-2, [s_2]-3, [s_3]-1, [s_3]-2, [s_3]-3, [s_4]-3, [s_4]-4, [s_4]-5, [i_2]-1, [i_2]-2, [i_3]-1, [i_3]-2, [i_4]-1, [d_2]-3, [u_2]-3, [u_3]-1, [u_3]-2, [u_3]-3
Tuple order	[s_6]-1, [s_6]-2, [s_6]-3, [s_6]-4

clause. Faults [s6]-1 to [s6]-4 share the same failure dimension and the perception of their effects are proportional to the cardinality of the expected set.

The fault types causing the attribute list dimension could also be associated to the attribute value list dimension, since they result in wrong tuples (or in values of wrong attributes). This fact was omitted from the data mapping because the attribute list dimension is related to the attribute domain and the attribute semantics. The former reasoning hinders the failure dimension understanding and, consequently, the failure causing discovery. The tester's primary attention could be the attribute list of the returned or defined tuple set rather than the attribute value list. However, what is commonly done is to check only the attribute value list.

Fault types related to the same structural item do not share the results of fault propagation analysis. These fault types can produce disjoint failure dimensions. It is the case for item [s1], where fault type [s1]-3 produces a particular failure dimension, contrasting its other fault types in this item. In item [s2], only one fault type, [s2]-3, is related to multiple failure dimensions. Nevertheless, all fault types in item [s6] have a well-known failure dimension. Although structural items do not capture the whole notion of failure dimension semantics, it was found that they organize the faults in groups to aid failure comprehension.

In the command analysis context, the fault propagation is controlled by the fault type, faulty command, and database state. For instance, consider fault type [s4]-4 in Table 3. Depending on the database state, the number of tuples or the attribute value list failure dimension, or both, will be manifested by the command execution. It complicates the tracking down of the relationship between a failure and its types of faults, since it augments the dubiousness of causing failure and hinders the use of a fault-revealing database. A typical database domain is infinite; it reinforces the necessity for systematic testing approaches.

The number of tuples and attribute value list dimensions are caused by a great diversity of fault types that encompass query and state changing commands. More than 50% of fault types were the reason for these failure dimensions. In this sense, testing strategies could give more attention to these dimensions to require fault revealing software elements.

Some of the fault types are associated to more than one failure dimension, such as [s4]-4, [d2]-3, [u3]-2. This fact is observed only for the number of tuples and the attribute value list dimensions, and the most fault type causing occurrences is in select command (9 of 13). It could be an evidence that select command has a great deal of real faults related to these failure dimensions, and they demand a lot of tester attention. Furthermore, almost all of the failure dimensions are exposed by more than one fault type. As a consequence, it is very unlikely to discover the causing failure at a first glance. Thus, there is a many-to-many mapping between faults and failures. This corroborates that the failure-to-fault way is non-trivial and the manipulation fault knowledge is crucial for SQL programming and testing of database applications.

8 Conclusions

This paper discusses issues arising in data manipulation failures and presents the results of an investigation aiming at understanding the relationship between faults and failures. The main motivation is the quality improvement of SQL-based applications, due to the extensive use of this language in database applications.

The structure of SQL manipulation commands was extracted from their basic constructions and organized in structural items. These items represent a way towards understanding and grouping faults in order to evaluate their propagation to the outside of the command. The concept of manipulation failure was examined and two data sets were introduced to capture the notion of command output, the returned and defined tuple set, related to query commands (select) and state changing commands (insert, delete, and update), respectively. Failure dimensions were discussed along with query and state changing operations. A list of manipulation fault types was presented by SQL command examples in a self-

explanatory fashion. A database was built and a set of faulty SQL commands has been used to build a mapping between fault types and failure dimensions.

The mapping data were discussed in the manipulation command context. The lessons learned with building the mapping and the results of data mapping analysis were highlighted. The analysis results indicated: i) there is a many-to-many mapping between faults and failures; ii) failure dimensions are dependent on fault type, faulty command, and the database itself; and iii) manipulation fault knowledge is crucial for SQL programming and testing of database applications.

8.1 Contributions

Some contributions of this article are:

- A data manipulation fault list is introduced and identified based on structural items of manipulation commands. The faults are described and illustrated by examples. This is the first element of a comprehensive investigation towards systematic testing approach proposal.

- The notion of manipulation command execution output presented by two expected sets. The returned and defined sets, respectively related to query and state changing commands, capture sufficient data for test case design and oracle correctness checking.

- A list of failure dimensions is proposed based on content of the returned and defined expected sets. These dimensions represent a resource for a major issue in general testing techniques: the determination of whether or not the output produced, as the result of running a test case, is correct.

- A mapping between faults and failures is built and preliminarily validated by applying a set of faulty commands, aiming at increasing the knowledge of fault discovery and fixing.

This work can be extended by applying test cases with real faulty manipulation commands, aiming at validating the fault list with concrete data, extracting fault-prone structural items, evolving the fault-failure mapping, and building fault ranking models to guide programming and testing.

The results of this study are an insight into the faults and failures related to data manipulation in SQL-based applications, and suggest that fault knowledge is crucial to software quality.

References

[1] Beizer, B. *Software Testing Techniques*. John Wiley and Sons Inc., New York, NY, 1990.

[2] Chays, D., Dan, S., Frankl, P. G., Vokolos, F. I., and Weyuker, E. J. A Framework for Testing Database Applications. In *Proc. of. the Intl. Symposium on Software Testing and Analysis*, Portland, Oregon, 2000.

[3] Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., and Wong, M. Y. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

[4] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. In *Communications of the ACM*, volume 13, pages 377–387, 1970.

[5] Daou, B., Haraty, R. A., and Mansour, N. Regression Testing of Database Applications. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, Las Vegas, Nevada, 2001.

[6] Elmasri, R. and Navathe, S. *Fundamentals of Database Systems*. Addison Wesley, Boston, MA, 4^{th} edition, 2003.

[7] Fortier, P. J. *Implementing the SQL Foundation Standard*. McGraw-Hill, 1999.

[8] Kapfhammer, G. M. and Soffa, M. L. A Family of Test Adequacy Criteria for Database-Driven Applications. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2003*, Helsinki, Finland, September 2003.

[9] Kelly, D. and Shepard, T. A Case Study in the Use of Defect Classification in Inspections. In *Proceedings of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, November 2001.

[10] Leitao-Junior, P. S., Vilela, P. R. S., and Jino, M. Mapping Faults to Failures in SQL Manipulation Commands. In *Proceedings of the 3rd ACS IEEE International Conference on Computer Systems and Applications (AICCSA-05)*, Egypt, Cairo, January 2005.