Big Data
Data Modeling

BY TED MALASKA

High Level Overview

- Day 1
  - Defining a basic foundation of Data Modeling
  - Fundamentals of Big Data Systems
  - Digging into Big Data Systems
  - Define the Use Case
  - Starting with Relational

High Level Overview

- Day 2
  - Try de-normalization
  - Try Nested Tables
  - Try Bucketed and Sorted
  - Try NoSQL
  - Try Time Series Metric
  - Try Time Series Entity
  - Try Lucene Indexing
  - Try Graph
  - What are we seeing?
  - Final Thoughts

Section 1 - Defining a basic foundation of Data Modeling

- Thinking in objects/tables
- Access Patterns
- Mutation Patterns
- Use cases
- Transactions

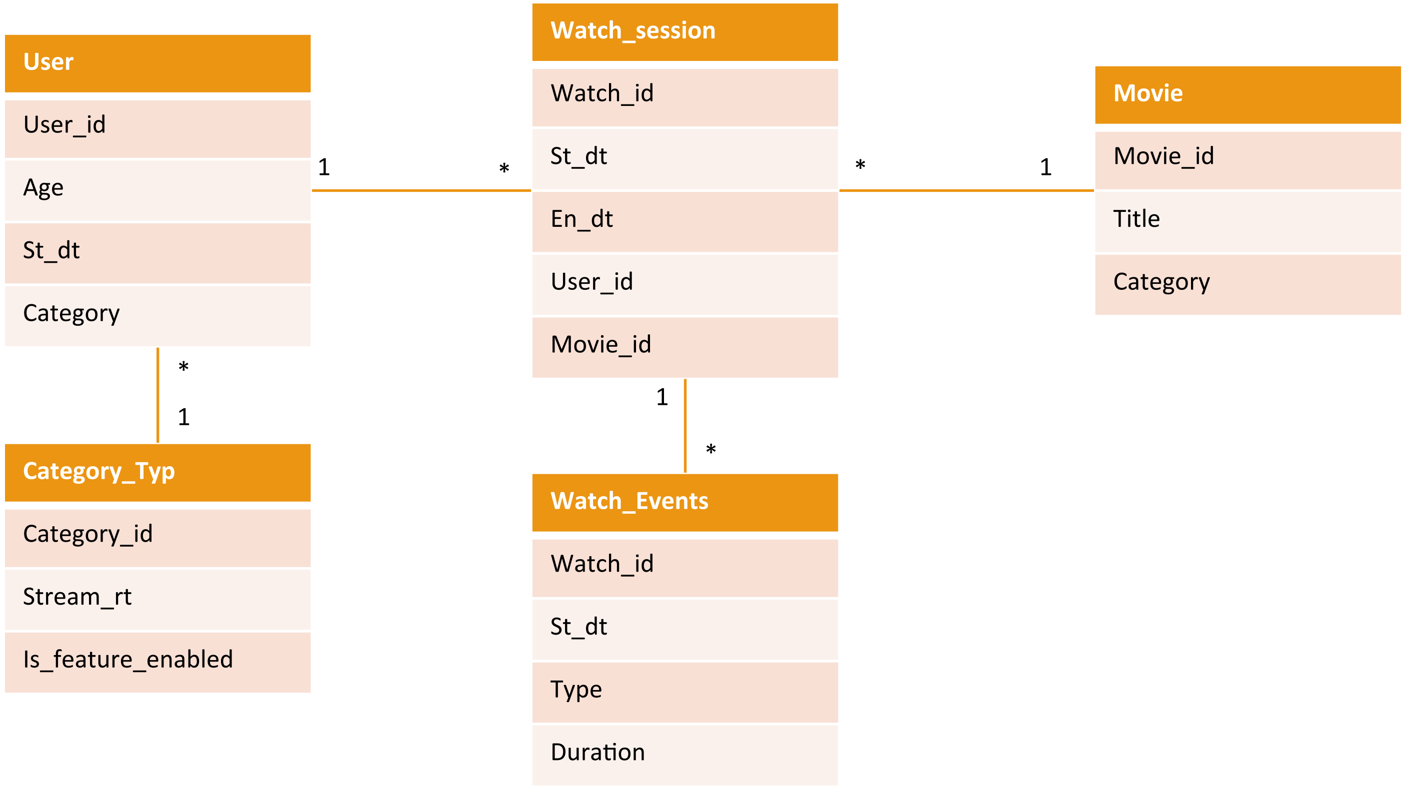
Thinking about Object/Tables

1. Lets start off easy
    1. Use Case: We are a Netflix type company and we have a log of movies watch that looks something like this.

User ID	Age	Account Start Date	Category Of User	Movie Watched	Movie Category	Start Time	Events List
Bob	42	12/12/2012	Basic	Die Hard	Action	5/4/2016 12:00	Play 0, pause at 15, FF at 40 to 55, E at 90
Kat	31	12/12/2012	Platum	Beauty and the Beast	Family	5/4/2016 12:00	Play 0, pause at 15, FF at 40 to 55, E at 90

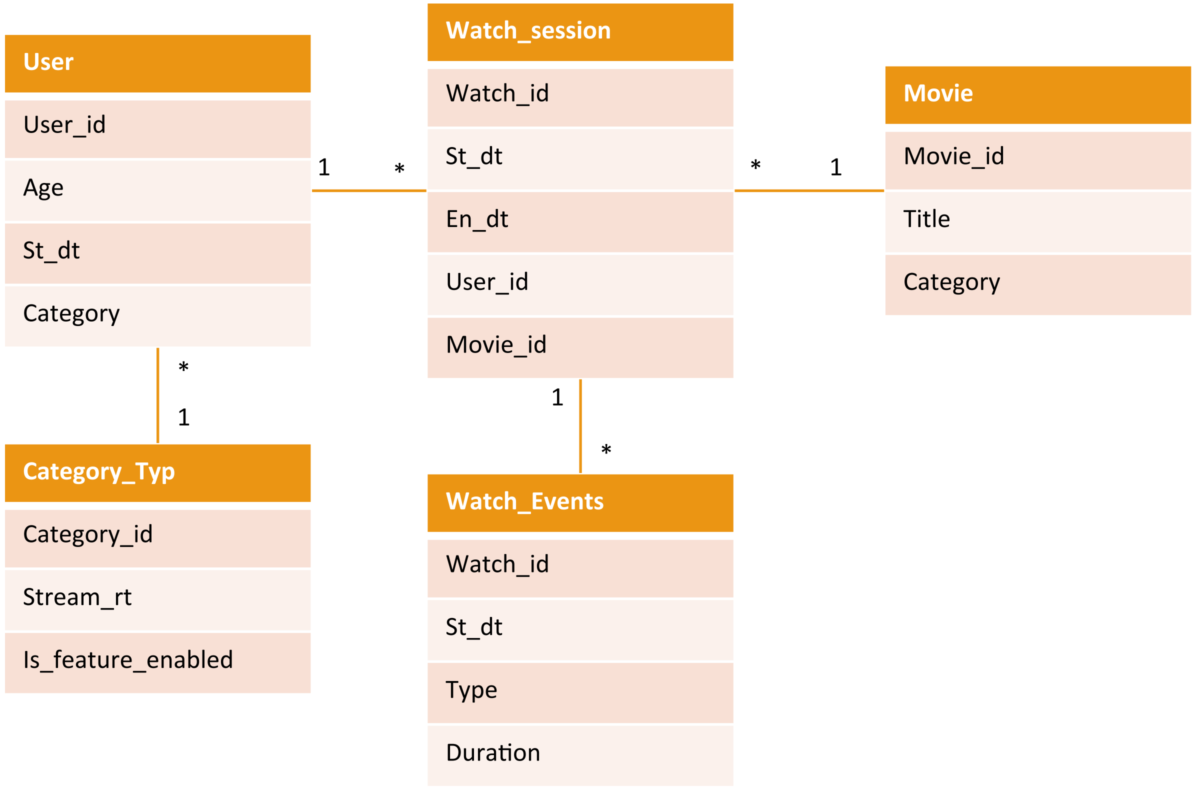
Thinking about Object/Tables

1. To make this into object we need to do some separation



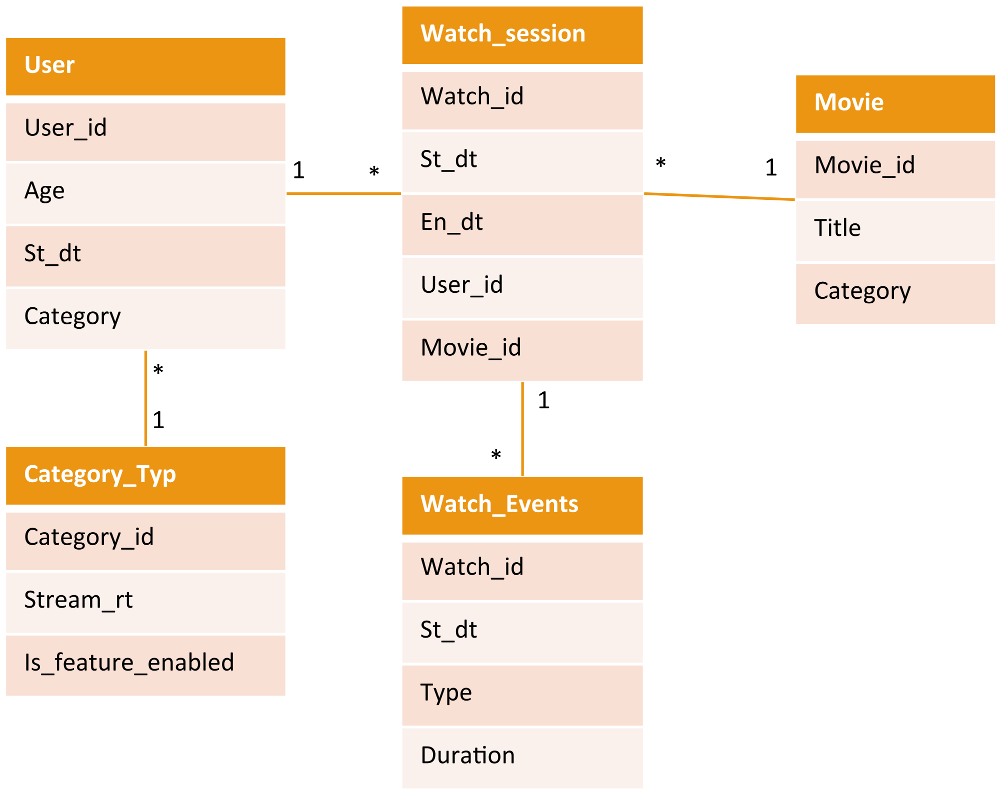
Thinking about Object/Tables

1. Why was this successful
   1. We could CRUD object records in isolation
   2. We could do joins to get any answer we needed
   3. We could mutate table definitions in isolation
   4. We could apply transaction across tables
   5. We have indexing and fourn keys



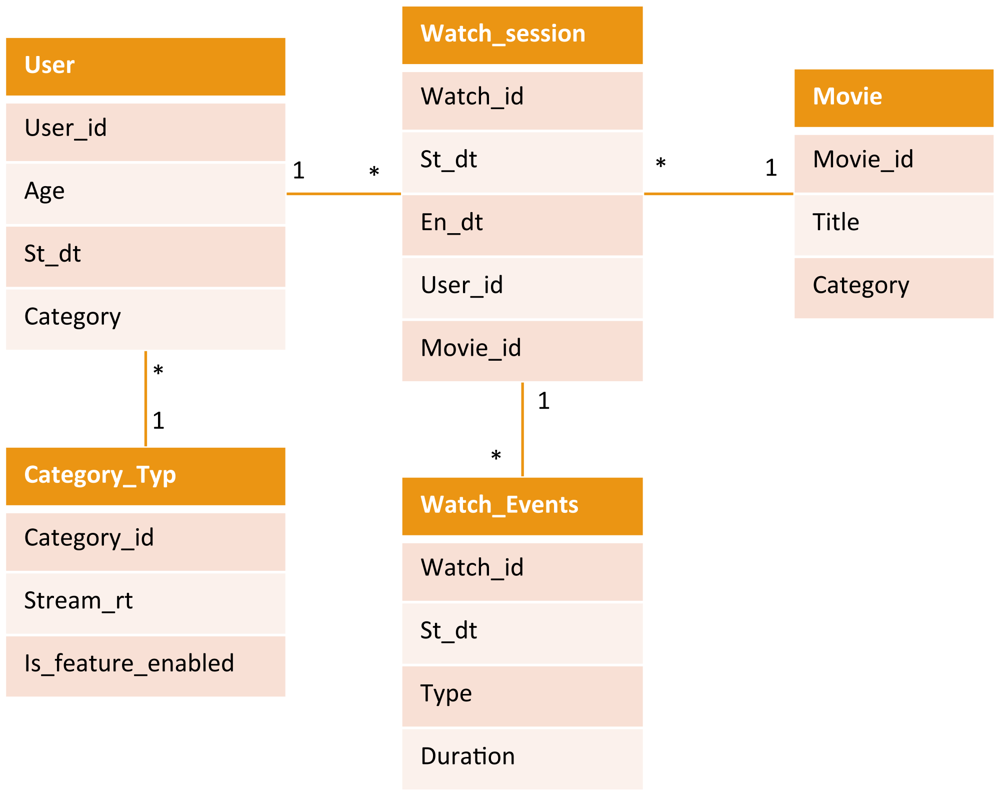
Access Patterns

1. Get movies watched and when
2. Get distinct movies watched
3. Get number of movies watched by age
4. Get the most skipped over seens???
5. Get most skipped over seen on first time watched



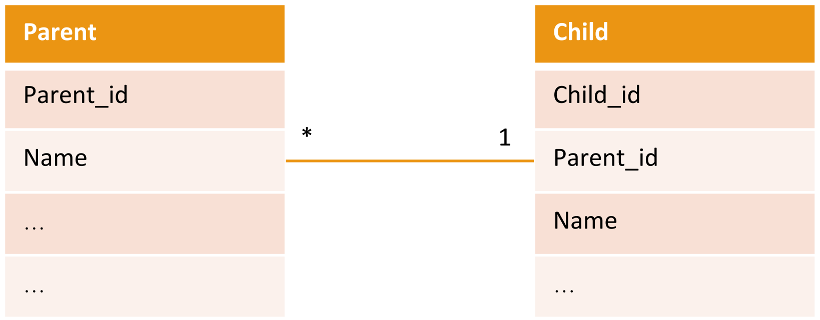
Transitions

1. ???



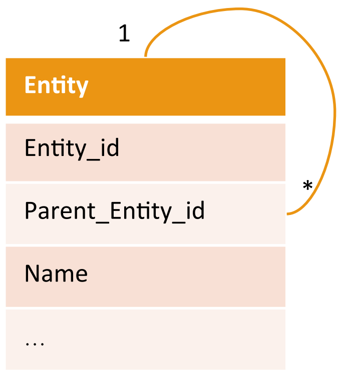
Types of Model Patterns

- Parent Child



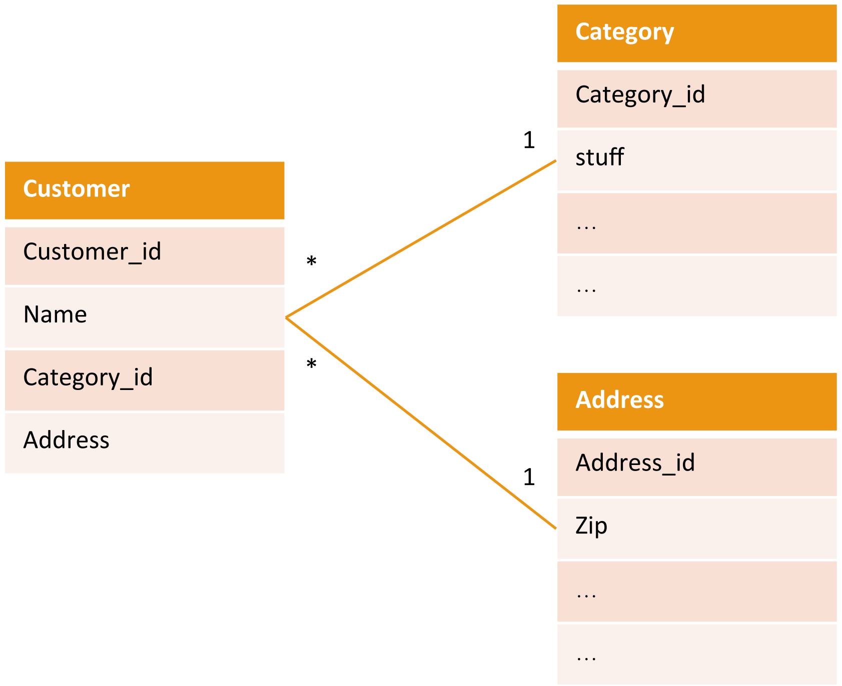
Types of Model Patterns

- Single Parent Child



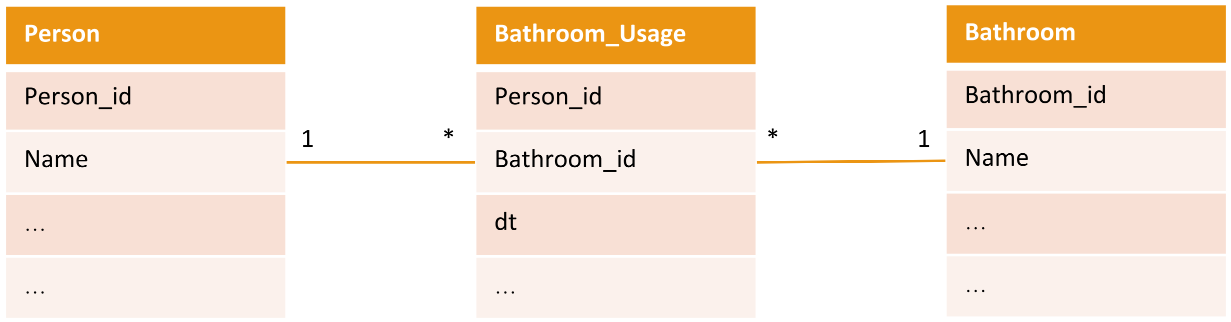
Types of Model Patterns

- Dimensional Table / Star Schema



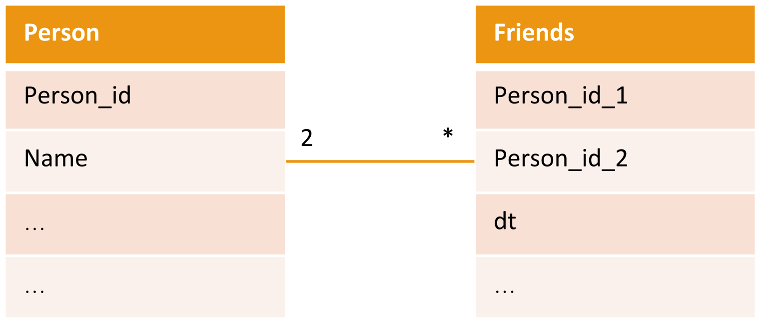
Types of Model Patterns

- Many to Many



Types of Model Patterns

- Single Many to Many



Types of Model Patterns

- Fact Table Types
  - Transaction
  - Snapshot
  - Accumulating Snapshot

Types of Model Patterns

- Transaction Fact Table

Key	Time	Value
A	2	102
A	1	101
B	3	103
B	2	102
B	1	101
C	2	102
C	1	101
D	1	101

Types of Model Patterns

- Snapshot Fact Table

Key	Time	Value
A	2	102
B	3	103
C	2	102
D	1	101

Types of Model Patterns

- Accumulating Fact Table

Key	Time	Value
A	2	203
B	3	306
C	2	203
D	1	101

Defining a basic foundation of Data Modeling Summery

- Somewhere after 2010 things changed
- Single Node database moved to expensive appliances
- The domain for data increase in volume and in speed
- The internet happened

Defining a basic foundation of Data Modeling Summery

- The cost of scans grow
- The cost of joins grow

Summary & Q/A

Foundations of Big Data

- Software vs hardware
- Partition or Sharding
- Replication and Failure
- MapReduce
- Mutability and Immutability
- Locking
- CAP theorem
- Masters and Headless
- Transactions

Big venders and their black boxes

- It was a great time to be a vender
  - Open source was untrusted and unproven but for some out liars
  - To get speed you needed special hardware
  - It was the area of SANS, GreenPlums, Netezzas, NetApp, EMC, Oracle, and IMB
  - Then came alone google

What did Google do

- They built large distributed systems that where expected to fail
  - Running on cheap hardware
  - Relying on software to handle recovery
  - Then they did the unthinkable. They gave the ideas away for free.
    - Google File System & MapReduce Papers where released

Then it just continues

- The down was open
  - More white papers
  - More open source projects
- Projects like
  - Hadoop
  - MapReduce
  - Hive
  - Cassandra
  - HBase
  - Kudu
  - Kafka
  - MongoDB
  - Spark
  - Impala
  - Presto
  - OpenTSDB/KairosDB

Why so many projects

- As data gets big data gets heavy
- Solutions get more specialized to given use cases

What do they share (Failure)

- Failure is a given
- Replication is built in with Software

What do they share (Partitioning)

- Partitioning is a given
- Many types of partitioning
  - Hash Mod
  - Round Robin
  - Lazy Round Robin
  - Range
  - Centrally Managed

What do they share (Partitioning)

- Partitioning is a given
- Many types of partitioning
  - Hash Mod
    - Skew is possible so pick your key wisely
    - May be a victim Broadcast overload
  - Round Robin
    - No Skew
    - None deterministic access pattern
    - May be a victim Broadcast overload
  - Lazy Round Robin
    - No Skew
    - None deterministic access pattern
  - Range
    - Skew is very possible
    - May be a victim Broadcast overload
  - Centrally Managed
    - No Skew
    - Single point of contention

What do they share (Mutability)

- Mutability is Bad
  - Require seeks
  - Require fixed length storage
  - Require additional locking
- Immutability is Good
  - Straight writes
  - Less to no locking
- Mutability through Immutability
  - Think about logs
  - Think about a Relation Time Series Table or WAL

What do they share (Mutability)

Key	Time	Value
A	1	101
B	1	101
C	1	101
D	1	101
E	1	101
F	1	101
G	1	101

Key	Time	Value
A	2	102
D	2	102
F	2	102
F	3	103
H	3	103

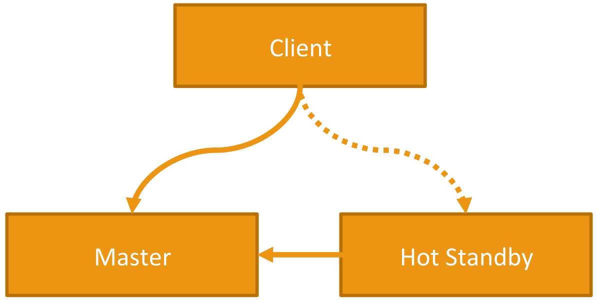
Key	Time	Value
A	2	102
B	1	101
C	1	101
D	2	102
E	1	101
F	3	103
G	1	101
H	3	103

What do they share (Headless or Head)

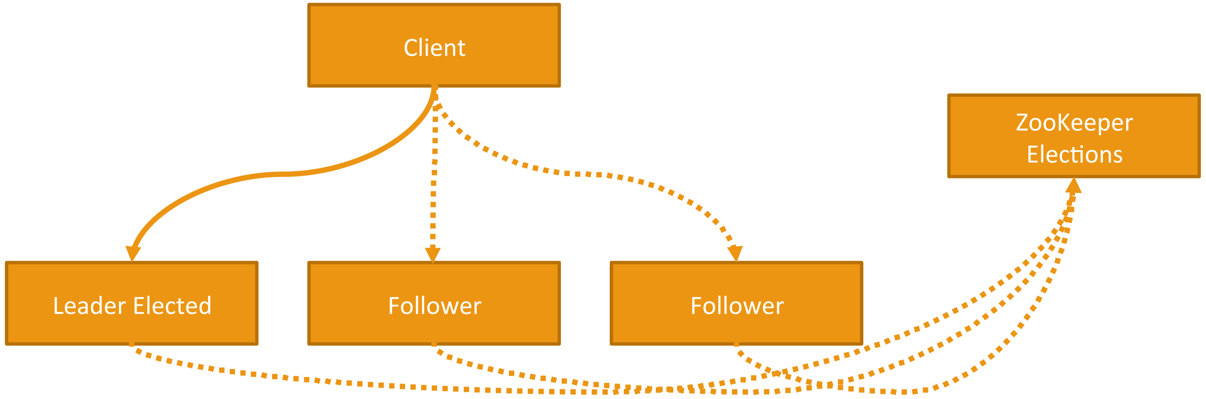
- Heads or Leaders
  - Easier to implement things that demand consistence
  - There is a switching cost to head failures
- Headless
  - No switching cost
  - Normally like to an eventually consistent model

What do they share (Head Types)

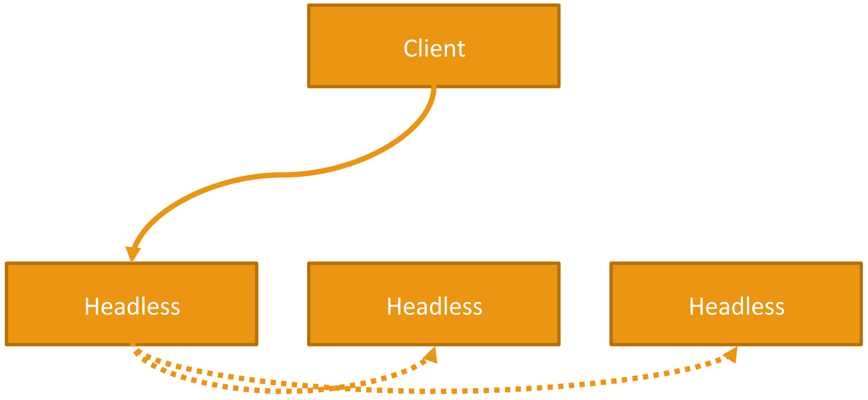
- Master and a HA

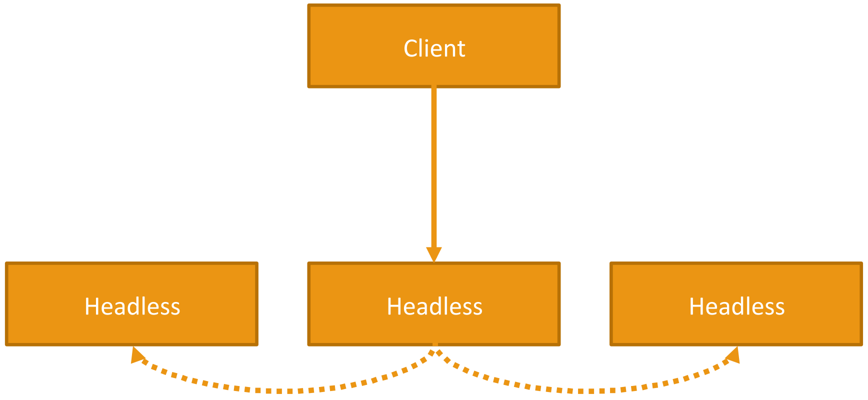


- Leader Election

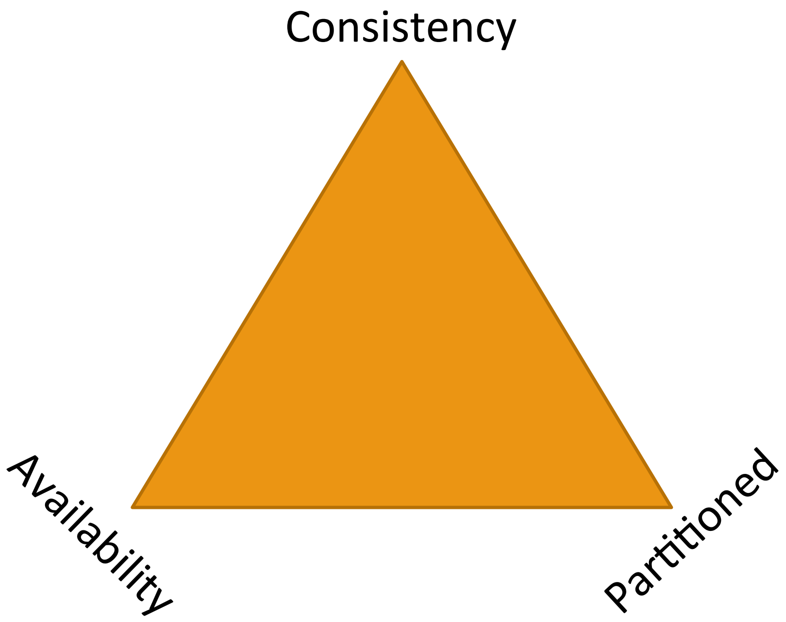


What do they share (Headless)

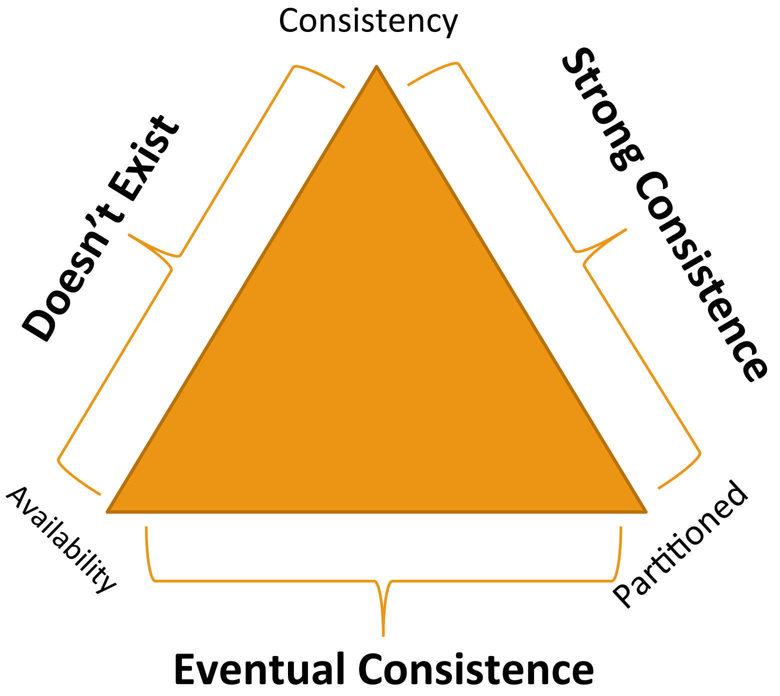




What do they share (CAP theorem)

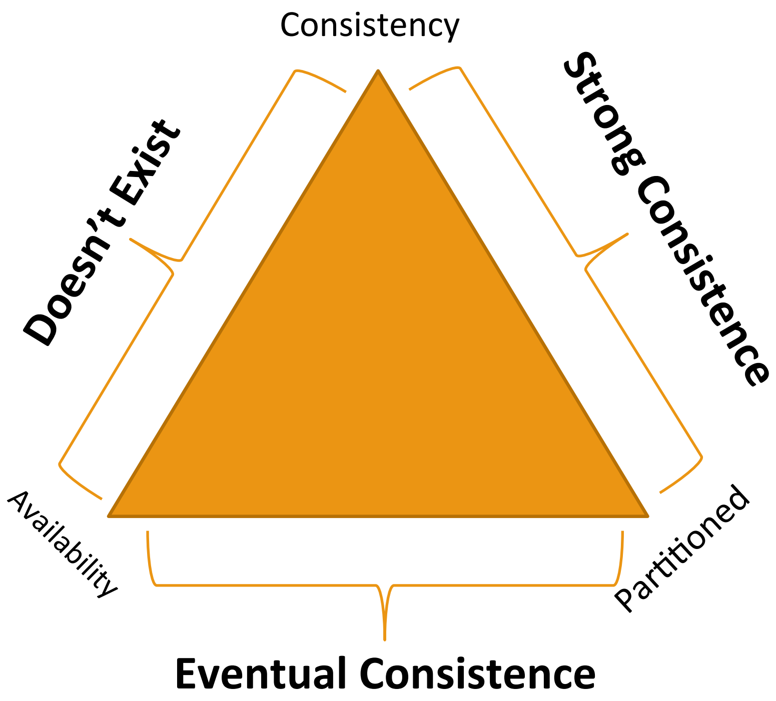


What do they share (CAP theorem)



What do they share (CAP theorem)

- Cheating the CAP Theorem
  - Cassandra is a good model
    - Where they expand the definition of failure with variable consistence
  - CAP still holds but …



What do they share (Transactions)

- It is hard to make a single record change Transactions would be harder and costly
  - Locks suck
- Most part Transaction are not in Big Data system
  - Other then the lowest unit of mutation
    - Files
    - Records
- Different design patterns in Big Data to replace traditional multi row transaction

Why we are going to Dig Deep

- Is this class about Data Modeling or Storage Systems and Execution Systems
- Understand is required for Data Modeling
  - Before RDBMS system where mostly alike
  - There where difference but those differences where incremental not evolutionary

About to Dive in

- What aspects do we need to be thinking about as we look into these different solutions
  - Storage Structure
  - Mutation Patterns
  - Access Patterns
  - Cost/Storage of storage
  - Cost of access

Summary & Q/A

Hadoop Distributed File System

- Component break down
- System fundamentals
- File Formats
- Mutation Patterns
- Access Patterns
- Cost considerations

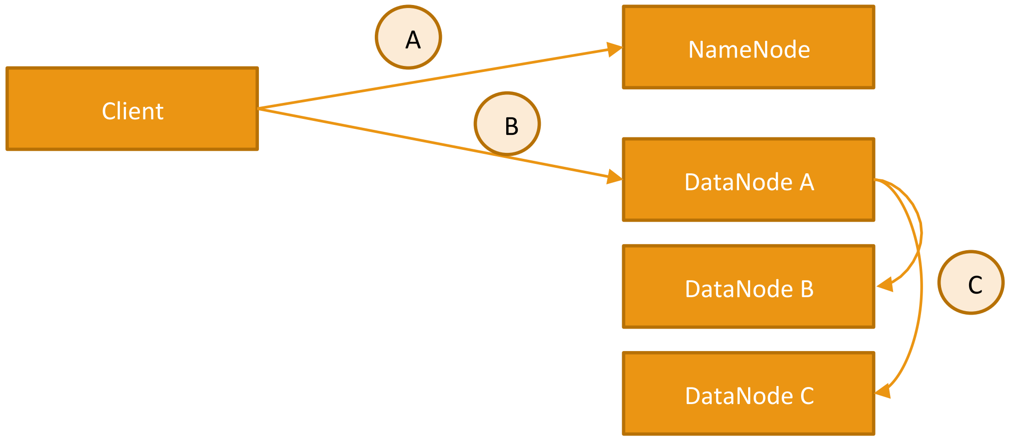
Basic of GFS => HDFS

- NameNode
  - Metadata of all the files/blocks
  - Which data node they are assigned too
  - Replication management
- Data Nodes
  - Metadata for each block location on disk

Basic of GFS => HDFS

Write Path

A. Ask Name Node for Location to Write
B. Write to DataNode with NN Instructions
C. DataNode does replication
D. Confirms file is persisted to client



Basic of GFS => HDFS

- File are immutable
- File can be of any type
- Files are block up into Blocks (128MB -> 1GB)
  - Metadata cost is at the Block not the data size
- File may be splittable or may not be when reading

Splitting a File

- Able to Split
  - Text
  - Seq
  - Arvo
  - Parquet
  - ORC
- Not able to Split
  - GZip file
  - Variable row length copybook files

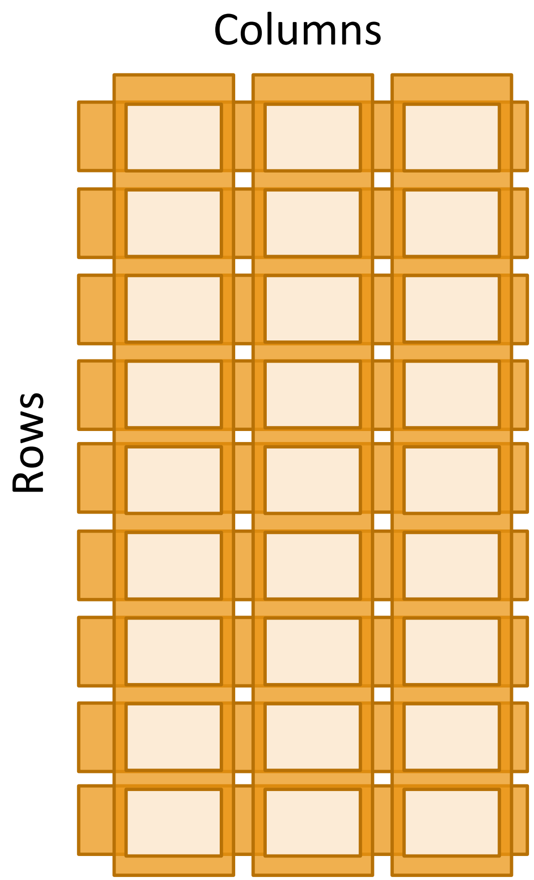
Compressing a File

- Able to Split
  - Text
  - **Seq**
  - **Arvo**
  - **Parquet**
  - **ORC**
- Not able to Split
  - **GZip file**
  - Variable row length copybook files

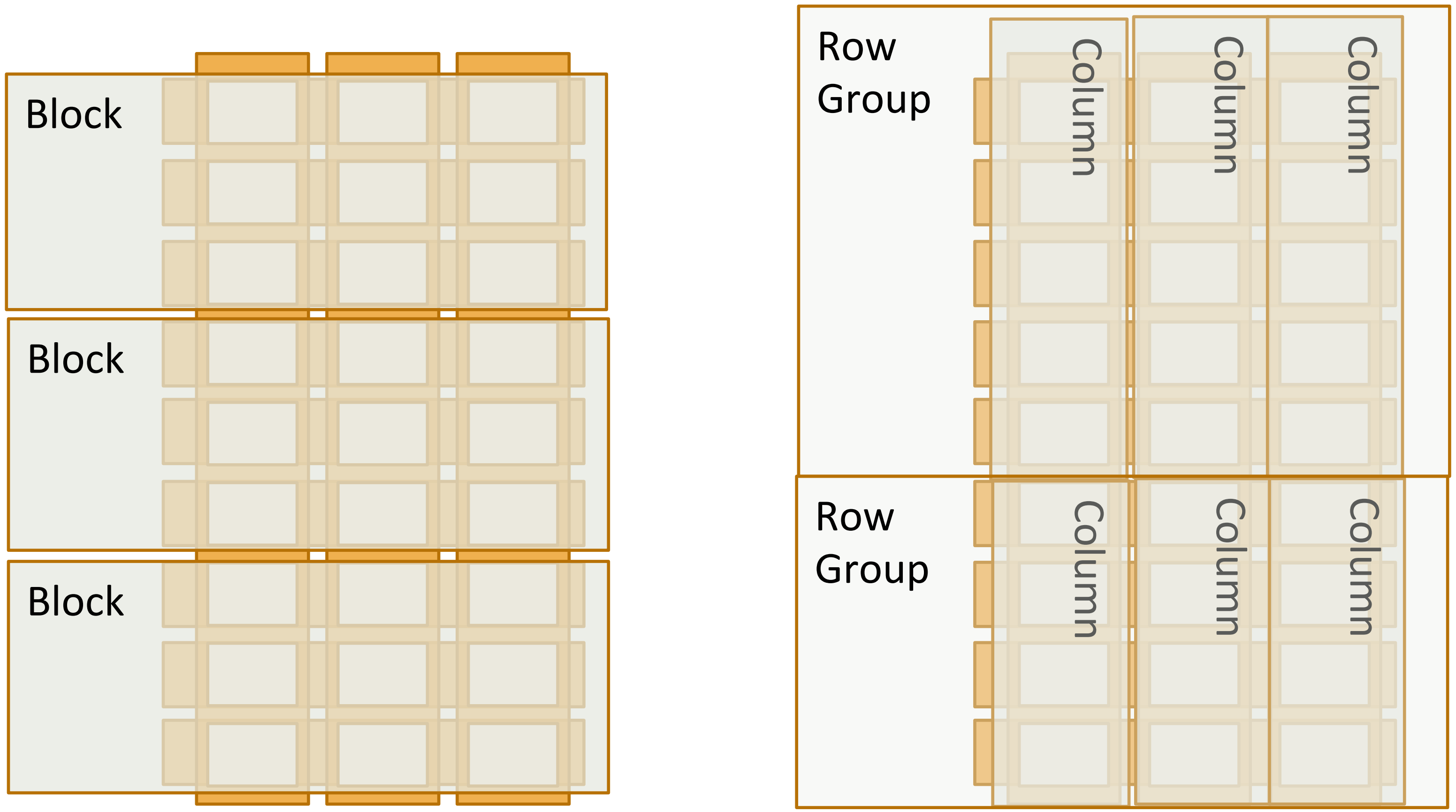
Compressing a File

	No Metadata	Metadata
Block	Sequence Files	Avro
Columnar	RC	Parquet/ORC

Compressing Styles and Entropy



Compressing Styles and Entropy



Compressing Styles and Entropy

- Pros Block
  - Less memory to read and write
- Pros Column
  - More memory to read and write
  - About 20% to 40% more compression
  - Able to select which columns to read

Compress Codec

- Snappy
- Lzo
- Gzip
- Default
- BZip2
- Others ..

- Always be skeptical

Compress Codec

- Snappy: 2x-3x : Fast Read, Fast Write
- Lzo : 2x-3x : Fast Read, Fast Write
- Gzip : ~8x: ~Fast Read, Normal Write
- Default : ~8x: ~Fast Read, Normal Write
- BZip2 : ~10x ~Fast Read, Slow Write
- Others ..

- Always be skeptical
  - All data compresses differently
  - Use your own data

Other Compression Considerations

- If you are reading remotely compression is even more important

Small Files Problem

- Metadata load on the NameNode and DataNode
- Read Speed Reduction
  - Smaller files = more seeks, less compression, shorter scans, and more metadata for the execution layer
- Ideally aim for 128MB to 1GB of compress file sizes

Introducing the Hive Metastore

- Hive Metastore
  - Add a table like metadata layer over a file system, block store, NoSql, or other
  - Allows for SQL access
  - Allows for greater security options
  - Allows for external metadata
  - Allows for partitioning

Typical Hive Table

- ParantFolder
  - TableFolder
    - Date=20171212
      - DataFiles
      - DataFiles
    - Date=20171211
      - DataFiles
      - DataFiles

Access Patterns

- Partitioning
- Filter push down
- Indexing should be consider poor
- Ideal for large scans

Query Considerations

- Data is normally big so
  - Partition respectively to access patterns
  - Join with care
  - Consider sampling or local testing before experimenting
- Data is files
  - Latency to accessibility it high – seconds, minutes or more.

Mutation Patterns

- File is written once and can not be mutated
  - Fine for append or snapshot use cases
- Mutation will require a compaction

Compaction Recap

Key	Time	Value
A	1	101
B	1	101
C	1	101
D	1	101
E	1	101
F	1	101
G	1	101

Key	Time	Value
A	2	102
D	2	102
F	2	102
F	3	103
H	3	103

Key	Time	Value
A	2	102
B	1	101
C	1	101
D	2	102
E	1	101
F	3	103
G	1	101
H	3	103

Mutation Patterns

- File is written once and can not be mutated
  - Fine for append or snapshot use cases
- Mutation will require a compaction
  - Normally a Shuffle is required
  - Else bucketing and sorting

Cost Considerations

- HDFS is really cheap comparably
- Can run on spinning drives
- Data can be compacted very tight with minimal read speed draw backs

Summary & Q/A

NoSQLs

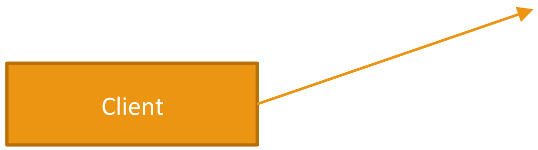
- Component break down (HBase and Cassandra)
- System fundamentals
- Mutation Patterns
- Access Patterns
- Cost considerations

What is a NoSQL

- It’s not NO SQL
- It’s not a Database
- Think of it more then a
  - HashMap
  - Log
  - Bucketed and Ordered

HashMap

- There is a Key and a Value
- It is really fast to grab a key/value
- It is really fast to add a key/value
- Iteration is also possible



Key	Value
A	1
B	1
C	1
D	1
E	1
F	1
G	1

Log with Compactions

- When new record come in they don't rewrite the old
- They compact in

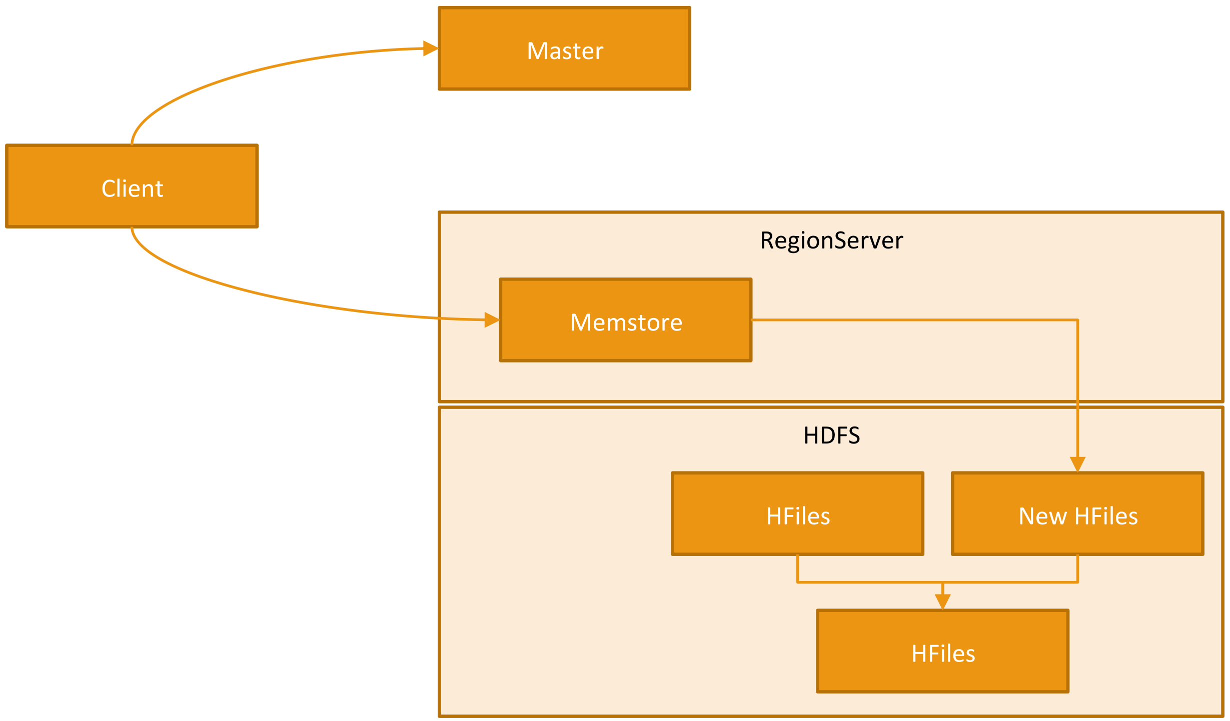
Key	Time	Value
A	1	101
B	1	101
C	1	101
D	1	101
E	1	101
F	1	101
G	1	101

Key	Time	Value
A	2	102
D	2	102
F	2	102
F	3	103
H	3	103

Key	Time	Value
A	2	102
B	1	101
C	1	101
D	2	102
E	1	101
F	3	103
G	1	101
H	3	103

Log with Compactions

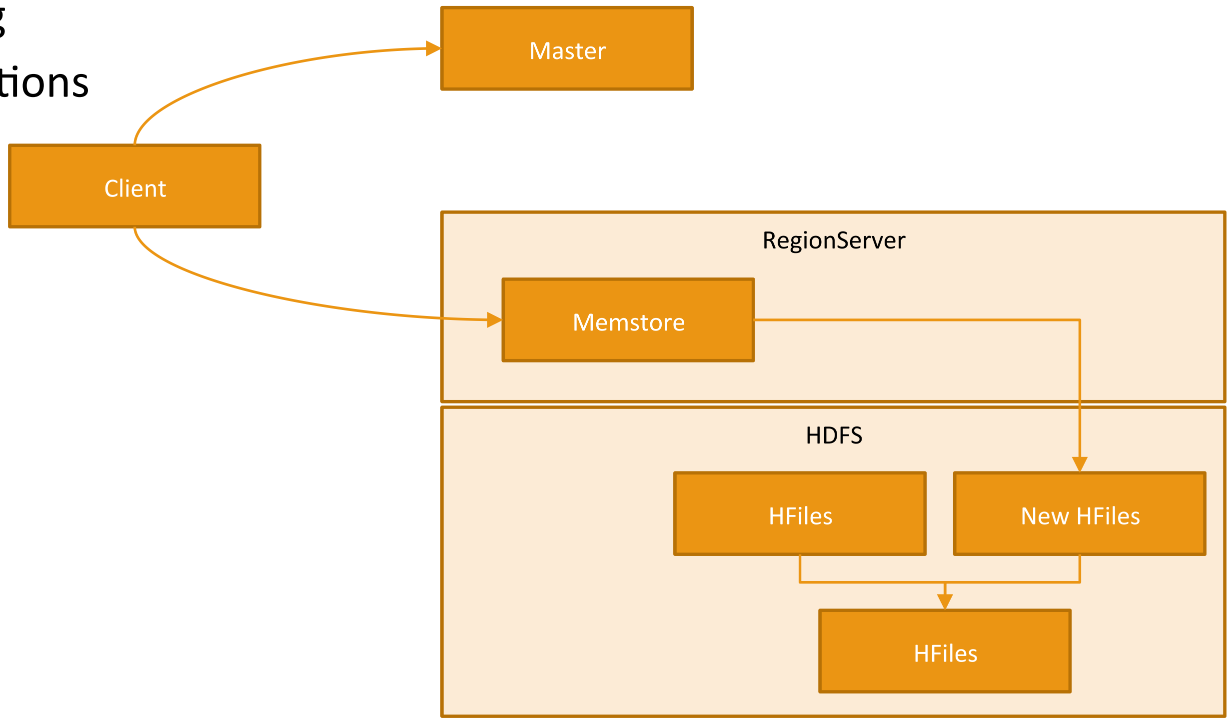
- Write Path
  - Get Local for Record (Cached)
  - First live in Memstore
    - Sorting & batching
  - Flush to New Hfile
  - Later Hfiles will be compacted



Ordered

- All Records Columns are ordered
- Ordering allows for simpler indexing
- Ordering allows for simpler compactions

- We will also use this ordering
  - Windowing
  - Time series
  - Local scaning



Bucketing or Partitions

- HBase
  - Out of the Box:
    - Range
  - Desired:
    - Salt for Bucketed HashMod
- Cassandra
  - Out of the Box:
    - HashMod
    - Bucketed HashMod

So what about SQL

- Well SQL could totally work
  - CQL for cassandra
  - Hive and SparkSQL on Hbase
- Why is it not the best idea
  - Built more for point look ups
  - Scans are not as fast as parquet
    - However the mutability may be more important then speed
  - Partitioning is not simple
    - It must be put into the key

What about Kudu

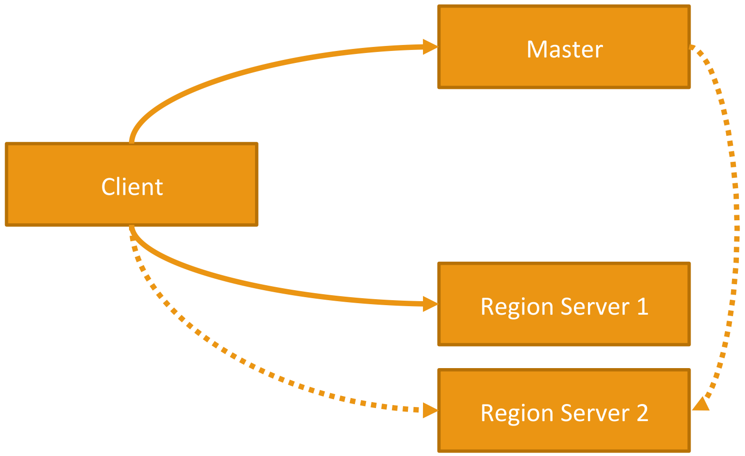
- Very much like it’s older siblings, but at the same time different
- It about trade offs
  - Giving up
    - some write speed
    - some point get speed
    - Data flexibility
    - Some ordering
  - Getting
    - Better compression
    - Better scan speeds

Let’s Talk about CAP for a Minute

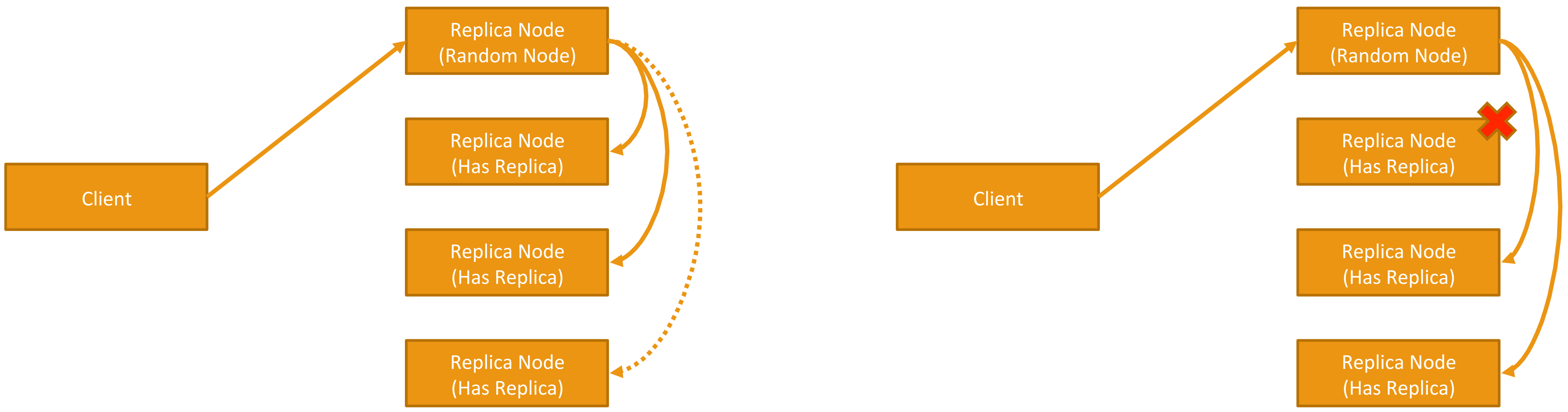
- Strong Consistence
  - HBase & Kudu
- Variable Consistence
  - Cassandra

HBase Model

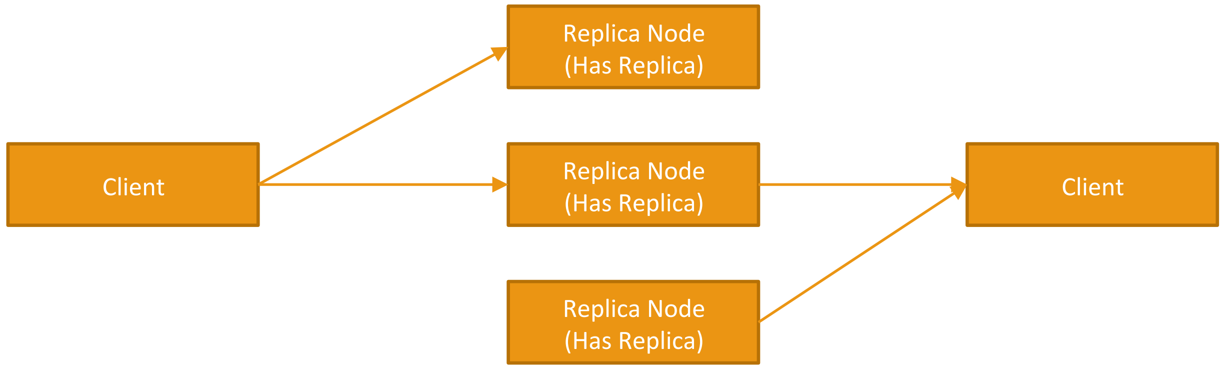
- Region Server owns range splits
- Region Server 1 fails
- Master needs to figure that out
- Master needs to assign new Region Server to own splits
- Region Server 2 has to get organized
- Region Server 2 is read to server reads and writes



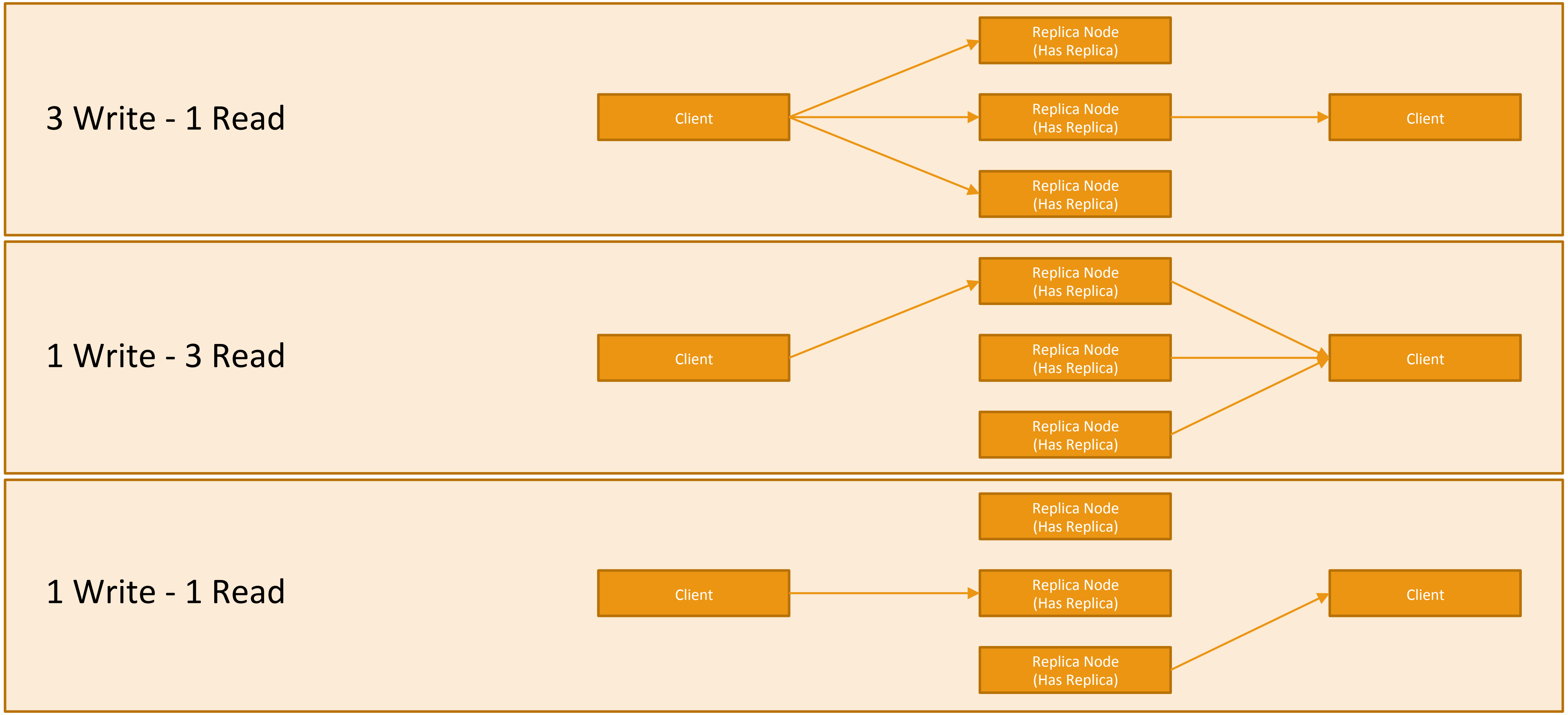
Cassandra Model



Cassandra Model



Cassandra Model (Common Models)



Race Conditions

- Almost no locking
- It all depends on the time value
- Time value can be time or custom
- Time can be set by the client or by cluster

Key	Time	Value
A	2	102
B	1	101
C	1	101
D	2	102
E	1	101
F	3	103
G	1	101
H	3	103

Summary & Q/A

Block Stores

- Component break down
- System fundamentals
- Mutation Patterns
- Access Patterns
- Cost considerations

Block Store (Like and not like HDFS)

- Like a HDFS
  - Contains files
  - Break up large files
- Not like a HDFS
  - Not really a file system and is more Key value like a NoSQL
    - Doesn’t have any metadata limit problem
    - Traversing Folder directories is more work
    - There is no rename, only copy and delete
  - Eventually consist issues with listing files
    - (seen with things like MR and Spark)
    - Can be mostly addressed with EMRFS

Block Store (Many Ideas Continue)

- Because it is like a File System
  - Almost everything you learned from HDFS stores true
    - Hive works the same
    - Files and file formats are the same

Block Store (Thinking Remote)

- Unlike HDFS the storage is always remote
  - Not on the same nodes as the execution
- Which allow you to save money in the cloud
  - Execution nodes are expensive vs storage only
- Network will be used to Read and Write
  - In fact you are normally throttled well before the network limit of your node
- You will want the highest rates of compression possible
  - To save money on storage
  - To read and write faster

Block Store (Read/Write Options)

- API
- HDFS File System API
- Hive
- Spark/Impala/Presto
- Cmd

Summary & Q/A

Lucene Indexed

- Component break down
- System fundamentals
- Mutation Patterns
- Access Patterns
- Cost considerations

Lucene Indexing

- Reverse Indexing
- That is normally severed through systems like
  - Solr
  - Elastic Search

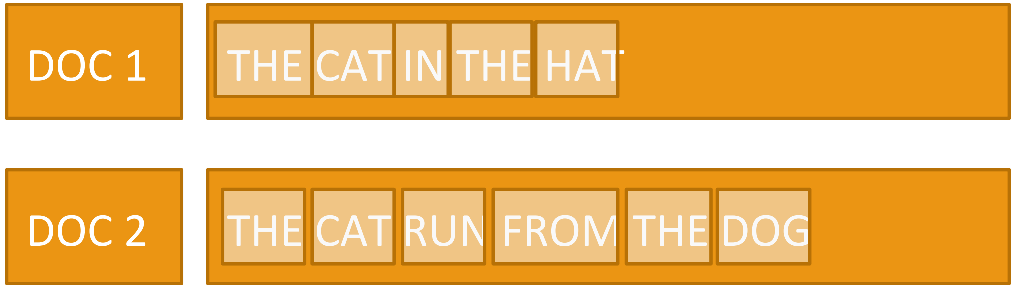
Lucene Indexing
(What is a reverse index?)

- Simple text example

DOC 1	THE CAT IN THE HAT
DOC 2	THE CAT RUN FROM THE DOG

Lucene Indexing
(What is a reverse index?)

- Simple text example



Lucene Indexing
(What is a reverse index?)

INDEXES	DOC1	DOC2
THE	*	*
CAT	*	*
IN	*	
HAT	*	
RUN		*
FROM		*
DOG		*

Lucene Indexing (Storage Cost)

- We are storing all the document
- We are storing all the indexes of every key and every document

Lucene Indexing (Features)

- We don’t have enough time in this whole class
  - Ordering logic
  - NGrams
  - Weights
  - Text Indexing
  - Translations
  - Facets *

Lucene Indexing (Facets)

- Facets are a side effect of out wonderful indexes
- It allows us to counts all the document that below to given indexes to produce
  - Grouped Counts
  - Charts and Graphs (kibana or Banana)
- People will also call this access pattern cubing a dataset

Lucene Indexing (Kibana & Banana)

Lucene Indexing (Facets Example)

- Time Series Example

Document ID	Hour of Day	User	State	Event
1	12	4201	MD	click
2	12	4202	VA	click
3	12	4203	VA	click
4	1	4201	MD	click
5	1	4202	VA	view
6	2	4204	CA	click
7	2	4205	VA	view
8	2	4201	MD	click

Lucene Indexing (Facets Example)

Document ID	Hour of Day	User	State	Event
1	12	4201	MD	click
2	12	4202	VA	click
3	12	4203	VA	click
4	1	4201	MD	click
5	1	4202	VA	view
6	2	4204	CA	click
7	2	4205	VA	view
8	2	4201	MD	click
9	2	4204	CA	click

Hour of Day				
12	1	2	3	
1	4	5		
2	6	7	8	9

State				
MD	1	4	8	
VA	2	3	5	7
CA	6	9		

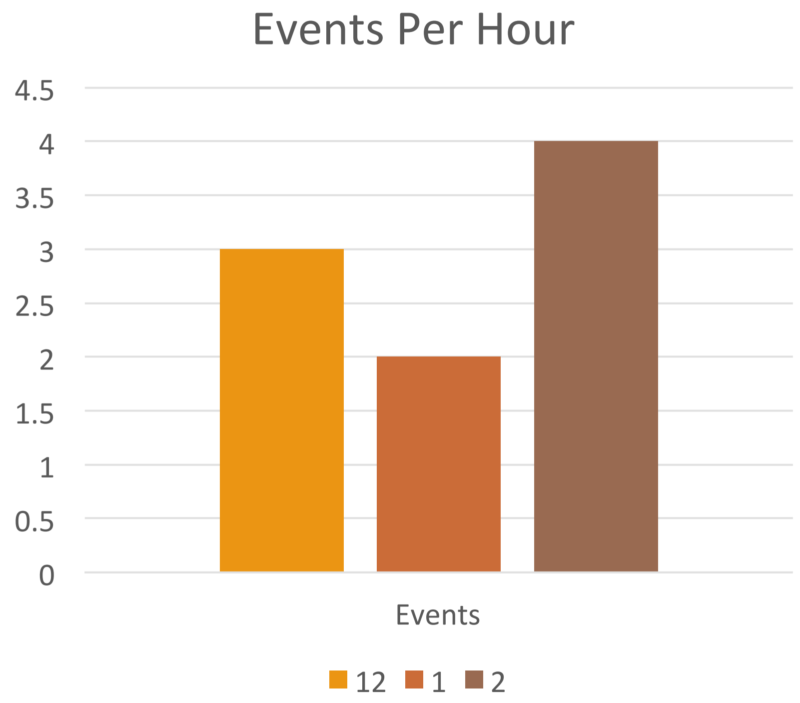
User			
4201	1	4	8
4202	2	5	
4203	3		
4204	6	9	
4205	7		

Event							
click	1	2	3	4	6	8	9
view	5	7					

Lucene Indexing (Facets Example)

- Events per hour
  - Simple array count

Hour of Day				
12	1	2	3	
1	4	5		
2	6	7	8	9

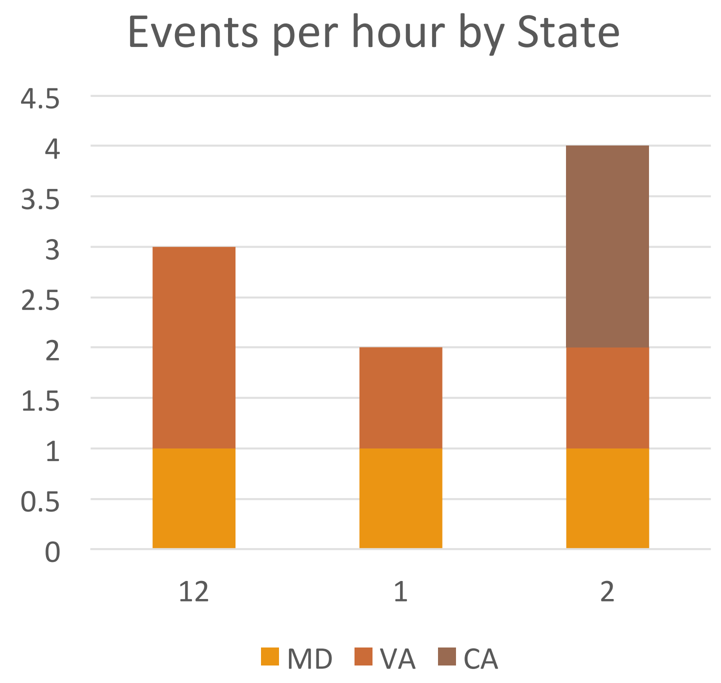


Lucene Indexing (Facets Example)

- Events per hour by State
  - Simple array count

Hour of Day				
12	1	2	3	
1	4	5		
2	6	7	8	9

State				
MD	1	4	8	
VA	2	3	5	7
CA	6	9		



Lucene Indexing (Facets Example)

- Note the bucketing and ordered pattern

Hour of Day				
12	1	2	3	
1	4	5		
2	6	7	8	9

State				
MD	1	4	8	
VA	2	3	5	7
CA	6	9		

Hour of Day 2	State MD	State VA	State CA
6	1	2	6
7	4	3	9
8	8	5	
9		7	

Lucene Indexing (Facets Example)

- Note the bucketing and ordered pattern

Hour of Day 2	State MD	State VA	State CA
6	1	2	6
7	4	3	9
8	8	5	
9		7	

Hour of Day 2	State MD	State VA	State CA
6	1	2	6
7	4	3	9
8	8	5	
9		7	

+1 CA

Lucene Indexing (Facets Example)

- Note the bucketing and ordered pattern

Hour of Day 2	State MD	State VA	State CA
6	1	2	6
7	4	3	9
8	8	5	
9		7	

+1 VA

Hour of Day 2	State MD	State VA	State CA
6	1	2	6
7	4	3	9
8	8	5	
9		7	

+1 MD

Hour of Day 2	State MD	State VA	State CA
6	1	2	6
7	4	3	9
8	8	5	
9		7	

+1 CA

Partitioning

- SolR and Elastic Search partition the document o land on all nodes
- This means
  - You have the power of the cluster when querying
  - This mean you are accessing the cluster when querying

Writing Latency

- Lucene Indexing is more expensive then NoSQL work
- Think of it as micro batching
  - Larger batches ~= better throughput
- Compaction is also invalid
- Deletes impact storage and performance until they are compacted

Storage Cost

- TTL is your friend
- Think of Lucene based systems as great if
  - You dataset is manageable in size
  - You have a good TTL strategy
  - You have a boat load of money

Summary & Q/A

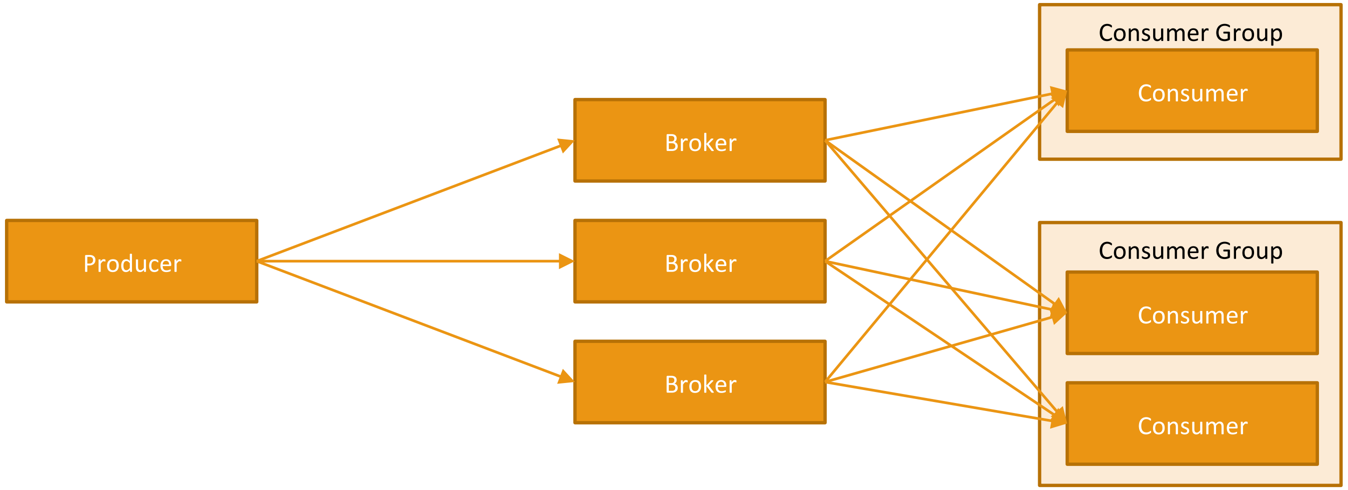
Kafka

- Component break down
- System fundamentals
- Mutation Patterns
- Access Patterns
- Cost considerations

Distributed Log

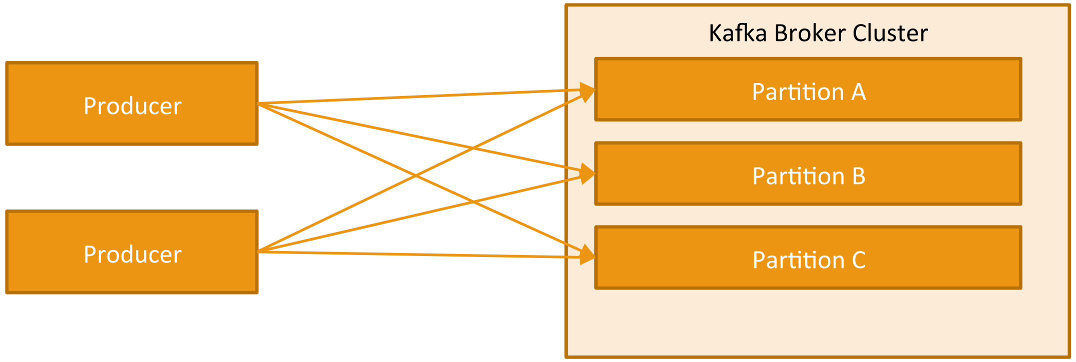
- Pub-sub vs distributed log
- One to one
- One to many

Basic Parts

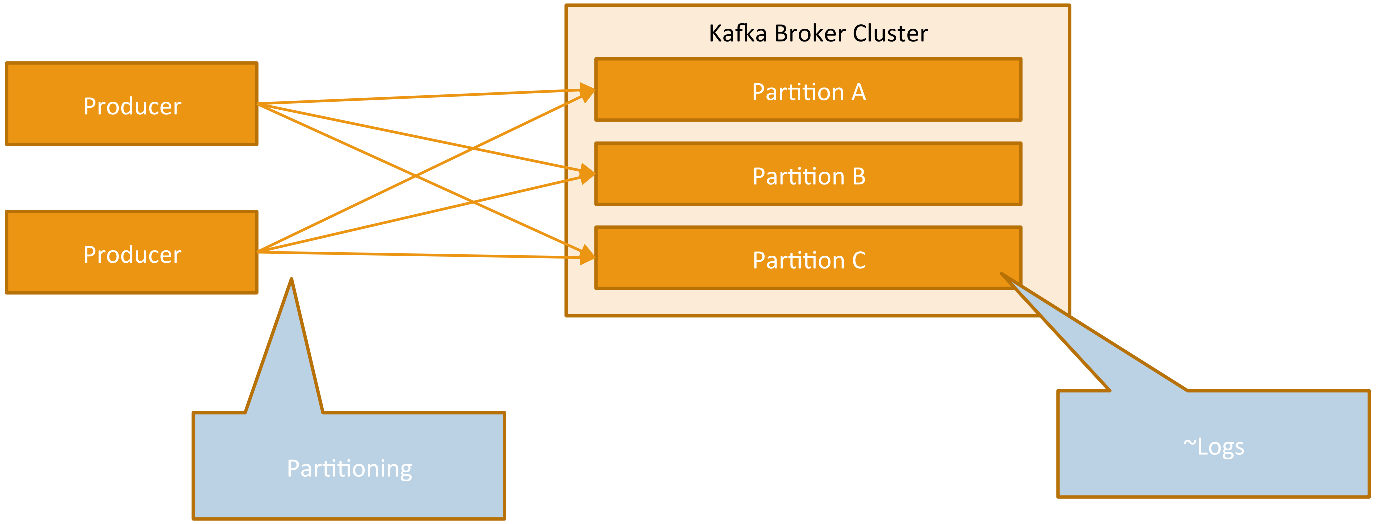


Producer
Broker
Broker
Broker
Consumer Group
Consumer
Consumer Group
Consumer
Consumer

View from the Producer



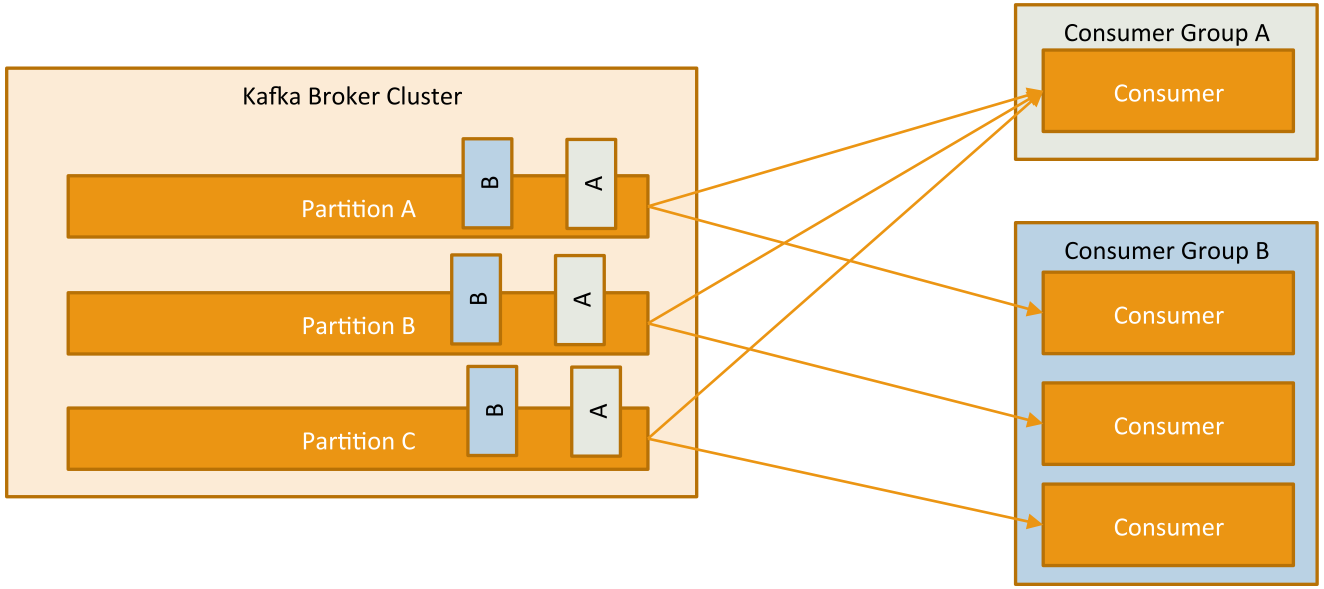
View from the Producer



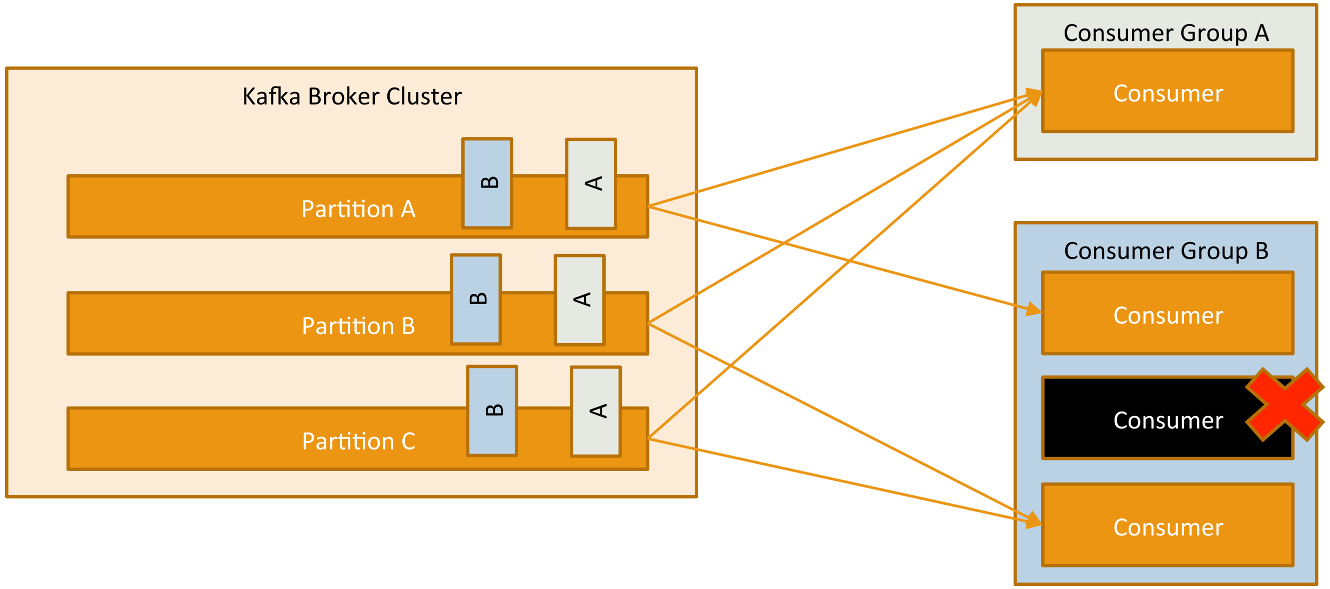
Partitioning

- Round Robin
- Lazy Round Robin
- Custom
  - Good and Bad
  - Avoid Skew

View from the Consumer



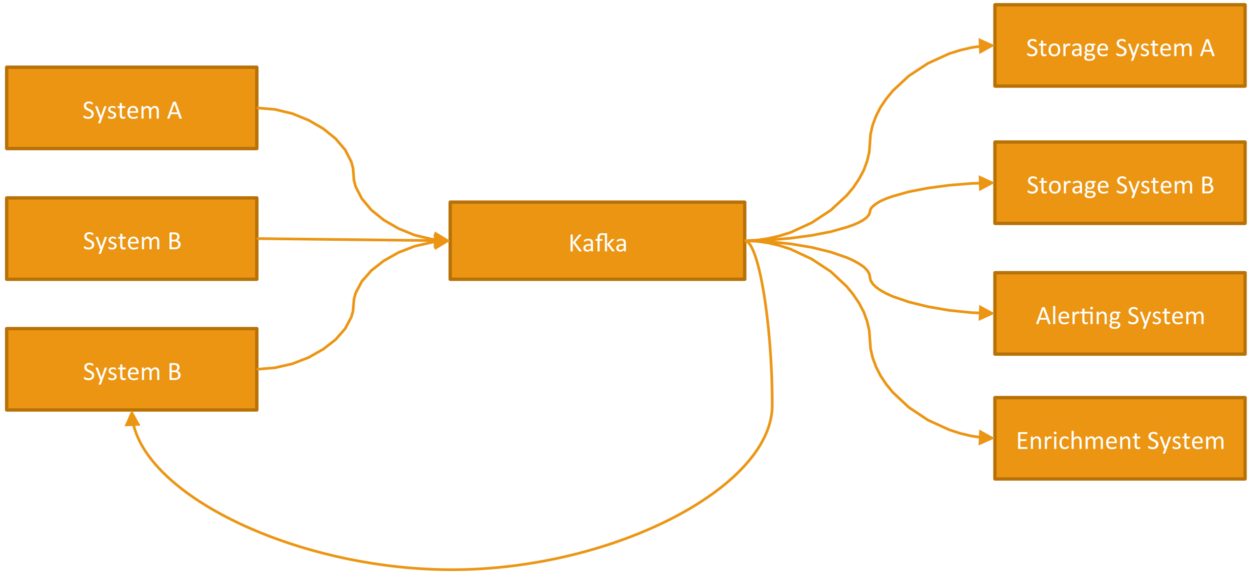
View from the Consumer



Why are we talking about Pub-Sub?

- Query on a stream
  - Think about Spark Streaming (Structured Streaming)
    - SQL on the fly
- Compaction Queues
- Golden Path

Golden Path (ESB)



Summary & Q/A

Quick System Summery

- Quick Review of System Types

Relational Refresher

- Data Models
  - One to one
  - Many to one
  - Many to many
  - Fact tables
  - Star schemas
- Access Patterns
- Mutation Patterns
- Transactions

Hadoop HDFS

- File System
- File Types and Strategies
- Access Patterns
- Read Patterns
- Hive

NoSQL

- HBase and Cassandra
- HashMaps
- Ordering
- Write Paths
- Read Paths
- Scan speeds
- Kudu

Block Store

- Looks like a file system but more like a NoSQL
- Remote vs local storage
- Compression is your friend
- Hive
- Consistence
- Metadata Management

Lucene Based Systems

- What was a Lucene Index
- Facets
- Access patterns
- Write Patterns

Kafka

- Pub-Sub vs a Distributed log
- Producers
- Consumers
- Consumer groups
- Brokers
- Compaction queues

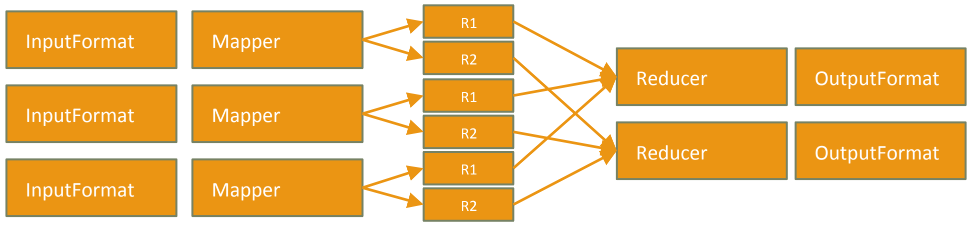
Summary & Q/A

Distributed Execution Engines

- Quick list of Engines
- What they are good for
- What they are limited at

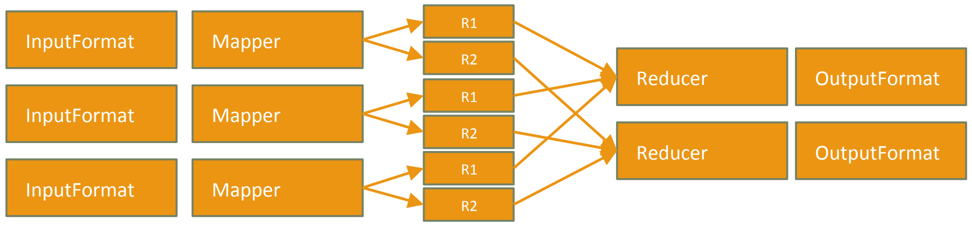
In the beginning

- MapReduce
  - InputFormat
  - Splits/Record Reader
  - Mapper
  - Combiner
  - Shuffle
  - Reducer
  - OutputFormat



Outcome of MR

- Pros of MapReduce
  - Process Huge amounts of data
  - Read and write to anything
- Cons of MapReduce
  - Hard to code
  - Hard to chain
  - Hard to debug
  - Long startup times
  - Very IO heavy



Attempts to make it better

- SQL
  - Hive
- Coding Styles
  - Cascading
  - Pig
  - Crunch

Then can Spark

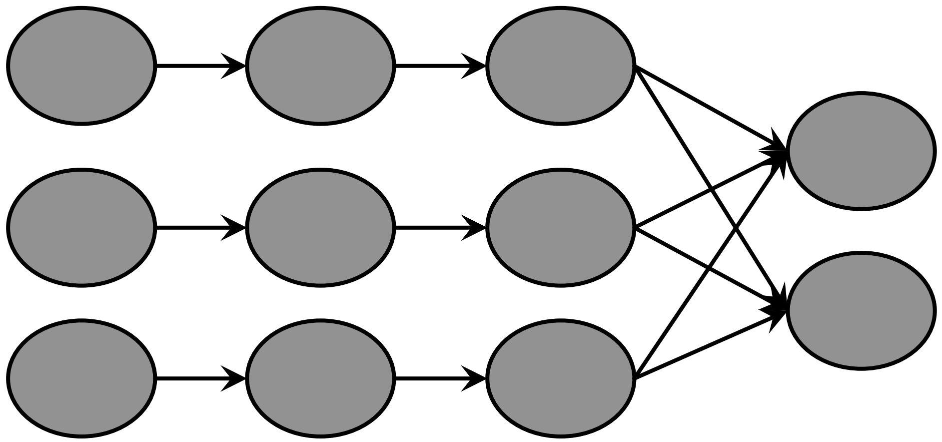
- Spark: Cluster Computing with Working Sets Paper 2010
  - By Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
- Addresses
  - Coding style
  - Chaining
  - Startup times
  - Debugging
  - IO

It continues

- Impala
- Presto
- Flink
- Others…

DAG

- Directed Acyclic Graph



Managing Your DAG

- Broadcast Join: Only if one side is small
- Coalesce: Only is you are down partitioning
- Partition Order: Only if you are ordering within a partition
- Bucketed and Ordered Join: Needs data to be prepped before read
- Shuffle
- Shuffle and Sort Join

SQL or Not

- Can you think of the SQL in less then 2 minutes?
- How painful are your joins?
- What level of concurrence are you expecting?

If not SQL then what?

- Spark code (java, scala, python)
- NoSQLs
- Lucene

Summary & Q/A

Use Case Definition

- List out use cases we are going to solve for

Use Cases

- Data Retention
- Analytical SQL
- Charting
- State change
- Event Logs
- Deep Learning
- Graph querying

Data Retention

- Data Cost
- TTL

Analytical SQL

- Can I get access to the data with tools I know
- Can I get answers fast
- Can I support many queries

Charting

- Can I get easy pretty charts
- Can they support exploration
- Can they support lot of concurrent users and queries
- Role ups and aggregations

State Changes & Event Logs

- How do I get a user experience like Gmail or facebook with that level of scale

Deep Learning and Graphs

- How do I get my data in a format to I can learn from it

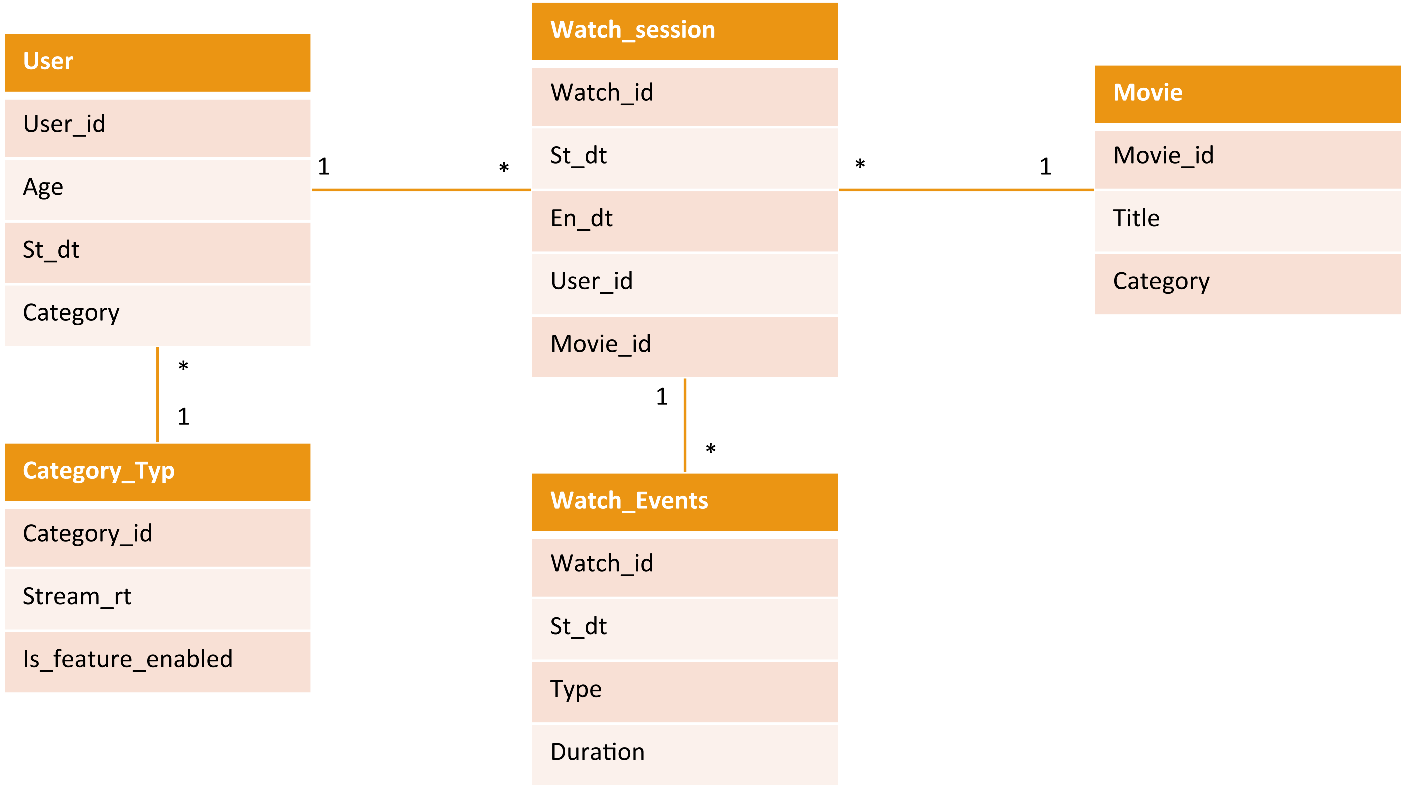
Starting with Relational

- Build out our use cases in a relational model

Always Start Relational

- It is what people know
- It is how humans naturally see the world
- It is easy for mutations

We’ll use the model we started with



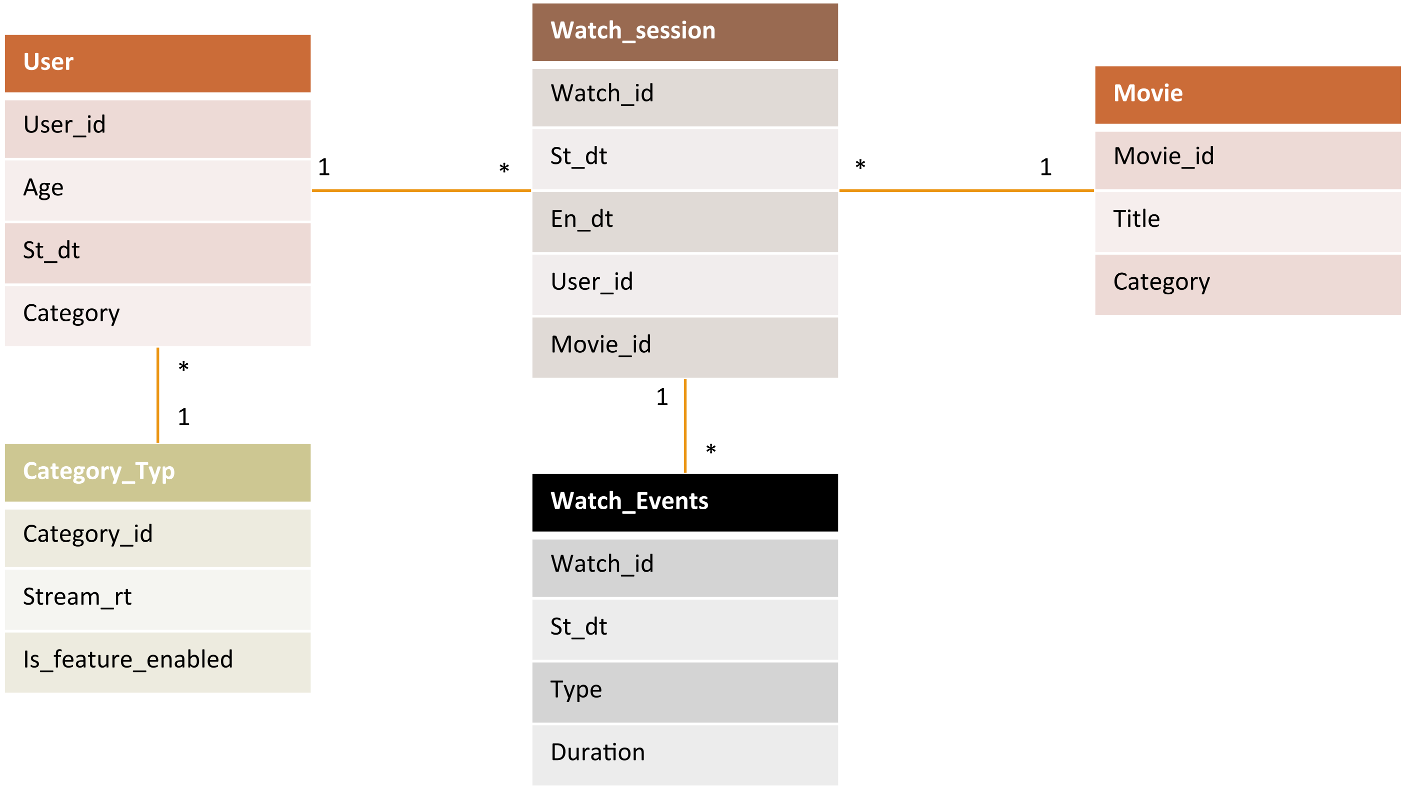
When to start moving away from Relational

- When things get painful
  - Latency
  - SLA
  - Queries per second

Partitioning can help

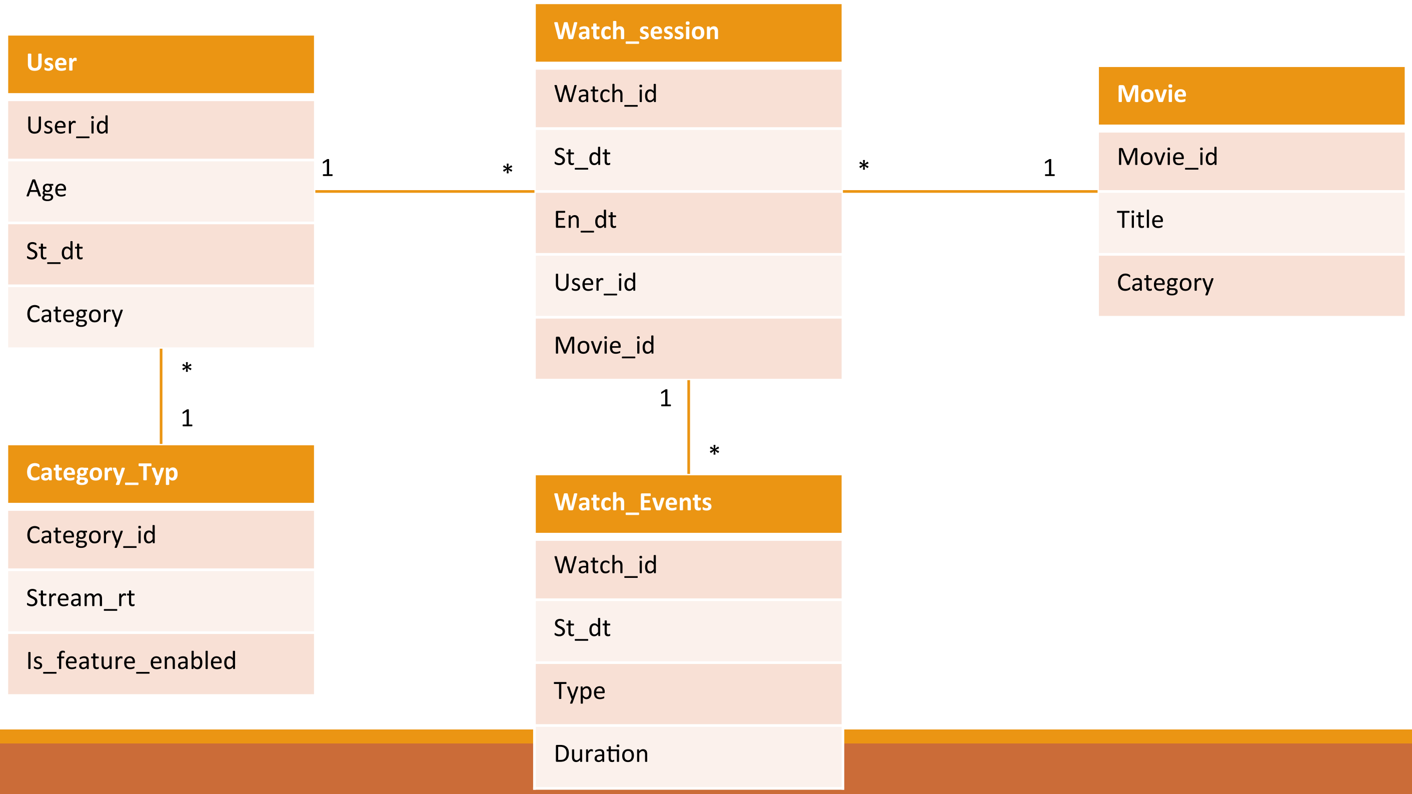
- **The Need:** We will quickly see to use a Big Data solution and relational that partitioning is a given
- **Flexible:** However partition is not Flexible enough for everything

Look for big tables



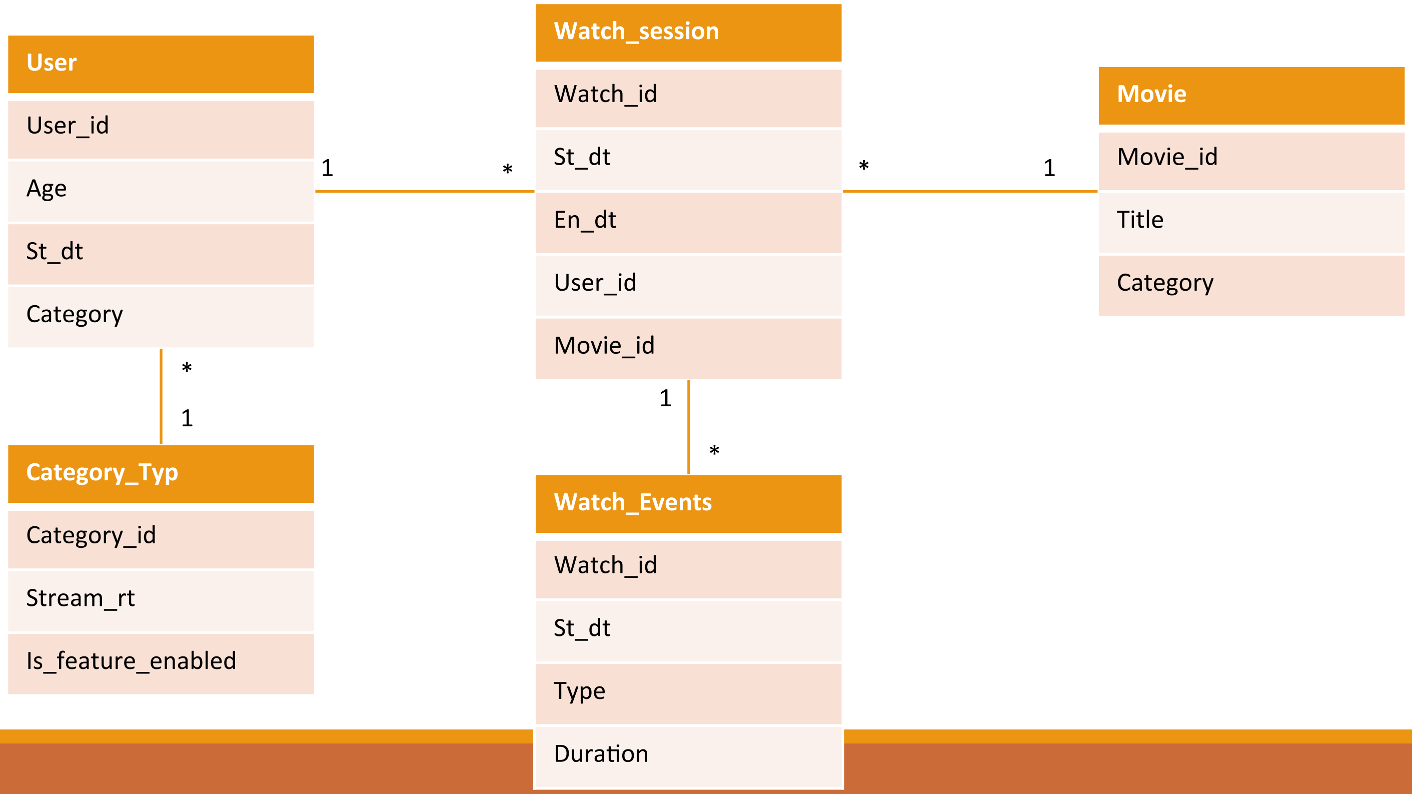
Will this solve the following

- Get user movie count for a day
- Find a user's favorite movie category
- Find the most watched movie for user category



Will this solve the following

- Get user movie count for a day: One Big Shuffle
- Find a user's favorite movie category: ~Two Big Shuffles
- Find the most watched movie for user category: ~Two Big Shuffles



Summary & Q/A

Applying Denormalization & Nesting

- The attempt to reduce super large joins

Materialized Views Considerations

- Parent Child Relationship
  - De-normalized & Nested
- Many to Many Relationship
  - De-normalized & Nested

De-normalized

User_id	Age	St_dt	Catageory	Watch_id	Watch_st_dt	Watch_end_dt	Watch_movie_id
4201	42	12/12/12	Normal	101	12/13/12	12/13/12	201
4201	42	12/12/12	Normal	102	12/14/12	12/14/12	202
4201	42	12/12/12	Normal	103	12/15/12	12/15/12	203
4201	42	12/12/12	Normal	104	12/16/12	12/16/12	204
4202	64	12/1/15	Normal	105	12/13/16	12/13/12	201
4202	64	12/1/15	Normal	106	12/13/16	12/13/12	202

Nested

User_id	Age	St_dt	Catageory	Watch_id	Watch_st_dt	Watch_end_dt	Watch_movie_id
4201	42	12/12/12	Normal	101	12/13/12	12/13/12	201
				102	12/14/12	12/14/12	202
				103	12/15/12	12/15/12	203
				104	12/16/12	12/16/12	204
4202	64	12/1/15	Normal	105	12/13/16	12/13/12	201
				106	12/13/16	12/13/12	202

Nesting Example

- {"**group**": "**A**", "**time**": 5, "**value**": 3, "**nested**": [{"**col1**": 0.1, "**col2**": 0.2},{"**col1**": 1.1, "**col2**": 1.2}]}

- {"**group**": "**A**", "**time**": 5, "**value**": 3, "**nested**": [
    - {"**col1**": 0.1, "**col2**": 0.2},
    - {"**col1**": 1.1, "**col2**": 1.2}
-]}

Spark Example

- **val** jsonDf = sparkSession.read.json(jsonPath)
- jsonDf.foreach(row => {
  println(row)
  })

- **row:[A,WrappedArray([0.1,0.2], [1.1,1.2]),5,3]**
- **row:[B,WrappedArray([1.0,2.0], [1.0,2.0]),5,3]**
- **row:[C,WrappedArray([1.0,2.0], [1.0,2.0]),5,3]**

Running SQL

- jsonDf.createOrReplaceTempView("**json_table**")

- sparkSession.*sqlContext*.sql("**select group, nested.col1 from json_table**").collect()
  .foreach(r => *println*("**sql:**" + r))

- **sql:[A,WrappedArray(0.1, 1.1)]**
- **sql:[B,WrappedArray(1.0, 1.0)]**
- **sql:[C,WrappedArray(1.0, 1.0)]**

sparkSession.*sqlContext*.sql(

"select group, a.col1 from json LATERAL VIEW explode(nested) as a").collect()
.foreach(r => *println*(**"sql:"** + r))

- **sql:[A,0.1]**
- **sql:[A,1.1]**
- **sql:[B,1.0]**
- **sql:[B,1.0]**
- **sql:[C,1.0]**
- **sql:[C,1.0]**

Hive Table Example

```
Create table car_ownership (
  Person string,
  Cars ARRAY < STRUCT <
              Title: STRING,
              Maker: STRING,
              Tires: ARRAY < STRUCT <
                         Size: String,
                         Pressure: String
              >>
   >>
)
```

Nested Options

- Structs
- Array of Structs
- Array of values

De-normalized vs Nested

- Nested Pros
  - Co-location
    - Faster to group by
    - Faster to window
    - Joins are free
    - Less data
    - Better compression
  - Tables and Columns can be read with out penalty from one not read
  - Great for limiting the effort are Cartesian Joins
- Nested Cons
  - Size limitation of parent row
  - Adding child requires the re-write the the whole parent record

Options for appending Nested

- It is all about the parent record
- We can add more then one Partition key for the parent
- In our use case
  - User & watch month or day

Storage and In Memory

- Also don’t limit the idea of nested to just tables
- In Spark they can be used as in memory constructs to
  - conserve on networking
  - In memory cost

Summary & Q/A

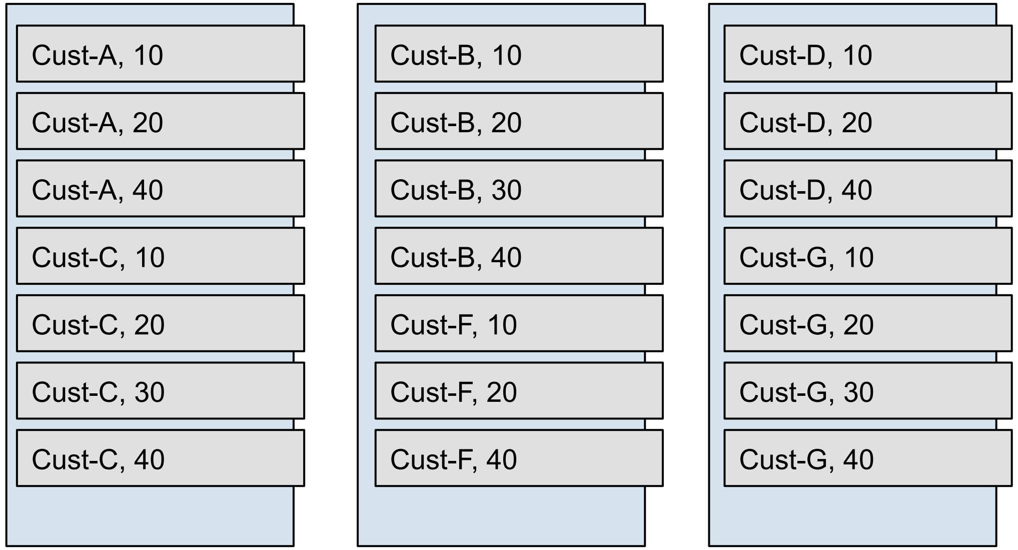
Applying Bucket & Sorting And Complex Types

- Build out our use cases in a relational model

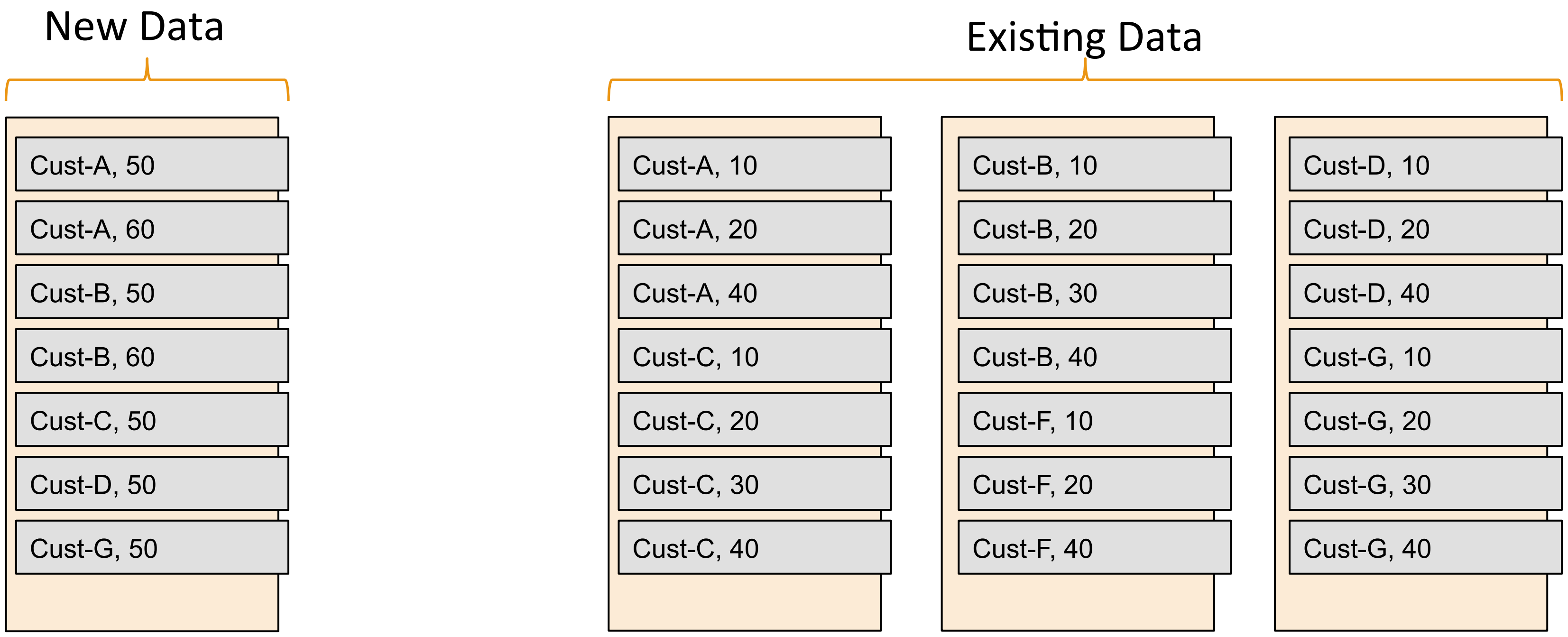
What is meant by Bucketing and Sorting

- Partitioning on a Key
- Then sorting on that key + another field(s)
- Example
  - User_id + Watch Event Time

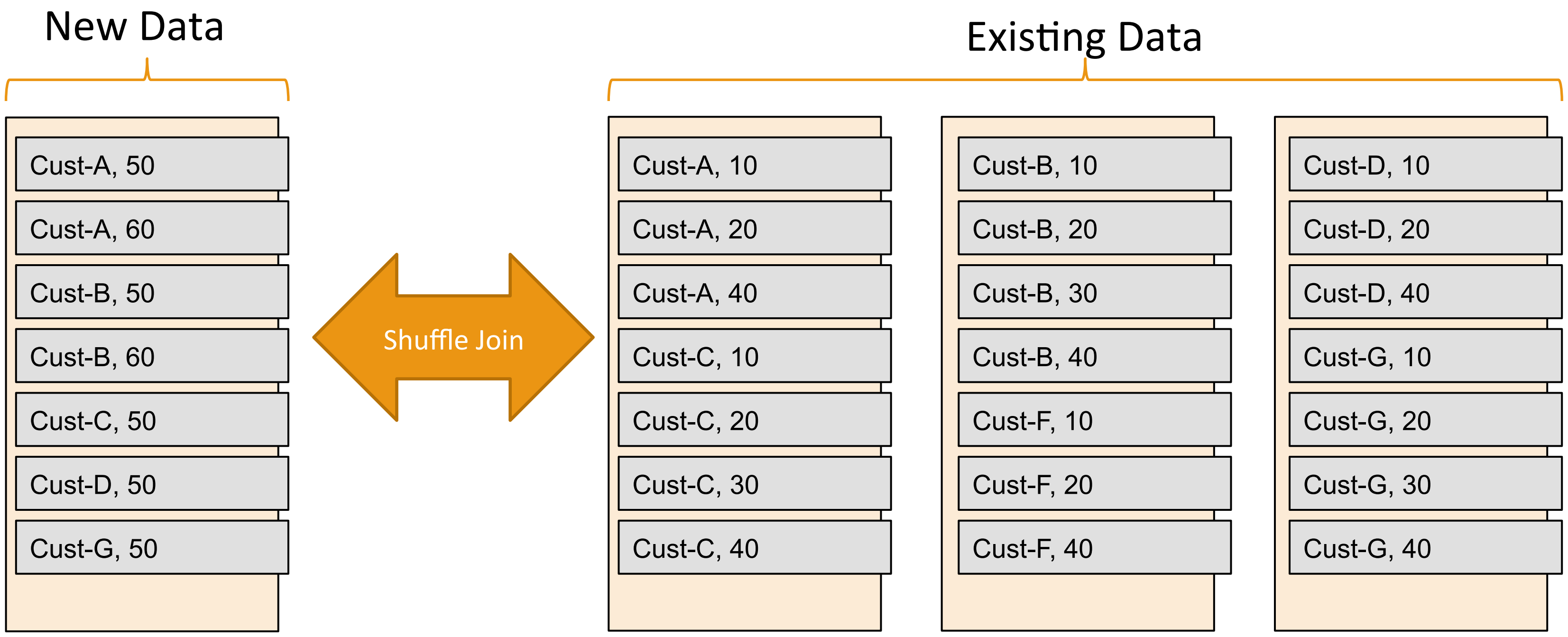
Example of Bucketed Sorted



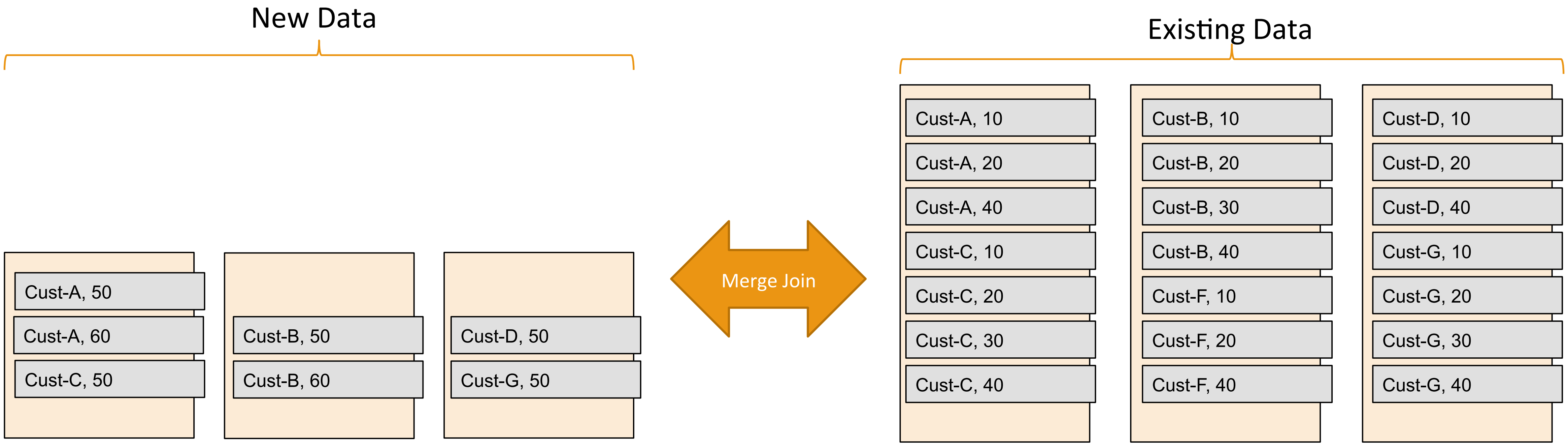
Good for appending Nested



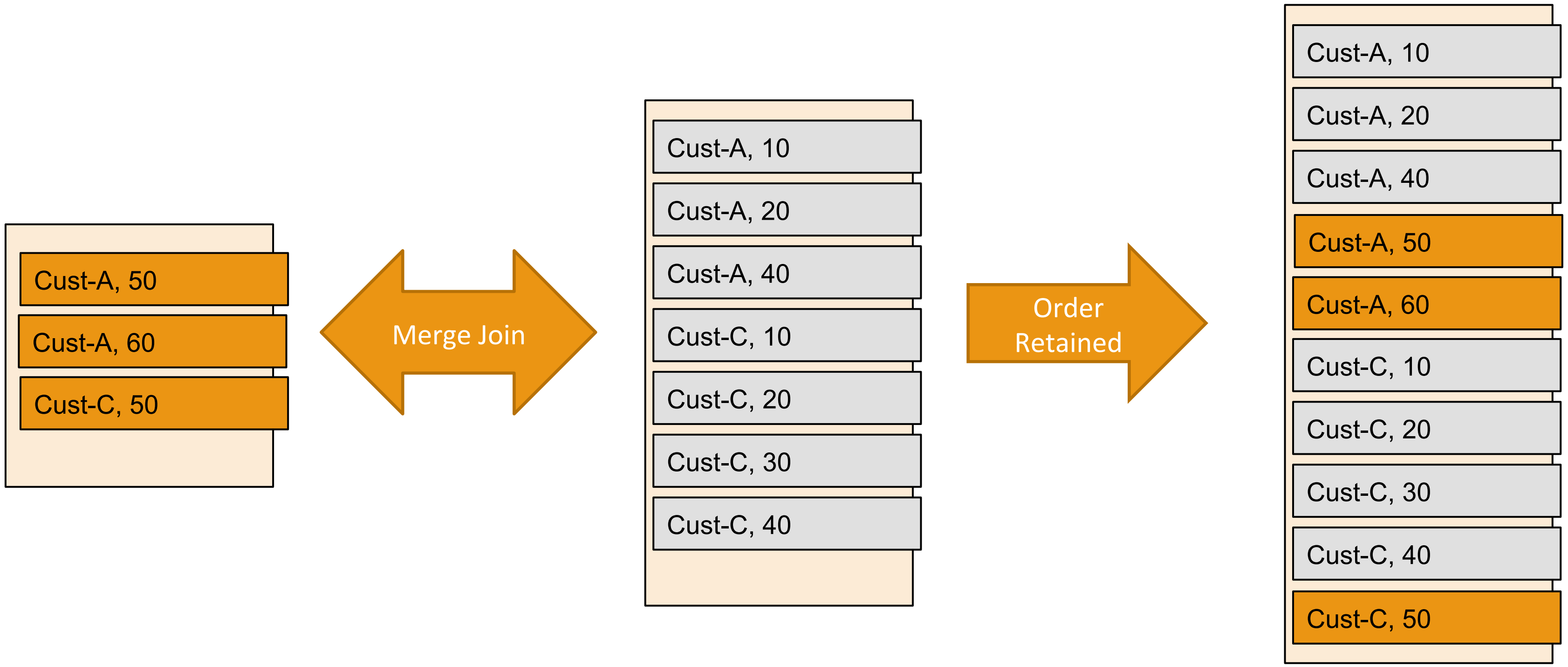
Good for appending Nested



Good for appending Nested



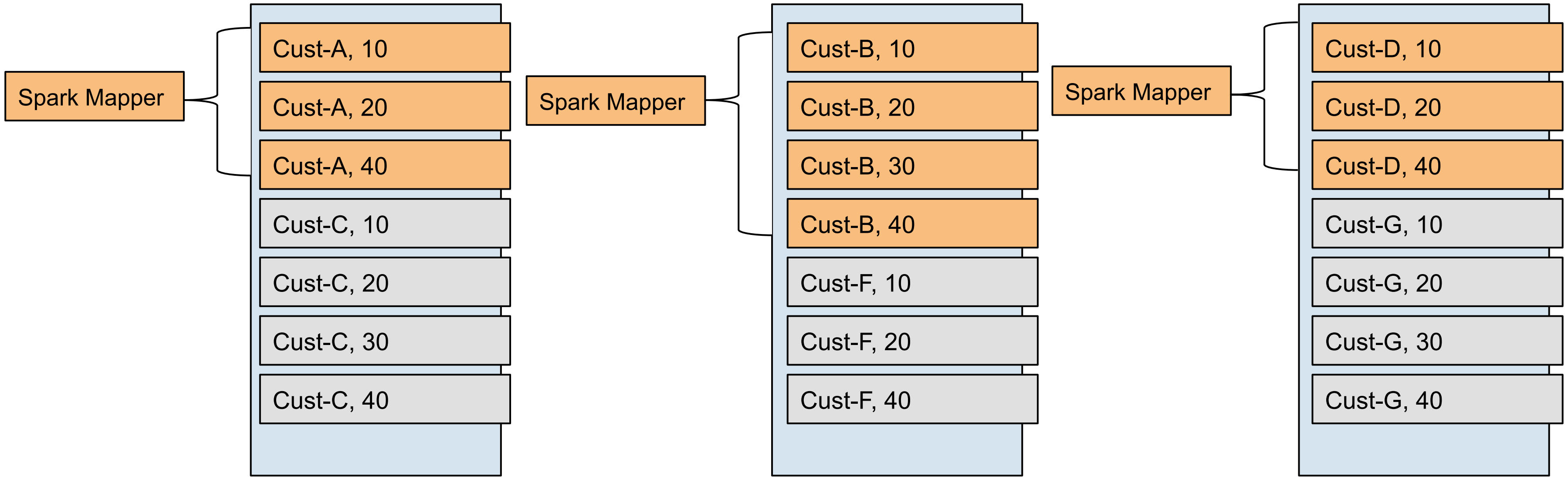
Good for appending Nested



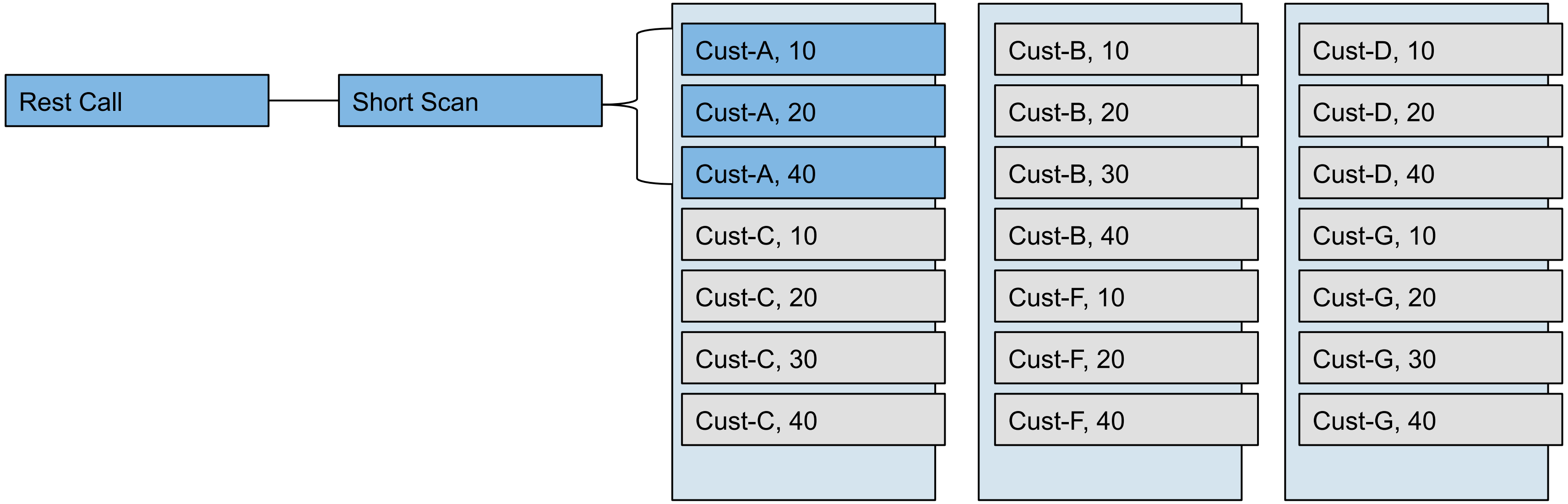
What else could be use Bucketing and Sorting for

- Windowing
- Point retrieval

Bucketed & Sorted for Windowing



Bucketed Sorted in a NoSQL



Summary & Q/A

Applying NoSQL

- Build out our use cases in a relational model

The Security to NoSQL

- It is all about the key and partitioning
  - Sorting
  - Partitioning
  - Access patterns are key
  - Limiting scans
- Co-locating everything you need

Time Series Databases

- OpenTSDB and KairosDB
  - Metric
  - Tags
  - Time
  - Value

Time Series Databases

- OpenTSDB and KairosDB
  - Metric = Hours of TV Watched per Area (Table Name)
  - Tags = Zip, State, TimeZone (Group by columns)
  - Time = Time Snapshot
  - Value = A number value

Hours of TV Watched per Area				
Zip	**State**	**TimeZone**	**Time Snapshot**	**Hours of TV Watch**
20878	MD	EST	1:00	101
20878	MD	EST	2:00	50

How is KairosDB stored on Disk

- KairosDB
  - Table 1: Filter down the metric and tags
  - Table 2: Get the values

KairosDB

- Filter Down Table
  - Partitioned by Mertic
  - Sorted by Metric and Tags

Metric	Tags
TvHourPerArea	State=MD,Zip=20878
TvHourPerArea	State=MD,Zip=20879
TvHourPerArea	State=MD,Zip=20878
TvHourPerArea	State=CA,Zip=91628
XHoursOfSleepPerArea	State=MD,Zip=20878
XHoursOfSleepPerArea	State=MD,Zip=20878

KairosDB

- Give Metric “TvHourPerArea” and Tag State=MD

Metric	Tags
TvHourPerArea	State=**MD**,Zip=20878
TvHourPerArea	State=**MD**,Zip=20879
TvHourPerArea	State=**MD**,Zip=20878
TvHourPerArea	State=CA,Zip=91628
XHoursOfSleepPerArea	State=MD,Zip=20878
XHoursOfSleepPerArea	State=MD,Zip=20878

Scan over TvHourPerArea

Fag all row with MD

KairosDB

- Second Table hold the time value data

Metric+Tags Key	Time Value Columns in Order
TvHourPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
TvHourPerArea:State=MD,Zip=20879	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
TvHourPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
TvHourPerArea:State=CA,Zip=91628	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
XHoursOfSleepPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
XHoursOfSleepPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]

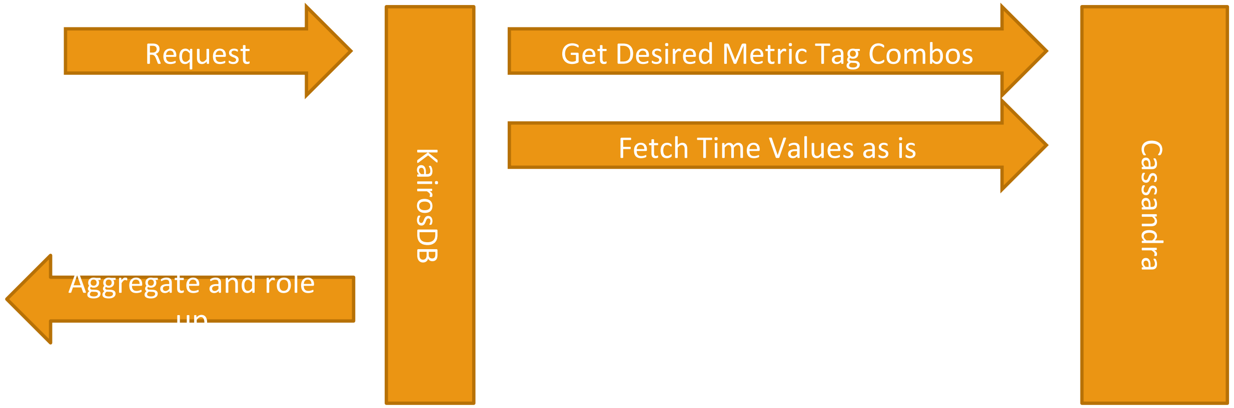
KairosDB

- Second Table hold the time value data

Metric+Tags Key	Time Value Columns in Order
TvHourPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
TvHourPerArea:State=MD,Zip=20879	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
TvHourPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
TvHourPerArea:State=CA,Zip=91628	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
XHoursOfSleepPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]
XHoursOfSleepPerArea:State=MD,Zip=20878	[Time,Value] [Time,Value] [Time,Value] [Time,Value]

Time Value & On the Fly Role Ups

- Time interval
- On Fetch the KairosDB server can do role ups of values
  - Request: Metric, Tag Pairs, Time Range, and desired intervals



Why Time Series on NoSQL

- Low cost of Storage
- Fast to get data
- Fast and easy rule ups
- Fast to aggregate up to the 10ks of values

Role Ups to long running queries

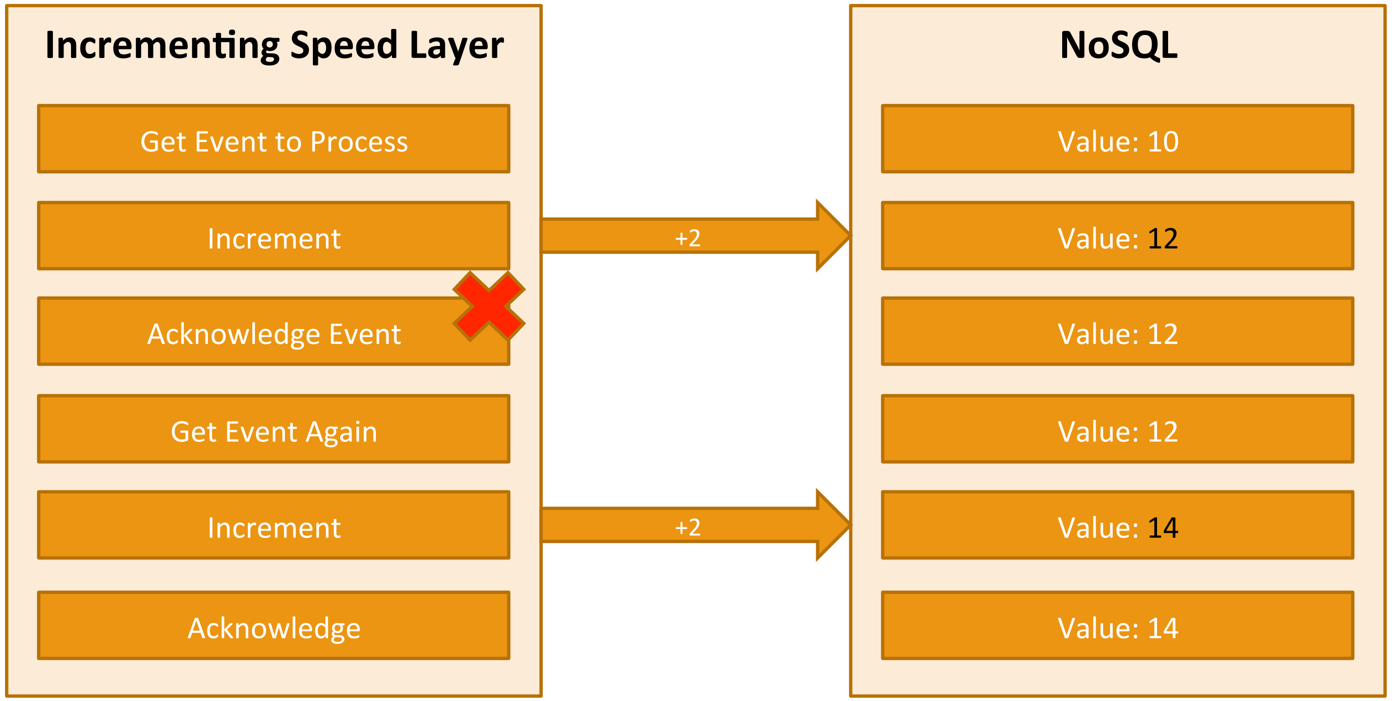
- If desired you can role up a metric tag combo by querying and rewriting under a different metric name

- Example minutes to 10 minutes for queries last years

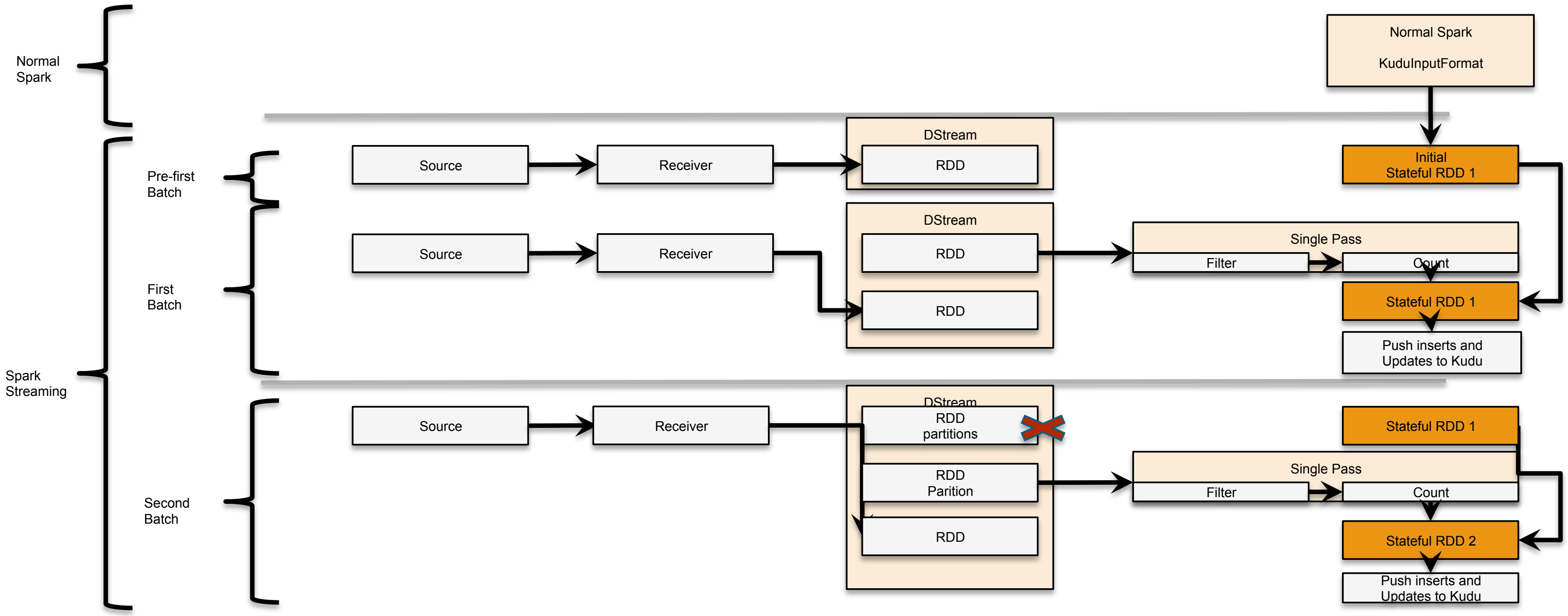
Beyond ~10k

- At some point this model of aggregating on query doesn’t work
- Which calls for Aggregation on write
  - This is a great case for something like Spark streaming
  - In a non-lambda way

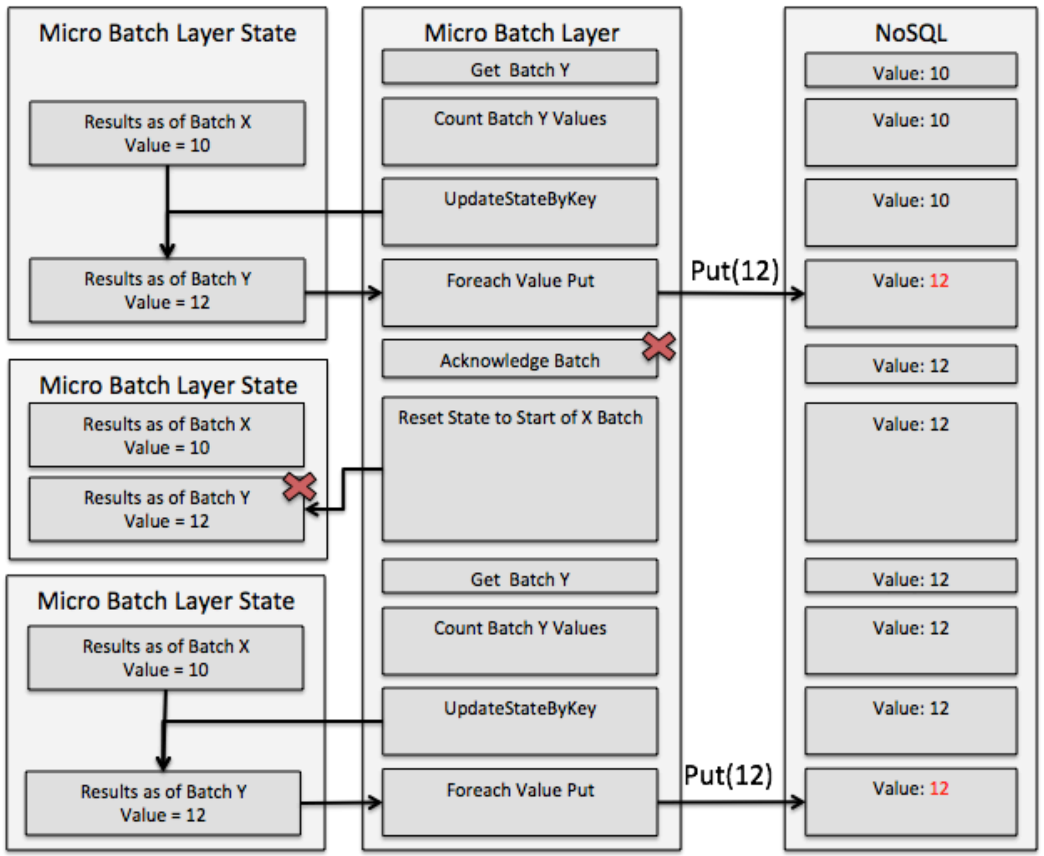
Normal Lamdba



Spark Streaming + Internal State



With State



Micro Batch Layer State
Results as of Batch X
Value = 10
Results as of Batch Y
Value = 12
Micro Batch Layer State
Results as of Batch X
Value = 10
Results as of Batch Y
Value = 12
Micro Batch Layer State
Results as of Batch X
Value = 10
Results as of Batch Y
Value = 12
Micro Batch Layer
Get Batch Y
Count Batch Y Values
UpdateStateByKey
Foreach Value Put
Acknowledge Batch
Reset State to Start of X Batch
Get Batch Y
Count Batch Y Values
UpdateStateByKey
Foreach Value Put
Put(12)
Put(12)
NoSQL
Value: 10
Value: 10
Value: 10
Value: 12
Value: 12
Value: 12
Value: 12
Value: 12
Value: 12
Value: 12

Appling to Profiles

- Putting a profile in a single row
  - Protecting you from transactions
  - Rows can be long but also contain complex types
- And keeping a partitioned sorted event log
  - Allowing for super fast paging

Versioning

- Be able to see record values of past changes
- This is totally configurable
- Consider having version stored in different table that primary access table
  - If past versions is not part of your normal access pattern

Summary & Q/A

Applying Cubing with Lucene

- Build out our use cases in a relational model

Summary & Q/A

Thinking about Graph

- Build out our use cases in a relational model

Summary & Q/A