A Provenance Model for Key-value Systems

Devdatta Kulkarni*
devdattakulkarni@gmail.com

Abstract

In this paper we present key-value provenance model (KVPM). In KVPM, provenance information can be collected for both, the data and the data's schema in a key-value system. Collection of the information is application-driven, and it can collected at different levels of the data model hierarchy. Here we present the capabilities of the KVPM system along with its design and implementation for Cassandra and initial evaluation.

1 Introduction

Many modern Internet-scale applications are using non-relational key-value stores as their back end data systems. Several such systems have been built recently such as, Cassandra, MongoDB, HBase. Such systems provide several advantages over traditional databases, such as ease of evolution due to schema-less data model, scalability, relaxation of data consistency requirements, and so on. Given such advantages, it would not be incorrect to assume that such systems would be used in more large scale distributed applications in the future.

Data provenance has emerged as one of the key requirements in building modern systems that support properties such as auditability, data lineage tracking, providing confidence in experimental results, and so on [7]. Provenance issues for file systems, databases, workflow systems have been an active area of research [7, 2, 6]. The focus of this paper is on the issues related to building provenance capabilities for modern key-value systems.

Such systems present unique set of challenges towards collecting and tracking of provenance information. Key-value systems store data in the form of attribute-value pairs corresponding to a key. The different keys in the system may have different attributes. Applications may add or delete attributes for any key at any time. Such a *schemaless* design makes the provenance of the data closely tied with the provenance of the data schema. Specifically, in order to track how the values of an attribute have changed over time (data provenance), it is important to know when that attribute was added for that key (schema provenance). In relational database systems, changes to the database schema are relatively infrequent compared to changes to the data itself. Therefore, it is possible, and may be preferable even, to separate provenance management for the database schema from the provenance management for the data itself [2]. However, for key-value systems this may not work for the reason identified above.

Second, most of the key-value systems support multiple versions for an attribute's value. For provenance purposes it might not be sufficient to track the versions of an attribute, but we may also want to track various versions of *different* attributes' values together. For instance, consider a key-value system used to store patient information in which the date of a patient's visit and the prescribed medication are stored as attributes corresponding to a patient's key in the key-value system. The different values for these attributes will correspond to their different versions. As part of provenance, it might be important to track which medication was prescribed on which date.

Third, most of the key-value systems have a hierarchical data model. For instance in Cassandra [3], data is grouped into the levels corresponding to column family, rows, and columns. The column family is the highest level of the data hierarchy whereas a column is the lowest level of the hierarchy. In certain situations we may need to know only the provenance information at the level of a column family, whereas in certain other situations we may need fine-grained information at the level of a row or a column. For instance, in the above mentioned patient information system, we may have a column family for storing all patients' information. In certain cases we may just want to know how the number of keys present in the column family have changed over time. This infor-

*Author affiliation: Rackspace. This work has been done independently and is not related to author's work at Rackspace.

mation can be obtained from the provenance information at the level of the column family. In other situations we may want to know what medication was prescribed to a patient over time. This information can be obtained from the provenance information corresponding to the column that stores the medication information for a particular patient. Thus, the provenance model needs to be flexible enough to support collection and querying of provenance information at different levels of data model hierarchy.

Fourth, it might be essential to give control to applications over the data for which the provenance information should be collected. This is because a typical key-value system is usually targeted to store large amounts of data. If the provenance system tracks provenance for all this data then the provenance data management may become a problem in itself. Therefore, it is essential to give applications control over the data for which they want to collect the provenance information.

2 Provenance Requirements

Consider a patient information system built using a key-value data store. Such a system stores information about patients, doctors, and nurses. The information associated with patients may include information about his or her date of visit, medication, the patient's allergies. The information associated with doctors and nurses may include their current patients and work hours.

We consider the following provenance requirements for such a system. As part of a patient's care, it is possible that his or her medications will change over time. The system should be able to query all the values of the medication prescribed to a patient. For audit purposes the system should be able to identify who has accessed a patient's records. It should be possible to track medication and its reaction, if any, as a logical unit. It is possible that for a patient, the information about a patient's allergies is added sometime after the initial patient information record is created. The provenance system should support querying information about when each attribute was added to the patient data.

We now describe how such an application can be built using Cassandra, which we use as a representative key-value system in this paper. In Cassandra, data is stored in a *Keyspace*. A keyspace may contain one or more *Column Families*. Within a column family, data items are stored in the form of attribute-value pairs for a key. Each data item is made up of a unique element (the row key) and number of attributes (columns). We model the patient information system by defining a keyspace named PI and three column families - *Patient*, *Doctor*, and *Nurse*. Here we show schema definition of the patient column family. *[Patient:] id, visit_date, medication, allergies*. The *id* attribute is the row key.

3 Key-value Provenance Model (KVPM)

In KVPM [4] collection of provenance information is done using *provenance policies* and querying of the collected provenance information is done using *provenance queries*. Both these are built using *resource_expressions* and a set of predefined operators.

A resource expression identifies a specific resource within the hierarchical data model of a key-value system. In Cassandra a resource can be a column family, a specific row, or a specific column of a specific row. A resource expression for Cassandra is specified as follows: */keyspace/column_family(id=value)/column*. The resource expression format is inspired from the XPath query language for XML documents. The specification *(id=value)* acts as a *filter predicate* on the column family to select only the row with the row key equal to the specified value. The filter predicate value can be specified as a literal, or to be obtained by querying a column, or as a *parameterized variable* that is bound at runtime.

3.1 Provenance policies

Provenance policies are used by applications to indicate the resources for which provenance information should be collected. A provenance policy is expressed as *<provenance_operator resource_expression>*. We define two categories of provenance operators, data provenance operators and schema provenance operators. In Cassandra data related changes can happen in one of the following ways: a) insertion of a row key in a column family, b) modification of the value of an already existing column for a row key, c) addition of a new *<column, value>* tuple to a row key, or d) addition of a new column family to a keyspace. We treat the first two as part of the data provenance and the latter two as part of the schema provenance.

The data provenance operators are *enable_dataprov_access* and *enable_dataprov_write(\$mid)*. The first operator can be applied to both a column family and a column, whereas the second operator can be applied only to a column. The operator *enable_dataprov_access* enables provenance collection for all the accesses (read or write) to a resource. The provenance information collected consists of the *access timestamp*, the identity of the accessor, and the type of the access. For write accesses to a column its value before modification is also stored.

The operator *enable_dataprov_write(\$mid)* takes in a parameterized variable. An application can pass a logical *marker* identifier as the value of this variable. All the write operations on the specified column are then tagged with the specified marker identifier (mid). This is useful in cases where an application may want to track updates

to a set of columns as one logical unit. For instance, with the following policies all the writes to the visit_date and medication columns for the patient John are tracked with the marker identifier m1.

enable_dataprov_write(m1) /PI/Patient(id=John)/visit_date
enable_dataprov_write(m1) /PI/Patient(id=John)/medication

The need for high-level abstractions for collecting provenance information has been identified in [6]. The *enable_dataprov_write($mid)* operator serves this purpose by providing a way to treat different columns as part of a high-level application action.

The schema provenance operator is *enable_schemaprov_modify*[1]. This operator can only be applied to the keyspace or to a row key. For a keyspace, a schema modification corresponds to addition or deletion of a column family. For a row key, a schema modification corresponds to addition or deletion of a column for that row. The provenance information collected includes the *access timestamp*, the identity of the accessor, current schema elements when schema provenance collection is enabled and addition or deletion indicators for subsequent changes to the schema.

3.2 Provenance queries

Provenance queries are used by applications to query provenance information. A provenance query is expressed as <*query_operator resource_expression*>.

The following provenance query operators are supported: *ALL*, *LAST*, *AT($val)*, *FOR($mid)*. The first three operators can be used with a keyspace, a column family, a row key, or a column. The *FOR($mid)* operator can be used only with a column. The operator *ALL* gives all the provenance information for a resource. The operator *LAST* gives the most recent provenance information. The operator *AT($val)* specifies a parameterized variable whose value can be a *version number*. In KVPM we store *access timestamp* as part of the provenance information (Section 3.1). Such a timestamp value can be used as this version number. This operator returns the provenance information at the specified version for the resource, if present. The operator *FOR($mid)* specifies a parameterized variable whose value can be a marker identifier. It returns the column value corresponding to that marker's identifier, if present.

4 Design and Implementation

The design and implementation of the KVPM system has been guided by the questions: *when* to collect the provenance information? *where* to store it? *how* to collect it?

With regards to *when* to collect the information there are two options. The information can be collected *before* a data access action is performed, or it can be collected *after* an action has been performed. One issue with collecting the information before an action's execution is what to do when the action is unsuccessful. One option could be to roll back the provenance information. Another option[2] is to use a two phase protocol in which, before an action's execution its intent is recorded, and after its execution, its result is recorded. This option circumvents the issue of rollback, however, it leads to double the amount of provenance information collected as compared to when it is collected only after an action's execution. Another minor advantage of collecting the information afterwards is that it is possible to use specific mechanisms, such as *triggers*, if available in the key-value system, to collect it.

Another issue is how to handle the data that is garbage collected by the key-value system. For instance a Cassandra column can have a time-to-live (TTL) field associated with it. The column gets deleted after the expiry of the TTL. Whether to monitor such a deletion event as part of the provenance information is a question.

For storing the provenance information there are several options, such as storing it in the key-value system itself, or storing it in a separate database, or storing it in log files. The advantage of using the key-value system itself is that all the data management capabilities of the system, such as fault-tolerance, replication, etc. become available for managing the provenance information. Using log files might be easy, but for certain key-value systems such as Cassandra, such files would be distributed on different nodes in the system, possibly complicating provenance information retrieval and inference.

For collecting the provenance information there are two options. One is to collect it by modifying the key-value system itself. The advantage of this option is that an application does not need to know about the existence of the provenance layer. The other option is to build the provenance management layer as a library which is interposed between an application and a key-value system. The advantages of a library-based design are, a) the library can be integrated with different key-value stores, and b) the library can be interfaced with different storage back ends for storing the provenance information. A disadvantage is that an application does not interface with the key-value system but has to go through the library.

We have implemented the KVPM system in three different configurations, as follows [4, 5]: a) within Cassandra, with provenance information stored in Cassandra itself (KVPMC), b) as a library, with provenance information stored in Cassandra (libKVPMC), and c) as a

[1] Currently under development.

[2] Approach suggested by one of the anonymous reviewers.

library, with provenance information stored in MySQL database (libKVPMD). We collect the information *after* a data access action has been performed. Since Cassandra does not support triggers, we collect this information in the *path* of the data access actions. We also do not track deletion of a column upon TTL expiry as part of provenance information.

4.1 Provenance information storage

For storing provenance information in Cassandra we defined the *Provenance* column family. A row key in this column family is the *resource_expression* from the corresponding provenance policy. For the *enable_dataprov_write($mid)* operator, the value of mid is appended to the resource expression, and then this composite value is used as the row key. The column names for a row key are the *access timestamps*, and their values are concatenation of the accessor, the type of access (get or put), and for write accesses to a column, its value before change. For storing provenance in MySQL we defined a table with the following columns: resource, access_timestamp, accessor, operation, value_before_change.

4.2 Query operators

For the provenance information stored in Cassandra, the query operators are implemented to query the *Provenance* column family. For the operators *ALL*, *LAST*, and *AT($val)*, we use the Cassandra's *get_slice* interface to obtain the required columns for the row key corresponding to the *resource_expression* specified in the provenance query. For the operator *AT($val)*, we return only that column's value whose name matches the specified version number. For the operator *FOR($mid)*, we append the mid value to the resource expression and then return the *LAST* value for that row. For MySQL, the query operators are implemented as appropriate SQL queries.

5 Evaluation

We have done preliminary evaluation of the KVPM system towards finding out how much overhead the provenance layer incurs on basic Cassandra operations (put and get). To evaluate this we ran a single node Cassandra cluster on a Intel Core 2 Duo CPU T64002.00GHz Linux Laptop with 4.00 GB RAM and 288 GB disk. We used Cassandra version 0.8.6 in our experiments. We used the patient information system for this evaluation. We created the *Patient* column family and added a single row key (*John*) with a single column (*medication*). We changed the value of this column five times corresponding to five runs of the experiment. There was no other data in the system. To avoid caching of rows and columns from affecting the results we set the values of the *keys_cached* and *rows_cached* properties for the *Patient* column family to zero.

	No Prov	KVPMC	libKVPMC	libKVPMD
get	26	148	185	1101
put	125	198	235	1249

Table 1: Provenance performance overhead

In Table 1 we present the median values in milliseconds across five runs for writing and reading data from Cassandra with and without provenance. We see that the configuration in which provenance information is collected within Cassandra itself (KVPMC) has the best performance.

To better understand the performance characteristics of the three design options, we plan to evaluate the KVPM system with different workloads, such as those described in [1], as part of our ongoing work.

6 Conclusion

In this paper we have presented a provenance model for key-value systems (KVPM). This model supports collection and querying of provenance information at different levels of data model hierarchy in a key-value system. The KVPM system is implemented as a library which can be interfaced with different key-value stores. Here we presented initial evaluations of its performance overhead.

References

[1] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[2] GAO, S., AND ZANIOLO, C. Provenance management in databases under schema evolution. In *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance* (Berkeley, CA, USA, 2012), TaPP'12, USENIX Association, pp. 11–11.

[3] HTTP://CASSANDRA.APACHE.ORG/.

[4] HTTPS://GITHUB.COM/DEVDATTAKULKARNI/CASSANDRA KVPM.

[5] HTTPS://GITHUB.COM/DEVDATTAKULKARNI/KVPM.

[6] MARINHO, A., MATTOSO, M., AND WERNER, C. Challenges in managing implicit and abstract provenance data: experiences with provmanager. In *Proceedings of the 3rd USENIX conference on Theory and Practice of Provenance* (Berkeley, CA, USA, 2012), TaPP'11, USENIX Association.

[7] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATEC '06, USENIX Association, pp. 4–4.