Introduction to HBase Schema Design

AMANDEEP KHURANA

Amandeep Khurana is a Solutions Architect at Cloudera and works on building solutions using the Hadoop stack. He is also a co-author of *HBase in Action*. Prior to Cloudera, Amandeep worked at Amazon Web Services, where he was part of the Elastic MapReduce team and built the initial versions of their hosted HBase product. amansk@gmail.com

The number of applications that are being developed to work with large amounts of data has been growing rapidly in the recent past. To support this new breed of applications, as well as scaling up old applications, several new data management systems have been developed. Some call this the *big data* revolution. A lot of these new systems that are being developed are open source and community driven, deployed at several large companies. Apache HBase [2] is one such system. It is an open source distributed database, modeled around Google Bigtable [5] and is becoming an increasingly popular database choice for applications that need fast random access to large amounts of data. It is built atop Apache Hadoop [1] and is tightly integrated with it.

HBase is very different from traditional relational databases like MySQL, PostgreSQL, Oracle, etc. in how it's architected and the features that it provides to the applications using it. HBase trades off some of these features for scalability and a flexible schema. This also translates into HBase having a very different data model. Designing HBase tables is a different ballgame as compared to relational database systems. I will introduce you to the basics of HBase table design by explaining the data model and build on that by going into the various concepts at play in designing HBase tables through an example.

Crash Course on HBase Data Model

HBase's data model is very different from what you have likely worked with or know of in relational databases. As described in the original Bigtable paper, it's a sparse, distributed, persistent multidimensional sorted map, which is indexed by a row key, column key, and a timestamp. You'll hear people refer to it as a key-value store, a column-family-oriented database, and sometimes a database storing versioned maps of maps. All these descriptions are correct. This section touches upon these various concepts.

The easiest and most naive way to describe HBase's data model is in the form of tables, consisting of rows and columns. This is likely what you are familiar with in relational databases. But that's where the similarity between RDBMS data models and HBase ends. In fact, even the concepts of rows and columns is slightly different. To begin, I'll define some concepts that I'll later use.

- **Table:** HBase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.
- **Row:** Within a table, data is stored according to its row. Rows are identified uniquely by their *row key*. Row keys do not have a data type and are always treated as a *byte[ ]* (byte array).
- **Column Family:** Data within a row is grouped by column family. Column families also impact the physical arrangement of data stored in HBase. For this reason, they must be defined up front and are not easily modified. Every row in a table has the same column families, although a row need not store data in all its families. Column families are Strings and composed of characters that are safe for use in a file system path.
- **Column Qualifier:** Data within a column family is addressed via its column qualifier, or simply, column. Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like row keys, column qualifiers do not have a data type and are always treated as a *byte[ ]*.
- **Cell:** A combination of row key, column family, and column qualifier uniquely identifies a cell. The data stored in a cell is referred to as that cell's value. Values also do not have a data type and are always treated as a *byte[ ]*.
- **Timestamp:** Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If a timestamp is not specified during a write, the current timestamp is used. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by HBase is configured for each column family. The default number of cell versions is three.

A table in HBase would look like Figure 1.

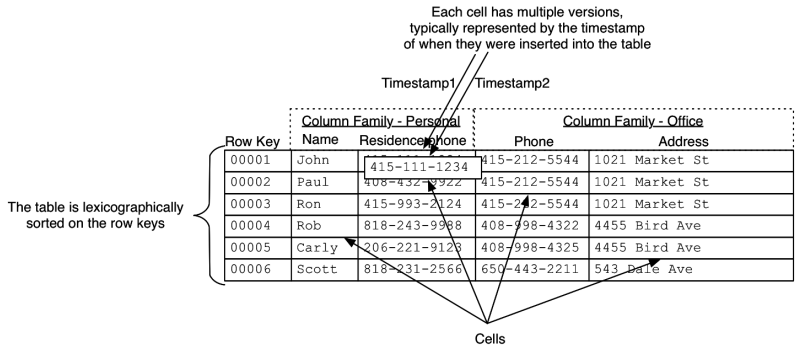


Figure 1: A table in HBase consisting of two column families, Personal and Office, each having two columns. The entity that contains the data is called a cell. The rows are sorted based on the row keys.

These concepts are also exposed via the API [3] to clients. HBase's API for data manipulation consists of three primary methods: *Get*, *Put,* and *Scan*. Gets and Puts are specific to particular rows and need the row key to be provided. Scans are done over a range of rows. The range could be defined by a start and stop row key or could be the entire table if no start and stop row keys are defined.

Sometimes, it's easier to understand the data model as a multidimensional map. The first row from the table in Figure 1 has been represented as a multidimensional map in Figure 2.

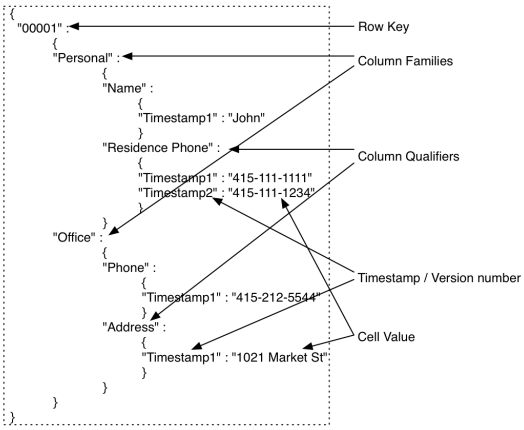


Figure 2: One row in an HBase table represented as a multidimensional map

The row key maps to a list of column families, which map to a list of column qualifiers, which map to a list of timestamps, each of which map to a value, i.e., the cell itself. If you were to retrieve the item that the row key maps to, you'd get data from all the columns back. If you were to retrieve the item that a particular column family maps to, you'd get back all the column qualifiers and the associated maps. If you were to retrieve the item that a particular column qualifier maps to, you'd get all the timestamps and the associated values. HBase optimizes for typical patterns and returns only the latest version by default. You can request multiple versions as a part of your query. Row keys are the equivalent of primary keys in relational database tables. You cannot choose to change which column in an HBase table will be the row key after the table has been set up. In other words, the column *Name* in the *Personal* column family cannot be chosen to become the row key after the data has been put into the table.

As mentioned earlier, there are various ways of describing this data model. You can view the same thing as if it's a key-value store (as shown in Figure 3), where the key is the row key and the value is the rest of the data in a column. Given that the row key is the only way to address a row, that seems befitting. You can also consider HBase to be a key-value store where the key is defined as row key, column family, column qualifier, timestamp, and the value is the actual data stored in the cell. When we go into the details of the underlying storage later, you'll see that if you want to read a particular cell from a given row, you end up reading a chunk of data that contains that cell and possibly other cells as well. This representation is also how the *KeyValue* objects in the HBase API and internals are represented. *Key* is formed by [row key, column family, column qualifier, timestamp] and *Value* is the contents of the cell.

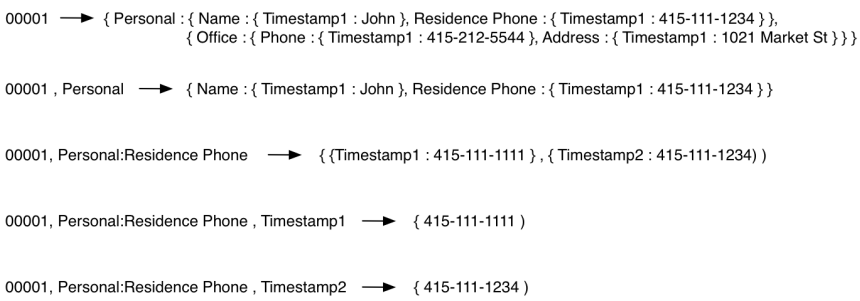


Figure 3: HBase table as a key-value store. The key can be considered to be just the row key or a combination of the row key, column family, qualifier, timestamp, depending on the cells that you are interested in addressing. If all the cells in a row were of interest, the key would be just the row key. If only specific cells are of interest, the appropriate column families and qualifiers will need to be a part of the key

HBase Table Design Fundamentals

As I highlighted in the previous section, the HBase data model is quite different from relational database systems. Designing HBase tables, therefore, involves taking a different approach from what works in relational systems. Designing HBase tables can be defined as answering the following questions in the context of a use case:

1. What should the row key structure be and what should it contain?
2. How many column families should the table have?
3. What data goes into what column family?
4. How many columns are in each column family?
5. What should the column names be? Although column names don't need to be defined on table creation, you need to know them when you write or read data.
6. What information should go into the cells?
7. How many versions should be stored for each cell?

The most important thing to define in HBase tables is the row-key structure. In order to define that effectively, it is important to define the access patterns (read as well as write) up front. To define the schema, several properties about HBase's tables have to be taken into account. A quick re-cap:

1. Indexing is only done based on the *Key*.
2. Tables are stored sorted based on the row key. Each region in the table is responsible for a part of the row key space and is identified by the start and end row key. The region contains a sorted list of rows from the start key to the end key.
3. Everything in HBase tables is stored as a *byte[ ]*. There are no types.
4. Atomicity is guaranteed only at a row level. There is no atomicity guarantee across rows, which means that there are no multi-row transactions.
5. Column families have to be defined up front at table creation time.
6. Column qualifiers are dynamic and can be defined at write time. They are stored as *byte[ ]* so you can even put data in them.

A good way to learn these concepts is through an example problem. Let's try to model the Twitter relationships (users following other users) in HBase tables. Follower-followed relationships are essentially graphs, and there are specialized graph databases that work more efficiently with such data sets. However, this particular use case makes for a good example to model in HBase tables and allows us to highlight some interesting concepts.

The first step in starting to model tables is to define the access pattern of the application. In the context of follower-followed relationships for an application like Twitter, the access pattern can be defined as follows:

Read access pattern:

1. Who does a user follow?
2. Does a particular user A follow user B?
3. Who follows a particular user A?

Write access pattern:

1. User follows a new user.
2. User unfollows someone they were following.

Let's consider a few table design options and look at their pros and cons. Start with the table design shown in Figure 4. This table stores a list of users being followed by a particular user in a single row, where the row key is the user ID of the follower user and each column contains the user ID of the user being followed. A table of that design with data would look like Figure 5.

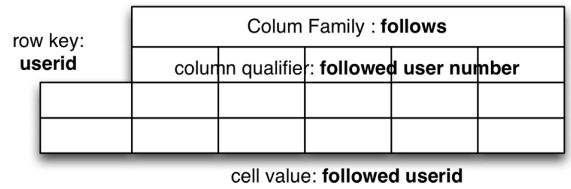


Figure 4: HBase table to persist the list of users a particular user is following

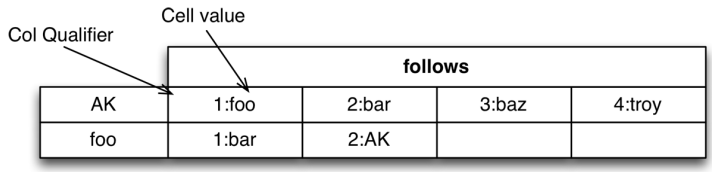


Figure 5: A table with sample data for the design shown in Figure 4

This table design works well for the first read pattern that was outlined. It also solves the second one, but it's likely to be expensive if the list of users being followed is large and will require iterating through the entire list to answer that question. Adding users is slightly tricky in this design. There is no counter being kept so there's no way for you to find out which number the next user should be given unless you read the entire row back before adding a user. That's expensive! A possible solution is to just keep a counter then and the table will now look like Figure 6.

	follows				
AK	1:foo	2:bar	3:baz	4:troy	count:4
foo	1:bar	2:AK	count:2		

Figure 6: A table with sample data for the design shown in Figure 4 but with a counter to keep count of the number of users being followed by a given user

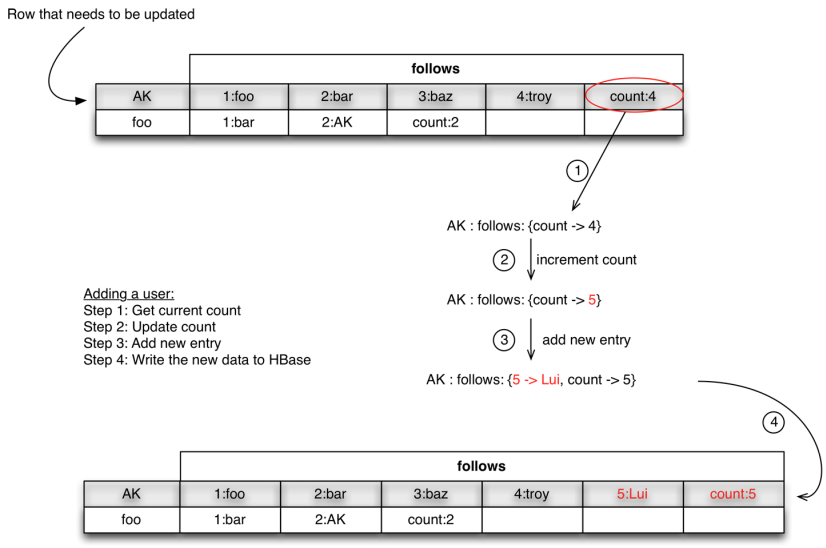


Figure 7: Steps required to add a new user to the list of followed users based on the table design from Figure 6

The design in Figure 6 is incrementally better than the earlier ones but doesn't solve all problems. Unfollowing users is still tricky since you have to read the entire row to find out which column you need to delete. It also isn't ideal for the counts since unfollowing will lead to holes. The biggest issue is that to add users, you have to implement some sort of transaction logic in the client code since HBase doesn't do transactions for you across rows or across RPC calls. The steps to add users in this scheme are shown in Figure 7.

One of the properties that I mentioned earlier was that the column qualifiers are dynamic and are stored as *byte[ ]* just like the cells. That gives you the ability to put arbitrary data in them, which might come to your rescue in this design. Consider the table in Figure 8. In this design, the count is not required, so the addition of users becomes less complicated. The unfollowing is also simplified. The cells in this case contain just some arbitrary small value and are of no consequence.

	follows			
AK	foo:1	bar:1	baz:1	troy:1
foo	bar:1	AK:1		

Figure 8: The relationship table with the cells now having the followed user's username as the column qualifier and an arbitrary string as the cell value.

This latest design solves almost all the access patterns that we defined. The one that's left is #3 on the read pattern list: who follows a particular user A? In the current design, since indexing is only done on the row key, you need to do a full table scan to answer this question. This tells you that the followed user should figure in the index somehow. There are two ways to solve this problem. First is to just maintain another table which contains the reverse list (user and a list of who all *follows* user). The second is to persist that information in the same table with different row keys (remember it's all byte arrays, and HBase doesn't care what you put in there). In both cases, you'll need to materialize that information separately so you can access it quickly, without doing large scans.

There are also further optimizations possible in the current table structure. Consider the table shown in Figure 9.

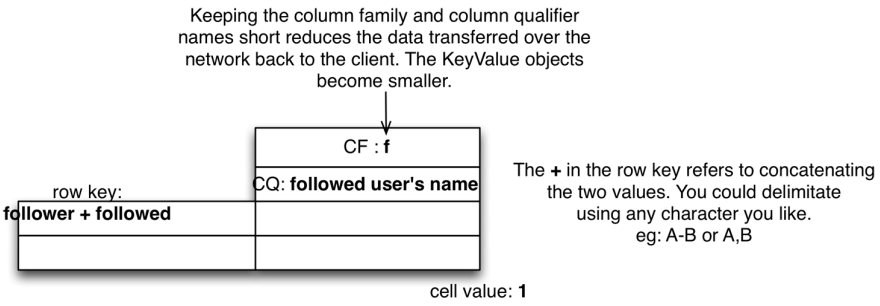


Figure 9: The relationship table with the row key containing the follower and the followed user

There are two things to note in this design: the row key now contains the follower and followed user; and the column family name has been shortened to f. The short column family name is an unrelated concept and could very well be done in the previous table as well. It just reduces the I/O load (both disk and network) by reducing the data that needs to be read/written from HBase since the family name is a part of every KeyValue [4] object that is returned back to the client. The first concept is what is more important here. Getting a list of followed users now becomes a short *Scan* instead of a *Get* operation. There is little performance impact of that as Gets are internally implemented as Scans of length 1. Unfollowing, and answering the question "Does A follow B?" become simple delete and get operations, respectively, and you don't need to iterate through the entire list of users in the row in the earlier table designs. That's a significantly cheaper way of answering that question, specially when the list of followed users is large.

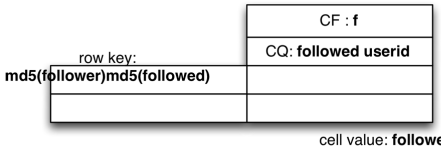
A table with sample data based on this design will look like Figure 10.

	f
AK+foo	James Foo:1
AK+bar	Jimmy Bar:1
AK+baz	Ricky Baz:1
AK+troy	Troy:1
foo+bar	Jimmy Bar:1
foo+AK	AK:1

Putting the user name in the column qualifier saves you from looking up the users table for the name of the user given an id. You can simply list out names or ids while looking at relationships just from this table. The downside of this is that you need to update the name in all the cells if the user updates their name in their profile. This is classic *Denormalization*.

Figure 10: Relationship table based on the design shown in Figure 9 with some sample data

Notice that the row key length is variable across the table. The variation can make it difficult to reason about performance since the data being transferred for every call to the table is variable. A solution to this problem is using hash values in the row keys. That's an interesting concept in its own regard and has other implications pertaining to row key design which are beyond the scope of this article. To get consistent row key length in the current tables, you can hash the individual user IDs and concatenate them, instead of concatenating the user IDs themselves. Since you'll always know the users you are querying for, you can recalculate the hash and query the table using the resulting digest values. The table with hash values will look like Figure 11.



Using MD5 of the user ids gives you fixed lengths instead of variable length user ids. You don't need concatenation logic anymore.

Figure 11: Using MD5s as a part of row keys to achieve fixed lengths. This also allows you to get rid of the + delimiter that we needed so far. The row keys now consist of fixed length portions, with each user ID being 16 bytes.

This table design allows for effectively answering all the access patterns that we outlined earlier.

Summary

This article covered the basics of HBase schema design. I started with a description of the data model and went on to discuss some of the factors to think about while designing HBase tables. There is much more to explore and learn in HBase table design which can be built on top of these fundamentals. The key takeaways from this article are:

- Row keys are the single most important aspect of an HBase table design and determine how your application will interact with the HBase tables. They also affect the performance you can extract out of HBase.
- HBase tables are flexible, and you can store anything in the form of *byte[ ]*.
- Store everything with similar access patterns in the same column family.
- Indexing is only done for the *Keys*. Use this to your advantage.
- Tall tables can potentially allow you faster and simpler operations, but you trade off atomicity. Wide tables, where each row has lots of columns, allow for atomicity at the row level.
- Think how you can accomplish your access patterns in single API calls rather than multiple API calls. HBase does not have cross-row transactions, and you want to avoid building that logic in your client code.
- Hashing allows for fixed length keys and better distribution but takes away the ordering implied by using strings as keys.
- Column qualifiers can be used to store data, just like the cells themselves.
- The length of the column qualifiers impact the storage footprint since you can put data in them. Length also affects the disk and network I/O cost when the data is accessed. Be concise.
- The length of the column family name impacts the size of data sent over the wire to the client (in KeyValue objects). Be concise.

References

[1] Apache Hadoop project: http://hadoop.apache.org.

[2] Apache HBase project: http://hbase.apache.org.

[3] HBase client API: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/package-summary.html.

[4] HBase KeyValue API: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/KeyValue.html.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '06), USENIX, 2006, pp. 205–218.