

Cassandra Modeling

Best Practices and Examples



Jay Patel

Architect, Platform Systems

@pateljay3001

That's me



Technical Architect @ eBay

Passion for building high-scale systems

Architecting eBay's next-gen data platform

Prototyped first version of eBay's cloud platform

Built various social applications at IBM & eBay

Implemented telecom softwares at an early startup

Pursuing machine-learning at Stanford

Entrepreneurial minded

<http://www.jaykumarpatel.com>

eBay Marketplaces

\$75 billion+ per year in goods are sold on eBay

112 million active users

Petabytes of data

2 billion+ page views/day

400+ million items for sale

Thousands of servers

Multiple Datacenters

Big Data

turning over a **TB** every **second**

24x7x365

Always online

Billions of SQLs/day

99.98+% Availability

Near-Real-time



eBay Site Data Infrastructure

A heterogeneous mixture

The Oracle logo, featuring the word "ORACLE" in a bold, red, sans-serif font with a registered trademark symbol.

Thousands of nodes
> 2K sharded logical host
> 16K tables
> 27K indexes
> 140 billion SQLs/day
> 5 PB provisioned



Hundreds of nodes
> 250 TB provisioned
(local HDD + shared SSD)
> 6 billion writes/day
> 5 billion reads/day



Hundreds of nodes
> 50 TB
> 2 billion ops/day



Hundreds of nodes
> 40 billion SQLs/day

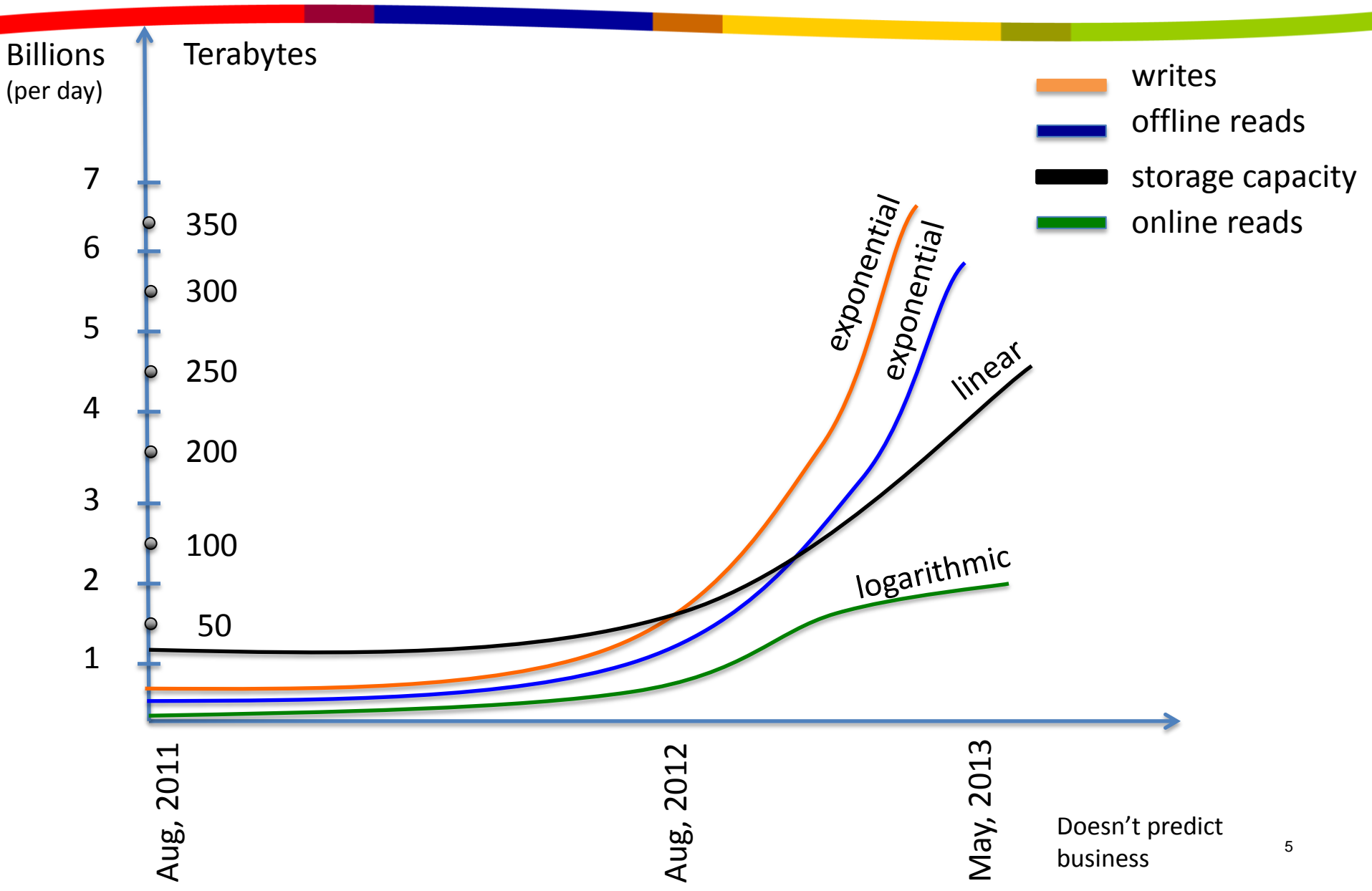


Thousands of nodes
The world largest cluster
with 2K+ nodes



Dozens of nodes

Cassandra at eBay



eBay's Real Time Big Data on Cassandra



- Social Signals on eBay Product & Item pages
- Mobile notification logging and tracking
- Tracking for fraud detection
- SOA request/response payload logging
- Metrics collections and real-time reporting for thousands of servers
- Personalization Data Service
- NextGen Recommendation System with real-time taste graph for eBay users
- Cloud CMS change history storage
- Order Payment Management logging
- Shipment tracking
- RedLaser server logs and analytics

More in upcoming Cassandra summit..

Cassandra Modeling

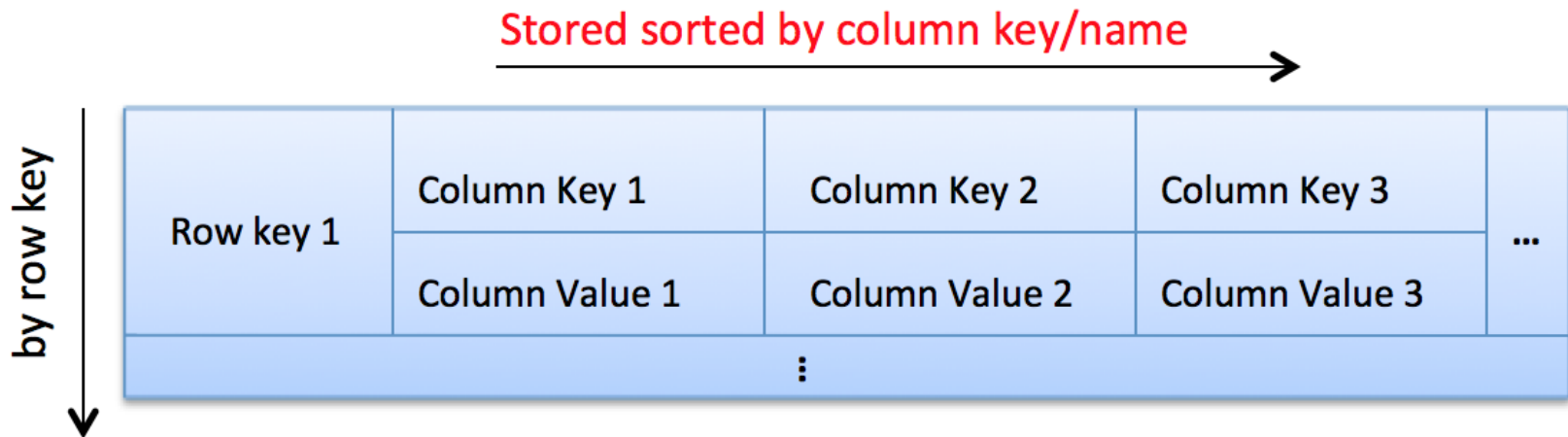
**The more I read,
The more I'm confused!**



Thanks to CQL for confusing further..

Intro to Cassandra Data Model

Non-relational, sparse model designed for high scale distributed storage



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

Data Model – Example Column Families

User

123456	Name	Email	Phone	State
	Jay	jay@ebay.com	4080004168	CA
⋮				

Static column family

ItemLikes

123456	Item Ids		...
	121212	343434	
	iphone	ipad	
⋮			

Dynamic column family
(aka, wide rows)

Data Model – Super & Composite column

User

123456	UserInfo		Likes		
	Name	Email	121212	343434	...
	Jay	jay@ebay.com	iphone	ipad	
⋮					

Grouping using
Super Column

User

123456	UserInfo Name	UserInfo Email	Likes 121212	Likes 343434	...
	Jay	jay@ebay.com	iphone	ipad	
⋮					

Grouping using
Composite column
name

Note – make sure to read practice 18 later in the slides.

1 Don't think of a relational table

Instead, think of a nested, sorted map data structure

`SortedMap<RowKey, SortedMap<ColumnKey, ColumnValue>>`

Why?

- Physical model is more similar to sorted map than relational

How?

- Map gives efficient key lookup & sorted nature gives efficient scans
- Unbounded no. of column keys
- Key can itself hold value

Each column has timestamp associated. Ignore it during modeling

Refinement - Think of outer map as unsorted

1

Map<RowKey, SortedMap<ColumnKey, ColumnValue>>

Why?

- Row keys are sorted in natural order only if OPP is used. OPP is not recommended!

Think of a below visual!

Stored sorted by column key/name



Row key 1	Column Key 1	Column Key 2	Column Key 3	...
	Column Value 1	Column Value 2	Column Value 3	
⋮				


How about super column?

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

Super column is the past! Instead, use Composite column names:

Think of a below visual!

Stored sorted by column name - 'subcolumn_one' first, 'subcolumn_two' second,...



Row Key 1	<Subcolumn_one Subcolumn_two ...> 1	<Subcolumn_one Subcolumn_two ...> 2	...
	Column Value 1	Column Value 2	
⋮			

e.g., <state/city>: <CA|San Diego>, <CA|San Jose>, <NV|Las Vegas>, <NV|Reno>

2 CQL should not influence modeling

Best is to forget CQL during modeling exercise

- It's a relational-style interface on top of non-relational model.
- Don't think in relational or CQL-way while designing model!

Or, make sure to understand how CQL maps to physical structure.

3

Storing value in column name is perfectly ok

Leaving 'column value' empty (Valueless Column) is also ok

- 64KB is max for column key. Don't store long text fields, such as item descriptions!
- 2 GB is max for column value. But limit the size to only a few MBs as there is no streaming.

4

Use wide row for ordering, grouping and filtering

- Since column names are stored sorted, wide rows enable ordering of data and hence efficient filtering.
- Group data queried together in a wide row to read back efficiently, in one query.
- Wide rows are heavily used with composite columns to build custom indexes.

Example: Store time series event log data & retrieve them hourly.

Event Log

ddmmyyhh	timestamp1	timestamp2	...
	payload1	payload2	
⋮			

But, don't go too wide!

Because a row is never split across nodes

Traffic:

All of the traffic related to one row is handled by only one node/shard (by a single set of replicas, to be more precise).

Size:

Data for a single row must fit on disk within a single node in the cluster.

5

Choose proper row key – It's your “shard key”

Or you'll end up with hot spots, even with Random Partitioner

Example:

Bad row key: “ddmmyyhh”

Better row key: “ddmmyyhh | eventtype”

Event Log

ddmmyyhh eventtype	timestamp1	timestamp2	...
	payload1	payload2	
⋮			

6

Make sure column key and row key are unique

Otherwise, data could get accidentally overwritten

- No unique constraint enforcement, of course.
- CQL INSERT and UPDATE are semantically the same – UPSERT

Example:

- Timestamp alone as a column name can cause collisions
- Use TimeUUID to avoid collisions.

Event Log

ddmmyyhh eventtype	timeuuid1	timeuuid2	...
	payload1	payload2	
⋮			

7

Define correct comparator & validator

Don't just use the default `ByteType` comparator and validator

- Inappropriate comparator can store data(column names) in inappropriate order.
- Costly or impossible to do column slice/scans later.
- Can't change comparator once defined, without data migration.
- Validator helps to validate column value & row key. Can change later.



Validator - Data type for a *column value* or row key.

Comparator - Data type for a *column keys* & defines sort order of column keys.

Composite column supports features of super columns & more

Concerns with Super column:

- Sub-columns are not indexed. Reading one sub-column deserializes all sub-columns.
- Built-in secondary index does not work with sub-columns.
- Can not encode more than two layers of hierarchy.

9

Order of sub-columns in composite column matters

Order defines grouping

Example: Items sold by seller per state per city

123456	<state city>				
	CA San Diego	CA San Jose	NV Las Vegas	NV Reno	...
	20	10	100	200	
⋮					

<state | city>

Ordered by State first and then by City. Cities will be grouped by state physically on the disk.

<city | state>

The other way around, which you don't want.

Order affects your queries

Example: User activities

<timestamp activitytype>					
123456	00000001 Buy	00000002 Sell	00000003 Buy	00000004 Sell	...
	{..}	{..}	{..}	{..}	
⋮					

Efficient to query data for a given time range.

<activitytype timestamp>					
123456	Buy 00000001	Buy 00000003	Sell 00000002	Sell 00000004	...
	{..}	{..}	{..}	{..}	
⋮					

Efficient to query data for a given activity type and time range.

Also, efficient for only activity type. But not efficient for only time range.

It's like compound index!

Assume,

CF with composite column name as <subcolumn1 | subcolumn2 | subcolumn3>

Not all the sub-columns needs to be present. But, can't skip also.

Query on 'subcolumn1|subcolumn2' is fine. But not only for 'subcolumn2'.

Sub-columns passed after the sliced (scanned) sub-column are ignored.

Query on 'subcolumn1|*slice of* subcolumn2|subcolumn3' will ignore subcolumn3.

Correct order of sub-columns ultimately depends on your query patterns.

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

- Not easy to tune or introduce new query patterns later by simply creating indexes or building complex queries using join, group by, etc.
- Think how you can organize data into the nested sorted map to satisfy your query requirements of fast look-up/ordering/grouping/filtering/aggregation/etc.



- Identify the most frequent query patterns and isolate the less frequent.
- Identify which queries are sensitive to latency and which are not.

But don't de-normalize if you don't need to.

It's all about finding the right balance.

Normalization in Relational world:

Pros: less data duplication, fewer data modification anomalies, conceptually cleaner, easier to maintain, and so on.

Cons: queries may perform slowly if many tables are joined, etc.

The same holds true in Cassandra, but the cons are magnified

Next few slides illustrate practice 10 & 11 through the example.

Example 1

“Likes” relationship between User & Item

- Get user by user id
- Get item by item id
- Get all the items that a particular user has liked
- Get all the users who like a particular item

User

UserID	Name	Email
123	Jay	jp@ebay.com
456	John	jh@ebay.com
⋮		

User_Item_Like

ID	UserID <fk>	ItemID <fk>	Timestamp
1	123	111	00000001
2	123	222	00000002
3	456	111	00000003
⋮			

Item

ItemID	Title	Desc
111	iphone	It's a phone
222	ipad	It's a tablet
⋮		

Option 1: Exact replica of relational model

User

	Name	Email
123	Jay	jp@ebay.com
⋮		

Item

	Title	Desc
111	iphone	It's a phone
⋮		

User_Item_Like

	UserID	ItemID
1	123	111
⋮		

Note: timestamp column is dropped for simplicity.

There is no easy way to query:

- Items that a particular user has liked
- Users who liked a particular item

The worst way to model for this use case.

Option 2: Normalized entities with custom indexes

User

	Name	Email
123	Jay	jp@ebay.com
⋮		

Item

	Title	Desc
111	iphone	It's a phone
⋮		

User_By_Item

111	123	456	...
	null	null	
⋮			

Item_By_User

123	111	222	...
	null	null	
⋮			

- Normalized entities except user and item id mapping stored twice
- What if we want to get the titles in addition to item ids, and username in addition to user ids.
 - How many queries to get all usernames who liked a given item with like-count of 100?

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

Example 1

User

	Name	Email
123	Jay	jp@ebay.com
⋮		

Item

	Title	Desc
111	iphone	It's a phone
⋮		

User_By_Item

111	123	456	...
	Jay	John	
⋮			

Item_By_User

123	111	222	...
	iphone	ipad	
⋮			

- 'Title' and 'Username' are de-normalized now.
- What if we want:
 - Given a item id, get all item data along with user names who liked the item.
 - Given a user id, get all user data along with item titles liked by that user.

How many queries in the current model? Can it increase further if user becomes active or item becomes hot?

Option 4: Partially de-normalized entities

User

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

Item

111	ItemInfo Title	ItemInfo Desc	LikedBy 123	LikedBy 456	...
	iphone	It's a phone	Jay	John	

- Looks messy. Just to save one query?
- If User and Item are highly shared across domains, I would prefer option 3 at a constant cost of one additional query.

Best Option for this use case – Option 3

User

	Name	Email
123	Jay	jp@ebay.com
⋮		

Item

	Title	Desc
111	iphone	It's a phone
⋮		

User_By_Item

	<small><timeuuid userid></small>		
111	120101010000 123	120101030000 456	...
	Jay	John	
⋮			

Introduced timestamp (when user liked item) back as part of column name, to have chronological order.

Item_By_User

	<small><timeuuid userid></small>		
123	120101010000 111	120101020000 456	...
	iphone	ipad	
⋮			

Example 2

Semi-structured event log data

Collecting time series event log, and doing real-time aggregation & roll ups

Event Log, in relational world

ID	EventType	EventData	Timestamp
1	BID	blah blah	120101010000
2	BUY	blah blah	120101020000
3	SELL	blah blah	120101030000
⋮			

- Data is never updated so duplication/de-normalization won't hurt
- Entities & relationships may not matter much, like in earlier use case.

Example Cassandra model

Events

eventtype yymmddhh	timeuuid	...
	payload	
⋮		

Rollups-minute

eventtype hour	yymmdd hhmm00	yymmdd hhmm00	...
	count	count	
⋮			

Rollups-hour

eventtype day	yymmdd hh0000	yymmdd hh0000	...
	count	count	
⋮			

Many ways to model. The best way depends on your use case & access patterns.

Rollups-day

eventtype	yymmdd 000000	yymmdd 000000	...
	count	count	
⋮			

12 Keep read-heavy data separate from write-heavy

So can benefit from caching read-heavy data

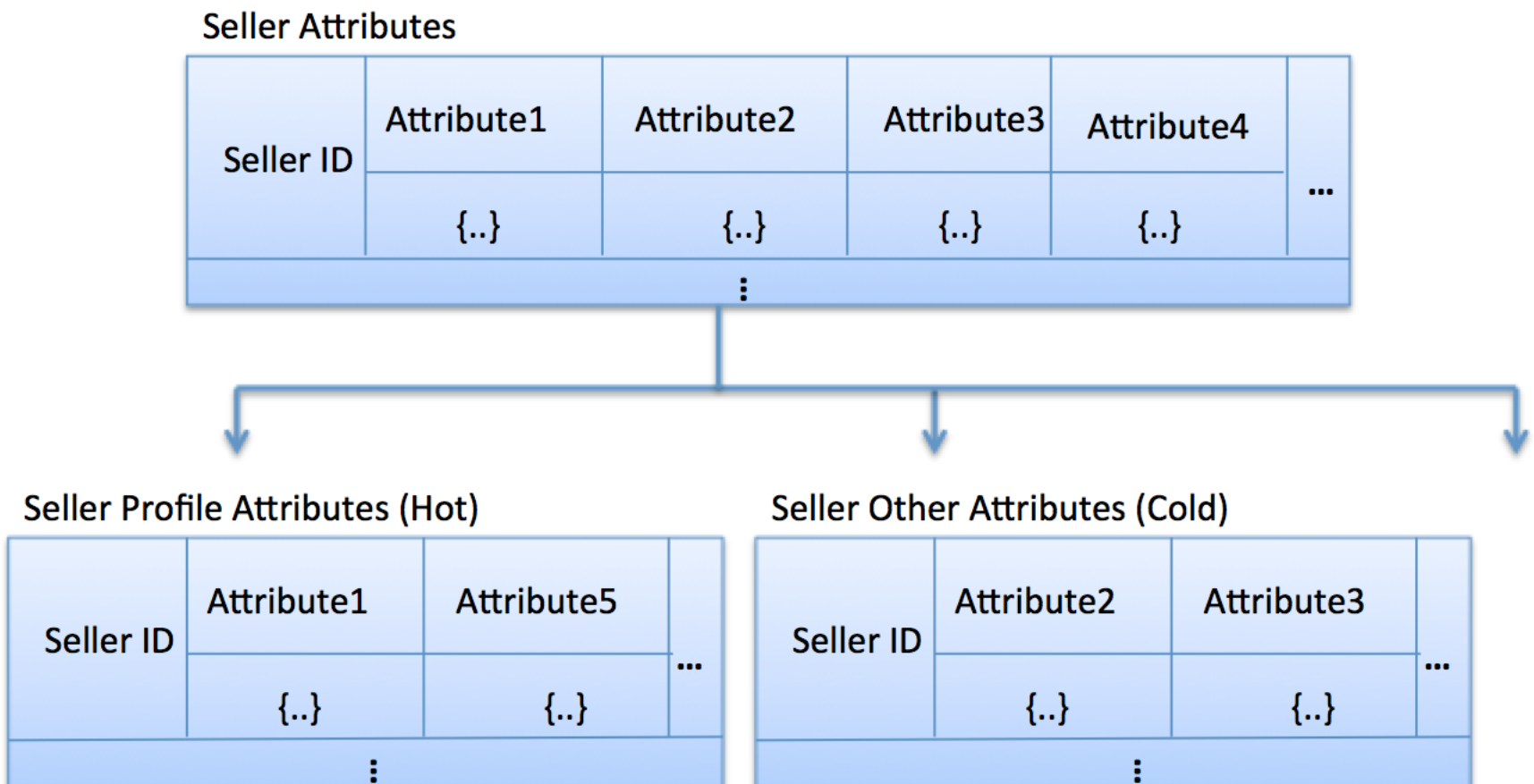
Irrespective of caching, it's always a good practice to keep read-heavy data separate from write-heavy since they scale differently.

Row cache caches the whole row. Be cautious before enabling for wide rows.

13 Isolate more frequent from the less frequent

Split hot & cold data in separate column families

Example: Seller Attribute data



But do use-case level splitting, first

- Compaction can run faster, in parallel.
- Better than built-in multi threaded compaction.
- Keep number of shards proportional to number of CPU cores.
- Don't have to match number of shards to number of nodes.
- Consider when lots of data in CF with high writes & reads.

It's painful, so don't pursue if you don't need to.

15

Design such that operations are idempotent

Unless, can live with inaccuracies or inaccuracies can be corrected eventually.

- Perceived write failure can result in successful write eventually.
- It's recommended to retry on write failure.

Retry on write failure can yield unexpected result if model isn't update idempotent.

ItemLikeCount

itemid1	"UserCounter"
	1000

Not Update idempotent

ItemLikeCount

itemid1	userid1	userid2	...
	username1	username2	

Update idempotent

But may not be efficient

- Counting users requires reading all user ids (million?) - Can't scale.
- Can we live with approximate count for this use case? - Yes.
- If needed, counter value can be corrected asynchronously by counting the user ids from update idempotent CF.

Cassandra write operations to regular CF is always idempotent.

Idempotency of Use case level operation(e.g. like/dislike item) is also useful, in case of eventual consistency.

Because it's stored repeatedly

- If column name holds actual data (in case dynamic CF), then that's great.
- But, in case of static CF, keep column name short!

For example:

Favor 'fname' over 'firstname', and 'lname' over 'lastname'.

17 Favor built-in composite type over manual

Because manual construction doesn't always work

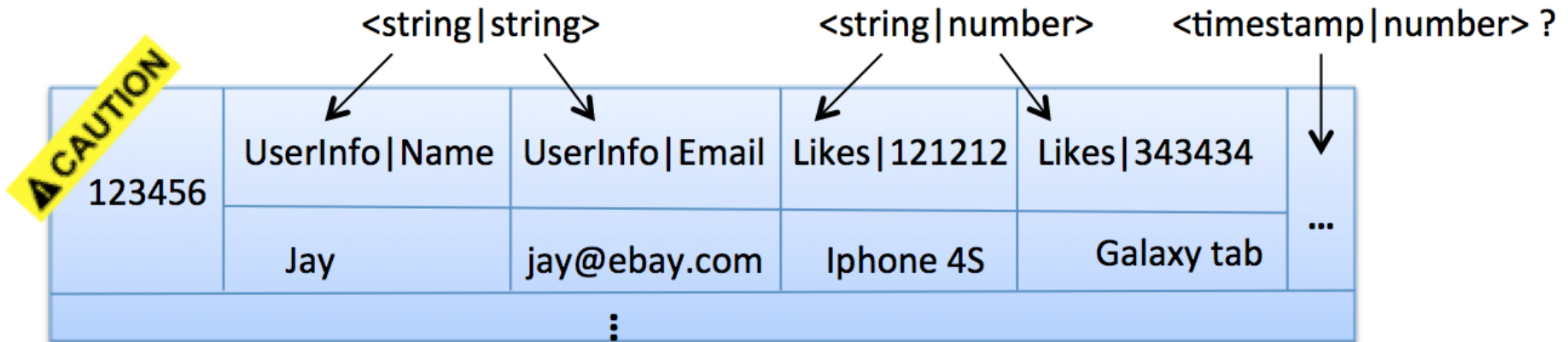
Avoid using string concatenation to create composite column names

- *Won't work as expected when sub-columns are of different types.*
<State|ZipCode|TimeUUID> won't be sorted in type aware fashion.
- *Can't reverse the sort order on components in the type*
<String | Integer> with the string ascending, and the integer descending.

18 Keep all data in a single CF of the same type

At least for row key & column key

In other words, favor static composite types over dynamic



CAUTION

123456	UserInfo Name	UserInfo Email	Likes 121212	Likes 343434	...
	Jay	jay@ebay.com	Iphone 4S	Galaxy tab	
⋮					

Annotations:

- `<string | string>` points to `UserInfo | Name`
- `<string | string>` points to `UserInfo | Email`
- `<string | number>` points to `Likes | 121212`
- `<string | number>` points to `Likes | 343434`
- `<timestamp | number> ?` points to `...`

Better to break into multiple column families.

Because it's not intended for this purpose

- Holds distributed counters meant for distributed counting.
- You can receive duplicate sequence numbers!
- ASK: Do I really need strictly sequential numbers?
- Prefer TimeUUID ([type-1 uuid](#)) as surrogate keys.

Think about query patterns & indexes from beginning

- Primary (Row key) Index
- Build-in secondary index
- Custom secondary index

- Built-in index, always used.
- Very efficient – Know which shard to hit!
- Good for equality predicate queries
(e.g., where rowkey = 'key1')
- Not useful for range queries
(e.g., rowkey > 'key1' and rowkey < 'key2')



Order Preserving Partitioner is almost never used. So, no range scans on row keys.

- From Cassandra 0.7 & later.
- Each index as a separate hidden Column Family per node.
- A query based on secondary index field will be sent to all the nodes, sequentially.



- It's an index on the column values, and not on the column keys.
- Column keys are always indexed & stored physically sorted.

Best used when

- Indexed Column is low cardinality, or read load is low.
- No scan (<, <=, =>, >) required. No 'order by' required.

Atomic index update	Yes
Is efficient?	No
In-equality predicate support (<, >, <=, =>)	No
Order by (return results in sorted order)	No
Maintenance overhead	No

- Column keys are stored physically sorted and indexed.
- This property of column keys is exploited along with composite columns to build custom secondary indexes.
- Custom secondary index is just yet another column family!

You actually build model as if you're building custom indexes!

Best used when

- Read load is high & indexed column is high cardinality.
- Range scan and/or 'order by' required, and has at least one equality predicate.

Atomic index update	No
Is efficient?	Yes
In-equality predicate support (<, >, <=, =>)	Yes, if at least one equality predicate (row key) is present
Order by (return results in sorted order)	Yes
Maintenance overhead	Yes

Note that Index updates can be hard.

Custom Indexing Examples

Let's create indexes for the below static 'Item' CF.

Item

itemid1	Title	Description	Seller Id	ListingDate
	title1	desc1	sellerid1	listingdate1

Select Title from Item where Seller = 'sellerid1'

Item_By_Seller_IX

sellerid1	itemid1	itemid2	itemid3	...
	-null-	-null-	-null-	

Better: Materialize 'title' in the index.

Item_By_Seller_IX

sellerid1	itemid1	itemid2	itemid3	...
	title1	title2	title3	

where Seller = 'sellerid1' order by Price

20.3

Item_By_Seller_IX

sellerid1	price1 itemid1	price2 itemid2	price3 itemid3	...
	title1	title2	title3	

What if seller changes price of the item? How to update index?

Delete <old_price | itemid> columns & insert <new_price | itemid>.

But,

- How to get old price in order to delete? Read before write?
- Any race condition? What if Consistency Level is eventual?
- Repair on read?

where Seller='sellerid1' and ListingDate > 10-11-2011
and ListingDate < 11-12-2011 Order by Price

20.3

Item_By_Seller_IX

sellerid1	listingdate1 price1 itemid1	listingdate2 price2 itemid2	listingdate3 price3 itemid3	...
	null	null	null	

Won't work 😊

Data won't be ordered by 'Price' across dates.

What to do then?

Key Takeaways



- **Don't think of a relational table**
 - Think of a nested sorted map, instead.
- **Model column families around query patterns**
 - But start with entities & relationships, if you can.
- **De-normalize and duplicate for read performance**
 - But don't de-normalize if you don't need to.
- **Many ways to model data in Cassandra**
 - The best way depends on your use case and query patterns.
- **Indexing is not an afterthought, anymore**
 - Think about query patterns & indexes upfront.
- **Think of physical storage structure**
 - Keep data accessed together, together on the disk.

There is more..



eBay Tech Blog:

Best Practices Part 1:

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

Best Practices Part 2:

<http://www.ebaytechblog.com/2012/08/14/cassandra-data-modeling-best-practices-part-2/>

Cassandra at eBay:

<http://www.slideshare.net/jaykumarpatel/cassandra-at-ebay-13920376>

Meet our operations:



Feng Qu

Principle DBA @ eBay

Cassandra Prod. operations expert



Baba Krishnankutty

Staff DBA @ eBay

Cassandra QA operations expert

Are you excited? Come Join Us!



Thank You

 **@pateljay3001**