university of groningen

faculty of mathematics and natural sciences

Multi-dimensional access control for distributed column-oriented databases

Master's thesis

February 2013

Student: Erwin Vast

First supervisor (Kalooga): Drs. Mathijs Homminga

Second supervisor: Dr. Alexander Lazovik

Third supervisor: Dr. Frank Brokken

ABSTRACT

Distributed NoSQL databases are increasingly used for data mining. One technique is to enrich analysis results with data of other databases. In environments as the government these databases can contain sensitive data. Most relational databases support fine-grained access control for individual data elements. In distributed databases this leads to significant performance loss when the results and access rights are stored on separate machines. Furthermore, the original rights can apply to different dimensions in the merged dataset.

This work proposes multi-dimensional access control for column-oriented databases. It is based on role based access control to secure a merged dataset. The access rights are stored within each database cell for the best performance. A prototype based on Apache HBase compares the cell location with two other locations: a separate table and in each row. The locations are compared quantitative with the Yahoo! Cloud Serving Benchmark on a computer cluster of 14 machines.

The results show that the location of the access rights has a significant influence on the performance. When the access rights and the data are stored physically closer, both throughput and disk usage increase. The best location depends on the required access control dimension. For a merged dataset where multiple dimensions have to be checked, storing the rights in each cell has the best performance.

ACKNOWLEDGMENTS

Many thanks go to Alexander Lazovik and Frank Brokken for their time and feedback. I would like to thank Kalooga for providing an inspirational working environment and Mathijs Homminga for his guidance and support during the research. I would also like to thank other people of Kalooga for their suggestions and ideas.

CONTENTS

1 INTRODUCTION 1
- 1.1 Problem background 2
- 1.2 Research overview 4
- 1.3 Contribution 5
- 1.4 Thesis organization 5

2 DATA CONFIDENTIALITY 7
- 2.1 Security Models 7
- 2.2 Relational databases 9
- 2.3 Non-relational databases 11
- 2.4 Conclusion 13

3 MULTI-DIMENSIONAL ACCESS CONTROL 15
- 3.1 Problem statement 15
- 3.2 Research method 16
- 3.3 Architecture 16
- 3.4 Application interface 19
- 3.5 Access control model 23
- 3.6 Storage locations 27

4 PERFORMANCE EVALUATION 33
- 4.1 Controlled experiment 33
- 4.2 Data collection 35
- 4.3 Process 39
- 4.4 Performance properties 40
- 4.5 Throughput experiments 42
- 4.6 Data size experiments 45
- 4.7 Evaluation 47
- 4.8 Limitations 48

5 CONCLUSION 49
- 5.1 Future research directions 50

BIBLIOGRAPHY 51

A APPENDIX A - DETAILED RESULTS 55

B APPENDIX B - AMAZON INSTANCES 59

LIST OF FIGURES

Figure 1 Example situation of data mining with multiple sources. 2
Figure 2 The CIA and AAA models. 7
Figure 3 Layered architecture of solution. 17
Figure 4 Class diagram. 18
Figure 5 Example structure. 24
Figure 6 Three locations of the storage data. D stands for data and R stands for access rights. 28
Figure 7 Overview of experiment design. The storage location is changed while measuring the throughput and database size consequences. 33
Figure 8 Setup of the distributed test system 36
Figure 9 Activity diagram of test procedure. 40
Figure 10 Throughput results for writing large amount of rows. 42
Figure 11 Throughput results for reading large amount of rows. 43
Figure 12 Throughput results for writing large amount of columns. 44
Figure 13 Throughput results for reading large amount of columns. 44
Figure 14 Data size after write test for large amount of columns. 45
Figure 15 Data size after write test for large amount of rows. 46
Figure 16 Amazon instances. 59

LIST OF TABLES

Table 1 Example dataset with personal records per citizen. 3
Table 2 Example dataset with financial records per citizen. 3
Table 3 Example merged dataset. Used abbreviations: Secret (S), Top Secret (TS), Fred (F), Sandra (S), Paul (P). 4
Table 4 Example of a view construction. (a) table with data and user ID's, (b) a list of users and (c) the view that Edward would have of (a). 10

Table 5 Example of a column-oriented structure. 12
Table 6 Comparison of four interface variants for multi-dimensional access control. 21
Table 7 Options available for performance improvements. 23
Table 8 Example of table storage location. 29
Table 9 Example of column family storage location. 30
Table 10 Conceptual new cell contents. 30
Table 11 All variables used for the experiments. Each combination leads to a configuration for the test. 34
Table 12 Summary of Hadoop and HBase services [1]. 35

LISTINGS

Listing 1 Query extension example 9
Listing 2 Pseudocode interface 22
Listing 3 Changes to the Yahoo! Cloud Serving Benchmark to support multi-dimensional access control (creating table). 37
Listing 4 Changes to the Yahoo! Cloud Serving Benchmark to support multi-dimensional access control (initializing model). 38

1 INTRODUCTION

The storage of data is increasing now more than ever. Popular websites like Facebook, Twitter and Yahoo! process vast amounts of information. Users all over the world post messages, upload videos and share knowledge. These websites are always available, fast and up-to-date. They use large computer clusters to support these qualities. For even better performance, distributed NoSQL databases are used [2]. NoSQL databases use Not Only SQL and can store unstructured data far more efficient than relational databases [3][4]. Cassandra, MongoDB and Apache HBase are examples of such databases.

Because of their success, distributed NoSQL databases are also increasingly used for data mining. Data mining often involves large unstructured datasets and requires massive processing power. This leads to challenges for environments with highly sensitive data. Applications such as electronic medical records (EMR) and surveillance programs (National Security Agency) use personal records, financial information or other sensitive data. Distributed NoSQL databases are developed primarily for performance. Therefore security is limited [5] [6]. One technique of data mining that requires security is the extraction of information from several databases. The database owners often restrict access to specific users. The access rights of the original databases should be stored and checked in the merged database.

To check the access permissions of each data element in the merged database, fine-grained access control is required. Several distributed NoSQL databases support security features as user identification and coarse-grained access control. However, most of these databases do not support fine-grained access control for individual columns and rows. While in most relational databases this is supported, they assume that the access rights and the database contents are stored on the same machine. In distributed databases this may not be the case. When the access rights are stored on separate machines, the data transfer of these rights can decrease the performance significant.

This thesis focuses on fine-grained access control for distributed column-oriented databases. For this type of database, this thesis proposes multi-dimensional access control in which the access rights are stored in each database cell. With this solution, application developers can give access permissions for tables, columns and rows following the structure of a column-oriented database. Data mining results in the merged database for which a user has no access are filtered from the results. This work provides a solution to apply access control for data mining with sensitive data in a distributed database.

1.1 PROBLEM BACKGROUND

Most relational databases can be used for Online Transaction Processing (OLTP) whereas NoSQL databases may be used for Online Analytical Processing (OLAP) [7] [8]. NoSQL databases focus on storing massive amounts of data in the order of terabytes and petabytes. Furthermore, large analyses can be performed to provide information for companies, also called business intelligence. A common approach is to analyze a large dataset and enrich the results with data from other databases for better analysis or more information [9] [10]. The data mining process consists of five steps [11]. These steps combined with access control are depicted in Figure 1 and listed below.

1. **Selection**: The phase of selecting the databases to extract data from for analysis. In Figure 1 these are Sources A and B.
2. **Pre-processing**: The data and access rights are copied to the Data Analysis Machines. Furthermore, noise reduction and data cleaning are performed.
3. **Transformation**: Makes the data usable for analysis by changing the data structure and reducing the amount of variables.
4. **Data mining**: Analysis of data to find relations or statistical information. After analysis, the results are stored in a database.
5. **Interpretation**: Finally, the results are given to the user. For example with a web application. Only people that have access to the original data may see the results.

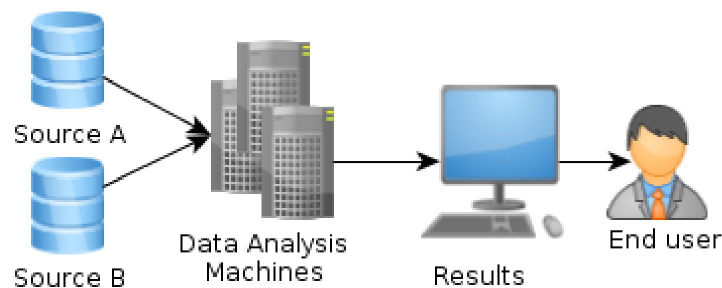

Figure 1: Example situation of data mining with multiple sources.

Another approach is to access remote data during the analysis phase instead of copying an entire database beforehand. However, in both cases there is a security issue. Each source database has its own access control information describing who may access the data. During data analysis the access rights from various databases are mixed. The database with the results requires access control that stores and checks the access rights in multiple dimensions. This problem is best shown using an example situation with two small source databases and a merged database.

For this example scenario a relational database structure is used. Table 1 shows a part of a database containing personal information about several citizens: the social security number (ID), the first name and surname, the address and the city. This database can only be accessed by the employees of the government Fred, Sandra and Paul. This access is visualized with colored rows in the table.

ID	First name	Surname	Address	City
83	Fred V.	Boyd	3350 Doctors Drive	Charleston
11	Edward	King	309 Stone Lane	Pottstown
67	Ryan	Weston	3532 Mattson Street	Salem
46	Patric	Gaston	2529 Shinn Avenue	Wilton
98	Sandra	Long	697 Melm Street	Providence
Access rights: Fred, Sandra and Paul				

Table 1: Example dataset with personal records per citizen.

A second source database is listed in Table 2 with information from the tax office. The citizens with the same ID as the first source database are selected. This table shows for each citizen the tax debt and money debt. This database has the rule that all information is Secret, but citizens with a money debt greater than €1.000.000 may only be seen with people having Top Secret access, visualized with colors in the table. Secret information is visible by John, Alba and Paul while Top Secret is only visible by John.

ID	Tax Debt	Money Debt	Security Level
83	€16.464	€516	Secret
11	€615	€5.612	Secret
67	€619.164	€1.164.167	Top Secret
46	€23.164	€642.165	Secret
98	€164.167	€5.267.167	Top Secret
Top Secret: John. Secret: Alba, John, Paul.			

Table 2: Example dataset with financial records per citizen.

Finally, the data are merged as shown in Table 3. This table shows for each citizen the merged first name and surname of the first database as full name and the city, tax and money debt of the second database. To maintain the access control, the full name and city columns are only visible for Fred, Sandra and Paul. The tax debt and money debt are only visible for users having the appropriate access label Secret or Top Secret as in the original second dataset.

ID	Fullname (F, S, P)	Tax Debt	Money Debt	City (F, S, P)
83	Fred V. Boyd	16.464 (S)	516 (S)	Charleston
11	Edward King	615 (S)	5.612 (S)	Pottstown
67	Ryan Weston	619.164 (TS)	1.164.167 (TS)	Salem
46	Patric Gaston	23.164 (S)	642.165 (S)	Wilton
98	Sandra Long	164.167 (TS)	5.267.167 (TS)	Providence

Table 3: Example merged dataset. Used abbreviations: Secret (S), Top Secret (TS), Fred (F), Sandra (S), Paul (P).

While in the original databases this information was stored in separate tables or rows, in the merged database the access must be checked in multiple dimensions. For example, to show the row with ID 83, the database has to check if the user has the name Fred, Sandra or Paul and Secret clearance. If this is not the case, the not allowed cells have to be filtered out or the row must be omitted completely. The focus of this work is to filter out the not allowed cells. With larger analyses the allowed information is still readable and more sensitive information can be read by users with higher permissions.

1.2 RESEARCH OVERVIEW

NoSQL databases differ in the structure of the stored data. Key-value stores, document stores, graphs and column-oriented databases are four common categories. This work is focused on distributed column-oriented databases. This type of database stores data in columns and is further explained in Section 2.3. The literature research in this thesis shows that relational databases have extensive security solutions. However, they assume that all data is stored on the same machine. As a result, research for the performance in distributed databases is limited as most research focuses on the SQL language to implement access control. In the field of column-oriented databases, most research shows that security is limited and should be improved.

To store a merged dataset with access rights, a multi-dimensional access control model is proposed in this thesis. This model checks for each database cell if it may be shown to the user based on the user's identity. When access is denied for parts of the data, that database information is hidden from the user. The model is based on role based access control and extended with rules to check multiple dimensions in a database e.g. tables, rows and columns. The model is implemented as an extension to Apache HBase, a column-oriented database. The developer of a data mining application can program rules that store the access rights for the data. Based on these rights, the user only sees the allowed results from the database.

Loading access rights from remote machines likely makes a distributed database significant slower. Distributed NoSQL databases store millions of rows and columns. Situations as described in Section 1.1 apply to a vast amount of cells in a dataset spread over a large number of machines. The purpose of this work is to show which location for storing the access rights for multi-dimensional access control has the best performance. Three storage locations for the access rights are compared: a separate table, a separate column within each row and within each cell. All locations provide the same storage options for the model. The primary difference between the three is the physical distance between the data and its access rights.

The performance of each location is tested with the Yahoo! Cloud Serving Benchmark. This benchmark compares the throughput when using each location. The data size of the database is retrieved from the Hadoop Distributed File System used by HBase. The test is run on a distributed system of 14 virtual machines with the Amazon Elastic Compute Cloud. The implementations are also compared with tests that has security disabled. The results show that security makes the database significantly slower as more computation steps are required. Of all three locations, multi-dimensional access control that stores the rights in each cell has the best performance.

1.3 CONTRIBUTION

This thesis contributes to the field of distributed NoSQL databases. It provides an implementation for role-based and fine-grained access control in Apache HBase. It can store data from various databases while maintaining the original access rights. Overviews are given of existing access control models and techniques that other databases use to implement those models. The implemented storage locations are tested by several experiments on a distributed system. The analysis of the results leads to a better understanding of access control in distributed NoSQL databases. The solution is based on the current state-of-the-art of access control in databases and provides the basis for security technologies such as cell-based encryption.

1.4 THESIS ORGANIZATION

This structure of this thesis is as follows. First, the results of a literature study are given in Chapter 2. Based on an analysis of the state-of-the-art, Chapter 3 describes the research questions, methodology and implementation details of a prototype. This prototype is used for extensive testing of the storage locations. Chapter 4 describes the testsetup, results and discussion. The thesis is concluded with an overview in Chapter 5 with the answers of the research questions and suggestions for further research directions.

2 DATA CONFIDENTIALITY

Access control management is extensively implemented in relational databases. In distributed NoSQL databases however, this a recent topic of research. After an introduction in access control models, the research in both areas is discussed. Finally, a conclusion describes which new research is required.

2.1 SECURITY MODELS

Preventing unauthorized access is a specific part of information security. This type of security is often associated with two models: CIA [12] and AAA [13]. Both models show a relation between other parts of information security. The emphasis of this work is shown in gray in both models as depicted in Figure 2.

The 'C' of CIA stands for confidentiality and is concerned with preventing that unauthorized persons read information they are not allowed to see. It is based on the 'need to know' principle in which only specific people have access to the data. Confidentiality is part of three attributes described within the CIA model for information security. The triangle in Figure 2 consists of confidentiality, integrity and availability. Beside confidentiality, integrity is concerned with that the data cannot be changed unnoticed by unauthorized persons. Availability is the quality term for the reachability of a service. This triangle is connected with lines to show the dependency between the attributes. This model shows that if a database implements access control, it will influence its availability.

The second abbreviation found in the security literature is AAA: authentication, authorization and accounting. The terms audit or administration are also used as synonyms for accounting. Authentication is used to identify a person, for example with a user name and password combination. Based on the identification of a user, an authorization system can give the user rights to do certain operations.

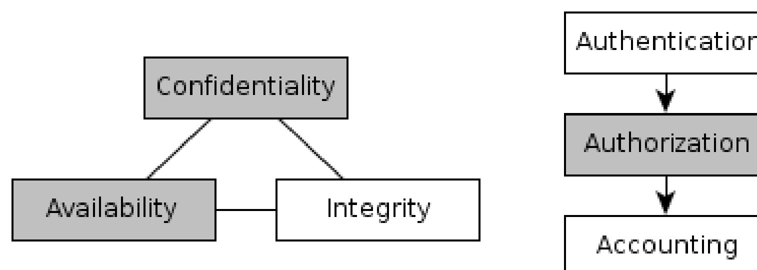

Figure 2: The CIA and AAA models.

For example, an administrator can access all systems, while a normal user may access only specific systems. Finally, accounting is used to log the actions of a user. The left image in Figure 2 shows the step-wise procedure of the AAA model. This indicates that to provide authorization, there must be an authentication system in place to identify who a user is. Furthermore, to provide accounting, the system must be able to check which operations a user did on which files or systems.

Databases can use several models for access control. These models grant access based on an identity and are all based on an access control matrix model. This matrix consists of objects, subjects and rights. All entities to be protected is the set of objects O. Examples of objects are files. The subject set S consists of identities, for example processes and users. The permissible actions are given from a set of rights R. The relationship between these entities is stored in a matrix A, where each entry is defined as [14]:

Definition 1. $\{a[s,o] | s \in S, o \in O, a[s,o] \subseteq R\}$.

This can be read as: a subject s has a set of rights $a[s,o]$ over the object o. With this table one can store the rights of each subject for an object. There are three common implementations of this model: Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Bases Access Control (RBAC) [15].

- DAC: Attaches a label to each object containing an owner and the access rights. The owner is responsible for the management of the access rights. An example are the data permissions in Unix. In this operating system, all data is labeled with the access rights for the owner, the group and an anonymous person.
- MAC: Also attaches a label to the object, but with a general permission level such as UNCLASSIFIED, SECRET and TOP SECRET. A subject can access the data if it carries that permission label. It has no mentions of operations. The labels and its possible actions are determined by a central administrator. This type is most common in government organizations e.g. FBI.
- RBAC: Each subject is associated with a role and each role gives the permission for certain operations on a given set of objects. The access rights are often stored in a separate database due to the many-to-many relationships of users and roles and can be modeled with a graph. This type is a policy neutral model as it can be managed by a central administrator as well as individual users. It is possible to store DAC and MAC in RBAC [16] [17].

Most relational databases implement one of these security models to prevent unauthorized access to the stored information.

2.2 RELATIONAL DATABASES

A field where security is extensively implemented is relational databases. They support fine grained access control for individual rows or columns. Based on the literature and database documentation, three implementation techniques are identified.

Query rewriting or extension is the first technique. Both Oracle's Virtual Private Database (VPD) [18] as SQL Server [19] use this technique. These databases provide fine-grained access control using query rewriting by appending conditions (predicates) to a query. These predicates check additional tables containing authorization information to link the access rights with the data. An example is shown in Listing 1. When the database should show the user only the data of the table `SensitiveData` for the sales employee with ID 26, the query would be extended by adding the predicate that forces that only data is shown which has the ID 26 attached to it. In more elaborate scenario's, such a predicate would check for the user in a separate table where all user information is stored.

The second method based on query extension are views. This technique is used by IBM-DB2 and SQL Server for row-level security [19] [20]. In general, database views show the contents of a database table with additional conditions. In the case of access control, such a view would show only the data that the user may see. For example, each row in a view has an underlying query that extends the query with the username as in the example of the previous technique. Furthermore, the access can be restricted to the view, preventing access to the base table. An example of a view is shown in Table 4. Table (a) shows a dataset with three rows each containing text and a user id. This user id is coupled with the user id stored in the usertable in table (b). When Edward access the view of table (a), the databases changes the underlying query as shown in Listing 1. Edward would see the data as shown table (c), excluding the rows that he is not authorized to see. The view implementation for SQL Server can be found in its documentation [19].

Finally, Sybase adds the possibility of using multiple access rule policies for each row [21]. Using an `AND` or `OR` predicate multiple access control lists can be used for checking the access rights. For example, when using an `AND` a user has to be in all of the given access lists. With an `OR`, a user has to be in only one of given lists. As with Oracle and SQL Server, Sybase uses query extension that extends the query with a predicate to check the access rights in a separate table.

Listing 1: Query extension example

```
1 SELECT * FROM SensitiveData;
2 SELECT * FROM SensitiveData WHERE UserId = 11;
```

SensitiveData	UserId
Top secret text	11
Public text	91
Secret text	37

(a) Table with sensitive data.

UserId	Surname	Lastname
83	Fred V.	Boyd
11	Edward R.	King
67	Ryan	Weston

(b) Table with users.

SensitiveData
Top secret text

(c) Table with data shown for Edward.

Table 4: Example of a view construction. (a) table with data and user ID's, (b) a list of users and (c) the view that Edward would have of (a).

Researchers from Microsoft [22] demonstrate how the SQL language can be extended with fine-grained authorization for a general standard. They argue that such a solution is faster than VPD of Oracle, but do not show how this proposed standard can be implemented and what the performance is. However, it seems as an interesting approach that would make the differences of access control implementations simpler. Furthermore, only that model could be compared with NoSQL, instead of multiple techniques described above.

All of the above techniques are part of an authorization model that change the original query. Rizvi et al. [23] name this model the Truman model and introduce the Non-Truman model. With the Truman model the outcome can differ for each person having different access rights. They give an example that calculates the average of a database column. They demonstrate that the Non-Truman model does not change the original query but rejects unauthorized queries completely. However, the implementation of this model is NP-complete and thus not decidable and can lead to various results in different database implementations [24] [25].

Byun and Li [26] store other information in the labels than access rights, but use a similar technique as this work. They extend a table with additional columns and demonstrate that "if the original relation contains large data elements ... the storage overhead of our labeling scheme becomes negligible". So a labeling scheme could be possible.

While Zhu and Lü [27] show that the performance of fine-grained access control has a linear performance, this and the listed research assumes that the data locality does not influence the performance. It seems that more quantitative research is required to show the dependency between access control and data locality. One research area where the distribution of the data and its access rights leads to performance problems, is in distributed NoSQL databases, discussed next.

2.3 NON-RELATIONAL DATABASES

NoSQL databases are often used for websites with high visitor count and storing information of data mining. Storing large amounts of data requires significant processing power and storage capacity. Therefore a distributed system is commonly used to deploy a NoSQL database. Relational databases traditionally use SQL to store information on mostly single systems. NoSQL is becoming increasingly popular for its more flexible database schema and better support for distributed systems [4]. Most relational databases support Atomicity, Consistency, Integrity and Durability (ACID) while most distributed NoSQL databases strive for Basically Available, Soft state and Eventual consistency (BASE) [28]. BASE databases convergence to consistency after some time and can change without input. NoSQL databases can be very fast for data storage but do not support transactions and JOINS. As data is distributed over many physical machines, queries with predicates spanning multiple tables would be very inefficient.

While NoSQL seems as a recent technology, database that store unstructured data are not new. Since the 1960's [4] databases exist that store information in graphs, trees and object oriented databases [29]. However, the newer databases are more focused on scalability and performance. The new term NoSQL is mostly concerned with databases that both are distributed and use a non-relational data store and does say little about the used language. This in contrast with SQL, which is a language. To further investigate the access control in unstructured and distributed databases, this literature study continues with distributed column-oriented databases (also called tabular databases). This type mostly resembles a relational structure.

A column-oriented structure differs significantly from a relational one. Whereas relational databases store information in a table form, a column-oriented database stores related information in columns [2]. This type of database can do fast calculations for a specific column. Table 5 shows an example of such a data structure (based on HBase). This table shows two rows in a column-oriented structure. Each row consists of a key to identify the individual row e.g. 16439287. Each row consists of column-families e.g. Personal. The family is not flexible and cannot be changed afterwards. However, the categories in a column-family, called column-qualifiers can be different for each row. In the table, the first row has a column-qualifier birthdate, while the second row has a column qualifier country. This prevents empty values that would be present in relational databases when a value for a column is empty. Most relational databases offer access control for unique rows and shared columns. In a column-oriented structure however, this is not possible as each row can have different column-qualifiers. Both the row as column-qualifier identifiers can be unique and the access rights for both dimensions have to be stored.

Row	16439287
Personal	Name: John
	Birthdate: 10-11-2012
	Country: United Kingdom
Row	61971673
Personal	Name: Bill
	Country: Germany
	Phone number: 0123-456789

Table 5: Example of a column-oriented structure.

Access control and even security in general is new in NoSQL databases. Where relational databases have matured security functionality, in recent NoSQL databases this is still in development. In a presentation about a security analysis, Schlesinger suggests "Don't store any confidential, secret, private data in Hadoop" [30]. Apache Hadoop MapReduce distributes analysis tasks over a distributed system and commonly uses Apache HBase. While this prevents sensitive information in the database, it is only a workaround for a real solution. If HBase has security issues than Hadoop also has security issues when it uses HBase.

As scalability and availability are two of the most important characteristics for most NoSQL systems, security is still in its infancy. Fernandez [6] discusses that "NoSQL databases lack strong security features". He explains that the then latest versions of Hadoop and HBase are missing access control. Any person or application can access the data. Furthermore, he states that "Hadoop cannot satisfy the requirements of applications handling medical or financial records". The author does not provide any technical solutions, but proposes a method for applying security solutions in various development phases.

Since 2011, Hadoop and HBase have improved security significantly. The HBase documentation [1] (Chapter 8) describes that authentication is implemented by incorporating Kerberos. Also a first version of access control is available in version 0.92 of HBase and later. HBase implements Discretionary Access Control by providing access control for tables, families and columns by access lists. However, it stores all access rights persistent in a separate table and temporary in memory when running. It seems likely that this solution will not work for fine-grained access control as storing the access rights for each cell cannot reasonably fit in memory. And loading access rights from a remote table probably causes significant performance loss. While HBase provides good functionality for course-grained access control, access control for a finer granularity requires a new solution.

Okman et al. [5] examine the security functionality of two NoSQL databases: Cassandra and MongoDB. They argue that these databases have security issues such as: no encryption of the data storage and communication, vulnerabilities for SQL injection and denial of service attacks. Furthermore, they observed that access control is limited and the reviewers conclude that both databases do not have "RBAC or fine-grained authorization". This supports the fact that access control seen in relational databases is still limited in NoSQL databases.

Enescu [31] describes the security in large-scale software systems from a scalability perspective. He provides an analysis for technologies including: intrusion detection, cryptography and data storage security. The conclusion of the author is "while the most attractive scalable databases (NoSQL solutions) lack in security features, they are still immature and there is no reason that they cannot be made more secure". However, he does not discuss the storage of access rights and how they influence the scalability of a distributed database.

One NoSQL database that provides cell-based security is Accumulo [32]. It is based on HBase, developed by the National Security Agency and responsible for data confidentiality in the government. They store a visibility label in the cell identifier that describes which users may access the cell's data. It supports AND and OR operators to support multiple users. However, roles or groups have to be managed by another application to prevent that large amounts of data have to be updated if the rights change. Furthermore, in Accumulo it is not possible to specify and check multiple dimensions as rows or columns.

Patil et al. [33] performed a benchmark of Accumulo's fine-grained access control. They show that with multiple clients it "reduces the insert throughput only by about 10%". However, "the scan throughput drops by about 45% once fine-grained ACLs are invoked" (ACLs are Access Control Lists). While they made the access rights more than 3 times larger than the rest of the cell, they also show the performance "remains [the] same for different number of entries in an ACL". Multi-dimensional access control could thus have a major impact for the performance. They did not test the performance of HBase as it did not support access control at the time of their writing.

2.4 CONCLUSION

Relational databases have extensive access control solutions. NoSQL databases however have limited security. The security issues are recognized, but fine grained access control is not available in most of these databases. Only Accumulo could be found with cell based security. However, checking multiple dimensions is not possible in Accumulo. Furthermore, access control can limit the scalability. Providing access control for a merged dataset requires a new solution. The next chapter discusses the workings of multi-dimensional access control.

3 MULTI-DIMENSIONAL ACCESS CONTROL

Security is limited in most column-oriented databases. They do not support access control as relational databases do. The primary reasons are: importance of performance, support for flexible data structures and large amount of different databases. However, there is a need for fine-grained access control as data mining mixes the access rights. This chapter describes the research design and proposes the interface, model and data storage of multi-dimensional access control for distributed column-oriented NoSQL databases.

3.1 PROBLEM STATEMENT

The performance of the implementation is mostly limited by the physical distance between the data and its access rights. Multiple storage locations are proposed in this chapter and compared by performance. This leads to the following research question:

Research question: *How does the storage location for multi-dimensional access control influences the performance of a distributed column-oriented database?*

The research question can be divided in three subquestions:

1. *How does multi-dimensional access control work?*
   This chapter describes the implementation details of the access control model in Apache HBase. Three storage locations are proposed: a separate table, within each row and within each cell. Each storage location stores the same data of the model to do a valid quantitative comparison.

2. *How are the performance results of the storage locations compared?*
   The performance is measured by the throughput and data size with the Yahoo! Cloud Serving Benchmark. The locations are compared on a distributed system. The design of the throughput and data size tests are explained in Chapter 4.

3. *How does the usage of each storage location affect the throughput and database size of a distributed NoSQL database?*
   The new access control functionality can be rejected if it makes the distributed database too slow. It is the purpose of this question to show a relation between the locations and performance. The results of the tests are discussed in Chapter 4.

3.2 RESEARCH METHOD

Multiple research methods can be used to retrieve the performance results. Following is a list of three methods that could be used for this work with their advantages and disadvantages. The first and second method are based on collecting new results, while the last method is based on a literature study.

- **Comparison of databases**: This method compares one new solution against other databases. However, such comparison is only possible if several databases support access control in multiple dimensions. They should all support fine-grained access control. Furthermore, they should implement the same underlying access control model (such as DAC or RBAC in Section 2.1). Otherwise databases with different properties are compared.
- **Controlled experiment of techniques**: The second method uses controlled experiments between the storage locations. Based on the results of the experiments against tests with security disabled, the best one can be selected for further research. The advantage of this approach is the retrieval of performance results that can be compared. However, it can cost significant time to implement each technique and perform the analysis on a distributed system.
- **Archival research**: Several papers have been written about access control models and how they are implemented in other databases. A comparison of existing benchmark results requires no programming as no new solutions are required. However, as described in Chapter 2, little research could be found with performance results for column-oriented databases.

For this work the **controlled experiment of techniques** is used. It is focused on quantitative analysis by measuring the throughput and data size. With the controlled experiments not only the best storage location is chosen, but it also gives application developers an impression of the performance penalties based on the original performance. The results of the archival research are not discussed as related research is limited. The controlled experiment of databases is not chosen as no other comparable databases with access control were found.

3.3 ARCHITECTURE

The following three sections of this chapter discuss the implementation of multi-dimensional access control. Figure 3 shows a diagram with a layered architecture. The application and database are existing systems showed with dashed lines. The three components in the middle implement the access control.

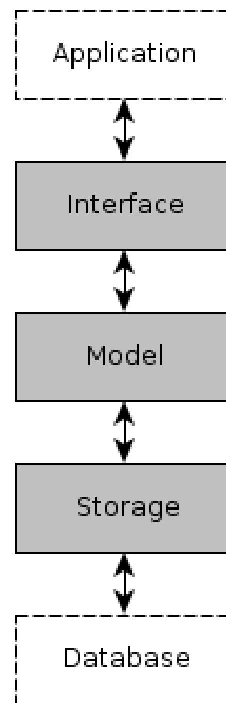

Figure 3: Layered architecture of solution.

The first component `Interface` (Section 3.4) is responsible for providing the developers the programming interface. They can give the users permissions to use the data of the database, to check the access rights and to delete the access rights. The interface provides the developers the functions and options for providing the access control solution without required knowledge of "lower" components.

The `Model` component (Section 3.5) is responsible for implementing role based access control for the multiple dimensions. It implements the interface and uses the storage layer to store the authorization information in the database. The model is independent of the storage layer to make it possible to support the three storage locations.

The `Storage` component (Section 3.6) provides the storage of the access rights for the model. Three storage locations are used: in a separate table, in a row and in each cell. Each storage location has the same interface and stores the same data.

The `Application` and `Database` are existing software components. The application (e.g. Hadoop MapReduce) could be an analysis application that uses a database. A layered solution prevents that unauthorized data can be accessed by an application. The middle layers filter the results for which the application has access. Making the model independent of the storage has as advantage that the throughput and data size of the storage locations can be compared.

For this work the RBAC model (see Section 2.1) is applied. It provides support for operations and can store both MAC and DAC models. RBAC describes which subjects may access an object. In this work, an object is considered a cell in a database, but limited to a dimension. For example, the rights of a row is stored for the object table/-family/qualifier/row. But when the right for a table are required, the object is table/. Thus when four dimensions have to be checked, four access rights are required for an object.

The implementation is programmed as an extension of Apache HBase. The access rights are stored in the same database as the data. The extension checks the user's permission for the read, write and delete operations. Read information that is not allowed is filtered out from the results. In essence a view is created that only shows the allowed data.

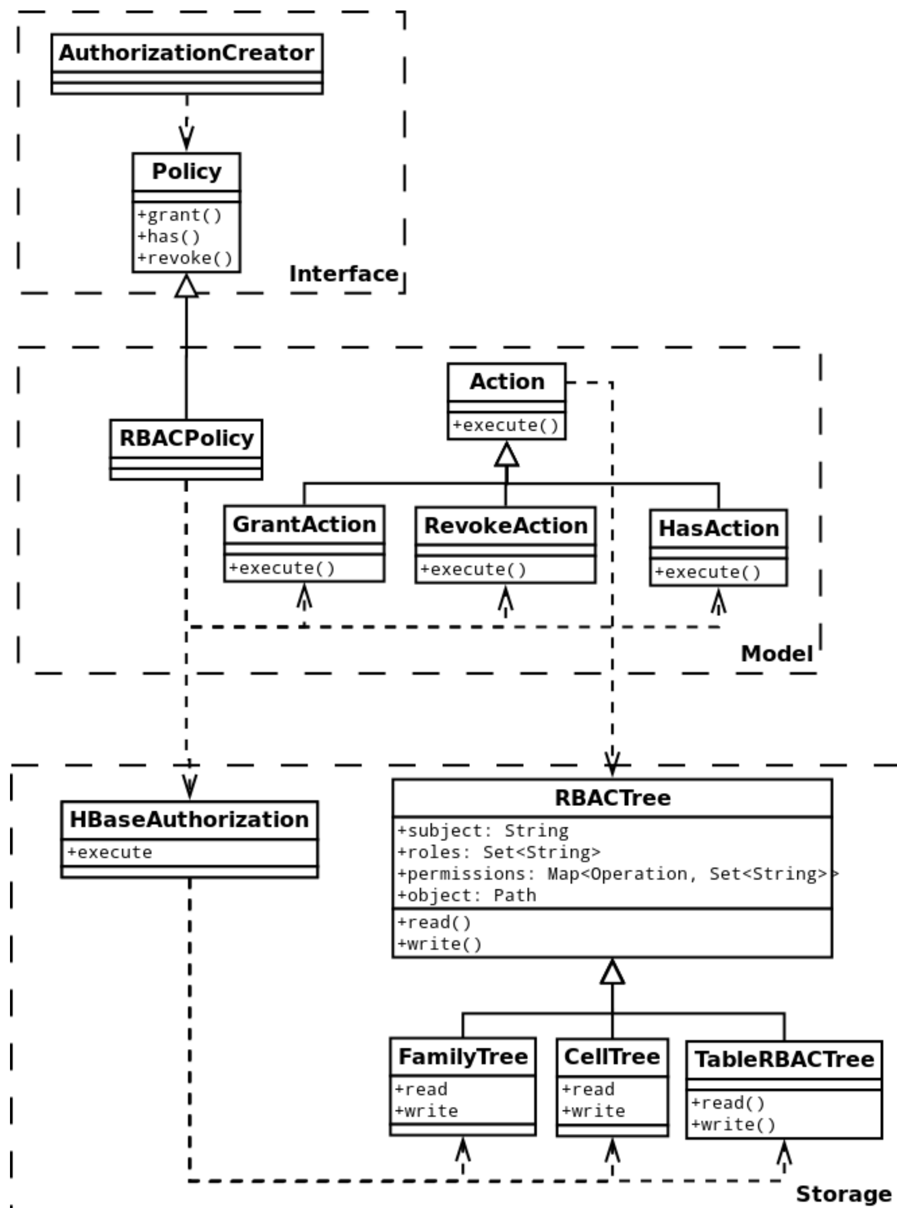

Figure 4: Class diagram.

However, such a view only works for viewing results. When calculations are based on this view, the results can differ for other users with different rights. This is also described as the Truman model [23] in Section 2.2. When using this model it is important that if operations are performed on this view, it is done by user(s) with the same permissions. Furthermore, the developers have to make sure that all data required is accessible to attain the correct results.

An high-level overview of the architecture is depicted in Figure 4. This class diagram shows a part of the functions and parameters for simplification. The architecture is split in an interface, model and storage part. A developer using the solution only has to use the functions of the interface.

The architecture components are explained below. Subclasses are described by the superclass in this listing.

- AuthorizationCreator (Interface): Acts as the interface for other applications to use the access control solution. Creates a `Policy` that implements the functions `grant`, `has` and `revoke`.
- Policy (Interface/Model): Responsible for applying the functions in multiple dimensions. It loads and caches the dimensionset from a special table and checks for each function which dimensions are required for processing. Can be extended to store MAC and DAC. The RBACPolicy implements the `Policy` and creates an `Action` object for the RBAC interface functions.
- Action (Model): Implements the logic for RBAC. Each action is for one dimension and uses the `RBACTree` for the data model.
- RBACTree (Storage): Contains the data required to implement the (RBAC) access control model: the subject, a list of roles for that subject, the permissions for that user and the path indicating the database cell. Contains a `read` and `write` function that is implemented by the three storage locations. The data loading and any optimization is the responsibility of the variant, as long as the required data is loaded. The `RBACTree` stores the permissions of one object in one dimension.
- HBaseAuthorization (Storage): Acts as an interface that chooses the required storage location. It uses the read function of a `RBACTree`, executes one of the `Actions` and writes the results to the database via `RBACTree`. For the table and family dimension, the access rights are always loaded from a table. For the column or row dimension, this is the table, row or cell.

3.4 APPLICATION INTERFACE

For this system the interface is build for the application developers that use a column-oriented database. Since application developers are used to build from source code, the interface should also act on this level. For the interface, the following keydrivers are identified.

- Usability: The interface should be easy to use for the developers. The functions and their parameters should be clear and the options should be easy changeable.
  Rationale: an easy solution makes it possible that developers use it faster to apply it than an other solution.
- Extensibility: The interface should be able to support new solutions, such as encryption.
  Rationale: it is likely that the solution is accompanied with more security options to increase the security of a database.

- Interoperability: The interface should be usable within existing development environments and the used frameworks as HBase and MapReduce.
  Rationale: the solution is likely only used if combined with other frameworks that require a database with multi-dimensional access control.
- Configuration management: The interface should provide options to optimize the performance for a certain dataset.
  Rationale: management of options makes it more easy for developers to adjust the settings to increase performance for a certain dataset.

The interface is the bus which the application programmer can use to apply access control on the data. For an application the interface of the access control solution is both the entrance point and exit point of a complex distributed authorization system. A developer does not have to worry about this system and requires only three the basic functions to use role based access control. The functions are:

- grant: gives the subject the rights to operate on the object
- has: checks if the subject has the rights to operate on the object
- revoke: deletes the subject the rights to operate on the object

Beside these basic functions a developer should also be able to specify the dimensions for which the access should be given and checked. In a column-oriented database such as HBase, the dimensions are: table, (column) family, (column) qualifier and row. The question is: how should a developer be able to specify these dimensions?

For this work four options are identified: using one dimension, an AND operator, an OR operator and free choice. These options are listed in Table 6 each with their description of the workings, advantages and disadvantages. Based on these options the interface implements the last option: free choosing. With this option a developer can freely choose which dimensions should be checked and which operators should be used. For example, a developer could give the following **dimensionset**:

Table || (Family & Qualifier)

This results in an access control solution where the user must have access to the table or both the column family and column qualifier. Furthermore, this solution requires the storage of access rights for each of these three dimensions. For example, when the following dimensionset is given by the developer:

Table || Row

Only two dimensions have to be stored: table and row.

<table>
<tr><th>Option</th><th>Description</th><th>Advantages</th><th>Disadvantages</th><th>Form</th></tr>
<tr><td>One dimension</td><td>Only gives the option to use a single dimension.</td><td>One permission for one dimensions has to be given.</td><td>Does not give the option to require access for multiple dimensions.</td><td>T
F
Q
R</td></tr>
<tr><td>Using AND</td><td>Usages the AND operator for all dimensions such that authorization for all dimensions are required per cell.</td><td>All dimensions can be changed.</td><td>Gives the option for multiple dimensions, but requires all dimensions always. Not one or two dimensions can be used.</td><td>T & F & Q & R</td></tr>
<tr><td>Using OR</td><td>Usages the OR operator for all dimensions such that authorization for all dimensions are required per cell, but a user has access if it has access to only one of these dimensions.</td><td>All dimensions can be changed.</td><td>Gives the option for multiple dimensions, but requires all dimensions always. Not one or two dimensions can be used.</td><td>T||F||Q||R</td></tr>
<tr><td>Free choosing</td><td>A developer can enter the dimensions using AND and OR operators to freely choose the dimensions. For example "F & (Q || R)" to require access for the family and the qualifier or row (using the brackets to avoid ambiguity).</td><td>Gives the option to choose one of the other options, or a combination.</td><td>Gives the developer more responsibility for choosing the dimensions.</td><td>Examples:
T || R
T || (F & Q)
(T & R) || Q
T & F & R</td></tr>
<tr><td colspan="5">T: Table, F: Family, Q: Qualifier, R: Row</td></tr>
</table>

Table 6: Comparison of four interface variants for multi-dimensional access control.

The dimensionset has both influence on the access control *rules* as the *storage* of access rights. The choice for a certain dimensionset is specified for a table at its creation. If for a certain dataset other dimensions are required, a separate table must be created. Supporting multiple dimensions within one table requires that the dimensions must always be stored within one cell, such that each cell has its own dimensionset. It is likely that this would be overly complex and probably not used. Therefore, the dimensionset is given per table.

The functionality mentioned so far is listed in pseudo-code in Listing 2. This code is a simplification of the actual Java source code. It gives an example of how the dimensions are set and how the permissions are granted, checked and revoked. Line 2 creates a policy based on the name and role of the user, and the operations, such as Read, Write and Delete. Line 5 until 10 show the creation of a factory and how the dimensions are set. This example uses the dimensions "r & (f || q)". This is according to the fourth option in Table 6, but with non-capital letters. For each dimension a different policy can be created. Finally, an authorizationCreator is created which makes it possible to reuse the dimensions. This prevent that for each grant the dimensions have to be given. Line 13 and 14 show the creation of a path. This path is required to specify a cell in a database which acts as the object within the subject-object relationship. A path is stored in the sequence of table, family, qualifier and row. Finally, line 15 until 18 show examples of the grant, has and revoke actions.

Listing 2: Pseudocode interface

```
1  // Create a policy
2  Policy policy(username, rolename, operations)
3
4  // Create an authorizationCreator
5  AuthorizationFactory factory;
6  factory.setDimension(table, "r&(f||q)");
7  factory.setPolicy(FAMILY, policy);
8  factory.setPolicy(COLUMN, policy);
9  factory.setPolicy(ROW, policy);
10 authCreator = factory.create();
11
12 // Examples of grant, has and revoke
13 Path path1(table, family, qualifier, row1)
14 Path path2(table, family, qualifier, row2)
15 authCreator.grant(path1);
16 authCreator.grant(path2);
17 hasAccess = authCreator.has(username, role, operations, path)
18 authCreator.revoke(COLUMN);
```

The has function as shown on line 17 is also used in an extension of HBase. This extension of HBase splits all retrieved query results and check for each cell if access is allowed. If not, the not allowed results (cells) are removed from the complete results.

Finally, several options can influence the behavior of the underlying system to increase the performance for specific situations. They are listed in Table 7 with a description and the default settings. The options are made available by the `AuthorizationFactory` class. The used benchmark uses two of these settings as shown in Section 4.2.

Option	Description	Default
autoflush	If true, submits access control information directly. If false, stores the given amount of megabytes locally before sending the queries. Reduces communication but increases time for one large query	true
result or list of results	Used for the function has. Before querying a get, appends the required queries for access control. Afterwards the function has retrieved the access rights without needing separate queries.	null (empty)
put/delete	Appends the grant or revoke rights to the given put or delete object. This results in less communications with the servers as less separate queries are required.	null (empty)
overwrite	Skips retrieving the old access control information. Useful when one knows before hand that for example a table is new. Should be used carefully as it overwrites any previous access control information.	false
tablecache and dimension-cache	Two caches for respectively storing the permissions for tables and storing the access control dimensions for performance improvement. The caches have no synchronization feature so a change in the caches on one server, is not known at another running server. If synchronization is required, caches should be disabled.	true and true

Table 7: Options available for performance improvements.

3.5 ACCESS CONTROL MODEL

The `grant` and `revoke` functions each use the specified dimensionset to check for which dimensions a function should be applied. For example, if the dimensionset is "r||q" it would grant the rights for the row and qualifier. The revoke works similar as the grant, but deletes the rights. The `has` function requires a more sophisticated solution as it has to incorporate binary operators i.e. the AND and OR operators.

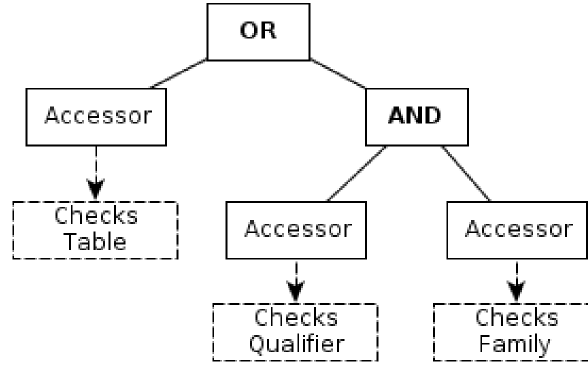

Figure 5: Example structure.

The checking of access rights can be visualized as a tree. In this case a small tree as the HBase database consists of four dimensions (table, family, qualifier and row). For example, if the dimensionset is "t | | (c & f)", the checking structure would be constructed as depicted in Figure 5. When executing, each accessor checks for a dimension as shown with the boxes with dashed lines. Each accessor returns the result and the end result is based on the binary result of the OR and AND operators. This end result states if a subject may operate on a given object.

The checking of access for tables, qualifiers and families all use the underlying RBAC model. This section will continue by formally describing what objects and subjects can do and what information should be stored. Many Axioms are tested using Java JUnit test classes in the implementation. The first subsection lists existing definitions from the literature. The second subsection continues with derived definitions that are used in the implementation.

3.5.1 *Existing definitions*

The RBAC (Section 2.1) model is used as the basis for multi-dimensional access control.

Definition 2. The $RBAC_0$ model [34] has the following components:

- U, R, P and S (users, roles, permission, and sessions respectively)
- $PA \subseteq PxR$, many-to-many permission to role assignment relation.
- $UA \subseteq UxR$, many-to-many user to role assignment relation.
- user: $S \rightarrow U$, function mapping each session s_i to the single user $user(s_i)$ (constant for the session's lifetime).
- roles: $S \rightarrow 2^R$, function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r|(user(s_i), r) \in UA\}$ (which can change over time) and session s_i has the permissions $\cup_{roles(s_i)}\{p|(p, r) \in PA\}$.

This work uses this model as follows. The PA and UA are stored in separate tables. The table location does this, while the family and cell location store the PA matrix in another way. The UA matrix is always stored in a separate table in this work and cached in memory when the system is active. The $S \to U$ is in this work managed by Kerberos. This system is responsible for the user authentication.

The last function links the components to convert a user's session to a set of permission. For this work this is not directly possible. Due to the large amount of rows and columns to secure, an object has to be given first to retrieve the rights. Ferraiolo and Kuhn [35] use the following formulations for RBAC:

- AR(s:subject) = {active role for subject s}.
- RA(s:subject) = {authorized roles for subject s}.
- TA(r:role) = {transactions authorized for role r}.
- exec(s:subject, t:tran)= true if and only if subject s can execute transaction t.

Ferraiolo and Kuhn [35] define four rules for RBAC, listed in Axiom 1 until 4. Each subject that can executing a transaction has an active role. This forces that subjects cannot execute a transaction without an active role and is called the *rule of role assignment*:

Axiom 1. $\forall s : subject, t : tran, (exec(s, t) \Rightarrow AR(s) \neq \emptyset)$.

Before a subject can do a certain action, it has to be checked if it the subject has an active role. This is shown in the next axiom and is called the *rule of role authorization*:

Axiom 2. $\forall s : subject, AR(s) \subseteq RA(s)$.

To relate the notion of roles with the notions of transaction, the following axiom shows that each subject executing a transaction has an active role with which it is allowed to do that execution. This is known as the *rule of transaction authorization*:

Axiom 3. $\forall s : subject, t : tran, (exec(s, t) \Rightarrow t \in TA(AR(s)))$.

Finally, the following rule (without name) does not bind the transaction to an object but uses an operation x e.g. read, write, delete. A function `access` checks if the subject has permission to do the given operation on the object.

Axiom 4.
$\forall s : subject, t : tran, o : object, (exec(s, t) \Rightarrow access(AR(s), t, o, x))$.

This rule thus does not only check the permission for a transaction, but splits the possible transactions in a number of operations.

3.5.2 *Extending definitions*

Based on the original definitions and axioms, the following additions are made. This section follows the conventions of Ferraiolo and Kuhn.

Definition 3. The multi-dimensional RBAC model has the following components:

- object(tb:table,f:family,q:qualifier,rw:row): {database cell}.
- DS(tb:table): {dimensionset for table t}.
- TAO(r:role, o:object): {transactions authorized for role r and object o}.
- TAC(r:role, o:object, d:dimensionset): {transactions authorized for role r and object o using the condition of dimensionset d}.
- D(o_1:object, d:dimensionset): {outputs object o_2 based on dimensionset d}
- exec(RA(s), o, x)= true if and only if subject s can perform operation x on object o

Instead of one role, a user can use multiple roles. In this model there is no usage for an active role. Furthermore, the permissions are based on an object. They can only be loaded when the object is known. For example, with the cell storage location the permissions are stored within a cell, therefore the object has to be known. To check one dimension, the dimensionset holds a single dimension. The D function can be used to convert and object to an object based on the required dimension. For example, the dimension 'q' leads to an object of the form 'table/family/qualifier/'. Based on Axiom 3, the following is the *rule of single-dimensional access*:

Axiom 5. $\forall s : \mathrm{subject}, t : \mathrm{tran}, o : \mathrm{object}, tb : \mathrm{table}, (\mathrm{exec}(s, o, t) \rightarrow t \in \mathrm{TAO}(\mathrm{RA}(s), \mathrm{D}(o, \mathrm{DS}(tb)))).$

This axiom combines Axiom 3 and Axiom 4 by making a transaction an operation. When the dimensionset includes multiple dimensions e.g. 'q&r' both dimensions have to be checked. This check is based on a condition which can be different for each table. The TAC analyzes this condition to check multiple dimensions and uses the *rule of single-dimensional access* for each single dimension. This is shown in the following *rule of multi-dimensional access*:

Axiom 6. $\forall s : \mathrm{subject}, t : \mathrm{tran}, o : \mathrm{object}, tb : \mathrm{table}, (\mathrm{exec}(s, o, t) \rightarrow t \in \mathrm{TAC}(\mathrm{RA}(s), o, \mathrm{DS}(tb)).$

This work compares three locations for the storage. Each storage location should provide the same functionality.

Definition 4. For comparison between storage locations, the following components are added:

- ST(tb:table): the storage location of the table t.
- TAOS(r:role, o:object, st:storagelocation): {transactions authorized for role r and object o using storage location s}.

Finally, each storage location should be able to store the same permissions. The table stores the table and family dimensions. The qualifier and row dimension can be stored in a table, row or cell. To force that all locations together form the entire set of permissions for an object, independent of storage location, followed is the *rule of storage similarity*:

Axiom 7. $\forall s : subject, o : object, tb : table$(TOAS(RA(s), o_t, table) + TOAS(RA(s), o_f, table) + TOAS(RA(s), o_q, ST(tb)) + TOAS(RA(s), o_r, ST(tb))) = TA

Based on the above model, the roles, permissions and operations have to be stored for each storage location. These implementations are discussed in the next section.

3.6 STORAGE LOCATIONS

The model requires a database to the store data of the `RBACTree`. To evaluate which storage location has the best performance, three storage locations for the access rights are proposed. The locations are listed below and depicted in Figure 6.

1. **Table**: All rights are stored in a separate table. This option is comparable with implementations in relational databases. It is included to compare it with the other two options. In a distributed database this table is stored on a separate machine. This location and the row location can be used with query extension (see Section 2.2 and the options at the end of Section 3.4 for more details).

2. **Row**: Not seen in other databases, but acts as an intermediate solution between the table and cell location. Based on the fact that the columns can vary for each row. This option stores the access rights for each dimension within a row in separate columns.

3. **Cell**: Stores the access rights for multiple dimensions within each cell (field in relational databases) together with the data. Comparable with implementations as Accumulo (see end of Section 2.3). This option requires changes to the HBase source code.

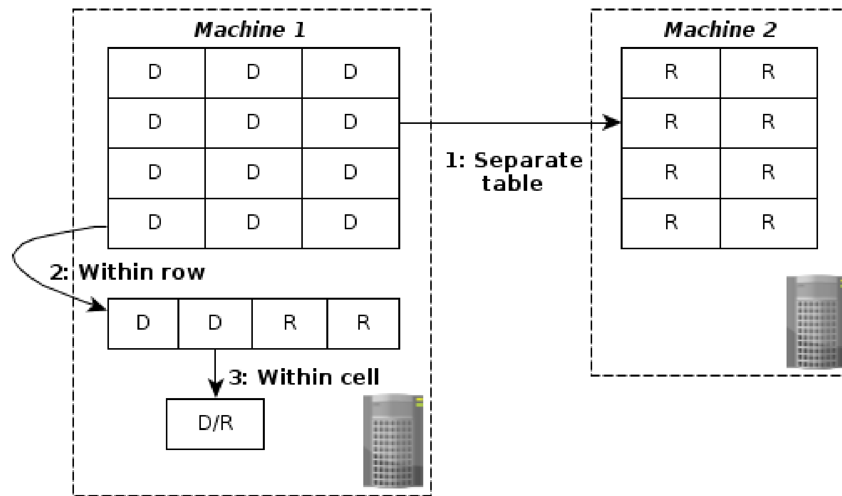

Figure 6: Three locations of the storage data. D stands for data and R stands for access rights.

To make a good comparison between the storage location, each location implementation stores the same information. Only the storage *location* may differ. The general rules for the locations are as followed.

- **Storage**: each location must store the object and the permissions. Each permissions consists of an authorized operation and required role.
- **Operations**: the permitted operations are stored as a number. For example: read=1 and write=2.
- **Permissions**: the list of permitted permissions are stored as a sequence separated by a comma and the operation and role are separated by a colon. For example: {1:developer,2:tester}.
- **Path**: the identification of a database value or cell is stored as a hierarchy of table/columnfamily/columnqualifier/row.

Beside the access rights storage that each location implements, two database tables exist for shared data that is required for all locations:

- **_security_dimensions_**: Stores for each table the dimensionset. Each table can use only one dimensionset.
- **subjects2Roles**: Stores for each subject the assigned roles.

The implementation details of all storage locations are discussed in the next subsections. For each location the examples used assume the dimensionset "q & r" for simplicity to show both dimensions are stored. The table and family dimension are always stored in the table. More information about the testing procedure can be found in Section 4.1.

3.6.1 *Table*

In the first storage location, all information is stored in a separate table. An example is shown in Table 8. This table shows the access rights for several objects in all dimensions and their permissions. In this table each `row` has the path as row key and the access rights in a separate column. For the dimensionset "t & f & q & r" each dimension has to be checked. For example, for the row table/personal/name/16439287 the other dimensions table/personal/name, table/personal and table have to be checked. The last two dimensions, table and family, are not shown in the example but stored in a similar way as for the qualifiers and rows.

Row	**Value**
table/personal/name	1:tester,3:finance
table/personal/name/16439287	2:tester
table/personal/address	1:tester,3:finance
table/personal/address/16439287	2:tester
table/finance/balance	2:tester
table/finance/balance/16439287	1:developer, 2:tester

Table 8: Example of table storage location.

The `value` column shows that the access rights are stored as a list where each operations has a list of roles. All subjects that have a role for an operation may perform that operation. This storage location is comparable with solutions in relational database described in Section 2.2. One major difference is that all rights are stored within a row. This is likely faster than using more tables to store the permission to role relationship as these tables can be spread over multiple machines. For autonomous systems or databases in which access rights can easily be cached this location can be quite fast. However, for large amounts of data all rights have to be retrieved from this table making it slow.

3.6.2 *Row*

The second location stores the information within the row in a separate column family. An example with sample data is shown in Table 9. This location does not require the storage of the table and family access rights, as that information is stored within the table location.

The access right values are stored in the same way as the table location. For each dimension the permissions are stored in the value of the special column starting with _s_. The rows with the qualifier rights start with _s_:_c_. The amount of security columns matches the

amount of normal columns (assuming that the rights for all columns are stored).

The columns must be identified with both the family name as the column name as two families can both store columns with the same name. The row is stored by the key _r_ with gives the access rights for the current row. Note that this location does not require multiple queries for using multiple dimensions. Assuming that all required security columns are part of the query based on the original columns in the query, only one query is required.

Furthermore, as the security columns are storing within a row, the columns are retrieved when the data is requested in one single query. The family location does exactly this by extending a query before executing the query in the database. Furthermore, it can add or delete rights by extending the HBase Put or Delete objects. These options are described in more detail in Section 3.4.

Row	**16439287**
personal:name	Jan Klaas
personal:address	Square 1
finance:balance	Euro 500,00
_r_	1:developer
_s_:personal_name	1:tester,3:finance
_s_:personal_address	2:tester
_s_:finance_balance	1:developer,2:tester

Table 9: Example of column family storage location.

3.6.3 *Cell*

The last location stores the access control information within each cell. Conceptually, an additional part is created within the key. Table 10 shows the new part securityInfo in bold. The remaining of this section discusses how this part is added.

row	family	qualifier	**securityInfo**	timestamp	keyType	value

Table 10: Conceptual new cell contents.

Each cell in HBase is essentially a byte array where all information is stored sequentially. The class responsible for storing this type is the KeyValue class in HBase. The contents of this cell are [36]:

byte[] keyValue = {int keyLength, int valueLength, int rowLength, byte[] row, int columnFamilyLength, byte[] columnFamily, byte[] columnQualifier, long timestamp, short keyType, byte[] value}

The `rowLength` until the `keyType` is considered the Key while the cell data contents are stored in the Value. Together they form the KeyValue pair. The int and shorts are converted to bytes to store them in a byte array. Note that the length of the column-qualifier is not stored. Using source-code analysis, the observation is that the columnQualifier length is calculated by subtracting the length of the timestamp and keyType of keyLength, as both are of fixed length. In HBase the new key with a part for the access rights, securityInfo in bold, would become:

byte[] keyValue = {int keyLength, int valueLength, int rowLength, byte[] row, int columnFamilyLength, byte[] columnFamily, byte[] columnQualifier, **int securityInfoLength, byte[] securityInfo,** long timestamp, short keyType, byte[] value}

However, this leads to one problem: how to calculate the position of `securityInfoLength`? The size of the `securityInfo` is stored in `securityInfoLength` so this cannot be calculated from the end as the securityInfo is of variable length. However, the `columnQualifier` length was also calculated from the end. This also does not work anymore and prevents calculation from the front, as the `columnQualifier` is also of variable length. To reuse the existing source code that calculates the lengths in HBase, the following key structure is used:

byte[] keyValue = {int keyLength, int valueLength, int rowLength, byte[] row, int columnFamilyLength, byte[] columnFamily, byte[] columnQualifier, **byte[] securityInfo, int securityInfoLength,** long timestamp, short keyType, byte[] value}

The `securityInfoLength` is placed *after* the `securityInfo`. Now the position of `securityInfoLength` can be calculated from the end similar as the column qualifier in the original key. With the `securityInfoLength` the start position of `securityInfo` can be calculated. Finally, by also subtracting both properties together with the timestamp and keyType, the position and length of the column qualifier can be calculated.

The KeyValue class has been changed to support this extra information in the key. Adjustments were made to other parts of the HBase source code to support this key. With this approach, no separate storage has to be used as the required access rights are stored within the cell. While retrieving data from the database the access rights are always available and require no additional queries. However, if all columns in a row require the same rights, duplicate information is stored in the cells in a row. The experiments in the next chapter show the consequences of this and of the other storage locations.

4 PERFORMANCE EVALUATION

Based on the implementation of multi-dimensional access control in HBase, the throughput and data size is tested with the Yahoo! Cloud Serving Benchmark. The tests are performed with the Amazon Elastic Computer cloud. The first four sections describe the design of the test. The final four sections show and discuss the results.

4.1 CONTROLLED EXPERIMENT

Building an experiment design requires to know what exactly has to be measured. Formally, it has to show the influence of changing one variable (independent variable) to one or more other variables (dependent variables) [37]. In the experiments only the storage location is changing and forms the independent variable (IV).

The performance is analyzed by measuring the throughput and data size. Both act as the dependent variables as they are dependent on the location. The throughput (DV1) is measured by counting the operations per second. The benchmark inserts data in the database as fast as possible. The duration of one action is the duration between an application request and received result from the database. The database size (DV2) is checked by retrieving the data size of the database on the file system. The data size is requested from the distributed file system after each completed benchmark. These variables in the context of other parts of the experiment are depicted in Figure 7. This diagram shows an application writing data to a database which uses access control and a certain storage location. The throughput and data size change dependent on the used storage location.

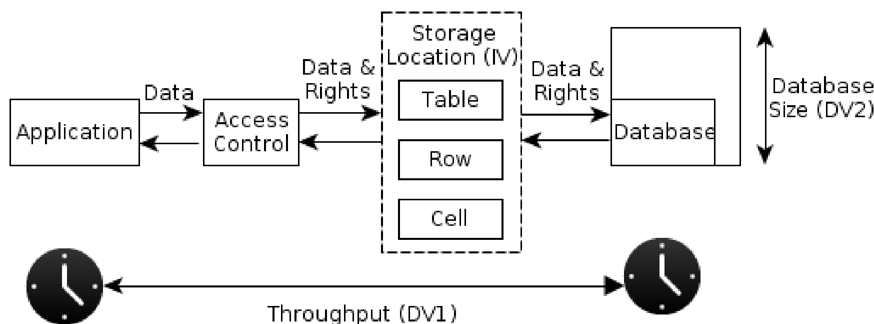

Figure 7: Overview of experiment design. The storage location is changed while measuring the throughput and database size consequences.

It is important that all other variables that can influence the throughput and data size are known and not changing. These are called the control variables (CV). Most control variables in this work are dependent on the used frameworks and environments. Therefore they are described after the description of data collection in Section 4.4. The database usage is dependent on the application. Some applications are write intensive whereas others are read intensive. Also different settings for access control are possible. Also the usage of rows and columns vary, dependent on the required security.

To cover these different aspects multiple experiments are performed with different configurations. Beside the storage location, this work uses the following configuration options: dimension, operation and scale. The types and settings are shown in Table 11.

First, the StorageLocation (IV) is the list of storage locations described in Section 3.6. The set of locations is extended with a None location. This type has security disabled. It provides the performance differences with security enabled. Second, the dimensions describe which rights to check. The access rights of the column or the rights of a row. A certain storage location might be faster for one dimension than another storage location. The table and family dimension are not included. There are not many tables and families and these can be cached in memory. HBase recommends not using more than three families [1] (Chapter 6.2. On the number of column families). Third, all experiments are performed for both read and write operations. Finally, the data is added in two scale options. The row type inserts a large number of rows while the amount of columns are small. The column inserts a large number of columns while the amount of rows are small.

These large amount of tests should provide valuable insight in the behavior of several configurations. One storage location might be the best. The best location could also be dependent on the configuration. In this case the experiment results give the possibility to evaluate which solution works best in which situation. The next section describes how the experiments are performed.

Type	**Set**
StorageLocation	Table, Family, Cell, None
Dimension	Column, Row
Operation	Read, Write
Scale	Rows, Columns

Table 11: All variables used for the experiments. Each combination leads to a configuration for the test.

4.2 DATA COLLECTION

Apache HBase is used for the database implementation. The primary reason of choice is that it is used by Kalooga, for which this work is performed. However, other arguments support this choice. First, this database already implements access control for tables, families and columns but for small amounts. To identify the database users, it uses Kerberos for authentication [1] (Chapter 8) [38]. This system provides HBase with the user's name based on a secure authentication model. Furthermore, row or cell based security is an open issue. It is also possible to extend this database with new functionality as it is open-source. Also the documentation is extensive and ensures that the database can be configured and run on a distributed system.

HBase can run on a single computer using the normal file system. A distributed setup of HBase requires the Apache Hadoop Distributed File System (HDFS). HDFS stores the files on a file system that is spread over multiple machines. It is accessible for all connected machines but it requires data transfer if the data is stored on another machine. Both HBase and Hadoop consist of a number of services that run on multiple machines. These services are explained in Table 12. Each service must be appointed to a computer.

Service	Subservice	Description
HDFS	Namenode	Manages the Datanodes. Maximum of 1.
	Secondary Namenode	Backup Namenode. Not used in this work.
	Datanode	Stores the data. Each physical machine can act as 1 Datanode.
HBase	HMaster	Manages the HRegions. One master can run at a time. Leader election is used when multiple masters are up as backups.
	HRegion	Stores data in the database and uses local filesystem or distributed HDFS (Datanode).
	Zookeeper	Manages communication and small data storage for Master and HRegion Servers. To reach a quorum an uneven number of servers is appropriate.

Table 12: Summary of Hadoop and HBase services [1].

The distribution of these services differs per situation. A small cluster can have one Master and Zookeeper, while a large cluster has multiple Masters and Zookeepers. For this work, the fault tolerance is not important as for the relative short duration of the experiment no machines are expected to crash. If during a test a machine crashes the test is performed again with another machine. Furthermore, the man-

aging services require relative little resources. As a result, the Master, Zookeeper and Namenode can all run on the same machine. Each other machine in the HBase cluster runs both a Datanode and HRegion, as this is the common setup. Each HRegion reads and write the data on the local Datanode to reduce data communication. This work has redundancy disabled and therefore each HRegion stores its data only on its local Datanode. For simplicity, the machine with the Namenode, Zookeeper and HMaster is called the Master in this chapter. The machines with the Datanode and HRegion are called Regions.

The experiment is performed on a rented computer cluster at the Amazon Elastic Compute Cloud (EC2). This service of Amazon supports configuring one instance as a template and clone the instance to several other machines. Figure 8 shows the setup of the cluster. The distributed system consists of 14 machines, of which 3 benchmark servers, one master server and ten region server. On each benchmark server the YCSB benchmark (explained hereafter) runs that sends data to or requests data from the database. All services run on Large Instances as configured by Amazon. Each instance consists of 7.5 GiB memory, 4 EC2 Compute Units (about 4 x 1.0-1.2 GHz 2007 Opteron processor) and high I/O performance. Appendix B shows the list of servers in Amazon's EC2 Console for this test.

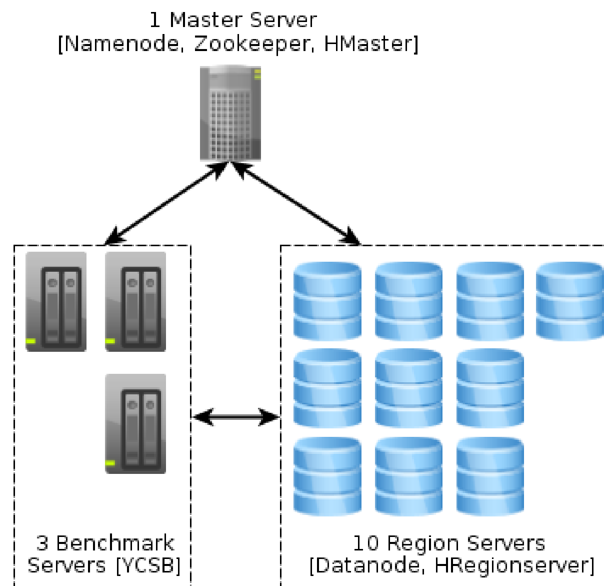

Figure 8: Setup of the distributed test system

A benchmark is required to measure the throughput and data size. It could be custom implemented for this work. However, the entire benchmark code should then be given and validated for correctness. Furthermore, one benchmarks exists that supports HBase. This work uses the Yahoo! Cloud Serving Benchmark (YCSB) [39]. The benchmark is created to compare the performance of several databases including HBase.

It tests the database by performing operations while measuring the throughput. The throughput is the amount of successful operations per second. This benchmark was chosen because it is open-source, used by other researchers and published results are available that can be used for validation. It does not compare the data size. This is calculated by requesting the file size of the HDFS. This can be done with the command *"./hadoop/bin/hadoop fs -ls /hbase/"*.

The YCSB benchmark has been extended to support multi-dimensional access control. Listing 3 shows a function (simplified) to create a table. Line 4 until 6 show existing code for the table creation using the HBase interface. The following two parts are new. In line 9 and 10 YCSB is granted permission to create a table. This thesis focuses on which storage location is the fastest, but the complete work also supports access control for table insertions and deletions. In production environments the rights to create a table should be managed by an administrator. This is done from the benchmark for simplification. In line 13 until 18 the data is split over 10 servers. The distribution is by numbers as the data entries all have a number as a key. The data distribution is further discussed in Section 4.4. When this function completes, a table has been successfully created.

Listing 3: Changes to the Yahoo! Cloud Serving Benchmark to support multi-dimensional access control (creating table).

```
1  private void createTable(String table, String role, String
       username, String columnFamily) throws IOException {
2
3      // create table
4      HTableDescriptor tableDescr = new HTableDescriptor(table);
5      tableDescr.addFamily(new HColumnDescriptor(columnFamily));
6      HBaseAdmin hbaseAdmin = new HBaseAdmin(HBaseConfiguration.
           create());
7
8      // grant YCSB to create a table
9      Policy policy = new RBACPolicy(username, new
           DatabaseOperations(Operation.CREATE, Operation.DROP,
           Operation.ENABLE, Operation.DISABLE));
10     policy.grant(role, new BytePath(table.getBytes()), new
           BytePath(table.getBytes()), new GrantOptions(false, true,
            null), "table");
11
12     // force the distribution of the data over 10 servers
13     byte[] bytesLow = new byte[] {'1'};
14     byte[] bytesHigh = new byte[] {'9'};
15
16     hbaseAdmin.createTable(tableDescr,
17         Bytes.add("user".getBytes(), bytesLow),
18         Bytes.add("user".getBytes(), bytesHigh), 10);
19 }
```

The `createTable` function of the previous part is called by the `init` function shown in Listing 4 (simplified). In line 3 until 6 the user name, dimensions and storage location (storageType) are loaded from the properties. YCSB uses a configuration file for all settings, of which these three are added. It is important that the correct user name is entered. The user executing the benchmark must be given in the configuration file and authenticated with Kerberos used by HBase. In line 9 an object is created which stores the operations. This object is managed internally by the AuthorizationFactory. In line 12 until 18, the model is initialized. In this example each dimension is given access rights. The user with the given user name may read, delete and write data for all of the dimensions table, family, qualifier and row. Furthermore autoflush is deleted and overwritePermissions enabled (see the options at the end of Section 3.4 for more details). In line 19 and 20 the dimensionset is stored in the database. The dimensionset is loaded and cached on each HBase server. It is used for checking the rights when operations are performed for the given table. Finally, on line 21 all the rights are stored with the `create` function.

Listing 4: Changes to the Yahoo! Cloud Serving Benchmark to support multi-dimensional access control (initializing model).

```
1  public void init() throws DBException {
2      // Retrieve from properties
3      username = getProperties().getProperty("username");
4      createTable("usertable", "tester", username, "family");
5      dimensions = getProperties().getProperty("dimensions");
6      storageType = getProperties().getProperty("storagetype");
7
8      // Set operations
9      TableOperation tableOperations = new TableOperations(
           Operation.READ, Operation.DELETE, Operation.WRITE);
10
11     // Initialize model
12     AuthorizationFactory af = new AuthorizationFactory();
13     af.setPolicy(DimensionStorage.Dimension.TABLE, new RBACPolicy
           (username, tableOperations));  // t
14     af.setPolicy(DimensionStorage.Dimension.FAMILY, new
           RBACPolicy(username, tableOperations)); // f
15     af.setPolicy(DimensionStorage.Dimension.COLUMN, new
           RBACPolicy(username, tableOperations)); // c
16     af.setPolicy(DimensionStorage.Dimension.ROW, new RBACPolicy(
           username, tableOperations));     // r
17     af.autoflush(false);
18     af.overwritePermissions(true);
19     new DimensionStorage().remove("usertable".getBytes());
20     af.setDimension("usertable".getBytes(), dimensions,
           storageType);
21     authCreator = af.create();
22 }
```

YCSB was configured to store data sequentially. From a total amount of data elements, the data was evenly spread over the benchmark servers. Each benchmark server sends the same amount of data to the database. Furthermore, a counter is used for the key creation such that each data element from the total element collection is unique. YCSB is configured such that each data element has a value consisting of 100 bytes.

4.3 PROCESS

To do a quantitative controlled experiment requires a deterministic benchmark. Both Hadoop as HBase have background tasks such as splitting tables and redistribution. If the database contents would not be deleted, such activities on the data of the previous benchmark can influence the performance results of the next benchmark. A counter-argument may be that the system should be tested with this configuration. However, to validly select the best storage location requires that other variables be controlled. Therefore, the database contents are deleted and the database services restarted after each read test. The contents after a write test are not deleted as the read test depends on the data of the write test.

For all experiments a Bash script is executed from one of the YCSB machines that communicates with the other machines using Secure Shell (SSH). This makes it possible to startup, control and shutdown all servers from only a single machine. The machine that runs this script also runs an YCSB benchmark as the other YCSB machines. The high-level procedure of the experiment coded in the script is as followed and depicted in Figure 9.

Before all experiments the Namenode of the HDFS is formatted and started on the Master. All Regions remove their HDFS directory on disk and start their Datanode. For each experiment, the following steps are taken. First the HMaster, Zookeeper are started on the Master, and the HRegions on each Region. Then the benchmark configuration for that particular test is send to all benchmark servers. After that, the benchmarks are started and the results are stored on the local benchmark machines. Then on the main machine a loop starts that waits for a few seconds and checks if the results are written yet on all benchmark machines. When completed, all results are retrieved and aggregated to one result list including all properties, throughput and data size results. After each test all HBase services are shutdown. After all tests the HDFS is shutdown.

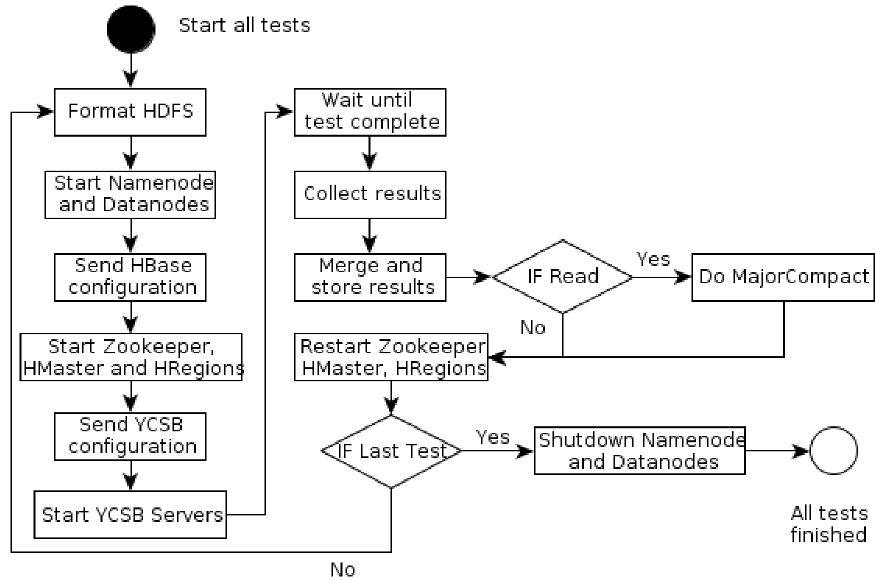

Figure 9: Activity diagram of test procedure.

4.4 PERFORMANCE PROPERTIES

During the testing of the experiments several problems were identified. Most problems affecting the quality of the results were influencing the performance negatively. They made the performance too fluctuating or leaded to significant performance loss. By changing several configuration options the performance was increased or more stable. These options can be seen as the control variables for this experiment. The most important issues and their solutions are explained next to ensure the reliability of the results.

- **Data distribution:** There are essentially two possibilities for splitting the data among the HRegions. First, letting HBase manage the distribution. Second, before the test manually assigning the distribution. The first option is the default one, where HBase splits the data to other machines after one HRegion has reached a certain data size. While this normal behavior, this could interfere with the experiment. When data is redistributed it requires more hard disk and network usage. Therefore the throughput could be fluctuating during the test. The second option requires that the distribution of keys in known in advance. For this benchmark this is possible as the keys are based on a counter which has a known range when the test starts. This work uses the second option as all databases servers are started from the beginning of the experiment. This is done by splitting the keys in ten parts and assigning each part to a HRegion.

- **Amazon stability:** Second, a problem is that the benchmark has only run once due to budget constraints. This could have negative consequences for the reliability of the results. Because the instances are virtual and run on machines by Amazon together with other instances, no performance guarantees can be given. Other instances of other business could also have a peek in the hard disk or network usage. If this happens with one location but not with another, it could influence the final conclusion which storage location is the best. Therefore plots were created of each test displaying the throughput. When a test has a relative large fluctuation it is performed again. For reference, the tests were also performed on a single (non-Amazon) machine.

- **Compaction:** To reduce the amount of files on the file system, HBase performs a compaction regularly [36]. This task takes several database files and merges them to one. This can increase performance significant as less files have to be read. This process is called a small compaction and done during run-time. A major compaction does also other maintenance tasks and is mostly started manually during off-peak moments. During the test a significant fluctuating was observed during the small-compaction. The throughput was none for several tests and on numerous moments. To make the throughput more stable, the small compaction was disabled and performed during major compaction. This is done after each write test to ensure good performance during a read test. The major compaction itself is not part of the tests. To disable compaction, the settings *hbase.hstore.compactionThreshold* and *hbase.hstore.blockingStoreFiles* were both set to 99999999.

- **Balancing:** The final problem was the correct balancing of the regions. Each HRegion stores its database data in one or more regions. A region is part of a table. Each table can be stored in various regions over multiple machines. A load balancer in HBase balances the amount of regions among the HRegions []. It occurred frequently that not all HRegions were assigned a region. This happens when the cluster is started but not all HRegions are booted yet. As a result some HRegions store multiple regions while others store no region at all. To overcome this issue the HMaster settings are changed to check if the cluster is balanced. If not, the regions are reassigned until the cluster is balanced. This does not influence the results as this is done during the load time of the cluster. The change was made by setting the *hbase.balancer.period* property to 10000. Then the HMaster checks the balance every 10 seconds. During a test in this work the booting period is longer that 10 seconds and therefore the cluster is always balanced before a test starts.

The observations of the results are given in the following sections. All raw results that were exported from the bash script are listed in Appendix A. This appendix contains all the results and settings for each experiment. The evaluation in the following sections is the explanation and interpretation of the raw results. To limit the length of this thesis, the results of the single machine are not discussed. They are comparable with the distributed system results, only the table location performs better due to less network communication as the data and the access rights are stored on the same machine.

4.5 THROUGHPUT EXPERIMENTS

The throughput test consist of 32 experiments. The results are compared by storage location. Two types of tests are performed: reading and writing. The dimensions used are qualifier and row. Each row consists of 10 column qualifiers for the row test and 2000 column qualifiers for the column test. The results are the sum of the three separate YCSB server results. As described in Section 4.1 the None storage location should have equal results for each type of test. With all tests the qualifier and row dimension are compared. But in the test with this storage location, access control is disabled. The None type stores no rights and therefore the qualifier and row should have the same results for this type. Thus with this control group, also the stability of the Amazon cluster can be validated. Due to the performance fluctuations of the Amazon cluster as described in Section 4.4, this chapter focuses on the significant performance differences.

Figure 10 shows the results of writing using the qualifier and row storage location. The table is the slowest location while the family and the cell location have a comparable speed. The storage location none has the largest speed. When comparing the dimensions the qualifier is slower for the table and family location, while for the cell and none type they are almost equivalent.

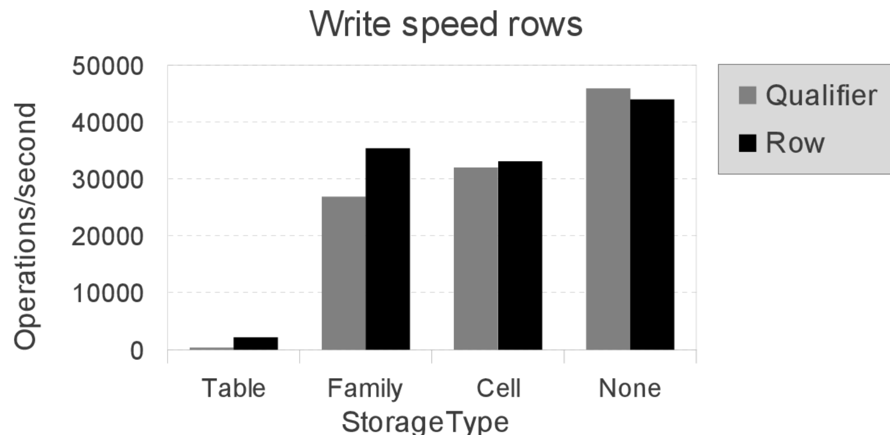

Figure 10: Throughput results for writing large amount of rows.

The second set of results has a similar construction as the write experiments, but now for the read operations. This test uses the scanner of HBase and not a single read. A single read depends to much on the network latency. During testing the results had smaller differences because they were limited by the network speed, which is similar for all locations. With the scanner 1000 rows at a time are retrieved, giving a better indication of a large read operation. The results of YCSB were multiplied with this factor, as one scan of 1000 rows is calculated as one operation in YCSB.

The read results are depicted in Figure 11. The table location is again the slowest for both dimensions. The family and cell location are faster. The none type reads the data the fastest. As with the first test, the row dimension is as fast as the qualifier for both the none and cell location. The family location is significantly faster for the row while for the table and cell locations the difference is minimal.

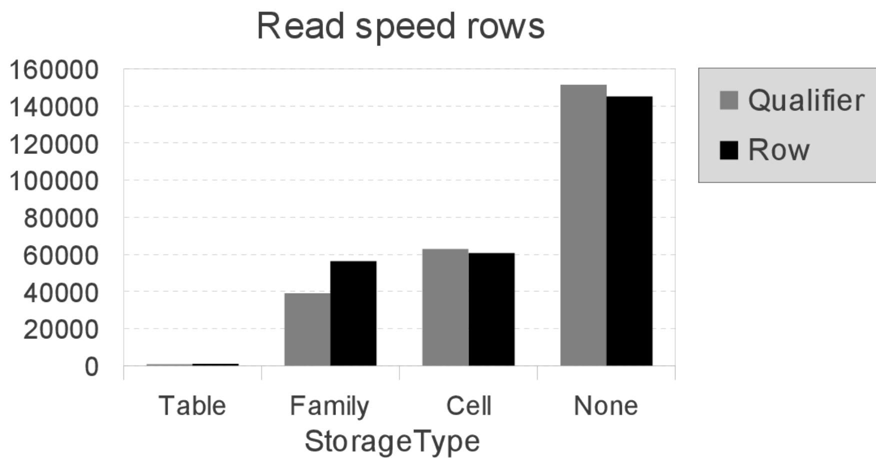

Figure 11: Throughput results for reading large amount of rows.

Similar tests are also performed for large amount of columns. The write results are depicted in Figure 12 and the read results are shown in Figure 13. Compared with the tests for rows, the amount of operations per second is much lower. YCSB counts the amount of operations as the number of rows processed. If each row counts 4000 columns, it is still calculated as one row. Therefore the amount of rows per second is lower while a larger amount of columns are present in a row.

The write test for columns shows a comparable result with the row write test. The table is the slowest whereas the family and cell are almost as fast. The read results show a much larger difference. The family in this test is also very slow, while the cell is much faster than the table and family location. In all of the tests until now, the cell has the highest throughput.

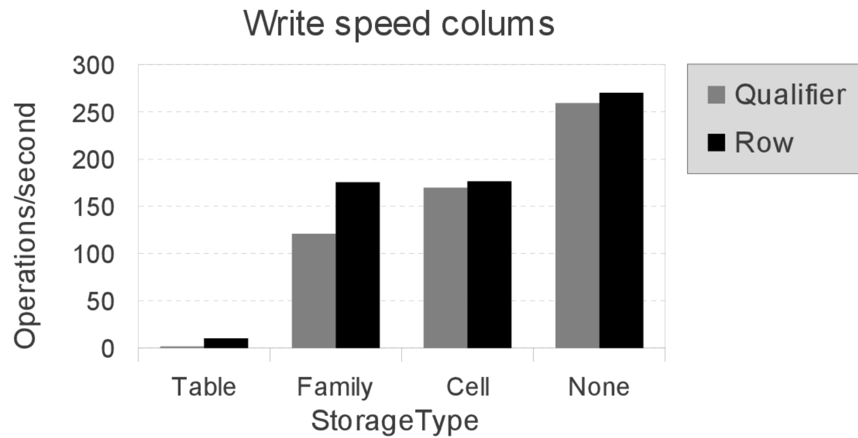

Figure 12: Throughput results for writing large amount of columns.

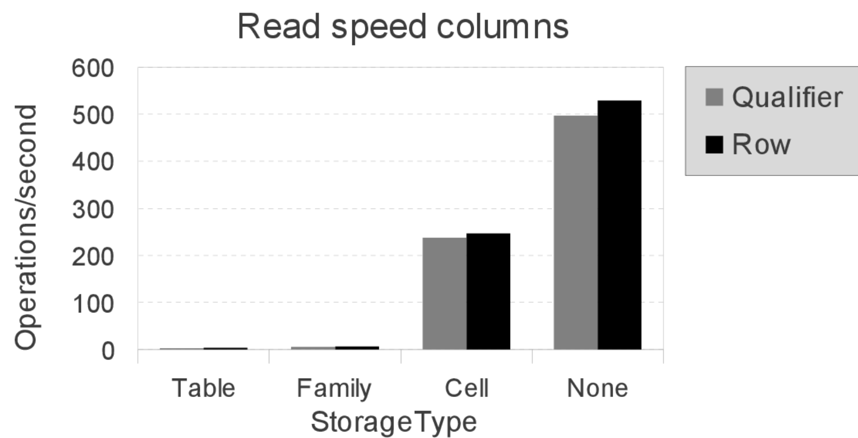

Figure 13: Throughput results for reading large amount of columns.

4.5.1 *Discussion*

A significant correlation between the throughput and storage locations can be seen in the results. In both the read and write test the table location is the slowest. The family is faster and the cell is the fastest. The experiments demonstrate that storing the rights in a separate table as in relational databases is not a good option. However, storing data more local to the data is faster. The family and cell locations have comparable results. However the family location is much slower when reading a large amounts of columns.

When comparing the throughput of the none type with the results of another YCSB paper [33], similar results can be found. They tested one YCSB server with six HBase machines and measured a throughput around 10.000 operations per second. This none type in this work writes 45.000 operations per second as shown in Figure 10. A factor of 4.5 is a reasonable factor when using three YCSB machines and ten HBase machines. This shows that the quality of the results are positively validated against existing throughput results.

However, all locations are not as fast as the none type. This makes sense as more operations are required to retrieve a cell. For each cell the access rights have to be checked. Furthermore, only the authorized results may be shown. It seems that the cell and partially the family location have reached their maximum performance and that the throughput is ceiled by the security model's operations. Improving this model can further improve the throughput results.

There is an important performance difference between the qualifier and row dimension. In both the read and write tests for rows and columns, the cell location can operate the qualifier equally fast, while the other locations are better for rows. Only Figure 13 does not clearly show this as the throughput is very low for the table and family locations. The table can store rows better than qualifiers. When storing per qualifier it has to store ten qualifiers for each row. When controlling rows, only one item has to be stored. As the benchmark operations are counted per row it shows a better result for rows. A similar explanation can be given for the family location. When controlling the columns, the family has to store the rights for each column. The row type requires only one column. While the row type seems faster, it also provides no granularity for individual columns.

4.6 Data size experiments

The second type of experiments are based on the size of the database after each set of write operations. The results for both a large amount of rows and columns are shown in Figure 14 and Figure 15, respectively. The histogram shows that the table has the largest row data size, while the qualifier is significant smaller. The family requires a bit larger storage for qualifiers than the cell, while the cell requires more than without access control.

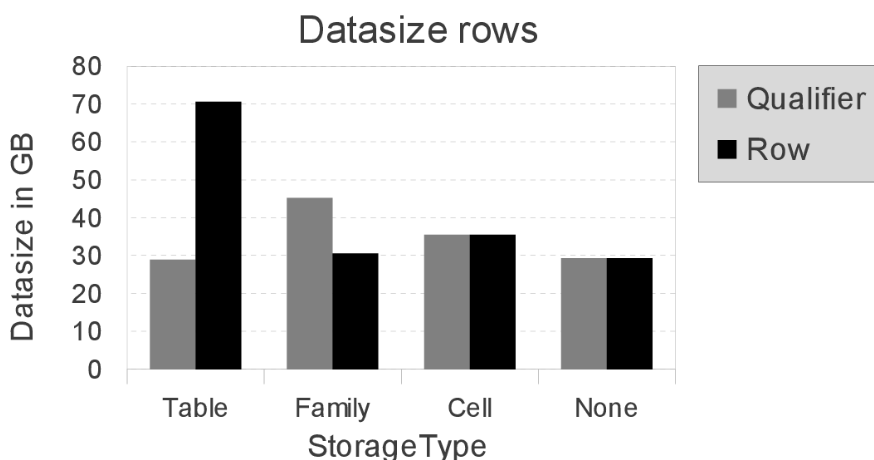

Figure 14: Data size after write test for large amount of columns.

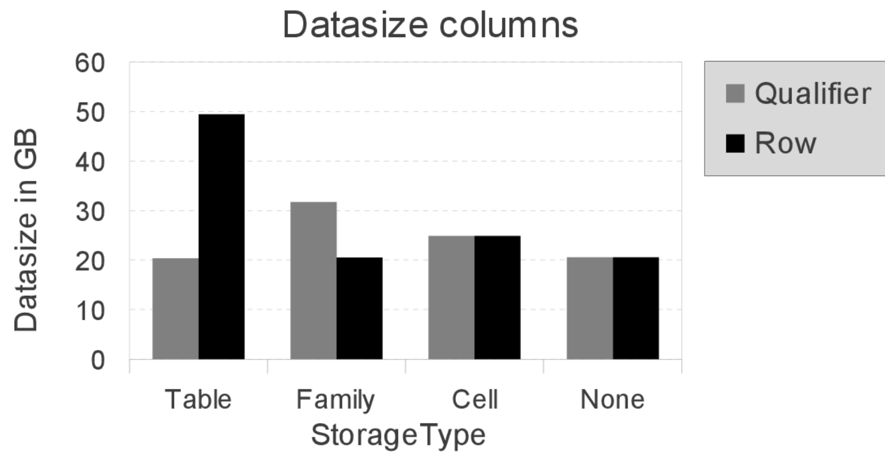

Figure 15: Data size after write test for large amount of rows.

The family location requires less space for rows than the cell location. The none variant requires almost the same amount for both storage locations. Similar results are found after the write test for columns. However, the data sizes are smaller than after the row test because less data is written. The duration of the tests were longer and therefore a smaller amount of data is used for the benchmark.

4.6.1 *Discussion*

The data size experiments show that the table location requires little space for the qualifier dimension. While a table location stores all qualifiers, they do not change for each row. The table location can reuse columns as they are stored on a single place. This does not apply for the family location. This location stores the columns within each row and requires more space. When the columns would differ per column family, the table location would require more space.

The row dimension however requires the storage of access rights for each row, resulting in a large database for the table location. The family location is far more efficient as it can store the rights within each row. The cell location results lie between the two family tests. Both cell tests have equivalent results. This seems reasonable as the rights are stored within each cell. The qualifier and row have the same data size, as the same information is stored within each cell. Only in one case the rights are for the qualifier dimension, while in the other for the row dimension. The consequences of these results are discussed in the next section.

4.7 EVALUATION

Applications that need access control may require different settings for the security model. Therefore, several configurations were used to extensively test each location in different situations. Based on the interpretations of the results, there is no location that is the best in all configurations. However, each location has particular characteristics that make it usable for specific situations. The table is only usable for small quantities of access rights, such as tables and families. These rights can be retrieved once and cached in memory. The family location is very fast for row level security, but not for columns. For each column, another column has to be retrieved from the database. The cell location has a slightly smaller throughput for rows than the family location, but has much better performance for columns. A disadvantage however is that the disk usage increases, but only slightly. While the tests show a data size increase of 20%, the values were only 100 bytes. In datasets with larger values this percentage likely further decreases. Byun and Li [26] also show this with a labeling scheme in relational databases.

To store a merged dataset, the cell location seems as the best option to store the access rights. If only rows have to be secured, the family location is used best. When using a large amount of columns, the cell location is also better. If only columns or both rows and columns have to be checked, the cell location has also the best performance. Chapter 1 showed that a merged dataset requires that multiple dimensions have to be checked. In this situation the cell location is used best to control the rows and columns. Because the rights are stored each cell, no additional queries are required to retrieve the rights. When data is loaded from the database, the database extension that checks the rights can always retrieve the rights from the cell making it very fast.

The difference between this solution and relational databases is significant. This work shows that storing the rights in a separate table is not efficient. In a distributed database this table is stored on another machines. The communication required between the machines to link the data with its access rights reduces the performance significant. The table location is usable for low performance environments or a small number of rights, but not for large systems where throughput is important. The proposed solution with the cell storage location differs from solutions in relational databases and supports a column oriented structure, uses local data, multi-dimensional, scalable and implements the Truman model. It stores the rights in each cell supporting the location of columns and is the nearest to the data. The interface supports providing access in tables, column families, column qualifiers and tables. It supports the Truman model by filtering the authorized data from the query results.

When comparing this work with the Accumulo database discussed in Section 2.3, both store the access rights in the cell. This work showed that the cell location is indeed the best direction if such a granularity is required. However, the HBase extension uses role based access control which supports granting access for specific operations. Furthermore, users can be coupled by roles on a central place and therefore the user to role relationship be managed without changing the access right's data. Accumulo cannot do this and requires that if the rights change, all data with these rights are updated. Furthermore, while Accumulo stores rights based on users, this solution provides no interface to store the rights for roles in the dimensions of tables, families, columns and rows.

4.8 LIMITATIONS

Measuring the performance is dependent on the applications requirements and background processes. The performance can be measured even better by running the benchmark with larger datasets for longer durations. Furthermore, the performance can be evaluated with different applications. The analysis of data may also take time and further reduce the performance overhead of access control. The results so far do show that for database intensive applications, the cell location is the best solution for storing a merged dataset.

It is important to define a correct policy before using the solution. When storing the access rights in a central database it is less difficult to change the rights of a role. When storing the rights in each cell the access rights are distributed over the entire database. For example, if the permitted list of operations of a role has to be changed concerning a part of the dataset, this entire dataset has to be edited. This can be done with a distributed MapReduce application. However, only the roles and operations are stored. The relation between users and roles are stored on a central place and can easily be edited.

Due to the scope of this work, it did not use real databases to retrieve data from for data mining. Retrieving the access rights from another database may not be as easy as assumed. Perhaps databases use also other access control techniques such as certificates. Furthermore, special applications or scripts may be required to retrieve the access rights from the database to store them in a merged database. But as long as the access control information can be converted to role based access control, the proposed solution can store the access rights for multiple dimensions.

5 CONCLUSION

One technique of data mining is the extraction of data from other databases. When they use access control to protect the data, the access rights should also be stored in the merged dataset to protect the sensitive data in the results. For distributed column-oriented databases this work proposes multi-dimensional access control. The performance in this database is dependent on physical location of the access rights. Therefore three storage locations are compared to provide the best access control performance. To answer the primary research question, the research subquestions are answered below.

1. *How does multi-dimensional access control work?*
   A developer can give access rights in HBase for tables, families, columns and rows. The model uses role based access control as the foundation. An HBase extension is developed which filters the not allowed data from the results. This work concentrated on the usage of a large number of columns and row, as the number of tables and families is limited. The access rights and options are configurable through an API. Three storage locations are used. The table location stores the information in a separate table and the family location within a row. The cell location stores the access control information in each cell. While the table and row location use the existing HBase interface, the cell location required changes to HBase's core source code.

2. *How are the performance results of the storage locations compared?*
   The locations are compared by throughput and data size using a controlled experiment with security disabled. All locations implement a shared interface such that each location stores the same access rights. The Yahoo! Cloud Serving Benchmark was used to test the locations of the access control model. The Amazon Elastic Compute Cloud is used to deploy the experiment on a distributed system.

3. *How do the implementations of these locations affect the throughput and database size of a distributed NoSQL database?*
   The experiments show that the table location has the worst performance. The row location is second. It works as good as the cell location for rows, but as bad as the table location for large amount of columns. The cell location has the highest throughput and smallest overhead of database size. While each location has several advantages and disadvantages, the cell location is the best location for storing a merged dataset.

The answers on the subquestions lead to the answer of the primary research question:

***Research question**: How does the storage location for multi-dimensional access control influences the performance of a distributed column-oriented database?*

The throughput is highly dependent on the physical distance between the data and its access rights. The closer the access rights to the data are, the higher the throughput. However, it also increases the database size as it produces more redundancy. However, this overhead decreases with larger values in the database. This thesis shows that storing the access rights in each cell for fine-grained access control has the best performance.

Based on this result, column-oriented databases that contain sensitive data from data mining can use multi-dimensional access control described in this thesis for better security. The implementation uses the cell location and protects the data by only showing the information for which the user has access. Not allowed data is filtered out from the results. Developers can restrict access to specific columns or rows for better management or to restrict data for specific elements.

5.1 FUTURE RESEARCH DIRECTIONS

Providing fine-grained access control in a database makes it more secure to show the results of analyses when they also use sensitive data. However, the security of the database can be further improved.

Application developers or administrators that have access to the hard disks can still read the data. It is more difficult to read due to the raw storage, but sensitive information may still be retrieved. One solution based on this work is cell-based encryption. By encrypting the data based on the authorized users of a cell, other users cannot access that data. This is a step further than access control as it also prevents access when the hard drives of a system are compromised. The choice of the encryption scheme is challenging as it can impact performance significant. Another question is: which data to encrypt? Beside the encryption of values, the row or column keys could also be encrypted as it can leak (parts of) sensitive data. However, this makes it more difficult to search a database based on a key.

Beside column-oriented databases, other NoSQL databases exist such as key-value stores and document stores. These databases have a different structure than column-oriented databases. But this work could be generalized such that they could also implement multi-dimensional access control. The performance can be evaluated and could lead to a better understanding of how access control can work efficiently with different data structures on distributed systems.

BIBLIOGRAPHY

[1] Apache HBase. The Apache HBase™Reference Guide. http://hbase.apache.org/book/book.html, Access July 2012.

[2] Robin Hecht and Stefan Jablonski. NoSQL Evaluation - A Use Case Oriented Survey. *International Conference on Cloud and Service Computing*, pages 336–341, December 2011.

[3] Laurent Bonnet, Anne Laurent, Michel Sala, Bénédicte Laurent, and Nicolas Sicard. REDUCE, YOU SAY: What NoSQL can do for Data Aggregation and BI in Large Repositories. *22nd International Workshop on Database and Expert Systems Applications*, pages 483–488, August 2011.

[4] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, February 2010.

[5] Lior Okman, Nurit Gal-Oz, Yaron Gonen, Ehud Gudes, and Jenny Abramov. Security Issues in NoSQL Databases. *International Joint Conference of IEEE TrustCom*, 2011.

[6] Eduardo B. Fernandez. *Handbook of Data Intensive Computing, Chapter 16, Security in Data Intensive Computing Systems*. Springer, 2011.

[7] Daniel J. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Computer Society Technical Committee on Data Engineering*, 32(1):3–12, March 2009.

[8] Rasmus Paivarinta and Yrjo Raivio. Applicability of NoSQL Databases to Mobile Networks: Case Home Location Register. *Cloud Computing and Services Science*, pages 225–242, 2012.

[9] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, March 1997.

[10] Ming-Syan Chen and Philip S. Yu Jiawei Han. Data Mining: An Overview from a Database Perspective. *IEEE Transactions on knowledge and data Engineering*, 8(6):866–883, December 1996.

[11] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From Data Mining to Knowledge Discovery in Databases. *American Association for Artificial Intelligence*, pages 37–54, 1996.

[12] Mark Stamp. *Information Security - Principles and Practice*. John Wiley & Sons, Inc., 2006.

[13] Christopher Metz. AAA PROTOCOLS: Authentication, Authorization, and Accounting for the Internet. *IEEE Internet Computing*, pages 75–79, November-December 1988.

[14] Matt Bishop. *Computer Security - Art and Science*. Pearson Education, Inc., 2003.

[15] James B.D. Joshi, Walid G. Aref, Arif Ghafoor, , and Eugene H. Spafford. Security Models For Web-Based Applications - Using traditional and emerging access control approaches to develop secure applications for the Web. *Communications of the ACM*, 44(2):38–44, February 2001.

[16] Matunda Nyanchama and Sylvia Osborn. Modeling Mandatory Access Control in Role-Based Security Systems. *Database Security*, pages 129–144, 1995.

[17] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000.

[18] Oracle. Oracle Database Security Guide - Using Virtual Private Database to Implement Application Security Policies. http://docs.oracle.com/cd/B19306_01/network.102/b14266/apdvpoli.htm.

[19] Microsoft. Implementing Row- and Cell-Level Security in Classified Databases Using SQL Server 2005. http://technet.microsoft.com/en-us/library/cc966395.aspx.

[20] A. Jangra, D. Bishla, Komal Bhatia, and Priyanka. Functionality and Security Analysis of ORACLE, IBM-DB2 & SQL Server. *Global Journal of Computer Science and Technology*, 10(7):8–12, September 2010.

[21] Sybase. Adaptive Server Enterprise 15.5, System Administration Guide: Volume 1, Managing User Permissions, Using row-level access control. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc31654.1550/html/sag1/CDDBCEEG.htm.

[22] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. Fine Grained Authorization Through Predicated Grants. *IEEE 23rd International Conference on Data Engineering*, pages 1174–1183, April 2007.

[23] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 551–562, June 2004.

[24] Govind Kabra, Ravishankar Ramamurthy, and S. Sudarshan. Redundancy and Information Leakage in Fine-Grained Access Control. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–144, June 2006.

[25] Faiz Currim, Eunjin (EJ) Jung, and Insoon Jo Xin Xiao. Privacy Policy Enforcement For Health Information Data Access. *Proceedings of the 1st ACM International Workshop on Medical-grade Wireless Networks*, pages 39–44, May 2009.

[26] Ji-Won Byun and Ninghui Li. Purpose Based Access Control for Privacy Protection in Relational Database Systems. *The International Journal on Very Large Data Bases*, 17(4):603–619, July 2008.

[27] Hong Zhu and Kevin Lü. Fine-Grained Access Control for Database Management Systems. *Proceedings of the 24th British national conference on Databases*, pages 215–223, July 2009.

[28] Dan Pritchett. BASE: An Acid Alternative. *ACM Magazine Queue - Object-Relational Mapping*, 6:48–55, May/June 2008.

[29] Jaroslav Pokorny. NoSQL Databases: a step to database scalability in Web environment. *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, pages 278–283, December 2011.

[30] Jason Schlesinger. Cloud Security in Map/Reduce. Accessed August 2012.

[31] Michael A. Enescu. Towards Scalable Security: On the Scalability of Security in Large-Scale Software Systems. http://blogs.ubc.ca/computersecurity/files/2012/04/MEnescu_paper.pdf.

[32] Apache Accumulo. http://accumulo.apache.org/, Accessed on May 2012.

[33] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++ : Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. *ACM Symposium on Cloud Computing 2011*, October 2011.

[34] Ravi S.Sandhu. Role-based Access Control. *Advances in Computers*, 46:237–286, 1998.

[35] David F. Ferraiolo and D. Richard Kuhn. Role-Based Access Controls. *15th National Computer Security Conference*, pages 554–563, October (1992).

[36] Lars George. *HBase - The Definitive Guide*. O'Reilly Media, Inc., 2011.

[37] Roger E. Kirk. *Experimental Design*. O'Reilly Media, Inc., 1982.

[38] B.C. Neuman. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32:33–38, 1994.

[39] Brian F. Cooper, Adam Silberstein, Erwin Tam, and Russell Sears Raghu Ramakrishnan. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, June 2010.

A

APPENDIX A - DETAILED RESULTS

The next two tables show the detailed results of the tests. The first table shows the results for the distributed system tests. The second table describes the single system tests. The last table is not discussed in the thesis as the focus is on distributed systems. However, this table is included such that other testers can compare the results also on a single system.

Distributed System

rows	qualifiers	read	insert	dim.	storageType	duration(ms)	throughput(ops/sec)	storesize(byte)	startup	shutdown
180000	10	0	1	q	table	1913275	282.2534603661	260199680	1	0
360	10	1	0	q	table	1708474	0.6322296305	260199680	0	1
180000	10	0	1	r	table	251034	2155.3510669946	635570709	1	0
360	10	1	0	r	table	1432072	0.7542453423	635570709	0	1
900	2000	0	1	q	table	1801057	1.4992322557	261360424	1	0
360	2000	1	0	q	table	354136	3.0497752018	261360424	0	1
900	2000	0	1	r	table	265442	10.2084805235	635203034	1	0
360	2000	1	0	r	table	321048	3.3864848437	635203034	0	1
10000000	10	0	1	q	family	1117144	26881.6957400272	22612402283	1	0
20000	10	1	0	q	family	1540262	38.9539568278	22612402283	0	1
10000000	10	0	1	r	family	849645	35322.5370998772	15280921382	1	0
20000	10	1	0	r	family	1065469	56.319234321	15280921382	0	1
30000	2000	0	1	q	family	746447	120.6490923531	13597666087	1	0
300	2000	1	0	q	family	152896	5.9321146995	13597666087	0	1
30000	2000	0	1	r	family	512837	175.5920884651	8782981216	1	0
300	2000	1	0	r	family	143317	6.363108919	8782981216	0	1
20000000	10	0	1	q	cell	1863007	32209.083949439	35578944555	1	0
48000	10	1	0	q	cell	2293420	62.8174195347	35578944555	0	1
20000000	10	0	1	r	cell	1817516	33045.1182326286	35561635018	1	0
40000	10	1	0	r	cell	1981260	60.6537870443	35561635018	0	1
70000	2000	0	1	q	cell	1561754	169.5861744416	24872380113	1	0
70000	2000	1	0	q	cell	884480	237.4206442183	24872380113	0	1
70000	2000	0	1	r	cell	1190242	176.5987872204	24871943883	1	0
70000	2000	1	0	r	cell	851686	246.5659089034	24871943883	0	1
20000000	10	0	1	q	none	1307926	45885.1625521019	29352409186	1	0
48000	10	1	0	q	none	954718	151.491612075	29352409186	0	1
20000000	10	0	1	r	none	1365055	43955.8940862383	29335111089	1	0
40000	10	1	0	r	none	841519	145.0957269484	29335111089	0	1
70000	2000	0	1	q	none	810564	259.169614867	20509778471	1	0
70000	2000	1	0	q	none	423285	496.5233687164	20509778471	0	1
70000	2000	0	1	r	none	777341	270.1444926605	20509308819	1	0
70000	2000	1	0	r	none	397222	529.0065195544	20509308819	0	1

Single System

rows	columns	read	insert	dim.	storageType	duration(ms)	throughput(ops/sec)	storesize(byte)	startup	shutdown
180000	10	0	1	q	table	897709	200.5104103891	260144803	1	0
36	10	1	0	q	table	186357	0.1931776107	260144803	0	1
180000	10	0	1	r	table	89125	2019.6353436185	635268004	1	0
36	10	1	0	r	table	219549	0.1639725073	635268004	0	1
900	2000	0	1	q	table	751085	1.1982665078	261315459	1	0
36	2000	1	0	q	table	51834	0.6945248293	261315459	0	1
900	2000	0	1	r	table	88783	10.1370757915	634736596	1	0
36	2000	1	0	r	table	58411	0.6163222681	634736596	0	1
2000000	10	0	1	q	family	282976	7067.730125523	4505054350	1	0
4000	10	1	0	q	family	580200	6.892450879	4505054350	0	1
2000000	10	0	1	r	family	230266	8685.5983948998	3046593838	1	0
4000	10	1	0	r	family	458186	8.7278965311	3046593838	0	1
10000	2000	0	1	q	family	255027	39.2076133115	4501041236	1	0
100	2000	1	0	q	family	222358	0.4452279657	4501041236	0	1
10000	2000	0	1	r	family	204048	49.0031757234	2912077878	1	0
100	2000	1	0	r	family	6499	15.2331127866	2912077878	0	1
2000000	10	0	1	q	cell	201611	9920.0837255904	3536031030	1	0
4000	10	1	0	q	cell	219530	18.2161891313	3536031030	0	1
2000000	10	0	1	r	cell	199144	10042.9739284136	3536061957	1	0
4000	10	1	0	r	cell	243497	16.4232002858	3536061957	0	1
20000	2000	0	1	q	cell	431870	46.3056012226	7090089692	1	0
20000	2000	1	0	q	cell	512087	39.0519579681	7090089692	0	1
20000	2000	0	1	r	cell	460378	43.4382181599	7090023198	1	0
20000	2000	1	0	r	cell	530617	37.6882007173	7090023198	0	1
2000000	10	0	1	q	none	157991	12658.9362685216	2902495427	1	0
4000	10	1	0	q	none	79548	50.271534168	2902495427	0	1
2000000	10	0	1	r	none	152344	13128.1704563357	2911800689	1	0
4000	10	1	0	r	none	79793	50.1171781986	2911800689	0	1
20000	2000	0	1	q	none	277058	72.1798323817	5845671788	1	0
20000	2000	1	0	q	none	235532	63.0515592634	5845671788	0	1
20000	2000	0	1	r	none	284319	70.3364882403	5845390651	1	0
20000	2000	1	0	r	none	300087	66.6406742045	5845390651	0	1

B

APPENDIX B - AMAZON INSTANCES

Figure 16 shows the management console of the Amazon Elastic Compute Cloud during testing. The first server was not used. The YCSB1 until YCSB3 servers are the benchmark servers. The 10 Region servers are the HBase database servers. The Master server runs the management tasks of the complete system.

Name	Instance	AMI ID	Root Device	Type	State	Lifecycle	State Transiti	Security Grou	Elastic IP
YCSBOLD	i-b9bdcdc	ami-3d4ff254	ebs	m1.large	stopped	normal	Client.UserInit	default	-
YCSB3	i-29143d5	ami-485fe521	ebs	m1.large	running	normal		default	-
YCSB2	i-2b143d57	ami-485fe521	ebs	m1.large	running	normal		default	-
YCSB1	i-43381f3f	ami-a28832cb	ebs	m1.large	running	normal		default	-
Region	i-1b49666	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-0d49667	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-0f496673	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-1549666	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-1149666c	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-1349666f	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-1749666	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-0310377f	ami-aa8933c3	ebs	m1.large	running	normal		default	-
Region	i-1f496663	ami-0664de6f	ebs	m1.large	running	normal		default	-
Region	i-1949666	ami-0664de6f	ebs	m1.large	running	normal		default	-
Master	i-0110377	ami-aa8933c3	ebs	m1.large	running	normal		default	-

Figure 16: Amazon instances.