K-VAC: An Access control Model for Non-relational (NoSQL) Systems

Devdatta Kulkarni
devdattakulkarni@gmail.com

May 27, 2012

Abstract

Many modern Internet-scale applications use non-relational data systems ("NoSQL" systems) for their data storage needs. Several such systems have been built recently, such as Cassandra, HBase, MongoDB. While such systems provide advantages such as scalability over traditional relational databases, in comparison, their access control models are limited in their functionality. The current access control models of such systems do not support complex access control requirements, such as restricting users' access to certain keys or values, or restricting access based on context conditions or user roles. In this paper we present K-VAC – key-value access control model for modern non-relational data systems. This model supports specification and enforcement of access control policies at different levels of resource hierarchy. The policies can be based on context information or the contents of a column. We have implemented this model as an application-level access control library which can be integrated with applications at design time. We present the bindings of this library for Cassandra, HBase, and MongoDB. The library leverages the underlying key-value system to store and manage multiple versions of access control policies. Through case-study applications we demonstrate the capabilities of this system. [1]

1 Introduction

Modern Internet-scale cloud-based applications are using NoSQL data stores (key-value stores) as their back end data systems. Many such NoSQL systems have been built recently, such as Cassandra, MongoDB, HBase, CouchDB. Such systems are typically non-relational in nature and provide several advantages over traditional databases. They are more scalable, their schema-less design supports ease of adding new attributes to data items, and some of them, such as Cassandra, also relax the data consistency requirements, following the eventual

[1] Author affiliation: Rackspace, Inc. This work has been done independently, and is not related to author's work at Rackspace.

consistency model. Given such advantages, such systems are bound to be used in more and more large scale distributed applications in the future. However, the access control capabilities of such systems are currently limited.

The focus of this paper is on the issues related to building fine-grained access control models and systems for key-value stores. In such systems data is stored in the form of rows of key-value pairs. Each data item has a unique row key. The various attributes associated with the data item are stored as columns associated with that row key. We identify the following access control issues for such systems.

- **Hierarchical access control:** This refers to the ability of the access control system to be able to specify access control policies for data at different levels of hierarchy within a key-value store. We should be able to specify access control policies at the granularity of the entire key-value store, or a grouping of some row, or on a specific row, or a specific column.
- **Content-based access control:** In some cases, we may need to restrict access to certain resources based on the *values* of certain keys or their attributes within a key-value store.
- **Context-based access control:** In certain situations the permissions granted to a user over certain resources may need to be constrained based on context conditions, such as a user's location, who a user is co-located with, and time.
- **Permission Versioning:** This refers to the ability of the access control system to maintain versioning information for the permissions defined in the system. This requirement arises from the need of the cloud computing applications to be "auditable" in order to provide adequate confidence in the cloud platform to the clients.

Towards addressing these issues, in this paper we make the following contributions. We present an access control model for key-value stores (K-VAC). In this model access control policies can be specified at the level of the entire key-value store, or at the level of logical groupings of certain rows, or at the level of individual rows and columns. These policies can be based on user identity, or on certain column values, and may also include context conditions.

We present the design and implementation of an application level library realizing K-VAC. This library can be integrated with applications at design time. We have developed bindings of this library for three different key-value stores namely, Cassandra, HBase, and MongoDB. Through two case-study applications we demonstrate the applicability of K-VAC. We make the K-VAC library available to the community [7].

This paper is organized as follows. In Section 2 we present a case-study application which we use to discuss the various access control issues for key-value stores. In Section 3 we present the K-VAC model. In Section 4 we present the design and implementation of the K-VAC library. In Section 5 we present

K-VAC's evaluation. In Section 6 we reflect on the K-VAC system. In Section 7 we compare our work with the related work and conclude in Section 8.

2 Access control requirements for key-value stores

Consider a patient information system hosted on a dedicated cloud infrastructure of some hosting provider. Such a system may contain different kinds of information. We concentrate here on information associated with patients, doctors, and nurses. Each patient has several different kinds of information associated with him or her. This includes, a patient's current medications, current doctor, care-giver nurse, medical history, emergency contact information, billing address. The information associated with doctors and nurses include, information about who their current patients are, their contact information, and their work hours. Additionally, information associated with doctors include, nurses that are assigned to them and the patient reports that a doctor would create. Similarly, additional piece of information associated with nurses includes which doctors they are assigned to. One piece of information that is associated with all three is their location. This may correspond to a room or a hospital ward where a doctor, a nurse, or a patient may be located. We consider the following access control requirements that such a system may need to support [1].

[R1] A doctor can access medical information of only those patients who are currently assigned to them.

[R2] A patient's current medications can only be accessed by the patient's doctors.

[R3] A nurse is allowed to check a patient's medical history only during her work hours.

[R4] A nurse can access information of only those patients corresponding to the ward in which the nurse is currently present.

[R5] A nurse is allowed to access a doctor's patient reports only while the nurse and the doctor are present in the patient's room.

We now describe how such an application can be supported using a key-value system. We use Cassandra [4] as a representative key-value system in this paper. First, we give a brief overview of Cassandra.

In Figure 1 we present the Cassandra data model. The top-level is called the *Keyspace*. It provides a grouping for all the keys that are stored in that Keyspace. The next level is called the *Column Family*. A column family is an abstraction for grouping together various row keys. It is analogous to a table in traditional databases. Within a Keyspace there can be multiple column families. Within a column family, data items are stored in the form of key-value pairs. Each data item is made up of a unique element (the row key) and number of other non-unique elements (columns). The columns in a column family can also be grouped together into an optional *super column*. In Cassandra, data is stored on a cluster which is made up of multiple storage nodes organized in the form of a DHT-style ring. The row key of a data item determines what node the data item is stored on. Logically, the Cassandra data model is organized as a

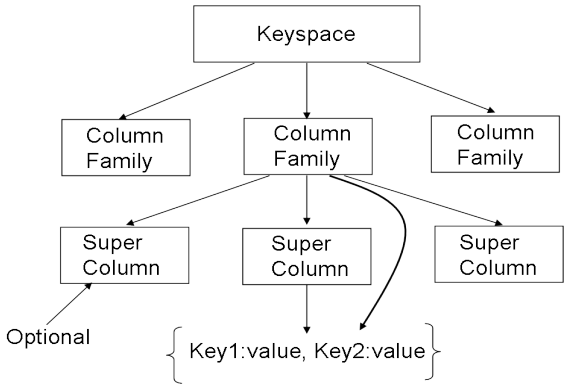

Figure 1: Cassandra data model

hierarchy.

Using Cassandra, the patient information system can be modeled as follows. We define a keyspace named PI, and define three column families (c.f.) corresponding to patients, doctors, and nurses.

[Patient c.f.:] id, curr_medications, medical_history, emergency_contact, billing_address, patient_rep, location, curr_doctor

[Nurse c.f.:] id, curr_patients, contact_info, work_hours, doctors_assigned_to, location.

[Doctor c.f.:] id, curr_patients, contact_info, work_hours, nurses_assigned, location.

The *id* attribute is the row key in each of the column families. The following attributes may have multiple values: *curr_patients*, and *doctors_assigned_to* in the *Nurse* column family, and *curr_patients*, and *nurses_assigned* in the *Doctor* column family. Moreover, the values of these attributes are the *ids* from the corresponding column families.

Let us consider how the access control requirements R1 to R5 can be realized in the above Cassandra data model. For R1, we need that a doctor should be able to access rows in the *Patient c.f.* for only those patients whose ids are present in the *curr_patients* column corresponding to that doctor's row in the *Doctor c.f.*. This means that we need to be able to restrict access to specific rows within a column family based on the contents of other columns in a different column family. For R2, we need that the *curr_medications* column in the *Patient c.f.* for a patient can be accessed only by a doctor whose *curr_patients* contains that patient's id. This means we need to be able to restrict access to a specific column within a column family. For R3-R5, we need to be able to express context-based access control requirements. For R3, we need to restrict access based on the time of the day, for R4 and R5, the access needs to be restricted based on the locations of nurses, doctors, and patients.

Based on this we see that the access control model needs to support the following. First, the model needs the ability to specify access control policies

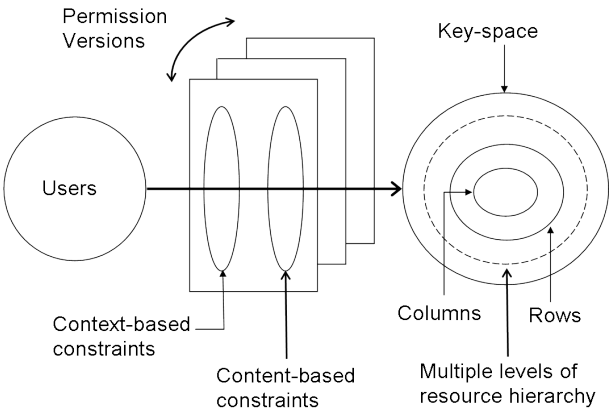

Figure 2: Key-Value Access Control Model (K-VAC)

at different levels of data model hierarchy. Policies may need to be specified at the level of a key store, or at the level of a row, or a column. Second, the model needs to support specifying policies that are based on *contents* of a row or a column. Third, policies also need to be based on context information, such as a person's location, or a person's co-location with other person, or time.

3 Key-Value Access Control Model

To address the access control requirements identified above, we have developed a key-value access control model (K-VAC) as shown in Figure 2. The main elements of this model are users, resources, permissions, and constraints. The resources are organized in a logical hierarchy. Permissions can be specified at the level of the entire key-value store, or at the level of rows (keys), or at the level of columns (values). The dotted circle indicates that there could be multiple levels of hierarchical resource groupings encapsulating rows and columns. For example, in Cassandra the resource hierarchy from outermost to the innermost consists of keyspace, column_family, rows, columns.

In K-VAC we define two kinds of constraints, context-based constraints and content-based constraints. These are used to specify context-based and content-based access control policies, respectively. A permission is successfully executed only if these constraints are satisfied.

3.1 Model constructs

An access control policy in K-VAC is defined as the following k-vac expression: <*permission_type*, *resource_expression*, *constraint_expression*>

The access specified as *permission_type* is granted to the user on the resource specified in the *resource_expression* only if the constraint specified in the

constraint_expression is true.

3.1.1 Permission

A *permission_type* could be one of *read row*, *read column*, *write row*, or *write column*.

3.1.2 Resource Expression

A *resource_expression* identifies the resource on which the access control policy is specified. Examples of a resource are, a keyspace, a column family, a specific row, or a specific column. A resource_expression is specified in the following format, which is inspired from the XPath expressions [9].

/keyspace/column_family([row_key[=(val)])/
[column_key[=val]]

Square brackets indicate that the specification of a value for a row_key or a column_key is optional. If a value is specified, we select only the row or the column that has the specified value. We support three different mechanisms for specifying the value. First, a value can be specified literally. For example, consider the patient information system presented in Section 2. Selecting row for the patient named John would be specified as follows:
/PI/Patient(key=John).

Second, a value can be specified to be obtained by querying a column. For example, selecting row for a doctor who is operating on a patient John would be specified as follows:
/PI/Doctor(key=/PI/Patient(key=John))/curr_doctor

The third mechanism for specifying a value is in the form of a *parameterized variable*. Such a variable is specified as a parameter within a specification. It is explicitly bound to a value at runtime. For example, consider that we want to specify that select a row for a doctor who is operating on a patient whose name is provided at runtime. This requirement would be specified as follows:
/PI/Doctor(keys=/PI/Patient(key=$patient_name))/curr_doctor.
Here *patient_name* is a parameterized variable.

3.1.3 Constraint Expression

The *constraint_expression* specifies context-based or content-based constraints. A constraint specification consists of logical conditions over resources defined in the system. Correspondingly, a *constraint_expression* may contain one or more *resource_expressions* identifying such resources.

A constraint_expression is defined as follows:

constraint_expression := **condition** *operator operand operand.*

Currently we support the following operators: *in, equal, and, or, minus.* An operand can be a *resource_expression*, current time, or an integer value.

3.2 Modeling Patient Information System

We now show how the various access control requirements identified in Section 2 can be specified using k-vac expressions.

3.2.1 Requirement R1

The k-vac expression shown in Figure 3 is used to specify the access control requirement R1. The resource over which the permission is specified is the row key of the Patient column family. The *constraint_expression* specifies that a Doctor is allowed to read a particular patient's records if that patient's id appears in the curr_patients column corresponding to the row in the Doctor column family whose id attribute matches the id of the user who is requesting the access. In our system, each user is mapped to a runtime object called *user* with the id attribute set to the identity of the user.

read row /PI/Patient/id
condition
/PI/Patient/id *in*
/PI/Doctor/id=(user.id)/curr_patients

Figure 3: K-VAC expression for R1

3.2.2 Requirement R2

The k-vac expression shown in Figure 4 is used to specify the access control requirement R2.

read column /PI/Patient/id/curr_medications
condition
/PI/Patient/id *in*
/PI/Doctor/id=(user.id)/curr_patients

Figure 4: K-VAC expression for R2

In Figure 4, read access to a specific patient's curr_medications column is granted to a doctor, if that patient's id appears in the curr_patients column of that doctor.

3.2.3 Requirement R3

The k-vac expression shown in Figure 5 is used to specify the access control requirement R3. A nurse is granted access to a patient's medical history column only if current time is within her work hours.

The variable *current_time* is a special variable, and is evaluated at runtime to obtain the current time.

read column /PI/Patient/id/medical_history
condition
current_time *in*
/PI/Nurse/id=(user.id)/work_hours

Figure 5: K-VAC expression for R3

3.2.4 Requirement R4

The k-vac expression shown in Figure 6 is used to specify the access control requirement R4. A nurse is granted read access to a patient's row if the value of the location column for that patient matches the location of the nurse who is requesting the access.

read row /PI/Patient/id
condition
/PI/Patient/id/location *equal*
/PI/Nurse/id=(user.id)/location

Figure 6: K-VAC expression for R4

3.2.5 Requirement R5

The k-vac expression shown in Figure 7 is used to specify the access control requirement R5. A nurse is granted read access to a patient's report if the value of the location column for that patient matches the location of the nurse who is requesting the access and also the location of the nurse matches the location of the patient's doctor.

read column /PI/Patient/id/patient_rep
condition
/PI/Patient/id/location *equal*
/PI/Nurse/id=(user.id)/location *and*
/PI/Nurse/id=(user.id)/location *equal*
/PI/Doctor/id=(/PI/Patient/id/curr_doctor)/location

Figure 7: K-VAC expression for R5

4 Design and Implementation

In implementing the K-VAC system, we investigated two options in the design space, as shown in Figure 8.

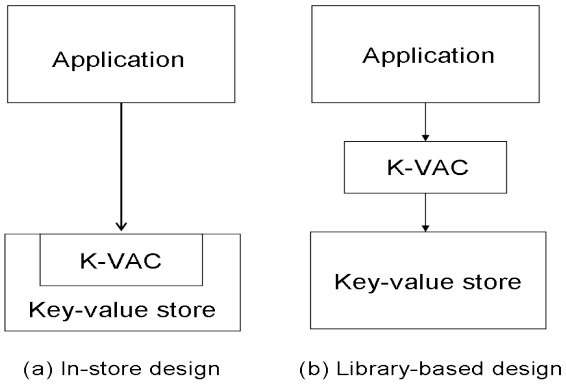

Figure 8: K-VAC design options

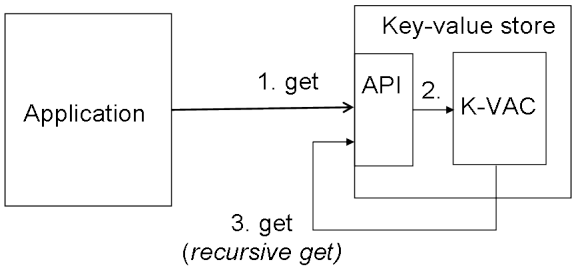

Figure 9: Recursive GET

4.1 In-store design

One design option, which is presented in part (a) of Figure 8, is to implement the K-VAC layer within the key-value store itself. The advantage of this approach is that an application does not have to be aware of the K-VAC layer. It interfaces with the underlying key-value store as it would normally. The key-value store enforces the specified K-VAC policies. We modified Cassandra's implementation to perform access control based on K-VAC using this approach [6]. However, there are at least two problems with this approach.

4.1.1 Recursive Get

The first is the *recursive get* problem. A *recursive get* is a *get* operation that is invoked as part of an already on-going *get* operation. Such operations arise when K-VAC is implemented within a key-value store itself, and they occur due to the following reason. The key-value store can begin the enforcement of an access control policy only after a user has made a call to access a resource, say, using a *get* operation. However, in order to enforce K-VAC policies, the key-value store may have to execute one or more *get* calls in order to find information that is needed for policy evaluation. For example, in the K-VAC expression

of Figure 4, there would be two *get* calls performed. The initial *get* would be on the *curr_medications* column for a particular patient. As part of this invocation, the access control system would make one more *get* call to find out the *curr_patients* for the doctor who is trying to access the *curr_medications* resource. From access control perspective, how should the key-value store handle this second *get* (a recursive get)? Should access control be performed on this resource (*curr_patients*) or not? This is the *recursive get* problem and is shown in Figure 9.

There are two approaches to handle the recursive get problem. One approach is to modify the design of the system to use different interfaces for accessing resources required as part of access control policy enforcement. Such interfaces would not have any access control enforcement in their execution. However, such interfaces would provide an attack vector thereby compromising the security of the key-value store. Other issues with this approach are, possible code duplication and cluttering of the public APIs of the system with methods that behave similarly.

The other approach to address recursive get problem is to design the access control layer such that it detects when a *get* is being invoked as part of another *get*. When such a situation is detected, the system should bypass any access control checks on such a recursive get. This option does not require addition of new interfaces to the public API of the key-value store, and hence is very appealing.

This is the approach that we took in implementing the K-VAC layer within Cassandra [6]. In the Cassandra code, we added a class called *KVACAuthority* which implements the K-VAC model. This class implements the *IAuthority* interface of Cassandra. In order to use this authority we modified the *DatabaseDescriptor* class to create *KVACAuthority* as the underlying authority, instead of the default *SimpleAuthority*. Finally, we modified the *get* and *get_slice* methods of the *CassandraServer* class to use *KVACAuthority*. The *KVACAuthority* detects when it is being invoked as part of an already on-going policy evaluation call. When such a condition is detected it returns without performing any access control checks on the recursive call.

4.1.2 Violation of the DRY Principle

The second problem with the *in-store* design approach is that it violates the *DRY* principle of software design [10]. DRY stands for *Don't repeat yourself* [10] and refers to the fact that there should be a single place in a software system where code for certain functionality should exist. In the design option where K-VAC layer is implemented within a key-value store, we would need to implement this layer for each key-value store separately. In loose terms we will be violating the DRY principle since we will be repeating the implementation of the K-VAC system across different key-value stores.

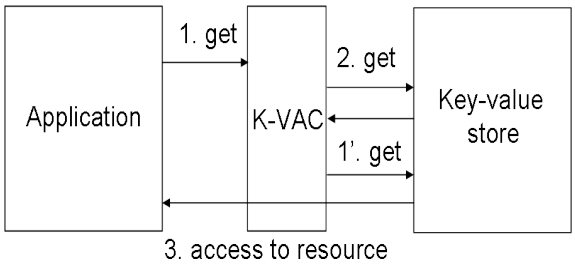

Figure 10: Library-based GET

4.2 Library-based design

The second design option is to implement K-VAC outside a key-value store as a library. The library-based design for K-VAC system is shown in part (b) of Figure 8. This is the approach that we have chosen for implementing the K-VAC system. We have implemented K-VAC library in Java.

There are several advantages of this library-based approach. First, no change is required to the key-value store. Second, there is no recursive get problem, since the K-VAC library initiates all read (*get*) calls corresponding to K-VAC policy enforcement. This is shown in Figure 10. Third, the same library can be interfaced with the different key-value stores. We have done this, and have implemented the library's bindings for Cassandra, HBase, and MongoDB. The drawbacks of this approach are that an application is not directly interfaced with the key-value store, which may affect its performance.

4.2.1 Elements of the Model

We use XML for representing the K-VAC policies. In Figure 11 we show the form of this policy. This has one-to-one correspondence with the *k-vac_expression* defined in Section 3. A policy consists of specification of a resource, a permission, and a condition. The specified permission is allowed to be invoked on the resource only if the condition is satisfied. The type of the resource, whether a row or a column, is specified through the *type* attribute of the *resource* element. The resource itself is specified as the value of the *resource* element and using *resource_expression* for this purpose. The *value* element, which is a child of the *permission* element, identifies the type of permission, either a *read* or a *write*. The *condition* element is used to specify the *constraint_expression*.

In Figure 12 we present the architecture of this system. The K-VAC library consists of two main components, the policy decision point (PDP), and the policy enforcement point (PEP).

4.2.2 Policy Decision Point

The PDP intercepts an application's request to access a resource, and determines the policy that is applicable for that <resource, permission> combination. Poli-

```
<policy>
        <resource type=...>
                .....
        </resource>
        <permission>
                <value>...</value>
                <condition>...</condition>
        </permission>
</policy>
```

Figure 11: K-VAC Policy Template

cies can be stored in a file store, or in the key-value store itself. If the policies are stored on a file system, we read them at the application start-up time and then store them in the key-value store. In Cassandra, we create a special column family called *Permissions* and store <resource, policy> as key-value record in it. The row key for such a record is the resource_expression of the resource. One advantage of storing the policies in the key-value store itself is that it is easy to maintain multiple versions of a policy. Systems such as Cassandra can maintain multiple versions for a column's value based on their time stamps. Maintaining previous versions of a policy as it changes over time can be useful for auditing purposes.

4.2.3 Policy Enforcement Point

The PEP evaluates the policy and grants access to the resource if it evaluates to true. For policy evaluation, the PEP may query the key-value store. In K-VAC library code, we have defined a class called *Evaluator* which implements the logic of evaluating the *constraint_expressions*. In the *Evaluator* we support the following binary operators: *in, equal, and, or, minus*. The operands for operators *in, equal, and, or* can be, values that are either directly specified, or those that are obtained by evaluating a *resource_expression*, or parameterized variables, or special variables: *user.id*, *thisKey* and *current_time*. The operands for the *minus* operator can be integers or *current_time*. The variable *user.id* refers to the identifier of a user, and at runtime, maps to the id of the user who is requesting access to a resource. The variable *thisKey* maps at runtime to the row key for the resource (row or column) to which access is being requested. For evaluating the *resource_expression* specified as part of the *constraint_expression* the *Evaluator* interfaces with the underlying key-value store.

5 Evaluation

In this section we evaluate the K-VAC model and the library-based implementation of the K-VAC system. We evaluate the model by demonstrating it for specifying access control policies for another application, a web-based social

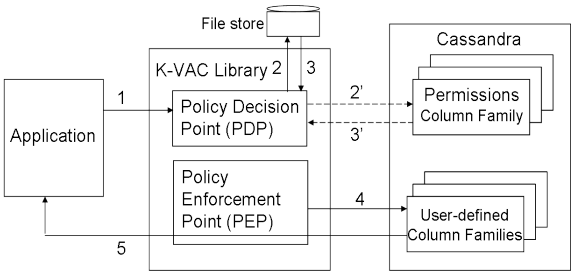

Figure 12: K-VAC System Architecture

sharing application. We have implemented data model and K-VAC policies for both patient information system and social sharing application [7]. Towards evaluating the library-based implementation of the K-VAC system, we discuss how we evolved the K-VAC library to interface with HBase and MongoDB.

5.1 K-VAC Model Evaluation

We consider a web-based social application (*social share* (SS)), which supports capabilities for users to share messages, pictures, and plans with their family and friends. We consider the following requirements for this application.

[R1] A family member can read a person's all data.

[R2] A friend can read a person's plans.

[R3] A friend can read a person's messages only if they are not more than seven days old.

Using Cassandra, the social share application can be modeled as follows. We define the keyspace *SS* for this application, and define two column families (c.f.) corresponding to persons and messages for it.

[Person c.f.:] id, friends, family, plans, message_ids, private_messages

[Message c.f.:] id, message, message_time_stamp

The *id* attribute is the row key in each of the column families. The *message_ids* column in the *Person* column family can contain multiple values.

5.1.1 Requirement R1

In Figure 13 we present the policy corresponding to the first requirement. Suppose that the *Person* column family has following data records.
record 1 : <id=John, friends=Jack, family=Pranav,
plan=Visit Austin>
record 2 : <id=Pranav, friends=John, family=Shyam>.
record 3 : <id=Jack, friends=John, family=Shyam>.
Suppose that the user Pranav is trying to access the data for the user John. The access control policy checks that the id of the user who is accessing the resource (*Pranav*) is present in the *family* column of the user whose data is being accessed (John). In our framework *thisKey* is a special variable which maps to the key of

the row that is being accessed. In the above example this is indeed the case and hence user Pranav will be granted access to John's data. On the other hand, if the user Shyam requests access to John's data, he won't be granted that access.

read row /SS/Person/id
condition
/SS/Person(id=user.id)/id *in*
/SS/Person(id=thisKey)/family

Figure 13: K-VAC expression for R1

5.1.2 Requirement R2

The policy for requirement R2 is shown in Figure 14. The resource to be accessed is the column *plans* of the *Person* column family. As part of the condition evaluation we check whether the id of the user who is requesting access to the *plans* column of a particular user, is present in the *friends* column of the target user. For example, using the data from requirement 1 above, user Jack will be able to access user John's plans.

read column /SS/Person/plans
condition
/SS/Person(id=user.id)/id *in*
/SS/Person(id=thisKey)/friend

Figure 14: K-VAC expression for R2

5.1.3 Requirement R3

In Figure 15 we present the policy for requirement R3. Suppose that the *Person* column family and *Message* column family contain the following data records, respectively.
record 1 : <id=John, friends=Jack, family=Pranav,
message_ids=m1,m2,m3>
record 2 : <id=m1, message=New message,
message_time_stamp=April 13>.
record 3 : <id=m2, message=Old message,
message_time_stamp=April 6>.
record 4 : <id=m3, message=Really old message,
message_time_stamp=April 1>.

Suppose that the user Jack is trying to access message with id m1 created by John. The first part of the condition (lines 3-4) checks whether Jack is a friend of John. The second part of the condition (lines 5-7) checks whether the time stamp for message with id m1 is within seven days of the current time. In line 4,

the row key for the Message column family needs to be that particular message's id whose access is being requested by the user. In this case it is message with id m1. We use a *parameterized variable* for specifying this id.

1.*read column* /SS/Person/message_ids(value=m1)
2.**condition**
3. /SS/Person(key=user.id)/id *in*
4. /SS/Person(key=thisKey)/friend *and*
5. /SS/Message(key=/SS/Person(key=thisKey)/
6. message_Ids(value=$p)/message_time_stamp *in*
7. current_time *minus* Days(7)

Figure 15: K-VAC expression for R3

5.2 K-VAC Library Evaluation

For evaluating the library based design approach of K-VAC, we developed K-VAC's bindings for two other key-value stores namely, HBase [5] and MongoDB [8]. Our decision of which other key-value stores to use was guided by two considerations. First, we wanted key-value stores whose data model is different as compared to that of Cassandra. This was crucial towards evaluating how easily K-VAC library can be integrated with different kinds of key-value stores. Second consideration was that we wanted key-value stores that are implemented in Java or which provide Java drivers. This requirement stemmed from a practical stand-point, as we have implemented the K-VAC library in Java.

HBase and MongoDB satisfied these requirements. In Figure 16 we show the logical data model for HBase, and in Figure 17 the logical data model for MongoDB. HBase's data model is similar to Cassandra's data model, the difference being there is no concept of *SuperColumn* in HBase. MongoDB is a *document-oriented* key-value store. There is no notion of a keyspace or a column family, but, there is a concept of a *collection* and a *document*, which are analogous. This satisfied our first requirement. HBase is implemented in Java, whereas MongoDB is implemented in C++ and provides Java drivers.

In order to integrate K-VAC library with HBase and MongoDB we had to make changes to the policy enforcement module. Specifically, the change corresponded to a single call corresponding to getting a key's attribute as part of policy evaluation. For example, consider the evaluation of the policy specified in Figure 15. The protocol/api for finding the values for the attribute *friend* for a person, or the attribute *message_time_stamp* for a message, is different for Cassandra, HBase, and MongoDB. Towards this we made the following changes to the K-VAC library. We made the *Evaluator* class as an abstract class and added an empty method *getAttributeValue* in it. We defined three new classes, *CassandraEvaluator*, *HBaseEvaluator*, and *MongoDBEvaluator* to inherit from

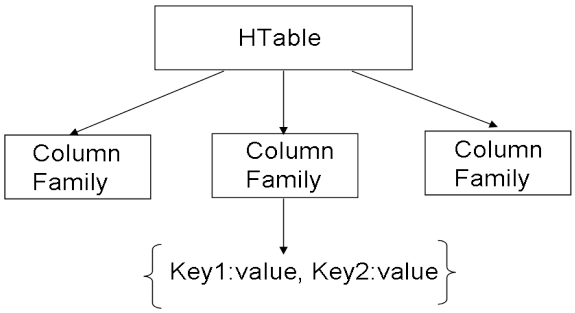

Figure 16: HBase data model

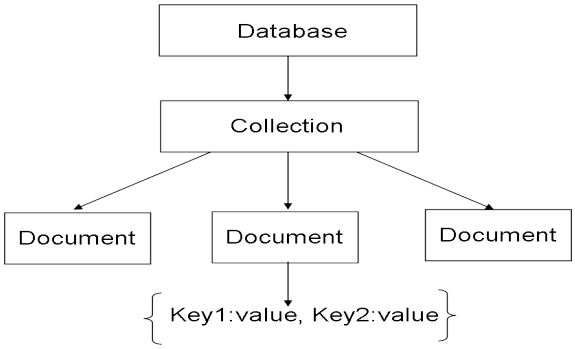

Figure 17: MongoDB data model

the *Evaluator* class. We overrode the *getAttributeValue* method in each of them to obtain an attribute's value using the appropriate protocol.

6 Discussion

For specifying a resources within a resource_expression, it is possible to use other formalisms, such as resource query languages provided by key-value stores themselves (e.g. the Cassandra Query Language (CQL)). We chose to define new formalism to identify resources for two reasons. First, we wanted a uniform mechanism to specify a resource independent of the key-value store being used. Second, for supporting evaluation of *constraint_expressions* we need to integrate resource specifications within the constraint expressions. For this, we wanted a simple model of identifying resources.

The *recursive get* problem mentioned in Section 4.1.1 happens because of the lack of *joins* in key-value stores. In traditional databases, the *join* operator provides the mechanism to query for resources in different tables, but key-value stores do not support this operator. This issue does not arise when K-VAC is implemented outside of a key-value store as a library.

Currently, the K-VAC policy specification supports four levels of hierarchy, corresponding to keyspace, column family, rows, and columns. Referring to Figure 2, we observe that if additional levels of resource hierarchies need to be supported, we would need to modify the specification model and the K-VAC library implementation to support them.

7 Related Work

The K-VAC model is an improvement over the current access control models for Cassandra, HBase, and MongoDB. Cassandra currently supports access control at the level of a keyspace and a column family, HBase does not have any form of access control, whereas MongoDB's access control is limited to controlling access to documents and collections based on whether a user is an admin or not. In contrast, in K-VAC access control policies can be specified at the level of a row or a column, along with keyspace and a column family. Moreover, we support policies that can be based on the *contents* of a row or a column. In comparison, currently Cassandra's access control policies are based on the *names* of the keyspace or the column family.

The content-based access control capabilities of K-VAC are similar to *attribute-based* access control models, such as
XACML [13] and UCON [14]. In contrast to these general purpose access control models, K-VAC is specifically designed to address the access control issues of key-value stores. Correspondingly, the resource specification and constraint specification closely follows the logical data model of key-value systems. The mechanism of *parameterized variables* in K-VAC for specifying a column's value at runtime is similar to the notion of *parameterized privileges* of [3]. The ideas on content-based access control can be traced back to the work on access control based on data types in programming languages [11].

Context-based access control has been studied for pervasive computing applications [12]. We support such requirements in K-VAC. In contrast to pervasive computing systems, we assume existence of a trusted context management system which reliably provides context information, about people's location and time.

The problem of secure access to data hosted in the cloud is addressed in [17]. The authors present a solution based on encrypting each data block with a different key. One drawback of this approach is that the system may not scale as the number of data items increase. In contrast, our focus in this paper is on access control issues of key-value stores.

In [16] authors provide a solution, based on trusted computing platform, for the problem of ensuring confidentiality of client data from the cloud service provider who is hosting that data. In contrast our focus is on providing and efficiently managing access control to various resources that are stored in a key-value store. For auditing purpose the access control model defines the notion of permission versions using which it is possible for the access control system to track the changes in the set of permissions.

In [2] authors present capability based access control for peer-to-peer storage overlays, and present protocols to identify malicious nodes in such a system. Rather than system-level issues, our focus is more on the application-level access control issues, where an application uses a key-value store for its data storage requirements.

For traditional data management systems, such as RDBMSes and Workflow systems, rich access control models, such as Role-based access control (RBAC) [15] have been developed over the years. In contrast, the current access control models of key-value systems are limited in their capability. Through K-VAC we provide fine-grained access control capabilities to such systems. In K-VAC, currently all the access decisions are made based on a user's identity. Role-based access control capabilities can be built into K-VAC by adding an attribute of role to a user, and extending the set of permissions to include application-specific permissions.

8 Conclusion

In this paper we have presented K-VAC – an access control model for modern NoSQL key-value stores. In this model, access control policies can be specified for resources at different levels of hierarchy, such as groups of rows, specific rows, or specific columns. The policies can be based on the *content* of rows and columns, and they may also include context information, such as user location and time of the day. We have designed and implemented the model as an application-level Java library. We have implemented the bindings of this library for Cassandra, HBase, and MongoDB. Through two case-study applications we have presented the capabilities of this model. The K-VAC library is available for download at [7].

References

[1] M. Evered and S. Bögeholz. A Case Study in Access Control Requirements for a Health Information System. In *ACSW Frontiers '04: Proceedings of the Second Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 53–61, 2004.

[2] A. Gehani and S. Chandra. Parameterized access control: from design to prototype. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, SecureComm '08, pages 35:1–35:8, New York, NY, USA, 2008. ACM.

[3] L. Giuri and P. Iglio. Role Templates for Content-based Access Control. In *RBAC '97: Proceedings of the Second ACM Workshop on Role Based Access Control*, pages 153–159, 1997.

[4] http://cassandra.apache.org/.

[5] http://hbase.apache.org/.

[6] https://github.com/devdattakulkarni/Cassandra KVAC.

[7] https://github.com/devdattakulkarni/KVAC.

[8] http://www.mongodb.org/.

[9] http://www.w3.org/TR/xpath/.

[10] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] A. K. Jones and B. H. Liskov. A Language Extension for Expressing Constraints on Data Access. *Commun. ACM*, 21(5):358–367, 1978.

[12] D. Kulkarni and A. Tripathi. Context-Aware Role-based Access Control in Pervasive Computing Systems. *SACMAT'08 Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, pages 113–122, 2008.

[13] T. Moses. OASIS eXtensible Access Control Markup Language (XACML) Version 2.0, OASIS Standard. pages 1–141, 1 February 2005.

[14] J. Park and R. Sandhu. The UCONABC Usage Control Model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174, 2004.

[15] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-based Access Control: Towards a Unified Standard. In *RBAC '00: Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63. ACM Press, 2000.

[16] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[17] W. Wang, Z. Li, R. Owens, and B. Bhargava. Secure and efficient access to outsourced data. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 55–66, New York, NY, USA, 2009. ACM.