Cassandra Fundamentals

Module 5

Cassandra Data Modeling

Module plan

- Query-driven data modeling
- Data modeling best practices
- Table and key design
- Secondary indexes

eBay's Best Practices

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

- Don't think of a relational table
- CQL should not influence modeling
- Storing value in a column name is OK
- Model column families around query patterns
- Denormalize and duplicate for read performance

Don’t Think Of A Relational Table

- Instead, think of a nested sorted map

Map<RowKey, SortedMap<ColumnKey, ColumnValue>>

  - External map is unsorted unless Byte Preserving Partinioner is user
- Physical model is more similar to map than to table
- Maps give efficient key lookup
- Unbounded number of columns
- Column key can itself hold a value

CQL Should Not Influence Modeling

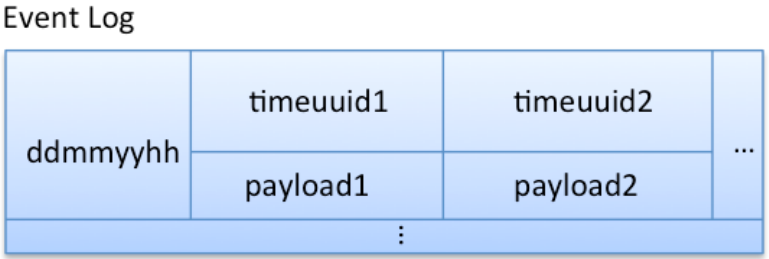
- It’s a relational interface to a non-relational data

Storing Value In A Column Name Is OK

- Leaving column value empty is also OK
- 64Kb max for a column key
  - Don’t store long text fields
- 2Gb max for a column value

Use Wide Rows For Ordering, Grouping And Filtering

- Since columns are stored sorted by name, 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

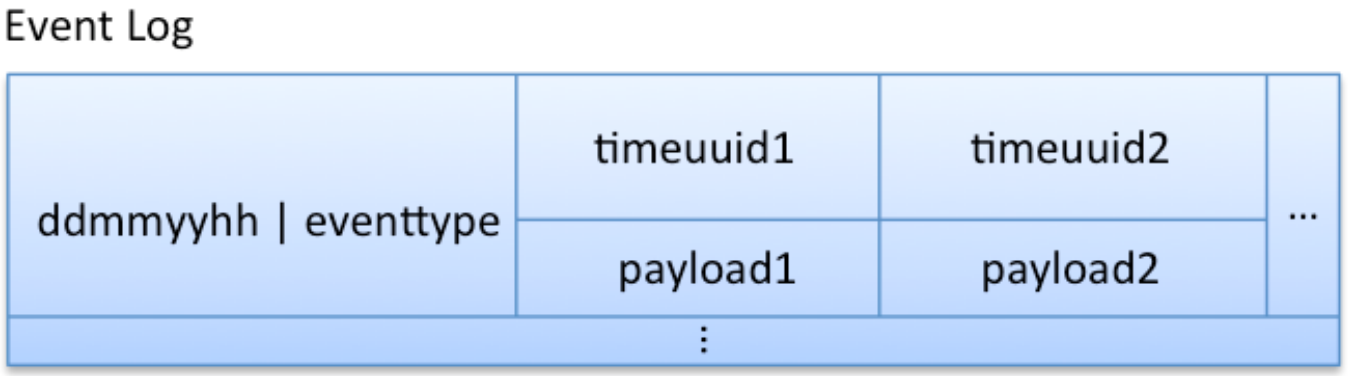


But Don’t Go Too Wide

- Row is never splits across nodes
- All of the traffic related to one row will be served by single replica set
- Data for one row must fit on one disk in a single node

Choose Proper Row Key – It's Your Sharding Key

- Or you'll end up with hot spots, even with hash-based partitioner
- Bad row key: "ddmmyyhh"
- Better row key: "ddmmyyhh|eventtype"

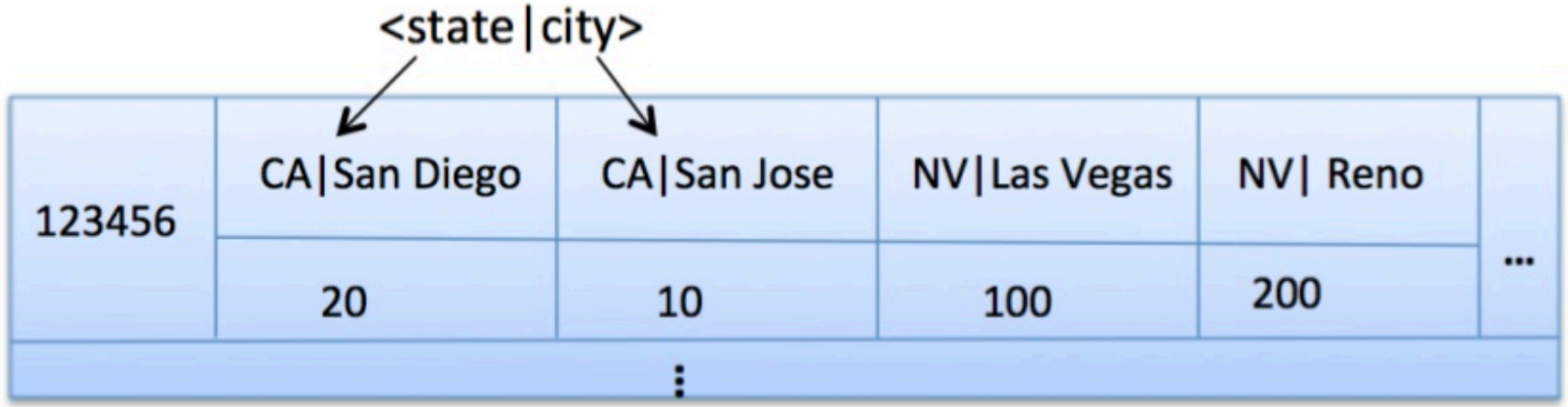


Make Sure Column Key And Row Key Are Unique

- Otherwise, data could get accidentally overwritten
- There’s no unique constraint enforcement
- CQL INSERT and UPDATE are the same UPSERT
- Timestamp as a column name can cause collisions
- Use TimeUUID

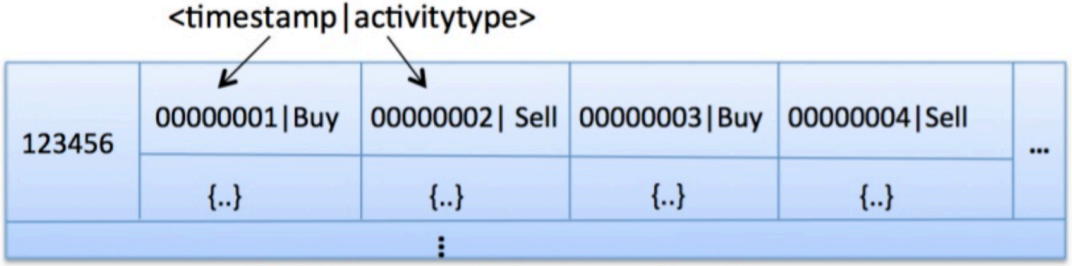
Order Of Sub-columns In Composite Column Matters

- Order defines grouping
- State|City – cities will be grouped by states
- City|State – states will be grouped by cities

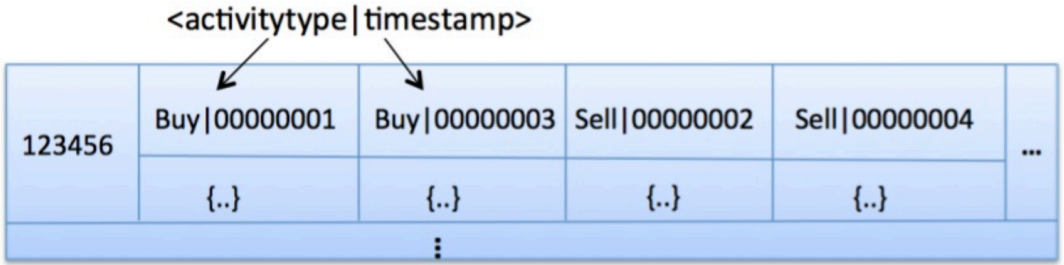


Order Affects Your Queries

- Efficient to query data for a given time range



- Efficient to query data for a given activity type and time range; not efficient for time range



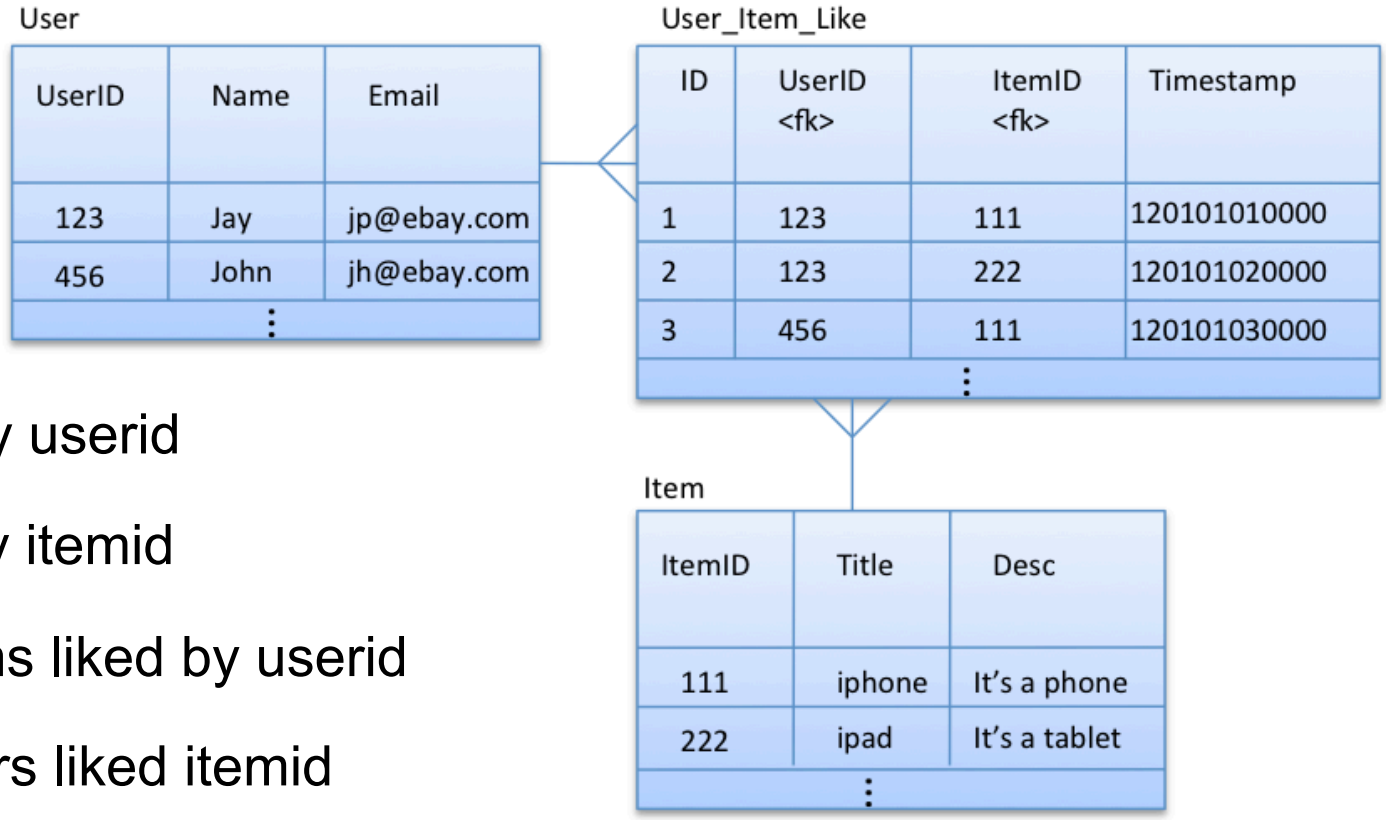
Column Family With Composite Column Is Like Compound Index

- Column name: <subcolumn1|subcolumn2|subcolumn3>
- Not all the subcolumns need to present. Only the prefix.
  - Query on subcolumn1|subcolumn2, but not on subcolumn2.
- Subcolumns passed after the sliced (scanned) subcolumn are ignored.
  - Query on subcolumn1|slice of subcolumn2|subcolumn3 will ignore subcolumn3

Model Column Families Around Query Patterns

- Start your design with ER (if you can)
- Not easy to introduce new query patterns later
- Think how you can organize data into the nested sorted map to satisfy fast lookup/ordering/grouping/filtering
- Identify most frequent queries
- Identify which queries are sensitive to latency and which are not

Denormalize And Duplicate For Read Performance: Example 1



- Get user by userid
- Get item by itemid
- Get all items liked by userid
- Get all users liked itemid

Option 1: Exact copy of relational model

User

	Name	Email
123	Jay	jp@ebay.com
⋮		

Item

	Title	Desc
111	iphone	It's a phone
⋮		

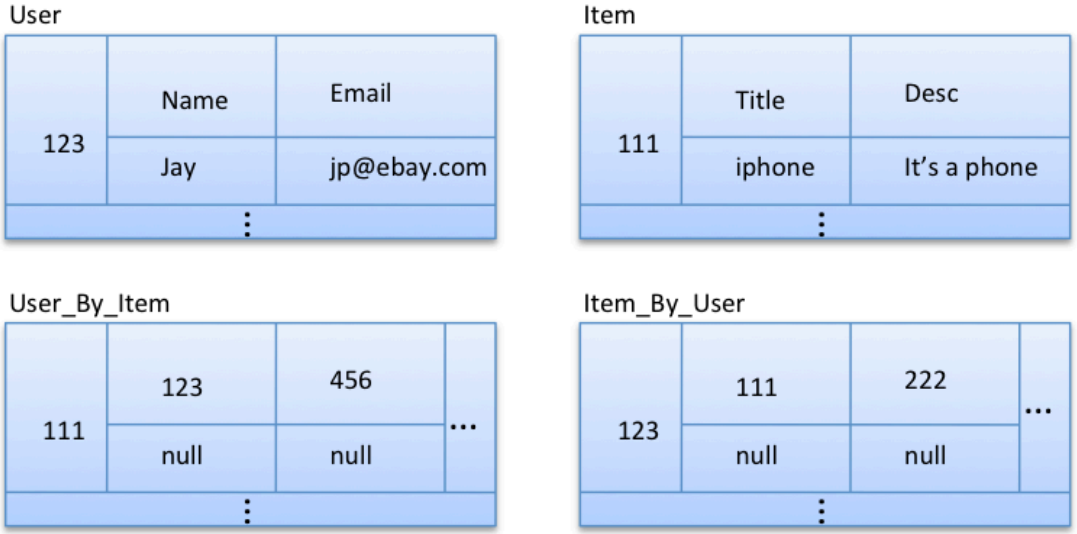
User_Item_Like

	UserID	ItemID
1	123	111
⋮		

No easy way to:

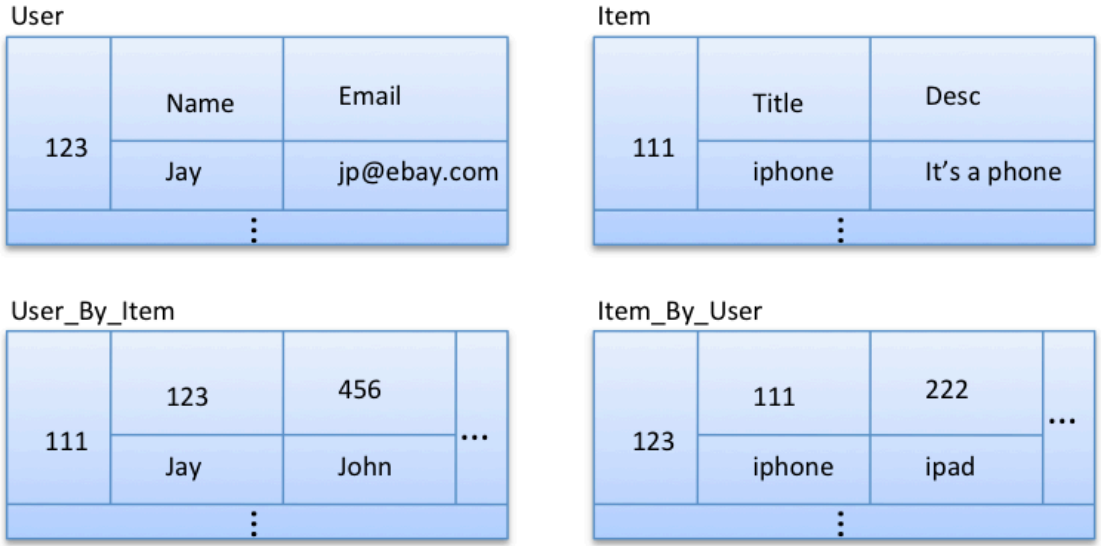
- Get all items liked by userid
- Get all users liked itemid

Option 2: Normalized entities with custom indexes



- Entities stored twise
- What if we want to get not only userids, but also usernames?

Option 3: Normalized entities with denormalization into custom indexes



- Title and username are denormalized
- What if we want to get not only userids and usernames, but all user data?

Option 4: Partially denormalized entities

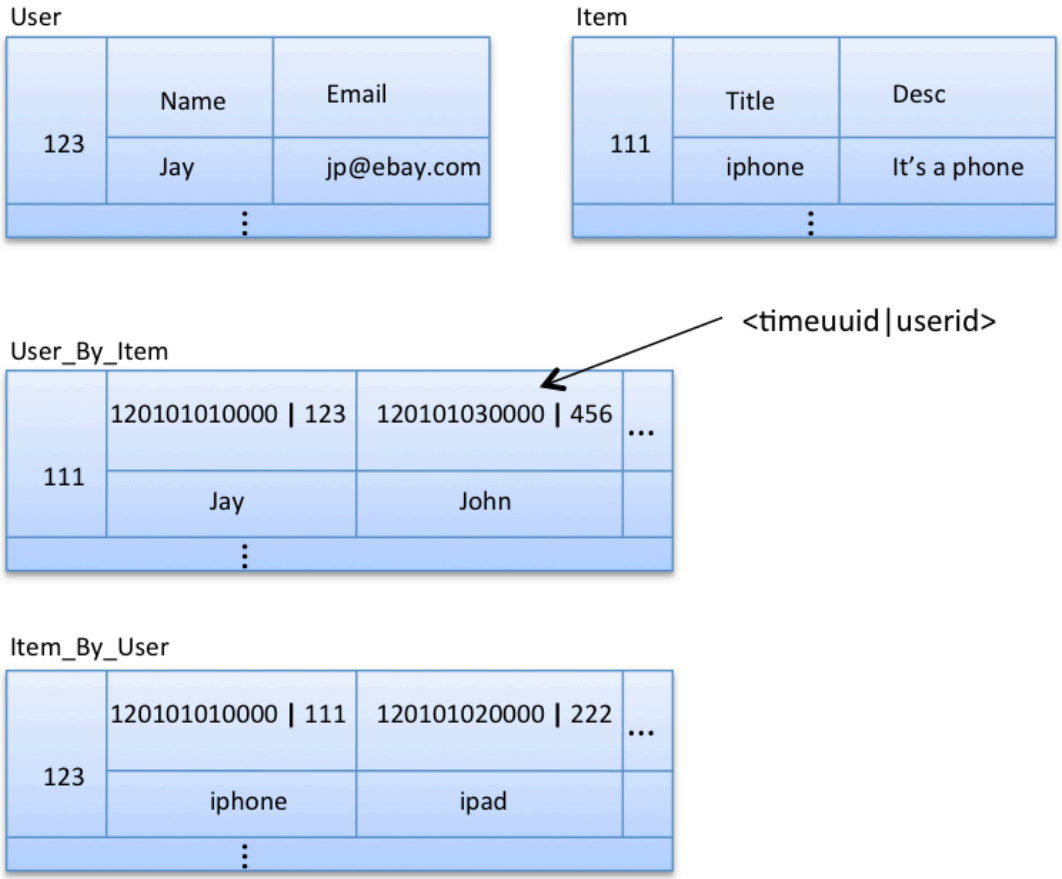
User

	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	
⋮					

Option 3bis: entities with timestamped custom indexes



Denormalize And Duplicate For Read Performance: Example 2

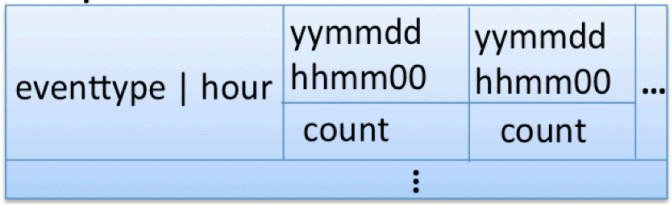
- Event log in relational world

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

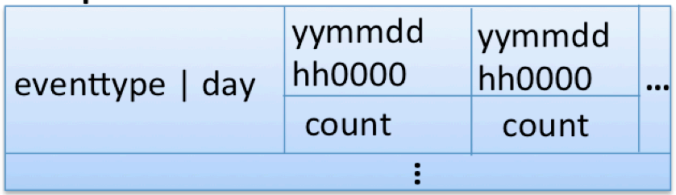
Example 2: C* model

Events

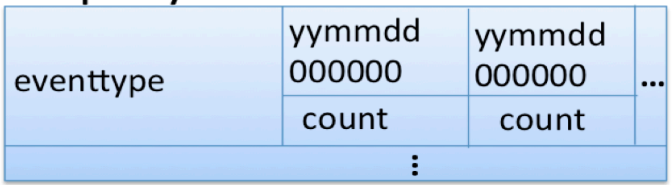
Rollups-minute



Rollups-hour



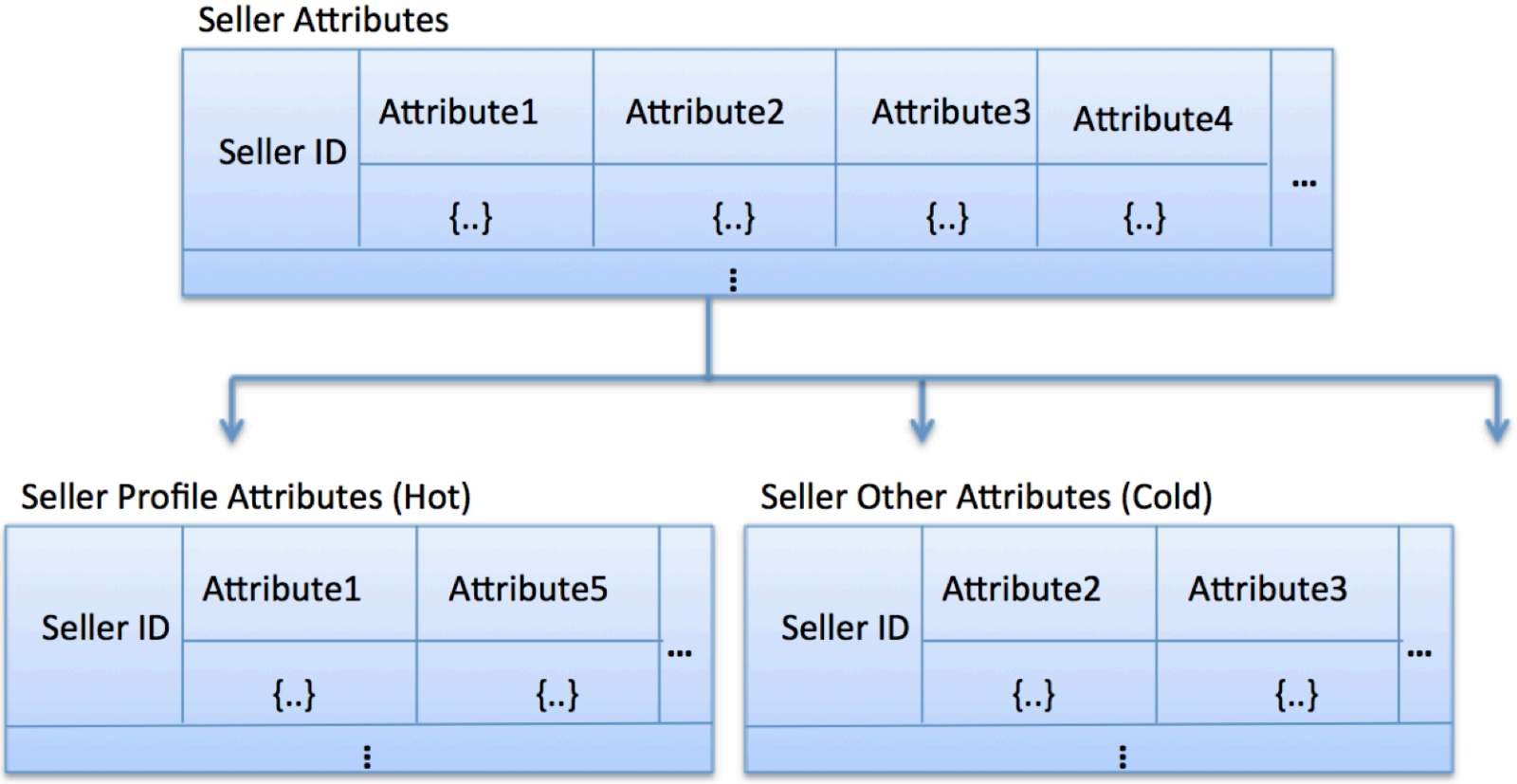
Rollups-day



Keep Read-heavy Data Separate From Write-heavy

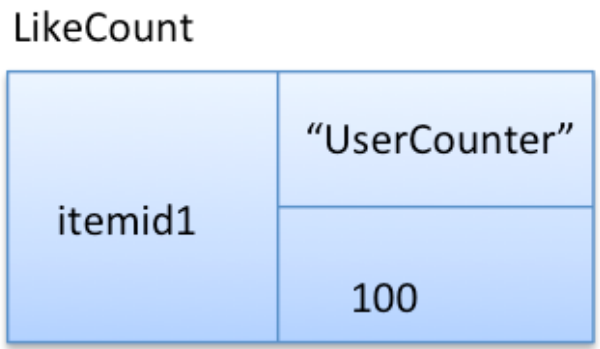
- This way you can benefit from Cassandra's off-heap row cache
- Row cache caches entire row

Split Hot and Cold Data Between Column Families

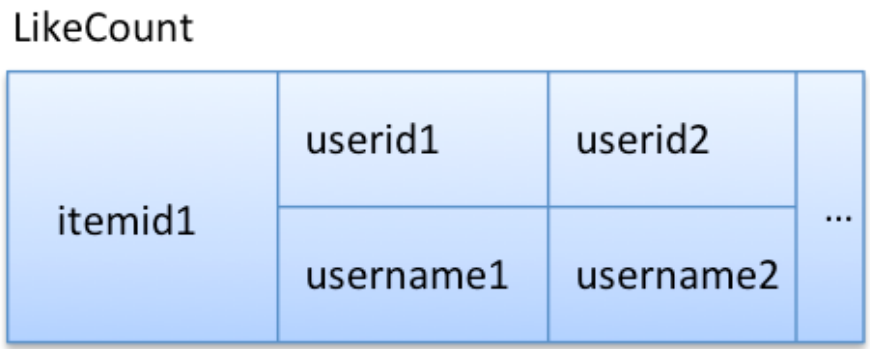


Design Idempotent Operations

- Or make sure you can live with inaccuracies or inaccuracies can be corrected eventually.



Not Update idempotent



Update idempotent

Keep Column Names Short

- Because it’s stored repeatedly
- Exception: you use column names to store real data

Favor Built-In Composite Types Over Manual

- Avoid using string concatenation to create composite column names
- Won’t work as expected when sub-columns are of different types
  - <State|ZipCode|TimeUUID>
- Can’t reverse sort order
  - <String|Integer> with string ascending and integer descending

Don’t Use the Counter Column for Surrogate Keys

- It’s not intended for this purpose
- Distributed counters meant for distributed computing
- Duplicates are possible
- Prefer TimeUUID for surrogate keys

Indexing Is Not an Afterthought

- Think about query patterns and indexes from beginning\
- Primary (Row-key)
- Built-in secondary
- Custom secondary

Primary Index

- Built-in, always used
- Not an index, rather a sharding scheme
- Good for point queries (key=“key”)
- Not useful for range queries

Built-in Secondary Index

- Index is per node only (no distribution)
- Index on column _values_ (not keys)
  - Column keys are already sorted and indexed
- In essence, separate hidden column family (per node)
- Automatically and atomically updated (on base table change)
- Query based on secondary index field will be sent to all nodes, sequentially

Built-in Secondary Index Best Used When

- Low cardinality or read load is low
- No scan (<,>) required, no ordering required
- Atomic with base table update
- No manual maintenance
- Not efficient (all nodes involved)

Custom Secondary Index

- Column keys are stored sorted and indexed
- So we can exploit it with composite columns
- Custom secondary index is just another column family, manually updated (in batch with base table update)

- You actually build model as you're building custom indexes

Custom Secondary Indexes Best Used When

- Read load is high and indexed column is high cardinality
- Range scan and/or ordering is required, having at least one equality predicate
- Has manual maintenance overhead
- Better to be updated in atomic batch

Thank you!

Questions?