Cassandra Data Modeling Best Practices, Part 1

by Jay Patel on 07/16/2012 [http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/]

in Data Infrastructure and Services

This is the first in a series of posts on Cassandra data modeling, implementation, operations, and related practices that guide our Cassandra utilization at eBay. Some of these best practices we've learned from public forums, many are new to us, and a few still are arguable and could benefit from further experience.

September 2014 Update: Readers should note that this article describes data modeling techniques based on Cassandra's Thrift API. Please see https://wiki.apache.org/cassandra/DataModel for CQL API based techniques.

August 2015 Update: Readers can also sign up a free online self-paced course on how to model their data in Apache Cassandra from https://academy.datastax.com/courses/ds220-data-modeling?dxt=blogposting.

In this part, I'll cover a few basic practices and walk through a detailed example. Even if you don't know anything about Cassandra, you should be able to follow almost everything.

A few words on Cassandra @ eBay

We've been trying out Cassandra for more than a year. Cassandra is now serving a handful of use cases ranging from write-heavy logging and tracking, to mixed workload. One of them serves our "Social Signal" project, which enables like/own/want features on eBay product pages. A few use cases have reached production, while more are in development.

Our Cassandra deployment is not huge, but it's growing at a healthy pace. In the past couple of months, we've deployed dozens of nodes across several small clusters spanning multiple data centers. You may ask, why multiple clusters? We isolate clusters by functional area and criticality. Use cases with similar criticality from the same functional area share the same cluster, but reside in different keyspaces.

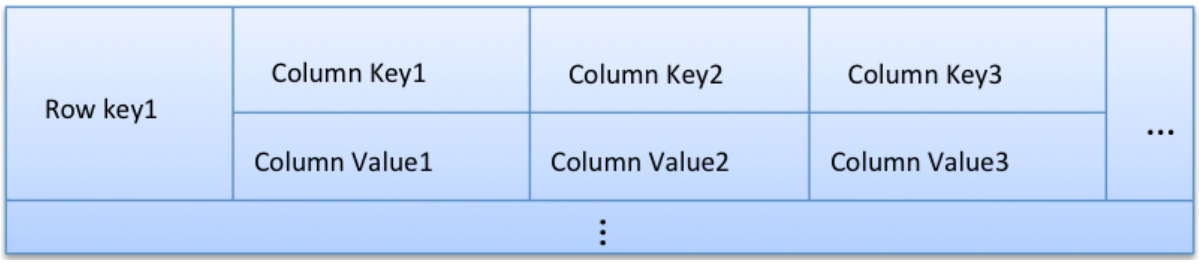
RedLaser, Hunch, and other eBay adjacencies are also trying out Cassandra for various purposes. In addition to Cassandra, we also utilize MongoDB and HBase. I won't discuss these now, but suffice it to say we believe each has its own merit.

I'm sure you have more questions at this point. But I won't tell you the full story yet. At the upcoming Cassandra summit, I'll go into detail about each use case, the data model, multi-datacenter deployment, lessons learned, and more.

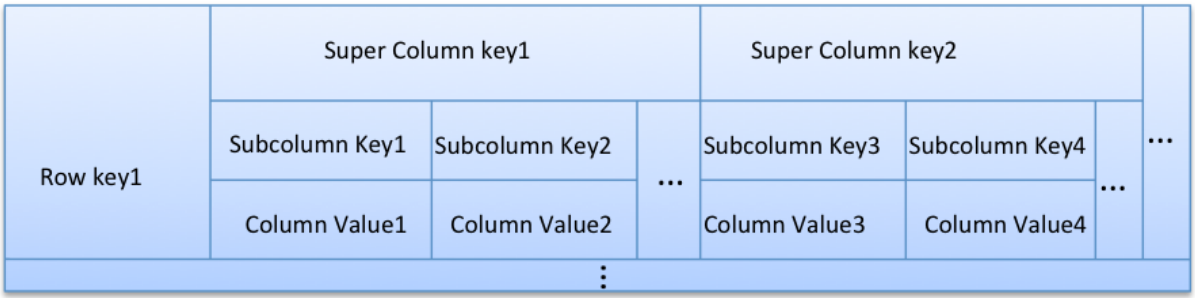
The focus of this post is Cassandra data modeling best practices that we follow at eBay. So, let's jump in with a few notes about terminology and representations I'll be using for each post in this series.

Terms and Conventions

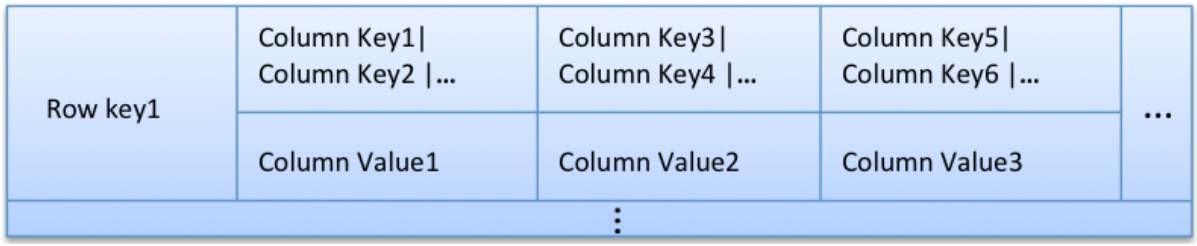
- The terms "Column Name" and "Column Key" are used interchangeably. Similarly, "Super Column Name" and "Super Column Key" are used interchangeably.
- The following layout represents a row in a Column Family (CF):



- The following layout represents a row in a Super Column Family (SCF):



- The following layout represents a row in a Column Family with composite columns. Parts of a composite column are separated by '|'. Note that this is just a representation convention; Cassandra's built-in composite type encodes differently, not using '|'. (BTW, this post doesn't require you to have detailed knowledge of super columns and composite columns.)



With that, let's start with the first practice!

Don't think of a relational table

Instead, think of a nested, sorted map data structure.

The following relational model analogy is often used to introduce Cassandra to newcomers:

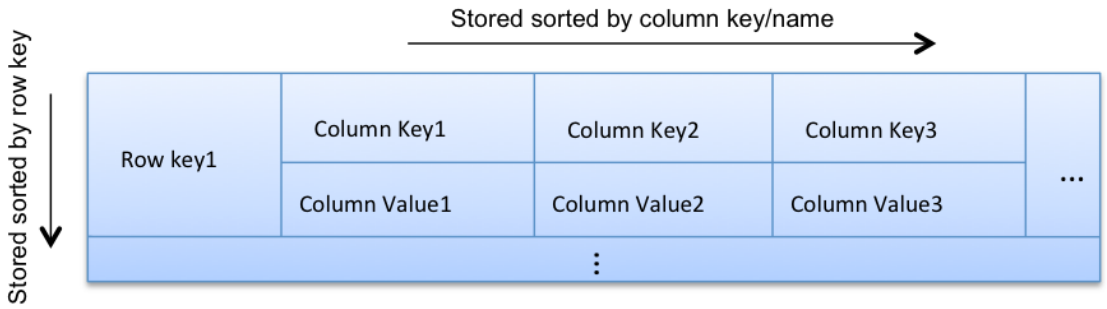
Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

This analogy helps make the transition from the relational to non-relational world. But don't use this analogy while designing Cassandra column families. Instead, think of the Cassandra column family as a map of a map: an outer map keyed by a row key, and an inner map keyed by a column key. Both maps are sorted.

SortedMap<RowKey, SortedMap<ColumnKey, ColumnValue>>

Why?

A nested sorted map is a more accurate analogy than a relational table, and will help you make the right decisions about your Cassandra data model.



How?

- A map gives efficient key lookup, and the sorted nature gives efficient scans. In Cassandra, we can use row keys and column keys to do efficient lookups and range scans.
- The number of column keys is unbounded. In other words, you can have wide rows.
- A key can itself hold a value. In other words, you can have a valueless column.

Range scan on row keys is possible only when data is partitioned in a cluster using Order Preserving Partitioner (OOP). OOP is almost never used. So, you can think of the outer map as unsorted:

Map<RowKey, SortedMap<ColumnKey, ColumnValue>>

As mentioned earlier, there is something called a "Super Column" in Cassandra. Think of this as a grouping of columns, which turns our two nested maps into three nested maps as follows:

Map<RowKey, SortedMap<SuperColumnKey, SortedMap<ColumnKey, ColumnValue>>>

Notes:

- You need to pass the timestamp with each column value, for Cassandra to use internally for conflict resolution. However, the timestamp can be safely ignored during modeling. Also, do not plan to use timestamps as data in your application. They're not for you, and they do not define new versions of your data (unlike in HBase).
- The Cassandra community has heavily criticized the implementation of Super Column because of performance concerns and the lack of support for secondary indexes. The same "super column like" functionality (or even better) can be achieved by using composite columns.

Model column families around query patterns

But start your design with entities and relationships, if you can.

- Unlike in relational databases, it's not easy to tune or introduce new query patterns in Cassandra by simply creating secondary indexes or building complex SQLs (using joins, order by, group by?) because of its high-scale distributed nature. So think about query patterns up front, and design column families accordingly.
- Remember the lesson of the nested sorted map, and think how you can organize data into that map to satisfy your query requirements of fast look-up/ordering/grouping/filtering/aggregation/etc.

However, entities and their relationships still matter (unless the use case is special – perhaps storing logs or other time series data?). What if I gave you query patterns to create a Cassandra model for an e-commerce website, but didn't tell you anything about the entities and relationships? You might try to figure out entities and relationships, knowingly or unknowingly, from the query patterns or from your prior understanding of the domain (because entities and relationships are how we perceive the real world). It's important to understand and start with entities and relationships, then continue modeling around query patterns by de-normalizing and duplicating. If this sounds confusing, make sure to go through the detailed example later in this post.

Note: It also helps to identify the most frequent query patterns and isolate the less frequent. Some queries might be executed only a few thousand times, while others a billion times. Also consider which queries are sensitive to latency and which are not. Make sure your model first satisfies the most frequent and critical queries.

De-normalize and duplicate for read performance

But don't de-normalize if you don't need to. It's all about finding the right balance.

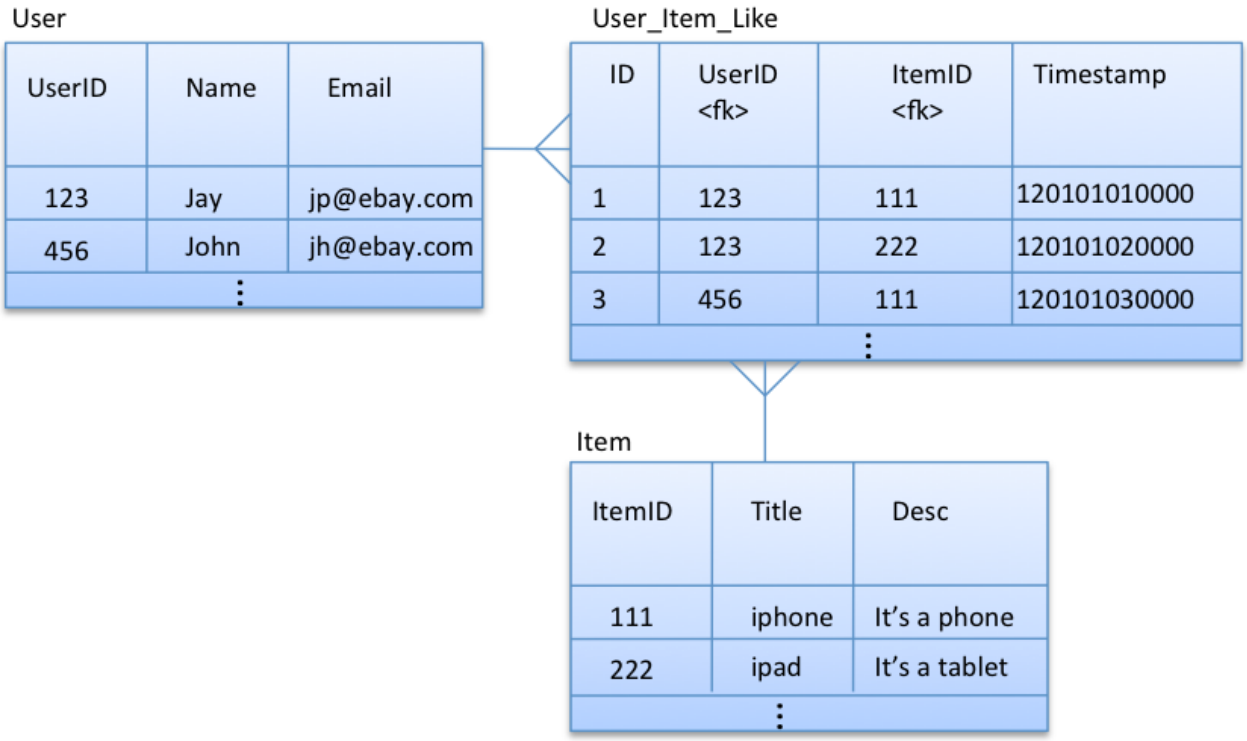
In the relational world, the pros of normalization are well understood: less data duplication, fewer data modification anomalies, conceptually cleaner, easier to maintain, and so on. The cons are also understood: that queries might perform slowly if many tables are joined, etc. The same holds true in Cassandra, but the cons are magnified since it's distributed and of course there are no joins (since it's high-scale distributed!). So with a fully normalized schema, reads may perform much worse.

This and the previous practice (modeling around query patterns) are so important that I would like to further elaborate by devoting the rest of the post to a detailed example.

Note: The example discussed below is just for demonstration purposes, and does not represent the data model used for Cassandra projects within eBay.

Example: 'Like' relationship between User & Item

This example concerns the functionality of an e-commerce system where users can like one or more items. One user can like multiple items and one item can be liked by multiple users, leading to a many-to-many relationship as shown in the relational model below:

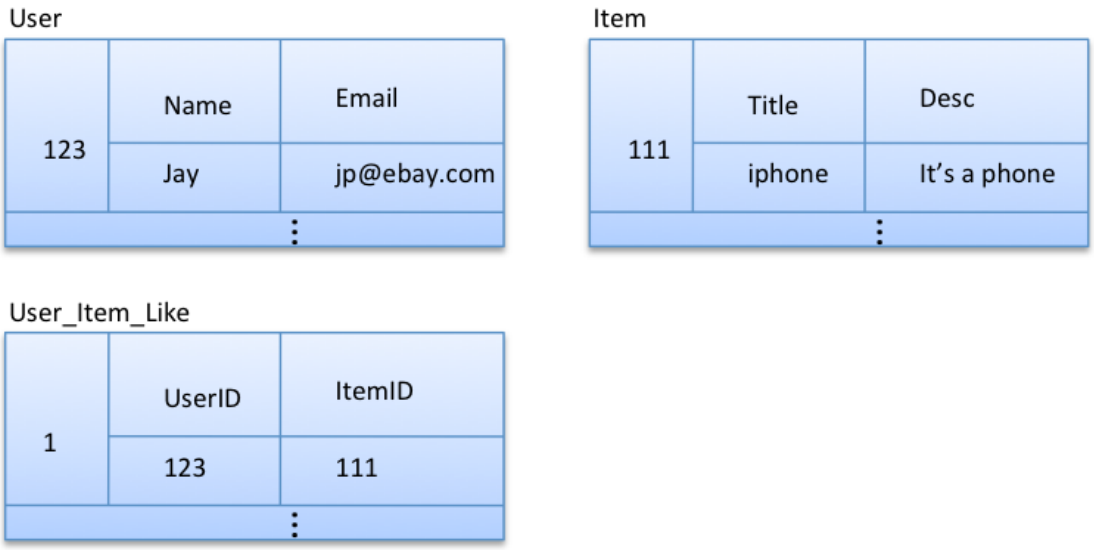


For this example, let's say we would like to query data as follows:

- *Get user by user id*
- *Get item by item id*
- *Get all the items that a particular user likes*
- *Get all the users who like a particular item*

Below are some options for modeling the data in Cassandra, in order of the lowest to the highest de-normalization. The best option depends on the query patterns, as you'll soon see.

Option 1: Exact replica of relational model

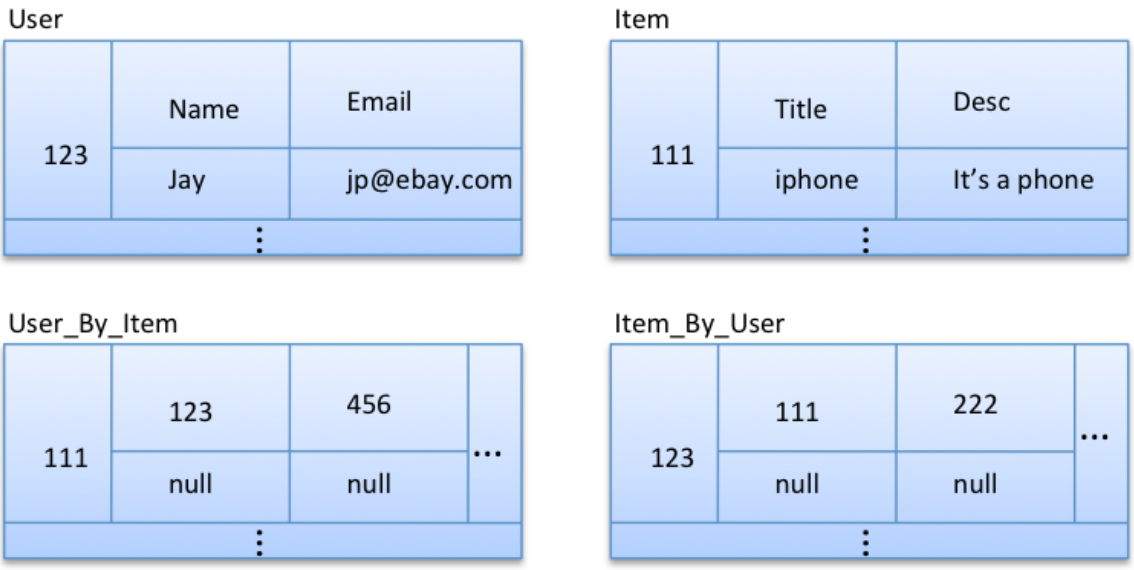


This model supports querying user data by user id and item data by item id. But there is no easy way to query all the items that a particular user likes or all the users who like a particular item.

This is the worst way of modeling for this use case. Basically, User_Item_Like is not modeled correctly here.

Note that the 'timestamp' column (storing when the user liked the item) is dropped from User_Item_Like for simplicity. I'll introduce that column later.

Option 2: Normalized entities with custom indexes



This model has fairly normalized entities, except that user id and item id mapping is stored twice, first

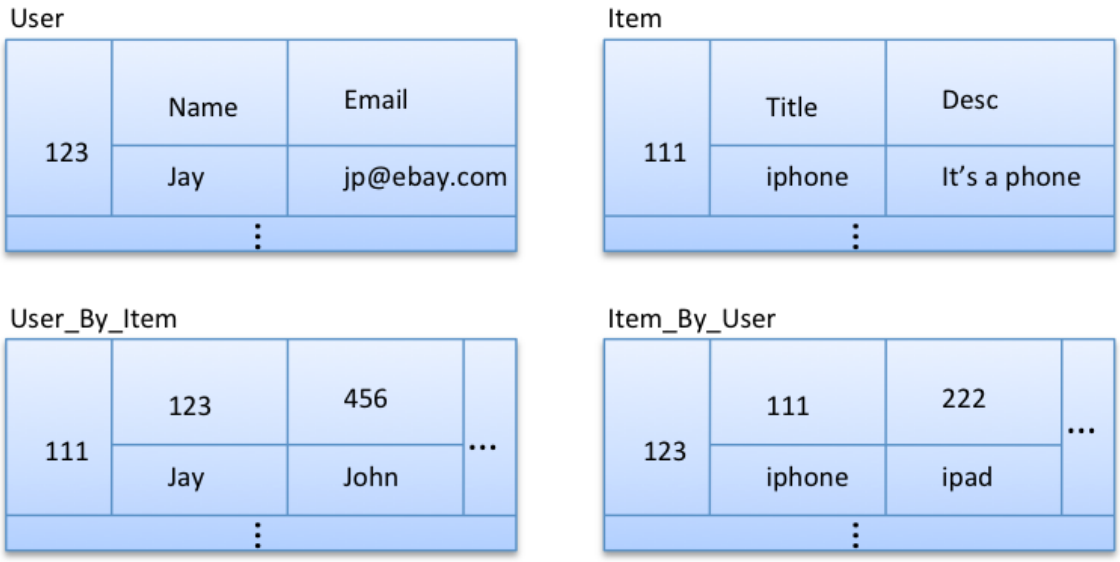
by item id and second by user id.

Here, we can easily query all the items that a particular user likes using Item_By_User, and all the users who like a particular item using User_By_Item. We refer to these column families as custom secondary indexes, but they're just other column families.

Let's say we always want to get the item title in addition to the item id when we query items liked by a particular user. In the current model, we first need to query Item_By_User to get all the item ids that a given user likes; and then for each item id, we need to query Item to get the title. Similarly, let's say we always want to get all the usernames in addition to user ids when we query users who like a particular item. With the current model, we first need to query User_By_Item to get the ids for all users who like a given item; and then for each user id, we need to query User to get the username. It's possible that one item is liked by a couple hundred users, or an active user has liked many items — which will cause many additional queries when we look up usernames who like a given item and vice versa. So, it's better to optimize by de-normalizing item title in Item_by_User, and username in User_by_Item, as shown in option 3.

Note: Even if you can batch your reads, they will still be slower because Cassandra (Coordinator node, to be specific) has to query each row separately underneath (usually from different nodes). Batch read will help only by avoiding the round trip — which is good, so you should always try to leverage it.

Option 3: Normalized entities with de-normalization into custom indexes



In this model, title and username are de-normalized in User_By_Item and Item_By_User respectively. This allows us to efficiently query all the item titles liked by a given user, and all the user names who like a given item. This is a fair amount of de-normalization for this use case.

What if we want to get all the information (title, desc, price, etc.) about the items liked by a given user? But we need to ask ourselves whether we really need this query, particularly for this use case. We can show all the item titles that a user likes and pull additional information only when the user asks for it (by clicking on a title). So, it's better not to do extreme de-normalization for this use case. (However, it's common to show both title and price up front. It's easy to do; I'll leave it for you to pursue if you wish.)

Let's consider the following two query patterns:

- For a given item id, get all of the item data (title, desc, etc.) along with the names of the users who liked that item.
- For a given user id, get all of the user data along with the item titles liked by that user.

These are reasonable queries for item detail and user detail pages in an application. Both will perform well with this model. Both will cause two lookups, one to query item data (or user data) and another to query user names (or item titles). As the user becomes more active (starts liking thousands of items?) or the item becomes hotter (liked by a few million users?), the number of lookups will not grow; it will remain constant at two. That's not bad, and de-normalization may not yield much benefit like we had when moving from option 2 to option 3. However, let's see how we can optimize further in option 4.

Option 4: Partially de-normalized entities

User

	UserInfo		Likes		
123	Name	Email	111	222	...
	Jay	jp@ebay.com	iphone	ipad	
⋮					

Item

	ItemInfo		LikedBy		
111	Title	Desc	123	4556	...
	iphone	It's a phone	Jay	John	
⋮					

Definitely, option 4 looks messy. In terms of savings, it's not like what we had in option 3.

If User and Item are highly shared entities (similar to what we have at eBay), I would prefer option 3 over this option.

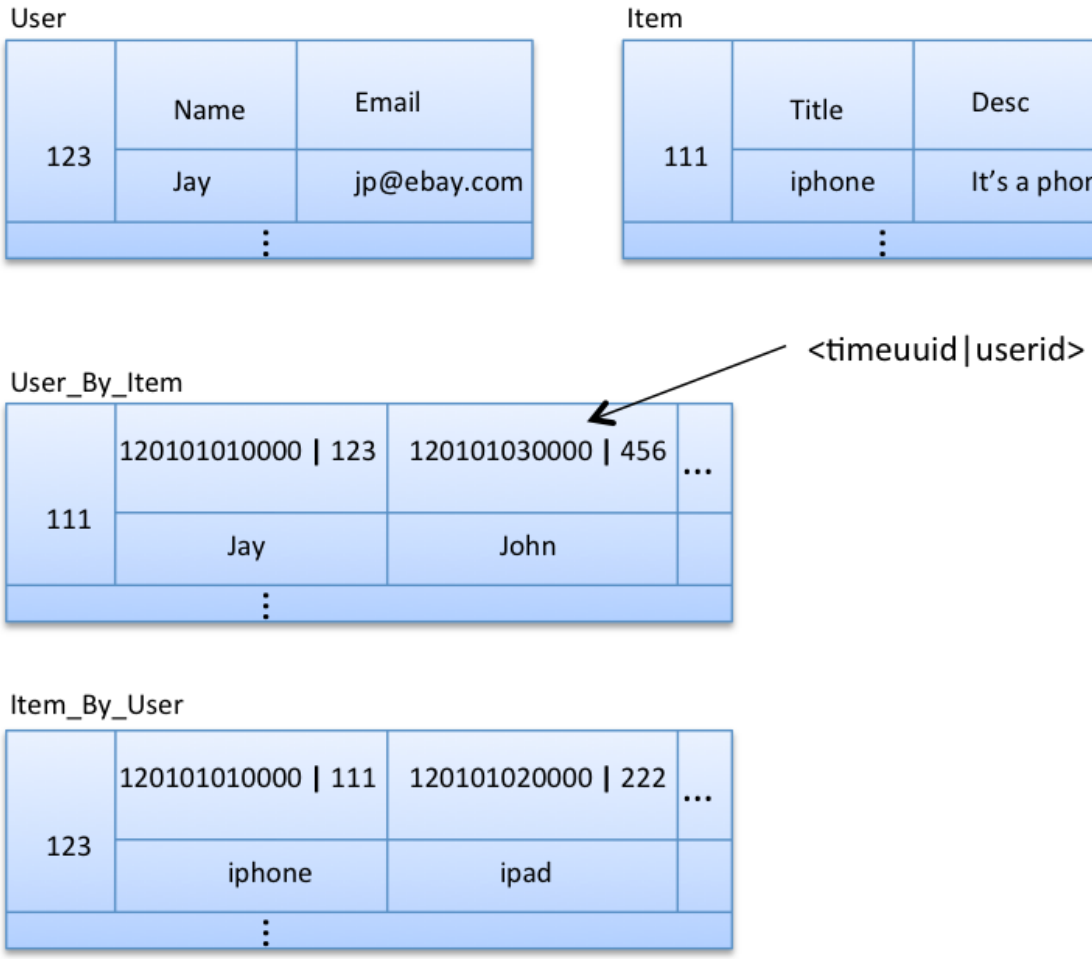
We've used the term "partially de-normalized" here because we're not de-normalizing all item data into the User entity or all user data into the Item entity. I won't even consider showing extreme de-normalization (keeping all item data in User and all user data in Item), as you probably agree that it doesn't make sense for this use case.

Note: I've used Super Column here just for demonstration purposes. Almost all the time, you should favor composite columns over Super Column.

The best model

The winner is Option 3, particularly for this example. We've left out timestamp, but let's include it in the final model below as timeuuid(type-1 uuid). Note that timeuuid and userid together form a composite column key in User_By_Item and Item_By_User column families.

Recall that column keys are physically stored sorted. Here our column keys are stored sorted by timeuuid in both User_By_Item and Item_By_User, which makes range queries on time slots very efficient. With this model, we can efficiently query (via range scans) the *most recent* users who like a given item and the *most recent* items liked by a given user, without reading all the columns of a row.



Summary

We've covered a few fundamental practices and walked through a detailed example to help you get started with Cassandra data model design. Here are the key takeaways:

- Don't think of a relational table, but think of a nested sorted map data structure while designing Cassandra column families.
- Model column families around query patterns. But start your design with entities and relationships, if you can.
- De-normalize and duplicate for read performance. But don't de-normalize if you don't need to.
- Remember that there are many ways to model. The best way depends on your use case and query patterns.

What I've not mentioned here are special, but common, use cases such as logging, monitoring, real-time analytics (rollups, counters), or other time series data. However, practices discussed here do apply there. In addition, there are known common techniques or patterns used to model these time series data in Cassandra. At eBay, we also use some of those techniques and would love to share about them in upcoming posts. For more information on modeling time series data, I would recommend reading

Advanced time series with Cassandra and Metric collection and storage. Also, if you're new to Cassandra, make sure to scan through DataStax documentation on Cassandra.

UPDATE: Part 2 about Cassandra is now published.

— Jay Patel, architect@eBay

51 Replies

51 thoughts on "Cassandra Data Modeling Best Practices, Part 1"

Aric Rosenbaum
07/17/2012 at 6:18AM

Nice article. Looking fwd to Parts 2 and 3.

In your 2nd and 3rd reference to:
SortedMap<RowKey, SortedMap>
Shouldn't it be?:
Map<RowKey, SortedMap>

Thx.

Andrey
01/16/2013 at 3:37PM

Shouldn't it be
HashMap<RowKey, SortedMap>

?

Jay Patel Post author
07/17/2012 at 9:23AM

Thanks Aric for spotting the typo :). It's corrected now.

Jonathan Ellis
07/17/2012 at 10:30AM

Thanks for the post, Jay! Looking forward to hearing more about your use case.

Pingback: Cassandra Data Modeling Best Practices, Part 2 — eBay Tech Blog

Darrell Denlinger
08/15/2012 at 10:12PM

What if your de-normalized data needs to change? For example John changes his name in his profile to Johnny. We can update the User CF, but will we know to hunt down the User_By_Item entry and update it?

Jay Patel Post author
08/16/2012 at 2:04PM

Yes, that's the price one would pay for read performance. Application needs to know all the places

where the data are duplicated & needs to correct them, synchronously or asynchronously. That's why, I recommend not to de-normalize if you don't need to; normalization still matters! Particularly for this example, I hope users won't be changing their names that often. If they do, we can make change in the User CF syncly and correct the User_By_Item asyncly. And, to avoid iterating all the data in the User_By_Item, we can first find out all the items liked by that user from the Item_By_User.

Jens
08/22/2012 at 1:53AM

Helpful post! Thx.

But how would a CQL 3 CREATE stmt look like for CFs in Option 2 for "Item_By_User" and "User_By_Item"?
or I don't understand >>We refer to these column families as custom secondary indexes, but they're just other column families.<<

Jay Patel Post author
08/22/2012 at 6:39PM

"Item_By_User" and "User_By_Item" are normal column families similar to "User" and "Item" except they're dynamic/wide as column names hold actual data (item ids or user ids). Particularly for this use case, they're serving a purpose of secondary indexes & so termed as custom-built secondary indexes, but they're just normal column families after all.
With CQL, one can use CREATE TABLE/COLUMN FAMILY statement like any other column family creation. Just to point out that "Item_By_User" and "User_By_Item" are not built-in Cassandra secondary indexes created by CQL's CREATE INDEX statement. I hope this clarifies.

Pingback: Data Modeling in Cassandra | Big Data | DATAVERSITY

Jens

08/23/2012 at 1:34AM

Thx for promt reply Jay.
Yes, clarified. The CFs work as a custom index for the useritem relation.

But is it possible to create such wide rows where colum names hold actual data with CQL 3.0?

E.g. this would not work:

CREATE TABLE user_by_item (
userid bigint,
itemid bigint,
PRIMARY KEY(userid, itemid)
);

because it says that there must be at least one column in CF which is not part of the composite primary key (which causes wide rows)!?
Did I get it right?
🙂

Jay Patel Post author
08/24/2012 at 3:48PM

It's a limitation of CQL3 today but I see it fixed for version 1.2.
https://issues.apache.org/jira/browse/CASSANDRA-4361

Pingback: Cassandra Data Modeling Best Practices, Part 1 | My Blog

Ozimyj
09/11/2012 at 4:16AM

So what if I need more information (e.g price, quantity etc) of the item? How would you design this schema?

Another example can be a person that has 5 addresses and each address consists of a city, country, line1, line2, phone etc.

Jay Patel Post author
09/17/2012 at 11:46AM

One way is to add qualifier as part of a composite column name. For example, in Item_By_User, column names can be of something like userid|"title" and userid|"price" with column values holding the actual title or price depending on the column name.
Another way: If there're no updates for title or price, and if use case needs to retrieve them together, it may be efficient to just dump everything as one column value (json object?). You can come up with more ways depending on the use case need. I hope this helps.

Idan
09/30/2012 at 1:52PM

What a mass. Their should by a new technology innovation that will save us from this awful mass. I would rather go with ScaleBase solution for MySQL. If my application needs to scale to high demand, I would probably make money to invest in a scalable MySQL solution like Clustrix or ScaleBase. Just think of all the updates/writes that you need to make in order to update the data in all the places that the data is duplicated. How did we come to times that people can land on the moon and databases are still lack the the technology to solve those problems. Hopefully NouDB will do something that be revolutionary. I also want to see those solution as a DBaaS. Developers should emphasize on the ideas and coding their apps, not becoming DBAs.

Tejinder Singh
11/05/2012 at 11:31PM

how can i implement the concept of primary key (auto_increment , unique) by using cassandra database model?

Pingback: Guidelines for Modeling and Optimizing NoSQL Databases - LaunchAny

Srdjan
01/02/2013 at 8:04PM

What do you think about this model:

ColumnFamily User_item_like:

TimeUUID
12010101000 123(111) 111(123)
12010102000 123(222) 222(123)
12010103000 456(111) 111(456)

We could use MapReduce to find all items liked by user 123 or all users who liked item 111.

Load balance depends on key selection so this would insert faster than if you insert a lot of timestamps for the same item key. If a million of users want to like the same page than inserts would not be balances because we are updating the same item_key but with this approach everything would work fine.
What do you think of this?

Srdjan
01/03/2013 at 12:37PM

Actually I was thinking of having a column family for every user and every item and also store likes in these tables. Is there a limitation that would prevent me to have a large number of column families?

Srdjan
01/04/2013 at 12:54AM

Well I can answer myself, large number of column families would make large number of files which would slow than the OS.
However we could have partitioning by item bucket (1000 items) but that would lead to more coding.

P.S. Sorry for spamming your blog.

bhaskar teja
01/08/2013 at 12:43PM

How to handle transactional consistency with multiple column family writes about the same data?

Chandan Patra
01/09/2013 at 11:41PM

Hi Jay,
Really nice article. One piece of information that I want to know is where these clusters are hosted. Any public cloud like AWS, Rackspace or your own private datacenters?

Thanks,
Chandan

Pingback: Cassandraクラスタと、DataStax OpsCenterの構築 | さぶみっと！JAPAN

SKJ
03/26/2013 at 4:44AM

Great article Jay. It was very much helpful me. I am trying to learn Cassandra from couple of days and was getting confused somewhere. This blog helped my to resolve my confusions. i like the example and picturizing the concepts using SortedMap.

Gaurav
05/21/2013 at 2:40AM

Thanks for nice article.

In option 4, How to create the schema of User_By_Item via CQL or CLI ?

Where Item id act as row id and timestamp+userid act as composite column

Jon Slenk
05/23/2013 at 3:39PM

A whacky thought comes to mind: Has anybody experimented with keeping track of the denormalizations in the database, so that one can automagically find the places to update when the time comes? Just thinking that it might be nifty to put as much db-meta-data into the db, and thus be able to make the application avoid hard-coded denormalization knowledge.

Another: Per DDD/CQRS, has anybody compare-contrast-experimented with a normalized SQL write-store than then writes out to a denormalized (SQL or Non) read-store? Wonder how much delay the business can stomach. (Only relevant if there are people who want to be able to run random arbitrary custom queries on the data.)

Arthur
06/30/2013 at 10:51PM

Hi Team:
I am the managing editor of InfoQ China(http://www.infoq.com/cn) which focus on international software development, including software development,operations.We like this series articles and plan to translate them into Chinese.

Before we translate them into Chinese,I want to ask for your permission first!!
In exchange,We will put the English title and link at the end of the Chinese articles,If our reader want to read more about this,they can click back to your website.

Thanks a lot,hope to get your help,any questions,contact me freely:)

Pingback: Cassandra | Annotary

Pingback: A good approach for writing a lot of data that can be liked or disliked | BlogoSfera

Pankaj M
09/22/2013 at 12:04PM

Jay,

Thanks for the very helpful post. I am new to Cassandra. I have not been able to figure out the CQL statements for option 3. I went through this document but couldn't get it:

http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/

If at all possible, I would like to get sample CREATE , INSERT and UPDATE statement for Option 3 : User_By_Item table.

Thanks a lot in advance.

Regards,

Pankaj

shuron
10/25/2013 at 4:44AM

Thank you. Notation as:
Map<RowKey, SortedMap<SuperColumnKey, SortedMap>>
helped me.

Is it possible to have normal columns not only inside also aside of super columns in one column family?

Pingback: Cassandra Explained in 5 Minutes or Less - blog.credera.com

Pingback: Divide and Conquer | Art of the DBA

tom haughey
01/18/2014 at 11:09AM

Jay, I have read and enjoyed your articles and I am an ardent user of eBay. I have a question about the use of RDBMS at eBay. I see Oracle in your PPTs. What use do you make of relational? Clearly, Cassandra is good for collecting WatchList, Bids, Follower and Following, and similar capabilities. But, when a bid has been accepted and a purchase made, is this transaction performed on Cassandra or Oracle? And again, is the payment transacted on Oracle or Cassandra?

I never looked to NOSQL for its transaction processing capability, at least for heavy duty transactions such as a stock trade, an insurance premium payment, a health care claim and such transactions involving risk and requiring integrity. I note that a recent Gartner Magic Quadrant report on OLTP databases includes both RDBMSs and NOSQL data stores as OLTP. RDBMSs in the leaders quadrant, and NOSQL data stores in the lower left followers quadrant. Gartner classifies both as transactional, which was a surprise to me. To me, light weight transactions such as a Tweet or FB post are not in the same category as the heavier duty transactions I listed above, such as a stock trade. In my opinion, the heavy weight transactions still belong in the domain of the RDBMS.

I was glad to see you point on the dangers of redundancy and the need to keep data as normalized as reasonable. Some NOSQL advocates seem a little frivolous about this.

I welcome your comments.

Tom

Brando Miranda
03/25/2014 at 5:14PM

Hi Jay,

Excellent article! Thanks for sharing your knowledge!
🙂

I have a question though, its probably really simple and its probably more for my lack in experience with Cassandra but, when you create some of your tables, say for example mapping userIDs to the items that users likes. Each user might have different amount of items it has relative to other users. Does that mean we have to know before had the max number of items *all* users might like? I.e. can the table just keep growing? I suspect I am wrong right? How do tell Cassandra that you want a varying number of additional columns?

Thanks!
🙂

Pingback: How to make Cassandra have a varying column key for a specific row key? | BlogoSfera

Pingback: How do you make cassandra store tuples of of values in one row, without actually storing a list? Is that even possible? | BlogoSfera

Pingback: Cassandra – Data Model for Twitter – Part 2 | Treselle Systems

Pritish Shah
04/24/2014 at 10:16AM

Hi Jay,

Awesome article.. It clears so many doubts.

Really appreciate that.
🙂

Thank you so much.

Pritish Shah
04/24/2014 at 10:19AM

Hey Brando,

I read one article about dynamic column family in Cassandra, where number columns are not restricted. For your scenario, it might be good fit. Hector allows to create dynamic column family.

Pingback: cassandra overview | VLDB.ir

Pingback: Why MySQL | Kevin

sreeni
07/10/2014 at 9:57AM

good article!!
one of the very few articles in web which goes into details of data modeling using column families. this is right on top.

big data training in Seattle
07/30/2014 at 10:56AM

Nice article, Train on latest Big data technologies at 3 days Big Data Bootcamp – Seattle (Aug 8-10, 2014) use offer code VLINKEDIN and Save Up to $50 , Register at

http://globalbigdataconference.com/34/seattle/big-data-bootcamp/event.html

---

Pingback: Cassandra Data Modeling Best Practices, Part 1 | KTH DAISY(Data Intelligence System)

gaurav
11/19/2014 at 1:57AM

In Item_By_User column family, how many items column can be stored?

Thanks
Gaurav

Pingback: Wide rows vs Collections in Cassandra - BlogoSfera

Pingback: confusion on cassandra data modeling - BlogoSfera

Pingback: Escaping From Disco-Era Data Modeling

Pingback: If Cassandra facilitates the use of non-normalized data, how do users edit it without creating inconsistencies? - BlogoSfera