Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

December 2013

An Access Control Model for NoSQL Databases

Motahera Shermin
The University of Western Ontario

Supervisor
Dr. Sylvia Osborn
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

© Motahera Shermin 2013

Follow this and additional works at: https://ir.lib.uwo.ca/etd

 Part of the Databases and Information Systems Commons, and the Information Security Commons

Recommended Citation
Shermin, Motahera, "An Access Control Model for NoSQL Databases" (2013). *Electronic Thesis and Dissertation Repository*. 1797.
https://ir.lib.uwo.ca/etd/1797

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.

An Access Control Model for NoSQL Databases

(Thesis format: Monograph)

by

Motahera Shermin

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Motahera Shermin

Abstract

Current development platforms are web scale, unlike recent platforms which were just network scale. There has been a rapid evolution in computing paradigm that has created the need for data storage as agile and scalable as the applications they support. Relational databases with their joins and locks influence performance in web scale systems negatively. Thus, various types of non-relational databases have emerged in recent years, commonly referred to as NoSQL databases. To fulfill the gaps created by their relational counter-part, they trade consistency and security for performance and scalability. With NoSQL databases being adopted by an increasing number of organizations, the provision of security for them has become a growing concern.

This research presents a context based abstract model by extending traditional role based access control for access control in NoSQL databases. The said model evaluates and executes security policies which contain versatile access conditions against the dynamic nature of data. The goal is to devise a mechanism for a forward looking, assertive yet flexible security feature to regulate access to data in the database system that is devoid of rigid structures and consistency, namely a document based database such as MongoDB.

.

Keywords

NoSQL, Access Control, RBAC, Document database, Security.

Acknowledgments

I humbly state my gratitude to the people who have guided me and contributed in making this thesis possible. First and foremost, I acknowledge my profound sense of reverence towards the Almighty for granting me the ability and confidence to successfully complete this thesis.

I express my deepest veneration for my supervisor and mentor, Professor Dr. Sylvia Osborn for guiding me all through the course of my graduation work. Her vast knowledge and deep understanding of different databases and access control mechanisms, her continuous guidance, scholastic and moral support as well as her patience throughout the entire period of my studies have been the constant catalyst. It has been a great pleasure and privilege for me to have her as my supervisor.

I recognize that this thesis work would not have been possible without the Western Graduate Research Scholarship and express my gratitude to Western University.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

1 Introduction

The IT industry has come a long way since Web 2.0 when the nature of computing started to shift from standalone software to a distributed service model and network models replaced the internet as a development platform with desktop applications migrating to cloud services. Since then it has been a constant race to develop the next best computing concept to serve the ever-growing data mass, ever-evolving traffic patterns, and constant high demand for performance. The market share of relational database management systems (RDMBSs) is still the largest when it comes to storing application data consistently regardless of the nature of the data or application. But the nature and size of data is emerging as the major factor when designing an application today.

Applications now run distributed over many servers, they get dynamically scaled on demand and even the delivery model has been enjoying new approaches with the Cloud. In contrast, RDMBSs have gotten less attention in terms of new approaches and as a result are becoming a less obvious choice as data storage. To fill this gap, new types of databases (DBs) have emerged called NoSQL (Not Only SQL) databases. These new databases come equipped with horizontal scaling, high availability, low cost, and high performance.

NoSQL databases present new storage architectures that are favorable to unrestrained growth of data and traffic. These databases facilitate devising parallel algorithms (i.e. MapReduce) designed to efficiently process the distributed data. The advent of NoSQL is leading the approaches and methods of storing and analyzing information rich user data to a brand new dimension. NoSQL exercises least amount of control over data where its relational counterpart maintains rigid control over the structure and occurrence of data. Relational systems use metadata to precisely control data type and heavily synchronized processes to maintain data integrity. NoSQL uses loosely synchronized processes to scatter data across many systems with little to no maintenance of consistency or integrity

constraints but at the same time it provides almost no support for security at the database level.

The main promoters of NoSQL databases are Web 2.0 companies such as Amazon, Twitter, LinkedIn and Google which are challenged with unprecedented data, operation volumes under tight latency constraints and massive infrastructure needs. NoSQL databases represent the much needed data storage evolution in enterprise application architecture, continuing the evolution of the past twenty years. In the 1990's, vertically integrated applications gave way to client-server architecture, and with Web 2.0, client-server architecture transcended to three-tier web application architecture. In parallel, the demands of web-scale data analysis added map-reduce processing into the mix and data scientists started trading transactional consistency for incremental scalability and large-scale distribution. The NoSQL movement emerged out of the later constellation [34].

Martin Fowler [24] states that NoSQL is an accidental neologism. NoSQL is an observation and aggregation of common characteristics, lacking any prescriptive definition. The evolution in data storage was motivated by the need to support large volumes of data with database running on clusters. Relational databases are not designed to run efficiently on clusters. Fowler aggregates the following common characteristics of NoSQL databases in his book.

- Not using the relational model
- Running well on clusters
- Open-source
- Built for 21st century web estates
- Schemaless

NoSQL, "Not Only SQL", is not a replacement for relational databases. These databases are developed to run on clusters consisting of commodity computers and therefore are designed to be distributed and failure tolerant. To achieve this, database architects have to step away from the ACID properties and make different trade-offs with transaction management, query capabilities and performance. NoSQL databases are usually designed to fit the requirements of most web services; most of them are schema-less systems that allow users to change data attributes at run time and bring their own query languages.

The NoSQL security model is for the databases to be buried deep within an organization and behind other systems. The security model expects the database to reside in a secure environment, protected by firewalls. The irony is that it's the web applications or social media applications which have adopted the NoSQL databases and have made it popular. Having an open front end, universally accessible from any web browser makes development easier, but the open user-interface concept conflicts with the behind-closed-door security model. This security concern is the main driving force in this thesis.

Access control is a fundamental security technique in systems in which multiple users share access to common resources. It is the process of stating and enforcing security policies that determine whether a subject (e.g., process, computer, user, etc.) is allowed to perform an operation (e.g., read, write, update, delete, etc.) on an object (e.g., a tuple in a database, a table, a file, a service, etc.). This mechanism maintains the subject's permissions (rights to carry out an operation on an object) in a system in order to achieve the desired level of security.

There are several access control models. Among those, role based accesses control (RBAC) models are most widely mentioned as they provide systematic access control security for Enterprise solutions. One of the main advantages of RBAC over other access control models is the ease of its security administration [6]. RBAC enables an organization to model its security mechanism to closely match the individual business process. RBAC models are policy neutral [26]; they can support different authorization policies, including mandatory and discretionary policies, through the appropriate role configuration.

The recent paradigm shift in application architectures such as cloud computing, distributed file system and non-structured data store imposes novel challenges for authorization that are not addressed by the well-established RBAC models [9]. Researchers have proposed several RBAC extensions to address an ever increasing scope of security requirements [2, 18, 32, 6, 10, 29, 23, 21, 21]. Most of these extensions are conceived for a particular system or business process. These approaches come up short when a different system comes into concern.

The increasing popularity of NoSQL databases and the large volume of user related sensitive information stored in these databases raises the concern for the confidentiality and privacy of the data and the security provided by these systems. NoSQL data models are flexible and can be designed according to application need or context. A context can be used to define the parameters for which an operation can be executed in a system. Context typically is application dependent. Usually, RBAC components are limited to user, role, permission and constraint. The classic RBAC model cannot support enforcing a security policy for contextual information such as state of data or application interface, let alone an unstructured data model such as seen in NoSQL. In this thesis we use one of the RBAC extensions to define an abstract security model for a specific NoSQL database type, a Document Database.

We start with a brief overview of NoSQL and RBAC. Then in Chapter 4, we discuss the architecture and security concerns of one of the popular document databases. Based on this discussion we present an abstract model in Chapter 5. The rest of the thesis is organized as follows. Chapter 6 contains the evaluation of the model against a sample internal enterprise system where traditional RBAC fails. We investigate the authorization requirements imposed by this system in this chapter. Chapter 7 contains conclusions and Future work. Comparison of SQL with NoSQL databases is given in Appendix A.

Chapter 2

2 Basics of NoSQL Databases

The intent of this chapter is to discuss common concepts, techniques and patterns related to NoSQL databases along with several classes of NoSQL and an individual NoSQL product. In recent years a number of NoSQL databases have been developed mainly by practitioners and web companies to fit their specific requirements regarding scalability performance, maintenance and feature-set. Some of these databases are conceived from the idea of Google BigTable or Amazon's Dynamo or a combination of both. Because of the variety of these approaches and the overlaps regarding the nonfunctional requirements and the feature-set, it could be difficult to get and maintain an overview of the non-relational databases. So there have been various approaches to classify NoSQL databases, each with different categories and subcategories.

2.1 NoSQL: Classifications

In his presentation "NoSQL is a Horseless Carriage" [33], Yen suggests a taxonomy that can be found in Table 1.

Table 1: Classifications – NoSQL Taxonomy by Stephen Yen [33]

Term	Database
key-value-cache	Memcached Repcached Coherence Infinispan EXtreme Scale Jboss Cache Velocity Terracoqa
Key-Value-Store	keyspace Flare Schema Free RAMCloud
Eventually-Consistent Key-Value-Store	Dynamo Voldemort

	Dynomite SubRecord Mo8onDb Dovetaildb
Ordered-Key-Value-Store	Tokyo Tyrant Lightcloud NMDB Luxio MemcacheDB Actord
Data-Structures Server	Redis
Tuple Store	Gigaspaces Coord Apache River
Object Database	ZopeDB DB4O Shoal
Document Store	CouchDB Mongo Jackrabbit XML Databases ThruDB CloudKit Perservere Riak Basho Scalaris
Wide Columnar Store	Bigtable Hbase Cassandra Hypertable KAI OpenNeptune Qbase KDI

Similarly to the classifications mentioned above, Rick Cattel classifies different NoSQL databases based on their data model [5] as shown in Table 2.

Table 2: Classifications – Categorization by Rick Cattell [5]

Category	Database
Key-value Stores	Redis Scalaris Tokyo Tyrant Voldemort Riak
Document Stores	SimpleDB CouchDB MongoDB Terrastore
Extensible Record Stores	Bigtable HBase HyperTable Cassandra

In his presentation [28] at codemesh 2010, Ben Scofield of Viget Labs gave generic introduction to NoSQL databases along with a categorization of different NoSQL databases. The presentation contained a short comparison of classes of NoSQL databases by nonfunctional categories as well as ratings of their feature coverage. Popescu [22] summarizes Scofield's ideas in his blog as presented in Table 3.

Table 3: Classifications – Categorization and Comparison by Scofield and Popescu [22, 28]

	Performance	**Scalability**	**Flexibility**	**Complexity**	**Functionality**
Key-Value Stores	high	high	high	None	Variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

The classification used for the purpose of this thesis is Document Store.

2.2 NoSQL: Principle concept

This section gives an overview of some basic concepts in the NoSQL design model. The CAP (consistency, availability and partition tolerance) theorem introduced by Eric Brewer [4], refers to the three properties of shared-data systems namely data consistency, system availability and tolerance to network partitions. This concept is widely adopted by prominent Cloud vendors as well as the NoSQL faction. This theorem basically states that current databases are better at consistency than availability and wide-area databases can't have both but this favors the side of NoSQL and influenced the design of non-relational databases.

Brewer mentions a tradeoff between ACID and BASE systems and proposed to select one or the other for individual use-cases: if a system or parts of a system have to be consistent and partition-tolerant, ACID properties are required and if availability and partition-tolerance are favored over consistency, the resulting system (i.e. NoSQL) can be characterized by the BASE properties. ACID database (RDBMS) is transaction based and it leverages data consistency and persistence principles from developers' responsibility. ACID transactions provide the following assurances:

Atomicity: All of the operations in the transaction will complete, or none will.
Consistency: The database will be in a consistent state when the transaction begins and ends.
Isolation: The transaction will behave as if it is the only operation being performed upon the database.
Durability: Upon completion of the transaction, the operation will not be reversed. [9]

The BASE approach, according to Brewer, forfeits the ACID properties of consistency and isolation in favor of "availability, graceful degradation, and performance" [4]. The acronym BASE is composed of the following characteristics: **B**asically **A**vailable, **S**oft-state, **E**ventual consistency.

Brewer contrasts ACID with BASE as illustrated in Table 4, yet considers the two concepts as a spectrum instead of alternatives excluding each other. So an application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known-state state eventually [14].

Table 4: ACID vs. BASE [4]

ACID	BASE
Strong consistency Isolation Focus on "commit" Nested transactions Availability? Conservative (pessimistic) Difficult evolution (e. g. schema)	Weak consistency – stale data OK Availability first Best effort Approximate answers OK Aggressive (optimistic) Simpler! Faster Easier evolution

2.3 NoSQL: Challenges

NoSQL database system takes a step away from the ACID principle of the relational database to ensure either partition or availability. According to Brewer's [4] CAP theorem, a distributed database system will either trade availability to make sure the database is consistent or it can favor availability over consistency when the database is partitioned; but neither at the same time. This implies that if consistency and availability are of primary concerns then it is achieved by not introducing partition in the database system. Most of the NoSQL databases give priority to availability over consistency.

NoSQL databases are not mature enough compared to RDBMS and thus face several challenges and concerns. One of the hurdles the application designers face is overhead and complexity created by the different query processing language or API in each NoSQL database in place of SQL. Each NoSQL database has different API and different query systems, requiring a full learning curve for developers every time a new NoSQL database is introduced. NoSQL databases are introduced to leverage large volumes of

data with simpler operations on them. As a result complex query programming for the databases can be often difficult. NoSQL databases are used primarily for specialized projects, such as those that are distributed, that involve large amounts of data, or that must scale. Lack of common query language, consistency support and transaction system create impedance towards the adoption of NoSQL databases in certain types of business applications like banking systems. RDBMSs are equipped with query optimization engine because relational models ensure data independence and provide high level support of ad-hoc query. But in NoSQL databases, the query needs to be implicitly optimized during design time based on the particular architecture of the database itself, the application nature and patterns of queries in the system. From this point of view application designers and architects need to evaluate strengths and weaknesses of each available NoSQL systems before its adoption and development. Lack of proper system and management tools or lack of a proper theoretical foundation can make developing a methodology for database design quite challenging.

A NoSQL database is not to be thought as a replacement for relational databases but instead a better option for certain types of business. Instead of one model for all, we need to look at the data and choose databases accordingly. The relational databases in use today are established and mature tools with large industry and theoretical foundations. But applications have different needs.

2.4 NoSQL: Type example

Although there are many classifications, two main classifications have gained the most popularity – Key-value-stores and Document databases. Key-value-stores have a simple data model in common: data is stored in unstructured records consisting of a key and the values associated with that record. It is also called a Row Store because all of the data for a single record is stored together, in something that we can think of conceptually as a row in relational model.

Document databases are more complex than a simple key-value-stores and model more meaningful data structures as they at least allow encapsulating key-value-pairs in a logical definition. This classification of NoSQL stores each record and its associated data in a concept called a “document” without any strict schema, thus eliminating the need of schema migration efforts [36]. Everything related to a database object is encapsulated together. Storing data in this way has the following advantages:

- Documents are independent units which makes performance better (related data is read contiguously off disk) and makes it easier to distribute data across multiple servers while preserving its locality.

- Application logic is easier to write. A document database basically stores data in bulk thus eliminates any need of having logic in data or its queries like SQL. This provides developers with full freedom to implement any business logic as the need to translate the queries from application to database is absent.

- Unstructured data can be stored easily, since a document contains whatever keys and values the application logic requires. In addition, costly migrations are avoided since the database does not need to know its information schema in advance. [37]

2.4.1 MongoDB

MongoDB is a schema-free document database written in C++ and developed in an open-source project by the company 10gen Inc. [36]. According to its developers, the main goal of MongoDB is to close the gap between the fast and highly scalable key-value-stores and feature-rich traditional RDBMSs.

2.4.1.1 Databases and Collections

MongoDB databases reside on a MongoDB server that can host more than one of such databases which are independent and stored separately by the MongoDB server. A database contains one or more collections consisting of documents. Collections inside databases are referred to by the MongoDB manual as “named groupings of documents”. Data in MongoDB has a flexible schema. Unlike SQL databases, where one must

determine and declare a table's schema before inserting data, MongoDB collections do not enforce document structure. This flexibility facilitates the mapping of documents to an entity or an object. Once the first document is inserted into a database, a collection is created automatically and the inserted document is added to this collection. Such an implicitly created collection gets configured with default parameters by MongoDB [35, 36].

MongoDB allows organizing collections in hierarchical namespaces using a dot-notation, e. g. the collections agency.campaign, agency.product and agency.report residing under the namespace *agency*. The MongoDB manual notes that "this is simply an organizational mechanism for the user. The collection namespace is flat from the database's perspective". ObjectId or $oid, a special 12-byte BSON type guarantees uniqueness within the collection. The ObjectId is generated based on timestamp, machine ID, process ID, and a process-local incremental counter. MongoDB uses ObjectId values as the default values for _id fields. Appendix A lists comparison of MongoDB with SQL in terms and concept.

2.4.1.2 Documents

The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. A document in MongoDB is a data structure comparable to an XML document, a Python dictionary, a Ruby hash or a JSON document. In fact, MongoDB persists documents by a format called BSON which is very similar to JSON but in a binary representation for reasons of efficiency and because of additional data types compared to JSON. Nonetheless, "BSON maps readily to and from JSON and also to various data structures in many programming languages" [35]. The following tables show the encapsulation of objects in document and collections –

{_id: <ObjectId1>, username: “123xyz”, address: “127.0.0.1”}
db.< collection >. find ({ username: “123xyz” }); db.< collection >. save ({ ... });

Figure 1: MongoDB data model [35]

There are two ways that allow applications to model their data in documents: references and embedded documents. With MongoDB, related data can be embedded in a single structure or document. This approach is generally known as “denormalized” models. Normalized data models on the other hand, describe relationships between related data using references between documents. These two approaches create interesting consequences while designing a security model on the context of data. Figures 1and 2 illustrate the approaches.

Figure 2: Data model with embedded fields that contain all related information [35].

The MongoDB manual gives some guidance when to reference an object and when to embed it as follows [36]-

Criteria for Object References –

- First-class domain objects (typically residing in a separate collection)
- Many-to-many reference between objects
- Objects of this type are often queried in large numbers (request all / the first n objects of a certain type)
- The object is large (multiple megabytes)

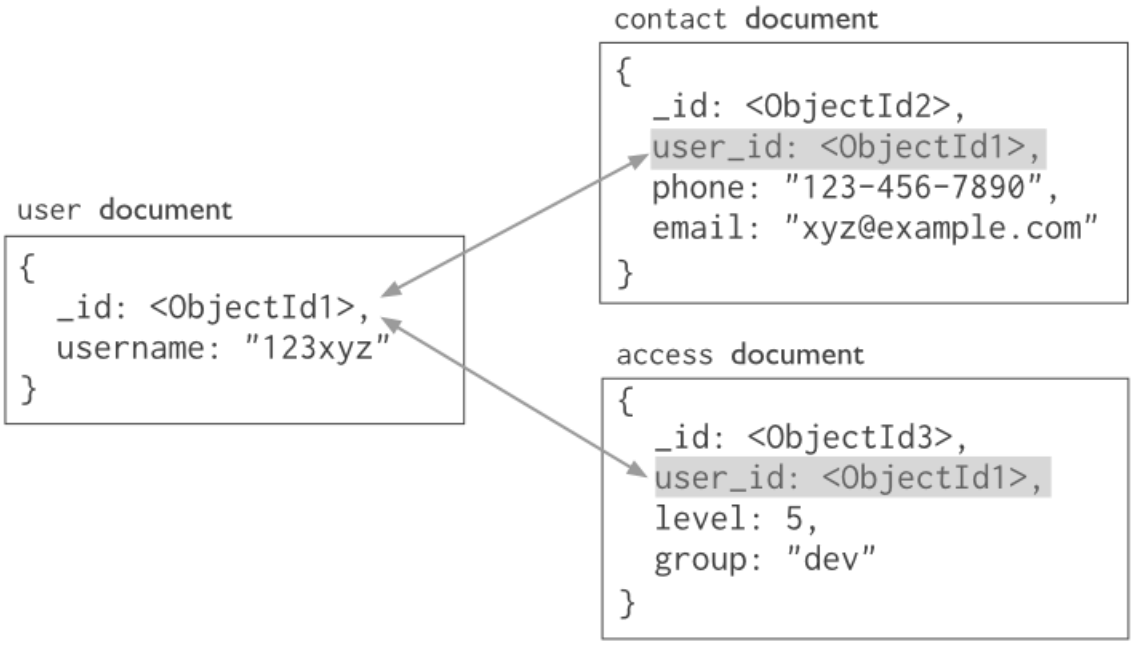

Figure 3: Data model using references to link documents. Both the contact document and the access document contain a reference to the user document [35].

Criteria for Object Embedding –

- 'Many' objects always appear with (i.e. viewed in the context of) detail properties
- Aggregation relationship between object and host object

- Object is not referenced by another object
- Performance to request and operate on the object and its host-object is crucial.

MongoDB developers claim that Embedding is like pre-joined data or single table inheritance. It can be seen as One-to-One Relationships. "Belongs to" relationships are often embedded that provide a holistic representation of entities with their embedded attributes and relationships [38].

2.4.1.3 Query and API

MongoDB has its own ad-hoc query language named Mongo Query Language. It is a JSON-like query language which is basically parameterized function calls using a variety of query operators (Comparison, Logical, Evaluation, Geospatial, etc.) with names that begin with a $ character. To retrieve certain documents from a database collection, a query document is created containing the field conditions to match and then returns a cursor. The cursor is iterated to access the resulting document. By default, the server automatically closes a cursor after 10 minutes of inactivity or if client has exhausted the cursor. To specify equality condition, a query document will have a form as { <field>: <value> } to select all documents that contain the <field> with the specified <value>. In MongoDB, the _id field is always included in results unless explicitly excluded. The following example selects all documents in a fiction collection where the value of the genre field is either 'Thriller' or 'Sci-Fi':

```
db.fiction.find( { genre: { $in: [ 'Thriller', 'Sci-Fi' ] } } );
```

The MongoDB server returns the query results in chunks. Chunk size does not exceed 4194304 bytes which is the maximum BSON document size. For most queries, the first chunk contains 101 documents or just enough documents to exceed 1 megabyte. Subsequent chunks are of 4 megabytes in size. For queries that include a sort operation without an index, the server loads all the documents in memory to perform the sort and returns all documents in the first batch [35].

MongoDB uses a RESTful API. REST (Representational State Transfer) is an architecture style for designing applications that run on a network platform. Its foundation is a stateless, client-server, cacheable communications protocol (e.g., the HTTP protocol). RESTful applications use HTTP requests to post, read data and delete data. The REST API of MongoDB provides additional information and write-access on top of the HTTP status interface. While the REST API does not provide any support for insert, update, or remove operations, it does provide administrative access, and its accessibility represents vulnerability in a secure environment.

2.4.2 MongoDB: Security Concerns

Security was not a primary concern of MongoDB designers. As a result there are quite a few security concerns in its design. MongoDB and other NoSQL databases are still very new in the field with respect to their feature set, especially with the respect to security, so fine-grained permissions or access control in these systems are yet to be provided. Based on this MongoDB or NoSQL databases in general lose in an adoptability study for large mission critical enterprises. Often developers need to devise scopes to maximizes its advantages and minimize its weaknesses.

The relational model takes data and organizes it into many interrelated tables that contain rows and columns. Tables reference each other through reference constraints that are stored in columns as well. When querying the data, the desired information gets collected from multiple tables, sometimes creating logical entities as ‘views’ having its own set of security constraints. In contrast to that, a NoSQL database i.e. MongoDB stores and aggregates data into documents using the JSON format. Each JSON document can be thought of as an object to be used by the application. A JSON document might, for example, take all the data stored in a row that spans several tables of a relational database and aggregate it into a single document. Aggregating this information may lead to duplication of information, but since storage is no longer cost prohibitive, the resulting data model flexibility, ease of efficiently distributing the resulting documents and read and write performance improvements make it an easy trade-off for web-based

applications. As MongoDB databases (like other NoSQL databases) do not have strictly defined database schemas, using JavaScript for query syntax allows developers to write arbitrarily complex queries against irrelevant document structures.

Let's say we have a MongoDB collection that contains some documents representing books, some documents representing movies, and some documents representing music albums. The following JavaScript query function will select all the documents in the specified collection that were written, filmed, or recorded in the specified year:

```
function() {
        var search_year = input_value;
        return this.publicationYear == search_year ||
        this.filmingYear == search_year ||
        this.recordingYear == search_year;
}
```

The statement in an application might look like this:

$query = 'function() {var search_year = \" .$_GET['year'] . '\';' . As most NoSQL database communicates to the server over the HTTP protocol, this function call from an application reads the year from the address of a web browser and can expose an entire database collection due to any ill intended request.

This illustrates that NoSQL database engines that process JavaScript (it is an interpreted language) containing user-specified parameters can be vulnerable to injection attacks. MongoDB, for example, supports the use of JavaScript functions for query specifications and map/reduce operations. Most of the internal commands are actually short JavaScript routines. JavaScript functions, stored in db.system.js collection, are also available to the database users. The following statements can all be used in order to perform the same equivalent query against the database with a where clause:

```
db.fiction.find( { year : { $gt: 2012 } } );
db.fiction.find( { $where: "this.year>2012" });
db.fiction.find("this. year > 2012");
db.fiction.find( { $where: function() {return this.year > 2012;}});
```

In the second and third statements the where clause is passed as a string that might contain values that were concatenated directly with values passed from the user. In the fourth statement the query document is passing an entire JavaScript routine to the database that will be executed against each document in the collection. This cannot be used to modify the database directly, since the $where function is executed with the database in read-only mode, however if an application uses this type of where clause without properly sanitizing the user input, then an injection attack should work against this form of where clause as well.

Chapter 3

3 Role Based Access Control

In Chapter 1 we introduced RBAC as one of the access control model candidate. The basic RBAC model consists of users, roles and permissions. A RBAC enabled system can enforce enterprise specific security policies, with complex role hierarchy and mutually exclusive roles. These leverage any manual process of regulating and administrating any access control management. RBAC can be implemented from very simple to very complex level depending on the system requirement. A properly-administered RBAC system enables users to carry out a range of authorized operations, and provides great flexibility and breadth of application. We mentioned before about RBAC being policy neutral, but it has the mechanisms to support three important security principles for any enterprise solution, namely – least privilege, data abstraction and separation of duty.

3.1 Overview

The National Institute of Standards and Technology (NIST) defines RBAC as "Access control based on user roles (i.e., a collection of access authorizations a user receives based on an explicit or implicit assumption of a given role). Role permissions may be inherited through a role hierarchy and typically reflect the permissions needed to perform defined functions within an organization. A given role may apply to a single individual or to several individuals."

The RBAC model has the following principle components [10]:

1. U, R, P which are respectively the sets of users, roles, and permissions

2. UA ⊆ U x R, which is a many-to-many user assignment relation assigning user to roles.

3. PA ⊆ P x R, which is a many-to-many permission assignment relation assigning permissions to roles.

4. RH ⊆ P x R is a partial order on R called the role hierarchy.

5. APA ⊆ AP x AR, a many-to-many permission to administrative role assignment relation.

6. AUA ⊆ U x AR, a many-to-many user to administrative role assignment relation.

The following logical assertions must hold at any given time by any system using RBAC [16]:

- $(\forall s)\ (\forall u)\ (\forall r) \mid U[s] = u : u\ \in M[r] \leftrightarrow r\ \in R[s]$; for any user u associated with subject s, a role r belongs to the set of authorized role R[s] if and only if u is authorized for r.
- $X : subject\ \times privilege\ \rightarrow boolean,\ is\ true\ when\ (\forall s)(\forall p): X[s,p] \rightarrow A[s] \neq\ \theta$; A[s] is the set of active roles for subject s. A subject can obtain a privilege only if the subject has been assigned an active role.
- $\forall s : r\ \in A[s] \rightarrow r\ \in R[s]$; a subject can only activate roles (A[s]) from its authorized role set (R[s])
- $(\forall s)(\forall p)(\forall r)$: $X[s,p] \rightarrow\ i \in A[s]\ \wedge\ p \in P[r]$; A subject can execute a privilege only if the privilege is authorized for an active role for the subject.

Along with these there is the concept of session in RBAC. A user may log into different sessions with different roles that he/she is member of. For example, in one session a user may be logged in as a creative officer and in another as a regional approver.

3.1.1 RBAC associations

The entities in RBAC are logically connected to each other. The base RBAC model is equipped with important association mechanisms. Basic RBAC relations are:

-Users are associated with roles.
-Roles are associated with permissions.
-A user has permission only if the user has an authorized role which is associated with that permission.

3.1.1.1 Permission to role association (PA)

Each role is associated with multiple permissions. These permissions define the privilege to carry out particular operations in a system on particular resources. For example, a role could grant a read permission to all executives, but an approve permission can only be granted to a supervisor role. Role associations can be easily updated when new permissions are introduced and roles can be deleted as organizational functions change and evolve.

3.1.1.2 User to role association (UA)

Under the RBAC framework, users are granted membership into roles based on the organizational structure. The operations that a user is permitted to perform are based on the user's role. User membership in roles can be revoked easily and new memberships can be created when needed. When a user activates a role, RBAC dictates that the user cannot be granted more privileges than necessary to perform the operation. This concept ensures the least privilege requirement [25].

These two associations simplify the administration and management of permissions; roles can be updated without updating the permissions for every user on an individual basis. Figure 4 illustrates the different models of RBAC over the years.

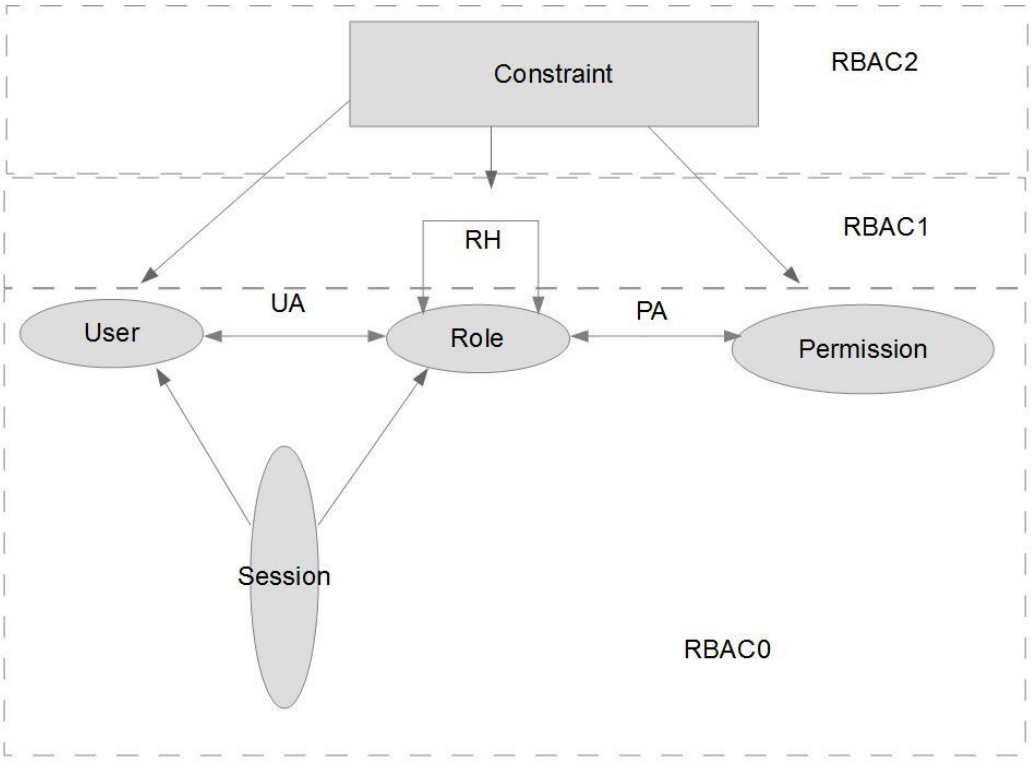

Figure 4: Different RBAC Models

The role hierarchy is introduced in $RBAC_1$. This creates the requirement for separation of duty. Roles can have overlapping permissions where different roles may execute the same operations. Users can also have private roles that cannot be shared with other users. This puts a constraint on the role inheritance. Constraints can be viewed as the most important entity in the current RBAC model. The presence of constraints in the model facilitates the expression of higher level security policy for organizations. For example, a constraint can be applied when a security system has two mutually disjoint roles and users cannot be permitted to gain membership in both roles. We discuss about constraint in detail in the following section.

3.1.2 Constraints

Constraints have been introduced in $RBAC_2$. Even though conflict of interest in the role concept was introduced first, real world applications demand more practical and flexible

solution. The concept of conflicting permissions addresses conflict in terms of permissions rather than roles. Defining conflict in terms of permissions ensures more fine grained security than defining it in terms of roles. Conflict defined in terms of roles allows provision of assigning conflicting permissions to the same role. But conflict defined in terms of permissions eliminates this security gap [1]. Sandhu states in his paper about RBAC models that the constraints in the user assignment relation have counterparts in the privilege assignment relation [21]. The first of these constraints is the mutually exclusive role. A mutual exclusion constraint is enforced on the assignment of users to roles to avoid conflict of interest or possible fraud. When two roles are defined as mutually exclusive, in terms of user to role association, it enforces that a user cannot belong to both the roles and in terms of permission to role assignment, it enforces that the same permission cannot be assigned to both roles. This constraint on PA restricts execution of probable malicious operations. The mutual exclusion constraint facilitates convenient role administration and decentralization without compromising any possible organizational security policy. There are two classifications of role exclusion: authorization time and runtime. They can be defined as:

Authorization time [16]:

$$\forall u(\forall i,j)|\ i\ \neq j:(i,j)\in E\ \rightarrow u\ \in M[i]\rightarrow u\ \notin\ M[j]$$

This implies that if two roles *i, j* are mutually exclusive, a user u can obtain one of *i, j* if only the user doesn't belong to the set of users authorized for the other role (*M[i]* or *M[j]*).

Runtime [16]:

$$(\forall u)(\forall s)(\forall i,j)|\ i\neq j;U[s]=u:(i,j)\in E\rightarrow u\ \in M[i]\wedge u\ \notin\ M[j]$$
$$\rightarrow i\ \in A[s]\rightarrow\ j\ \notin A[s]$$

This is a less restrictive constraint. This basically lets a user to be authorized for both the mutually exclusive roles (E) but puts restriction by the condition that only one role can be

activated at a time. This weakness in constraint is addressed by the security principle of least privilege role set.

3.1.3 Security principles

Mutual exclusion constraint specifies several security principles to be ensured in RBAC. Separation of duty and least privilege are not entities of RBAC but rather the RBAC model supports development of such framework. Least privilege grants only the necessary rights to complete a task and separation of duty places restrictions on assignment of disjoint roles and permissions.

3.1.3.1 Least privilege

The principle of least privilege requires that any module in a computer abstraction layer can only have access to information or privilege, pertaining to the purpose it's supposed to satisfy. This is a key design concept for protection of data and related logic. Cholewka et. al. [7] present strict least privilege as a mechanism to solidify the security concern around the privilege set available for a subject. According to them it distinguishes between a person's job and the tasks that a person must fulfill as part of his job. Strict least privilege thus signifies the requirement of providing the user with just the permissions to perform any steps of a workflow activity at any given time.

3.1.3.2 Separation of duties

One of the important advantages of using RBAC is that it provides enough granular control that system administrators can implement proper separation of duties (SoD) in any targeted applications. SoD is a classic security principle concerning conflict of interest, the appearance of conflict of interest, wrongful acts, fraud, abuse and errors. A crucial aspect of security is to separate permission in such a way that users cannot abuse their privileges in order to execute or hide nonconforming activity. Proper separation of duty requires that users cannot have conflicting responsibilities or be responsible for reporting on themselves or their senior positions. Separation of duty requirements are often formulated as business rules such as "a person may not approve his own travel

allowance" or "any bank cheque requires two different signatures". NIST explains SoD the following way:

"RBAC is . . . well suited to separation-of-duty requirements, where no single individual has all permissions needed for critical operations such as expenditure of funds. Proper operation of RBAC requires that roles fall under a single administrative domain or have a consistent definition across multiple domains, so distributed applications might be challenging" [17].

Separation of duty can be interpreted in various forms but the two main concepts in the implementation are strong exclusion or static separation of duty (SSD) and weak exclusion or dynamic separation of duty (DSD). Static separation of duty is the simplest form of SoD; it defines role memberships that are mutually exclusive, and thus the conflicting roles cannot both be in UA for a given user.

Dynamic Separation of Duties states that the same person can activate the purchasing role and the approving role, but they cannot approve their own purchase. In the SSD model, a user may not be members of both roles. In the DSD model, a user could be a member of both roles, but could not function in both capacities for the same linked transactions.

The above two classifications of SoD address the issue of when an exclusion can be enforced. This is not flexible enough in many business models while implementing RBAC. The need to associate privileges of mutual exclusive roles to other nonexclusive roles creates additional complexity. Kuhn [16] addressed the issue of how and which privileges can be shared with mutual exclusion of roles and other roles by introducing additional variety in mutual exclusion. In his paper, privileges in mutually exclusive roles can be characterized as disjoint, denoted as complete exclusion in permission assignment. This type of exclusion implies that if two roles *i* and *j* are defined to be mutually exclusive (denoted as the mapping E (i , j)), then a privilege that is assigned to either of *i* or *j* cannot be assigned to any other role *k* in the RBAC module; $(\forall i, j, k)(\forall p) \mid i \neq j, i \neq j : (i, j) \in E \rightarrow p \in P[i] \wedge p \notin P[k]$. This rule along with SSD can better enforce SoD policies [16, 20].

Another type of exclusion is partial or shared. It says that between two mutually exclusive roles (*E(i , j)*), the privilege set cannot be complete equal; at least one privilege in *i* must not be in *j*; $(\forall i, j)(\exists p) |\ i \neq j : (i, j) \in E \ \rightarrow p \ \in P[i] \rightarrow p \ \notin \ P[j]$. These two classifications in Kuhn's paper [16] along with the two mutual exclusion rules described in the Section 3.1.2 give us a variety of four SoD relations: authorization-time complete, runtime complete, authorization-time partial and runtime partial. Authorization-time/ complete exclusion is the safest of the constraints. However, often implementation of RBAC demands flexibility and then run-time/complete or run-time/shared exclusion can ensure safety within a single user session.

3.2 Incorporating User Context Information into RBAC

Although RBAC makes an authorization decision based on a user's role, it is not sufficiently powerful to enforce rules that are dependent on runtime parameters or contexts. A context-aware access control enables administrators to specify more precise and fine-grain authorization polices for any application. Contexts can be information about a user's current physical location, the devices being used, the network access node, and current user activity. Additionally, there can be other conditions and characteristics that may be relevant in defining a context e.g. temporal attributes of an operation to be performed, geospatial attributes of resources to be accessed.

However, incorporating user context information to control access to a system is complicated. First, it requires that extracting and embedding context information in a system meets proper authenticity and integrity requirements. Second, the dynamic nature of information may require revoking role membership with a change of context information. This is challenging as it involves dynamic changes to policies during deployment. A Context-Aware Role-Based Access Control (CA-RBAC) has been proposed by Kulkarni and Tripathi [18] to support such requirements.

In this model the system sends a context query to one of the context agents for each access request. The context agents authenticate and validate the data integrity aggregated

from interfaces. The access control layer is responsible for managing context-based access to resources based on personalized permissions. The CA-RBAC programming framework consists of operational layers - context management and access control. The context management layer is responsible for aggregating data to generate context information required by any system. Figure 5 shows the composition of the CA-RBAC model.

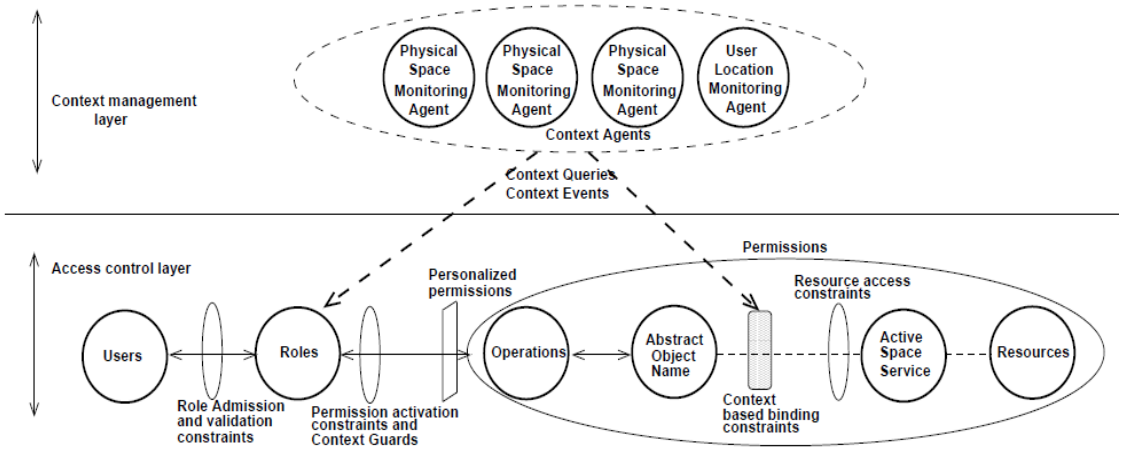

Figure 5: Context-Aware RBAC Model (CA-RBAC) [18, Figure 1]

Kulkarni and Tripathi [18] established five key constraints on a system using the context notion. The constraints are as follows:

1. Role admission and validation constraints: These constraints specify context-based conditions that need to be satisfied before activating a role for a user, and also for continuing a user's membership in a role.

2. Context-based role permissions: Dynamic object binding causes role operations to interface with different privileges under different context conditions.

3. Personalized role permissions: The operations that are accessible through role permission may be different for different role members and may depend on the context

information associated with a role member. In basic RBAC a role permission invoked by any member of a role is executed on the same object. The CA-RBAC model transcends the basic model when a permission invoked by different role members needs to be invoked on different object instances based on each role member's individual context.

4. Context-based permission activation constraints: These constraints are associated with specific role permissions, and specify context-based conditions that need to hold for a role member to execute such permissions.

5. Context-based resource access constraints: These constraints restrict a role member's access to a subset of resources that are managed by an active space service.

Chapter 4

4 Motivation and Objectives

Developers now can turn any sound idea into software with the help of a number of open source development frameworks and relational databases; add the Cloud into the equation and developers freely ignore system maintenance, load balancing, scaling, etc. The elasticity provided by cloud computing enables computational resources to be scaled up with just seed funding instead of huge capital investments. This agility often makes the software developers overlook long term effects. The community and collaborative nature of the current web often makes a service bloom in unprecedented volume. A large number of users produce a large amount of data. Suddenly, this rapid growth cannot be contained within the relational database making the new software another start-up disaster where overloaded database machines and resulting high response times destroy a previously good user experience.

Modern relational database systems often impair a relatively successful service in the area of flexibility and scalability. Due to data independence, SQL databases often disguise a potential expensive operation with its high level interfaces. This can cause a considerable amount of performance drop when the database grows beyond the planned scale. This scalability failure and lack of a clear picture of consequences due to data growth are leading more and more developers towards highly data dependent NoSQL options. NoSQL database technology offers the flexibility, scalability and performance that present-day web scale application developers demand.

4.1 Why NoSQL

An agile development cycle facilitates better solutions in terms of meeting real life scenarios. As real life scenarios rarely follow a fixed set of parameters and end users' expectations of consumable services change between morning and evening, developers

are hard pressed to find the next best solutions for applications that can serve almost without any prerequisite computational parameters. This boundless evolution in service results in three major trends which are challenging the status quo of traditional databases. These three trends are the industry favorites – Big data, Big users and Cloud computing. Organizations are in a constant race in making more and more services available to end users, to alleviate every aspect of a person's everyday activities. They are exploring more and more interfaces for users to connect with. This automatically makes more and more users converge in the IT frontier and the more users, the bigger the volume of data with little to no apparent structure. With numerous parties involved, the demand for meaningful services run seamlessly among interfaces with ever-improved performance in terms of highly responsive applications allowing more complex processing of data is constantly raising more and more challenges for IT professionals. These are the driving forces in the development and adoption of new technologies. One such technology is NoSQL. NoSQL is becoming a popular alternative to relational databases, especially as more enterprises recognize that operating at scale is better achieved on clusters of standard, commodity servers and a flexible data model is often better for the variety and type of data captured and processed.

4.1.1 Challenges

Big Data refers to the colossal growth in amount, diversity and rate of information being produced and the bulk of applications that generate, store and process this data. While this trend represents an incredible opportunity for enterprises across all industries, delivering on the promise of Big Data is no trivial task. The main challenges in a database context identified by MongoDB Inc. [41] are:

- **Data Volumes.** In this era of social networks, applications can go viral overnight. The underlying database is required to support rapid spikes in data volume and throughput without downtime, without custom adoption and without compromising performance. Application success depends on flexible and resilient data systems.

- **Real-Time Processing.** Ever since the bloom of Web2.0, users expect applications to behave as if the computational process doesn't even exist. Applications more often need to derive knowledge from collected data before users identify the need for the information. This demands in-place, in real-time processing of data without moving it to a separate module. The trivial batch analytics in offline settings are not adequate for today's service model. Lightweight real-time analytics should be devised to serve content retroactively, enable interactivity and improve user experience.
- **Rapid Evolution.** This is a highly competitive market; organizations need to address changes in business as fast as possible to be able to survive. An agile development process promises rapid adoption of changes in user demand. One of the ways is to keep adoption costs light. Schemaless data structures can offer developers the leverage to keep pace in the race to serve the ever evolving, highly demanding user group. Whoever satisfies the big user, succeeds.
- **Flexible Deployment.** Requirements for deploying enterprise and consumer applications have changed radically in only the last few years. As more organizations engage in real-time interactions with consumers, they measure response times in milliseconds and downtime means losing money. This demands flexible replication over multiple data centers. Let it be an internal enterprise application or a social media application, organizations need to deploy the application or part of it on their own premises, in the cloud or in hybrid environments to better utilize the various deployment models. [39]

4.2 Play of NoSQL

Traditional technologies struggle to accommodate the growth, diversity and volume of data produced in current applications. For example relational databases were designed for a static data model in which growth of data was controlled and predictable, predefined queries were in place and the database was normally housed in single server in one data

center or even in-house. NoSQL plays a valid alternative in such a context where system designers have limited control over structure, amount and nature of data.

4.2.1 Schemaless

As discussed before, schemalessness is one of the main reasons for interest in NoSQL databases. Application developers have been frustrated with the staggering mismatch between the relational model and the in-memory data structures. More often than not, developers had to compromise in features or an efficient business model because the relational database wasn't sufficiently flexible.

In a relational database a tab le is created with command similar to the following-

Create table module (id int, modulename text, parentmodule int, submodule int)

Then this command to insert data in the above mentioned table fails due to the mismatch of number of columns to insert as well as the mismatch of the data types: *Insert into module values (15,"xyz", "rty")*

But NoSQL allows one to store any data –

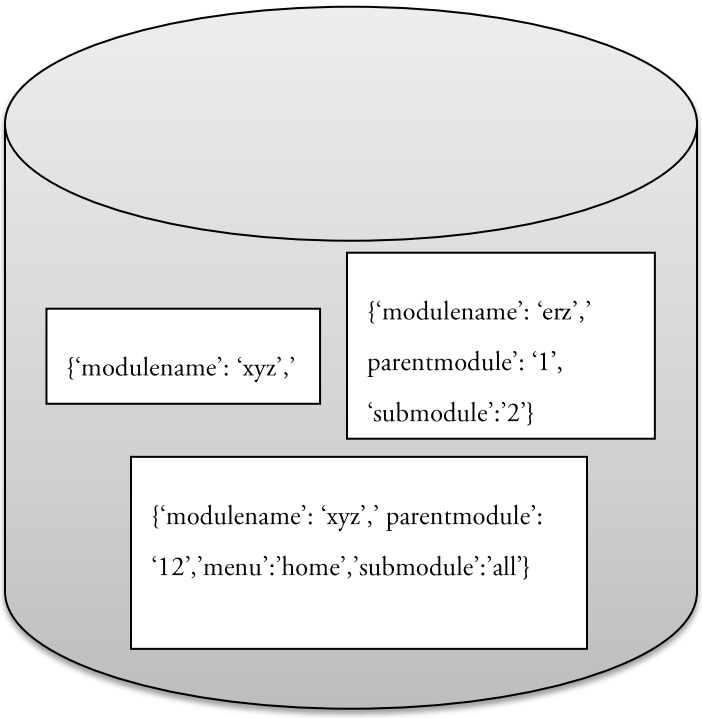

Figure 6: NoSQL Datastore

Even though relational database can have blobs of unstructured data, it still cannot translate the business model as seamlessly as a fast response model demands. The query optimization fails once the data growth surpasses every forecast the developers made. This massive data is processed in memory and again in the relational database. Even with a design goal of minimizing to have business rules in the database, relational models have their own structure to follow, thus limiting how an application can behave.

In NoSQL, developers can treat the database as just a place to dump the data. With minimum maintenance, this becomes an alternative solution that is sympathetic to users and thus it is very easy to get business owners to accept. Applications supported by NoSQL get delivered faster, produce less impedance in any user flow and support implementation of new modules more efficiently than relational models. At any point in time an application module can be extended by introducing non uniform types. Developers can even give users more control over their data by introducing custom fields, which play less of a role in actual business process but rather they serve in user interface for convenience. Being schemaless, NoSQL supports these last minute changes in applications out of the box. But more often than we care to admit, developers get excited about implementing the module and overlook the security holes presented by the absence the of relational model's access control. None of the NoSQL pioneers has a detailed level security mechanism at place. Often developers will point out that it's not efficient to implement a fine grained security mechanism in place for NoSQL as it might feel like tracing our steps back to the relational model.

Schemaless structures still have an implicit schema. Any code that manipulates the data needs to make some assumptions about its structure, such as the name of fields. Any data that doesn't fit this **implicit schema** will not be manipulated properly, leading to errors. A class definition defines the logical fields one can use to manipulate it. This is effectively a schema.

4.2.2 Schema Migrations

Businesses change hands, merge with other parties even change their service nature. But information is money and every business wants all their data migrated in the new setup. From a higher level view, migration seems safer in a relational database. But a relational database is less tolerant of slight schema changes. It saves every change and puts version control and more than often complicates the migrated model. In NoSQL on the other hand there is no schema update, all developers need is to handle the codes handling implicit schema. Schemaless stores can ease migration, since access code can be defined to read from either the old structure or the mutated structure. Developers can make the access codes to gradually migrate old structures to mutated structures. Once the old data count becomes zero, the overhead becomes obsolete.

4.2.3 Scalability and performance advantages

The data stored in a relational database is rigidly structured by the layout of the tables, the predesigned relations among these tables and the types of the columns. The need to scale a relational database can be achieved by running it on more powerful and specialized servers; organizations often acquire custom-made servers to scale their relational database. Yet to scale beyond a certain point, the database must be distributed across multiple servers. Relational databases don’t work inherently in a distributed manner because joining their tables across multiple machines is expensive. Also, relational databases aren’t designed to function with data partitioning, so distributing their functionality is very tedious to achieve. To deal with the increase in concurrent users (Big Users) and the amount of data (Big Data), applications and their underlying databases need to scale using one of two choices: scale up or scale out. Scaling up implies a centralized approach involving servers with larger capacity. Scaling out implies a distributed approach that leverages many standard, commodity, physical or virtual servers [19].

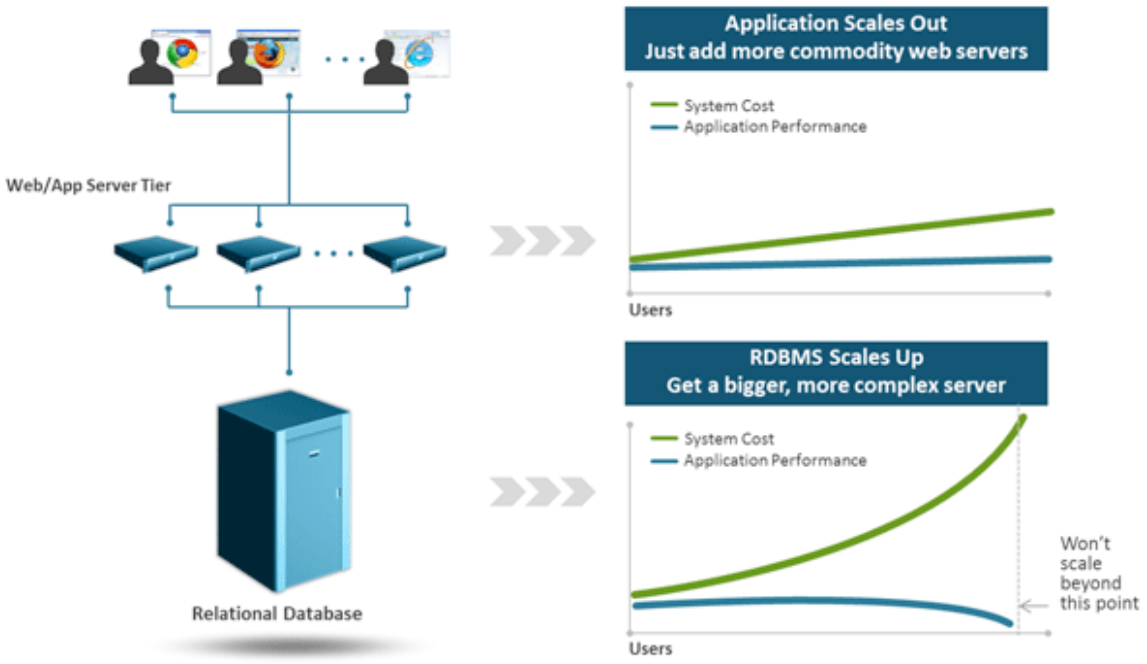

Figure 7: Scale up with relational technology [40]

Scaling up is the natural choice in applications with the relational model as the underlying data storage. This choice is influenced by the fundamentally centralized, shared-everything architecture of relational databases. These characteristics of a relational database make it a less feasible candidate for scaling out. In order to scale out database administrators have to be really careful about how to create and organize the schema and structures, know about all the idiosyncrasies involved in a relational database and these eventually results in making the maintenance and deployment exponentially more complex and less resilient towards failure. Due to these reasons enterprises would opt for increasing server capacity in terms of CPU, memory and discs. But this solution is highly proprietary and increasingly expensive, unlike the low-cost, commodity hardware typically used so effectively at the web/application server tier [40]. This is illustrated in Figures 7 and 8: as the number of user increases, for both types of databases, an application can scale out just by adding more web servers without affecting performance and causing very little rise in system cost. Whereas in the case of database system, the RDBMS scales out up to certain point by adding more complex database servers and

causing a sharp increase in system cost and reduced application performance. The picture is different when NoSQL is in place; the system cost and application performance curve is almost the same for application scale out and database scale out.

NoSQL databases have been conceived with the principle target to gain distributed, scale out databases. They use a cluster of standard, physical or virtual servers to store data and support database operations. To scale, additional servers are joined to the cluster and the data and database operations are spread across the larger cluster. Since commodity servers are expected to fail from time-to-time, NoSQL databases are built to tolerate and recover from such failure making them highly resilient.

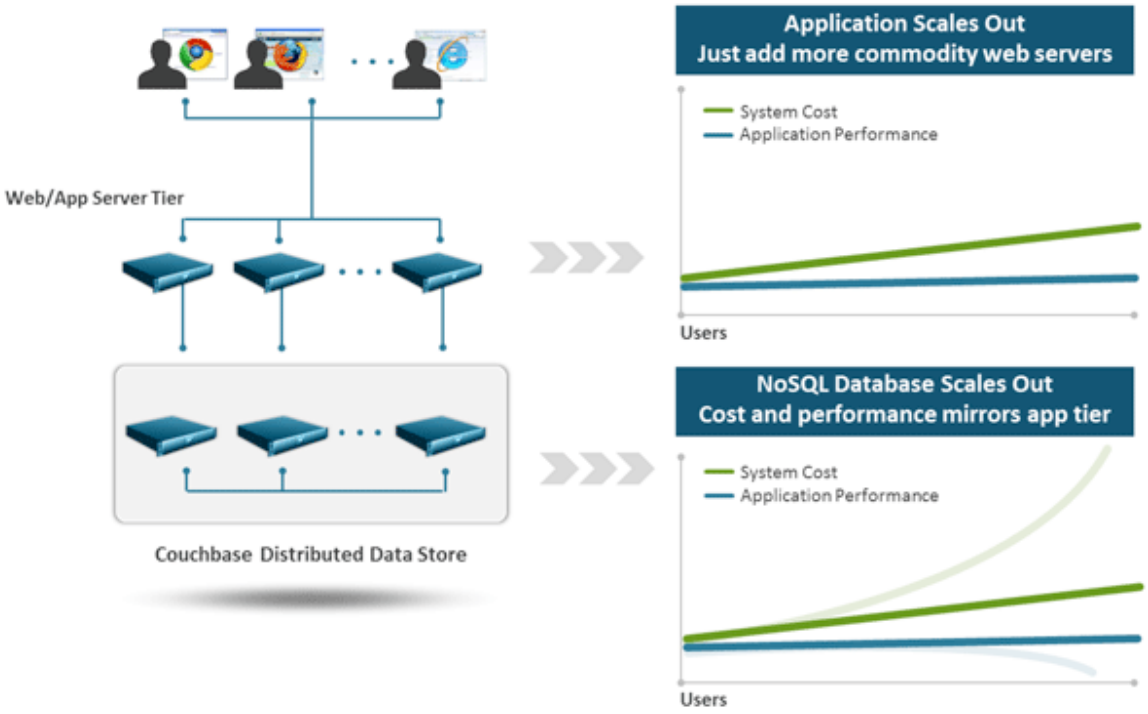

Figure 8: Scale out with NoSQL technology at the database tier [40]

NoSQL uses an easier and linear approach for database scaling. With the increase of users, NoSQL simply adds new database servers to the cluster when a threshold is met. It is often part of the database mechanism and thus abstracted away from application layer. Applications continue to serve the users without any hindrance or downtime. The

application always sees a single (distributed) system no matter how many replications have taken place.

A distributed scale out approach is the less expensive between the two alternatives. The servers to support scaling with required fault tolerance are complex in design and often are custom made to serve individual business. Also licensing costs of commercial relational databases end up being really high when used for scaling out, as most of them provide pricing scheme with single server in the equation. NoSQL databases don't have either of the above issues. NoSQL databases are generally open source with priced for maintenance, runs seamlessly in distributed setting and are relatively inexpensive.

A NoSQL database automatically distributes data across servers, without requiring applications to participate and influencing the design of data or process. Servers can be added or removed from the data layer on the go without causing service disruption. NoSQL databases also support data replication to ensure high availability and support disaster recovery and sharding to facilitate write-scale. For some NoSQL database (i.e. MongoDB) the sharding can be programmed so that a load balancer only starts moving data when a predefined threshold is met. This leverages monitoring and makes the process easier. Couchbase [40] claims "A properly managed NoSQL database system should never need to be taken offline, for any reason, supporting 24x365 continuous operations of applications".

4.3 Motivation

In the current software industry system architects and developers are tirelessly looking for ways to deliver software faster. The distinction between what is urgent and what is important is now becoming a matter of debate. The IT industry is growing faster and wider, making this horizontal growth urgent while detailed security is the important aspect and often derived from existing frameworks. While implementing security measures with traditional frameworks, what developers often do is that they place tweaks and patches rather than creating a new solution. As a result they often encounter scenarios where they need to compromise between feature set and over-all security. Still today the design principle of NoSQL is to give the application developers as much

freedom in designing the data model as possible. Fast delivery and fast scale out business model often leave the fine grained security mechanism behind.

NoSQL movement is a lean approach; in many cases it omits or reduces functionality of data storage for the sake of speed. In simple one user centric applications, this scenario is acceptable, but when an enterprise solution adopts NoSQL for the sake of scalability and flexibility, to support the running business models developers often need to reinvent the wheel especially in case of shared information and at that time access security becomes a hurdle to cross. It becomes more complicated when this security concern is raised after the application is deployed and has been in use for a period of time.

The purpose of this thesis is to present an access control mechanism for aiding application developers in such a case. The database candidate for the purpose is a Document based NoSQL database while RBAC is chosen as the base access control mechanism. Normally, RBAC works better with structured data while in NoSQL the meaning of the same data change depending on operation/user context. Even though the security concerns discussed in Chapter 2 seem natural, the solutions available are not standardized. The schemalessness, high fault tolerance and support for non-uniform types make it complicated to add any security restrictions. As the data model is dynamic, one can keep on adding attributes to records when the application is being used. So a suitable approach for a security model in the context of this architecture must have some notion of forward-looking security. So, understanding what happens with the new attributes that are introduced to the database and what the privileges should be granted for the new attributes are challenging. This is a concept that is an aberration and very few have ventured into it.

In spite of being schemaless and devoid of all the idiosyncrasies of a relational database, NoSQL databases have options to configure a full array of security capabilities. For example, in an RDBMS, while inserting, if the data doesn't adhere to the schema, the insertion will fail automatically. In NoSQL one can choose to trigger on the insertion of a new record or document rules that have been executed to ensure malformed data is not being inserted. However one of the big issues is that there are very few users who would

know how to implement these options now a day when everything has a package or library. Very few developers start coding from the library level. New users in NoSQL will likely make big configuration mistakes initially as the available expertise is still insufficient, thus making a very shallow knowledge pool. Practically everyone is new to NoSQL; the first thing they care about is to make it work, and everything else can wait a year or so. Most developers who are attracted to NoSQL also tend to be in startups with little to no security experience in the first place. In contrast to RDBMS which has preliminary and basic security in place, NoSQL requires a certain amount of trial and error to figure out which type of security mechanisms to put in place. Most often, business simply doesn't have the time or resources to support it.

The use of NoSQL databases is still dominant in social network applications rather than enterprise solutions. In most cases, the database tends to sit behind the firewall and it ties to this application and usually isn't available for other parts of the organization to tap into. This kind of dependence on perimeter security related to NoSQL is a significant development impediment for Enterprise solutions where a database collects and aggregates lots of the business data that need to be accessed through out various in-house applications. Most NoSQL solutions have very weak authentication and few of them have a complete concept of how to secure multiple users and their access to the data. Usually if one has any account one can access the whole data store. This doesn't stop businesses from adopting it. Often the available solution is to invest in the best of the developers; they use veteran programmers because a lot of the security is going to go into the client software. This also involves allocating additional time buffers in deploy times and frameworks to avoid any unseen complexities with the new mechanisms in the relatively new database systems. Even though this model works, it's counterproductive because, except for the scaling facilities, this model too takes the same or sometimes more time and effort than if RDBMS was chosen. Ironically, the advances in database application security in recent years have very little to do with databases themselves. Popular programming platforms are now better equipped with features to create fool proof and secured access models for almost any sort of database.

The above discussion summarizes the goals for this research as follows:

Goal 1: To provide application developers an easy to adopt security module without compromising their freedom to design realistic applications.
Goal 2: To leverage security feature for multi tenancy. Even though it is lot cheaper to simply create new ‘boxes’ for new clients of the same service, it is not efficient. So the goal is to leverage the security from each box to a central node.
Goal 3: Serve light weight security features while letting an application module to create branches at run time.

4.4 A target example

To illustrate the access control mechanism for a document based NoSQL database, let us consider a reporting system on a data warehouse for a globalized agency solution. A 3-tier web solution collects numerous data from agency personnel, their clients and end users with a goal of optimizing marketing operations, simplifying the organization to enhance global brand governance, reduce the time to market and lower overall cost. A data warehouse periodically collects a large amount of various forms of data from a number of interfaces and number of business solutions which are all part of a main system.

Data from marketing campaigns all over the world, resource metadata from design houses, multiple brand governance, campaign briefing, client and agency budgets as well as certain a degree of personal information of all parties get dumped in the data warehouse. This massive repository of seemingly unintelligent data is open inside the agency. People from simple desktop publishing, planning, creative design, legal, marketing to regional and global directors and even Chief Executive/ Operation/ Finance/ Information Officers (CXOs) have access to the data. Occasionally, upon client demand, the reporting service is accessible to certain clients. As part of marketing research or brand governance, often some users even have to create a copy of an instance of a data collection and enhance it with new data, which can be of base type or custom type. The amount of data an agency creates every hour from all corners of the world is massive and

it is very hard to predict which form of data will be transferred to the reporting database. Thus it's impractical to enforce any schema for it; thus the data warehouse is housed in a NoSQL database.

Recalling what was discussed in Chapter 3 the current level of security mechanism that gets shipped with the database package is mainly at the instance level. This means any user having access to a database instance has complete access to all data. Considering this, access control in the above reporting service presents a daunting challenge. For example, clients should not be able to configure a report which may involve other clients' data or any mission critical information from the agency. Campaign/budget/personal information report should not be accessed across global, regional and local business nodes. The reporting service also needs to govern who can create a copy of certain data, enhance and give further access to it. In contrast with a relational database where permissions can be set on a schema level and where every user of the database can be configured with accessible relations, in a NoSQL database the data is very flat. None of the traditional approaches works for the above example. The unique problems in designing an access control for the above example are:

- Grouping data and giving proper access to certain users with appropriate privileges.
- Manageability of data growth and dynamic permission set for enhancement of data.
- Devising proper context for different parts of a data collection.

Chapter 5

5 Proposed Solution

Schemaless design makes the access control requirement very complex. The shape of a collection of data at any given point of time is unpredictable. A global agency enterprise houses hundreds of users, roles, resources and permissions to model. In an agency there are a variety of applications that require making complex access control decisions. Let us suppose, in our reporting example, a ‘Creative’ role cannot access any campaign data unless that access is authorized by the ‘Product Lead’ leading that campaign. Although RBAC can make an authorization decision based on a user’s role, it is not flexible enough to enforce rules that are dependent on any particular state of data. This makes the security requirements in our reporting example very dynamic and in need of flexible policy enforcement. Even in the normal case, there are many intricate user-data relationships that must be managed by the security framework; in addition, the security model should also accommodate delegation of access to data in mission critical situations. In our proposed access control model, the policy execution mechanism is highly dynamic and independent of any particular business model.

5.1 A Context Aware RBAC Model

As we recall in RBAC, a user has a role which is a set of privileges. In a standard implementation of RBAC, there are implicit subset constraints on objects which are represented by Privileges. Often these constraints are static and manually programmed beforehand. For example, in a small business, an employee can only see his/her annual performance evaluation, a supervisor/manager can only see the performance report for the employees who directly report to him/her and a department head can see the performance reports of all the employees in his/her department. At any given time there is a predictable number of employees and performance reports in a system, thus a precompiled list of authorized access control policies can establish the proper control over such a system. Now we overlay structure to our enterprise example. Not only do we not always know beforehand which objects are to be under access control, the number of

the objects is also dynamic. This gets more out of hand when compound objects come into the scene. Standard RBAC will have to store a large number of pre-authorized lists/policies which would have to be searched every time the system is accessed. This is not only inefficient but also not scalable. In document based database, data is an unstructured document. The same data body can be accessed by different users for different purposes, where access can be partial. In standard RBAC we can introduce constraints to regulate which roles to be activated at any given time, but it's not quite possible to specify the purpose or part of a large object. With the goal to not over complicate the design principles of NoSQL, what we propose in this study is to introduce Context to extend a standard RBAC solution.

An overview of the basic concept of the solution is presented in Figure 9. Solid lines represent relationships among the standard RBAC participants; whereas the dashed lines introduce the relationship of Context regulating the standard relationships. The given model is based on the standard RBAC model which is extended with the following three concepts: Database instances, Data groups and Context. In the proposed model the definition of the session is taken from the standard RBAC model [10].

Let us first define our terminology. An appropriate definition by A. K. Dey is - "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [11].

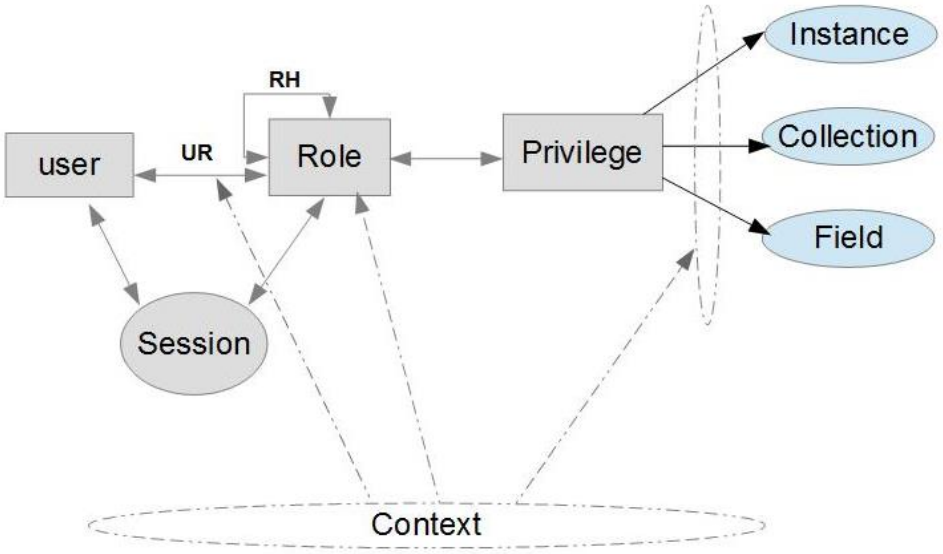

Figure 9: Basic concept of a Context facilitated RBAC

5.1.1 Definition of Model Elements

5.1.1.1 Data Object (O)

A data object is the smallest unit to be accessed and protected in an application. In a document database it can be a database instance, data collection or individual fields. A data collection can consist of simple fields and thus require a simple policy definition, or the fields can be embedded data objects thus requiring a more intelligent access control policy.

5.1.1.2 Data Group (G)

A data group is a group of data objects with the same level of security preference. For instance, all types of business process create similar records called “Report”. Data groups can be used to combine data objects and create a compound definition of object. Thus “Regional Report” is a group of data composed of subsidiary objects such as product, budget, finance, legal, etc.

5.1.1.3 Data Set (DS)

A data set is the set of all data objects within an application.

5.1.1.4 User (U)

A user set is the set of potential entities that will access the data objects in the data set of an application; it is analogous to subject in RBAC. Let U be the set of all users in the system, and S be the set of users' sessions. The predicate usersSession(u, s) verifies if the session s (s ∈ S) belongs to the user u (u ∈ U). In the case of RBAC-based models, assignment of roles to a user can depend on user's context.

5.1.1.5 Role (R)

A role is a set of relevant set of responsibilities or authorities conceptualized by participants in an application. Roles can be enabled, disabled or active. A set of roles can be available to being activated for a given user, while only one role can be activated in one session. Whether a role will be in the enabled or disabled state depends on the context condition. Therefore, we define a role as a tuple (Rn , RCtx, State); Rn is the name of the role and 'State' can be disabled, enabled, or active. Here RCtx defines the context condition for enabling/disabling role Rn for a user in a given session with respect to which state 'State' it is in.

5.1.1.6 Context term (CT)

A context term can be defined as necessary attributes for all the relationships or operations in an application. In simple cases, a context term may be a concrete property familiar in everyday life, such as time or location. In a more complex scenario, a context type can also be used to describe an abstract concept such as interface and other security identification for single-signon authentication. By analyzing the system security requirements, application designers determine the context terms for an application. For example, suppose we have a security requirement that a product designer can only see reports on current campaigns he/she is directly assigned in or which belong to the same business unit/interface he/she signs in from. From this requirement, we need a concept to describe if there is a logical relationship like "current" or "regional" between a user and a

campaign report. Context terms are pre-defined but system administrators may need to introduce new contexts. As a document based database accepts query on HTTP connections through REST, context terms can be defined in Web Service Definition Language (WSDL) [42].

5.1.1.7 Context expression (CtxExp)

In order to make this abstract concept of context term usable when authorization decisions are made, we evaluate each context term by some context expression forming conditions and clauses. A context expression can be presented as a logical expression (shown below). Based on this format, an access control schema is capable of specifying any complex context related constraint to describe all kinds of security requirements. The context expression describes operations to be executed and the conditions under which these can be executed. We adopt the context expression/condition as a logical expression which may consist of queries, context functions, logical operators {¬ ; ∧ ; ∨} and comparison operators {< ; ≤ ; > ; ≥ ; = ; /= } [25]

Context Expression: $=\text{Clause}_1 \vee \text{Clause}_2 \ldots \vee \text{Clause}_i$
Clause: $= \text{Condition}_1 \wedge \text{Condition}_2 \ldots \wedge \text{Condition}_j$
Condition: = <CT> <OP> <VALUE>, where OP is a logical operator in the set {>, ≥, <, ≤, ≠, =} and this set can be extended to accommodate user-defined operators as well; VALUE is a specific value of CT.

The context expressions are essentially conditions for the assignment relations (a proper assignment relation will be established if the assigned context condition is satisfied) and for role definitions in order to enable/disable roles. Let us suppose, we have a security rule such that in an advertisement agency, a campaign report can be accessed when the logical location of the campaign and the user requesting access is same or the user is an agency user (can be defined as function ‘user belongs to the team’) and the campaign status is not archived (can be defined as function ‘status of campaign is not archived’). For this security policy, the context expression can be presented as follows:

```
Context Expression: = (Region (User) = Region (Campaign)) ∨ (IsRunning(Campaign)
∧ User ∈ Team(Campaign))
IsRunning: = Status (Campaign)! = 'Archived'
Team: = Personnel (Campaign)
```

Figure 10: Example of Context Expression

5.1.2 Relation Assignments

5.1.2.1 User - Role Assignment

In a standard RBAC user role assignment is static. In our proposed solution, a static assignment can be defined as the predicate sRoleAssign(r, u, ctx); assignment of the role r to the user u is possible if the context ctx is evaluated as true. But systems that are hosted in a NoSQL database enjoy the ability to grow without predefined implicit schema and it is necessary to assign certain roles only to the users who match certain context.

In the case of such a scenario, system administrators can add new roles and privileges for newly added data objects without being concerned who will access it. This approach is then can be defined as dynamic role creation for the dynamic data growth. Suppose, in the reporting system, the campaign collection gets a new mission critical field. System administrators just need to create a new role accompanied with a role assignment context expression and its member privileges.

The predicate roleCondAssign(r, roleCtx) defines the role assignment constraint roleCtx that must be satisfied, to assign the role r to a user. This condition (roleCtx) may contain any context data to match with the user context or the operation context. This way a user-role assignment for unstructured data growth can be supported without compromising security and without much need of system maintenance. This can be expressed as:

dRoleAssign(r) ⇐ roleContextAssign(r, roleCtx) ∧ evalContext(roleCtx, u)

If the predicate dRoleAssign(r) is satisfied against the current user's context whose roles are loaded, the role r can be activated for the user in the current session.

5.1.2.2 Role - Role Assignment

The role hierarchy (RH) is defined as a partial order over the set of roles R ($RH \subseteq R \times R$) [9], i.e. as an inheritance relation, marked as $\succcurlyeq$, where $r_1 \succcurlyeq r_2$ means r_1 inherits r_2. A user u can activate a new role rj only when rj is predecessor of the roles already activated for the user. A formal definition of role assignment in the presence of hierarchy is:

$$\text{canObtainRole}(u, r_j) := \text{isRoleAssigned}(u, r_j) \vee ((r_i \succcurlyeq r_j) \wedge \text{canObtainRole}(u, r_i))$$

5.1.2.3 Privilege assignment/authorization

In a document database, a collection can contain basic type fields, reference type fields or fields containing an embedded document. Thus like roles, data objects can also be hierarchically organized.

Privileges can be associated with a database instance or a group of collections (data group) or a document or set of certain fields in a collection. A Privilege defined for a group applies to all its collections, documents and fields. The object privilege can be defined as: $p_o = (o, op)$; $o \in O$; $op \in Op$, while the group privilege is defined as: $p_g = (g, op)$; $g \in G$; $op \in Op$. So P can be defined as $p_o \cup p_g$.

In case of an embedded document or field containing a reference to another data object, the privilege assignment becomes a little complicated. When a document contains another document within, a user's current privilege can extend to an additional required privilege only if both the collections are part of the same data group or only if their associated contexts are satisfied in case of a compound operation.

We can formulate the logical definition as –

The privilege p_i can be extended by the permission p_j if:

– p_i and p_j are defined for the same operation and data group, or

– p_i and p_j are defined for data objects which are contextually related.

5.1.3 Access definition

We define an access request REQ as <U, O, OP, RTC>; U is an authorized user who initiates the request, O is the data object user wants to access, OP is the operation that needs to be performed, RTC is the set of contexts available in current user session. The access control mechanism makes use of an Access Control policy as a triple, ACP = (S, P, C); S is the subject in this policy, which could be a user or a role with a context attribute; P is the target privilege in this policy, which is defined as a <OP, O>, where OP is an operation defined as {READ, CREATE, APPEND, DELETE, UPDATE}, O is a data object or data group; C is a context expression in this policy.

REQ is granted only if there exists an authorization policy ACP (S, P, C), such that U ∈ S, OP on O can be implied from p ∈ P and C evaluates to true with RTC. We consider the normal administrative user-role and role-permission assignments by security administrators.

5.1.4 Context Evaluation

In this section we define the algorithm for evaluating context supporting the schema discussed in the previous section.

RequestAccess (U , O , OP, RTC)
initialize candidate policy set PS = {}
for every ACP in policy set of the application
    if (U ∈ S in ACP) and (O , OP) ⇐ P in ACP)
        put ACP into PS
    end if
end for
initialize result = "deny"
if RTC = null Call Basic RBAC
else
for every ACP in PS
    if (EvaluateExpression (C in ACP, RTC) is true)
        result = "grant"
        break
    else
        result = "deny"
    end if
end for
return result

EvaluateExpression(ContextExpression Exp, RuntimeContext RTC)
for every clause CL in Exp
    for every condition CD in CL
        evaluate context expression in CD with RTC
        if (CD = false)
            CL = false
            break
        end if
    end for
    if (CL = true) return true
    else continue
end for
return false

The central concept in the above two algorithms is to evaluate a context condition at runtime. We define a context implementation as a service call. There are several ways to implement contexts which can be dynamically evaluated such as with a dynamic link

library or a web service, both of which support dynamic invocation at runtime. In our solution, we model all the required contexts evaluation as web services to facilitate extensibility and interoperability.

Chapter 6

6 Solution verification/evaluation

We have discussed one way to extend a standard RBAC with the help of Context. This approach has been used in complex business processes and workflow systems. But the schema we discussed in the previous section has been minimized to maintain flexibility and simplicity in NoSQL and the Context concept has been designed to run seamlessly with the design principles of document databases. In this section we will evaluate the proposed schema against the example given in Chapter 4 Section 4.

We define several collections in the database, contexts and a role hierarchy for our example scenario. Let us name the collections in the system as:

- Campaign
- Product
- Finance
- Operations

Users can request reports from the reporting service. In our example the objects which must have an access control policy regulated are:

- Campaign Report (Derived mainly from “campaign” collection)
- Product Backlog (Derived from “product” collection)
- Audit Report (Derived mainly from “finance” and “operations” collection)

We can define a document in the ‘Campaign’ collection as presented in Figure 11.

```
Campaign =
{
        _id: {   "$oid" : "528e00cfcc93743934048747"   },
        clientId: {   "$oid" : "528e011fcc93743938528560"   },
        name: { “Holiday Deals” },
        productLine: {
                        [{specline:”S1”, “price”:435},
                        {specline:”S2”, “price”:133}],
                        rebate: 879},
        promoProductLine: {   "$oid" : "528e011fcc93743934048748"   },
        marcomm: {
                region: {   "$oid" : "528e011fcc93743934048800"   },
                quarter: “Q1”,
                deadline: ISODate("2012-10-10T14:00:00Z"),
                reviewer: [{   "$oid" : "528e011fcc93743934048800"   }
                        , {   "$oid" : "528e011fcc93743934048705"   }
                        , {   "$oid" : "528e011fcc93743934048963"   }],
                reviews : [ {
                        reviewer:”….”,
                        issues:”…….”,
                        Internal_suggestions:”….”,
                        date :”…..”},
                        {reviewer:”….”,
                        issues:”…….”,
                        Internal_suggestions:”….”,
                        date :”…..”}],
                approval: { by: ,on: } },
        status: {
                stage: {name:”live”, division:”Marketing”},
                lastmodifiedon: new Date(“Jul 17, 2013”),
                lastmodifiedby :{name:{last:”…”, first:”...”}},
                nextstage: {name:”EOL”, division:”Briefing”}  },
        operation: { manager: {   "$oid" : "528e011fcc93743934044569"   },
                members: [{   "$oid" : "528e011fcc93743934042287"   }
                , {   "$oid" : "528e011fcc56743934078452"   }
                , {   "$oid" : "528e011fcc56743934048711"   }] },
        finance: {budget:9876, opcost:9871, profit:10},
        lastAccess: {on: ISODate("2013-11-22T09:30:00Z"), by: {"$oid" :
"528e011fcc94743934048844" }}, IsArchived:false

        }
```

Figure 11: "Campaing" Collection

A sample hierarchy of roles in the system at any given point of time is depicted in Figure 12. We have “visitor” as minimum role and “Chief Information Officer (CIO)” as maximum role in our example role graph [21]. Roles “Product Lead” and “Global

Finance" have privileges that are are mutually exclusive with authorization-time/complete exclusion restriction. "Operations Lead" is a role that has runtime/shared exclusion (denoted by dashed relations) with each of the roles "Reviewer" and "Regional Finance", where access control rule employs the least privilege principle.

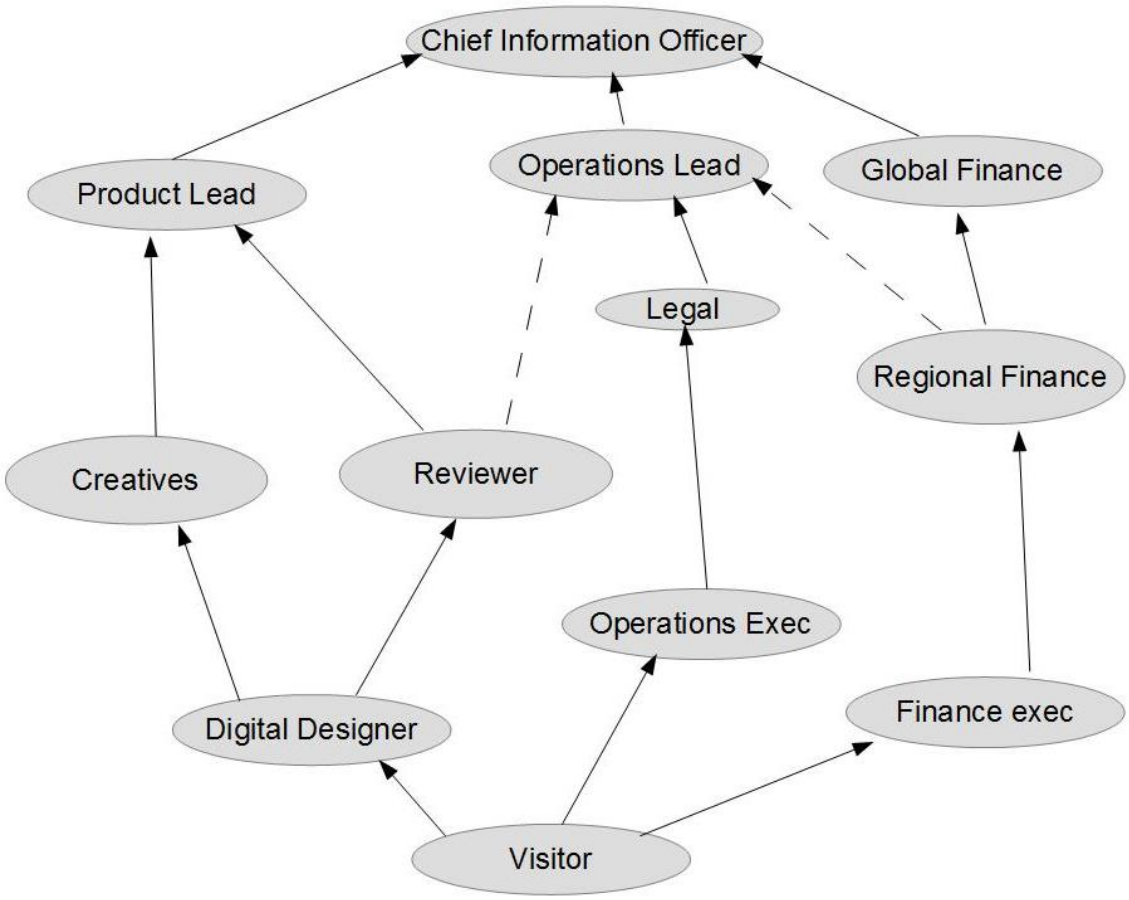

Figure 12: Sample Role Hierarchy with min, max and mutually exclusive

We refer to Figure 13 to explain the access control with respect to the data model and operation flow. The ovals represent the reports the system produces and the rectangles represent the different data objects/groups available in the system. Figure 13 gives us a snapshot of a regional (context: location) data composition. We can see that a campaign report consists of product, operations and finance information; these rectangles in Figure 13 are demonstrated as embedded fields or reference fields in Figure 11. We can derive a

use case from this as when a user has privilege to read a campaign report, he/she may not have a sufficient set of privileges to read a complete campaign because of mutually exclusive roles for accessing financial or operations fields. In an agency a user can have access to all fields of a campaign document but he/she may not be able read all the documents in the campaign collection if the campaign documents do not belong to the same logical location as the user. To access all campaign documents, a user has to have the role “Product Lead” or higher activated in his/her session.

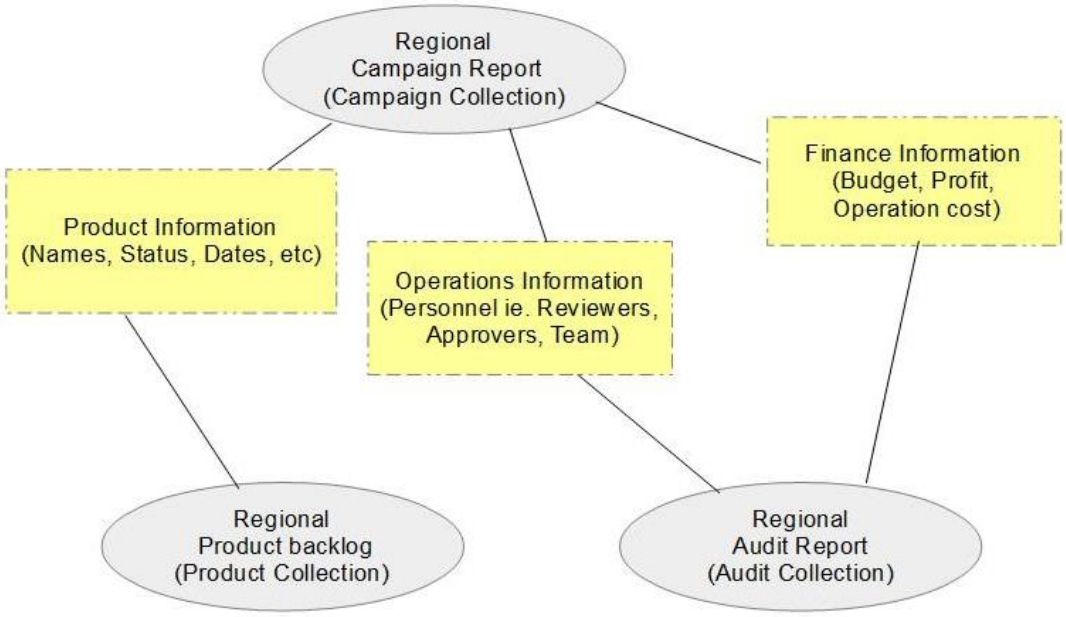

Figure 13: Sample data composition

A Campaign report requires users to have different types of privileges to perform different operations. A user with product lead role can create a campaign report or product backlog, which also implies when this role is activated the user will have the full set of privileges on either reports through its junior roles. But in case of users having roles junior to “Product lead” activated, they may need dynamic role activation depending on the state/context of a campaign document. For example, a user session containing the update privilege for a campaign may need to activate update privileges on

the product backlog if a particular campaign document has the fields referring to product backlog.

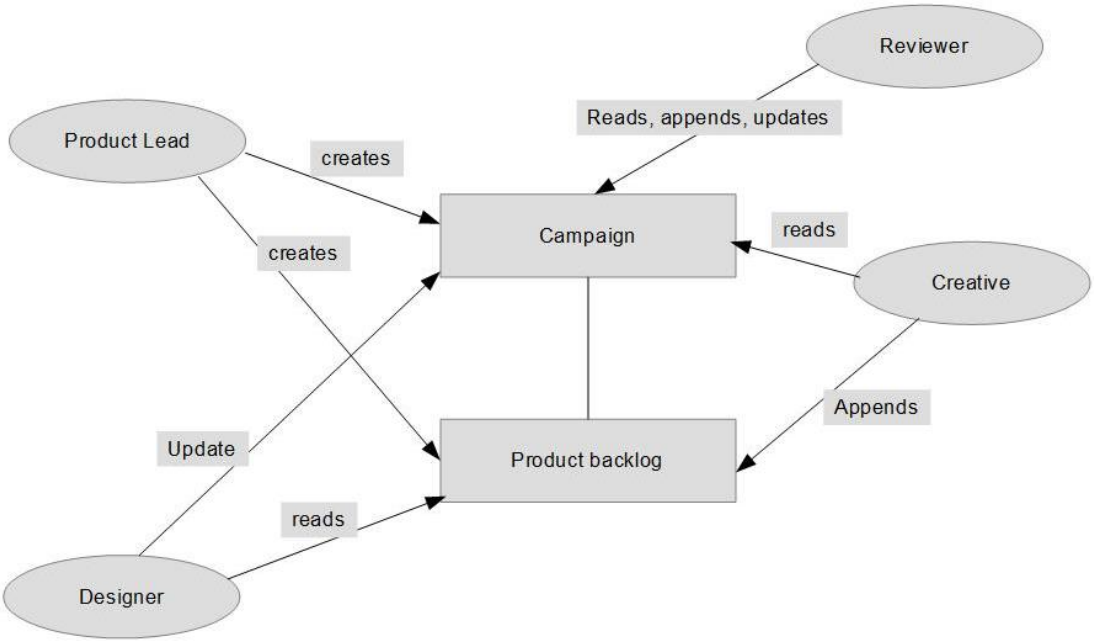

Figure 14: Available privileges on Campaign and Product data

Figure 14 illustrates the different privileges providing different access to these two reports. The Product Lead role has the privilege to create and the Creative role has "append" privilege on product backlogs. Since both of these roles are senior roles of "Digital Designer", both of these roles also have the update, read, and delete permissions on campaign. But a "Digital Designer" role can only have update privileges on campaign report.

One special scenario on access will be that a visitor may get read privilege to agency information of a campaign document if the context of the visitor and the context of the client of the campaign match.

These use-case scenarios demonstrate the purpose of extending RBAC with the notion of context. It represents the cases when certain roles are dynamically assigned to users based on the information contained in documents in a collection. Also, these roles will have the

permissions to execute certain operations only for the specific instances depending on users who acquire these roles.

For the purpose of providing an example we only consider context of the user and context of the data in the context service. The user session contains the logical location of a user as “Region” and the authentication context (Agency, External, Client, etc.), whereas the context for data is multifaceted. Data objects also have the same context terms as users for location, as well as its status, type and sensitivity. The contexts can be evaluated as values or conditions through a context. We show an example set of contexts in Figure 15. Status, Location, Type and Sensitivity are context terms (CT) that can be used in context expressions. Status can have values of ‘Archived’ or ‘EOL’ which are of lowest security level. Status can have values of ‘InPlanning’, ‘InReview’, ‘MarcommApproved’ or ‘Live’; each one is less significant in terms of security than the one following it, thus a context expression for status can be formed with comparison operators. Status and sensitivity represent the state of a campaign document in our example. Location is a logical segregation of business units and campaigns. Users and campaigns are grouped according to the location context to form the same access level data group. A type is a context which is applied to the users of the system.

Status:= Archived || EOL < InPlanning < InReview < MarcommApproved | LegalApproved < Live

Location:= Global > Nordic | (Asia-Pacific-Japan) APJ | Americas | Benelux

Type:= Agency > ExternalAgency > Client > General

Sensitivity:=

0 – Campaigns of 12 month or more old and with non-reference fields | Campaigns with "Archived" Status

1 – Campaigns of previous quarter | Budget/Operations reports from Campaigns with "Archived" Status | Product Backlog of all campaigns with status higher than "InPlanning"

2 – Campaigns with status higher than or equal "InReview" Personnel attributes | Campaigns with Reviews/Approvals

3 – Campaigns with status "Live" with relevant finance attributes

Figure 15: Example set of contexts for a global agency

6.1 Access control enforcement

The information architecture of our example agency can be roughly translated to a set of rules. These rules regulate the data access in the system. We formulate a number of rules from the scenarios mentioned in the previous section.

Rule 1: Visitors can only view campaigns that ran over a year ago. They cannot see any agency internal information. They may get access to their own campaigns minus its 'marcomm' fields if they are clients.

Rule 2: Digital Designers and executives can read, delete and update respectively campaign reports and audit reports of his/her location. Digital Designer can read the product backlog for their location.

Rule 3: Campaign reports and the product backlog can be appended (implies update, delete) to by the creative role for the location of the user.

Rule 4: Reviewers can only append Campaign reports for their location.

Rule 5: Campaign and audit data can be appended to by members of the Legal role in their own location.

Rule 6: Product lead is the only role which can create (imply append) new campaign reports and product backlog.

Rule 7: Operations lead role can create new audit report and access finance. But an Operations Exec member can append operation information.

Rule 8: Finance information in campaign and audit reports can be appended to by Regional Finance within their location whereas only a Global Finance user can append regardless of which location the campaign or audit is for.

A security infrastructure can be modeled taking the use case scenarios and these rules into consideration. This infrastructure can be seen as a means of access control for any application using the document database. The process of access control enforcement is the heart of the security infrastructure which can be performed through three steps:

– Evaluation of context for the subject and the requested object to determine the required privilege set (described in Figure 16).

– Role Activation (described in Figure 17).

– Verification and execution of the requested operation; that the operation is permitted in the current session under respective contexts (described in Figure 18).

The first two steps calculate required permission and activate the appropriate role independently using the corresponding context. The 3rd step takes the outputs of the first two steps and verifies if the user with the activated role contains the privilege to access the requested object. We can model these three steps as functions calls.

```
Function CreatePrivilegeList(Object obj, Operation op, RuntimeContext RTC) :
Privilege
begin
        OC:= ObjectContext(obj, op)
        OPC:= CalculateOperationalContext( OC, RTC)
        Prv:= PrivilegeRequired(obj, op)
        Privilege: = CalculatePrivilegeList(Prv , OPC)
        Return
end
```

Figure 16: Function call for privilege list

The function in Figure 16 evaluates the requestor and resource context to create a dynamic session/operational context. This session context is then used to filter the required set of privileges that needs to be available for the requestor during the execution of the access request.

```
Function ActivateRoles(User u, RuntimeContext RTC) : Void
begin
        UR:= UserRoles(u)
        For each r ∈ UR do
                Evaluate rctx against RTC and state in r to enable or disable r
                if isEnabled (r) is true then
                        ActivateRoleInSession (r)
        Return
end
```

Figure 17: Function call for role activation

```
Function InitializeRequest (User u, Object obj, Operation op, RuntimeContext RTC):
Void
Begin
        P: = CreatePrivilegeList( Object obj, Operarion op, RuntimeContext RTC)
        ActivateRoles (u, RTC)
        Initialize R with Session Roles
        Initialize RP
        RP:= AssociatedRoles(P)
        Excl: = CheckMutualExclusion(R, RP)
        Case: Excl is “no exclusion”
                        ROP:= Intersect( R, RP)
                        Execute operation as ROP //request granted for role ROP
        Case: Excl is “Shared”
                For each r ∈ R do
                        if P is not associated with r then
                                Check for least privilege constraint
                                If satisfies then
                                        Assign(p, r)
                                        Execute operation as r //request granted for role r
                                Else
                                        Throw UnauthorizedAccessRequestException
                                        //request denied
        Case: Excl is “Complete”
                Throw UnauthorizedAccessRequestException //request denied
End
```

Figure 18: Function call verifying request

The first step in Figure 17 is to load the roles assigned to the user, and then, from the set of the loaded roles, it evaluates role context for each role and enables/disables the role accordingly. Basically this means that the context expression is evaluated against runtime

context values to change the state of each role. Depending on the context an enabled role can be turned disabled or a disabled role can be enabled. The algorithm for the activation of roles can be changed in order to meet the requirements of different systems. The above function simplifies role activation where all the enabled roles assigned to the user are activated.

The last step in Figure 18 combines results of the above two calls and does a final verification of the request. The subject can access the object if the permission that allows the particular type of access is contained in the permission authorized in the subject's role. This function takes the privilege list and the roles in the session. Then it checks if the roles activated for the user and the roles associated with the required privileges have any mutual exclusion in terms of roles and permissions both. If there is no exclusion present, then it gets the common set of roles and executes the request on behalf of the role. The intersection of role sets may be empty; in that case the execution routine throws an exception. If there is a shared exclusion, implying that the roles are defined as shared or the privileges associated with each of the set of roles can be associated with the other one only if they satisfy a least privilege restriction defined by the security administrator; this is the runtime/shared mutual exclusion [16]. The request is denied if there is any authorization time complete exclusion defined for the roles in the current session.

Let us take an example request which a random visitor initiates- <"Sam", "Campaign", "find name", ["APJ","Non-client"]>
The high level execution steps for this request according to the rule 1 from Section 6.1 are as follows:

1. Find roles for "Sam"; [{"visitor","location = global",active}]
2. Create list of privileges for finding names of available campaigns.
    2.1. Object context for finding name of Campaign: [campaign.marcomm.region = RTC→location] and [campaign.status.stage <for> RTC→userStatus] or [...] (<for> is a user-defined function)

2.2. Evaluate context: [campaign.marcomm.region = “APJ” and campaign.status.stage = “Archieved”]
2.3. Privilege required: [“read-archived”, “read-basic”, ”read-all”, ”read-product”, ”read-marcomm”,”…”]
2.4. Calculate privilege list for current session: [“read- archived”]

3. Available roles containing “read- archived” privilege: [{“visitor”, ”location = global”, enabled},[{“visitor”, ”location = APJ”, enabled}] (inherited by all other roles in ‘Global’ and ‘APJ’ context)
4. Roles in step 1 and 3 are not mutually exclusive and intersect is not empty so the request gets executed.

We will discuss one more use-case when an agency user initiates a request as:

<”Aaron”, “Campaign”, “Update price for spec S1”, [“Benelux”,”Agency”,"sesitivity: 2"]>

The high level execution steps for this request according to the rules from Section 6.1 are as follows:

1. Find roles for “Aaron”; [{“Operations Lead”,”location = Benelux",Active}, {“Legal”,”location = Nordic",disabled},{“Reviewer”,”location = Global",Active}]
2. Create list of privileges for updating price for product in campaign.
   2.1. Object context for update in product in campaign: [campaign.marcomm.region = RTC→location] and [status.stage < live] or [sensitivity<= 2]
   2.2. Evaluate context: [campaign.marcomm.region = “Benelux” and sensitivity<=2 is true]
   2.3. Privilege required: [“read-product”, ”update-campaign”, "read-finance"]
3. Available roles containing either of the [“read-product”, ”update-campaign”, "read-finance] privileges: [{“Digital Designer”, ”location = Benelux”, enabled},{“Reviewer”, ”location = Benelux”, enabled}, {“Regional Finance”, ”location = Benelux”, enabled}] (according to rule 3 and 8)
4. Roles “Operations Lead” and “Regional Finance” have shared mutual exclusion in terms of privilege.

4.1. Role intersection doesn’t occur.

4.2. {“Regional Finance”,”location = Benelux”, enabled} role can be assigned to “Aaron” after checking least privilege restriction e.g. “Aaron” cannot have all the privileges to append finance information and change a legal (inherited from Legal role which is junior to Operations Lead) approval in the same session.

5. The request is granted

Chapter 7

7 Conclusions and Future Work

The model presented in this thesis acts as an extension of RBAC [10]. Even though context sensitivity has been suggested as an entity in security frameworks for web architectures, one might debate the use of recent developments in access control. The main drawback in RBAC is the inadequate role granularity for fine-grained authorization which leads to role explosion. Role administration gets exponentially complex for precise and fine grained user-role and role-permission assignment. Eventually the number of extensions to the core RBAC models increases to allow provision for situational factors e.g. time, location, task etc. [15, 17, 27].

7.1 Discussion and Conclusions

Intuitively, an attribute is a property in data that can be viewed as context. Sandhu [27] defined attributes as security labels, clearances and classifications (LBAC), identities and access control lists (DAC) and roles (RBAC). He claims that attribute based access control (ABAC) compliments RBAC by making provision for additional attributes such as location, time of day, strength of authentication, departmental affiliation, qualification, frequent flyer status, and so on into the framework. This endorsement of attributes in access control seems to be a good fit for NoSQL where a security level changes on the same data object depending on the elements it consists of at run time. An attribute based security policy may work better with environmental contexts when a cloud service model or workflow is considered; but NoSQL and its nonconformance of structure inside a data object presents unlimited combination of attributes leading to the same role or permission explosion as with the basic RBAC. Sandhu [27] cautions in his paper that ABAC with its flexibility may further confound the problem of role design and engineering. With this we can conclude that when NoSQL is of concern a context sensitive RBAC provides flexible, dynamic and light weight security without creating more complex framework which is the design principle for any NoSQL supporting application.

Most of the popular NoSQL package gets shipped with coarse-grained security mechanism. To mitigate the security concerns raised by the lack of proper access control, we opted for a model which dynamically allows and restricts access to inherently complex data objects. Beside basic privacy ensured by RBAC, our model has been evaluated to support dynamic assignment of roles to users and dynamic assignment of permissions to roles depending on run-time contexts. This access control depends on different context factors, which vary with the data it protects. In this thesis we present a context based access control model that supports these requirements.

Lacking the structure of RDBMS, NoSQL adopts an open principle when it comes to designing an application. This results in access control for such applications varying depending on different unpredictable factors. A possible solution for this problem is to adopt a forward looking security concept, for example context of data, operations and users. The context is used to handle access control requirements which evolve as the nature of data change over the course of time.

The dynamic context based access control model discussed in this thesis extends the traditional RBAC model and gains advantages from its context sensitive parameter. Our research motivation comes from the complicated access control requirements inherent to schemalessness of NoSQL database. Traditional RBAC is not able to guarantee a sufficiently fine-grained access control or specify constraints that should be applied to an access policy for non-structured data. We incorporated the support for context-sensitive RBAC to introduce access control and contribute towards a much needed privacy framework in the relatively new NoSQL database.

7.2 Future Work

The main focus of our model is to ensure access control and we discussed how it is applied in a single application. But web applications today are highly collaborative and thus the provision for access control to address security issues raised by service to service communication also need to be taken into account. With our model, most context types can be implemented as web services located within the enterprise, but there are use cases where a third party outside a system perimeter may need to configure contexts. In our

future version of model we plan to incorporate mechanism to address context-sensitivity when the requestor is a service outside the system by extending single domain RBAC to multi domain. An external service request needs to acquire proper membership in the authorization process. We plan to utilize WS-SecureConversation; a Web Services specification, created by IBM and others, that works in conjunction with WS-Security, WS-Trust and WS-Policy to allow the creation and sharing of security contexts [44] to evaluate the aforementioned requirements.

NoSQL databases provide good admin-level security for their replication sets and auto-sharding. Because of that, the model discussed in this thesis doesn't have the provision for context sensitivity for the sharding or replication concept. But it needs to be addressed in its future version, as without providing secure and detailed control on data growth, any access control model for NoSQL is not complete. Our initial model ensures security for data in storage not the growth itself.

References

[1]Ahn, Gail-Joon, and Ravi Sandhu. "Role-based authorization constraints specification." *ACM Transactions on Information and System Security (TISSEC)* 3.4 (2000): 207-226.

[2] Bertino, Elisa, Piero Andrea Bonatti, and Elena Ferrari. "TRBAC: A temporal role-based access control model." *ACM Transactions on Information and System Security (TISSEC)* 4.3 (2001): 191-233.

[3] Bhatti, Rafae, Elisa Bertino, and Arif Ghafoor. "A trust-based context-aware access control model for web-services." *Web Services, 2004. Proceedings. IEEE International Conference on*. IEEE, 2004.

[4] Brewer, Eric A. "Towards robust distributed systems." Portland, Oregon, July 2000. *Keynote at the ACM Symposium on Principles of Distributed Computing (PODC).* http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[5] Cattell, Rick. High Performance Scalable Data Stores. February 2010. – http://cattell.net/datastores/Datastores.pdf

[6] Chakraborty, Sudip, and Indrajit Ray. "TrustBAC: integrating trust relationships into the RBAC model for access control in open systems." *Proceedings of the eleventh ACM symposium on Access control models and technologies*. ACM, 2006.

[7] Cholewka, Damian G., Reinhardt A. Botha, and Jan HP Eloff. "A context-sensitive access control model and prototype implementation." *Information Security for Global Information Infrastructures*. Springer US, 2000. 341-350.

[8] Damiani, Ernesto, et al. "Fine grained access control for SOAP E-services." *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001.

[9] Dan Pritchett. BASE: An Acid Alternative, May 1, 2008. - http://queue.acm.org/detail.cfm?id=1394128

[10] Ferraiolo, David F., et al. "Proposed NIST standard for role-based access control." *ACM Transactions on Information and System Security (TISSEC)* 4.3 (2001): 224-274.

[11] Dey, Anind K. "Understanding and using context." *Personal and ubiquitous computing* 5.1 (2001): 4-7.

[12] Ferraiolo, David F., John F. Barkley, and D. Richard Kuhn. "A role-based access control model and reference implementation within a corporate intranet." *ACM Transactions on Information and System Security (TISSEC)* 2.1 (1999): 34-64.

[13] Hoff, Todd. “A Yes for a NoSQL Taxonomy”. November 2009. –. http://highscalability.com/blog/2009/11/5/a-yes-for-a-nosql-taxonomy.html

[14] Ippolito, Bob. Drop ACID and think about Data. March 2009. http://blip.tv/file/1949416/

[15] Jin, Xin, Ravi Sandhu, and Ram Krishnan. "RABAC: role-centric attribute-based access control." *Computer Network Security*. Springer Berlin Heidelberg, 2012. 84-96.

[16] Kuhn, D. Richard. "Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems." *Proceedings of the second ACM workshop on Role-based access control*. ACM, 1997.

[17] Kuhn, D. Richard, Edward J. Coyne, and Timothy R. Weil. "Adding attributes to role-based access control." *IEEE Computer* 43.6 (2010): 79-81.

[18] Kulkarni, Devdatta, and Anand Tripathi. "Context-aware role-based access control in pervasive computing systems." *Proceedings of the 13th ACM symposium on Access control models and technologies*. ACM, 2008.

[19] Leavitt, Neal. "Will NoSQL databases live up to their promise?." *Computer* 43.2 (2010): 12-14.

[20] Li, Ninghui, Mahesh V. Tripunitara, and Ziad Bizri. "On mutually exclusive roles and separation-of-duty." *ACM Transactions on Information and System Security* (TISSEC) 10.2 (2007): 5.

[21] Nyanchama, Matunda, and Sylvia Osborn. "The role graph model and conflict of interest." *ACM Transactions on Information and System Security (TISSEC)* 2.1 (1999): 3-33.

[22] Popescu, Alex. Presentation: An Interesting NoSQL categorization. February 2010. – http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interestingnosql

[23] Ray, Indrakshi, Mahendra Kumar, and Lijun Yu. "LRBAC: A location-aware role-based access control model." *Information Systems Security*. Springer Berlin Heidelberg, 2006. 147-161.

[24] Pramod J. Sadalage, Martin Fowler, NoSQL Distilled, ISBN-10: 0321826620, Publication Date: August 18, 2012

[25] R. Sandhu. , et al. "Role-based access control models." Computer 29.2 (1996): 38-47.

[26] Sandhu, Ravi. "Role hierarchies and constraints for lattice-based access controls." *Computer Security—ESORICS 96*. Springer Berlin Heidelberg, 1996.

[27] Sandhu, Ravi. "The authorization leap from rights to attributes: maturation or chaos?." *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*. ACM, 2012.

[28] Scofield, Ben: NoSQL – Death to Relational Databases(?). January 2010. – *Presentation at the CodeMash conference in Sandusky (Ohio).* http://www.slideshare.net/bscofield/nosql-codemash-2010

[29] Shen, H. and Hong, F., “A context-aware role-based access control model for web services," *In Proceedings of the IEEE International Conference on e-Business Engineering*, Beijing, China 2005.

[30] Sladic, Goran, Branko Milosavljevic, and Zora Konjovic. "Modeling context for access control systems." *Intelligent Systems and Informatics (SISY), 2012 IEEE 10th Jubilee International Symposium on*. IEEE, 2012.

[31] Wang, Jingzhu, and Sylvia L. Osborn. "A role-based approach to access control for XML databases." *Proceedings of the ninth ACM symposium on Access control models and technologies*. ACM, 2004.

[232] Zhang, Xinwen, Sejong Oh, and Ravi Sandhu. "PBDM: a flexible delegation model in RBAC." *Proceedings of the eighth ACM symposium on Access control models and technologies*. ACM, 2003.

[33] Yen, Stephen: NoSQL is a horseless carriage. November 2009. – http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf

[34] Oracle NoSQL Database White Paper, http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf

[35]MongoDB Manual, November 21, 2013. http://docs.mongodb.org/v2.4/MongoDB-manual.pdf.

[36] http://www.christof-strauch.de/nosqldbs.pdf

[37] http://www.mongodb.com/document-databases

[38] http://www.mongodb.com/presentations/schema-design-12 (to 14)

[39] Big data, http://www.mongodb.com/solutions/big-data

[40] Couchbase, http://www.couchbase.com/why-nosql/nosql-database

[41] MongoDB, Inc. http://www.mongodb.com/

[42] WSDL, http://msdn.microsoft.com/en-us/library/ms996486.aspx

[43] http://docs.mongodb.org/manual/reference/sql-comparison/

[44] http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/ws-secureconversation.html

Appendices

Appendix A: SQL to MongoDB Mapping Chart

Table 5: Terminology and Concepts [43]

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline

Table 6: DDL Query [43]

SQL Schema Statements	MongoDB Schema Statements
CREATE TABLE users (id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id))	db.createCollection("users") or db.users.insert({ user_id: "abc123", age: 55, status: "A" })
ALTER TABLE users ADD join_date DATETIME	db.users.update({ }, { $set: { join_date: new Date() } }, { multi: true })
ALTER TABLE users DROP COLUMN join_date	Collections do not describe or enforce the structure of

SQL Schema Statements	MongoDB Schema Statements
	its documents; Still one can use $unset – `db.users.update( { }, { $unset: { join_date: "" } }, { multi: true } )`
CREATE INDEX idx_user_id_asc ON users(user_id)	db.users.ensureIndex({ user_id: 1 })
CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)	db.users.ensureIndex({ user_id: 1, age: -1 })
DROP TABLE users	db.users.drop()

Table 7: DML Query [43]

SQL SELECT Statements	MongoDB find() Statements
SELECT * FROM users	db.users.find()
SELECT id, user_id, status FROM users	db.users.find({ }, { user_id: 1, status: 1 })
SELECT user_id, status FROM users	db.users.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM users WHERE status = "A"	db.users.find({ status: "A" })
SELECT user_id, status FROM users WHERE status = "A"	db.users.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })

SQL SELECT Statements	MongoDB find() Statements
SELECT * FROM users WHERE status != "A"	db.users.find({ status: { $ne: "A" } })
SELECT * FROM users WHERE status = "A" AND age = 50	db.users.find({ status: "A", age: 50 })
SELECT * FROM users WHERE status = "A" OR age = 50	db.users.find({ $or: [{ status: "A" } , { age: 50 }] })

Curriculum Vitae

Name:	Motahera Shermin
Post-secondary Education and Degrees:	University of Dhaka Dhaka, Bangladesh 2000-2004 B.Sc. The University of Western Ontario London, Ontario, Canada 2012-2013 M.Sc.
Honours and Awards:	Western Graduate Research Scholarship (WGRS) 2012-2013
Related Work Experience	Teaching Assistant University of Western Ontario 2012 - 2013 Development Team Lead, SoftwarePeople, Sep 2010 – Aug 2012 Sr. System Developer, SoftwarePeople, Sep 2009 – Aug 2010 .Net Developer (MVC), Obout Inc. November 2009 - July 2011 System Developer, SoftwarePeople, Mar 2008 – Aug 2009 Development Engineer, Pageflakes, Aug 2006 – Jan 2008 Software Engineer, Somewhere In Ltd, Mar 2006 – Feb 2007 Web developer, Zanala Bangladesh, 2004 - 2006