Materialized Views in Cassandra

Tilmann Rabl [#†], Hans-Arno Jacobsen [#]
[#] *Middleware Systems Research Group, University of Toronto*
[†] *IBM Canada Software Laboratory, CAS Research*

Abstract

Many web companies deal with enormous data sizes and request rates beyond the capabilities of traditional database systems. This has led to the development of modern Big Data Platforms (BDPs). BDPs handle large amounts of data and activity through massively distributed infrastructures. To achieve performance and availability at Internet scale, BDPs restrict querying capability, and provide weaker consistency guarantees than traditional ACID transactions. The reduced functionality as found in key-value stores is sufficient for many web applications.

An important requirement of many big data systems is an online view of the current status of the data and activity. Typical big data systems such as key-value stores only allow a key-based access. In order to enable more complex querying mechanisms, while satisfying necessary latencies materialized views are employed. The efficiency of the maintenance of these views is a key factor of the usability of the system. Expensive operations such as full table scans are impractical for small, frequent modifications on Internet-scale data sets. In this paper, we present an efficient implementation of materialized views in key-value stores that enables complex query processing and is tailored for efficient maintenance.

1 Introduction

In recent years, many companies have realized the value of user activity data. As a result, increasing amounts of data have to be processed and stored. Because of the scalability issues of ACID compliant database systems, more and more simplified data management systems are being developed. Many of these systems are following the principal design choices of Google's BigTable [2] and Amazon's Dynamo [5]. Core features of these systems are a high degree of distribution, key-value access, and relaxed consistency requirements. Due to the form of data management and data access, typically driven by a global key, these systems are called key-value stores.

Use cases like application performance management [8] or smart traffic management require the high performance data insertion and data access as they are provided by key-value stores. However, they also require an on-line view on certain conditions of the managed system, that goes beyond simple key-value lookups. In this paper, we present our research on materialized views in key-value stores. In contrast to previous work in this area (e.g., [6, 10]), our views can be organized in hierarchies and thus materialize complex SQL queries. The rest of the paper is organized as follows. In the next Section, we give an overview of key-value stores and the general data and query model that is basis of this work. In Section 3, we describe the different types of views and view maintenance within our model.Section 4 describes our implementation in Cassandra. We conclude with future work in Section 5.

2 Key-Value Stores

In this section, we review the characteristics of common key-value stores. We use the following notation in the rest of this paper. A base table B contains a number of records that can be addressed by an arbitrary key. The key must be unique within the table. Therefore, a record can be represented by

Copyright © 2014 IBM Corp. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

a tuple (k, v), where k is the key and v is the value. We do not place any restrictions on the value. In an analogy to relational database systems, we say B has the *schema* $B(\underline{K}, V)$. In our model, we also consider records that have more than one value, e.g. consider the table $B'(\underline{K}, V_1, V_2)$ which contains up to 2 values per key. Furthermore, we assume that a key can consist of multiple attributes, e.g. table $B''(\underline{K_1}, \underline{K_2}, V)$. Although many current key-value stores actively support these types of schemas they can also be mapped to a pure key-value store. We will consider the so called *extensible record stores* in the rest of this paper [1], which allow multiple values per key. These systems store a schema with each value. As mentioned above, the same functionality can be achieved in pure key-value stores by encoding the information in the key and the value. Examples for extensible record stores are BigTable [2], and Cassandra [7], examples for pure key-value stores are Dynamo [5], and LevelDB [4]. Most systems provide the following basic API, although, as the systems mature more and more complex operations are added.

get(key) Retrieve a value or row from the store. This is the primary way of accessing data. Typically, the function is efficiently implemented in key-value stores.

insert(key,values) Insert a value or row with key in the store. This function often does not consider duplicates. Therefore, if previous values were written for a certain key, the value will be overwritten. In case of extensible record stores, an insert with additional attributes will extend the previously stored value. Consider for example a row (401E,401,E) with schema (Key, Number, Direction), if a row (401E,EAST,Ontario 401 Express) with schema (Key, Direction, Name) is inserted only the value of attribute Direction is overwritten. The resulting row will be (401E,401,EAST,Ontario 401 Express).

delete(key) Delete a value or row and the according key from the store. Typically, rows are not deleted in a key-value store right away but marked as deleted. The actual delete is then asynchronously performed by some sort of garbage collection process. Extensible record stores also allow deleting only specified columns of a row.

Sensor Reading

Key	Sensor	Date	Time	Speed	Occupancy
401DE0070DEC-2013-11-27- 14:46:00.0	401DE0070DEC	2013-11-27	14:46:00.0	81	4
401DE0020DET-2013-11-27- 14:46:00.0	401DE0020DET	2013-11-27	14:46:00.0	93	1
401DE0060DWC-2013-11-27- 14:46:00.0	401DE0060DWC	2013-11-27	14:46:00.0	102	3

Road Sensor

Key	Address	Road
401DE0070DEC	401-EC - E OF ALLEN ROAD	401E
401DE0020DET	401-ET - E OF ALLEN RD	401E
401DE0060DWC	401-WC - E OF ALLEN	401W

Road

Key	Number	Direction	Name
401E	401	EAST	Ontario 401 Express
401W	401	WEST	Ontario 401 Express
407E	407	EAST	Express Toll Route

Figure 1: Example Schema

update(key,values) Change the value/row that is stored under a given key. As mentioned above, this function is often not different from the insert command. Thus, not all key-value stores implement the command separately. However, it is important to consider the different semantics of new insert and update in materialized views.

The majority of key-value stores are designed for large data sets and are, therefore, inherently distributed. However, there are also single node key-value stores, such as LevelDB [4]. In the following, we will concentrate on distributed stores that persist data on disk or SSD.

Two architectures of distributed key-value stores are prevalent, BigTable-like architectures, which distribute data in horizontal partitioned shards, and Dynamo-like systems, that use the principle of distributed hash tables (DHT) as distribution strategy. Systems in the former group are BigTable and HBase, in the later group are Cassandra, and Project Voldemort. An interesting in-between architecture is Riak, which distributes data in ranges, which then are distributed in DHT fashion.

An important characteristic of key-value stores is their alternative way of dealing with consistency. In traditional databases, all data is typically consistent, while in key-value stores, the state of a replicated table might be outdated for a certain period of time. This eventual consistency is a challenge for view maintenance.

3 Materialized Views in Key-Value Stores

There are multiple different types of materialized views. These can be classified by the basic query operators that they materialize. To visualize the different classes of views, we will use a simplified traffic monitoring example shown in Figure 1. We

will first discuss views that only a single query operator.

Projection View A projection view materializes the projection π in relational algebra. It typically removes attributes in the query result. Depending on the implementation, it does a duplicate reduction as the relational algebra operator does or not as common in SQL implementations.

Index View An index view is not based on a relational operator but is a special case of a projection. In the index view, another attribute or set of attributes is used as key. Thus, the index view allows fast access using non-key attributes.

Selection View The selection operator σ filters rows based on selection criteria (predicates). Only rows for that the selection criteria hold are part of the selection.

Aggregation View An aggregation view materializes a grouping and aggregate of an attribute such as the sum, average, minimum, maximum, or median.

Join View The join view materializes a relational join between tables. Our implementation is generic and can emulate different types of joins, e.g., equi-joins, outer-joins, and semi-joins.

Hierarchical View Often it is more desirable to have multiple views dependent on each other rather than having a single view implementing a specific query. This way, multiple different complex queries can be based on the same underlying basic materialized views. This is highly related to multi-query optimization [9] and can lead to substantial performance improvements due to the de-duplication of query operations. An example of a hierarchy of views can be seen in Figure 2.

In our prototype, we have implemented the index view and the join view and the possibility of creating hierarchical views. In our implementation, all views are read-only. Our prototype is built by extending Cassandra, however, the same extensions can easily be applied to other systems like HBase.

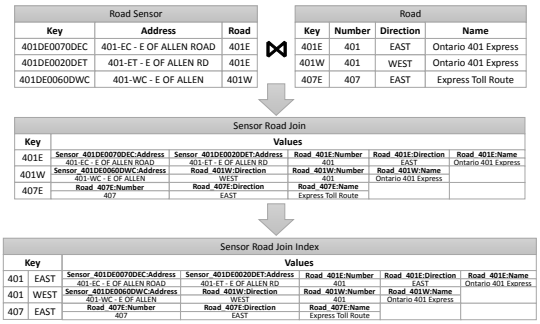

Figure 2: Example of a Hierarchy of a Join and Index View

⟨*statement*⟩ ::= `CREATE VIEW` view_name `OF` ⟨*table_defs*⟩;

⟨*table_defs*⟩ ::= ⟨*table_def*⟩
| ⟨*table_def*⟩ `AND` ⟨*table_defs*⟩

⟨*table_def*⟩ ::= table_name [⟨*key_list*⟩]

⟨*key_list*⟩ ::= ⟨*column_name*⟩
| ⟨*column_name*⟩, ⟨*key_list*⟩

⟨*column_name*⟩ ::= column_name
| `KEY`

Figure 3: Grammar for the Creation of Materialized Views in CQL

4 Implementation in Cassandra

The presented technique for managing materialized views in key-value stores was fully implemented in Cassandra, Version 2.0. Below, we present details of the changes in the architecture as well as in the query language. Cassandra features a SQL-like query language, the Cassandra Query Language (CQL). We extended CQL to enable the creation of materialized views. The grammar for creating a view is shown in Figure 3.

The extension integrates seamlessly with CQL's data modeling language [3]. An example for the creation of an index view can be seen in Listing 1. The listing creates a view that indexes sensors by the roads that they are installed on. The view will be updated automatically whenever the base table is updated.

```
CREATE VIEW Sensor_Road_Index OF
    Sensor[Road];
```

Listing 1: Creation of an Index View

The syntax for creating join views can be seen in Listing 2. The command creates a column fam-

ily (named Sensor_Road_Join), which stores all combinations of Sensors and Roads with matching Road. It is an instance of a key-value join, which is indicated by the keyword *key* on the left side of the join. Joins can be key-key joins, key-value joins, and value-value joins.

```
CREATE VIEW Sensor_Road_Join OF
    Road[key] AND Sensor[Road];
```

Listing 2: Creation of a Join View

It is also possible to create views with multiple columns as index or views materializing joins that join multiple columns an example can be found in Listing 3. The view shown will group roads with same number and direction. In the case of joins the number of joined columns must be equal on both sides and the order determines the join partners.

```
CREATE VIEW Road_Index OF
    Road[Number,Direction];
```

Listing 3: Creation of an Index View with Multiple Indexed Colums

Finally, it is possible to join more than two tables and it is also possible to do self-joins. Furthermore, our implementation supports hierarchical views, i.e., views that have views as input tables. An example can be seen in Listing 4, it creates two hierarchical views as presented in Figure 2. Tables Road and Sensor are joined on the road key and an index is created on the road number and direction.

```
CREATE VIEW Sensor_Road_Join OF
    Road[key] AND Sensor[Road];
CREATE VIEW Sensor_Road_Join_Index
    OF
    Sensor_Road_Join[Number,Direction];
```

Listing 4: Creation of an Index View on a Join View

All views are *read-only*, therefore, the only way of updating a view is updating the base table. Due to their dependency to the base table, the views are also implemented with *cascade drop*. If a base table is dropped, all depending views are dropped as well.

5 Conclusion

In this paper, we present our research on materializing complex SQL queries in key-value stores. We have shown that using the basic index view and join view and the possibility to create hierarchies of views, complex SQL queries can be materialized. Our technique is fully implemented in a prototype that is based on the Apache Cassandra key-value store.

In future work, we will address the creation of views during run time as well as dynamic materialization of queries based on a cost model.

References

[1] R. Cartell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39(4):12–27, 2010.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[3] DataStax, Inc. Apache Cassandra Documentation. `http://www.datastax.com/documentation/cassandra/2.0/webhelp/index.html`, 2013.

[4] J. Dean and S. Ghemawat. LevelDB - A fast and lightweight key/value database library by Google. `https://code.google.com/p/leveldb/`, 2011.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, pages 205–220, 2007.

[6] C. Jin, R. Liu, and K. Salem. Materialized Views for Eventually Consistent Record Stores. In *ICDEW*, pages 250–257, 2013.

[7] A. Lakshman and P. Malik. Cassandra: a Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[8] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankowskii. Solving Big Data Challenges for Enterprise Application Performance Management. *PVLDB*, 5(12):1724–1735, 2012.

[9] T. K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[10] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding Frenzy: Selectively Materializing Users' Event Feeds. In *SIGMOD*, pages 831–842, 2010.