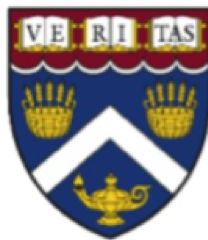


CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova



Lecture 15 Big Data Processing Use Cases 2

@Marina Popova

Agenda

Today:

- Last Lecture! Yeah!
- Admin info
- Review and Analysis of a few Real-Life BigData Processing Architectures

Next week:

- Final Projects presentations - party time!!!



Admin Info

- **Final Project Due: Dec 14, midnight EST - NO EXCEPTIONS**
- **Next Lecture: Final Projects Presentations: Dec 17**
- **Logistics:**
 - **All students that submitted Final Project will be presenting it**
 - **Presentation included: show YouTube video, 1 min for Q&A**
 - **If local - please come into the class!**
 - **If remote - please be available for Q&A online (if possible)**
- Lab this week: "Introduction to Apache Beam and Flink" by Rahul
- End of class (and video access) - Dec 26
- Evaluation Forms!

BigData Processing - Use Cases

Why review use cases?

How will we review them?

- Architecture overview
- Identify processing tiers
- Identify common concerns/ problems they were trying to solve
- Identify solution approaches
- Identify what type of the system (Type 1 through 4) is implemented

Use Case: Airbnb

Who/What is Airbnb? - **airbnb.com**

- supports over 100M users browsing over 2M listings
- Has an ability to intelligently make new travel suggestions

Data Scale Profile (2016 data)

- HDFS: ~11 Petabytes
- S3: similar

Airbnb

Book unique homes and
experience a city like a local.

🔍 Try "Beijing"

Search

Explore Airbnb



Homes

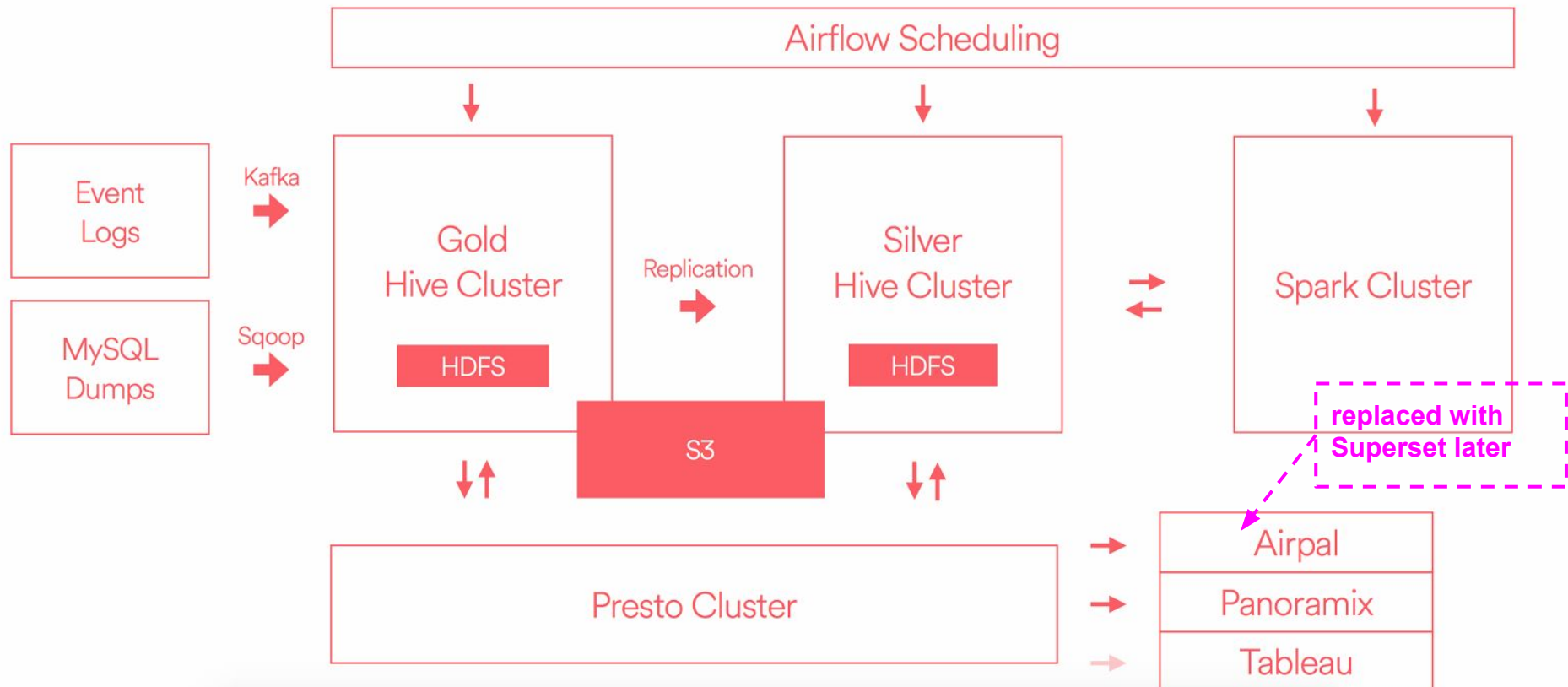


Experiences



Restaurants

AIRBNB DATA INFRA



Use Case: Airbnb

Main Components/Tiers

Data Sources

- Application log events sent to Kafka
- RDS SQL DB dumps via Scoop

Data Storage: Two HDFS with Hive clusters: "Gold" and "Silver"

- Input data is first stored into the Gold cluster, then replicated into the Silver one
- Why two clusters:
 - Isolation of the compute and storage resources
 - Disaster recovery (HA)
 - "source-of-truth" data on Gold cluster - only critical jobs with strict SLAs
 - free-for-all exploration on the Silver cluster - all ML jobs/ reporting, ad-hoc queries run there

S3:

- Used to off-load data for long-term storage
- Can be used as a backend by Hive tables to run historical queries

Use Case: Airbnb: The Data Manifesto

Data is Gold!

Hive tables are treated as the central source and sink for data.

The most important principles used to guarantee data accuracy/quality and consistency:

- All data is immutable
- All derivations/variations of data are reproducible
- Presto over Hive is used for all ad-hoc reporting

"we discourage the proliferation of different data systems and do not want to maintain separate infrastructure that sits between our source data and our end user reporting. In our experience, these intermediate systems obfuscate sources of truth, increase burden for ETL management, and make it difficult to trace the lineage of a metric figure on a dashboard all the way back to the source data from which it was derived. **We do not run Oracle, Teradata, Vertica, Redshift, etc. and instead use Presto (later Druid) for almost all ad hoc queries on Hive managed tables**"

Use Case: Airbnb

Stream and Batch Processing Tiers:

- Spark (ML)
- Presto --> Druid (later)
 - Presto: an open source distributed SQL query engine for running interactive analytic queries against HDFS/Hive stores
- AirPal --> Superset
 - a web based query execution tool backed by Presto - built in-house and open sourced
 - main interface to run ad hoc SQL queries against the warehouse
 - <https://medium.com/airbnb-engineering/superset-scaling-data-access-and-visual-insights-at-airbnb-3ce3e9b88a7f>

Airflow:

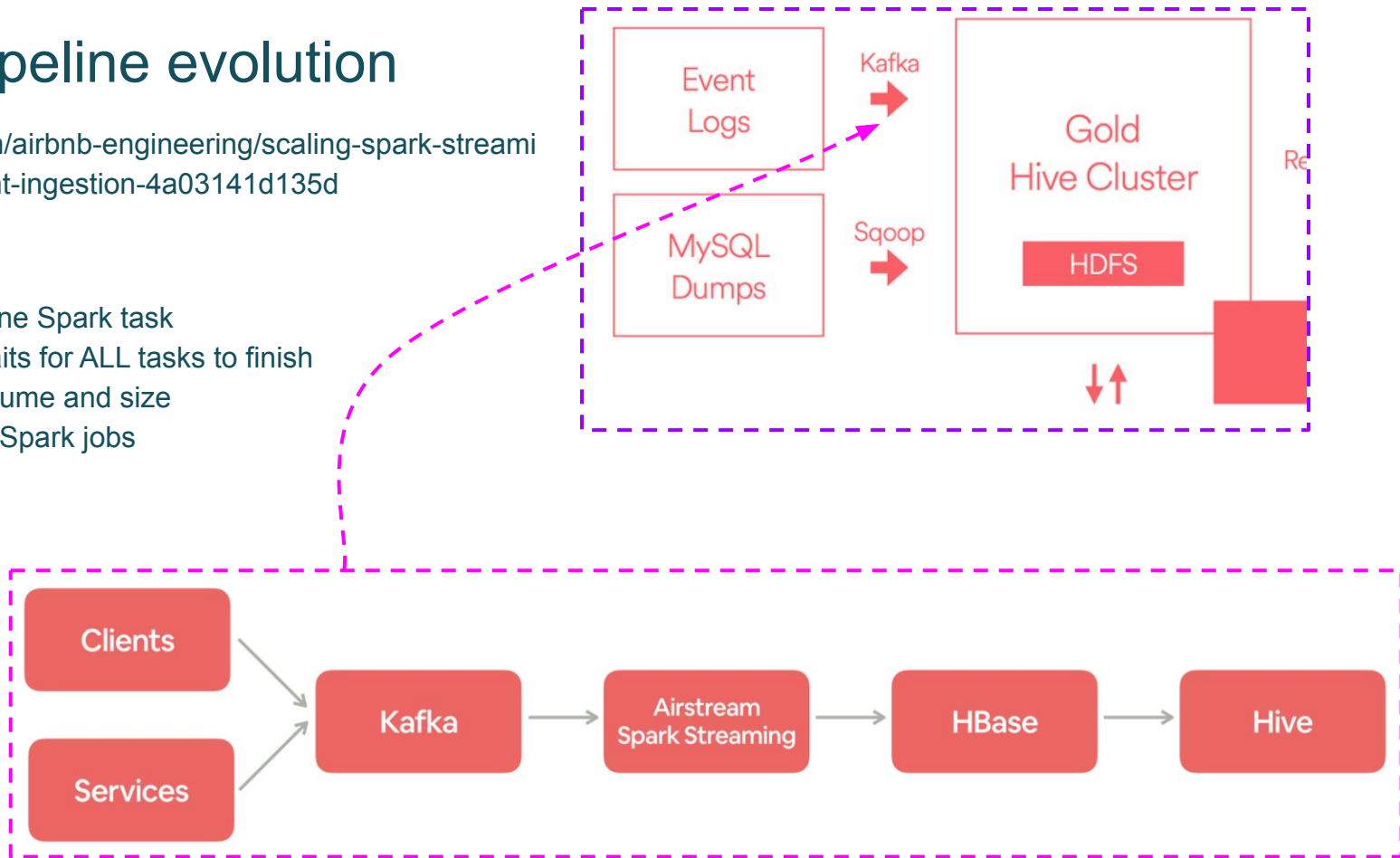
- a platform to programmatically author, schedule, and monitor data pipelines that can run jobs on Hive, Presto, Spark, MySQL, and more
- also built in-house and open sourced:
- the Airflow jobs run on the appropriate sets of clusters, machines, and workers

Airbnb: pipeline evolution

<https://medium.com/airbnb-engineering/scaling-spark-streaming-for-logging-event-ingestion-4a03141d135d>

Issues:

- one partition -> one Spark task
- one Spark job waits for ALL tasks to finish
- skew in event volume and size
- no headroom for Spark jobs to catch up



Airbnb: pipeline evolution

Solution:

custom Balanced Spark Kafka Reader:

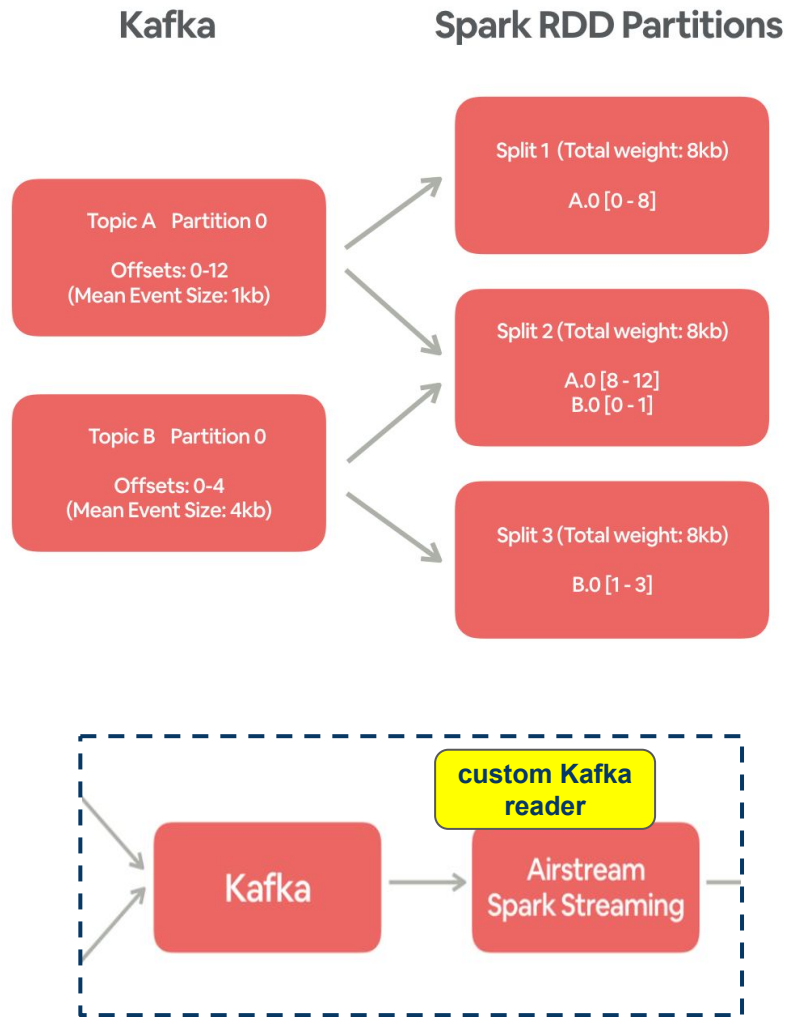
- allows arbitrary number of splits to increase parallelism
- calculates splits based on event volume and size

Compromise:

- ordering of events is not important

How it works:

- pre-computes AVG event size per topic, stores in CSV
- numberOfSplits is a parameter of Spark jobs
- for each topic: compute full offset range to process
- using **balanced partitioning algorithm** - compute and assign offset ranges for each Spark job split



Airbnb: pipeline evolution

2018: Scaling challenges with the 2016 pipeline architecture:

<https://medium.com/airbnb-engineering/druid-airbnb-data-platform-601c312f2a4c>

As data volume grew significantly - it started taking too long to run aggregate queries on ALL data in Hive using Presto - at query time

Why?

"Hive/Presto have to read all the data and aggregate them on demand, resulting in all necessary computation getting invoked at query time."

"Even if those engines are used to pre-compute the aggregation and store them, the storage format is not optimized for repeated slicing and dicing of data that analytics queries demand"

Airbnb: pipeline evolution

Solution: Introduce Druid as a middle-layer between raw HDFS storage and Dashboards

Overview



Analyze event streams

Druid provides fast analytical queries, at high concurrency, on both real-time and historical data. Druid is often used to power interactive UIs.



Utilize reimagined architecture

Druid is a new class of data store that combines ideas from [OLAP/analytic databases](#), [timeseries databases](#), and [search systems](#) to enable new use cases with streaming and batch data.



Build next generation data stacks

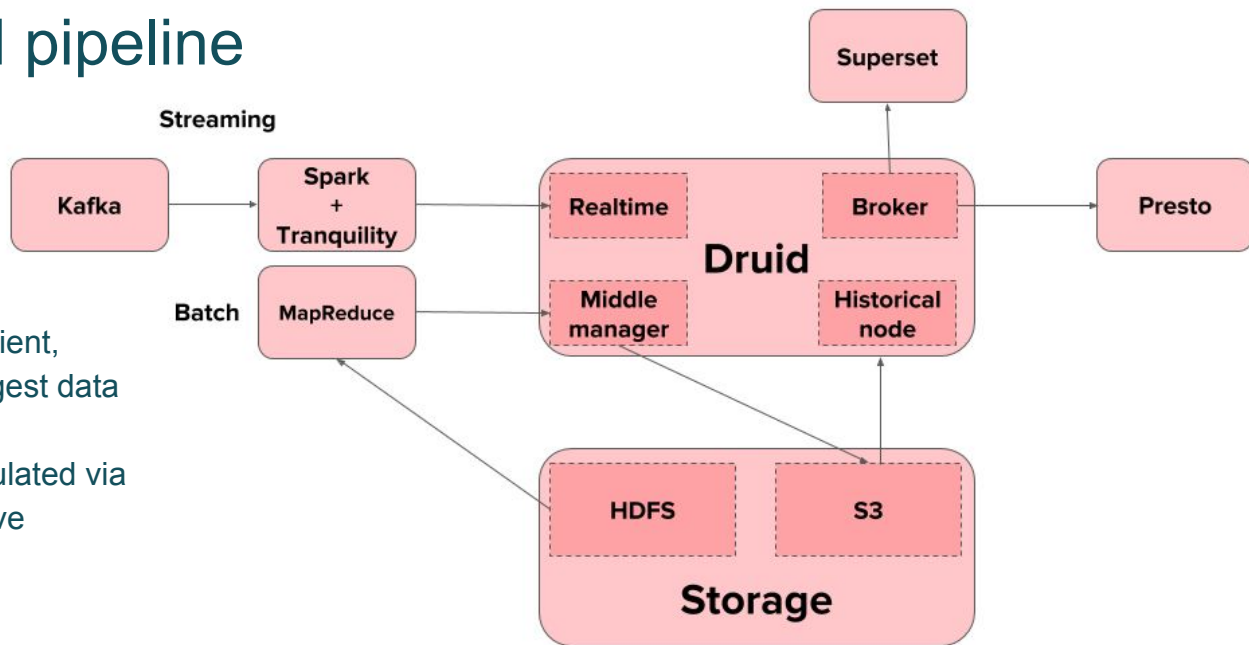
Druid integrates natively with message buses (Kafka, AWS Kinesis, etc) and data lakes (HDFS, AWS S3, etc).
Druid works especially well as a query layer for streaming data architectures.



Unlock new workflows

Druid is built for rapid, ad-hoc analytics on both real-time and historical data. Explain trends, explore data, and quickly iterate on queries to answer questions.

Airbnb: Druid-based pipeline



How it works:

- Spark Streaming jobs use Druid's client, Tranquility, to pre-aggregate and ingest data into Druid
- historical data: aggregates are calculated via batch jobs reading data from the Hive clusters - scheduled via Airflow

Two separate Druid cluster:

1. First: for critical pre-defined data sources and metrics: uses scheduled batched jobs over Hive
2. Second: for Real-Time data - ingested via Spark Streaming + Tranquility

Compromises:

- data sources (metrics/ dimensions) are pre-configured
- data is pre-aggregated at ingestion!

Airbnb: pipeline evolution

How it works: continue

How "Ad-Hoc" the actual "Ad-Hoc" is? :)

- there are many data sources - applications and service can produce data
- each application (users of) has to define how its data has to be aggregated to be ingested into Druid
- see example of such definition -->>
- once ingested into Druid - dashboards/other users can now run "ad-hoc" queries on this pre-aggregated data

so, is it truly Ad-Hoc??

NO !!

```
{
  name = "foo"
  owner = "foo@airbnb.com"
  dimensions = ["section", "operation"]
  metrics_spec = [
    {"type": "count", "name": "count"}
  ]

  batch_ingestion {
    ....
  }

  realtime_ingestion {
    source {
      type: air_events
      event_name: contact_us_flow
      event_data {
        section: string
        operation: string
      }
    }
  }
}
```

Airbnb: pipeline evolution

What about Ad-Hoc over **already collected** Historical data?

That's the real challenge!

- it requires re-run of MapReduce jobs over HDFS storage - very long-running
- why is it needed? users want to change dimensions/metrics
- how?
 - re-run of MapReduce jobs over HDFS storage - very long-running
 - deployment approach: "inactive" vs. "active" view of data in Druid
 -

Other challenges:

- Druid only supports ingestion/queries from one data source at a time
- joins are not supported
- use Presto for that

so, is it truly Ad-Hoc??

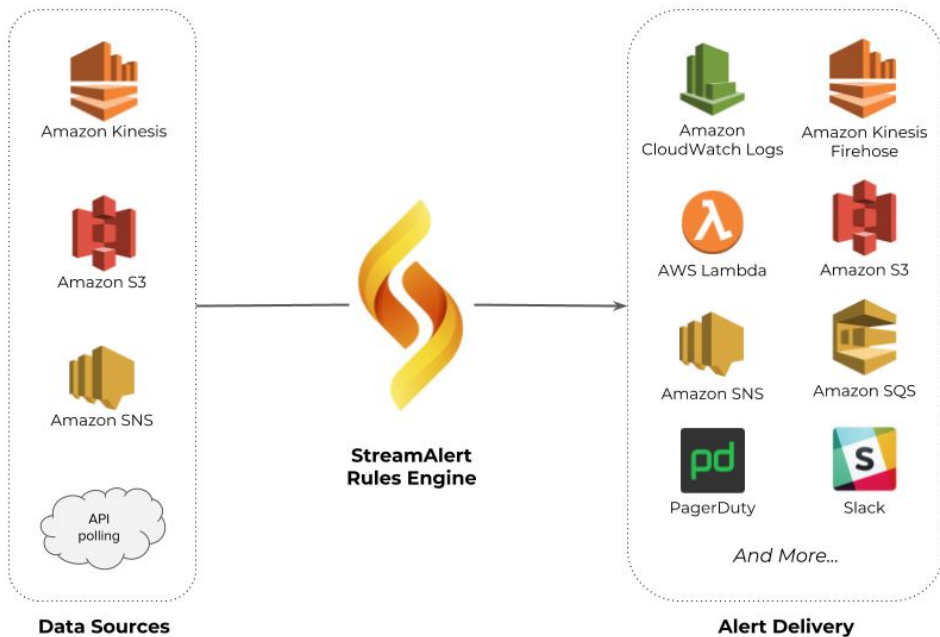
NO !!

Use Case: Airbnb: StreamAlert

<https://medium.com/airbnb-engineering/streamalert-real-time-data-analysis-and-alerting-e8619e3e5043>

<https://streamalert.io/en/stable/>

*StreamAlert is a **serverless**, real-time data analysis framework which empowers you to ingest, analyze, and alert on data from any environment, using data sources and alerting logic you define*



Use Case: Airbnb: StreamAlert

Functional Features:

- Rules are written in Python; they can utilize any Python libraries or functions
- Ingested logs and generated alerts can be retroactively searched for compliance and research
- Serverless design is cheaper, easier to maintain, and scales to terabytes per day
- Deployment is automated: simple, safe and repeatable for any AWS account
- Secure by design: least-privilege execution, containerized analysis, and encrypted data storage
- Merge similar alerts and automatically promote new rules if they are not too noisy
- Built-in support for dozens of log types and schemas
- Built-in collection of broadly applicable community rules
- Fully open source and customizable: add your own log schemas, rules, and alert outputs

Use Case: Airbnb: StreamAlert

Design points:

- Serverless—StreamAlert utilizes AWS Lambda, which means you don't have to manage, patch or harden any new servers
- Scalable—StreamAlert utilizes AWS Kinesis Streams, which will “scale from megabytes to terabytes per hour and from thousands to millions of PUT records per second”
- Automated—StreamAlert utilizes Terraform, which means infrastructure and supporting services are represented as code and deployed via automation
- Secure—StreamAlert uses secure transport (TLS), performs data analysis in a container/sandbox, segments data per your defined environments, and uses role-based access control (RBAC)
- Open Source—Anyone can use or contribute to StreamAlert

Recap of BDP System Types:

Type 1: Real-time [Limited range] + Ad-hoc queries systems:

These are systems that can answer any [ad-hoc] questions based on the **raw data** received during some small recent time window

Type 2: Full Historical + Limited Real-Time + Pre-defined queries systems:

systems that allows you to ask questions based on the historical pre-calculated data points (metrics) - not on all historical raw data - and on the limited latest real-time raw data

Type 3: Full Historical + No Real-Time + Ad-hoc queries systems:

systems that allow you to ask any question (ad-hoc query) using the stored raw data points, over the whole set of stored data - but no real-time up-to-the-second data

Type 4: Full Historical + Real-Time + Ad-hoc queries systems:

systems that can answer ANY question based on ALL data collected so far, including up-to-the-second data

Use Case: Airbnb

Lets analyze Airbnb architecture - what type of system is it?

??????

Answer: Type 1 + Type 2 + Type 3
Maybe: Type 4 if they combine results



Use Case: Netflix Storage Architecture

Time Series Data - Member Viewing History

<https://medium.com/netflix-techblog/scaling-time-series-data-storage-part-i-ec2b6d44ba39>

<https://medium.com/netflix-techblog/scaling-time-series-data-storage-part-ii-d67939655586>

We will see how Netflix has evolved a time series data storage architecture through multiple increases in scale !

Problem Statement:

- Netflix members watch over 140 million hours of content per day. Each member provides several data points while viewing a title and they are stored as viewing records. Netflix analyzes the viewing data and provides real time accurate bookmarks and personalized recommendations
- Netflix streaming has grown to 100M+ global members in its first 10 years
- Viewing history data increases along the following 3 dimensions:
 1. As time progresses, more viewing data is stored for each member.
 2. As member count grows, viewing data is stored for more members.
 3. As member monthly viewing hours increase, more viewing data is stored for each member.

Use Case: Netflix Storage Architecture

The first cloud-native version of the viewing history storage architecture used Cassandra for the following reasons:

- Cassandra has good support for [modelling time series data](#) wherein each row can have dynamic number of columns.
- The viewing history data write to read ratio is about 9:1. Since Cassandra is [highly efficient with writes](#), this write heavy workload is a good fit for Cassandra.
- Considering the [CAP theorem](#), the team favors eventual consistency over loss of availability. Cassandra supports this tradeoff via [tunable consistency](#).

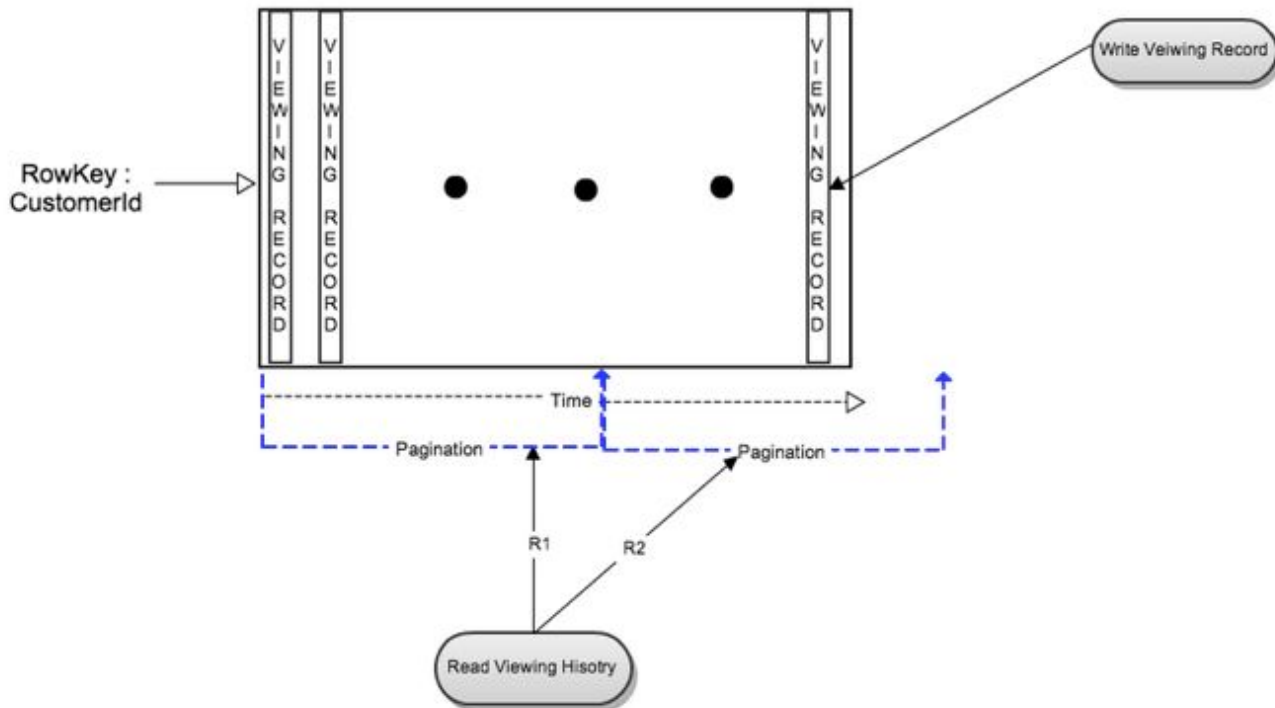
First table design:

- each member's viewing history was stored in Cassandra in a single row with **row key:CustomerId**.
- This horizontal partitioning enabled effective scaling with member growth and made the common use case of reading a member's entire viewing history very simple and efficient.

Use Case: Netflix Storage Architecture

Read/Write Flow: Version 1:

- single view record write - very fast
- full row read - get all data for user:
 - fast if row is small
 - slow otherwise
- time range query: inconsistent due to variable size
- full row read via pagination:
 - better for Cassandra
 - worse latency



Use Case: Netflix Storage Architecture

Issues as number of users and their viewing history grew:

- row size and overall data size increased
- high storage and operational costs
- slower performance for users with large viewing history:
 - performance of 'writes' is OK
 - performance of 'reads' is bad

Reasons for 'reads' latency increase for large rows (large viewing history):

- the number of SSTables increased
- since all data could not fit in memory - reads had to access both memtables and SSTables
- compaction, read repair and full column repair became much slower too

Use Case: Netflix Storage Architecture

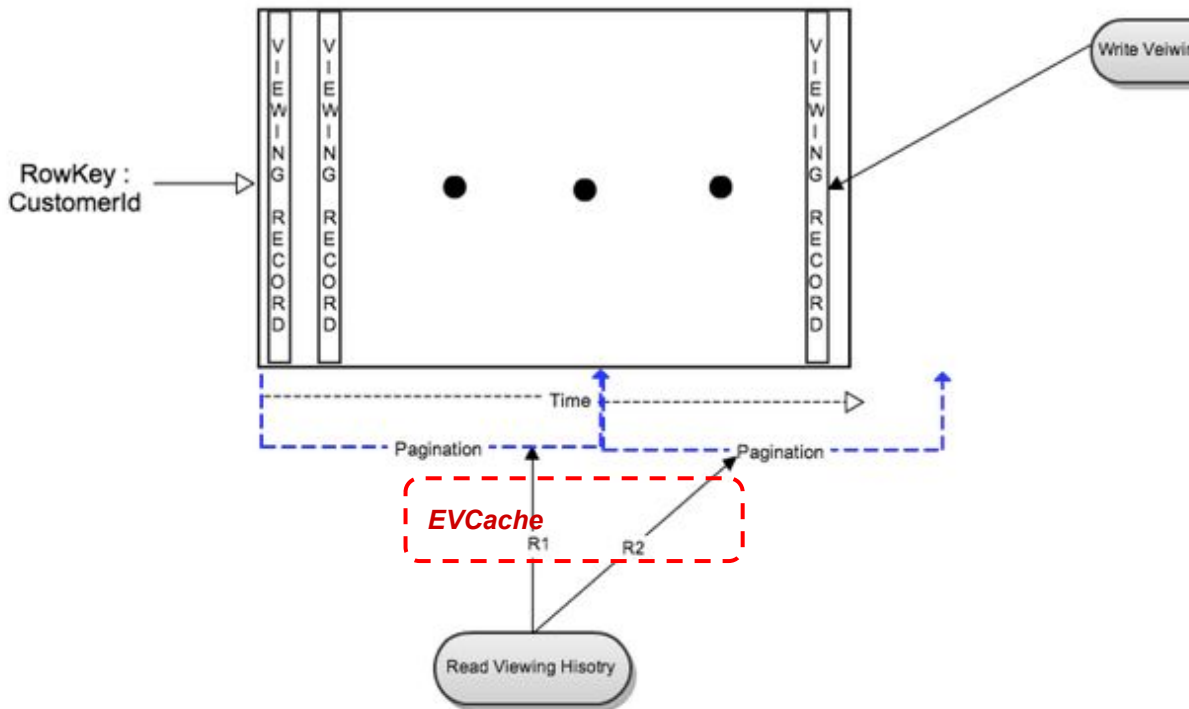
Design Change #1:

Add **in-memory caching layer** in front of Cassandra for recent data reads

- EVCache
- key: customerId
- value: compressed binary representation of the viewing data

This worked for a few years!

NOTE: this is a great example of using a distributed cache! Could be Redis as well



Use Case: Netflix Storage Architecture

Next stage:

As number of users and their viewable content grows - the same performance issues arise.

Goals for the next 5+ year of grows:

- smaller storage footprint
- consistent read/write performance

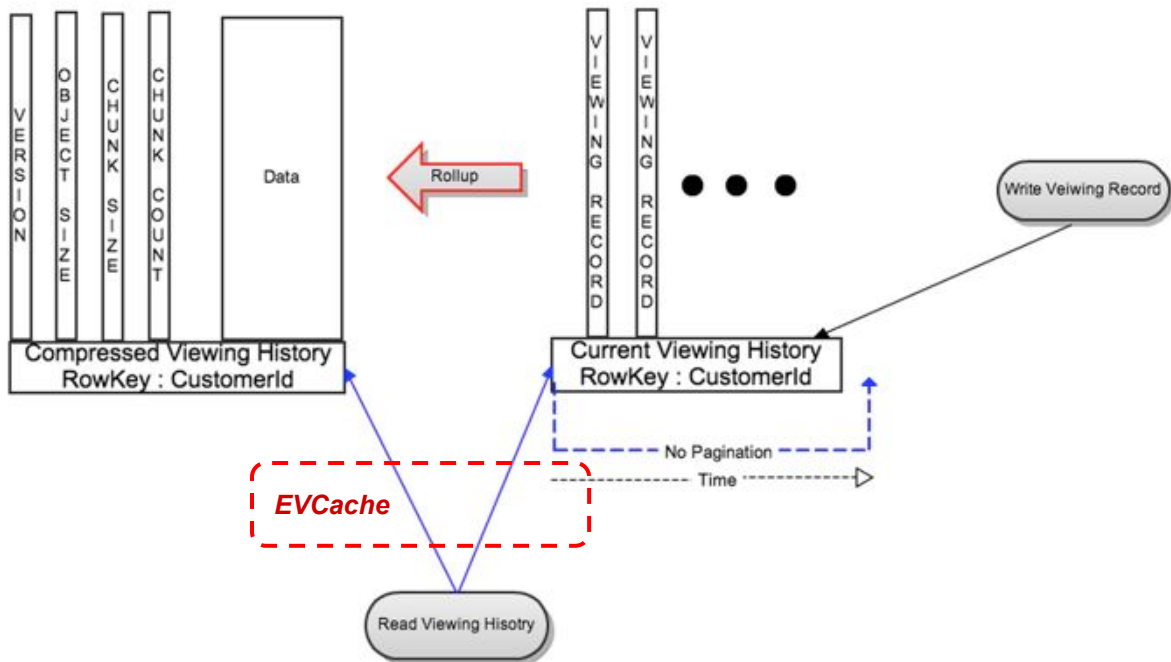
Solution - Design Change #2: **Separate Data into "cold" vs "hot" storage:**

- For each user, data is divided into two sets:
 - Live View History (LiveVH): small number of recent data, with frequent reads and updates; uncompressed
 - Compressed/Archived Viewing History (CompressedVH): large number of older records with infrequent reads and updates; compressed and stores as one column value per key (same key: customerID)
- LiveVH and CompressedVH are stored in different tables and tuned differently
- LiveVH:
 - frequent compactions, repair and full column repair - for consistency
 - small "gc_grace_period" to decrease number of SSTables
- CompressedVH:
 - manual or infrequent compactions
 - consistency checks are done during rare updates

Use Case: Netflix Storage Architecture

Read/Write/Update paths:

- Write - no changes
- Reads:
 - Recent data: from LiveVH only, most frequent
 - Full history: parallel reads from LiveVH and CompressedVH
- Updates:
 - as LiveVH is read - if it exceeds predefined MAX size - compress and store into CompressedVH
 - same key, new version
 - old versions are removed



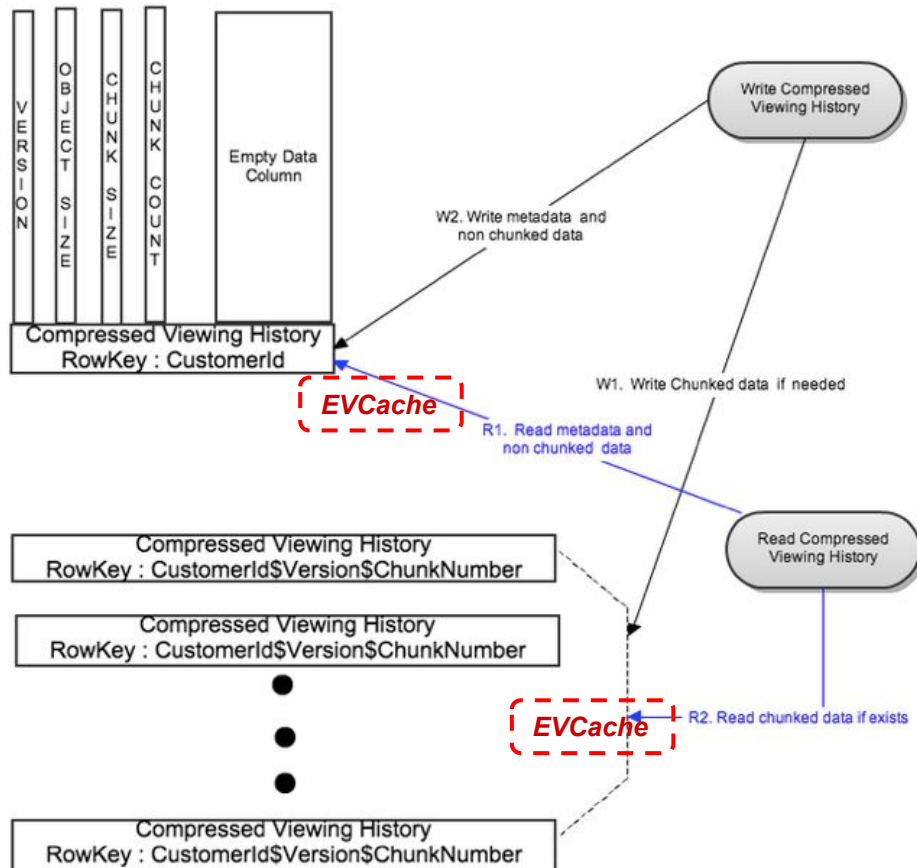
Use Case: Netflix Storage Architecture

Next Scaling Challenge:

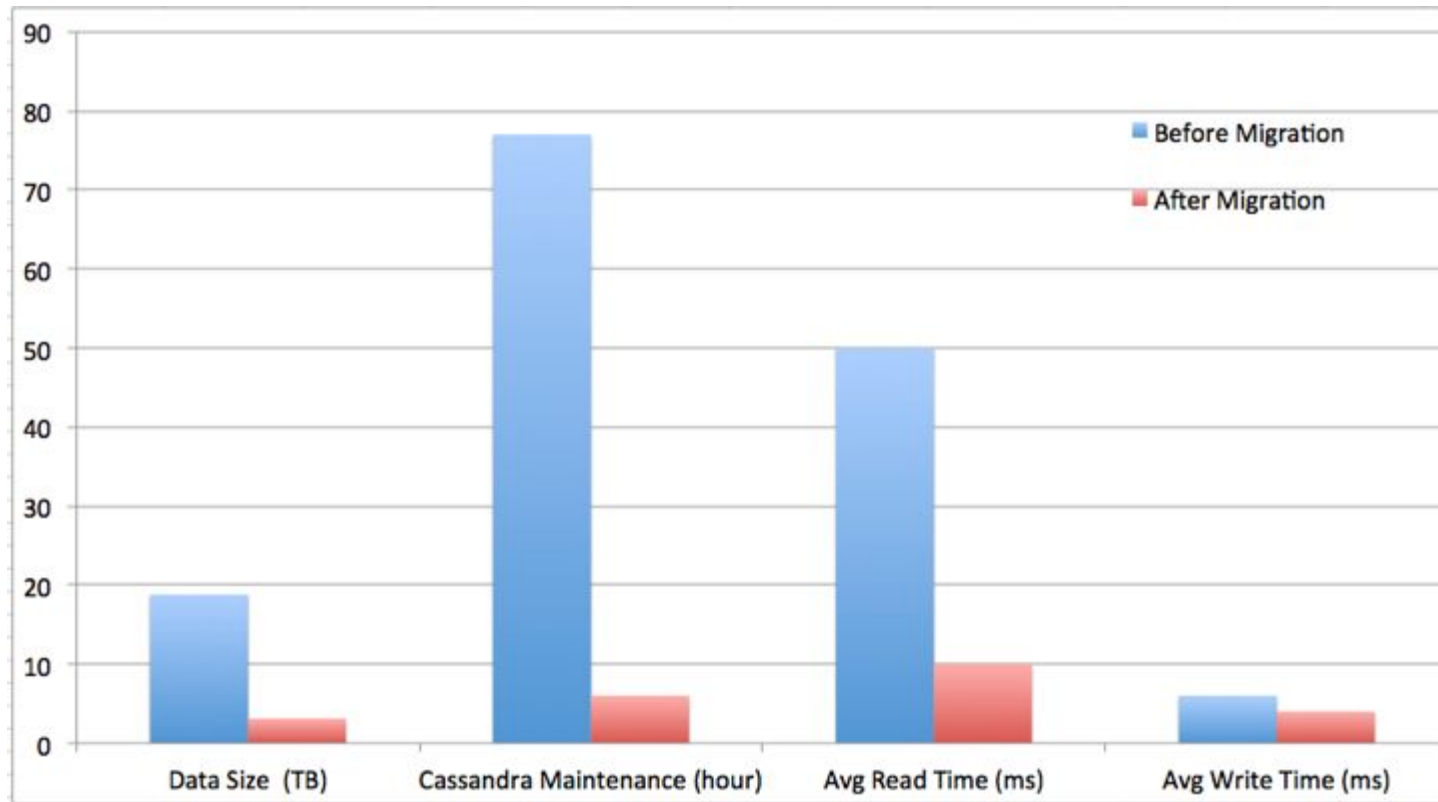
For users with huge history data - one compressed column per key was causing read/write latency issues

Solution - Design Change #3:

- Split the CompressedVH data into multiple chunks if the size of one chunk exceeds some limit
- Store chunks on different Cassandra nodes - to increase parallelization of reads, by using a different key: **key = customerID+version+chunkNumber**
- EVCache: same changes: split stored compressed value into chunks



Use Case: Netflix Storage Architecture - Results:



@Marina Popova

Use Case: Netflix Storage Architecture

Next Scaling Challenges:


- have to support 5x load
- existing design won't scale that far

Analyses:

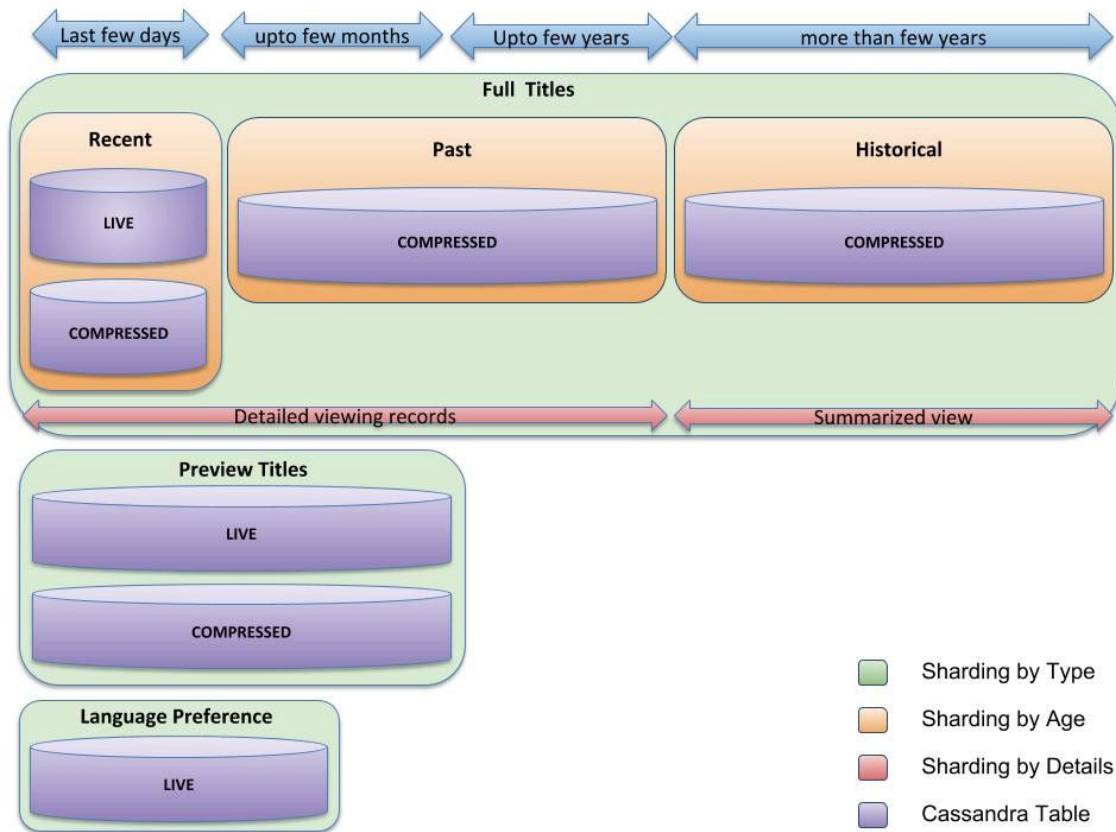
- different access patterns for different categories of data:
 - full title plays
 - video previews
- most details are needed for recent data; less - for older

Solution - Design Change #4: **Separate Data into "cold" vs "hot" storage** - by type/ age/ level of details

- Type/Categories
 - shard (partition) by data type (full vs preview vs other)
 - include min number of details
- Age/ level of details:
 - partition by age
 - recent data: expire after TTL - full details
 - historical data: summarize and compress into archived storage

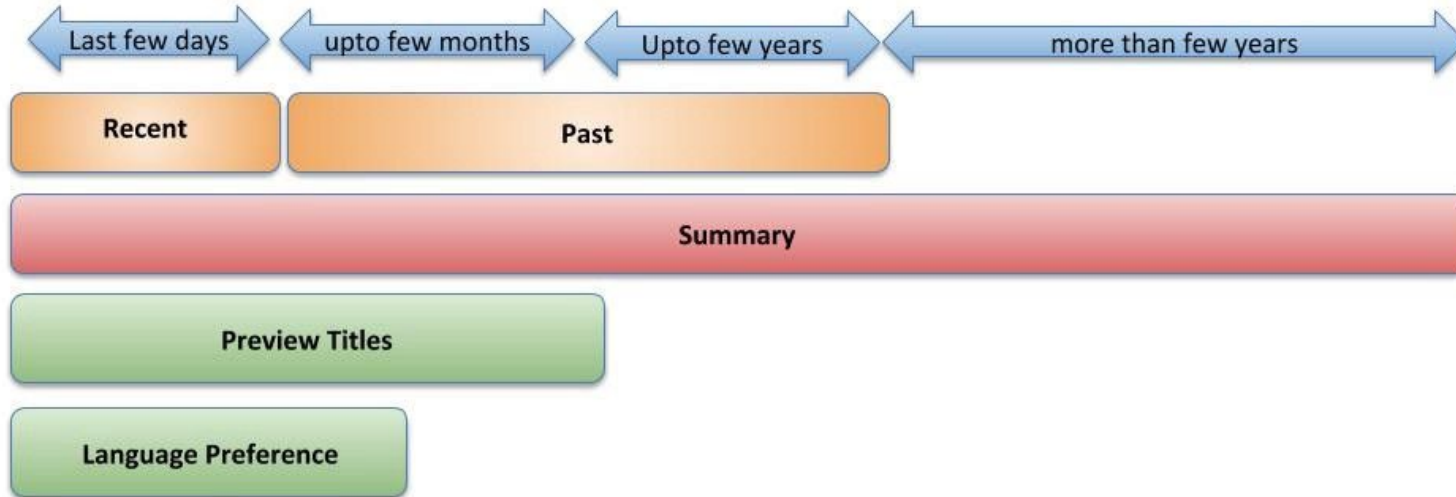
Category	Age	Level of detail
Video previews	Recent	Filtered
Full title plays	Recent	Complete
Full title plays	Historical	Summary
Language Preference 	Recent	Deltas

Use Case: Netflix Storage Architecture



Use Case: Netflix Storage Architecture

Caching Layer changes:



Use Case: Netflix Storage Architecture

Results:

- big improvements in the Cassandra operational metrics - based on the new sharding/partitioning
- data storage cost reduction - due to less details/more compression
- more to come!

on to Pipelines next ...

Use Case: Netflix Data Processing Pipelines

<https://medium.com/netflix-techblog/evolution-of-the-netflix-data-pipeline-da246ca36905>

Almost every application at Netflix uses the main data pipeline:

2016-2017:

~500 billion events and ~1.3 PB per day

~8 million events and ~24 GB per second during peak hours

There are several hundred event streams flowing through the pipeline. For example:

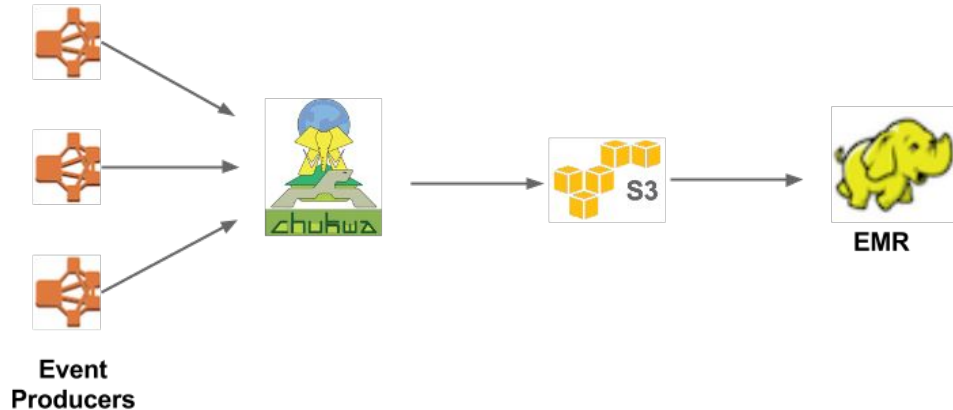
- Video viewing activities
- UI activities
- Error logs
- Performance events
- Troubleshooting & diagnostic events

Use Case: Netflix

V1.0

Original Chukwa data pipeline: aggregate and upload events to Hadoop/Hive for batch processing.

- **Chukwa**: collect events and write them to **S3** in Hadoop sequence file format
- batch jobs process those S3 files and write to **Hive in Parquet format**
- end-to-end latency is up to 10 minutes
- that is sufficient for batch jobs which usually scan data at daily or hourly frequency.

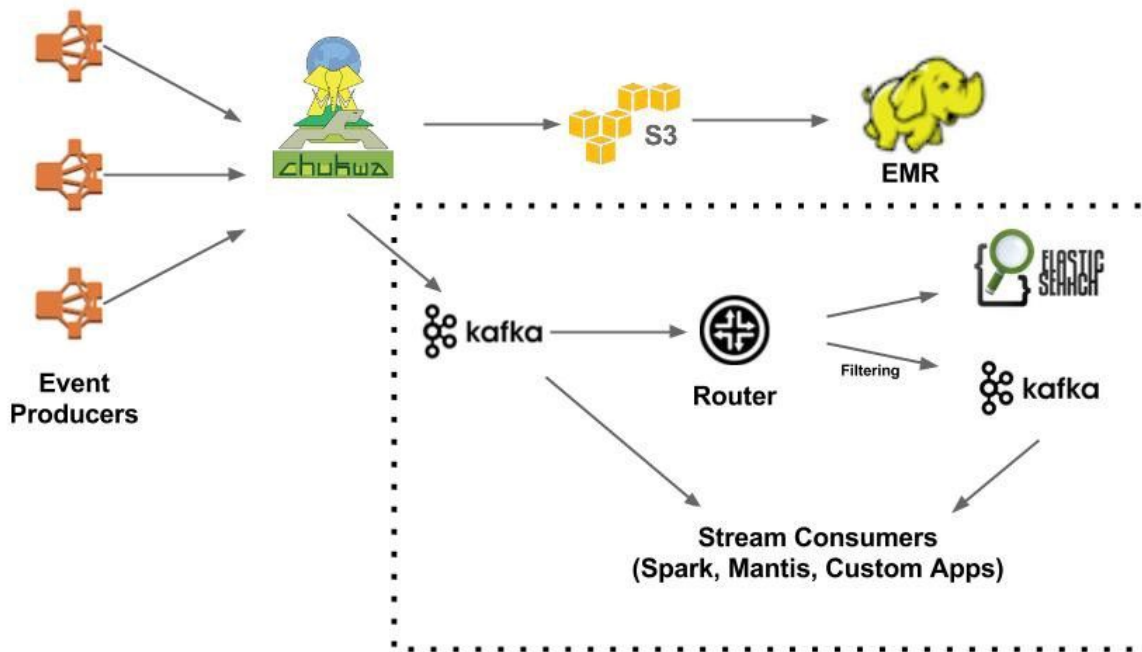


Netflix Pipelines: V1.5 - adding real-time analytics

Added Kafka and ElasticSearch

Problem areas: Chukwa

- Kafka implements replication that improves durability
- Chukwa doesn't support replication
- Simplify the architecture



Netflix Pipelines: V2.0 - Keystone(Kafka-based) pipeline

Data Collection:

- custom Java apps write events to Kafka directly
- send to an HTTP proxy which then writes to Kafka.

Messaging Tier: (Data Buffering)

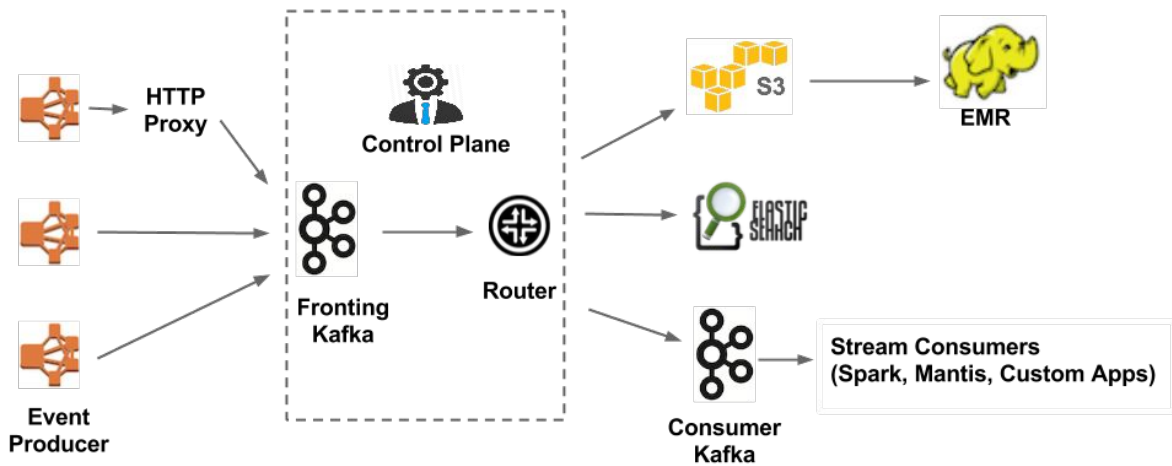
- Kafka serves as the replicated persistent message queue
- ibuffer for tmp outages from downstream sinks

Data Routing and Batch/Stream Processing:

- The routing service (custom app) is responsible for moving data from fronting Kafka to various sinks: S3, Elasticsearch, and secondary Kafka
- EMR

Analytics/ Visualization:

- ElasticSearch



Netflix: Continuous Deployment

Goals and Best Practices:

<https://medium.com/@NetflixTechBlog/tips-for-high-availability-be0472f2599c>

- Red/Black deployment strategy
- "Chaos Monkey"
- Canary Analyses with Kayenta: <https://github.com/spinnaker/kayenta>
- Deployment is successful only when all services are healthy

Spinnaker is an open source, multi-cloud continuous delivery platform for releasing software changes with high velocity and confidence.

Created at Netflix, it has been battle-tested in production by hundreds of teams over millions of deployments. It combines a powerful and flexible pipeline management system with integrations to the major cloud providers.



Netflix Pipelines Evolution: 2018 - now

<https://medium.com/netflix-techblog/keystone-real-time-stream-processing-platform-a3ee651812a>

Moving to Flink for stream processing and a Self-Serve approach to job definitions, scheduling and monitoring

Scale:

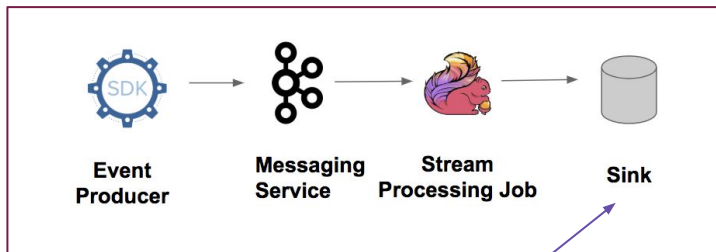
- 130M+ subscribers from
- 190+ countries
- trillions of events per day
- Petabytes of data per day

Today, the Keystone platform offers two production services:

- **Data Pipeline:** streaming enabled Routing Service and Kafka enabled Messaging Service, together is responsible for producing, collecting, processing, aggregating, and moving all microservice events in near real-time.
- **Stream Processing as a Service (SPaaS):** enables users to build & operate custom managed stream processing applications, allowing them to focus on business application logic while platform provides the scale, operations, and domain expertise.

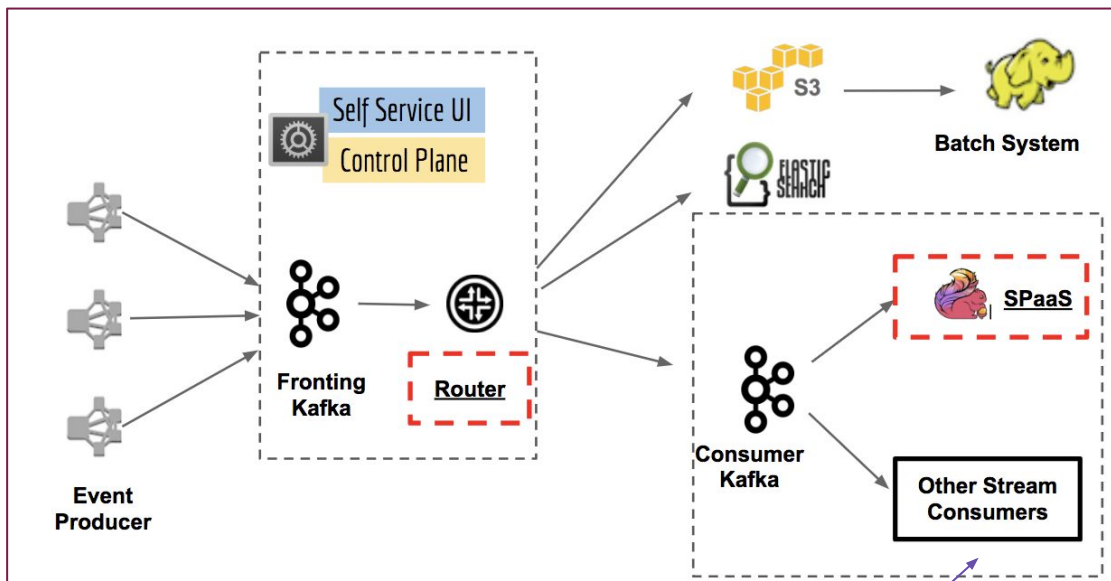
Netflix Pipelines Evolution: 2018 - now

Anatomy of a single streaming job:



- *Cassandra*
- *ElasticSearch*
- *RDS*
- *MySQL*
- ...

... and the platform is managing these jobs:



- *Spark ML/ Batch*
- *Presto*
- ...

@Marina Popova

Netflix Pipelines: 2018: Requirements and Goals

Job declaration and configuration:

- users define delivery route: source and sinks
- users can define filters and aggregations
- users can choose tradeoffs between:
 - latency vs duplicates
 - strict ordering vs not ordered
 - consistency vs availability
 - exactly-once vs at-least-once delivery
- data re-processing semantics

Operational Requirements/Goals:

- thousands of user-defined streaming jobs running in parallel!
- multi-tenancy: jobs have to be isolated from each other (resource and state-wise)
 - each job maintains its own non-shared state
 -

Main Design Points:

- users use UI to declare desired job attributes, source and sinks
- this is stored as the "source of truth" state in AWS RDS
- the platform generates a **Flink template** for the job
- the platform then schedules, coordinates and monitors the execution of the job
- if there are any issues - the system tries to recover the job using the "source of truth" job state

Netflix Pipelines: 2018: Job Config UI Examples

Keystone Self Service Streams Kafka-share Streaming Jobs Admin Help

Streaming Jobs » spaaszxu-testapp

CONFIG DEPLOYMENTS

0.0.3-rc.1-candidate

example-kafka-source Job dynamicsink

Job

Properties

Resources

Security Groups

PROPERTY

source	Value
spaas.spaas-upgrade.source.example-kafka-source.kafka.consumer.group.id	spaasjob_spaas-upgrade_zou
spaas.spaas-upgrade.source.example-kafka-source.kafka.consumer.max.partition.fetch	10002000
spaas.spaas-upgrade.source.example-kafka-source.kafka.consumer.metric.reporters	com.netflix.spaas.metrics.KafkaMetricsReporter
spaas.spaas-upgrade.source.example-kafka-source.kafka.sampling.enabled	true
spaas.spaas-upgrade.source.example-kafka-source.kafka.sampling.percentage	0.01

controlling source configuration

controlling job's re-processing settings

Keystone Self Service Streams Kafka-share Streaming Jobs Admin Help

Streaming Jobs » spaaszxu-testapp

Job spaaszxu-testapp updated successfully

Owner: zxu@netflix.com

PROD

Image Version: 0.0.3-rc.1-candidate

Job

Properties

Resources

Security Groups

Deploy Streaming Job: spaaszxu-testapp

Please choose how to deploy this job.

- ☒ Restart from existing checkpoint
- ☐ Restart from existing savepoint
- ☐ Take a new savepoint and restart
- ☐ Start new job without existing state

Choose an existing checkpoint to deploy from:

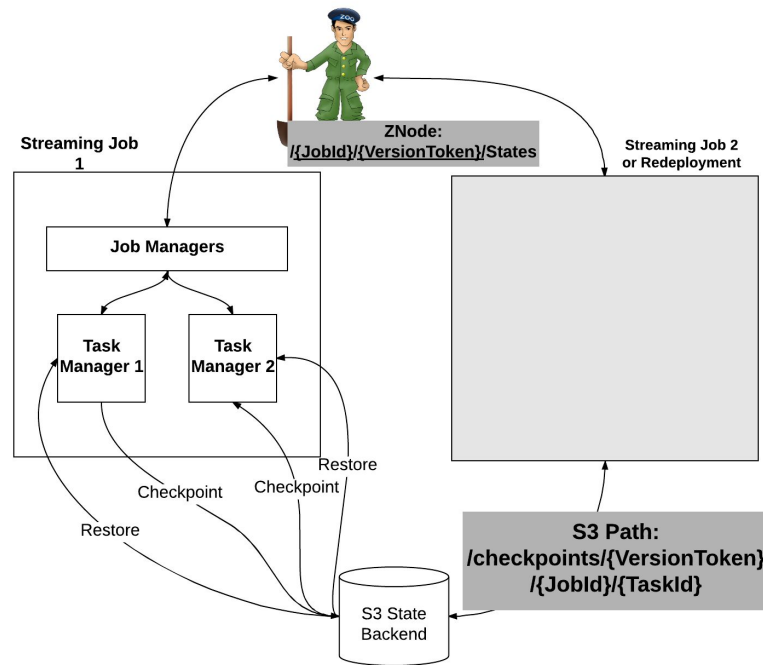
Time	Path
06/01/2018 03:27:31 PM	s3://eu-west-1.spaas.prod/spaaszxu-testapp/extCheckpoints/checkpoint_metadata-87ae64a80bc7
06/01/2018 03:22:30 PM	s3://eu-west-1.spaas.prod/spaaszxu-testapp/extCheckpoints/checkpoint_metadata-836c0a884651
06/01/2018 03:17:30 PM	s3://eu-west-1.spaas.prod/spaaszxu-testapp/extCheckpoints/checkpoint_metadata-3e79180aa94b
06/01/2018 03:12:34 PM	s3://eu-west-1.spaas.prod/spaaszxu-testapp/extCheckpoints/checkpoint_metadata-27a9108752c5

Resume from checkpoint: s3://eu-west-1.spaas.prod/spaaszxu-testapp/extCheckpoints/checkpoint_metadata-87ae64a80bc7

Cancel Deploy

Netflix Pipelines: 2018: SPaaS

- jobs are schedules and deployed via Spinnaker
- the jobs are deployed on the Titus container management platform:
<https://medium.com/netflix-techblog/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436>
 - based off of Mesos
 - tightly integrated with AWS security, IAM, EC2 resource management, AWS Auto Scaling, etc.
- for each job - an independent Flink cluster is created!
 - the only shared resources: ZK for job coordination and S3 for job checkpoints state



Use Case: Netflix

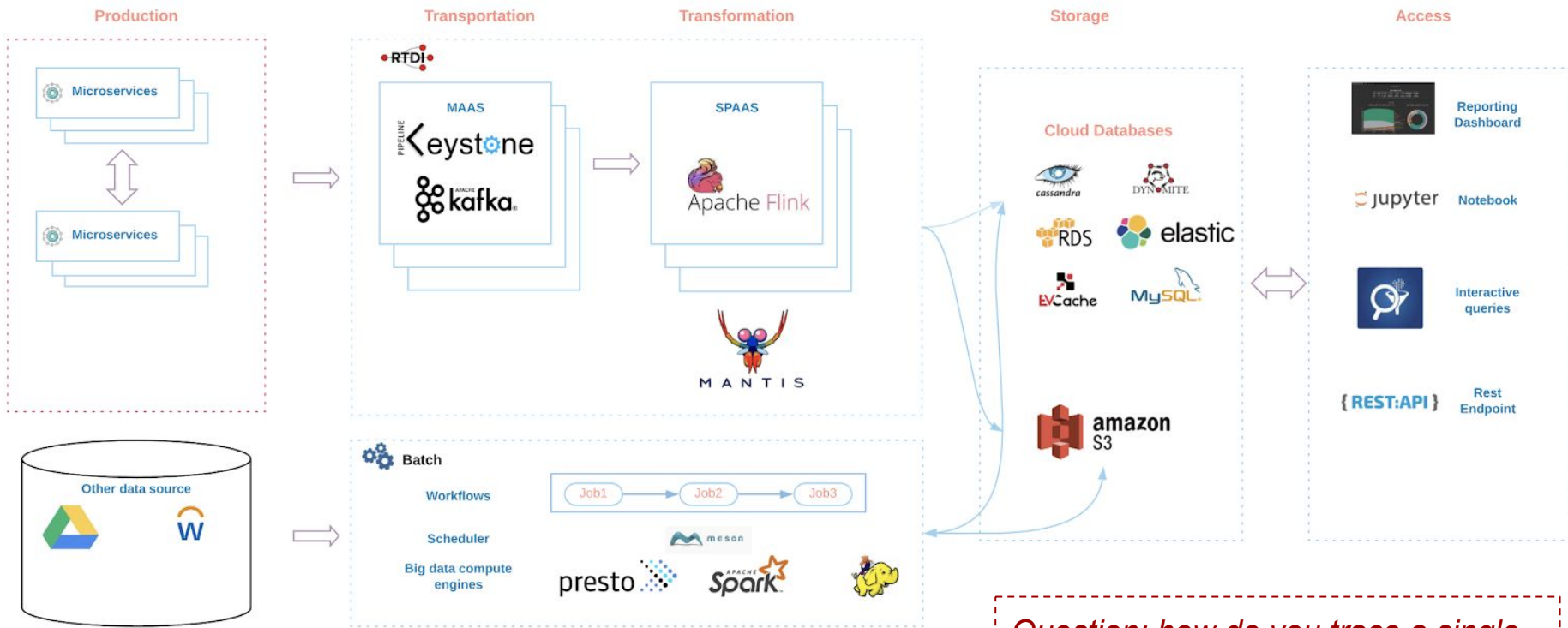
Lets analyze Netflix architecture:

What type of system is it?



Netflix: Architecture Evolution: 2019: Data Lineage

<https://medium.com/netflix-techblog/building-and-scaling-data-lineage-at-netflix-to-improve-data-infrastructure-reliability-and-1a52526a7977>



@Marina Popova

Question: how do you trace a single event through this mess? :)

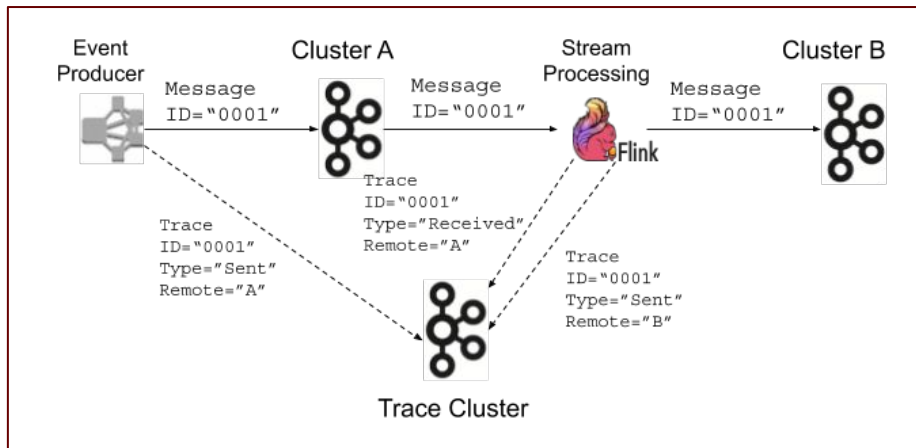
Netflix: 2019: Data Lineage

Goals:

generate “traces” for messages being transported in our system and analyze those traces to detect loss and derive the metrics we need

The trace messages will include the following attributes

- ID of the message being traced
- Location where the trace is generated
- Corresponding Kafka cluster name
- Trace type: Sent or Received
- Timestamp
- Kafka TopicPartition and offset for the message being traced
- Any custom attributes that can help to recover the message if it is identified as lost, for example, row ID in RDBMS



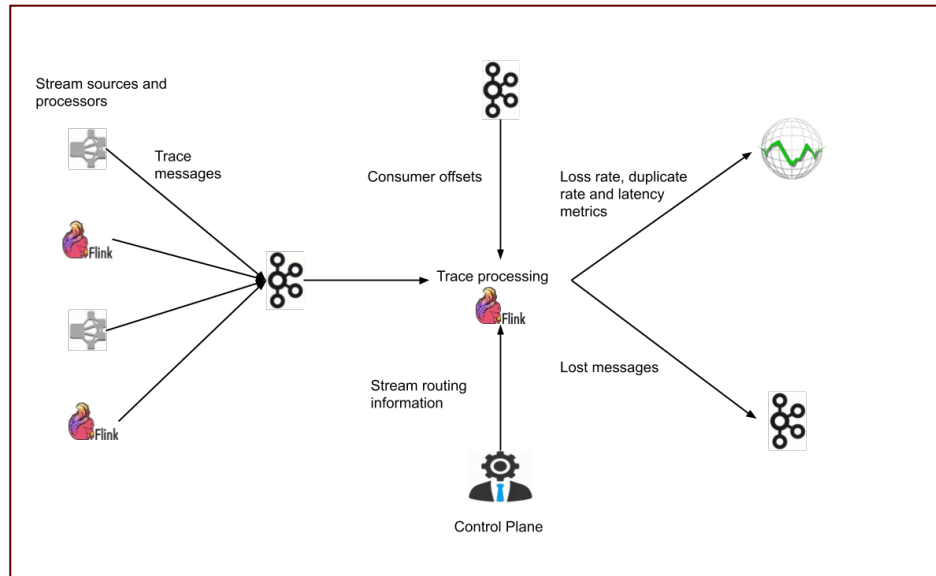
Netflix: 2019: Data Lineage

Traces are generated and sent to Kafka from all components in the pipelines.

The delivery of traces has to be very reliable. To ensure that the following measures were taken:

- Kafka cluster for traces is isolated from other clusters and over provisioned
- The cluster is configured for high durability and consistency
- three replicas with two minimal in sync replicas
- unclean leader election disabled
- broker instances backed by AWS EBS
- ack=all for trace producers to ensure consistency

Trace system architecture:



A Few Bits and Pieces ...

A few quick examples of specific problems/solutions solved in BDPs in different companies ...



Lyft: the Elasticsearch Journey

<https://www.elastic.co/blog/operational-logging-at-lyft-migrating-from-splunk-cloud-to-amazon-es-to-self-managed-elasticsearch>

About Lyft

In 2017:

- 375.5M rides given (up from 162.5M in 2016)
- >2,000 drop-offs/sec Halloween 2017
- >2M rides given NYE 2017
- ~2,100 employees (up from ~1,100); >700 engineers
- 200+ microservices
- 10,000+ EC2 instances

= lots of logs

• Splunk Cloud

– Pro:

- Powerful query language
- No predefined schema

– Con:

- ~14 days retention
- High load ⇒ ingest backs up (logs up to 30 minutes late)
- \$\$\$

• Splunk contract up for renewal Oct 2016

==>> move to ES instead
Managed AWS ES first ... ==>>

Lyft: the Elasticsearch Journey

Not the best experience with Managed AWS ES:

- ES cluster size limits
- Red cluster state! - when any node overloads/goes down
 - no access to re-start/re-create nodes
 - have to open support case

Solution: move to self-managed ES !

Apologia Pro Vita Sua AWS Elasticsearch

What AWS Elasticsearch is:

- Push-button solution
- Great for many use cases

What it isn't: a fully functional Elasticsearch cluster

- The whole thing is behind a gateway
 - Round-robin load balancer
 - 60s timeout (on everything)
- Most APIs are obfuscated
- Configuration change ⇒ whole new cluster

You have opened a new Support case

- 1. Open a support ticket
 - Wait (sometimes for hours) (during business hours)
 - First-line support: “I see that your cluster is red”
 - “Please give us the output of these API endpoints ...”
- 2. Escalate to ES team engineers
 - “We see that one of your nodes needs to be shot”
 - “We see JVM memory pressure is high, please try to reduce it”
 - “Can you maybe stop logging so much?”
 - Wait some more
- 3. Expedite, option 1: call the TAM
 - Eventually started going directly through TAM to engineers, who knew the routine



Uber: AresDB for Real-Time Analytics

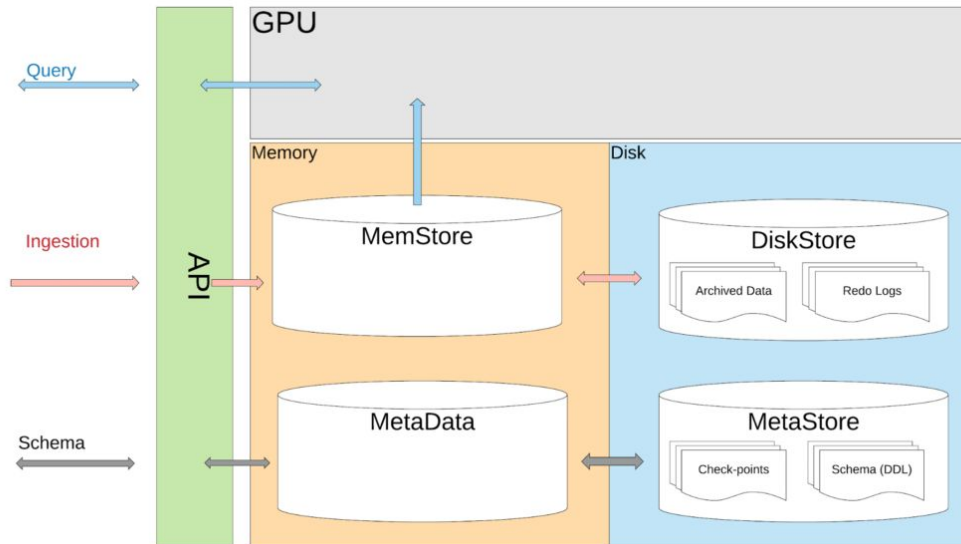
<https://eng.uber.com/aresdb/>

Needs:

- time-series analytics - **fast** ad-hoc aggregations
- support for joins and complex GEO-queries

Main design points:

- in-memory DB solution
- pure Vertical scaling - single server setup
- use GPU-based computing for massively parallel query execution
- utilize CUDA for query execution
- most data is stored in host memory
- at query time, Ares transfers data from the host memory into GPU's memory
- Columnar-based storage with compression on disk



Uber: Hudi for Large-Scale Analytics

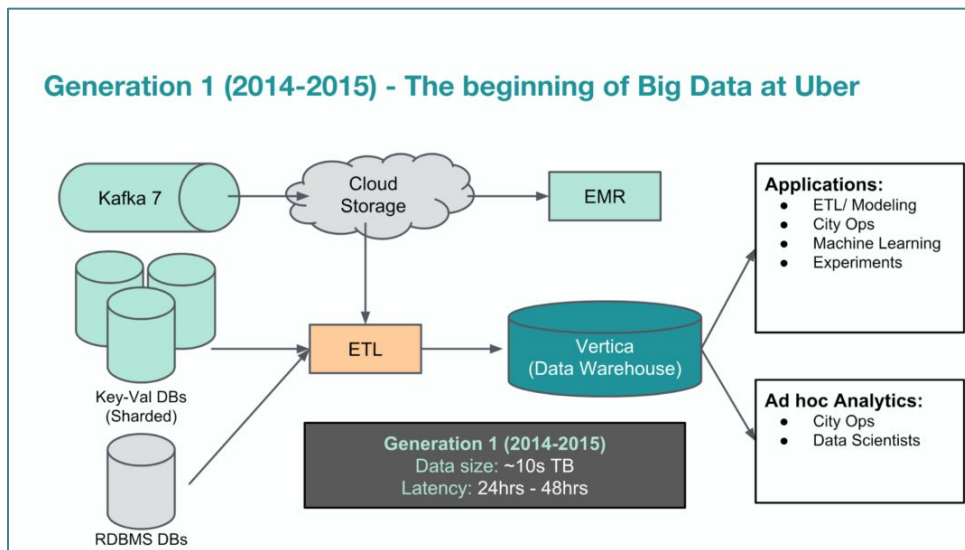
<https://eng.uber.com/uber-big-data-platform/>

Needs: Ad-hoc analytics across Historical data

Solution: one huge Vertica-based Data Lake

"The first generation of Uber's Big Data platform allowed us to aggregate all of Uber's data in one place and provide standard SQL interface for users to access data"

Issues: Not Scalable!



Uber: V2 with Hadoop

V2: Move to Hadoop:

- Presto for Ad-Hoc queries
- Spark for programmatic access to raw data
- Hive for super-huge queries

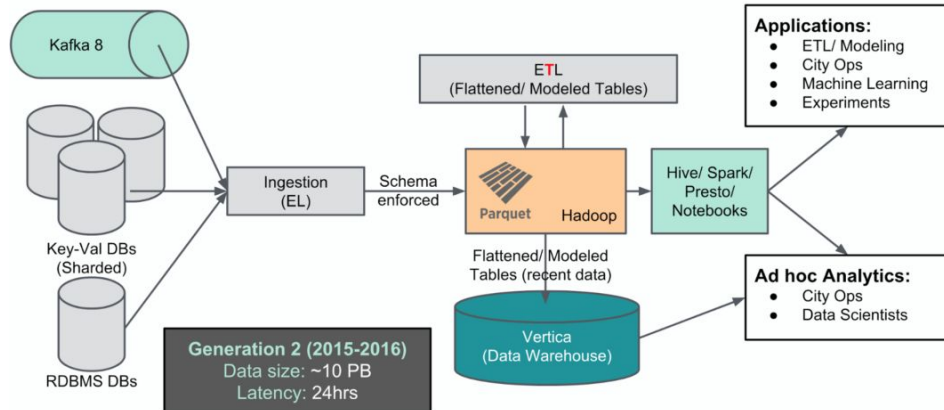
"The second generation of our Big Data platform leveraged Hadoop to enable horizontal scaling"

Issues: very high data latency - 24 hrs !

"While Hadoop enabled the storage of several petabytes of data in our Big Data platform, the latency for new data was still over one day, a lag due to the snapshot-based ingestion of large, upstream source tables that take several hours to process"

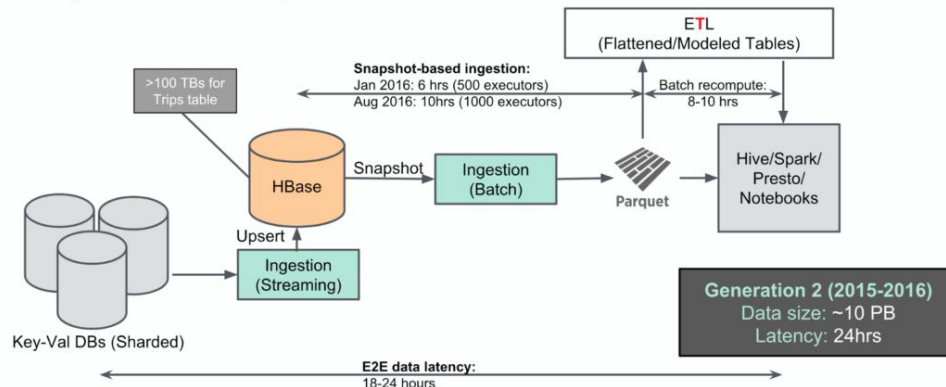
@Marina Popova

Generation 2 (2015-2016) - The arrival of Hadoop



Generation 2 (2015-2016) - The arrival of Hadoop

Why does data latency remain at 24 hours?

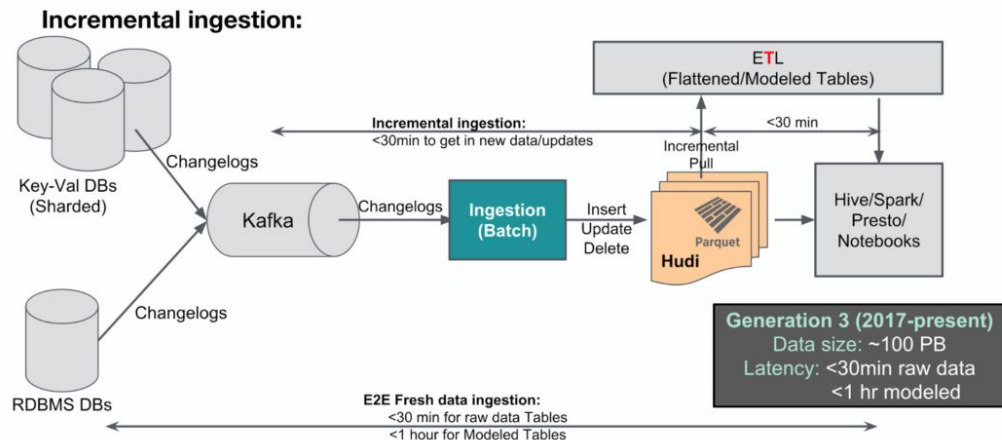


Uber: Hudi V3

V3: Move to "Hudi":

- an open source Spark library that provides an abstraction layer on top of HDFS and Parquet
- can be used from any Spark job, is horizontally scalable, and only relies on HDFS to operate
- enables us to update, insert, and delete existing Parquet data in Hadoop
- allows users to incrementally pull out only changed data, significantly improving query efficiency and allowing for incremental updates of derived modeled tables

Generation 3 (2017-present) - Let's rebuild for long term



"The third generation of our Big Data platform incorporates faster, incremental data ingestion (using our open source [Marmaray](#) framework), as well as more efficient storage and serving of data via our open source [Hudi](#) library"

What Have We Learned?

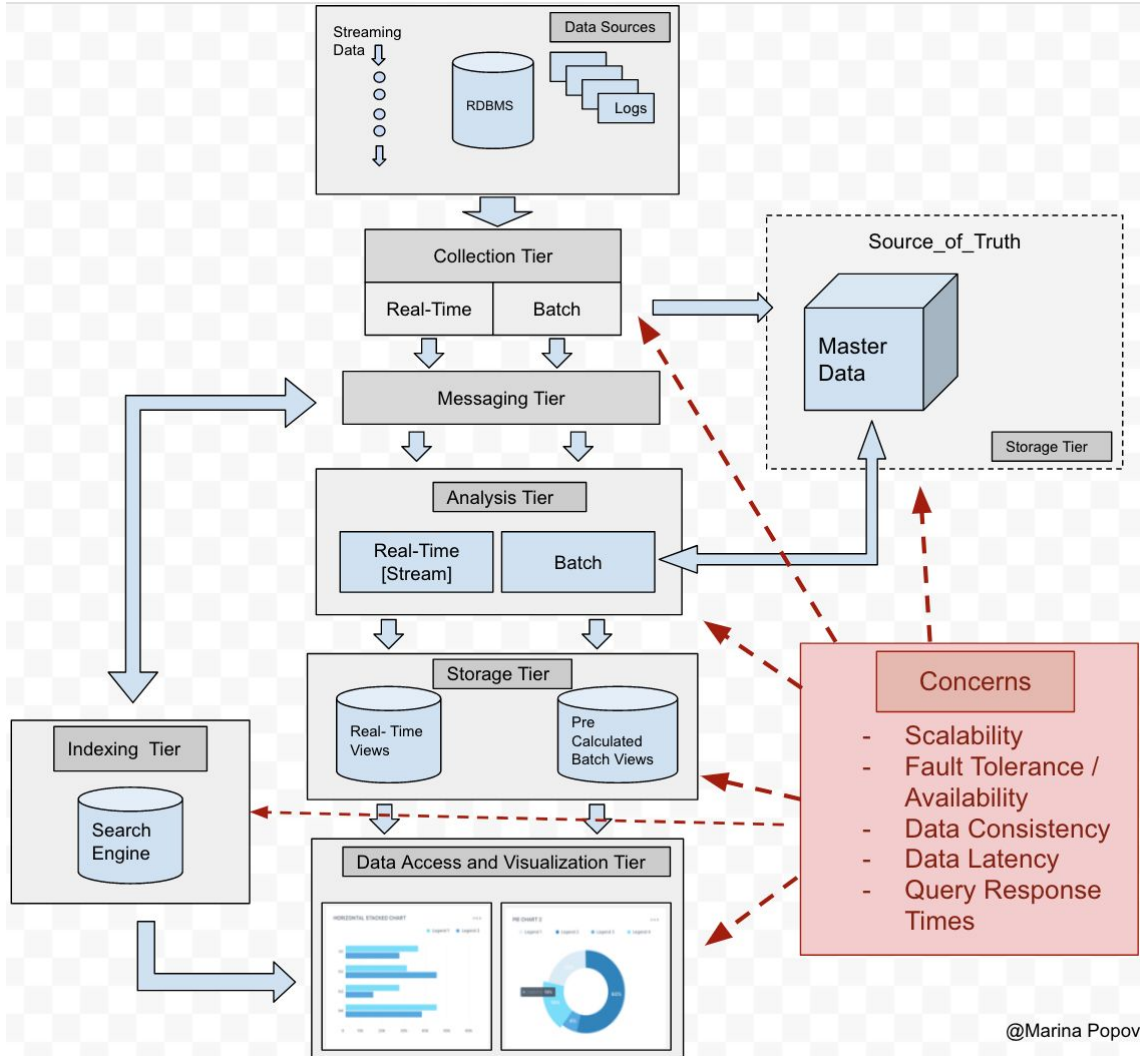
A LOT !!!

core principles of big data

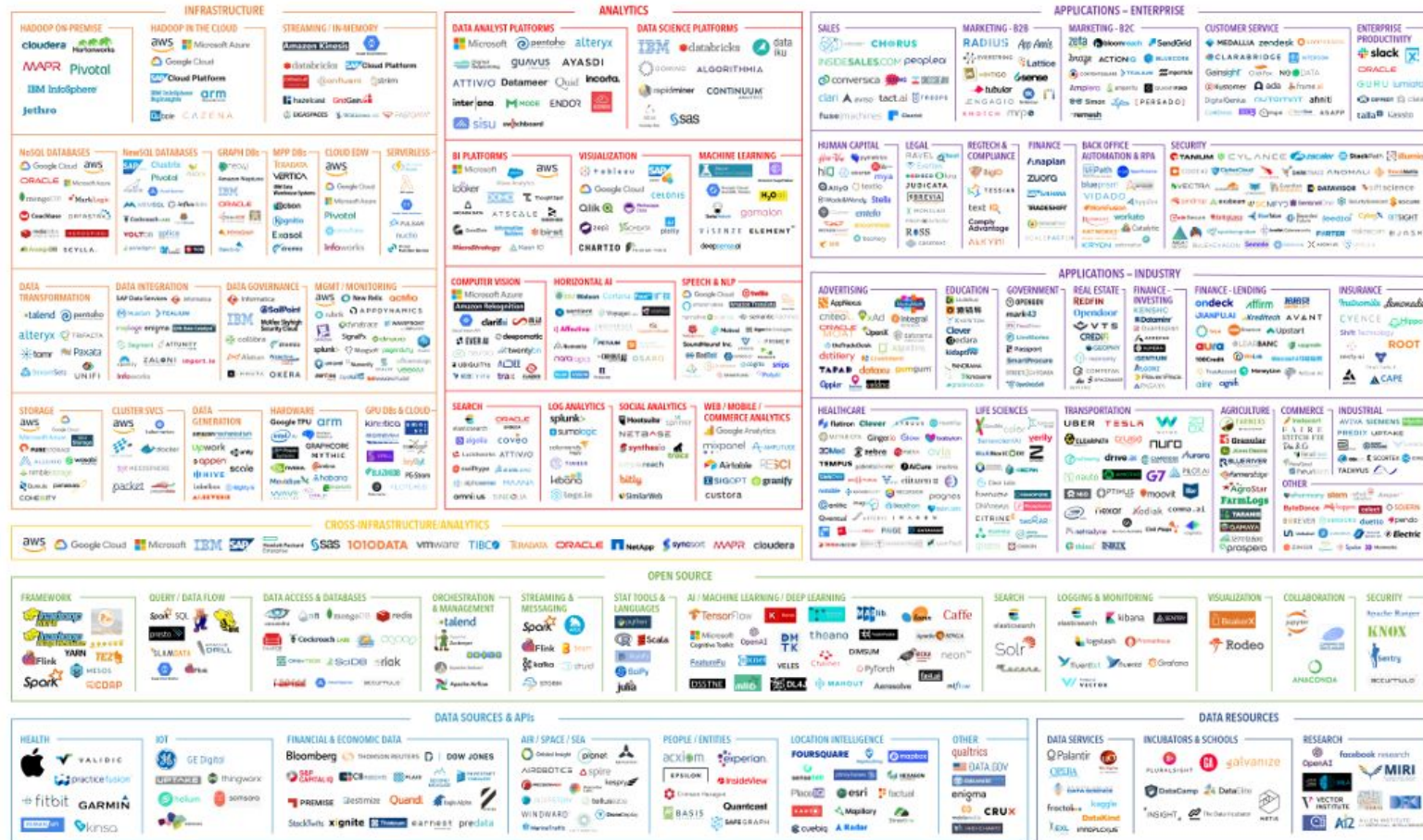
- distributed processing
- ingestion
- storage
- indexing
- search

.... and where to go from here ...

@Marina Popova



@Marina Popova



That's All Folks!

