COMPROBACIÓN DE ASERTOS EN BASES DE DATOS RELACIONALES

MIHÁLY PALENIK

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y PROGRAMACIÓN
Universidad Complutense de Madrid

TRABAJO FIN DE GRADO

February 1, 2017

Director:
Rafael Caballero Roldán

Contents

Abstract

In this work we consider the problem of detecting errors in large sets of SQL relations. In order to detect possible bugs the user can introduce assertions using a simple, set-like language indicating properties like inclusion or membership. Then, the system checks these assertions, reporting to the user if any assertion violation is detected. The assertions include options that allow the system to consider relations both as sets and as multisets and also to take the tuple order into account. These options can be included by the user at the same time the assertions are defined. We present a working prototype developing these ideas.

Keywords: relational databases, postgresql, assertions, debugging, relational algebra, Java, ANTLR, multisets, testing, SQL views

Resumen

En este trabajo consideramos el problema de la detección de errores en grandes conjuntos de relaciones SQL. Para facilitar la detección de errores, el usuario puede introducir asertos utilizando un lenguaje sencillo con notación conjuntista, que permite definir propiedades como la inclusión o pertenencia. El sistema comprueba estos asertos e informa al usuario si se detecta que algún aserto no se verifica. Los asertos incluyen opciones que permiten al sistema considerar las relaciones tanto como conjuntos como si se tratara de multiconjuntos, a la vez que se permite tener en cuenta el orden entre las tuplas. Estas opciones se especifican por el usuario a la vez que se definen los asertos. Presentamos también un prototipo que desarrolla estas ideas.

Palabras clave: bases de datos relacionales, postgresql, asertos, depuración, álgebra relacional, Java, ANTLR, multiconjuntos, pruebas, vistas SQL

Chapter 1

Introduction

Nowadays databases are huge, it is hard to understand the whole one especially in case of lot of views. Unfortunately developers sometimes make mistakes that cause wrong queries. This kind of problems are hard to detect, mainly when the original tables and views contain plenty of data. But what can developers do? How can errors are detected? It may happen, that developers realize the generated view is wrong, but the query which is generate this view is certainly good, then what is wrong? Views, which are part of the query, what generate the view? A lot of debugging and time are required to find the error source.

Is there some way to make debugging easier? In this work we propose to use assertions. Sometimes we do not know that two views are correct or not but know relation between them. For example, we consider two view: countries in Europe and countries in Eurasia. It is trivial that European countries are also Eurasian countries. Therefore, we can introduce an assertion representing this inclusion relation. If the assertion fails, then we find the symptom of a bug, and we can start to analyze views employed in the assertion.

Assertions can be checked in a dynamic way. For instance, it is possible that some views are incorrect but the particular instance of data do not show this mistake. However, if we modify the original tables by adding more data, it is possible that the error shows up. Thus assertions should not be considered a static property that can be checked just the first time, but properties that can be checked each time the database instance change.

As formal language supporting our assertions we consider relational algebra, representing assertions as first order formulas over set predicates such as inclusion and constants like the empty set. In particular we employ the extended relational algebra with its well-founded semantics to model the database assertions. The difference between relational algebra and extended relational algebra is that while in the former relations are considered sets, the extended version works with multisets and includes more operations for queries. Our assertion language is based on set operations of relational algebra.

In our setting, assertion can take two different forms. The first one involves two relations and one operator:

- <= : right inclusion
- < : right strict inclusion
- => : left inclusion
- > : left strict inclusion
- = : equality
- <> : inequality

Our previous example can be easily express with this form. Assume European countries and Eurasian countries has *eur_countries* and *eas_countries* names in database. Then assertion will be:

$$eur\_countries <= eas\_countries$$

Originally, we can think of this tables like sets, but it can be add *rep* argument to switch to the multiset interpretation. These arguments are included between square brackets in this form. Another one is *order* which means set or multiset has ordering. This is usually defined by $ORDER\ BY$ or storage of table.

The second form of assertions is element checking, with the form

$$(A_1, A_2, \ldots, A_n)\ in\ Relation$$

where $(A_1, A_2, \ldots, A_n)$ is a tuple representation of one row in a relation. Meaning of this is identical with element checking of set.

Let us see an example. Suppose that we have a *countries* table with some country entries, and two user views are created: *ord_pop_ms* which is middle sized countries (between 200000 and 700000 km^2) ordered by area and *ord_countries* which is countries table ordered by area. These are possible instances of these relations:

name	continent	area	population	is_monarchy
'Hungary'	'Europe'	93036	9893899	false
'Spain'	'Europe'	504782	46449565	true
'UK'	'Europe'	244820	65102385	true
'Italy'	'Europe'	301318	60233948	false
'Croatia'	'Europe'	56594	4284889	false
'Poland'	'Europe'	312679	38483957	false

'Estonia'	'Europe'	45226	1313271	false
'Finland'	'Europe'	338145	5501043	false
'Mexico'	'America'	1972550	120286655	false
'Venezuela'	'America'	916445	30405207	false
'Canada'	'America'	9984670	35702707	true
'Columbia'	'America'	1141748	47704427	false
'Brazil'	'America'	8547404	200361925	false
'Chile'	'America'	756950	17619708	false
'Uruguay'	'America'	176215	3407062	false
'Guatemala'	'America'	108890	15468203	false
'Japan'	'Asia'	377915	127110047	true
'Nepal'	'Asia'	147181	30986975	false
'Bhutan'	'Asia'	47000	753947	true
'India'	'Asia'	3287263	1263200000	false
'Mongolia'	'Asia'	1564116	2953190	false
'Ghana'	'Africa'	238540	25904598	false
'Zimbabwe'	'Africa'	390757	14149648	false
'Zambia'	'Africa'	752614	14309466	false

Table 1.1: *countries* table

name	continent	area	population	is_monarchy
Ghana	Africa	238540	25904598	f
UK	Europe	244820	65102385	t
Italy	Europe	301318	60233948	f
Poland	Europe	312679	38483957	f
Finland	Europe	338145	5501043	f
Japan	Asia	377915	127110047	t
Zimbabwe	Africa	390757	14149648	f
Spain	Europe	504782	46449565	t

Table 1.2: *ord_pop_ms* view

name	continent	area	population	is_monarchy
Estonia	Europe	45226	1313271	f
Bhutan	Asia	47000	753947	t
Croatia	Europe	56594	4284889	f
Hungary	Europe	93036	9893899	f
Guatemala	America	108890	15468203	f
Nepal	Asia	147181	30986975	f
Uruguay	America	176215	3407062	f
Ghana	Africa	238540	25904598	f
UK	Europe	244820	65102385	t
Italy	Europe	301318	60233948	f

Poland	Europe	312679	38483957	f
Finland	Europe	338145	5501043	f
Japan	Asia	377915	127110047	t
Zimbabwe	Africa	390757	14149648	f
Spain	Europe	504782	46449565	t
Zambia	Africa	752614	14309466	f
Chile	America	756950	17619708	f
Venezuela	America	916445	30405207	f
Columbia	America	1141748	47704427	f
Mongolia	Asia	1564116	2953190	f
Mexiko	America	1972550	120286655	f
India	Asia	3287263	1263200000	f
Brasil	America	8547404	200361925	f
Canada	America	9984670	35702707	t

Table 1.3: *ord_countries* view

Now let us define our assertion.

$$ord\_pop\_ms <= ord\_countries\,[ord]$$

It is easy to check that this assertion is true. But what if we introduce *countries* table in place of *ord_countries*? The assertion will fail because of *ord* argument. This is really an inclusion but it has different ordering. You can see in *coutries* table the entry of UK is earlier appear than entry of Ghana, which is the first row of *ord_pop_ms*. But without *ord* argument the assertion is true.

Chapter 2

Relational algebra

The theoretical bases of assertion is extended relational algebra. In this work almost all assertions are expressed in this setting because it is easy to make connection between relational algebra and SQL queries. Before we define the exact meaning of each assertion, let us see relational algebra. As I mentioned it is an algebra with well-founded semantics to be modeling database's query. All relation is set of tuples. Relational algebra defines operations on this sets.

2.0.1 Operations

Set operations Because relation is set we can use set operation which is:

- $\cup$: union
- $\cap$: intersection
- $\setminus$: difference

Projection Assume we have a relation scheme $R(A_1, \ldots, A_N)$ and we want to see just specific column. In this case we can use projection (π). The result relation has same number of row like the original one but just with the specific column. For example if we just want to see 2nd and 5th column we can write $\pi_{A_2,A_5}(R)$.

Selection Selection (σ_{cond}) is an operation of selection based on a condition. In the condition logical operation can be used. For example consider relation with $Countries(name, continent)$ scheme and collect European countries. The result expression is $\sigma_{continent='Europe'}(Countries)$.

Rename When we use some of the above operations then it is possible to add name for the created relation with rename (ρ_{name}). Or we can modify whole scheme of relation also with form of $\rho_{name(A_1,\ldots,A_N)}$.

2.0.2 Joins

Cartesian product This ($\times$) is also a set operation between two sets. First, take the columns of the first relation and after take the second one. Column number of result relation is the sum of the columns in the two relations. For example we have $R_1(A_1, \dots, A_N)$ and $R_2(B_1, \dots, B_N)$ schemes then $R_1 \times R_2$ will be $(A_1, \dots, A_N, B_1, \dots, B_N)$. Columns name have an additional prefix – relation name – with dot, if there are some collision between columns name.

Natural join It ($\bowtie$) can be useful when two relation schemes have one or more columns with the same name. In that case the generated relation has all columns from original relations, but columns with the same name appear just once. Rows, where values of these columns are equal, are merged into one row. Assume $R_1(A_1, A_2, B_1)$ and $R_2(B_1, B_2)$ relation schemes, then $R_1 \bowtie R_2$ will has (A_1, A_2, B_1, B_2) scheme, and rows are the merge of two rows where data from B_1 is equal in the R_1 and R_2.

Theta join Theta join ($\bowtie_\theta$) is similar like Cartesian product but it has a selection which is defined in θ part. If R_1 and R_2 a relation then theta join is $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$.

2.0.3 Extended relational algebra

Extended relational algebra is exactly the same like relational algebra except it works with multisets. It is like set but it allows to have repetition. The above operators and joins work in a similar way as I am defined but it has additional operation.

2.0.4 Additional operation in extended relational algebra

Grouping Grouping ($\gamma_{columns}$) can be useful when we have multiple row which are equal. In *columns* part you can enumerate columns, and grouping is based on these ones.

Ordering You can define ordering with $\tau_{columns}$. It is based on columns which is enumerated in *columns* part.

Distinct With distinct operator (δ) multisets can be mapped to set. It just drops the duplicates. This is the relation between relational algebra and extended relational algebra.

Aggregate functions Aggregate function can be added in a *condition* or *columns* part. It is used with grouping. There are five ones and all has one parameter which is a column name.

- *AVG* : average of the column's value

- SUM : sum of the column's value
- MIN : minimum value of the column
- MAX : maximum value of the column
- $COUNT$: counts values of the column

Chapter 3

Assertion language grammar

In the introduction I added an informal definition of assertion language grammar. I use BNF form which is a notation technique for context-free grammar, which is usually used to define syntax for programming languages.

For implementation I use ANTLRv4 as you will see in implementation details. It can be easily defined your language syntax with that tool, and generate source code to handle the process of syntax tree building.

It can be seen below the BNF form of that grammar.

```
⟨assertion⟩   ::= ⟨relation⟩ ⟨operator⟩ ⟨relation⟩ '[' ⟨arguments⟩ ']'
               |  '(' ⟨row⟩ ')' 'in' ⟨relation⟩


⟨arguments⟩   ::= ⟨arguments⟩ ',' ⟨arguments⟩
               |  'order'
               |  'rep'


⟨row⟩         ::= ⟨row⟩ ',' ⟨row⟩
               |  ⟨bool⟩
               |  ⟨string⟩
               |  ⟨int⟩


⟨bool⟩        ::= 'true'
               |  'false'
```

```
⟨string⟩      ::= ”’ ([a-z]|[A-Z]|[0-9])* ”’

⟨int⟩         ::= [0-9]+

⟨relation⟩    ::= ([a-z]|[A-Z]|’_’)[’.’|’_’|’$’|0-9|a-z|A-Z]*

⟨operator⟩    ::= ’=’
                |   ’<>’
                |   ’<=’
                |   ’=>’
                |   ’<’
                |   ’>’
```

The structure of assertion grammar is easy now. But it can be extend. The first rule *assertion* contains all possible form of assertions, which are now two. But if it is necessary to extend with more operator, it just have to be added to *OPERATOR* rule.

Chapter 4

Formal definition of assertions

For a better implementation it is important to define well the meaning of assertions. It is not enough informal explanation because that causes a lot of misunderstanding. In the next subsections you get to know all formal definition of assertion which is in relational algebra form.

4.0.1 Operators

In this section all assertion without arguments ($order, rep$) are defined with extended relational algebra. This algebra is well defined and it is possible to check SQL query – implementation – and assertion equivalency. Extended relational algebra works with multisets, that means database's relations have duplicates. First inclusion is considered. It is equivalent with inclusion of set. Because of this we must eliminate duplicates, and have to define inclusion with relational algebra.

$$A \subseteq B \equiv A \setminus B = \emptyset$$

This can be expressed in relational algebra.

Definition (Inclusion)**.**

$$R_1 <= R_2 \overset{\text{def}}{=} \delta(R_1) \setminus \delta(R_2) = \emptyset$$

In the left side the inclusion operator (<=) is seen, and in the right side it is the definition of inclusion which is given by extended relational algebra. δ is necessary because of conversion multiset to set. All rest of assertion is based on inclusion assertion. Consider the equality, in mathematical sense is expressed in two way of inclusion and this is exactly the equality assertion.

Definition (Equality)**.**

$$R_1 = R_2 \overset{\text{def}}{=} R_1 < = R_2 \wedge R_2 <= R_1 =$$
$$\delta(R_1) \subseteq \delta(R_2) \wedge \delta(R_1) \supseteq \delta(R_2)$$

Inequality checking is opposite meaning of equality. In the right-hand side the equality is an assertion which easily can be traced back to mathematical definition.

Definition (Inequality)**.**

$$R_1 <> R_2 \overset{\text{def}}{=} \neg(R_1 = R_2)$$

Strict inclusion is similar to inclusion but the two set must not be equal.

Definition (Strict inclusion)**.**

$$R_1 < R_2 \overset{\text{def}}{=} R_1 <> R_2 \wedge R_1 <= R_2$$

The *rep* argument

In grammar's argument part it can be chosen *rep* argument which is abbreviation of repetition. In general the meaning is the repetition takes into consideration. This part is where nature of multiset are used. Multiset inclusion – signed with $\subseteq_m$ – is also expressed with multiset difference, and connection with set functions have been already defined.

$$A \subseteq_m B \equiv A \setminus_m B = \emptyset$$

In below definition of inclusion the right-hand side is an extended relational algebra expression, not an assertion.

Definition (Assertions with *rep* argument)**.**

$$\begin{aligned} R_1 <= R_2[rep] &\overset{\text{def}}{=} R_1 \setminus R_2 = \emptyset \\ R_1 = R_2[rep] &\overset{\text{def}}{=} R_1 <= R2[rep] \wedge R_2 <= R_1[rep] \\ R_1 <> R_2[rep] &\overset{\text{def}}{=} \neg(R_1 = R_2[rep]) \\ R_1 < R_2[rep] &\overset{\text{def}}{=} R_1 <> R_2[rep] \wedge R_1 <= R_2[rep] \end{aligned}$$

The *order* argument

Next argument is *order*. Unfortunately extended relational algebra does not have ordering ([Wid09]). That is why added an extra column - called $\#O$ - to all relation which is number of sequence from one to the number of rows. These numbers simulate row ordering which is equal with the row number in a real database. Thus original $R(A_1, \ldots, A_N)$ relation is considered $R(\#O, A1, \ldots, A_N)$. Because of lack of *rep* argument, it should be avoid the duplicates from the relation and keep ordering.

δ_o **function** To reach the required relation we have to use δ_o function which keeps just the first appearance of element.

Definition.

$$\delta_o(R(A_1, \ldots, A_n)) = \gamma_{COUNT(\#lMin) \rightarrow \#O, A_1, \ldots, A_N}(Prec)$$

where $Prec$ is

$$Prec(\#lMin, \#gMin, A_1, \ldots, A_N) :=$$
$$\rho_{FirstsNum}(\pi_{\#min}(Firsts)) \bowtie_{Firsts.\#min \geq FirstsNum.\#min} Firsts$$

where $Firsts$ is

$$Firsts(\#min, A_1, \ldots, A_N) = \gamma_{MIN(\#O), A_1, \ldots, A_n}(R).$$

$Firsts$ is the table which contains all first appearance of different rows with original row number, and $Prec$ contains all data from $Firsts$ and it is paired with the row number of previous elements and itself. You can see an example below which give you a better understanding. Consider this relation:

#O	A_1	A_2	A_3
1	A	A	B
2	B	A	B
3	A	A	B
4	B	A	B
5	C	C	B
6	C	C	B
7	A	A	B
8	D	D	D
9	C	C	B

$Firsts$ is:

#min	A_1	A_2	A_3
1	A	A	B
2	B	A	B
5	C	C	B
8	D	D	D

The left table of $Prec$ – $FirstsNum$ – is:

#min
1
2
5
8

And after theta join with $Firsts$ table the $Prec$ is:

#lMin	#gMin	A_1	A_2	A_3
1	1	A	A	B
1	2	B	A	B
2	2	B	A	B
1	5	C	C	B
2	5	C	C	B
5	5	C	C	B
1	8	D	D	D
2	8	D	D	D
5	8	D	D	D
8	8	D	D	D

And finally the δ_o:

#O	A_1	A_2	A_3
1	A	A	B
2	B	A	B
3	C	C	B
4	D	D	D

Now assertions can be defined. Be careful because now R_1 and R_2 is a relation with additional $\#O$ column. That is why projection used in assertion definition.

Definition (Assertions with *order* argument)**.**

$$\begin{aligned} R_1 <= R_2[order] &\overset{\text{def}}{=} \pi_{A_1,\ldots,A_N}(R_1) <= \pi_{A_1,\ldots,A_N}(R_2) \land Cond \\ R_1 = R_2[order] &\overset{\text{def}}{=} R_1 <= R2[order] \land R_2 <= R_1[order] \\ R_1 <> R_2[order] &\overset{\text{def}}{=} \neg(R_1 = R_2[order]) \\ R_1 < R_2[order] &\overset{\text{def}}{=} R_1 <> R_2[order] \land R_1 <= R_2[order] \end{aligned}$$

where $Cond$ is:

$$\sigma_{R_1.\#O > R_2.\#O}(\delta_o(R_1) \bowtie \delta_o(R_2)) = \emptyset$$

order **and** ***rep*** **at the same time**

That is the hardest part of this section. First, I tried to express these options also in extended relational algebra. After trying in many different ways, finally I realized that the problem was the expressivity of relational algebra. In this formal language, you can only express things in a global – I mean table – scope. But these options, introduce locality which cannot be expressed with relational algebra. This is not surprising because relational algebra is not Turing complete. I have not proved formally that expressing these operations in relations algebra is not possible. Thus, this remains an open question. However, in this work I decided to give up this way. I think that even if the relational algebra expression for these operations exist, they would be too complex and inefficient.

Definition (Inclusion assertion with *order* and *rep* arguments)**.** Consider two relations:

$$R_1(\#O, A_1, \ldots, A_n), R_2(\#O, A_1, \ldots, A_N).$$

Then $R_1 <= R_2[rep, order]$ assertion is true if

$$\{(1, a_{1_1}, \ldots, a_{1_n}), \ldots, (Y, a_{Y_1}, \ldots, a_{Y_n})\} \in R_1$$

and

$$\{(1, b_{1_1}, \ldots, a_{1_n}), \ldots, (Z, b_{Z_1}, \ldots, a_{Z_n})\} \in R_2$$

where

$$Y = |R_1| \wedge Z = |R_2| \wedge Y \leq Z.$$

Then

$$\forall(i, a_{i_1}, \ldots, a_{i_n})(i \in [2..n-1]) : \exists(j, b_{j_1}, \ldots, b_{j_n}) : j_{prev} < j < j_{next}$$

where j_{prev} is the value which belongs to $i-1$ and j_{next} is the value which belongs to $i+1$. In the border is same except in case of $i = 1$ the condition is $j < j_{next}$, and in case of $i = n$ is $j_{prev} < j$. The one element inclusion is trivial.

The definition may be a little hard. But think about regular expressions. If $a_1, \ldots, a_n$ are the rows of R_1 then the inclusion operator holds if $*, a_1, *, a_2, \ldots, *, a_n, *$ are the rows in R_2.

Now, it is easy to define the rest of the assertion.

Definition.

$$\begin{aligned}
R_1 = R_2[rep, order] &\overset{\text{def}}{=} R_1 <= R2[rep, order] \wedge R_2 <= R_1[rep, order] \\
R_1 <> R_2[rep, order] &\overset{\text{def}}{=} \neg(R_1 = R_2[rep, order]) \\
R_1 < R_2[rep, order] &\overset{\text{def}}{=} R_1 <> R_2[rep, order] \wedge R_1 <= R_2[rep, order]
\end{aligned}$$

4.0.2 Element checking

The *membership* assertion follows a different syntax. It is defined by a tuple, which represents one row of relation, and the relation where the tuple should be.

Definition. Assume we have a relation with scheme $R(A_1, \ldots, A_m)$. Then

$$(a_1, \ldots, a_n) \; in \; R$$

assertion is true if

$$n = m \wedge \exists(b_1, \ldots, b_m) \in R : \forall i \in [1..n] : a_i = b_i$$

Chapter 5

Implementation details

In implementation part of assertion a lot of tools are used. The implementation language is Java. I did not need to create a separate program for assertions, because this is a part of a bigger project. Therefore, I started from a framework (called SBuggy) where I could add my implementation.

SBuggy is a debugging tool for database views. With that program, the user can mark which views are incorrect, and SBuggy finds the related views or tables which occur in the view query. If the tool finds a bug in the selected view query, then the debugger finishes. However, if the query is correct, the debugger continues debugging the previously marked views or tables. In this project we add assertions as a new feature that improves that program ability to help developer during debugging.

The tool ANTLRv4 is used for the grammar definition and translation into Java. It is easy to write language syntax with that tool. The user can write it in BNF form. ANTLR generates code which builds up a syntax tree. The programmers can add their own part for this generated code e.g. it is possible to add a particular error handling method or include additional information during the tree traversal phase.

SBuggy already has graphical user interface, so I have a base to put my user interface. It is created with *Swing*, which is a graphical user interface widget toolkit for Java.

The assertions are translated into SQL queries. These queries are executed from Java by using the JDBC interface. JDBC is an API that allows Java to connect to external databases, and run SQL queries. The database connection was already implemented in SBuggy, so I just had to use that connection.

The relational database management system employed in this project is *postgreSQL*, which is a free software. I did not use any special syntax in the queries, therefore, it is not important what dialect of SQL is used. The advantage of this is that our tool could easily work with other database systems such as MySQL or Oracle.

5.0.1 Assertions interpreted in SQL query

In this subsection I show the equivalent SQL query for each assertion. This is straightforward since there is a well-known connection between relational algebra and standard SQL.

Inclusion

Because every assertion is expressed with inclusion assertion, that is why I just have to create equivalency in that part. We assume that the mentioned relations are stored in a database as a table or view.

Without arguments First, consider inclusion assertion without argument. SQL has set operation like union, intersection or difference. That is why is easy to express that part.

R1 <= R2

```
SELECT * FROM R1
  EXCEPT
SELECT * FROM R2
```

This SQL query is exactly the same like difference in case of sets. If the result is an empty relation then inclusion assertion is true otherwise is false. You can easily see I tried to approximate to the mathematical definition.

***rep* argument** Here we have to take into account repetition. Fortunately, relational database tables already resemble multisets. And the set operations in SQL have syntax for multiset treatment, just adding the *ALL* keyword.

R1 <= R2 [rep]

```
SELECT * FROM R1
  EXCEPT ALL
SELECT * FROM R2
```

After this, the tool needs to check the result, like in the previous SQL query. If the result is empty then assertion is true, and otherwise it is false.

***order* argument** Checking assertions including the *order* option in SQL is more complex. Thus, I split the code in small parts, like in the relational algebra definition. If originally the table or view does not include an ordering section, then ordering is the storage of rows in database. Unfortunately, if *GROUP BY* is used in query then it is not guaranteed that it keeps ordering. Because of this, I use the trick like in relational algebra to add a new column, which contains the number from one to the last row number, and consider this the ordering. Consider a table or view $R1(A1, \ldots, AN)$. Then, we can add that plus column with the below SQL query:

Add ordering to R1

```
SELECT row_number() over(), [R1 columns]
FROM R1
```

Result is $R1$, with the additional *row_number* columns. Square brackets are used to marks the part where all columns name are enumerated. This table can be used to define SQL query for $\delta_o(R1)$ relation.

Query to create $\delta_o(R1)$

```
SELECT row_number() over(order by MIN(row_number)),
        [R1 columns]
FROM [relation with additional row_number] AS tabley
GROUP BY [R1 columns];
```

This query does exactly what we define in relational algebra. With postgreSQL we can define not just a new columns with number but we can define how we want order. This syntax is same with ORDER BY syntax. It can be seen I added alias for the result, it is necessary because of syntax reason. For simplicity above query gets a name, which is *reducedRelation(R1)*, and has one parameter, which is the table or view name, and I will use it in my SQL query. Think about that like a macro.

R1 <= R2 [order]

```
SELECT [lefty.A1, ..., lefty.AN]
FROM reducedRelation(R1) AS lefty,
     reducedRelation(R2) AS righty
WHERE [lefty.A1=righty.A1 AND ... AND lefty.AN=righty.AN]
       AND lefty.row_number>righty.row_number
```

Above query is the SQL query of inclusion assertion with *order*. In the last row it does exactly what is the selection part of the definition in relational algebra. Below you can see the whole query.

R1 <= R2 [order]

```
SELECT [lefty.A1,..., lefty.AN] FROM
  (SELECT row_number() over(order by MIN(row_number)),
          [R1 columns]
  FROM
    (SELECT row_number() over (), * FROM R1) AS tabley
    GROUP BY [R1 columns])
  ) AS lefty ,
    (SELECT row_number() over(order by MIN(row_number)),
            [R2 columns]
    FROM
      (SELECT row_number() over (), * FROM R2) AS tabley
        GROUP BY [R2 columns]))
    AS righty
WHERE [lefty.A1=righty.A1 AND ... AND lefty.AN=righty.AN]
      AND lefty.row_number>righty.row_number
```

But before run that query like in the definition, we have to check first the original inclusion without arguments $R1 <= R2$. If it fails it is not important to check the second query.

***order* and *rep* argument** Unfortunately I could not add definition in relational algebra in that case. So implementation is based on regular expression definition.

R1 <= R2 [order, rep]

```
while (R2.next() && !R1.isAfterLast()) {
  if (isEqual(R1, R2)) {
    R1.next();
  }
}
```

Here R1 and R2 are two *ResultSet* which contain the original *R1* and *R2* table. *isEqual* function is responsible for two rows equality checking.

Other assertions

As other assertions with operator $(<, =, <>)$ are defined in relational algebra, as it is implemented in Java. All assertion, which are not inclusion, first they use inclusion and after other checking. For example in equality checking, first see right inclusion and after left inclusion. This is similar in inequality, because if equality is true then inequality is false, and as we know equality use inclusion.

Element checking This was an easy part to create SQL query from assertion. In a tuple, which represent one row in a table, it can be written three type of value, which are integer, boolean value and string between apostrophe. This is just a trivial checking, which uses all value in WHERE clause.

5.0.2 Test

Program has an SQL script – called *test_db.sql* – which contains SQL statements to create test database. It can be run with **psql -f [filename]** command. This script assumes, we have already a *test* schema in postgreSQL.

Script has create statements of one table and 13 view. The table name is *countries* and has countries with its data, like area or population. It has 24 rows.

Create test.countries table

```
CREATE TABLE test.countries (
    name VARCHAR(20),
    continent VARCHAR(20),
    area INT,
    population INT,
    is_monarchy BOOLEAN
);
```

I try to create test database, where all test have a little sense, not just some random value. This goal sometimes is achieved sometimes not, because of complexity of assertions. Below you can see a little explanation for all view with name:

- *eu_and_mon*: European monarchy
- *middle_sized*: Countries with middle sized area [200000 - 700000]
- *density*: Population density
- *dense_continents*: Countries with density>100
- *continents*: All continents
- *ord_middle_sized*: *middle_sized* ordered by area
- *ord_countries*: Countries ordered by area
- *ord_pop_ms*: *middle_sized* ordered by pop
- *u_ms_mon*: Unio of *middle_sized* and all monarchy
- *twice*: Countries Descartes
- *countries_pop*: Order by population

- *union_pop*: Twice *middle_sized* minus *eu_and_mon* ordered by population
- *union_3_pop*: Three times *middle_sized* ordered by population

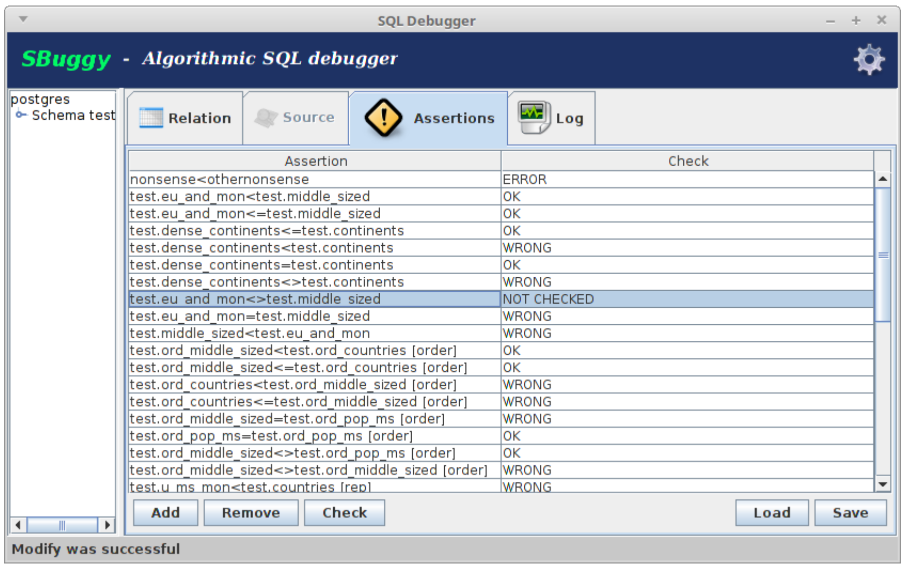
Above views and table are the test database. In **[project_folder]/asserts** folder you can see two files. **test.txt** contains all assertion, which are based on the test database. For example *test.eu_and_mon* < *test.middle_sized*, which means European monarchies, is included by middle sized countries in the database (between 200000 and 700000 km^2).

Expected output is in the same directory in the **out.txt** file. The above assertion is passed so in this file in the same row like assertion has an OK. Here are other example from this file *nonsense* < *othernonsense*. Because test database does not have *nonsense* and *othernonsense*, table SBuggy got an exception from SQL connection. This appear in out file with ERROR.

Chapter 6

User documentation

The above detailed implementation is part of the SBuggy software. This program has a GUI so I had a skeleton to insert my part. It appears in a different tab.



Assertion	Check
nonsense<othernonsense	ERROR
test.eu_and_mon<test.middle_sized	OK
test.eu_and_mon<=test.middle_sized	OK
test.dense_continents<=test.continents	OK
test.dense_continents<test.continents	WRONG
test.dense_continents=test.continents	OK
test.dense_continents<>test.continents	WRONG
test.eu_and_mon<>test.middle_sized	NOT CHECKED
test.eu_and_mon=test.middle_sized	WRONG
test.middle_sized<test.eu_and_mon	WRONG
test.ord_middle_sized<test.ord_countries [order]	OK
test.ord_middle_sized<=test.ord_countries [order]	OK
test.ord_countries<test.ord_middle_sized [order]	WRONG
test.ord_countries<=test.ord_middle_sized [order]	WRONG
test.ord_middle_sized=test.ord_pop_ms [order]	WRONG
test.ord_pop_ms=test.ord_pop_ms [order]	OK
test.ord_middle_sized<>test.ord_pop_ms [order]	OK
test.ord_middle_sized<>test.ord_middle_sized [order]	WRONG
test.u_ms_mon<test.countries [rep]	WRONG

Modify was successful

Figure 6.1: SBuggy

On the above picture you can see the *Assertions* tab. In the bottom corner there are three button *Add*, *Remove* and *Check*. With *Add* you can add one assertion to the table, which is in the middle of the tab. Easily can remove selected lines with *Remove* button. Multiselection also works. If you click on the *Check* button, then SBuggy start to check all assertion. If an assertion is already checked it is not problem, because program check that again. You can modify an existing assertion with double click on the row.

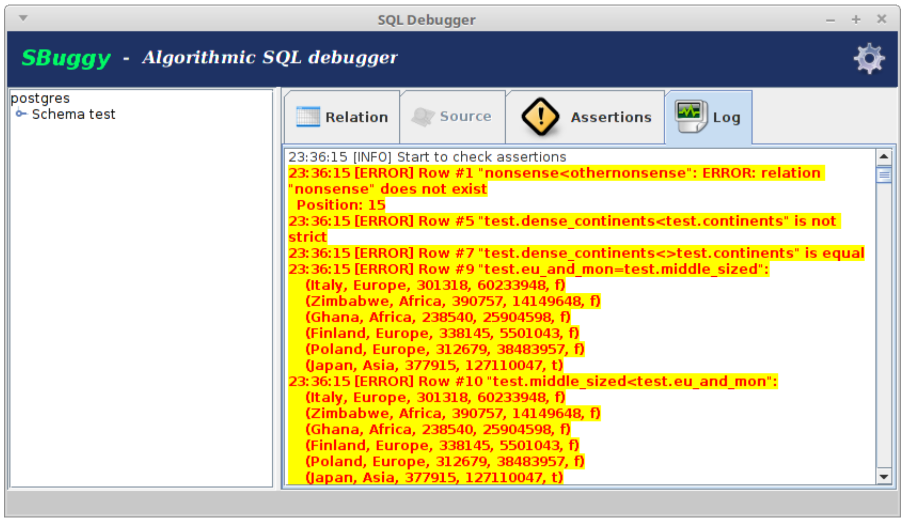
The bottom right corner contains the *Load* and *Save* buttons. The assertions can be saved and loaded, using a simple text file as output format. When we load assertions or add them directly through program, the syntax checker runs, displaying an error message if a syntax error is found.

All error appear in the Log tab. The type of error are:

- lexical or syntactical
- failing assertion
- database errors

In the middle table you can see assertions in the first column, and the state of assertion checking in the second column. The states are:

- NOT_CHECKED: assertion does not checked
- WRONG: assertion fails
- OK: assertion succeed
- ERROR: sign there is an error meanwhile assertion checking. Mainly errors with databases.



```
23:36:15 [INFO] Start to check assertions
23:36:15 [ERROR] Row #1 "nonsense<othernonsense": ERROR: relation
"nonsense" does not exist
 Position: 15
23:36:15 [ERROR] Row #5 "test.dense_continents<test.continents" is not
strict
23:36:15 [ERROR] Row #7 "test.dense_continents<>test.continents" is equal
23:36:15 [ERROR] Row #9 "test.eu_and_mon=test.middle_sized":
   (Italy, Europe, 301318, 60233948, f)
   (Zimbabwe, Africa, 390757, 14149648, f)
   (Ghana, Africa, 238540, 25904598, f)
   (Finland, Europe, 338145, 5501043, f)
   (Poland, Europe, 312679, 38483957, f)
   (Japan, Asia, 377915, 127110047, t)
23:36:15 [ERROR] Row #10 "test.middle_sized<test.eu_and_mon":
   (Italy, Europe, 301318, 60233948, f)
   (Zimbabwe, Africa, 390757, 14149648, f)
   (Ghana, Africa, 238540, 25904598, f)
   (Finland, Europe, 338145, 5501043, f)
   (Poland, Europe, 312679, 38483957, f)
   (Japan, Asia, 377915, 127110047, t)
```

Figure 6.2: Assertion errors in logger

6.0.1 Config file

SBuggy has a config file (*sbuggy.conf*) where specific configuration are stored like database's url, user name or logger info level. For assertions there are two config key which is used: *defaultFolder* and *assertionError*.

defaultFolder has a value of path. It is used, when user wants to load a file, which contains assertions. If this open dialog run first time, then initial path is the value from key *defaultFolder*. If user use load operation after the first time, then initial path is the last path from where user load assertions.

When assertion is correct syntactically and database connection and everything is good (we do not get exception) but assertion fails, we want to know why? In this case *assertionError* config key is important. It has a value of **ALL** or **ANY**. If user just wants to know one example, why an assertion is failed, **ANY** is used. If user wants to know all rows which fail the assertion, **ALL** value is used.

Other config options

To use SBuggy comfortably, it has a lot of different config options in config file. Mainly these are related to the database connection.

Database In *url* key you can set postgreSQL's url, where you can reach the database. It can be easily turn on SSL authentication with *ssl* key. And last one *database*, which is also related to database connection, it contains the database name, which you want to reach.

Logger Other config options is related to logger. Easily can be set up logger *infolevel*. With *logconf*, which contains a path, you can add, where do you want to save your logging.

Chapter 7

Working process

In the beginning of work the first thing was get to know SBuggy, because this is the framework of assertion implementation. The second thing was to find out the Assertion grammar, and mainly the question was what kind of assertion do we need, and make sense in the industrial area. Parallel with this, start to design user interface, and get to know Swing tool.

First implementation was to create user interface, so I had a skeleton, and I knew what kind of function do I have to implement. This continues easier function implementation, like load and save, adding assertion. As assertion syntax outlined, I started to create grammar in ANTLR, and analyzed generated output.

The hardest part was to provide a well-formed definition of assertions. In some cases it was tricky to find a suitable definition in relational algebra. But after that, the implementation was easier, because of equivalency between SQL query and relational algebra. Finally I reviewed the code, re-factor and add precise description of function in comment to facilitate the later work.

Chapter 8

Conclusions and Future Work

This project was a great implementation practice in Java and a very useful brain training. I was getting into a lot of different tool. By the way I created an assertion language with my teacher not just theoretically way but practical way.

I tried to do my best in this project. We had a lot of goal, and I achieved a lot from them. But this program have to extend more. Firstly the grammar. There are a lot of idea, which assertion structure will be useful. It is not easy to find out. It has to discuss with a lot of professional, mainly who works in industrial area. That is why SBuggy is different like other project. This is not just a tool to play in university, but it is used in production environment. After discussion, it will be easy to extend grammar, because I kept in mind to achieve this. Now we already have an idea for new assertion, which is to check how many rows are contained in a table. You will be add range or exact number.

Most important part now to finish, to create real connection between SBuggy and assertion checking. Now just it connects with database, adds assertion after check, and that is all. It is already useful but not enough. The real goal is to mark views, which is wrong with SBuggy, and you can trace back the source of error.

For do this connection, it has to be improve graphical user interface. With some dialog, you will get some question about errors, which tables have wrong rows, which tables cause the error It is also a good thing to write errors in logger, but this is just for logging, it is not so user-friendly. Somehow has to be find out, which is the best way to show errors. Moreover now to add assertion through the program is really dull. A small editor – maybe special for assertion – would be the best way to improve that part.

Other part, which also connected to the user interface, is how to SBuggy checks the assertions. Now if user click to check button, all assertion will be evaluated, no matter it is already succeed or not. The first step is to modify checking behaviour to check just failed assertions. Moreover it should be create check all button, and separate check which is evaluate just the selected assertions in table. Maybe it would be good to add a force checking option, which do the same checking like now.

The current class hierarchy seems suitable in our setting, that includes a small number of assertions. However, when it will be more, it would be a great idea to re-factor AssertionChecker class before it will be a god class. Maybe this one just should be responsible for handle checking, and other classes should be create the SQL queries for assertions, and another classes which handles the database connection.

At that moment, the SQL query representations of assertions are created during the assertion checking phase, and SBuggy do not store them. It is a real drawback because every time, when user starts assertion checking, program recreates SQL queries, which is unnecessary. An easy improvement for future version, which can really accelerate the program, is to create SQL statements directly after syntax checking, and store the code together with assertions. In this way the transformation from assertions into SQL, will be performed just once, and if the assertion is changed by the user, then it has to be recreated SQL query.

Except to find out new assertions, all thing that I mentioned, only suppose small changes, although the final result is spectacular. I hope I can do these changes myself, and improve SBuggy further not just in my foreign scholarship.

Chapter 9

Bibliography

[CGS12] Rafael Caballero, Yolanda Garcia-Ruiz, and Fernando Sáenz-Pérez. Algorithmic debugging of sql views. In *Proceedings of the 8th international conference on perspectives of system informatics*. Volume 7162. In PSI'11. Springer-Verlag, Novosibirsk, Russia, 2012, pages 77–85. ISBN: 978-3-642-29708-3. DOI: `10.1007/978-3-642-29709-0_9`.

[Ora17] Oracle. Programming With Assertions. `http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`. Retrieved: 8 January 2017.

[Wid09] Jeffry D. Ullman - Jeniffer Widom. *Adatbázisrendszerek - alapvetés*. Panem Kft., 2009.