Data Warehouse Query Log Analysis using MapReduce

by
Willy Lai

Supervised by
Prof. Dr. Donald Kossmann
Dr. Maxim Grinev

Feb 2011 – Aug 2011
Abstract

Current auditing techniques on the data warehouse of Credit Suisse, a large Swiss multinational financial service provider, are either not flexible enough to answer ad-hoc queries or very costly to maintain. This master thesis proposes a new scalable solution in analyzing the raw query logs based on distributed computing using Hadoop and Pig. Our approach acts on raw data, hence allows total flexibility in formulating ad-hoc queries. In addition, it does not rely on maintaining a dedicated data warehouse, but can run on any cluster infrastructure on a utility basis, thus being very cost effective.

Our solution has been implemented and tested on synthetic test data, as well as on a set of production data at Credit Suisse. An estimation has shown that we are able to process 93% of all the query log data, even in the presence of malformed data. In experiments, our implementation has proven to be highly scalable when processing synthetic query logs for a generic use case. Its performance however is in need of improvement, but might be compensated by the high scalability in the meantime.
Acknowledgements

I want to express my utmost gratitude to my thesis supervisors Prof. Dr. Donald Kössmann and Dr. Maxim Grinev for giving me the opportunity to be part of their research, providing me with the topic and their guidance throughout the thesis work. I am also very grateful for the collaboration and support of Dr. Kurt Stockinger at Credit Suisse, who also enabled me to perform some tests on production data.

In addition, I greatly appreciated the input I received from Dr. Maria Grineva (ETH Zurich), Johannes Buck (Credit Suisse) and Georg Polzer (ETH Zurich), who regularly attended our meetings. I also want to thank my sister Peychyn Lai, who proof-read my writings and helped me in enhancing the clarity of this document.
Contents

1 Introduction ................................. 4
   1.1 Motivation .................................. 4
   1.2 Related Work ................................ 5
   1.3 Outline of the Thesis ......................... 5

2 Auditing ........................................ 7
   2.1 Auditing Use Cases defined at Credit Suisse ................. 7

3 Contextual Information about the Working Environment and available Data ......... 9
   3.1 Background Information ....................... 9
   3.2 Audit Log Files ................................ 10
   3.3 Schema Snapshots ............................... 10
   3.4 Oracle Database Concepts ....................... 11
      3.4.1 Schema ..................................... 11
      3.4.2 Synonym .................................... 12

4 Setting the Scope ............................. 13
   4.1 Use Cases revisited ............................ 13
      4.1.1 1st Case Regular Compliance Monitoring (SOX, internal controls) .... 13
      4.1.2 2nd Case - IT Investigations by CERT ............... 13
      4.1.3 3rd Case - Detect Suspicious Behavior ............... 14
      4.1.4 4th Case - Economic Analysis .................... 14
   4.2 Defining the Use Case we focus on .................. 14

5 Design and Implementation of proposed Solution .................................. 15
   5.1 Technologies .................................. 15
      5.1.1 MapReduce ................................... 15
      5.1.2 Apache Hadoop ................................ 16
      5.1.3 Apache Pig .................................. 16
   5.2 Architecture ................................... 16
      5.2.1 Overview ................................... 16
   5.3 Audit Log File Processing ...................... 18
   5.4 Schema Snapshot Processing .................... 18
   5.5 Schema Matching ............................... 19
   5.6 Implementation Details ....................... 20
      5.6.1 XML Parsing ................................ 20
Chapter 1

Introduction

1.1 Motivation

Recent incidents in the financial industry, most notably the global financial crisis happening in early 2007, have led to new auditing requirements and stricter regulations on equity capital as well as internal assessment processes\(^1\) for financial service providers. Credit Suisse\(^2\), as one of the two largest Swiss financial service providers, is strongly affected by these regulations and is continually investing in improving their internal audits. Credit Suisse is currently storing raw audit data of their data warehouse that need to be analyzed for auditing purposes, whose specific use case questions are not known in advance. Clearly, manual analysis on the complete raw data is infeasible. Auditing techniques that are based on limited samples [4] are reasonable when it is sufficient to get results that are correct with high probability (e.g. based on our audit sample, this table is accessed 50-75 times per hour with probability of 95%). Their challenge is to find an appropriate sampling method in order to get a representative sub set of the entire data set. These techniques however, are not suited for use cases that require to spot rare occurrences in a very large data set (e.g. who has accessed table X on a Sunday between 2am and 3am?), as without accessing the complete log data, they are very likely to miss the relevant log entries. Computer aided auditing techniques are required in these cases. Concepts thereof exist that extract and transform the complete set of raw data and loads structured results into a separate data warehouse, allowing auditors to formulate their questions in SQL. This approach has two major drawbacks.

- Maintaining a dedicated data warehouse is very costly.

- Generally the raw data is preprocessed and loaded into a specific structure, restricting the auditors flexibility in posing ad-hoc queries.

We propose an auditing enabling processing technique that directly acts on the raw data and is based on distributed processing. Our solution comes with the following benefits.

\(^1\)http://en.wikipedia.org/wiki/Sarbanes-Oxley_Act#Sarbanes-Oxley_Act\_Section_404:_Assessment_of_internal_control (accessed 14-July-2011)

\(^2\)www.credit-suisse.com
• It is fully automatic (with some exceptional cases elaborated in chapter 6), thus saving costs induced from manual work.

• It acts on raw data, providing the full flexibility in formulating ad-hoc queries that may be even outside the scope of auditing (e.g. for business intelligence, interested in resources that are never used).

• It is based on open source software, highly scalable and can be run on commodity machines; thus is very cost effective.

1.2 Related Work

Initial research on data warehouse query log analysis using Hadoop\(^3\) and Pig\(^4\) has been done in ETH Zurich in Systems Group\(^5\) [1]. Their work was exclusively focusing on finding users that have accessed a certain object in the data warehouse. The processing approach was limited on query logs only and exploiting XQuery and regular expression. We adopt the original idea, generalize the use case and develop a more sophisticated process, including SQL analysis, from scratch.

1.3 Outline of the Thesis

The remainder of the thesis is structured as follows.

• Chapter 2 gives an overview about auditing in general and the use cases defined at Credit Suisse in particular.

• Chapter 3 provides contextual information about the working environment and the available data.

• Chapter 4 revisits the use cases defined at Credit Suisse and sets the scope of our work.

• Chapter 5 presents the design and implementation of our solution.

• Chapter 6 elaborates the current limitations of our solution.

• Chapter 7 describes the test setup and presents the results from various experiments.

• Chapter 8 synthesizes our findings in a conclusion and discusses future work.


\(^4\)http://pig.apache.org/ (accessed 12-July-2011)

\(^5\)http://www.systems.ethz.ch/ (accessed 24-July-2011)
Chapter 2

Auditing

An audit in the general sense is an evaluation process to ensure that certain requirements are met. Auditing in our context involves certain use cases regarding information access at Credit Suisse. These use cases are raised by auditing regulations, or by an internal department, interested in optimizing resource usage.

2.1 Auditing Use Cases defined at Credit Suisse

This section presents the use cases that were defined at Credit Suisse for data warehouse log analysis.

- **1st Case Regular Compliance Monitoring (SOX, internal controls)**
  Compare approved access control list (ACL) and user profiles with actual access logs

- **2nd Case - IT Investigations by CERT**
  Scenario verification, i.e. check if an individual or a group has performed certain actions (e.g. access to data, modification/deletion of data, execution of transactions). Evidence gathering, i.e. collect and secure logs to prove misconduct, fraud, etc. as supporting material in legal cases

- **3rd Case - Detect Suspicious Behavior**
  Set a real-time alert if certain conditions are met in a similar way to an intrusion detection system (IDS). The following parameter might be considered to define an alarm trigger: amount of data accessed (e.g. number of client records), frequency of access, weekday/time of access, critical functions (e.g. data export)

- **4th Case - Economic Analysis**
  Usage of resources: Which data has been accessed most vs. which data has never been accessed? Consumers: Who did access which data, how often, and for which purposes (read, modify)?

\[^1\text{http://de.wikipedia.org/wiki/Audit (accessed 14-July-2011)}\]
Chapter 3

Contextual Information about the Working Environment and available Data

To enable auditing, information about past actions need to be persisted. These past actions are captured in so called *audit trails* that log each action as a separate entry. Since in our case an action is equivalent to an issued SQL statement, we will use the term *audit trail* and *query log* interchangeably in the remainder of this document. This section provides information about the working environment and the available data for our analysis.

3.1 Background Information

The audit trail stems from a central data warehouse (DWH) based on Oracle Database 11g\(^1\) at Credit Suisse. It is deployed on multiple testing levels, one production level and is backed up by a complete replication outside Zurich. At the time of the master thesis, the total data volume in the DWH was in the ballpark of 800 terabytes, generating audit logs of estimated 6 terabytes every month.

Oracle Database provides the ability to store audit information in a dedicated table `SYS.AUD$` that can be queried with SQL. However, experience has shown that this method introduces noticeable resource overhead\([6]\) and might therefore not scale well with increasing data sizes. Internal tests at Credit Suisse have indeed showed performance decrease when logging into a dedicated system table.

With the release of Oracle Database 10g\(^2\), the possibility to store the audit trails in XML format has been introduced\(^2\). This method has several advan-


\(^2\)http://www.oracle-base.com/articles/10g/Auditing_10gR2.php (accessed 21-July-2011)
tages [8] over storing the audit information in SYS.AUD$.

- XML can be processed by tools that do not depend on the database.
- The external XML files can be protected using mechanisms that are completely independent of the privileges granted by the database roles, preventing even database administrators to view or modify the audit trails.
- The availability of the audit trails is decoupled from the database uptime, enabling analysis to be run, even when the database is not available.

Credit Suisse currently exploits this feature and stores raw XML audit log files. Figure 3.1 illustrates the current situation of the available data.

![Figure 3.1: Overview of the current architecture](image)

### 3.2 Audit Log Files

The DWH maintains the audit trail in real-time and generates audit log files in XML every now and then. A sample audit log file can be seen in appendix A.1. It contains multiple audit records, holding information about a single database action on the DWH. These information include the timestamp on which the action has been logged, the user who initiated the action, the terminal from which the action has been sent, the issued SQL statement including potential SQL bindings of the action and further properties.

### 3.3 Schema Snapshots

Schema snapshots are generated once a day and contain metadata about the whole schema of the DWH. One schema consists of four XML files, containing
data about following objects. We use the term object, schema object or database object when referring to any type of data structure in the DWH (e.g. tables, views, indexes, sequences, synonyms, ...).\(^3\)

- **Tables**
  This file contains information of all the tables in the schema, including owner, table name, tablespace name, cluster name, status, data block specific settings and other information (Appendix A.2.1).

- **Views**
  This file contains information of all the views in the schema, including owner, view name, the underlying SQL query and other information (Appendix A.2.2).

- **Dependencies**
  This file contains information of all schema object dependencies. Views, as an example, can reference other tables (or views) in their definition. For each of these references a dependency record is stored in this file (Appendix A.2.3).

- **Synonyms**
  This file contains information of all synonyms defined in the schema. A synonym is an alias for any table, view or other objects defined in the schema (Section 3.4.2) (Appendix A.2.4).

We will refer to specific files of a schema snapshot by using schema table file, schema view file, schema dependency files, schema synonym file in the remaining sections of this document.

### 3.4 Oracle Database Concepts

Oracle Database has concepts that are important to be aware of. This section introduces these concepts on a high level that is sufficient for understanding the processing steps that will be introduced in chapter 5.6.

#### 3.4.1 Schema

The concept of schema in Oracle Database is slightly different from the general definition of schema in relational database course books, in which a schema is defined as the information defining the complete database structure (e.g. tables, columns, relationships, views, indexes, ...).

In Oracle Database, schema has the following definition:

"A named collection of database objects, including logical structures such as tables and indexes. A schema has the name of the database user who owns it."\(^4\)

\(^3\)http://download.oracle.com/docs/cd/E11882_01/server.112/e16508/glossary.htm#CHDICIIC (accessed 7-July-2011)

\(^4\)http://download.oracle.com/docs/cd/E11882_01/server.112/e16508/glossary.htm#CHDLCICIE (accessed 22-July-2011)
Every user owns a single schema and has access to its local schema, but can be granted access rights to other user's schema. Schema objects owned by different users do not necessarily have unique names. When referring to an object, we therefore need to resolve that reference to the right user schema. In order to avoid any notational confusion, we use the term user schema or owner throughout the document when referring to Oracle Database's definition of a schema.

3.4.2 Synonym

In Oracle Database, synonyms are aliases for database object names. They introduce an additional layer between SQL and the underlying database and improve maintainability by masking the name and owner of an object of interest. This, besides simplifying the SQL statements allows for changes of the underlying objects for specific users, without having to rewrite every SQL statement.
Chapter 4

Setting the Scope

This chapter revisits the use cases defined at Credit Suisse (Section 2.1) and sets the scope for our work.

4.1 Use Cases revisited

Before going into the design of our proposed solution, we take a top-down perspective and, for each use case defined in section 2.1, investigate the following points.

- What data and information do we need, in order to solve this use case? Do we have that data?
- Assuming we have all data we need, how does the processing look like? Can we process the data offline?

4.1.1 1st Case Regular Compliance Monitoring (SOX, internal controls)

In order to answer this use case, we need the audit log files of the DWH and snapshots of the access control list (ACL), which contains the role assignments for every user and all the permissions granted per role. Answering this use case involves scanning through all the DWH audit log entries, extracting the user and SQL statement issued and checking whether the accessed objects in the SQL statement conforms to the permissions granted for all the roles of the corresponding user.

4.1.2 2nd Case - IT Investigations by CERT

In order to answer this use case, we need the audit log files of the DWH and the specific actions of interest. Answering this use case for a single action, involves scanning through all the DWH audit log entries while maintaining a certain structure that eventually allows for verification whether the specified action has taken place or not.
4.1.3 3rd Case - Detect Suspicious Behavior

In order to answer this use case, we need real-time access to the actions happening in the DWH, the definition of all alerts and a rule set that specifies the triggers of those alerts. Answering this use case requires real-time monitoring integrated into the DWH environment of Credit Suisse, as well as interfacing to the systems that should be notified in case suspicious behavior has been detected.

4.1.4 4th Case - Economic Analysis

In order to answer this use case, we need the audit log files of the DWH and its schema definition. Answering this use case involves scanning through all the DWH audit log entries, and counting the numbers of accesses for every object of interest.

4.2 Defining the Use Case we focus on

We have worked out that the 1st, 2nd and 4th use case can be processed offline, while the 3rd use case can only be processed online. Online processing has fundamentally different functional and non-functional requirements that are not in the scope of our solution. Note, that the remaining use cases are very generic and include a wide range of questions.

- Who has modified this specific column in the last week?
- What tables did user $u$ access today from 11am until 6pm?
- Who has deleted view $v$?
- When was the last time table $T$ was read?

At the time of this master thesis the specific questions that would need to be covered by each of these use cases were not defined and no data other than a few audit log files and schema snapshots were available. Hence, we focused on the more general use case containing the following questions.

- **Question 1**: For each table: What users have accessed this table at what time?
- **Question 2**: For each column: What users have accessed this column at what time?
- **Question 3**: For each user: What tables and columns were accessed by this users at what time?

We are especially interested in read accesses, although our approach supports write accesses as well.
Chapter 5

Design and Implementation of proposed Solution

5.1 Technologies

We have decided to base our solution on Apache Hadoop (Section 5.1.2) and Pig (Section 5.1.3). Their processing model fits well to our problem, as our tasks for audit log file analysis basically consist of reading, transforming and grouping of data (Section 5.3), which is essentially what MapReduce provides. Pig allows us to abstract from the MapReduce underlyings and provides us with a high level language to formulate our analysis programs for ad-hoc queries. Both technologies have been proven to be stable and highly scalable in many projects.\(^1\)

5.1.1 MapReduce

MapReduce \(^2\) is a programming model developed at Google for processing very large data sets. Its idea originates from the map and reduce functions in functional programming. Users of the MapReduce programming model need to express their input data as key/value pairs and model their processing steps as map and reduce functions. A map function takes as input a key/value pair and outputs zero or more intermediate key/value pairs. All intermediate pairs are then grouped by their key and passed to a reduce function. The reduce function takes as input a pair consisting of a key and a list of the values having that key and outputs a list of result values. If adhering to this model, the user does not need to consider how to parallelize the processing, distribute the data or handle failures, as the implementation of the MapReduce framework will take care of that.

\(^1\)http://wiki.apache.org/hadoop/PoweredBy (accessed 12-July-2011)
\(^2\)http://wiki.apache.org/pig/PoweredBy (accessed 12-July-2011)
5.1.2 Apache Hadoop

Apache Hadoop\(^3\) is an open source Java library that enables the development of distributed and scalable applications inspired by the Google File System [3] and MapReduce (Section 5.1.1). It provides a reliable shared storage and an implementation of the MapReduce framework.

5.1.3 Apache Pig

Apache Pig\(^4\) is a platform for analyzing very large data sets. It comes with its own declarative language called Pig Latin [5] that provides a means to express data analysis programs in a procedural fashion as data flow sequences. Apache Pig compiles Pig Latin into a sequence MapReduce programs that can be run on Apache Hadoop.

5.2 Architecture

This section presents our solution architecture (Figure 5.1) for solving our use case defined in section 4.2. Our approach performs the DWH audit trail analysis by processing the available DWH audit logs files and schema snapshots. The implementation details of the components will be elaborated in section 5.6.

5.2.1 Overview

Our solution is based on distributed processing; hence major parts of the computation are done within a cluster of computing nodes. In particular, we have a cluster of nodes that form our Hadoop Distributed File System (HDFS) [7]. HDFS is a scalable distributed storage, designed to store very large data sets reliably by replicating data chunks over different machines. An HDFS cluster consists of many DataNodes, storing the data and a single NameNode, which maintains metadata (e.g. filesystem index) about the HDFS instance. It is contacted by a DataNode prior to any read or write, and therefore a single point of failure. In addition, we have a MapReduce processing cluster, consisting of multiple TaskTrackers executing map and reduce tasks and a single JobTracker scheduling the MapReduce jobs to be run. These nodes stream data from the HDFS for processing.

Note, that in general practice the HDFS nodes and MapReduce nodes are physically on the same machine, which is reasonable as Hadoop aims to assign processing tasks to local data as much as possible. The separation done in figure 5.1 is for illustration purpose only.

For the analysis we preprocess the schema snapshots (Section 5.4) and copy the resulting output and all audit log files to HDFS. Thereafter, we execute a Pig Latin script, describing our processing flow, on the JobTracker running Pig, that compiles the script into multiple MapReduce jobs and schedules these on the processing cluster. The result files are then outputted to HDFS and can be copied locally.

---


\(^4\)http://pig.apache.org/ (accessed 12-July-2011)
Figure 5.1: Overview of the solution architecture
5.3 Audit Log File Processing

We treat the processing of the audit log files, without considering the schema snapshots, as an embarrassingly parallel problem\(^5\). Treating every single audit record as an independent entity, allows us to distribute the audit trail processing in a very simple way, when using our technologies. From a MapReduce perspective, every audit record can be seen as a value, whose key can be arbitrarily defined (e.g. its timestamp). The map task initially extracts the relevant information from the XML audit record, which in our case are the following.

- `<DB_User>` the user, who issued the SQL statement
- `<Extended_Timestamp>` the timestamp (with microsecond precision), at which this SQL statement has been issued
- `<Sql_Text>` the issued SQL statement

Subsequently, the SQL contained in `<Sql_Text>` is parsed and the resulting abstract syntax tree is analyzed. The SQL analysis is described in detail in section 5.6.3. In a second step, the map task performs the matching of the object names, extracted in the SQL analysis, to the objects defined in the corresponding schema snapshot (Section 5.5). This is necessary, since we are interested in the base tables and base columns that have been accessed. In case an object name matches to a view in the schema, further resolution needs to be done. The map phase might produce a tuple for every accessed object including the user and the timestamp. This would result in \(n + m\) tuples per audit record, where \(n\) is the number tables and \(m\) the number of columns. The reduce phase would then group and sort all tuples in three individual runs. Once by table, once by column and once by user, generating the three result list we are interested in (Section 5.7).

The beauty of using Pig is that we do not have to care about defining the map function and the reduce function. Once we have learned Pig Latin, we can formulate our use case questions as processing steps in a procedural way, but still make use of some declarative constructs, which is very convenient and natural in our context. Section 5.6.4 will present our Pig Latin script.

5.4 Schema Snapshot Processing

The processing of the schema snapshots composes of the processing of every schema file.

- **Schema table file:** Since we are only interested in knowing whether an object name refers to a schema table or not, processing the schema table files is fairly simple. As their structure is flat, we only need to maintain a set of table names including their owner.

- **Schema view file:** Its structure can conceptually be thought of as a network. A view should not have a reference to itself, but can have references to any other view or table. Processing involves unfolding the views, with an unknown depth of references.

• **Schema dependency file:** This file can be ignored in our case, as we compute the view dependencies from the SQL statement. This approach has also turned out to be faster than processing the dependency files. Furthermore, the latter provides us only with the dependencies on a table level. In order to resolve columns to columns of another view, we would require analyzing the SQL statement anyway.

• **Schema synonym file:** This file is essentially a mapping of names that needs to be maintained in a structure for efficient lookup.

Using our technologies, distribution of the schema snapshot processing cannot be done in a straightforward way. Since the schema view file is structured as a network, it is not possible to split it up into independent chunks that can be processed in isolation. We would require multiple MapReduce iterations, each unfolding the views for one level (Section 8.2.3). In the current state however, the schema snapshots are generated once a day for the purpose of this analysis and the data size for one complete schema snapshot is relatively small (∼50 MB). On top of that the schema snapshot files are malformed in various ways (Section 6.2), forcing us to pre-process them before being able to parse them. Thus, we have decided to include the schema view file unfolding in our preprocessor. The preprocessor is run sequentially on a single machine for the time being and persists intermediate results for schema matching (Section 5.5). We have planned to integrate the preprocessor in the application that generates the daily schema snapshots, avoiding additional computation time when performing a DWH audit trail analysis.

### 5.5 Schema Matching

Schema matching refers to the task of mapping the object names, particularly the table/view names extracted during SQL analysis, to the corresponding objects in the schema snapshot. The main reason for executing this step is to increase the precision of our analysis. Synonyms mask their underlying table/view while views mask their underlying tables and columns. As a consequence, analyzing solely audit log files would return the synonym names and view names and their projected aliases, missing the original underlying objects. Schema matching needs to be done in audit log file processing, as well as in schema snapshot processing, when performing SQL analysis. For audit log file processing we need to find the corresponding schema snapshot for every audit record. The newest snapshot that precedes the timestamp of the audit log is chosen in our implementation. Having snapshots in a day granularity, results in some limitations that will be described in section 6.1.

In order to match an object name in an SQL statement contained in an audit record or view to the corresponding object (table, view or synonym) in the schema snapshot, we need to be aware of the user that has issued the SQL statement or the owner of the view respectively. Generally, we never know whether an object name belongs to a table, view or another type of object. We therefore need to check against the schema table file and schema view file when performing schema matching. In addition, we need to consider the schema
synonyms. Following order of precedence is implemented.

1. Match to a table/view owned by the user
2. Match to a synonym owned by the user
3. Match to a public synonym

In case the object name matches to a synonym, we will repeat the schema matching with the newly obtained name, until we find a match to a table/view or fail in finding any match. If we find a match to a synonym, we update the corresponding object name to the referenced object name. If we find a match to a view, we complete the lists of accessed tables and columns with the objects found in the view. An example will be shown in section 5.8.

5.6 Implementation Details

This section describes the implementation details of our proposed solution and especially elaborates SQL analysis.

5.6.1 XML Parsing

The first step in our data processing chain is the syntactic analysis of the XML structure of the schema snapshots and the audit log files. Since no XML tree manipulation is required and only a sub set of the schema snapshot data is relevant for our use case, we have defined our own object model (Figure 5.2) and use a SAX parser\(^7\) to construct these objects.

\[\text{Figure 5.2: Custom object model for XML content}\]

5.6.2 SQL Parsing

The second step in schema snapshot processing, as well as in audit log file processing involves SQL parsing. We have evaluated three different SQL parsers by running them on a set of 30 different SQL statements, each containing a special clause or construct that we expect to encounter in the production data. The DWH is based on Oracle 11g, so we made sure that we also covered some Oracle Database specific SQL grammar. Those three parsers have been evaluated.

\(^6\)http://www.db-oracle.com/concepts/synonyms.htm (accessed 14-July-2011)

\(^7\)Simple API for XML, is a XML parser style that processes XML in a sequential, event-based manner

20
• **Zql:** Zql\(^8\) is an open source SQL parser licensed under GNU GPLv3, written and built with JavaCC. It started as a proprietary SQL parser and was first released back in March 10, 1988, after becoming open source in 2010. The parser has not been actively developed in the last few years.

• **JSqlParser (JSql):** JSqlParser\(^9\) is an open source parser licensed under GNU LGPL, developed by Leonardo Francalanci and is written and built with JavaCC.

• **General SQL Parser (GSP):** General SQL Parser\(^10\) is a proprietary SQL parser. It comes with a wide range of features for parsing, formatting, modification and analysis.

The results (Table 5.1) are quite astonishing and made us aware of the difficulty of writing a parser for a language, whose specification is not completely public, as well as subject to changes. In any case, we clearly see from the results that the General SQL Parser is far superior to its open source competitors. So we decided to purchase a license of the parser and used it throughout the master thesis.

### 5.6.3 SQL Analysis

After successfully parsing the SQL statements we get an abstract syntax tree that needs to be analyzed. According to our initial questions in section 4.2, we are mostly interested in the following.

- What tables are accessed in this SQL statement?
- What columns are accessed in this SQL statement?

As we will see in the analysis, these two questions raise new questions that need to be answered when dealing with more complex SQL statements.

- What columns are projected in the result set of this SQL statement?
- Does the SQL statement select `star` (*) from certain tables, if yes, which tables?
- Are there any sub selects in the SQL statement?

Our SQL parser features the extraction of selected columns and tables, however at the time of our master thesis this feature contained bugs and did not consider any star projections\(^11\). Thus, we used its parsing capabilities only, and built our own procedures on top of it.

Our processing flow can be divided into an *extraction phase* and a *resolution phase*. The extraction phase gets all the relevant object names from our SQL statement. The resolution phase is only needed for columns and identifies

---


\(^11\) We have opened several tickets in the bug tracking system regarding the described issues and recent updates seems to have fixed those. [http://www.driver.com/blog/get-columns-and-tables-in-sql-script/](http://www.driver.com/blog/get-columns-and-tables-in-sql-script/) (accessed 28-July-2011)
<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>ZQL</th>
<th>jSql</th>
<th>GSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Select (SELECT, FROM, WHERE) statement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement including nested Select statement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with JOIN ON clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with JOIN USING clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with NATURAL JOIN clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with CROSS JOIN clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with FULL OUTER JOIN clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with LEFT OUTER JOIN clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with RIGHT OUTER JOIN clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with OUTER JOIN (+) operator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with ORDER BY clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with GROUP BY, HAVING clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with simple CASE expression</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with searched CASE expression</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with IN expression clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with EXISTS expression clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with hierarchical query clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with common table expression</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with interval expression</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with multiset operator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with DECODE function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with CAST function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Select statement with TRIM function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Insert statement with values</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Insert statement with sub query</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Update statement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Delete statement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Create table statement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Create table statement with sub query</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Truncate statement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.1: Test results of our SQL parser evaluation. Each row corresponds to one SQL statement, containing a short description and a check mark for each SQL parser that could parse it.
the tables, to which they belong. We implemented a visitor,\(^\text{12}\), that performs
the extraction phase by traversing the abstract syntax tree and storing the
object names in an extraction result object. We developed an algorithm that
performs the resolution phase. It reads the extraction result object obtained
by the visitor and produces a resolution result object that can be refined when
matching resolved object names to the schema.

Developing the Algorithm

Before presenting the data structures and the resolution algorithm, we go
through several examples that show the subtleties one must be aware of. The
first part contains the examples that concentrate on \texttt{SELECT} statements only.
The second part will give an overview of how we handle other types of state-
ments. We will develop the resolution algorithm in the course of illustrating
the examples. We aim to be data structure independent and therefore use the
generic term of a \textit{collection} when referring to any data structure. The schema
snapshots are not taken into account for the sake of simplicity. A complete ex-
ample including schema matching is illustrated in section 5.8. We will base our
examples on the \textit{MySQL Employees Sample Database Schema}\(^\text{13}\), illustrated in
5.3.

Note, that the SQL statements presented in the course of this chapter are
for illustration purpose and do not follow any SQL guidelines or best practices.

Let us start with the following simple SQL statement (Listing 5.1).

```sql
SELECT
  e.first_name, e.last_name
FROM
  employees e, titles t
WHERE
  e.emp_no = t.emp_no AND
  t.title = 'Manager'
```

Listing 5.1: Simple SQL statement

We extract the following tables and columns into two separate collections. A table collection and a column collection.

```plaintext
employees e, titles t
e.emp_no, 
e.first_name, 
e.last_name, 
t.emp_no, 
t.title
```

We will then resolve the columns by iterating through the column collection and for every column get the table in the table collection that matches the prefix of the column.

```plaintext
employees.emp_no, 
employees.first_name, 
employees.last_name, 
titles.emp_no, 
titles.title
```

In the previous SQL statement the table aliases that were prefixed to the columns, enabled us to assign the corresponding table to the columns clearly. Unfortunately, this is not always the case, as SQL statements can also contain unprefixed columns that can be resolved by the database, since it knows the table definitions. At the current state of the project, table definitions were not included in the schema snapshots. We therefore had to deal with certain ambiguity when resolving.

Let us modify the previous SQL statement by removing the redundant\textsuperscript{14} prefixes and analyze the resulting SQL statement (Listing 5.2).

```sql
SELECT
  first_name, last_name
FROM
  employees e, titles t
WHERE
  e.emp_no = t.emp_no AND
  t.title = 'Manager'
```

Listing 5.2: Simple SQL statement, unprefixed columns

\textsuperscript{14}If the table definitions were known.
We extract the following table collection and column collection.

```
employees e, titles t
```

```
first_name, 
last_name, 
title, 
e.emp_no, 
t.emp_no 
```

Now the column resolution is ambiguous, because we do not know whether first_name, last_name, title belong to employees e or titles t. In our case avoiding false negatives is more critical than avoiding false positives. Hence when having a column without prefix, we match it to every table in the table collection increasing the size of the column collection.

```
employees.emp_no, 
employees.first_name, 
employees.last_name, 
employees.title, 
titles.emp_no, 
titles.first_name, 
titles.last_name, 
titles.title 
```

We note that our column collection now contains columns that do not exist in the database. These can be seen as false positives; however, they do not have severe impacts on our use case. (Section 6.4)

Before moving further to some more complex queries, we consider the following SQL statement (Listing 5.3) that projects \( * \) from the table salaries.

```
SELECT
  *
FROM
  salaries
```

Since we do not have the table definitions, we do not know the columns that are accessed. We therefore treat the \( * \) as a column name in the extraction phase.

```
 salaries
```

Listing 5.3: Simple SQL statement, select *

```
*
```

The \( * \) is then resolved as usual. However, it has a special meaning and is treated differently, as we will see in the next example.

```
salaries.*
```

We distinguish between prefixed and unprefixed \( * \) and name the later global \( * \).

\(^ {15} \text{We will use } * \text{ and } star \text { interchangeably.}\)
Now that we have covered the special cases in a simple SQL statement, let us move on to some more complex SQL statements containing nested statements (Listing 5.4).

```sql
SELECT
    ee.engineer_title, s.salary
FROM
    salaries s,
    (SELECT
        e.*, t.title as engineer_title, t.from_date, t.to_date
    FROM
        employees e,
        titles t
    WHERE
        e.emp_no = t.emp_no AND
        t.title LIKE '%engineer%') ee
WHERE
    ee.emp_id = s.emp_id AND
    ee.from_date <= s.from_date AND
    s.to_date <= ee.to_date
ORDER BY
    s.salary DESC
```

Listing 5.4: SQL statement with nested SELECT statement of type 1

We will build up the following table collection and column collection (duplicates included):

<table>
<thead>
<tr>
<th>salaries s, employees e, titles t</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.*, e.emp_no, ee.emp_id, ee.engineer_title, ee.from_date, ee.to_date, s.emp_id, s.from_date, s.salary, s.to_date, t.emp_no, t.title, t.title</td>
</tr>
</tbody>
</table>

The columns `s.salary`, `s.emp_id`, `s.from_date`, `s.to_date`, `e.*`, `t.title`, `t.from_date`, `t.to_date` and `t.emp_no` can be resolved to the corresponding tables as shown in the last examples. The resolution of the columns `ee.engineer_title`, `ee.emp_id`, `ee.from_date` and `ee.to_date` is a little more complex. We need to assign them to the sub select `ee` and resolve them in there. From a sub select point of view, these columns stem from a parent
statement. We therefore refer to them as external columns, likewise we use the
term internal columns/tables. In order to resolve external columns, we need to
keep track of all sub selects in a sub select collection and modify our resolution
procedure in the following manner: We start by extracting tables and columns
to our two collections as usual; in addition, we build up the collection contain-
ing all the sub statements. Subsequently, we start resolving the columns.
Whenever we encounter a column, whose prefix matches to a sub select, we
add it to a dedicated external column collection assigned to the sub statement.
After iterating through all columns in the SQL root statement we perform
the extraction and resolution phase recursively on every sub statement. On
this occasion we also consider the external column collection when resolving.
Once a sub statement has been processed, its resolved objects are added to the
corresponding collections of its parent.

Resolving external columns is different from resolving internal columns. We
need to maintain two additional structures.

- The result columns, including their alias that are projected. In the remain-
der referred as projected column collection.
- The tables from which * have been projected. In the remainder referred
  as star table collection.

We match the external columns to the projected columns or to every single
star table. This is elaborated in the continuation of the example, where our
sub select ee contains the following projected column collection ee

```sql
engineer_title=titles.title,
from_date=titles.from_date,
to_date=titles.to_date
```

and the following star table collection.

```sql
employees
```

```sql
  ee.engineer_title, ee.from_date and ee.to_date match to
titles.title, titles.from_date and titles.to_date titles.title, since
the names of the former three match with the alias (or name, in case no
alias is present) of the latter three. We can therefore ignore and remove
ee.engineer_title, ee.from_date and ee.to_date from our collection, as
its underlying columns have already been added to the column collection, when
processing ee. ee.emp_id does not have a match in the projected column
collection, hence will be matched to every star table, resulting in the following
resolved columns eventually.

```sql
e.*,
employees.emp_id,
employees.emp_no,
employees.from_date,
employees.to_date,
salaries.emp_id,
salaries.from_date,
salaries.salary,
salaries.salary,
```
salaries.to_date,
titles.emp_no,
titles.title,
titles.title

Note,

- when having ambiguity in column resolution in a sub select (Listing 5.2), a projected column alias might point to multiple columns, thus introducing false positives.

- when having multiple nesting levels, where a parent statement projects a global * or x.*, where x is a sub select that also projects *, the star tables in the sub select are transitive and need to considered in the star table collection of the parent statement. We handle this transitivity by maintaining a star statement collection.

We have seen how to handle sub selects of the above type, by adding the matching columns from the parent statement to a dedicated collection assigned to the sub select. There is another type of sub select, where the inverse needs to be done. These sub selects can select columns from tables of their parent statement, whereas the parent statement has no access to the sub select’s objects. The next SQL statement (Listing 5.5) illustrates this case.

```sql
SELECT DISTINCT *
FROM employees e
WHERE EXISTS (SELECT *
    FROM salaries s
    WHERE e.emp_no = s.emp_no AND s.salary >= 150000)
```

Listing 5.5: SQL statement with nested SELECT statement of type 2

The sub select in our EXISTS clause selects emp_no from employees in the parent statement. However the parent statement has no access to any objects in the sub select; hence, we do not need to maintain an external column list for this sub select, but we need to take the table collection of the parent statement into account, when resolving the columns of the sub select.

Extraction and resolution phase of the parent statement when ignoring its sub select is straightforward and results in the following tables and resolved columns.

```sql
employees e
employees.*
```

We will process its sub select accordingly, by first extracting the following table and column collection.

```sql
salaries s
```
s.emp_no and s.salary can be resolved as usual, while for e.emp_no, we need the table collection of the parent statement. Taking the latter into consideration we end up with the following resolved columns:

salaries.*,
employees.*, 
employees.emp_no,
salaries.emp_no,
salaries.salary

It is therefore important to distinguish between two types of sub selects (e.g. by keeping separate collections).

Other statement types So far we have particularly focused on SELECT statements. In the next few paragraphs, we will give an overview on how we handle other kind of statements; remember, that we are interested in potential read access. We will cover other types of Data Manipulation Language (DML) statements, Data Definition Language (DDL) statements and Data Control Language (DCL) statements.

DML: INSERT, UPDATE, DELETE

- **INSERT statement**: There are two forms of INSERT statements. INSERT statements that explicitly specify the values to be inserted and INSERT statements that insert the value returned by a sub select. We ignore the former, as this does not read any data of the database. In the latter we perform the complete SQL analysis on the sub select, as data may be leaked, when inserting rows from a confidential table into a publicly accessible table (Section 6.1).

- **UPDATE statement**: Similar to the INSERT statement, an UPDATE statement can either contain constant values or arbitrary sub selects for every value to be updated. We ignore constant values and perform the complete SQL analysis on all sub selects, as data can potentially leak, when updating the values of a public table with content from a confidential table (Section 6.1).

- **DELETE statement**: Although important to keep track of for other use cases, DELETE statements may be ignored for our particular use case, as data is neither read nor can be leaked.

Although we are not analyzing all DML statements, we write them into a separate result file for manual analysis on potential correlations (Section 6.1).

DDL: CREATE, ALTER, DROP
**CREATE statement:** CREATE statements per se do not leak any information. However, combined with DML statements, there are scenarios that we need to be aware of. Thus, we scan possible sub selects and treat the results as regular accesses (Section 6.1).

**ALTER statement:** ALTER statements in itself do not read nor leak any data. However, it can pose similar threats as CREATE statements (Section 6.1).

**DROP statement:** Same as DELETE statements, DROP statements are important to spot for other use cases, but may be ignored for our particular use case.

Although we are not analyzing all DDL statements, we write them into a separate result file for manual analysis on potential correlations (Section 6.1).

**DCL: GRANT, REVOKE** We currently do not analyze DCL statements. Reasoning about DCL statements might be part of future work in 8.2.2.

### Data Model and Structures

<table>
<thead>
<tr>
<th>GSPTable</th>
<th>GSPColumn</th>
<th>CanonicalGSPColumn</th>
</tr>
</thead>
<tbody>
<tr>
<td>-user : string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-name : string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-alias : string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-prefix : GSPTable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-name : string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-alias : string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-baseColumn : GSPColumn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-resolvedColumnList : List&lt;GSPColumn&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.4: Object model for database objects

We have defined the objects GSPTable and GSPColumn\(^{16}\) (Figure 5.4) to represent a column and a table. We model the columns to be resolved using CanonicalGSPColumn (Figure 5.4). CanonicalGSPColumn holds a GSPColumn as its base, which is the column extracted by our visitor. In addition to its base, it holds a list of GSPColumns that represents the possible resolved forms of the column. Having this additional layer allows us to handle ambiguous resolution scenarios quite naturally by appending all possible resolved forms of a column to its list. When having a single list of GSPColumns, we would have to deal with concurrent modifications while iterating over it, as in general, new columns are generated for every ambiguous case.

We have defined an object GSPAnalyzeResult (Figure 5.5) that holds all the relevant data structures of an SQL statement (Section 5.6.3). It will be constructed and populated by our visitor when traversing the abstract syntax tree.

### Resolution Algorithm

This section presents the resolution algorithm, the core of our SQL analysis. Listing 5.1 and 5.2 shows the algorithm in pseudo code, in which all findings in

\(^{16}\)GSP stands for *General SQL Parser* and was prefixed because different data structures for different parsers were used in an initial phase of the project.
section 5.6.3 are reflected. We have summarized the parsing and the traversing by our visitor in a function called ParseAndVisit. We have also wrapped a sub select with its external column list in an object called SQLWrapper.

There are cases, when CanonicalGSPColumn end up with an empty resolved column list.

1. The column has no prefix and there is no accessible table or sub select.
2. The column has a prefix that matches a sub select that does not project any alias that matches the column and there is no star table in scope.
3. The column has a prefix that matches no accessible table or sub select.

This can be due to Oracle Database specific constructs that are treated as table or column names by our SQL parser. We are currently writing these column names to a log file.

### 5.6.4 Pig Latin Script

Pig Latin has following data types:

- A **field** is a piece of data.
- A **tuple** is an ordered set of fields.
- A **bag** is a collection of tuples.
- A **relation** is a bag.

A Pig Latin script composes of a sequence of Pig Latin statements. A statement (except statements used for IO such as LOAD, DUMP or STORE) takes a relation as input and outputs another relation. A typical Pig Latin script has the following structure:

1. A LOAD statement reads data from the file system, producing a relation $r_1$.  

---

31

17Note that the concrete implementation in our object model (Figure 5.5) uses an equivalent object TSelectSqlStatementWrapper.

18http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html (accessed 23-July-2011)
Algorithm 5.1 Resolution Algorithm Part 1

1: % external column list
2: procedure Analyze(SQLWrapper sw, List<CanonicalColumn> cc)
3: % parse and visit the SQL statement and get the AnalyzeResult object
4: ar ← ParseAndVisit(sw.sql)
5: % note, that we deal with canonicalColumn here
6: for all cc in ar.allColumnList do
7: % check prefix and match to table/view/subselect
8: if cc.base has prefix then
9: if cc.base.prefix in ar.accessibleTableMap then
10: create c with the matched table as prefix
11: add c to cc.resolvedColumnList
12: else if cc.base.prefix in ar.accessibleSubstatementMap then
13: s ← ar.accessibleSubstatementMap.get(cc.base.prefix)
14: add cc to s.externalColumnList
15: else
16: % no match found for prefix
17: end if
18: else
19: for all t in ar.accessibleTableMap do
20: create c with t as prefix
21: add c to cc.resolvedColumnList
22: end for
23: for all s in ar.accessibleSubstatementMap do
24: add cc to externalColumnList of s
25: end for
26: end if
27: % check for *
28: if cc is * then
29: if cc.base has prefix then
30: if cc.base.prefix in ar.accessibleTableMap then
31: add matched table to ar.starTableList
32: else if cc.base.prefix in ar.accessibleSubstatementMap then
33: s ← ar.accessibleSubstatementMap.get(cc.base.prefix)
34: add s to ar.starSubstatementSet
35: else
36: % no match found for prefix
37: end if
38: else
39: set ar.globalStar = true
40: for all t in ar.accessibleTableMap do
41: add t to ar.starTableList
42: end for
43: for all s in ar.accessibleSubstatementMap do
44: add s to ar.starSubstatementSet
45: end for
46: end if
47: end if
Algorithm 5.2 Resolution Algorithm Part 2

48: % analyze all subselects
49: for all s in ar.allSubstatementList do
50:   sr ← Analyze(s.sql, s.externalColumnList)
51:   if s can access parent statement objects then
52:     for all cc in sr.allColumnList do
53:       if cc.base.prefix in ar.accessibleTableMap then
54:         create c with the matched table as prefix
55:         add c to cc.resolvedColumnList
56:       end if
57:     end for
58:   end if
59:   for all t in sr.allTableList do
60:     add t to ar.allTableList
61:   end for
62:   for all cc in sr.allColumnList do
63:     add cc to ar.allColumnList
64:   end for
65:   if ar.globalStar and s in ar.starStatementSet then
66:     for all t in sr.starTableList do
67:       add t to ar.starTableList
68:     end for
69:   end if
70: end for
71: % resolve all external columns
72: for all cc in ec do
73:   if cc.base.name in ar.getProjectedColumnMap then
74:     mec ← ar.getProjectedColumnMap.get(cc.base.name)
75:     add all columns from mec to cc
76:   else
77:     for all st in ar.starTableList do
78:       create c with the st as prefix
79:       add c to cc.resolvedColumnList
80:     end for
81:     for all s in ar.accessibleSubstatementMap do
82:       add cc to s.externalColumnList
83:     end for
84:   end if
85: end for
86: end for
87: return ar
88: end procedure
2. A series of 'transformation' statements take \( r_i \) as input, perform certain computations and output a relation \( r_{i+1} \).

3. A \texttt{STORE} statement writes the result \( r_{end} \) to the file system.

The most common transformation statements include \texttt{FOREACH} statements that iterate over the relation and applies a function on every tuple, \texttt{FILTER} statements that filter tuples according to a boolean expression and \texttt{GROUP} statements that group tuples in a relation by a key. Pig provides interfaces for defining \textit{User Defined Functions (UDFs)} that can be used in a wide range of transformation statements, providing the user with full flexibility in defining custom functions to suit their needs.

Our Pig Latin script is structured in the following manner.

1. A \texttt{LOAD} statement, using a generic XML load function, reads the XML audit log files from HDFS and treats every audit record as a single tuple, producing \( r_{xml} \).

2. A \texttt{FOREACH} statement, using our XML parser, parses the XML audit records in \( r_{xml} \) and outputs the relevant data as tuples, producing \( r_{auditRecords} \).

3. A \texttt{FOREACH} statement, using our XML analyzer, analyzes the tuples from \( r_{auditRecords} \) and extends the tuples with lists of accessed tables and columns, producing \( r_{analyzed} \).

4. A \texttt{FOREACH} statement takes the tuples from \( r_{analyzed} \) and flattens the list of accessed tables, producing \( r_{flattendTables} \).

5. A \texttt{FOREACH} statement takes the tuples from \( r_{analyzed} \) and flattens the list of accessed columns, producing \( r_{flattenedColumns} \).

6. A \texttt{GROUP} statement takes the tuples from \( r_{flattendTables} \) and groups them by table name, producing \( r_{groupedTables} \).

7. A \texttt{GROUP} statement takes the tuples from \( r_{flattendColumns} \) and groups them by column name, producing \( r_{groupedColumns} \).

8. A \texttt{GROUP} statement takes the tuples from \( r_{analyzed} \) and groups them by user name, producing \( r_{groupedUser} \).

9. Some subsequent statements format and sort \( r_{groupedTables} \), \( r_{groupedColumns} \) and \( r_{groupedUser} \).

10. \texttt{STORE} statements then write the resulting relations to HDFS.

Note, that step 3 returns relation \( r_{analyzed} \) containing all resolved audit records. Based on \( r_{analyzed} \), we can easily modify the script (e.g. by introducing different \texttt{GROUP} and \texttt{FILTER} statements) to formulate ad-hoc questions.
5.7 Output Formats

This section presents the formats of the result files written to HDFS, after DWH audit trail analysis and the intermediate results when preprocessing the schema snapshots. We will use the following notation: user\textsubscript{i} for users, t\textsubscript{i} for timestamps, s\textsubscript{i} for user schemas/owners, tbl\textsubscript{i} for tables\textsuperscript{19}, v\textsubscript{i} for views\textsuperscript{20}, col\textsubscript{i} for columns\textsuperscript{21} and a\textsubscript{i} for aliases.

5.7.1 DWH Audit Trail Analysis Result Files

We generate three results that answer our questions posed in section 4.2.

- **Accessed tables list:** A list sorted by table names (Listing 5.6). Every line contains one table, followed by a list of user access tuples (user\textsubscript{i}, t\textsubscript{i}).

```plaintext
... tbl\textsubscript{1} \{(user\textsubscript{1,1}, t\textsubscript{1,1}), (user\textsubscript{1,2}, t\textsubscript{1,2}), \ldots \}
tbl\textsubscript{2} \{(user\textsubscript{2,1}, t\textsubscript{2,1})\}
tbl\textsubscript{3} \{(user\textsubscript{3,1}, t\textsubscript{3,1}), (user\textsubscript{3,2}, t\textsubscript{3,2}), (user\textsubscript{3,3}, t\textsubscript{3,3}), \ldots \}
tbl\textsubscript{4} \{(user\textsubscript{4,1}, t\textsubscript{4,1}), (user\textsubscript{4,2}, t\textsubscript{4,2}), \ldots \}
...
```

Listing 5.6: Accessed tables list format

- **Accessed column list:** A list sorted by column names (Listing 5.7). Every line contains one column, followed by a list of user access tuples (user\textsubscript{i}, t\textsubscript{i}).

```plaintext
... col\textsubscript{1} \{(user\textsubscript{1,1}, t\textsubscript{1,1}), (user\textsubscript{1,2}, t\textsubscript{1,2}), \ldots \}
col\textsubscript{2} \{(user\textsubscript{2,1}, t\textsubscript{2,1})\}
col\textsubscript{3} \{(user\textsubscript{3,1}, t\textsubscript{3,1})\}
col\textsubscript{4} \{(user\textsubscript{4,1}, t\textsubscript{4,1}), (user\textsubscript{4,2}, t\textsubscript{4,2}), (user\textsubscript{4,3}, t\textsubscript{4,3}), \ldots \}
...
```

Listing 5.7: Accessed column list format

- **List of user accesses:** A list sorted by user names (Listing 5.8). Every line contains one user, followed by a list of time tuples (t\textsubscript{i}, list\textsubscript{i,tbl}, list\textsubscript{i,col}) containing a timestamp t\textsubscript{i}, a list tables list\textsubscript{i,tbl} and a list of columns list\textsubscript{i,col} that have been accessed at time t\textsubscript{i}.

```plaintext
... user\textsubscript{1} \{(t\textsubscript{1}, \{(tbl\textsubscript{1,1}), \{(col\textsubscript{1,1})\}), \ldots \}
user\textsubscript{2} \{(t\textsubscript{j}, \{(tbl\textsubscript{j,1}), \{(col\textsubscript{j,1})\}), \ldots \{(tbl\textsubscript{j+1,1}), \{(col\textsubscript{j+1,1})\}} \ldots \}
user\textsubscript{3} \{(t\textsubscript{k}, \{(tbl\textsubscript{k,1}), \{(col\textsubscript{k,1})\}), \ldots \}
user\textsubscript{4} \{(t\textsubscript{l}, \{(tbl\textsubscript{l,1}), \{(col\textsubscript{l,1})\}), \ldots \}
...
```

Listing 5.8: List of user accesses format

\textsuperscript{19}The prefix is included in our notation: tbl\textsubscript{i} \equiv s\textsubscript{i}.tbl\textsubscript{i}

\textsuperscript{20}The prefix is included in our notation: v\textsubscript{i} \equiv s\textsubscript{i}.v\textsubscript{i}

\textsuperscript{21}The prefixes is included in our notation: col\textsubscript{i} \equiv s\textsubscript{i}.tbl\textsubscript{i}.col\textsubscript{i}

35
Besides other result files, generated for debugging purposes, we also generate a raw result file (Listing 5.10), after step 3 in our Pig Latin Script (Section 5.6.4). In this file, every line corresponds to one audit record, listing the user that initiated the action, the timestamp, SQL statement type (e.g., SELECT), the target table if available (e.g., affected table in INSERT statement), the list of tables accessed in the SQL statement and the list of columns accessed in this SQL statement.

Listing 5.9: List of resolved audit records

5.7.2 Schema Snapshot intermediate Result Files

We have to preprocess all schema snapshot files, due to the issues described in section 6.2, resulting in parseable XML files (Appendix A). On top of that, the preprocessing also involves SQL analysis of the schema view file that unfolds the network structure and writes the flattened view information into a separate intermediate result file. This file stores for every user user\_i, a list of all its views v\_i containing the following information.

- A list of all accessed tables tbl\_i.
- A list of all accessed columns col\_i.
- A list of all projected columns including its aliases a\_i=col\_i.
- A list of all star tables tbl\_i\^*.

The format used in our implementation is the following.

Listing 5.10: Schema view file result format

5.8 Example

5.8.1 Schema Snapshot Processing

This section goes through a complete example, in order to illustrate how the described components fit together. We assume that the schema in figure 5.3 is owned by user ALICE and can be accessed by BOB. User BOB creates bob.high_sal_employees (Listing 5.11).
CREATE VIEW high_sal_employees AS
SELECT e.*
FROM employees e JOIN salaries s ON e.emp_no = s.emp_no
WHERE
  s.salary >= 150000 AND
  s.from_date <= CURRENT_DATE AND
  CURRENT_DATE <= s.to_date

Listing 5.11: View definition of bob.high_sal_employees

Bob grants access to ALICE, who creates a local synonym alice.high_sal_employees for bob.high_sal_employees.

The corresponding schema snapshot is generated as follows (Listings 5.12, 5.13, 5.14).²²²³

²² We ignore the schema dependency file, as it is never used in our approach.
²³ For the sake of clarity, we ignore line wrap issues and generate well-formed XML.
Listing 5.13: Example: Schema view file

We preprocess our schema snapshots and write an intermediate schema view result file. The processed schema snapshot files are copied to HDFS.

Listing 5.14: Example: Schema synonym file

Listing 5.15: Example: Schema view result file

5.8.2 Audit Log File Processing

Now, assume that we have copied the following audit log file (Listing 5.16) to HDFS.

Listing 5.16: Example: Audit log file
Depending on the size, this file might be split into multiple chunks that are processed individually. Our Pig load function returns single <AuditRecord> elements that are subsequently handled by our XML parser. The XML parser passes 2011-08-01T23:59:99.999999, ALICE and the value in <Sql_Text> to the XML analyzer. The XML analyzer performs extraction and resolution phase (Section 5.6.3), resulting in the following table/view/synonym names and column names.

```
alice.high_sal_employees, alice.titles

alice.high_sal_employees.first_name,
alice.high_sal_employees.last_name,
alice.high_sal_employees.emp_no,
alice.titles.title,
alice.titles.emp_no,
alice.titles.to_date,
alice.titles.from_date
```

The XML analyzer will look up the corresponding schema snapshot for the timestamp 2011-08-01T23:59:99.999999 and load the preprocessed schema snapshot files prior to schema matching. In schema matching, we iterate through our list of extracted table/view/synonym names and follow the order of precedence (Section 5.5). We start with high_sal_employees and look up alice.high_sal_employees in the schema table file and schema view result file. No match is found here. We continue by looking up the schema synonym file and find that alice.high_sal_employees is a synonym for bob.high_sal_employees, so we update the object name and repeat the previous steps using bob.high_sal_employees and eventually find our match in the schema view result file. As views mask underlying tables and its columns, we add all the tables accessed in bob.high_sal_employees to our table names.

```
br.bob.high_sal_employees, alice.titles, alice.employees, alice.salaries
```
Next, we resolve the columns that have `bob.high_sal_employees` as a prefix, namely `bob.high_sal_employees.first_name`, `bob.high_sal_employees.last_name`, and `bob.high_sal_employees.emp_no`. We do not find a match to any projected columns, as `bob.high_sal_employees` projects `e.*` only. Hence we match all these columns to the only star table in `bob.high_sal_employees alice.employees`, and add the resolved columns to our column names.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bob.high_sal_employees.first_name</td>
<td>First name of the employee</td>
</tr>
<tr>
<td>bob.high_sal_employees.last_name</td>
<td>Last name of the employee</td>
</tr>
<tr>
<td>bob.high_sal_employees.emp_no</td>
<td>Employee number</td>
</tr>
<tr>
<td>alice.titles.title</td>
<td>Title of the employee</td>
</tr>
<tr>
<td>alice.titles.emp_no</td>
<td>Employee number</td>
</tr>
<tr>
<td>alice.titles.to_date</td>
<td>To date of the title</td>
</tr>
<tr>
<td>alice.titles.from_date</td>
<td>From date of the title</td>
</tr>
<tr>
<td>alice.employees.*</td>
<td>All columns from the employees table</td>
</tr>
<tr>
<td>alice.salaries.emp_no</td>
<td>Employee number</td>
</tr>
<tr>
<td>alice.salaries.salary</td>
<td>Salary</td>
</tr>
<tr>
<td>alice.salaries.from_date</td>
<td>From date of salary</td>
</tr>
<tr>
<td>alice.salaries.to_date</td>
<td>To date of salary</td>
</tr>
</tbody>
</table>

Next, we lookup `alice.titles` in the schema table file and schema view result file and find out that `alice.titles` matches to a table. No further processing steps need to be done in this case. We have finished our analysis of one audit records and continue with the next.
Chapter 6

Limitations

Our approach has shown to successfully parse most SQL statements and cover a big deal of possible scenarios. Nevertheless, we have discovered cases that we have not yet completely solved in our solution. This chapter specifies these limitations and elaborates the countermeasures we have introduced in order to mitigate their impact.

6.1 Series of correlated SQL Statements

There are many scenarios that involve a series of dependent SQL statements that lead to potential data accesses that cannot be spotted, when treating audit records individually. Most of them, involve DDL or DML statements creating or modifying certain structures in the DWH, followed by DML statements that make use of the modifications and read data, possibly closed up by DDL or DML statements that revert the previous modifications.

An example involving a single user might be the following. Assuming that we have a vicious DWH administrator Eddie, who does the following.

1. Eddie creates a view `my_secret` that accesses rows of a confidential table.
2. Eddie accesses the confidential table via `my_secret`.
3. Eddie drops `my_secret`.

Our analysis will extract the table `my_secret`, but when performing schema matching, we will not find a match, as `my_secret` was created during the day. Neither will we find a match in a newer schema snapshot, as `my_secret` has been deleted before the second snapshot was generated. This scenario can be handled by performing SQL analysis on the `CREATE` statement and treat the results as regular DWH access. This would register read access on the underlying tables and columns, when the view is created.

Analysis becomes more difficult, when multiple users are involved.

We can repeat the same scenario with an additional vicious user Fred.

1. Eddie creates a view `my_secret` that accesses rows of a confidential table.
2. Eddie grants read access to Fred on `my_secret`.

41
3. Fred accesses the confidential table via my_secret.

4. Eddie drops my_secret.

When treating the single audit records individually, there is no clear indication that Fred has accessed the confidential data. Our analysis will spot that Eddie has created my_secret and therefore has accessed the underlying table and columns, but we will not figure out that Fred has also accessed the underlying table and columns. We counteract this by writing all DDL statements to a separate result file and marking all the object names that have no match in the schema snapshot. Whenever we encounter an object name without schema snapshot match, we need to manually analyze the potential correlation to DDL statements.

Similar scenarios can be modeled by using ALTER statements, where Eddie renames a table or DML statements, where Eddie inserts data from a critical table into a noncritical table or updates its columns with values of a critical table. Our countermeasure in a single user case is likewise to analyze the INSERT and UPDATE statements and treat the results as regular accesses. Compared to the scenarios, where DDL statements were involved, we have no indication of abnormal behavior (e.g. object names that do not match to any schema snapshot object). As a consequence, the multiple user case becomes a tough problem to solve. A scan through all DML and DDL statements might be needed to work out all conspicuous modifications.

6.2 Schema Snapshot Generation Issues

Although conceptually straightforward, it turned out that the schema snapshot generation process at Credit Suisse suffered from a bug, presumably originating from Oracle Database. The schema snapshots are generated with a vast number of randomly positioned line breaks, one occurring on average every 15 lines. We categorize these abnormal line breaks according to its implication on our processing.

- **Category 1**: Line breaks that are placed between ending XML tags and starting XML tags, leading to wrong indentation, without any consequences when parsing.

- **Category 2**: Line breaks that are placed in the text between starting XML tag and ending XML tag, invalidating some values, especially SQL statements.

- **Category 3**: Line breaks that are placed within an XML tag, rendering the whole XML file unparsable.

Obviously category 3 line breaks prevented us from processing the schema snapshots. We came up with a heuristic that fixed this issue, by removing all line breaks of terms matching following regular expression.

\[
<\text{\textless} >
\text{\\s}*
\text{\textgreater}\text{\textgreater}
\]
We applied similar patterns on category 2 line breaks. None of them were effective though, as in many cases we cannot distinguish between intentional or abnormal line breaks.

In addition to our line break issue, the schema snapshot files have two more problems that prevented us from processing the complete schema snapshot.

1. SQL statements are not escaped, making the XML files unparsable when containing `<` or `>` in their expressions. This issue could be easily solved, once the line break issue for the XML tags have been fixed by wrapping a CDATA\(^1\) section around the SQL statement.

2. Some SQL statements, especially large ones containing more than 2,500 characters, are truncated. This is an information loss that cannot be recovered without additional data.

Thus, limiting schema processing to parseable objects only, which fortunately make up the majority (Chapter 7.2).

### 6.3 General SQL Parser

Although we are using a powerful SQL parser, we have encountered several SQL statements that could not be parsed. An incomplete list is presented here.

- **TRUNCATE** statements

  ```sql
  TRUNCATE salaries
  ```

- String concatenation as part of function argument

  ```sql
  SELECT translate(e.gender, 'm' || 'f', 'm' || 'w') 
  FROM employees e
  ```

- Nested common table expressions

  ```sql
  SELECT id FROM (
      WITH females AS ( 
        SELECT * FROM employees WHERE gender = 'f'
      ), males AS {
        SELECT * FROM employees WHERE gender = 'm'
      }
      SELECT first_name, last_name 
      FROM females
      UNION 
      SELECT first_name, last_name 
      FROM males
  )
  ```

In practice however, only a tiny fraction of all SQL statements are of the above mentioned types (Section 7.2).

\(^1\)http://www.w3.org/TR/xml/#sec-cdata-sect (accessed 22-July-2011)
6.4 Missing Table Definitions

This case has been illustrated in figure 5.2 already. Since we have no information about the table definitions, we are not able to precisely resolve columns with no prefix. We match them to all accessible tables and return all the possible resolved forms. This introduces false positives, which do not have severe negative impacts on our processing, as we notice, when referring back to our use case in section 4.2 and the result files in section 5.7.1.

In the results of question 1 and question 2, these false positives reflect as additional rows, declaring columns that do not exist. This does not affect the rows we are interested in.

In the result of question 3, these false positives will be found in the list of column access tuples, not affecting the true positive in the list.
Chapter 7

Experiments and Measurements

7.1 Experiments on synthetic Data at ETH Zurich

We ran experiments on a cluster infrastructure at ETH Zurich, in order to show the scalability of our approach. Since all data generated at Credit Suisse is highly confidential, we were not allowed get a copy of that data and perform experiments internally at ETH Zurich. Neither could we arrange to work onsite at Credit Suisse. Hence, we generated synthetic data according to limited samples of production data that we were allowed to view. Examples thereof can be seen in Appendix A.

We were deploying Hadoop on a cluster of 4 nodes, using the following configuration:

- AMD Opteron 2376 2.3 GHz, 8 cores, 16 GB RAM
- 1 DataNode, 1 TaskTracker per node
- 2 mappers, 2 reducers per TaskTracker
- Replication factor 2 (for $clustersize > 1$)

We generated following synthetic data\(^1\)

- 2 schema snapshots
- 25,000 tables per snapshot
- 30,000 views per snapshot
- 10,000,000 audit records, split up in 400 audit log files

\(^1\)We have ignored the schema synonym files in our experiments on synthetic data.
The synthetic data contain simple SQL statements with a FROM clause, containing 1-6 tables, 0-3 views and 0-1 sub select, a SELECT clause, projecting 1-2 columns of every FROM clause object and a WHERE clause, accessing another 1-2 columns of every FROM clause object.

We performed time measurements on our analysis for different configurations, varying the following two parameters.

- Number of audit records ranging from 25,000 to 10,000,000 with data sizes from \( \sim 30 \) MB to \( \sim 12 \) GB.
- Number of cluster nodes ranging from 1 to 4.

We ran every time measurement 5 times and calculated the mean execution time and the 95\% confidence interval. Figure 7.1 shows the results.

![Figure 7.1: Execution time with various cluster sizes and data sizes](image)

For irrelevantly small data sizes \( < 250,000 \) audit records we can perceive the overhead induced by Hadoop. Other than that for reasonable sized input data, our solution shows linear scalability. Processing 10,000,000 audit records using a single cluster node takes approximately 13.6 hours. When using four cluster nodes, the computation time decreases linearly by 75\%, down to approximately 3.4 hours. Projecting the results to the data sizes generated per month at Credit Suisse, we notice however that the performance, in contrast, is low. Rounded up, the estimated amount of 6 TB audit log file data generated per month is 500 times larger than the largest data set in our experiment. Using a cluster with 32 nodes, the whole data would still require 8.5 days to be fully processed. Reasons for that, most probably lie in the computation intensive SQL analysis performed on every audit record.
7.2 Experiments on production Data at Credit Suisse

Of course we are interested in how our DWH audit trail analysis solution performs on real data. Due to time and availability reasons, we were not able to perform complete time measurements. Nevertheless we managed to perform experiments, obtaining numbers reflecting the quality of our solution.

These experiments include:

- **SQL parsing**: How many real SQL statements can be parsed?
- **Schema matching**: How many extracted object names, can be matched to objects in the schema?

The first experiment shows how well our SQL parser performs on real data. We have executed these tests on a set of audit log files, as well as on the schema view file of a schema snapshot.

The results in parsing the SQL of audit log files are very good. Only 100 SQL statements, which corresponds to 0.47% of all SQL statements encountered in the audit log files, cannot be parsed. Also a large proportion of the schema view file SQL statements can be parsed without problems. Out of the 1,860 (8.63%) unparsable SQL statements, we estimate that the majority of the statements are invalidated due to line breaks, described in Section 6.2.

The second experiment either verifies the precision of our extraction procedure and the quality of the generated schema snapshots. In any case, these numbers are useful in estimating the amount of audit records that can be completely processed. Note, that the numbers include duplicate object names.

Again, we have very decent results for the table/view names accessed from SQL statements in the audit records\(^2\). Only 0.34% of the table/view names ended up without a match in the schema snapshot. We took a closer look at these names and identified the following types.

- Table functions\(^3\) that return a result set that can be selected like a table. Obviously, these are not persisted in the schema.

\(^2\)Obviously, only the SQL statements that could be parsed were considered.

\(^3\)http://download.oracle.com/docs/cd/E11882_01/appdev.112/e17126/tuning.htm#LNPLS915 (accessed 26-July-2011)
- Database links\(^4\) that point to a remote database, whose schema is not defined in the schema snapshot of the local data warehouse.

- Regular table/view name, whose match in the schema snapshot cannot be parsed due to the line break issues described in 6.2.

- Regular table/view names, that cannot be found in the scope of the current user, due to schema modifications, privilege revocations or other reasons.

For the table/view names accessed in the view definitions in the schema view file, we find 1,669 (3.5\%) table/view names without schema snapshot match. Those are of the same types as the ones identified in the audit log files.

We want to estimate the number of view definitions that can be completely processed in the worst case. Thus, we assume that all the 1,669 names that have no match, stem from a separate view definition. Subtracting this number from the successfully parsed view definition, results in 18,018 view definitions that have been successfully parsed and whose accessed table/view names have a match in the schema snapshot. This is still 83.62\% of all views in the schema snapshot. Fixing the line break issue in the schema snapshot generation would significantly improve this ratio.

Now, we want to estimate the number of audit records that can be completely processed in an average case. We make following assumptions.

- The set of 21,066 audit records is representative for all audit log files ever to be generated.

- All accessed table/view names (38,215) are uniformly distributed on all tables (29,494) and views (21,547, where 83.62\% can be successfully processed).

- All synonyms references are also uniformly distributed on all tables and views, so we do not have to consider them in our estimate.

We have 20,966 audit records that can be parsed. Assuming the worst case, where all the 131 accessed table/view names without schema snapshot match,\(^4\)http://download.oracle.com/docs/cd/E11882_01/server.112/e17120/ds_concepts002.htm#ADM112003 (accessed 26-July-2011)
belong to individual audit records, we are left with 20,835 statements that can be parsed and whose accessed table/view names can be matched. Of all the schema matches, 57.79% point to tables, whereof 100% can be parsed and 42.21% point to views, whereof 83.62% can be completely processed, eventually resulting in 93.07% of the audit records that can be processed.
Chapter 8

Conclusion and Future Work

8.1 Conclusion

We have shown that we are able to develop an alternative solution for query log processing for a generic use case, that is flexible, highly-scalable and very cost effective, using Hadoop and Pig.

Most of our processing tasks can be easily adapted to the MapReduce processing model. Solely schema snapshots with their inherent network structure do not fit well into the input format of individual key/value pairs and we have implemented a reasonable workaround. We are not claiming that schema snapshots cannot be processed using MapReduce, it might simply need significantly more effort, which in our case was not worth investing. Pig Latin has turned out to be very easy to learn and flexible in its usage. We have defined our UDFs that perform XML parsing and SQL analysis. These constructs form a basis for a wide range of ad-hoc queries that can be customized using simple \texttt{FILTER} and \texttt{GROUP BY} statements.

When treating audit records as independent pieces of data, we are not able to reason about any correlation between SQL statements. We have come up with artificial scenarios involving collaborating DWH users, where one user might read data and go unnoticed by our automatic analysis. We have implemented a pragmatic workaround that involves manual work in order to spot these cases.

Our current solution has gone through experiments on real data at Credit Suisse and has proven to be able to process more than 90% of the all audit records in the DWH audit trail, even in presence of malformed schema snapshots. When fixing these issues, this ratio will significantly increase, leaving only a tiny fraction of audit records that need to be analyzed manually. We have executed multiple time measurements on synthetic data with varying the input size from 25,000 to 10,000,000 audit records and cluster sizes from 1 to 4 nodes. The results show linear scalability when using sufficiently large input sizes. The performance of our current implementation is in need of improvement. An estimation projects that the processing time for all audit log files generated in a month at Credit Suisse might take a little more than one week on a 32 node cluster. We might currently counteract the low performance by adding more nodes in our cluster.
An organizational challenge during our work, was dealing with confidential data that we were not allowed to view. We extracted a specification out of a limited set of production data and generated synthetic data for our tests. When handing over our application to Credit Suisse, in order to run experiments on real data, we were only allowed to access part of the results and in some cases even had to adapt our solution to return anonymized results. This was very time consuming and it took quite a few cycles until we had tuned our implementation to cover most SQL statements.

All in all, the results are promising, nevertheless applicability in practice needs to be evaluated and future work has already been identified.

8.2 Future Work

8.2.1 Evaluate Applicability in real Audit Case

Our experiments have shown to achieve a relatively high ratio of audit records and schema objects that can be processed. The proven high scalability might compensate low performance. The next open step is to apply our DWH query log analysis approach on a real audit case and evaluate its applicability and usability in practice.

8.2.2 Refine Solution Concept for exceptional Cases

We have seen in section 6.1 that certain scenarios require reasoning about the correlation between SQL statements. Some cases have been solved in a pragmatic way involving manual work, for others only a solution concept exists. Clearly, all of them are artificially made up and we also need to evaluate whether they are relevant in practice. Not a lot of research has been done on evaluating all these scenarios. A systematic approach thereof needs to be executed and a clean solution for the relevant cases has to be worked out.

8.2.3 Distribute Schema Snapshot Processing

Our current approach is to preprocess the schema snapshots and to load the complete result files from HDFS, on demand, into data structures that allow for efficient matching. This happens within one single UDF which works fine as long as the schema snapshots are reasonably sized. In case the schema snapshots grow to a size, equivalent to the audit log files size, schema processing and matching will become bottlenecks. Distributing the schema snapshot processing would resolve this problem. Conceptually we would need to run multiple iterations of the same MapReduce jobs, each unfolding one level of the nested view dependencies. In practice, the number of iterations needed might be quite small, as the number of required iteration steps is limited to the maximum depth of all view references.

Schema matching can be conceptually described as a join on the table/view or synonym name. Pig Latin supports an operation that performs joins between two relations. This operation could be exploited to implement such a construct for matching.
8.2.4 Performance Tuning

Our solution has shown to be highly scalable. Its performance however, is very slow due to heavy SQL analysis being run on every audit record. We need to profile our current solution, identify the computation heavy components and find solutions for optimizing them. We should also clarify the relevant scenarios in practice. In case we do not have star projections in our SQL statements, the resolution algorithm can be significantly simplified, by simply returning the union of the accessed columns of all (sub) statements. If we are only interested in accessed tables, we can completely skip the resolution algorithm.

A recent update of the General SQL Parser\(^1\) states to support table and column extraction. Evaluation of the quality and performance of this feature, is to be done.

Appendix A

XML Formats

A.1 XML Format: Sample Audit Log File

```xml
<Audit>
  <AuditRecord>
    <Audit_Type>0</Audit_Type>
    <Session_Id>97767</Session_Id>
    <StatementId>4</StatementId>
    <EntryId>6</EntryId>
    <Extended_Timestamp>2011-06-01T03:00:00.000000</Extended_Timestamp>
    <DB_User>ZNSEZXLX</DB_User>
    <OS_User>MGSHGAAWR</OS_User>
    <Userhost>YNLPBLUART</Userhost>
    <OS_Process>ZBYSIUVCVA</OS_Process>
    <Terminal>LIDLDEGKWD</Terminal>
    <Instance_Number>8</Instance_Number>
    <Action>7</Action>
    <Returncode>2</Returncode>
    <Comment_Text>QXBIHMKBNI</Comment_Text>
    <Scn>-2052704139260945350</Scn>
    <Sql_Text>
      SELECT QPWBA.CPZMNUNKKU
      DECODE(QPWBA.FJXSRTBJZL, '5', 'I', '3', 'N', 'KTVKJ')
      YZZWD,
      XVERI.FAUUB MTXNA,
      FROM ADVKYMGG.T2629 QPWBA,
      BELQPOLZ.V3327 XVERI
      WHERE QPWBA.UAMYMDEZBL = 0 AND XVERI.GDHWY = 0 AND 1=1
    </Sql_Text>
  </AuditRecord>
</Audit>
```
A.2 XML Format: Schema Snapshot

A.2.1 XML Format: Schema Snapshot - Table File

```
<ROWSET>
  <ROW>
    <OWNER>ADVKYMGG</OWNER>
    <TABLE_NAME>T2629</TABLE_NAME>
    <TABLESPACE_NAME>ADVKYMGG</TABLESPACE_NAME>
    <CLUSTER_NAME>C_P#</CLUSTER_NAME>
    <STATUS>VALID</STATUS>
    <PCT_FREE>0</PCT_FREE>
    <PCT_USED>0</PCT_USED>
    <INI_TRANS>0</INI_TRANS>
    <MAX_TRANS>0</MAX_TRANS>
    <INITIAL_EXTENT>999999</INITIAL_EXTENT>
    <NEXT_EXTENTS>9999999</NEXT_EXTENTS>
    <MIN_EXTENTS>99</MIN_EXTENTS>
    <MAX_EXTENTS>9999</MAX_EXTENTS>
    <PCT_INCREASE>99</PCT_INCREASE>
    <FREELISTS>9</FREELISTS>
    <FREELIST_GROUPS>9</FREELIST_GROUPS>
    <LOGGING>YES</LOGGING>
    <BACKED_UP>N</BACKED_UP>
    <NUM_ROWS>0</NUM_ROWS>
    <BLOCKS>999</BLOCKS>
    <EMPTY_BLOCKS>0</EMPTY_BLOCKS>
    <AVG_SPACE>0</AVG_SPACE>
    <AVG_ROW_LEN>0</AVG_ROW_LEN>
    <DEGREE>1</DEGREE>
    <INSTANCES>1</INSTANCES>
    <CACHE>N</CACHE>
    <TABLE_LOCK>ENABLED</TABLE_LOCK>
    <TABLESPACE_NAME>ADVKYMGG</TABLESPACE_NAME>
    <SAMPLE_SIZE>0</SAMPLE_SIZE>
    <PARTITIONED>N</PARTITIONED>
    <TEMPORARY>N</TEMPORARY>
    <SECONDARY>N</SECONDARY>
    <NESTED>N</NESTED>
    <BUFFER_POOL>DEFAULT</BUFFER_POOL>
    <ROW_MOVEMENT>DISABLED</ROW_MOVEMENT>
    <GLOBAL_STATS>YES</GLOBAL_STATS>
  </ROW>
</ROWSET>
```
A.2.2 XML Format: Schema Snapshot - View File

```
<ROWSET>
  <OWNER>BELQPOLZ</OWNER>
  <VIEW_NAME>V3327</VIEW_NAME>
  <TEXT_LENGTH>924</TEXT_LENGTH>
  <TEXT>
    SELECT
      PTRYG.UZLIIVBPCR KGLJT,
      HGIK0.MQIPZFFKWK HZPXX,
      SUGKZ.JAVGJIZHUA GDHWY,
      DECODE(GDNTQ.BCBJVMNCRF, '8', 'L', '6', 'V', 'DNVMJ') KURFL,
      JM.JT.RJATX TZGOp,
      DECODE(JM.JT.KFQTZ, '0', 'M', '8', 'B', 'LWHSV') FAUUB,
      CAST(SUGKZ.CSLIBGYYWW AS DATE) BODHQ
    FROM
      VILKQHBT.T1385 PTRYG,
      YCGYQPPL.T1137 HGIK0,
      IKTHFDVM.T528 SUGKZ,
      ADDFMPMM.T997 GDNTQ,
      (SELECT
        CHHBR.EGIMUKYOByC UWWTY,
        HGYU.APWHYZVYYWQ KFQTZ,
        CHHBR.DYNLKCSTBY DBLVD,
        WXUWJ.PMBHJNKEK RJATX,
        HGYU.RZKTTLDAV BMETN,
        WXUWJ.NYHHQBEQSN QYXAF
      FROM
        NEBYEUXI.T2115 WXUWJ,
        ZAWRMKBO.T3685 HGYU,
        ADDFMPMM.T2726 CHHBR
      WHERE
      ...)
```

WHERE

PTRYG.ZBLSVIWNTB = 0 AND
HGIKD.QRHKJSMEYO = 0 AND
SUCKZ.YZZFPCBIZ = 0 AND
GNTQ.PHWPYXXEH = 0 AND
GNTQ.MZUVUVWQS = 0 AND
JMJT.DLBVD = 0 AND
1=1

A.2.3 XML Format: Schema Snapshot - Dependency File
A.2.4 XML Format: Schema Snapshot - Synonym File
Bibliography


