HØGSKOLEN I AGDER
Agder University College

IKT407 Student Project
Faculty of Engineering and Science

User behaviour logging for data warehouse tuning

by

Eirik Aanonsen
Lars S. Breistrand
Morten Kråkvik

November 18, 2004

Project Report
Web mining and data analysis

Abstract

The goal of the European Accessibility Observatory (EIAO) is to contribute to better e-accessibility and to increase the use of standards for online resources. ROBACC (ROBot assessing web ACCessibility) is a crawler that assesses the accessability of a web sites and stores it in a database. The goal of our project was to design tools to log the users behaviour, analyse the information gathered and use this to improve database performance and the web interface. To do this task we have designed two tools, one for analysing the database log and one for analysing the web log. The database analysis tool gives us an overview over all queries (the most common, slowest, most time consuming, a summary). The web log analysing tool looks at the web log in a similar way. It gives us an output of the values selected by users when searching for information on the ROBACC-site. Since the EIAO project is still in an early phase the number of visitors to the ROBACC-site is limited and this has made our task a bit more difficult since we have not had much data to base our recommendations on. Nevertheless we have found areas were we have suggestions to improve the database and web interface.

Preface

We would like to thank the following people for helping us through this project.

- **Mikael Snaprud and Ole-C. Granmo**
  For being our supervisors in this project, giving us advises on how we should approach to solve the problems and giving us feedback during the project period.
- **Morten Goodwin Olsen**
  For being very helpful with solving any technical issues related to Steinbit.

Contents

Preface **i**

1 Introduction **1**
1.1 Background . . . 1
1.2 Goals and scope . . . 1
1.3 Limitations . . . 1

2 Problem descriptions **2**
2.1 Database logs . . . 2
2.1.1 Configuring PostgreSQL logging . . . 2
2.1.2 Looking at the database log output . . . 2
2.1.3 Database analysing tool features . . . 4
2.2 Web server logs . . . 4
2.2.1 Configuring Apache web server logging . . . 4
2.2.2 Looking at the web server log output . . . 4
2.2.3 Web server analysing tool features . . . 5

3 Designing the tools, implementation **6**
3.1 Database log analyser . . . 6
3.1.1 Design . . . 6
3.1.2 Problems and some solutions . . . 7
3.2 Web server log analyser . . . 8
3.2.1 Design . . . 8

4 Analysing logs, discussion **9**
4.1 About database performance tuning . . . 9
4.2 Database log . . . 9
4.2.1 Most time consuming queries . . . 10
4.2.2 Slowest queries . . . 10
4.3 Web server log . . . 14

5 Conclusions **15**

Bibliography **16**

List of Figures

2.1 Database logging format configuration 2
2.2 Database log format example 3
2.3 Apache web server log format 4
2.4 Web server log format example 5

3.1 Class diagram for database log reader 6
3.2 Class diagram for web server log reader 8

4.1 Database query statistics, part 1 10
4.2 Database query statistics, part 2 11
4.3 Database query statistics, part 3 12
4.4 Database query statistics, part 4 13
4.5 Database query statistics, SELECT statement 14

Chapter 1

Introduction

1.1 Background

The Web Mining and Data Analysis course is based on projects where the students regularly meet with a mentoring team. Our choice of project was "User behaviour logging for data warehouse tuning".

In this project, we will be working against Steinbit's[2] ROBACC[1] database, which is a data warehouse powered by an Apache[3] web server with PHP[5] support and a PostgreSQL[6] database.

1.2 Goals and scope

Our goal with this project is to design a set of tools to log user behaviour for tuning the data warehouse and improving the web interface. We will look at both database and web log. An analysis of both these sources may yield information that we can use to suggest improvements on both the data warehouse and web interface.

1.3 Limitations

There are several methods to log database usage, and log files have different formats. We will therefore be focusing on supporting the log format which is currently being used on Steinbit's PostgreSQL database.

An analysing tool for looking at click streams has not been implemented in this project as this requires somewhat complex client side scripting, which would record a user's click stream and submit this to the web server. Another reason why we rejected this option is that the web server log contains about the same amount of information we would get from developing the complex, client side scripting tool.

The number of visitors to the ROBACC search site is limited, and therefor the amount of statistical data, which we will base our conclusions on, are limited as well. Because of this, some assumptions were made.

Chapter 2

Problem descriptions

2.1 Database logs

Our first problem will be developing a tool for analysing a PostgreSQL log file. From a such file we would get information about every query that has been executed on the database. With this information, we can show what is the most frequently executed query, and by this create a basis for database tuning.

PostgreSQL also supplies information about the duration of a query. This information will be used to track down slow queries.

2.1.1 Configuring PostgreSQL logging

The ROBACC database has not been logging any other information than error messages, which is not very useful in our case. Because of this, we do not have any history of the database usage.

By setting the parameters in `postgresql.conf` as shown in figure 2.1, we get a sensible logging format, which our database log reader will be aimed to support.

```
log_connections = true
log_pid         = true
log_statement   = true
log_duration    = true
log_timestamp   = true
```

Figure 2.1: Database logging format configuration

2.1.2 Looking at the database log output

By setting the parameters as listed in figure 2.1, we get a log output similar to what is shown in figure 2.2 on the following page.

We will, in essence, only be looking at `statement` and `duration` entries. We will be handling the four most important SQL statements; `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

```
2004-11-14 16:14:49 [13549] LOG:  statement: SELECT
                                 generator,
                                 round(AVG(pt.htmlerrs)) AS "Errors",
                                 case when AVG(pt.size)=0 then 999999
                                 else 1000*cast(AVG
                                   (pt.htmlerrs) as
                                 float)/cast(AVG(pt.size) as float)
                                   end as errorskb,
                                 COUNT(*) AS "Sites"
                               FROM
                                 pagetest pt,
                                 location l,
                                 provider p,
                                 lasttime lt
                               WHERE

                                 pt.location=l.id AND
                                 pt.provider=p.id AND
                                 pt.testtime=lt.lastcheck
                               GROUP BY generator
                               ORDER BY ROUND(AVG(pt.htmlerrs))
```

Figure 2.2: Database log format example

2.1.3 Database analysing tool features

Our database analysis tool should have the following features:

- Show total number of queries, unique queries, max durations.
- Summarise all queries by the types `SELECT`, `INSERT UPDATE` and `DELETE`.
- Show the most time consuming queries on the database.
- Show the slowest queries.

2.2 Web server logs

Our second problem will be developing a tool for analysing the web server log files. From these files we would get information about what web pages that have been visited and any `GET` or `POST` submissions to the server.

2.2.1 Configuring Apache web server logging

The web server on Steinbit[2] has been logging requests for about a year, so we have some data which will be analysed.

The parameters in `httpd.conf` were set as shown in figure 2.3[1].

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
```

Figure 2.3: Apache web server log format

2.2.2 Looking at the web server log output

By setting the log format as shown in figure 2.3, we get a log output as shown in figure 2.4 on the next page.

There are two different web interfaces to the ROBACC[1] database – one interface which contains drop down boxes for selecting predefined values for given parameters, and another interface which allows you to execute an arbitrary SQL query statements.

The most interesting web interface is the one with the drop down boxes, since this is the web interface which has been accessed most frequently. When selecting the values on the web interface and then pressing “Search”, the values are submitted to a file called `dosearch.php`, where the SQL query statements are built and executed – this is the file reference we are looking for in the web server logs.

The relevant information we get from the web server log is the parameters and values passed to the `dosearch.php` file. This information allows us to compare the SQL query statements generated from `dosearch.php` and the actual queries executed on the database.

[1] This format is also known as the “Combined Log Format” in Apache.

```
62.73.249.252 - - [31/Oct/2004:17:10:16 +0100]
"GET /robacc/dosearch.php?language=english&country=Ireland&\
 category=All&city=All&type=All&sort=country&results=100&\
 urisearch=&doctype=xhtml HTTP/1.1" 200 5723 \
"http://steinbit.agder-ikt.hia.no/robacc/search3j.php" \
"Mozilla/5.0 (X11; U; Linux i686; rv:1.7.3) Gecko/20041003 Firefox/0.10.1"
```

Figure 2.4: Web server log format example

2.2.3 Web server analysing tool features

Our web server analysing tool should have the following features:

- Find all parameters which has been passed to `dosearch.php` and their values.
- Summarise equal values for each parameter.

Chapter 3

Designing the tools, implementation

The Python[4] programming language is used for implementing both the database- and the web server analysing tool.

3.1 Database log analyser

The database log format is relatively fixed, any variation would be a missing time stamp or missing PIDs. However, these variations will be handled by our analysing tool.

3.1.1 Design

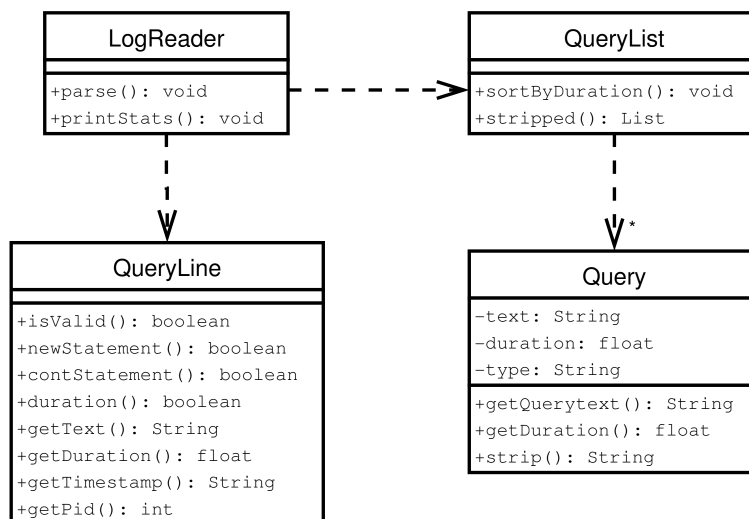

Figure 3.1: Class diagram for database log reader

The design of the database analysing tool is fairly simple. Python[4] has some very nice built-in functions for handling lists and dictionaries, which makes

the implementation a relatively easy task. We will briefly explain some of the main functions of our classes in the following sections.

LogReader

The `LogReader` is the main class of the analysing tool. This class has reference to the log file and is responsible for parsing it. Any output will be initiated from this class.

QueryLine

A `QueryLine` object is created for each line we read from the log file. From this object, we can identify a log entry to be either a *new statement*, a *continuation line*, a *duration line* or an unrecognised line.

Query

A `Query` object will be created whenever we hit a `QueryLine` which is identified as *new statement*. Next, any time stamp or PID values will be added to the object, as well as the duration of the query.

QueryList

The `QueryList` object is wrapper for `List`, which contains every `Query` object that has been created. This object also has a few additional functions for calculating statistical data for the list of query objects.

3.1.2 Problems and some solutions

Although the log file is somewhat fixed, we did encounter some problems.

Regular expressions

Making a computer program interpret a line of text like a person does, is not as easy as one should think. We want to extract information such as time stamps, PID numbers, durations, statements, and so on. In our case, each log line would have different length, making a simple "*substring solution*"[1] useless. The solution for our problem would be *regular expressions*. By using *regular expressions*, we are able to recognise patterns in text with variable length, and performing advanced modifications on them. However, by introducing *regular expressions*, one might also say that we introduce another problem. *Regular expressions* are extremely powerful, but they might be just as error-prone as writing any complex programming code.

[1] A "*substring solution* would be a solution where we would be cutting a string at a known index and length for removing the unwanted characters.

3.2 Web server log analyser

The work of developing a log analyser for the web server is similar to the work of developing the database log analyser. We rely on regular expression when extracting the parameters and values.

3.2.1 Design

The design of the web server log analyser is simpler than our database log analyser.

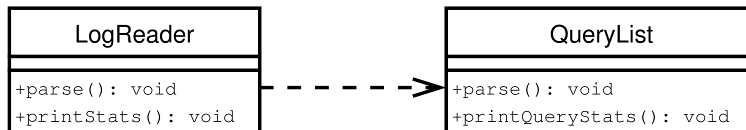

Figure 3.2: Class diagram for web server log reader

LogReader

The `LogReader` is the main class of the analysing tool. This class is responsible for extracting the queries from the log and storing them in the list of queries.

QueryList

The `QueryList` is responsible for calculating statistical data of itself. This class will summarise values for each parameter.

Chapter 4

Analysing logs, discussion

4.1 About database performance tuning

Database tuning is the activity of making a database run as fast as possible. Tuning is difficult because it requires global knowledge of the whole system. From the hardware to the operating system and the transaction subsystem, including the queries. The key features in database tuning are often

- Concurrency control
  How to minimise the lock contention.
- Recovery
  How to manage the writes to the log/dump.
- OS
  How to optimise buffer size, process scheduling.
- Hardware
  How to allocate CPU, RAM, and disk subsystem resources.

We will in this report take a closer look at tuning using index. In an large database there are often a strong need for indexes for making queries faster. But indexes may be faster or slower than scans since some data are updated too often to be used in an index. You can end up running multi-table joins for hours, because the wrong indexes are defined. It is also important not to insert indexes that are never used, because they take up a lot of resources, especially when being updated.

4.2 Database log

By running our database log analyser we get an overview over the slowest queries. the most frequent queries and the most time consuming queries.

As shown in figure 4.1 on the following page there are 3654 *unique* queries. However, by replacing all string values with '' and all integer values with 0 in the query statement, we get a *stripped* query. And when we compare the *unique* and *unique stripped* queries, there are only 160 *unique stripped* queries, which

```
Total number of queries: 11796  Unique: 3654    Unique stripped: 160
---------------------------------------------------------------------
SELECT: 4834    (40.98%)
INSERT: 3449    (29.24%)
DELETE: 12      (0.10%)
UPDATE: 37      (0.31%)
```

Figure 4.1: Database query statistics, part 1

is a significant decrease in number of statements. This allows us to get a better overview when looking at the most time consuming queries executed on the server.

4.2.1 Most time consuming queries

As we see in figure 4.4 on page 13, the most time consuming query, with a large margin down to number two, is `COMMIT WORK`. However, this operation marks the end of a successful implicit or user defined transaction, and it is not to be considered as a candidate for general performance tuning.

The runner up is an `INSERT` statement which inserts data into a table named *rawdata*. This table has no indecies and should not have any.

4.2.2 Slowest queries

If we disregard the slowest query shown in figure 4.2 on the following page, which is a custom query executed by us, the slowest query would be the

```
SELECT id, english FROM language
```

query at 1179ms, which has a running time of nearly twice the query on the following place. This indicates that this query would be a good candidate for tuning. However, when looking at the overview over the most time consuming queries (see figure 4.4 on page 13), the total running time for this query is 4747ms and in the list of how many times a query has been executed, we find that this specific query has been executed 70 times. When this query on one occasion has taken 1179ms to execute, the average for the 69 other times is $\frac{(4747-1179)}{69} = 52$ms, which is an acceptable amount of time. Why this instability in execution time we can not say, but this might be something to look into.

The queries that are the most obvious to look at are the `SELECT` queries with parameters that utilises tables that are not subject to frequent updates. One such query is shown in figure 4.5 on page 14,

where the result of the query depends on the tables *pagetest*, *location*, *provider*, and *lasttime* and their columns *location*, *provider*, *testtime* and *generator*. All of these columns are indexed except *generator*, so our recommendation would be to index that column as well.

The same goes for

```
Slowest queries:
1221.088 ms     select count(*) from rawdataineu;
1179.094 ms     SELECT id, english FROM language
1034.399 ms     select count(id) from rawdata select count(id) from
                rawdata;
798.758 ms      SELECT id, english FROM language
664.539 ms      INSERT INTO lasttime SELECT pt2.location,
                MAX(pt2.testtime) FROM pagetest pt2 GROUP BY
                pt2.location;
615.009 ms      insert into temp SELECT pt2.location,MAX(pt2.testtime)
                FROM pagetest pt2 GROUP BY pt2.location;
586.769 ms      insert into temp SELECT pt2.location,MAX(pt2.testtime)
                FROM pagetest pt2 GROUP BY pt2.location;
570.066 ms      INSERT INTO lasttime SELECT pt2.location,
                MAX(pt2.testtime) FROM pagetest pt2 GROUP BY
                pt2.location;
566.556 ms      insert into temp SELECT pt2.location,MAX(pt2.testtime)
                FROM pagetest pt2 GROUP BY pt2.location;
532.842 ms      SELECT generator, round(AVG(pt.htmlerrs)) AS
                "Errors", case when AVG(pt.size)=0 then 999999 else 1
                000*cast(AVG (pt.htmlerrs) as float)/cast(AVG(pt.size)
                as float) end as errorskb, COUNT(*) AS ‘‘Sites’’ FROM
                pagetest pt, location l, provider p, lasttime lt WHERE
                pt.location=l.id AND pt.provider=p.id AND
                pt.testtime=lt.lastcheck GROUP BY generator ORDER BY
                ROUND(AVG(pt.htmlerrs))
```

Figure 4.2: Database query statistics, part 2

```
Most frequent queries:
3334    select version()
3334    COMMIT WORK
3334    select version()
3334    COMMIT WORK
2567    insert into rawdata (uri, pageuri, htmlerrs, htmlparseok,
        images, altimages, styleimages, server, contenttype, date,
        lastmodified, contentlength, setcookie, etag, connection,
        cachcontrol, contentlanguage, contentencoding, doctag,
        generator)
        values('','',0,0,0,0,0,'','','','','','','','','','','','','');
749     insert into rawdataineu (uri, pageuri, htmlerrs, htmlparseok,
        images,altimages,styleimages, server, contenttype,
        date, lastmodified, contentlength,setcookie,etag,connection,cachcontrol,
        contentlanguage, contentencoding,doctag) values('',
        '',0,0,0,0,0,'','','','','','','','','','','','');
504     SELECT DISTINCT city FROM location WHERE country='' and city
        IS NOT NULL ORDER BY city
72      SELECT DISTINCT type FROM provider WHERE category='' AND type
        IS NOT NULL ORDER BY type
70      SELECT id, english FROM language
68      INSERT INTO log (ip, host, referer, agent, self, query, time)
        VALUES ('','','','','','','')
```

Figure 4.3: Database query statistics, part 3

```
Most timeconsuming queries:
220496.564 ms    (3334) COMMIT WORK
8683.194 ms      (2567) insert into rawdata (uri, pageuri, htmlerrs,
                        htmlparseok, images,altimages,styleimages,
                        server, contenttype, date,lastmodified,
                        contentlength,setcookie,etag,connection,cachcontrol,
                        contentlanguage, contentencoding,doctag,
                        generator)
                        values('','',0,0,0,0,0,'','','','','','',
                        '','','','','','','');
5535.636 ms      (3334) select version()
4747.46 ms       (70)   SELECT id, english FROM language
4319.749 ms      (68)   INSERT INTO log (ip, host, referer, agent,
                        self, query, time) VALUES ('','','','','','','')
4094.658 ms      (25)   insert into temp SELECT
                        pt2.location,MAX(pt2.testtime) FROM pagetest
                        pt2 GROUP BY pt2.location;
2442.993 ms      (749)  insert into rawdataineu (uri, pageuri,
                        htmlerrs, htmlparseok,
                        images,altimages,styleimages,
                        server, contenttype, date,lastmodified,
                        contentlength,setcookie,etag,connection,cachcontrol,
                        contentlanguage,contentencoding,doctag) values('',
                        '',0,0,0,0,0,'','','','','','','','','','','','');
2325.918 ms      (25)   create temporary table temp(id int, lastcheck
                        timestamp);
2063.276 ms      (9)    SELECT l.uri, pt.htmlerrs,case when size=0
                        then 999999 else 1000*cast(htmlerrs as float)/
                        cast(size as float) end as errorskb,
                        l.country, p.category, p.type, pt.generator,
                        date_trunc('',pt.testtime) AS "testtime" FROM
                        provider p, location l, pagetest pt, temp t
                        WHERE pt.location=l.id AND pt.provider=p.id
                        AND t.id=pt.location AND
                        t.lastcheck=pt.testtime
                        ORDER BY uri ASC LIMIT 100 OFFSET 0;
1882.456 ms      (8)    INSERT INTO lasttime SELECT pt2.location,
                        MAX(pt2.testtime) FROM pagetest pt2 GROUP BY
                        pt2.location;
1519.688 ms      (5)    SELECT generator, round(AVG(pt.htmlerrs)) AS
                        "Errors", case when AVG(pt.size)=0 then 999999
                        else 1000*cast(AVG (pt.htmlerrs) as float)/
                        cast(AVG(pt.size) as float) end as errorskb,
                        COUNT(*) AS "Sites" FROM pagetest pt, location
                        l, provider p, lasttime lt WHERE
                        pt.location=l.id AND pt.provider=p.id AND
                        pt.testtime=lt.lastcheck GROUP BY generator
                        ORDER BY ROUND(AVG(pt.htmlerrs))
```


Figure 4.4: Database query statistics, part 4

```
1519.688 ms      (5)    SELECT generator, round(AVG(pt.htmlerrs)) AS
                        "Errors", case when AVG(pt.size)=0 then 999999
                        else 1000*cast(AVG (pt.htmlerrs) as float)/
                        cast(AVG(pt.size) as float) end as errorskb,
                        COUNT(*) AS "Sites" FROM pagetest pt, location
                        l, provider p, lasttime lt WHERE
                        pt.location=l.id AND pt.provider=p.id AND
                        pt.testtime=lt.lastcheck GROUP BY generator
                        ORDER BY ROUND(AVG(pt.htmlerrs))
```

Figure 4.5: Database query statistics, SELECT statement

```
SELECT DISTINCT city FROM location WHERE country='' and city
  IS NOT NULL ORDER BY city
```

and

```
SELECT DISTINCT type FROM provider WHERE category='' AND type
  IS NOT NULL ORDER BY type
```

where it could be wise to index *country*, *city*, *category* and *type* in the tables *location* and *provider* if they should increase in number of rows. At the moment they are so small that indexing them will not improve the performance.

4.3 Web server log

From the web server log we can see confirmations to what we already know from the database log analysis. Almost every one of the values is set to their standard choice – typically "all" $>$ 80% of the time. The two exceptions are "Category" and "Country". This indicates that the tables containing these two columns should be optimised if their tables should grow large. When focusing on the design of the web interface these two drop down boxes ("Category" and "Country") should have an eminent position which makes it easy for the user to spot them, typically on the top and to the left. This is their current placing so no change is necessary here.

Chapter 5

Conclusions

Through this project we have designed two tools, one for analysing a database log, and another for looking at the web log. This is for trying to find as much information as possible about the users behaviours. With the data that has been available to us in this project there is not possible to draw any large conclusion.

The database log

By analysing the database log we have found areas that we think can be improved. This mainly deals with indexing different columns in tables that are not subject to frequent updates. We have found that the column *generator* in table *pagetest* and possibly also the columns *country*, *city*, *category* and *type* in the tables *location* and *provider* should be indexed should their number of rows increase.

The Web log

It is difficult to draw conclusions on basis of just the analysis of the web log. But what we have found here supports some of our conclusions based on database log analysis.

Bibliography

[1] Agder University College. Robot assessing web accessibility. http://osys.grm.hia.no/fou/ROBACC[1]/ROBACC[1].html.

[2] Agder University College. Steinbit. http://steinbit.agder-ikt.hia.no.

[3] Apache Software Foundation. Apache http server project. http://httpd.apache.org.

[4] Python Software Foundation. The python programming language. http://www.python.org/.

[5] The PHP Group. Php hypertext protocol. http://www.php.net.

[6] The PostgreSQL Global Development Group. Postgresql database management system. http://www.postgresql.org/.