Department of Informatics, University of Zürich

MSc Thesis

Design and Implementation of a Statistics and Visualization Engine for the Oshiya Degugger and Analyzer

Patrick Leibundgut

Matrikelnummer: 05-913-041

Email: patrickleibundgut@uzh.ch

July 3, 2012

supervised by Prof. Dr. Michael H. Böhlen and Christian Tilgner

University of Zurich[UZH]

Department of Informatics

This thesis is dedicated to my parents who have given me the opportunity of an education from the best institutions and support throughout my life.

Acknowledgements

First of all I would like to thank Professor Michael H. Böhlen for giving me the opportunity to write my Master Thesis and for letting me participate in the research of his group.
It gives me great pleasure in acknowledging the supervision of my advisor, Christian Tilgner, who has been abundantly helpful and has assisted me in numerous ways. The discussions I had with him were invaluable.

Abstract

This thesis is about extending the Oshiya Debugger and Analyzer (ODA) with statistics and visualizations. ODA is an application which allows comparing and analyzing scheduling protocols. The implemented statistics collect statistical measures that can be displayed with charts and tables. To extend the statistical measures, the user can define his own statistical measures by setting up statistic queries. In addition to the statistics, ODA has been extended to support breakpoints. With the concept of break and analyze queries a user can set up breakpoints and further analyze the results.

Zusammenfassung

Diese Arbeit beschreibt die Erweiterung des Oshiya Debugger und Analyzer (ODA) mit Statistiken und Visualisierungen. ODA ist eine Anwendung, mit welcher man Scheduling Protokolle vergleichen und analysieren kann. Die hinzugefügte Statistik sammelt statistische Daten, die man mit Graphen und Tabellen veranschaulichen kann. Um die statistischen Daten zu erweitern, kann der Benutzer seine eigenen Abfragen mittels Statistic Queries erstellen. Zudem wurden Breakpoints hinzugefügt. Durch das Aufsetzen von Break und Analyze Queries kann ein Benutzer die Applikation anhalten und die Resultate analysieren.

Contents

1 Introduction **9**
- 1.1 Tasks . . . 11
  - 1.1.1 Conceptual design for Statistics inside ODA . . . 11
  - 1.1.2 Implementation of a Statistics and Visualization Engine . . . 11
  - 1.1.3 Writing a Scientific Demonstration Paper . . . 12
- 1.2 Contributions . . . 12

2 Sample Scenario **14**

3 Break and Analyze Queries **16**
- 3.1 Break Query . . . 16
- 3.2 Analyze Queries . . . 17
- 3.3 Challenges with the Break and Analyze Queries . . . 18

4 Global Statistics **19**
- 4.1 Challenges with the Global Statistics . . . 20

5 Statistic Queries **21**
- 5.1 Challenges with the Statistic Queries . . . 22

6 Visualization **23**
- 6.1 Statistics . . . 25
  - 6.1.1 Line Charts . . . 25
  - 6.1.2 Bar Charts . . . 26
  - 6.1.3 Pie Charts . . . 26
  - 6.1.4 Table . . . 27
- 6.2 Progress Bar . . . 27
- 6.3 Schedule . . . 28
- 6.4 Visualization of the Scheduling Algorithm . . . 29
- 6.5 Query Browser . . . 29
- 6.6 Challenges with the Visualization . . . 30

7 Implementation **31**
- 7.1 Documentation of Code . . . 31
- 7.2 Visualization . . . 33
- 7.3 Additional Features . . . 33

7.4 Testing 34
7.4.1 Testing Components 35
7.4.2 Testing Graphical User Interface 35
7.5 Challenges with the Implementation 35

8 Conclusions and Future Work **38**

1 Introduction

The Oshiya Debugger and Analyzer (ODA) is the main application that has been extended and enhanced throughout this thesis. ODA is a tool for debugging, visualizing, and comparing scheduling protocols. The word Oshiya indicates that the protocols are executed with the Oshiya declarative scheduling model [1].
ODA implements and provides the following key features:

Single- and Multi-version Protocols ODA supports the use of single-version protocols as well as multi-version protocols. In a single-version protocol there exists only one version of a data item x for all the transactions. For a multi-version protocol there are multiple versions of a specific data item x_i. This means that every time when a write request is done on a data item x_i a new version of that item is generated. At the beginning of a transaction each transaction gets a set of all the data item versions which have been committed before.

Simultaneous Protocol Execution ODA allows the user to execute different protocols simultaneously. It is possible to run single- and multi-version protocols at a time.

Navigational Debugging The user can debug a scheduling protocol by navigating through the steps of execution. Navigational debugging is possible in both directions, forward and backward. The possibility to go step by step or iteration by iteration is also offered to the user. While debugging the protocols, ODA displays the current scheduling state[1] as well as the state of the database with its underlying relations.

Workload Generator The user can generate his own workload with the workload generator. A workload is generated to be processed by the scheduling protocols. A user can define multiple clients each having multiple transactions. While execution ODA takes the requests[2] of the clients in the workload as an input for the scheduling.

Data Relation The data relation is the relation containing the data items. The data items get accessed by the requests within transactions. During the protocol execution these data items get modified by the write requests. The user can define his own data relation at the beginning of the scheduling execution.

[1] The state consists of the iteration and the step, e.g. state = {4,3} indicates that the algorithm is in iteration 4 and at step 3

[2] A request is an operation (read, write, commit, abort) in a transaction of a client. These transactions are specified in the workload

ODA takes one request per client at the beginning of a scheduling iteration. This indicates that ODA is working with a set of requests per iteration and not one request after the other. ODA then executes the requests according to Oshiya algorithm [1] which is the core of the ODA. Scheduling is performed in scheduling iterations where each of those consists of seven steps:

1 $\mathcal{R} = \mathcal{R} - \mathcal{E}$;
2 $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$;
3 $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$;
4 $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$;
5 $Execute(\mathcal{E})$;
6 $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$;
7 $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$;

Figure 1.1: 7 Steps of the Scheduling Algorithm

- In *step 1* all requests that have been executed in the previous scheduling iteration get deleted from the relation $\mathcal{R}$ which contains all the pending requests.
- *Step 2* adds the set of the next requests (one request per client) from the workload into the relation $\mathcal{N}$ which holds all the new request. These request are also inserted into the relation $\mathcal{R}$.
- $Q_{Revoked}$ identifies all requests in $\mathcal{R}$ that can not be executed, deletes them from $\mathcal{R}$, and adds an abort request to $\mathcal{E}$ in *step 3*.
- $Q_{Schedule}$ identifies in *step 4* which requests are executable and adds them to the relation $\mathcal{E}$ where all requests which are scheduled for execution are stored. Executability is determined by $Q_{Schedule}$. A request is executable if it is not accessing a data item in the data relation which is locked by another transaction.
- In *step 5* all requests that got selected in step 4 get executed by the *Executer* and possibly change values in the data relation. The *Executer* is a function which accesses the data relation and changes the values of the data items in the case of a single-version protocol, or inserts new versions of data items in the case of a multi-version protocol.
- *Step 6* adds the executed request to the relation $\mathcal{H}$. Relation $\mathcal{H}$ is the history with all requests that are relevant for future scheduling decisions of requests. This means that the requests which are irrelevant for $Q_{Schedule}$ are not in $\mathcal{H}$.
- In *step 7*, the requests which are irrelevant for future scheduling decision are deleted from $\mathcal{H}$. The request are identified by the scheduling query $Q_{Irrelevant}$.

ODA gives the user several advantages when developing a protocol. To create an own protocol, the user has to write three SQL queries for the three different scheduling queries

($Q_{Revoked}$, $Q_{Schedule}$, $Q_{Irrelevant}$). After setting up these queries the user can start the scheduling using ODA. With the navigational debugging it is possible to step through each and every single step of the algorithm and observe the protocol in its behavior.
Another scenario where ODA supports the user is the comparison of different protocols with a workload. The workload is scheduled for the defined protocols and the user can observe the execution of the workload by looking at the scheduling relations ($\mathcal{R}$, $\mathcal{E}$, $\mathcal{H}$) and the data relation.

1.1 Tasks

In this section the tasks of this thesis are described. The tasks are divided into three parts, namely the statistics, the visualizations, and the demonstration paper.

1.1.1 Conceptual design for Statistics inside ODA

ODA offers a simultaneous execution of different protocols with one workload. The comparison of the protocols or the analysis of a specific protocol is based on observing the states of the scheduling relations. In order to be able to measure and compare the performance of the protocols or to be able to analyze an execution of a protocol there is a lack of information. The given possibilities are limited and only low level information (state of the scheduling relations) is provided.
A concept of gathering statistical information for the users has to be elaborated. This statistical information supports the user in comparing and analyzing protocols. Therefore a statistics engine is developed. The design of the statistics has to be defined in a way, that ODA supports protocol specific as well as statistics which are not specific to a protocol. The question of what information can be stored and what is useful for anyone using the application is key. Not every information is useful an therefore is not worth to be collected. The implemented design has to support these needs.

1.1.2 Implementation of a Statistics and Visualization Engine

The collection of the statistical data itself is not very useful for the user. The user is interested in having these information displayed in a meaningful way. The visualizations of the statistics emphasizes the use of ODA for protocol analysis and comparison.
The implementation of the visualization covers the analysis of different kinds of charts for the measures. The visualizations support the user with a graphical interface. The user can choose the graphical representation of the statistical measures and can interact with the different chart types. This gives the user the opportunity to personalize the analysis of the protocols.
The implementation of the statistics has to be designed in a manner which allows to extend the application easily. In addition to the internal storage, management, and visual representation of the statistical measures[3], the ODA application has to support the export of the data, too.

[3] all the collected data (e.g. number of aborts, number of commits, or self-defined figures) is stored internally as a measure

1.1.3 Writing a Scientific Demonstration Paper

The other part of the thesis was to write a demonstration paper which has been submitted to the Very Large Database (VLDB) conference 2012. The contributions of the ODA had to be identified and described. The task inside the demonstration paper is to develop a sample scenario which serves as central theme in the whole paper. This scenario needs to be as simple as possible to guide the user through the benefits of ODA. After having created a scenario, in a interactive writing process together with several parties (Michael H. Böhlen, Boris Glavic, Christian Tilgner) the paper is being written.

1.2 Contributions

The main contributions of the thesis are the following:

Global Statistics The global statistics is a set of statistical measures which are considered to be standard statistic measures and provide a basic set for a user to interact with. The statistics are collected with the protocol execution as soon as the measure is available. All these measures can be selected for visualization. A measure consists of an actual value, a total value and an average value. Since a measure is stored every scheduling iteration, the actual value represents the statistical measure in the actual iteration. The total value is an accumulated figure of the actual value of a measure. The average value is the total value divided by the number of iterations.

Statistic Queries We allow the user to model custom measures by creating statistic queries. A statistic query is an SQL statement that accesses the scheduling state, database state, or the results of the break and analyze queries, which are described in the next paragraph. After a user has set statistic queries, they get executed after every scheduling iteration and get stored internally. A custom measure can be used just like a normal measure of the global statistics for all the visualizations.

Break and Analyze Queries Breakpoints are modeled as break queries inside ODA. Break queries are defined by the user with an SQL statement before starting the algorithm. These queries get executed after each scheduling iteration. Whenever a break query returns a non-empty result, the scheduling algorithm stops. This result is then presented to the user in separate tab. With the help of analyze queries, matching tuples are highlighted in the scheduling state. As well as the break queries, the analyze queries are written in SQL with the difference that they take parameters of the results from the break queries.

The concept of the break and analyze queries supports the user in detecting and analyzing errors in protocol executions and provides a framework to identify the behavior of a protocol with a certain workload.

Visualization The most important point for the user is the representation of all the statistical information. In a separate part of ODA the user can select different charts to display the selected statistical measures. We support a tabular representation, line charts, bar charts

and pie charts.
On top of this customizable part of the visualization, there are also some predefined visualizations. When ODA initializes and the execution of a protocol starts, the user finds the schedule history of each protocol on the start screen as well as progress bars of the protocols in another tab to observe the various selected protocols.
The concept of break and analyze queries is supported with a graphical interface. A seperat tab with a query browser allows the user to investigate the scheduling relations by executing SQL statements.

2 Sample Scenario

With the use of a sample scenario the benefits for the user by using ODA is presented. The sample scenario is a comparison and analyzing of Oshiya implementations of snapshot isolation and serializable snapshot isolation.

SI Snapshot Isolation (SI) is a multi-version concurrency control protocol. It creates a new version of an item x with every write operation $w(x)$. Every transaction accessing x, reads the the latest version x_i of the item x that is valid at the beginning of the transaction. This leads to reads that are not delayed. This is the reason that SI is ideal for read intensive workloads. A negative aspect of SI is, that it requires disjoint write-sets of concurrent committed transactions. This can be achieved by using the First-Committer-Wins rule. This rule ensures that one of the involved concurrent transactions with an overlapping write-set gets aborted. A write-set is the set containing all the items that a transaction is writing on [1].

SSI When we consider the Serializable Snapshot Isolation (SSI) protocol, which is also a multi-version concurrency control protocol, we have a different behavior compared to snapshot isolation. The difference is because SSI checks not only on the overlapping write-set, but also on (at least) two sequential rw-dependencies which can lead to a potential pivot structure (PPS). A potential pivot structure is a pivot structure where the involved transactions have not yet committed. As soon as the involved transactions commit, it turns into a pivot structure. A pivot structure is a combination of two consecutive rw-dependencies (t_1 reads an object and t_2 writes the same object) between two concurrent transactions. We can see two of these rw-dependencies in Figure 2.2 marked with the dotted arrows. Fekete has shown, that there is a pivot structure in every non-serializable history [2]. The SSI protocol introduced by Cahill identifies PPS and breaks them by aborting one of the transactions that are connected by rw-dependencies and this leads to a serializable history. This process guarantees that the resulting histories are serializable. A drawback of this method is that there may be false positives[1], because not every pivot structure leads to a non-serializable history. A pivot structure is only a necessary condition [3].

Example 1 *Consider a bank that maintains accounts of the customers Alice and Bob and these customers make individual transactions on the accounts. The two customers own a*

[1] A false positive is when the protocol aborts a transaction after having detected a PPS which does not turn into a non-serializable history if all the transactions commit. A history becomes non-serializable if the pivot structures occured in a cycle [1].

savings and a checking account which both belong to both customers. The bank stores account data in relation $Accounts$ as shown in Figure 2.1. Acc denotes the unique account number, Bal is the account balance, Type is the type of the account, and Owner specifies the account owner. The bank opens accounts with a contract. The contract says that the combined balance of their contract has to be positive all the time, i.e., there exists a constraint $C : x + y >= 0$.

$Accounts$

	Acc	**Bal**	**Type**	**Owner**
a_1	v	50	checking	D
a_2	w	50	saving	D
a_3	x	50	checking	AB
a_4	y	50	saving	AB

Figure 2.1: Account Relation

Assume Alice withdraws $70 from checking account x and at the same time Bob withdraws $70 from savings account y. Before each payment the ATM reads the balances of all the accounts of the owner AB and checks if C still holds if the withdrawal is executed. The resulting transaction of Alice and Bob can be modeled as follows:

$$t_a = r_a(x) r_a(y) w_a(x) c_a$$
$$t_b = r_b(x) r_b(y) w_b(y) c_b$$

ODA allows to schedule workloads by multiple protocols simultaneously in order to compare the different behaviors or the performance of the protocols.
Hence, we let ODA schedule the transactions of Alice and Bob with SI and SSI. We use this scenario to view the different behaviors of the two protocols with respect to the serializability of their histories. A history h is not serializable if there exists no serial execution of the transactions which is equivalent to the resulting history h. [3]

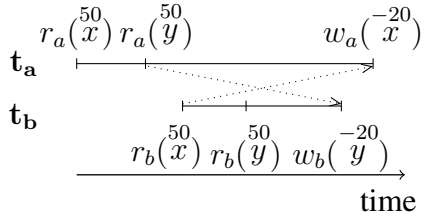

Figure 2.2: Banking Scenario

Figure 2.2 shows the scenario in a graphical representation. We will use that later to explain the differences of the two protocols in Chapter 3.

3 Break and Analyze Queries

In this chapter we address the concept of breakpoints, which are achieved by creating break and analyze queries. In Section 3.3 we will focus on the challenges with the concept of the break and analyze queries.

3.1 Break Query

To stop the execution of a protocol at a specific occurrence of an event, the user can set up breakpoints. A breakpoint is of use if a user wants to test a protocol execution on specific events or errors. Instead of debugging the execution until an event happens he can define a breakpoint and the execution stops automatically whenever this event occurs. ODA allows to model breakpoints as break queries. These break queries get executed after each scheduling iteration and the algorithm stops whenever one of the break queries returns a non-empty result. A break query is defined as a select query which returns some tuples. The result of a break query is shown to the user.
If we consider Example 1 and want to use the concept of the break and analyze queries to set up a breakpoint with a subsequent analysis whenever a constraint violation occurs.

Example 2 *First we set up a breakpoint whenever a constraint violation of constraint C occurs. We model the constraint C as a break query (BQ_{CV}) for the snapshot isolation protocol. The break query is defined as follows in relational calculus expression:*

$$\begin{aligned} BQ_{CV} =& \{O, S \mid Aggr(O, S) \wedge S < 0\} \\ Aggr =& \{O, SUM(B) \mid Accounts(\_, B, \_, O)\} \end{aligned}$$

This break query BQ_{CV} stops the execution of the scheduling whenever the sum S of the balances of all the accounts of an owner is negative. The sub query $Aggr$ sums the balances B for all the accounts of each owner O.
In the case of the serializable snapshot isolation we set up a different break query. Since the resulting history h_{SSI} of the SSI protocol is serializable we set up a breakpoint whenever a potential pivot structure is detected. The domain relational calculus is defined as follows:

$$\begin{aligned} BQ_{PPS} =& \{T_1, T_2, T_3 \mid RW(T_1, T_2) \wedge RW(T_2, T_3)\} \\ RW =& \{T_1, T_2 \mid \exists O : \mathcal{H}(\_, T_1, \_, r, O, \_) \wedge \\ & \qquad \mathcal{H}(\_, T_2, \_, w, O, \_) \wedge Conc(T_1, T_2)\} \end{aligned}$$

BQ_{PPS} detects the three involved transactions where two rw-dependencies occur. An rw-dependency exists between two concurrent transactions t_i and t_j when t_i reads a version of x and t_j wrote a later version of x. This is easier to understand if we look at Figure 2.2. In this figure we see two rw-dependencies indicated by the dotted lines. These two rw-dependencies form a potential pivot structure and as we know SSI then aborts either transaction t_a or t_b.

3.2 Analyze Queries

When a break query stops the scheduling it is interesting to the user why the execution has stopped. For the analysis of the result of a break query, the user can define several analyze queries to highlight matching tuples in the scheduling state[1]. An analyze query is defined as an SQL statement which takes some parameters of the result of the break query. Such a parameter is defined with the name in $ signs, e.g., '$owner$' for the parameter *owner*. For an example see the analyze query in Example 3.

When we want to analyze the involved tuples from the underlying relation, we set up analyze queries. An analyze query is defined specifically to a break query and highlights a subset of tuples in relation S. The purpose of an analyze query is to identify tuples in relation S that belong to the result of the break query. Usually an analyze query takes parameters of the result of its break query via referenced attributes of the break query.

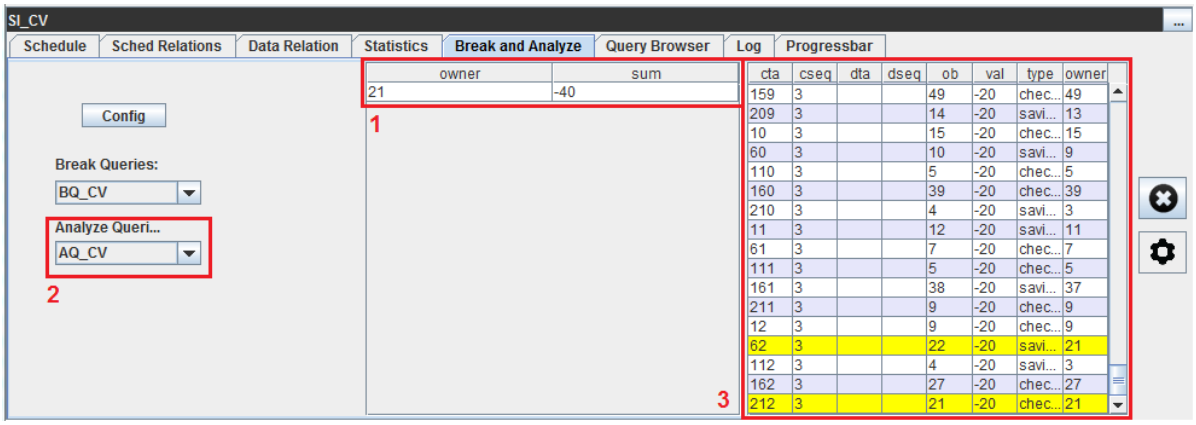

Figure 3.1: Break and Analyze Query Tab with showing CV

Example 3 *In the case of Example 2 we want to know which tuples in the $Accounts$ relation cause the constraint violation. This functionality is achieved with the analyze query AQ_{CV}. As shown in Figure 3.1 the execution stopped because BQ_{CV} for the SI protocol returned a result. This result is shown in the table in mark 1. By selecting an analyze query (mark 2) and a specific tuple in the result table (mark 1), the related tuples in relation S ($Accounts$) are highlighted in the table in mark 3. The analyze query is an SQL query that looks as follows: AQ_{CV}:*

[1]The scheduling state is the state of all the scheduling relations at a certain point in time.

SELECT $*$ **FROM** S **WHERE** owner=$owner$;

This analyze query selects all the $accounts$ *in the data relation* S *where the attribute* $owner$ *is the same as the selected owner of the result of the break query.*

In Figure 3.2 we see that the break query BQ_{PPS} returned two results and therefore the algorithm stopped. By clicking on one of the tuples in mark 1 the registered analyze query AQ_{PPS} gets executed (mark 2). This analyze query highlights the requests in relation $\mathcal{H}$ which participate in the PPS. You can see the highlighted rows in the table in mark 3.

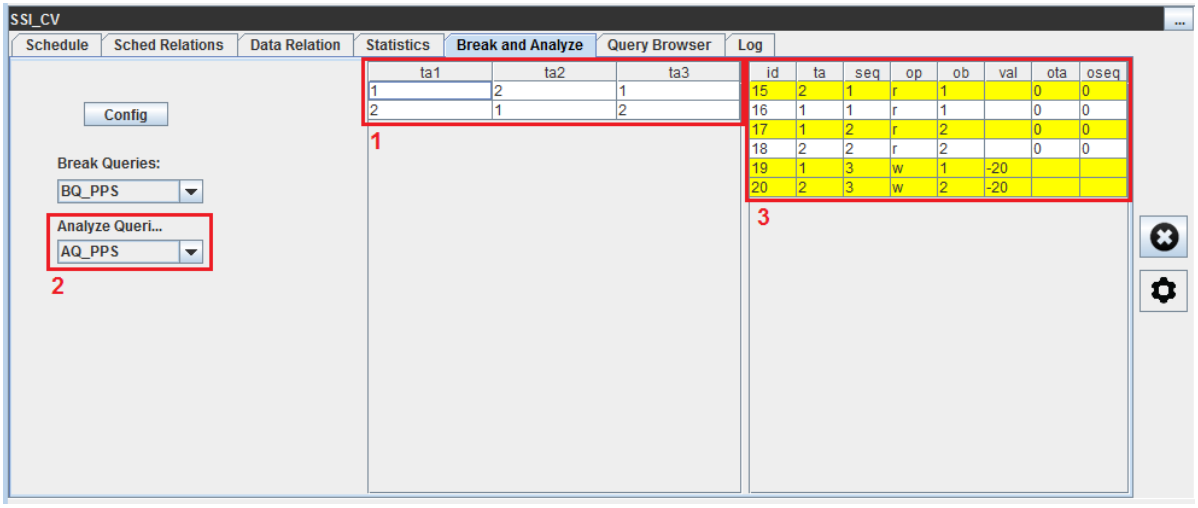

Figure 3.2: Break and Analyze Query Tab with showing PPS

3.3 Challenges with the Break and Analyze Queries

For the break and analyze queries the main challenge was the concept with one break query and several analyze queries defined per protocol. A new way of storing and executing the queries has to be defined since it is not really behaving like statistical measures (see Chapter 4). Also the different kind of query with the parameters where the prepared statement is no longer a good way of doing it. A self defined generation of queries is now doing this on the fly filling of the parameters.
A second challenge was the displaying of the results of the two different kinds of queries. The highlighting of the analyze queries is more difficult than the one for the break queries since you first have to get the right relation where you later on then highlight the matching tuples in the relation. This is realized by analyzing the underlying SQL statement of the analyze query. Out of that statement the relation can be gathered by looking at the $FROM$ statement in the first (sub-)query.

4 Global Statistics

In this chapter the concept of the global statistics is discussed. In the Section 4.1 the difficulties and challenges are described.
The concept of the global statistics is about collecting a basic set of measures. This set of measures is collected for every protocol during the scheduling. The global statistics are available for the visualizations in order to give the user the possibility to analyze the protocols. The measures are set up from the beginning, therefore the user does not have to specify a statistic query (see Chapter 5) for these measures. The set of measures is a list containing measures that we have used for our protocol comparison and analysis throughout the thesis.
The statistical measures are the following:

Aborts This measure represents the number of aborted transactions per iteration. It gets updated after the requests are written in the relation $\mathcal{E}$ in step 4 where it is the number of request with an 'a' (indicating an abort) as operation. The number of aborts is a measures which shows the behavior of a protocol.

Commits The number of commits corresponds to the committed transactions per iteration. As well as the aborts, the commits get updated at the same time just after the requests are inserted in $\mathcal{E}$ with an 'c' (indicating a commit) as operation. The number of commits is a key measure when analyzing or comparing different protocols.

Throughput The throughput is the number of requests that are scheduled per iteration. It is defined as follows:

$$throughput = \frac{\#commits}{\#iterations}$$

The throughput of a protocol is an indication of the performance and therefore also in the basic set of measures. This measure gets updated in step 4.

Selectivities of the three Scheduling Queries The selectivities of the different scheduling queries are the number of tuples returned by the individual queries $Q_{Irrelevant}$, $Q_{Schedule}$, and $Q_{Revoked}$. The selectivities are updated in the steps where the individual scheduling query is executed. When analyzing a scheduling of a protocol the selectivities of the three scheduling queries are core for the analysis.

Execution Times of the three Scheduling Queries In addition to the selectivities the execution time of the three scheduling queries is an interesting measure as well. Not only the selectivities of the scheduling queries but also the execution times have to be

optimized. The execution time is calculated with a native timer in Java which is started right before the execution of the query and then stopped after the result comes back. This means it also includes the overhead of the communication with Java and not only the pure execution time.

Execution Times for the Individual Steps As the execution times for the queries, the times for the steps are also measured with the native Java timer. The timer starts at the beginning of the step and lasts till the end of the step. This indicates that the execution time of a step also includes the updating of the statistics and the JDBC overhead.

Theses measures are updated every iteration at the point when they are available. The user can not only select the measure but also the kind of the value of the measure. There is the possibility of choosing between the actual value, which is the value in the current iteration, the total value, which represents the sum of all the actual values and the average value, that is the calculated average of the individual actual values per iteration.

4.1 Challenges with the Global Statistics

For the challenges there are two parts basically. The first thing was the process to the set of measures which belong to global statistics. The problem here is to identify the key measures of a protocol execution. After some iterations and further changes later in the project the final set was defined. The second point is the integration into ODA. Where is it possible to get the data and what is the best way with the fastest response so that it does not slow down the algorithm in general.

5 Statistic Queries

In this chapter the concept of the user defined measures is discussed. A user can define measures as statistic queries.
A user can not only collect data of the predefined measures but he can define his own ones. These measures are realized declaratively as statistics queries within ODA. This concept opens the application to be personalizable for a specific need of any user.

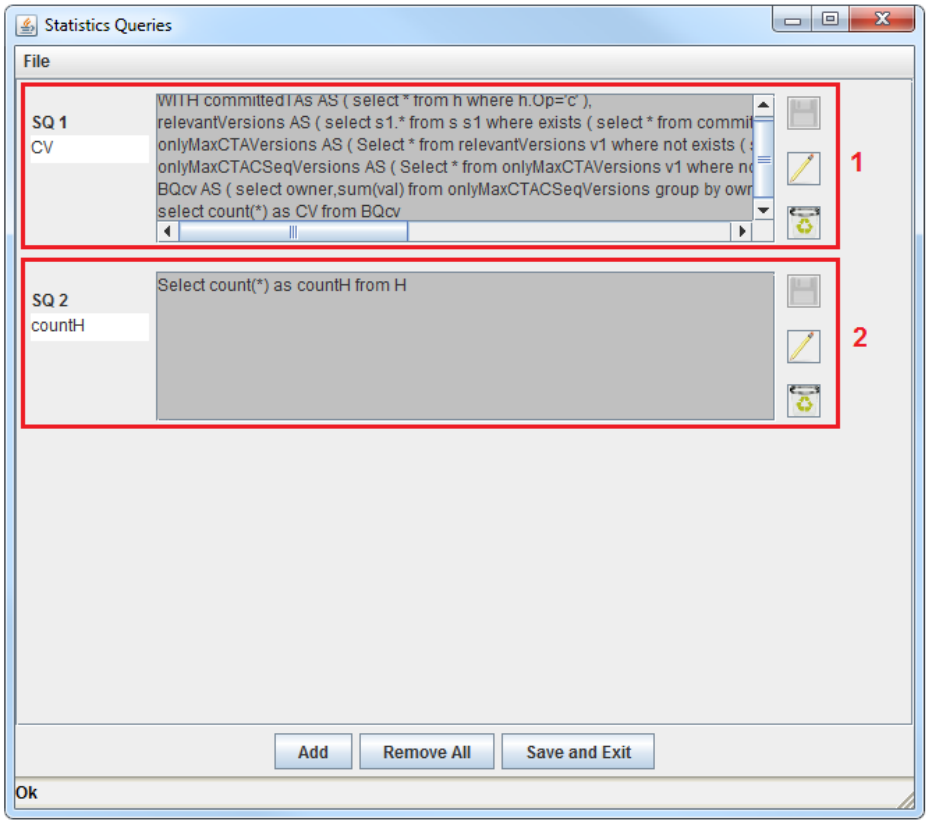

Figure 5.1: Statistic Queries

These user created measures are realized by SQL queries. A statistic query has to follow a certain schema to be compatible with the application. Every query has to start with "SELECT count(*) as <name>" where the count(*) is the value which gets stored after every execution after step 6 in a scheduling iteration. The name of the measure is given by the word that follows the "as".

After having set a statistic query the user can select the measure in the statistics tab to include it into the visualizations which will be explained in Section 7.2. The measure starts to get collected by the time the query is saved. You can configure the statistic queries any time, but the measures do not get recalculated historically. So the best way to set up the statistic queries is right at the beginning before even starting the algorithm or to restart the complete scheduling after changing statistic queries. To make it more convenient for the user to set up the statistic queries there is the possibility to load and save the defined queries. The internal saving of the queries is done automatically, the queries are set up internally and the collection of the resulting measures is started.
Statistic queries are executed for every executed protocol and it is in the responsibility of the user, that a query can be executed for every database schema of all the protocols.

Example 4 *Consider Example 1 where we have seen a banking transaction with a constraint. To discover the number of occurrences of these constraint violations we set up a statistic query which counts the number of constraint violations.*
The SQL code for the statistic query is equivalent to the break query in Example 2. The only difference is the "select count(*) as CV from BQ_{CV}" at the end of the query (mark 1 in Figure 5.1).
The second example of a statistic query is the cardinality of $\mathcal{H}$. Since $\mathcal{H}$ is the relation containing the relevant history for future scheduling, not every protocol behaves the same. The cardinality influences the performance of the scheduling queries and is a measure that can be used for analyzing protocols.
The SQL query for that measure is the following (mark 2 in Figure 5.1):

```
SELECT count(*) as countH
FROM H
```

5.1 Challenges with the Statistic Queries

The challenges with the statistic queries are the concept of the internal storage of the queries and the user interaction. Since the statistic queries are done for all the protocols but the underlying code is for every protocol separate it was quite a challenge to achieve this propagation in directions of the top level down to a internal protocol level[1]. As mentioned in the standard statistics section, the statistic measures are stored on a protocol level. Whenever a statistic query is set up the individual statistics of the protocols have to know what to execute and to store. The same propagation applies if a statistic query gets changed or deleted.
The second main challenge in this concept was the user interface. An intuitive graphical interface has to be designed in order to support the user with all the necessary features. The loading and saving of the queries have to be taken into account as well.

[1]For a detailed explanation of the concepts with the top and protocol levels see Chapter 7

6 Visualization

This chapter discusses the concept of the visual representation of the collected statistical data to the user.
The task is not only to collect all the statistical data but also to display it to the user. There are basically two different approaches to achieve this goal. One is that you exactly know all the data and you know how the different users will look at the data. The other approach is to provide basic functionality and let the user fully decide on how he wants to look at the data. Because the users are very heterogeneous the second approach has been chosen.

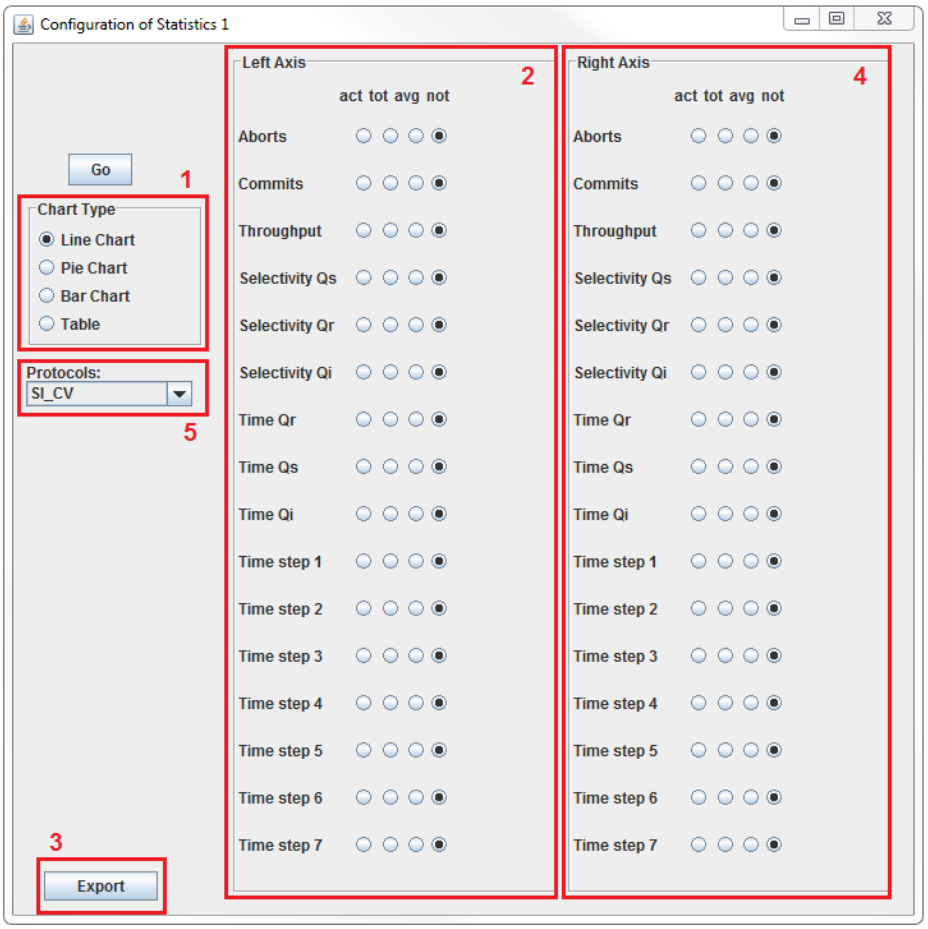

Figure 6.1: Statistic Config

In the end the user can select out of four types of representation of the data. There is the choice of a line chart, a bar chart, a pie chart and a normal table. For every chart the user can

then select the measures per protocol. In the table it always shows all the three values (actual, total, average) of the selected measures, whereas, for the other graphs the user selects the ones he wants to be displayed. To illustrate this possibilities there is Figure 6.1 which shows the chart configuration for the statistical representation of the data. In mark 1 there is the type of the charts which can be generated. By checking one of the boxes the chart is dynamically generated on the fly with the selected measures. Mark 2 shows the measures with the different kind of values that can be selected for the charts. Left axis means that theses measures get plotted on the left y-axis of the line chart. For the other types of charts this is not relevant. For the second y-axis of a line chart there is a separate listing in mark 4. The switching between the protocols and the belonging measures is achieved with the combo box in mark 5. It does not automatically select the measures for all the protocols.

6.1 Statistics

This section shows the different types of representation of the statistic measures.

6.1.1 Line Charts

Line charts are a widely used method to display temporal data to a user. It can show data changing over time and you can find out some dependencies or commonalities with the different measures when looking at the charts. A line chart is not optimal to compare the actual values of the measures, because of the jumps that a measure does, i.e., commits do not happen every scheduling iteration.
Line charts are generated on a very generic way, which means that the limitation of the programmed line chart generator is as small as possible. The user has the possibility to select measures and the left-hand y-axis as well as on the right-hand side. The x-axis is the number of iteration all the time since all the measures are stored on a scheduling iteration basis.

Example 5 *Consider Example 1 with the constraint. In Figure 6.2 we have a sample line chart with one y-axis. It shows the measures of a snapshot isolation and a serializable snapshot isolation protocol for the aborts and the commits. The values for all the measures are the total values. The actual iteration the execution stopped is 9. One can see that SSI has one abort in iteration 7 where SI has no aborts so far. In Figure 6.3 you see that there were some constraint violations in the protocol execution of the snapshot isolation. This figure is taken at a later stage in the scheduling, but with the same settings.*

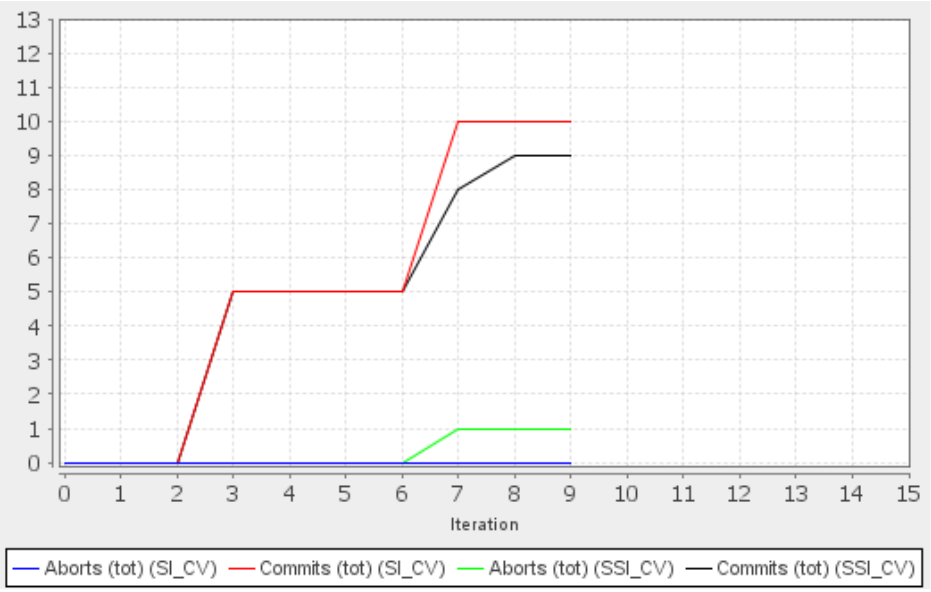

Figure 6.2: Linechart for two Protocols

6.1.2 Bar Charts

A bar chart allows to compare different measures of the selected protocols. Usually you compare total or average values of various measures. There is one color per protocol and the measures are on the x-axis. A bar chart provides a visual presentation of categorical data or data which can be grouped in distinct groups. Since the chart is generated dynamically as the scheduling proceeds, you can observe the change over time, but it shows only the latest data of a measure.

Example 6 *When we look at Figure 6.3 we see a comparison of two protocols, namely SI and SSI. According to the described sample scenario in Example 1 we see the total number of aborts and constraint violations (CV). As we expect we have no constraint violations when using the SSI protocol but the number of aborts is much higher for the same.*

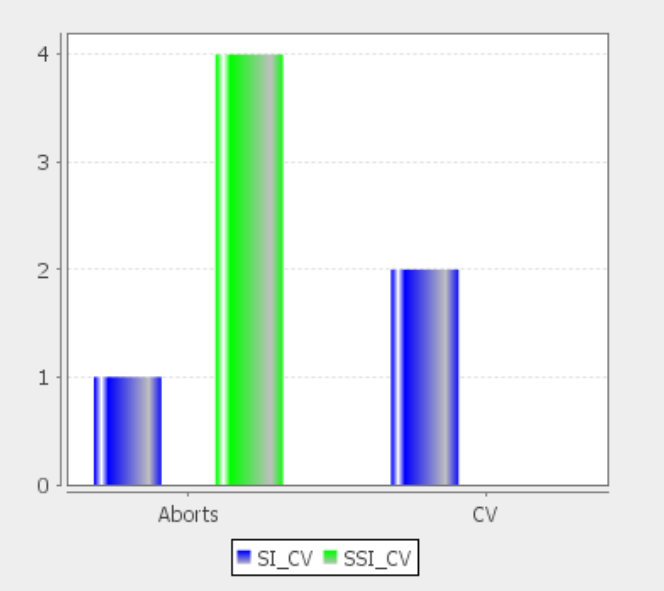

Figure 6.3: Barchart for two Protocols

6.1.3 Pie Charts

A pie chart is a circular chart that gives the possibility to see the proportional representation of the quantities with respect to the whole. In the example chart below you see the times of the different steps. You see the time in milliseconds and the percentage of the share. Typically one selects the average time to get the proportions.

Example 7 *The pie chart in Figure 6.4 shows a comparison of the average execution times of the individual steps of the iterations of the snapshot isolation protocol. We can see that the blue part, which shows step 2, is the biggest piece of the pie. This indicates that the inserts of the new request from the workload takes most of the time.*

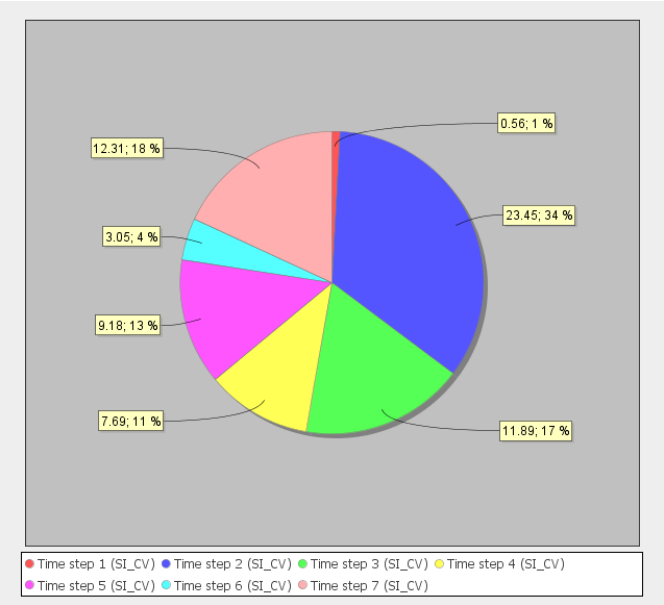

Figure 6.4: Pie Chart with Execution Times

6.1.4 Table

A table offers a container to see the plain numbers of the selected measures. It automatically selects all the three values of the selected measures (actual, total, average). The table can be used for a plain representation of the numbers.

Example 8 *In Figure 6.5 there is an illustration to the sample scenario from Example 1 there is a screen shot of the table containing the different values for the measures aborts and commits. The measures are for the two executed protocols (SI and SSI).*

Measure	Act.	Avg.	Tot.
Aborts (SI_CV)	0	0.0	0.0
Commits (SI_CV)	4	1.12	19.0
Aborts (SSI_CV)	1	0.12	2.0
Commits (SSI_CV)	2	0.94	16.0

Figure 6.5: Table

6.2 Progress Bar

In a separate tab called "Progressbar" in the panel of the top protocol (Figure 6.7 mark 4) there is a kind of summary of the progress for all the protocols as shown in Figure 6.6. You see a progress bar for every protocol with the number of transactions and the completed transaction.

The progress bar represents the percental completion of the workload. A workload consists of several transactions which either get committed or aborted. When all the transactions are done, the progress bar shows a 100% completion. In the middle of the figure the calculated total number transactions as well as the number of completed transactions is shown. These numbers are collected from the workload (total number of distinct transactions in the workload) and in relation $\mathcal{H}$ with all the history (total number of completed transactions).

Example 9 *In the example here we have the two protocols, snapshot isolation and serializable snapshot isolation, processing both the same workload. Both have the same number of transaction in total, 250. SI has processed 15 transactions so far, which is 6% of the whole workload. SSI has executed 14 transactions which is 5%.*

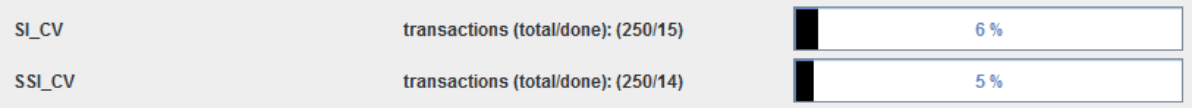

Figure 6.6: Progress Bar

6.3 Schedule

As a starting point for the user of ODA there is the schedule tab that you see on Figure 6.7. On the left hand side (mark 1) of this tab the actual schedule of the clients is presented to the user and it contains also a summary of the most important informations like the number of clients, the number of transactions, the number of aborts and the number of commits on the right hand side (mark 3), as you can see on the screen shot.
The schedule displays the scheduled requests of all clients in a table and this table gets updated after every scheduling iteration. There is no total ordering because ODA processes the requests in sets. One set consists of one request per client. The possibility of having delays in a workload was the tricky point to deal with when implementing this feature. A delay is like a placeholder in the workload where no request of the specific client is scheduled. Because the scheduler takes one request per client per iteration, it offers more possibilities to the user in creating a workload. It is now possible for a client to delay the scheduling of requests. Since a delay is not part of a schedule they are skipped because they do not belong in the schedule.

Example 10 *In Figure 6.7 there is a screen shot of two transactions. The transactions of the two clients are the following:*

$$t_1 = r_1(1) dr_1(1) dw_1(1) dc_1$$
$$t_3 = r_3(1) r_3(1) ddc_3$$

The resulting schedule for the first seven iterations is shown in the figure below.

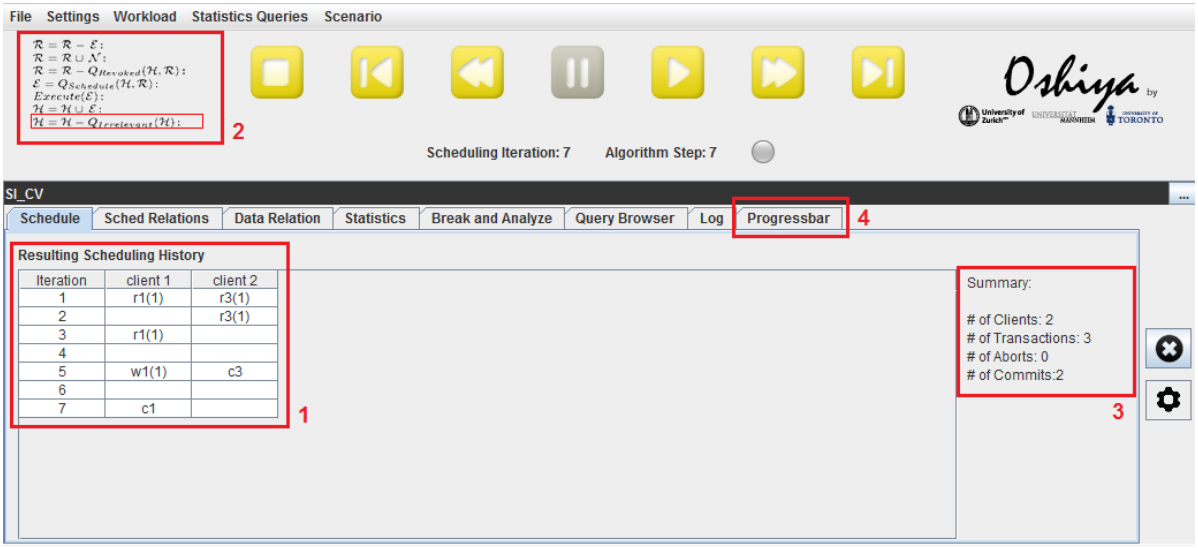

Figure 6.7: Schedule Tab with two Clients and Delays

6.4 Visualization of the Scheduling Algorithm

In the top left corner of the screen of ODA (Figure 6.7 mark 2) you see the scheduling algorithm. The red bar shows the current step of the algorithm. The bar moves prior to the execution of the step at the time when the internal state of the application changes.

6.5 Query Browser

The "Query Browser" tab (shown in Figure 6.8) gives the opportunity to execute select and update statements to all the relations (S, $\mathcal{N}$, workload, $\mathcal{R}$, $\mathcal{E}$, $\mathcal{H}$ and the underlying system tables[1]) in the database. It is split into two sub query windows. On the left of the tab (mark 1) you can enter an SQL statement and the result is shown in the table on the right hand side (mark 2). The queries are protocol-specific which means that you have a different connection on the different protocol tabs and therefore only access to all the tables belonging to the chosen protocol. It is not possible to do comparisons of the tables across the protocols. In other words, if you have two protocols selected then you have the two protocols one above the other. If you want to query the second protocol, you have to use the window at the bottom in order to be able to query the tables belonging to the second protocol.

[1]The system tables are the underlying tables to the relations S, $\mathcal{R}$, $\mathcal{E}$, $\mathcal{H}$ which contain all the tuples. The tuples in these tables never get deleted.

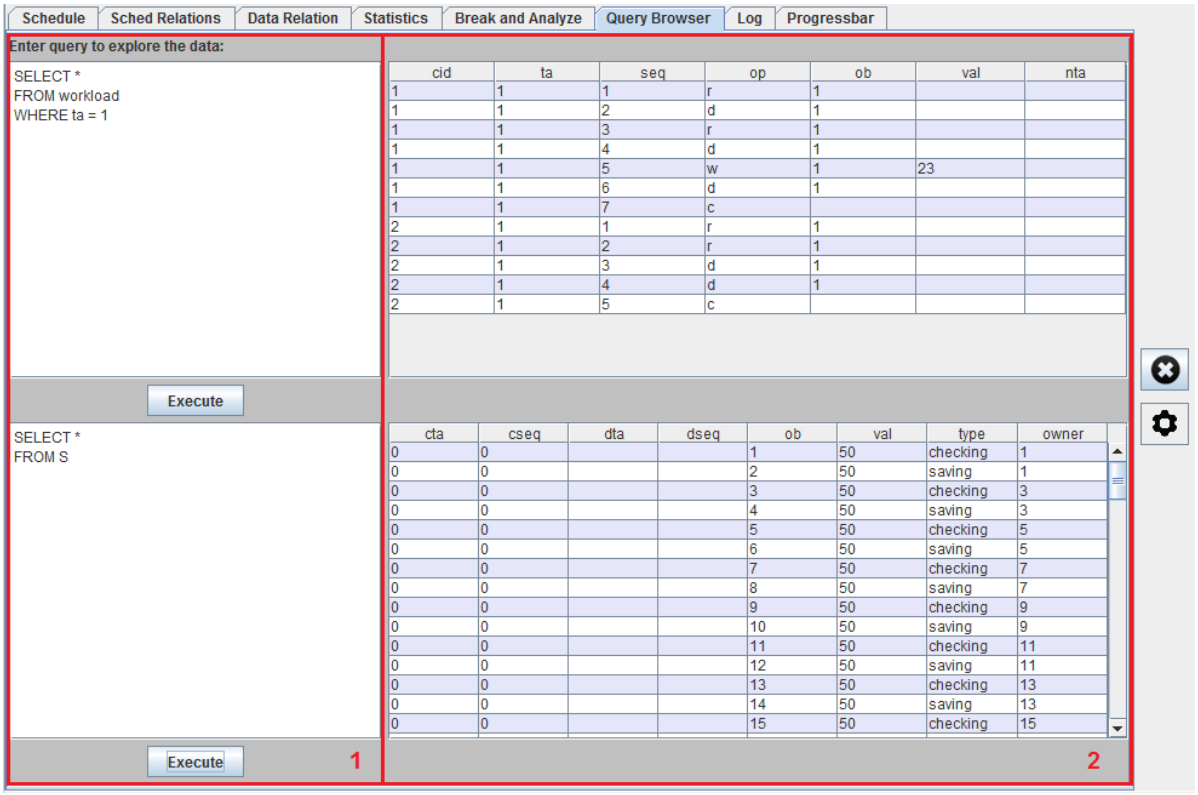

Figure 6.8: Query Browser Tab showing two Sample Queries for one Protocol

Example 11 *Figure 6.8 shows two examples of select queries for a protocol. The first query selects all the tuples from the workload where the transaction is 1. There are tuples for client 1 and 2, since in the workload the numbering of the transactions for a client starts with 1. The second query select all the tuples from relation S. Relation S is the data relation containing all the accounts as we have seen in Example 1. The additional attributes are necessary for the internal scheduling algorithm which is not part of this thesis.*

6.6 Challenges with the Visualization

The main problem was the heterogeneity of the users that use the application and the large possibility of measures that could be collected. The set of different supported charts has grown initially and afterwards narrowed down step by step. The code was designed to be generic, which allows the user to define all the measures and presentation himself. The challenge here was in the programming of this flexibility and in the selection of the final set of supported charts.

7 Implementation

In this chapter, the focus is on the implementation of the concepts discussed in the previous chapters with its challenges (in Section 7.5).

7.1 Documentation of Code

An overview of the code for the statistics is described in this section. It is not meant to be a detailed documentation but a summary of the most important classes and their functions regarding the different concepts and the underlying model which holds all the information at a protocol or at an application level.

Model - View - Controller

The design pattern that is used at a code level is the Model-View-Controller (MVC). The aim of this design pattern is to separate functionality. There are three main packages in the project and the classes are separated with respect to their functionality. All classes that store data are placed inside the model package. The gui package (view) is purely for classes with a visual representation. These classes do not store data and they have no logic in them. The controller package with its classes is responsible for the logic.

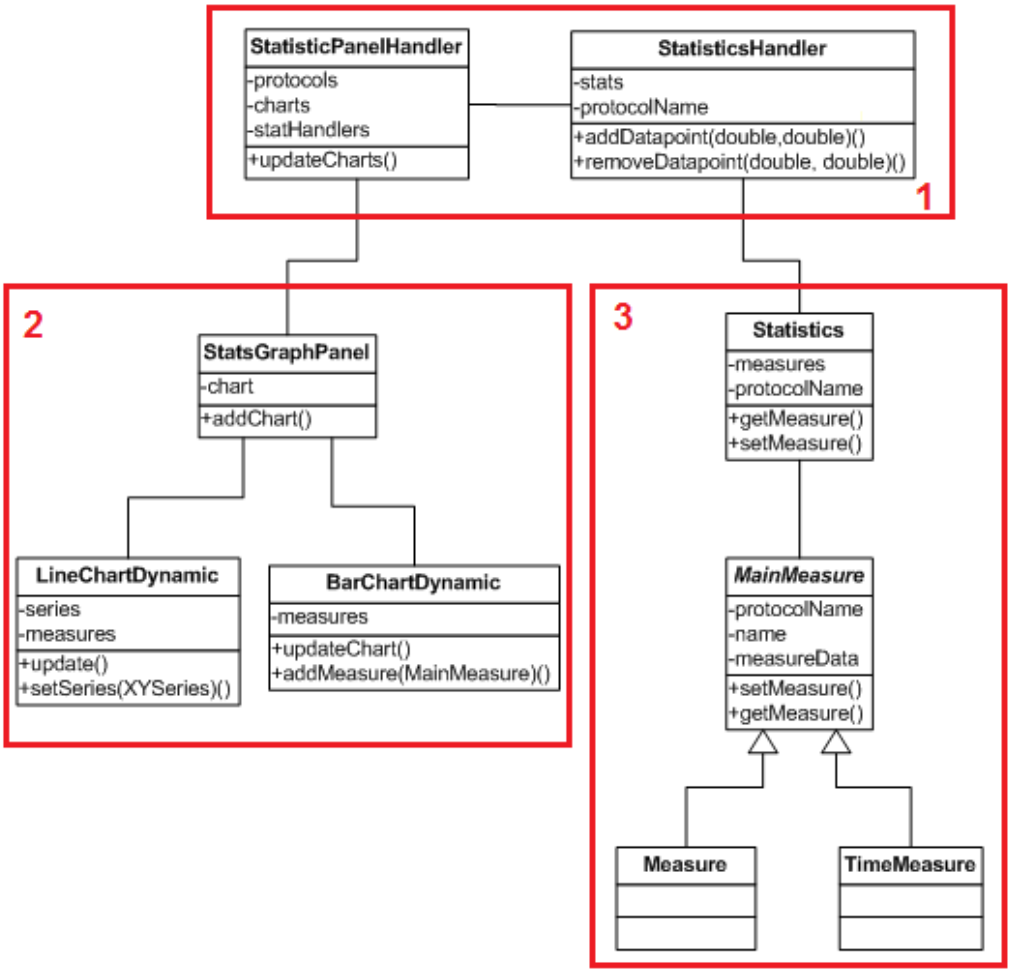

Figure 7.1: Class Diagram

Example 12 *In Figure 7.1 there is a simplified overview of the main classes that were implemented for the statistics and the charts.*
In mark 1 there are two controller classes. The StatisticsHandler is responsible for the Statistics of a protocol. The Statistics itself contains all the measures that are stored for a protocol. This includes not only the basic set of MainMeasures but also the statistic queries that got set up. A MainMeasure is either a normal Measure or a TimeMeasure having times per iteration (execution times). There is a different handling of the data types for those two classes, that is why they inherit from an abstract class MainMeasure. All the classes in mark 3 are considered to be in the model of the statistics. In mark 2 we have the classes which are responsible for the view. On the top there is the StatsGraphPanel that is the area where the charts are displayed. To simplify the figure, there are only two different types of charts in the figure. We have the LineChartDynamic and the BarChartDynamic. Both classes stand for a chart. One can draw a new chart by initializing the class and passing the measures. There is also a method update() which updates the charts by checking the measures for changes.

The distinction of the functionalities of the classes and the clear separation of the packages supports the understanding of the underlying code. The second big plus is the possibility to extend the code. If you want to add a new chart type, you write a new class for this chart, add it to the controller and include it in the views. The same applies for new measures. To add more features to the application you stay with the concept and add classes in the appropriate packages.

7.2 Visualization

This section is about the visual representation of the statistical concepts. It emphasizes the different possibilities offered to the user of ODA to support the analysis of the protocols.
The data for all the charts is the statistic model which is stored internally per protocol. The application transforms the data into the required data structures for the individual chart types. Therefore you have all the collected data ready to be displayed even historically, given that the measure has been introduced before.
All the graphical programming was done with the *Swing* library[1]. A very useful tool was the *WindowBuilder*[2], which is a plugin for Eclipse[3]. This builder is kind of a "WYSIWYG" editor for graphical user interfaces. With the use of that tool it was possible to generate an initial code that was manually adjusted to our needs for the graphics. The main reason why we had to adjust the code manually was because we have a lot of generic code. The generic components are not well supported in the WindowBuilder. For the three first types of charts the library *JFreeChart*[4] was used. JFreeChart is a free Java library and API for many different chart types. Thanks to to good documentation of the API it is easy to extend and customize the existing classes and charts. There are a lot of samples available on their website.

7.3 Additional Features

This section contains additional extensions to ODA that were done in the thesis.

Save Scenario Functionality for Statistics

ODA supports to save all the setting up of a scenario. By saving a scenario you save the protocols, the workload, the data relation, all the statistic queries, all break and analyze queries and also the different charts. This functionality helps to cut the preparation time to run a certain scenario and also allows the user to load a specific scenario from other users. The concept of storing these data structures was difficult for the internal representation of the charts and queries.
The queries are stored by the file names of the different queries, which indicates that by the

[1] http://docs.oracle.com/javase/tutorial/uiswing/
[2] http://www.eclipse.org/windowbuilder/
[3] Eclipse is a software development environment. http://www.eclipse.org/
[4] http://www.jfree.org/jfreechart/

time you load a scenario you have do make sure that all the files containing the queries are there as well.
The charts are stored with the type of the chart and all its different measures which will then later on be populated.

Export of Underlying Data of the Charts

The export feature allows the user to get at the statistics data and use it also outside ODA. The format for the exported data was chosen to be in a text file which can easily used for gnuplot[5]. The export button is placed in the statistics configuration for the graph in Figure 6.1 in mark 3. Clicking the button saves the selected measures into a text file which is stored in the folder $Resources/Exports$. The first line of the file has the list with the measures. After the second line there are the data points for each measure. The first number on every line is the iteration and this is followed by all the measures.

Example 13 *Consider an example export where the actual values of the aborts and the commits have been selected for two protocols, namely snapshot isolation and serializable snapshot isolation. In the first line there is the information about all measures. The first number on every following line is the scheduling iteration starting from 0 to 5.*
You can see for example that in iteration 4 both protocols show one commit.

```
MeasureNames    Aborts (act) (SI_CV)    Commits (act) (SI_CV)    Aborts (act) (SSI_CV)    Commits (act) (SSI_CV)
0.0     0.0     0.0     0.0     0.0
1.0     0.0     0.0     0.0     0.0
2.0     0.0     0.0     0.0     0.0
3.0     0.0     0.0     0.0     0.0
4.0     0.0     1.0     0.0     1.0
5.0     0.0     0.0     0.0     0.0
```

Figure 7.2: Export of Measures

7.4 Testing

During th implementation period, the size (lines of code) and complexity of ODA increased. In order to guarantee the correctness and the stable functioning of ODA with all its features it is very important to have a testing phase. This phase has to be at the end and in line with the implementation phase.
It is not only important to have the testing at the very end of the project but also to have some testing going throughout the implementation phase. The difference of the two kind of tests are the scope. While you are implementing new features it is mostly the testing of the individual components. Towards the end of the implementation phase the test are covering more and more components in one test. In the end the testing is also on a GUI level. This testing is the trickiest part because you try to cover as many possible behaviors of a user as possible to prevent uncontrolled behaviors of the application.

[5] http://www.gnuplot.info/

7.4.1 Testing Components

After completing the implementation of a new component the correctness of the internal functionality and the integration into the overall application had to be tested. The internal functionality is first the correct execution at the right time and second the right storage of the correct data. Most of the time this is tested with handmade calculations compared to the actual results of the internal numbers. The tests are made for several critical setups and settings.
The testing of the integration is harder to fulfill. Because the effects of the new component was not always clear, the testing was by using the application in all its behaviors and the functionality of the buttons. If the results of the tests are all as expected and there are no errors showing up while execution the test was successful.

7.4.2 Testing Graphical User Interface

The third part was the testing of the GUI. This is difficult to test because some parts of the GUI are dynamically generated and therefore several different settings and user behaviors need to be tested.

7.5 Challenges with the Implementation

This thesis is based on a running application which has been developed by two other students in a Vertiefungsarbeit and a Bachelor thesis.
The state of the application at the start of the thesis was a running tool that implemented the Oshiya algorithm and supported the use of one protocol. As well as the protocol one could select a workload that was used for the execution of the seven steps of the Oshiya algorithm. The logic of the debugging feature was also implemented and with the use of the navigational controls you can step backwards as well as forwards.
Throughout the phase of the implementation there were following main issues:

The design goal changed from one protocol to multi protocol The fact that the application now has to support the comparison of multiple protocols changed a lot in the underlying code. Unfortunately the concept that was chosen at the very beginning of the life cycle of ODA was not supporting the extension to run with multiple protocol. In order to make that big change possible a refactoring at the beginning of the new phase was indispensable.
Since there were a lot of dependencies in all the different classes and the separation of model, controller and view was not given, the implementation started with major adjustments in the existing code with respect to the agreed concept of the new ODA. The goal was to do as few changes as possible but as many as needed.

Team of students that are programming at a time Another challenge was the fact that there was not only one student working on the application. The communication and the definition of the interface was key to prevent upcoming problems. It was possible to

develop the statistical part in own packages for the model, the controller and the view with only a few dependencies in already existing and therefore conflicting classes.
The impacts of the changes had to be carefully taken into account and the other parties had to be informed about major changes. With the use of a version control tool the problems with the code were solved and conflicts in different versions of classes could be successfully avoided.
The use of separate packages for the statistics has the advantage of the independence of the other part, the change of the application from one protocol to many. It was possible to implement the statistics using only the version which supported one protocol and later on change only parts of the code to support also the multi protocol version of ODA. Since this change was known at the beginning but the interfaces with the new classes were not known, a generic definition of the interfaces in the statistics was used.

Generic programming to allow extensions The final state of ODA was clearly not known at the beginning of the thesis and has been revalidated all the time. It was important to allow the addition of other statistics or further changes in the implementation, the storing of the statistics, the collecting of statistical data or the representation of the statistics measures and the user interactions. Therefore a generic programming was the best choice where ever possible. This was not easy in the beginning but saved a lot of time when the extensions or changes came true.
One aim of the thesis was also to make the application extensible. With the way how the statistics are implemented, not only the programmer can extend the statistics but also the user of ODA can extend the statistics with his own measures and definitions of statistical informations he wants to store and display by using statistic queries. The generic programming allows the user to treat his own measures like the predefined ones and sees no differences.

Changes of the requirements As mentioned in the point above the requirements were subject to changes. To develop the statistics, we evaluated the steps every time after the implementation to improve the overall application. The use of a measure or a concept was critically analyzed and changed if it was necessary. This leads to an optimal result but is hard to deal with while developing. Many times code had to be adapted and concepts had to be rethought to get a better outcome for the user or the benefit of ODA. In the beginning the task was to build a statistical engine and a visualization. The goal was still very broad but in order to narrow it down also the code needs to be adaptive and open for extensions.

Challenges with the visualization The graphics are programmed using the Java library *Swing*. By using the eclipse plugin *WindowBuilder* it was possible to get a first sketch on the screen. But if you want to adapt the GUI and make it dynamic it was very hard to achieve. A second issue is the compatibility of Windows and Mac. Even though Java is supposed to be platform independent it is not with respect to the graphical representation.
Another big challenge was the design of the user interfaces. This problem occurred for example when designing the GUI for the configuration of the graphs in the statistics

tab. How is it possible to draw an as intuitive interface as possible. The result of this evaluation process was not directly a good one at the beginning, but after four iterations and analyze phases a solution was implemented.
With the use of the library JFreeChart you have a good and solid framework, but the bigger the possibilities the more important it is to focus on the usability and on the purpose. It makes no sense to allow everything but to make a decision about what it should look like and what gives the most benefit for the purpose of ODA.

8 Conclusions and Future Work

In this thesis, the advantages of ODA were enhanced. The addition of a statistics engine together with concepts to visualize the statistical data emphasizes the characteristics of ODA. The concept of breakpoints which is acheived with break and analyze queries supports the user in analyzing and comparing scheduling protocols. To give a summary of the concepts, we take three different scenarios to illustrate the benefits:

Developing protocols ODA with the implemented concepts of this thesis is a helpful tool to develop an own protocol. It offers an interface to define scheduling queries. With the help of the navigational debugging it is possible to step through the scheduling iteraions of the Oshiya algorithm and analyze the activities at a very granular level (requests in the scheduling relations $\mathcal{R}$, $\mathcal{E}$, $\mathcal{H}$). To get an overview of the performance and the proper functionality of the executed protocols, there is the concept of the statistics. By using the graphs or even defining own measures with statistic queries the protocol execution can be observed on a higher level than just monitoring the resulting histories or the tuples being selected and executed in the scheduling relations. The dynamic generation of the schedule provides a digram with the resulting histories, which is similar to the schedules shown in common database literature about databases. By defining break and analyze queries the user has the possibility to interrupt the protocol execution when a break query returns a result. When developing a protocol, one likes to know when errors occur. Therefore the user has to set up a break query which detects errors. With an appropriate analyze queries the user can then further investigate the underlying relations with the involved tuples to identify the problems in the protocols.

Comparing protocols ODA simultaneously executes all selected protocols with a generated or selected workload and creates a schedule history per protocol. The protocols are one below the other. As already shown throughout the thesis the user can use predefined statistical measures of the global statistics or he can define his own statistical measures by setting up some statistic queries. Depending on the type of comparison the users defines some charts which can include measures of both protocols. The availability of the measures of all the executed protocols simplifies a graphical comparison of statistical data of the protocols.

Statistical protocol analysis The developed statistics can be used to analyze a protocol with a specific workload and data relation. One can define workloads with specific characteristics and define break and analyze queries. To include the break queries into the statistical charts a user adds a statistic query with an modification of the query.
ODA offers flexibility to analyze a protocol statistically in a customizable way.

Future Work

Web application The application uses the local file system to load or save files for the settings and the stored queries. Therefore it is not designed as a web application. With a web application the flexibility of the use of ODA would increase significantly. The possibility to analyze or compare protocols with a web application is a feature that has not yet been implemented.

Extending the statistics The statistical part in ODA can be extended by additional types of charts. A limitation of the statistics is that a statistic query is defined for all the protocols. It would be nice to have different statistic queries for the different protocols in order to be possible to collect protocol specific measures.

List of Figures

1.1 7 Steps of the Scheduling Algorithm 10

2.1 Account Relation 15
2.2 Banking Scenario 15

3.1 Break and Analyze Query Tab with showing CV 17
3.2 Break and Analyze Query Tab with showing PPS 18

5.1 Statistic Queries 21

6.1 Statistic Config 23
6.2 Linechart for two Protocols 25
6.3 Barchart for two Protocols 26
6.4 Pie Chart with Execution Times 27
6.5 Table 27
6.6 Progress Bar 28
6.7 Schedule Tab with two Clients and Delays 29
6.8 Query Browser Tab showing two Sample Queries for one Protocol 30

7.1 Class Diagram 32
7.2 Export of Measures 34

Bibliography

[1] C. Tilgner, B. Glavic, M. H. Böhlen, and C.-C. Kanne, "Declarative Serializable Snapshot Isolation," in *ADBIS*, pp. 170–184, 2011.

[2] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making Snapshot Isolation Serializable," *TODS*, vol. 30, no. 2, pp. 492–528, 2005.

[3] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable Isolation for Snapshot Databases," *TODS*, vol. 34, no. 4, pp. 1–42, 2009.