# Building a Distributed Column Store for Production Observability

# Building a Distributed Column Store for Production Observability

# Please meet Retriever

# Please meet Retriever

Distributed column store

Analytic query engine

Schemaless data model

# Please meet Retriever

Distributed column store

Analytic query engine

Schemaless data model

...

Let's back up a second...

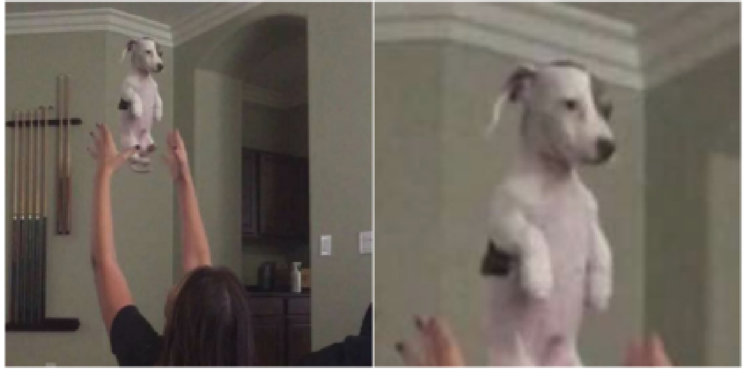
# wat

**mat**
@breadgirl

*record scratch*

*freeze frame*

yup thats me. youre probably wondering how i ended up in this situation



12:15 AM - 2 Sep 2016

# Retriever is a domain specific data store

# What is Honeycomb?

Debugger for production

Help engineers understand and troubleshoot distributed systems

In between metrics and log aggregation

...

# How Honeycomb works

Your systems send us events

- *aka structured logs*
- *aka JSON blobs*


```
{
  "endpoint": "/dashboard",
  "hostname": "app32",
  "response_time_ms": 435,
  "mysql_latency_ms": 102,
  "status": 200,
  "user_id": 42
}
```

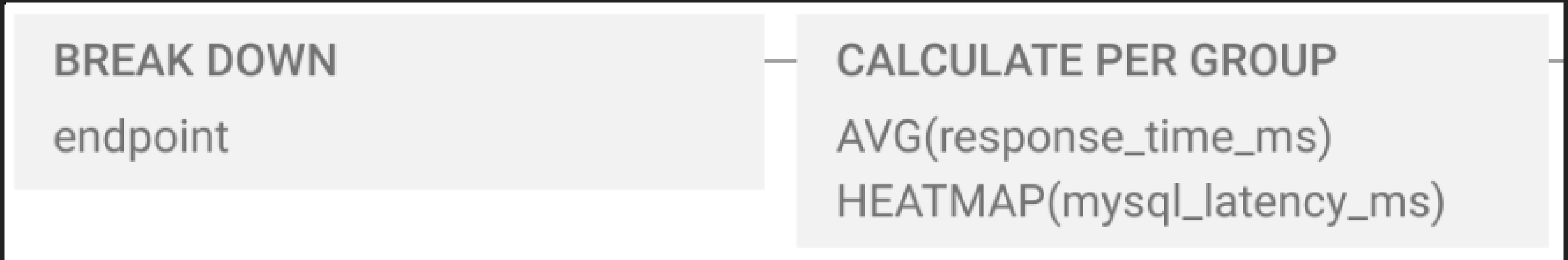
# How Honeycomb works

We store them all

| Timestamp → UTC | endpoint | hostname | response_time_ms |
|---|---|---|---|
| 2017-09-26 15:10:43.593 | "/" | "app6" | 15 |
| 2017-09-26 15:10:45.456 | "/account/update" | "app12" | 362 |
| 2017-09-26 15:10:45.681 | "/dashboard" | "app32" | 435 |
| 2017-09-26 15:10:46.974 | "/" | "app16" | 62 |
| 2017-09-26 15:10:48.668 | "/" | "app0" | 189 |

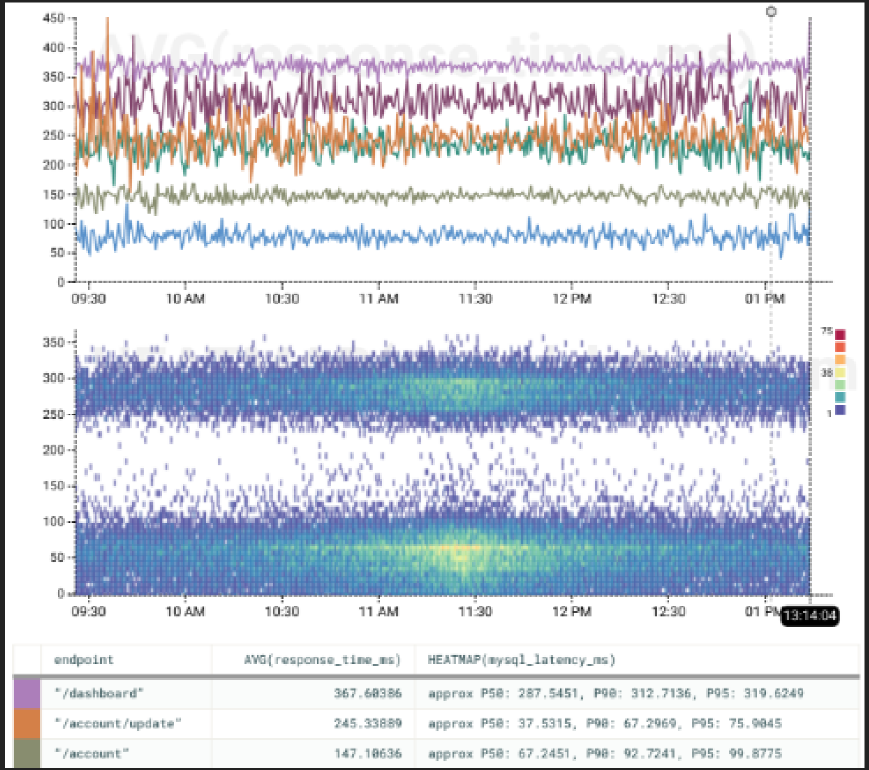
# How Honeycomb works

You query the events



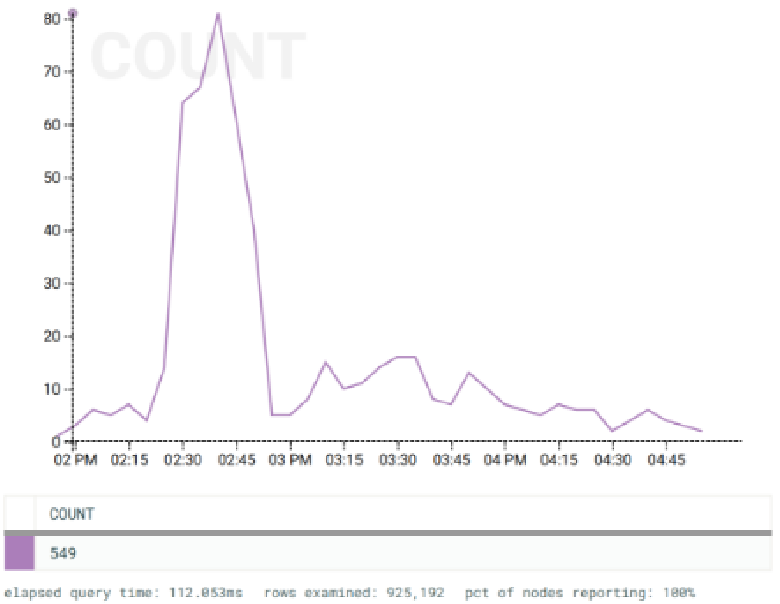
# How Honeycomb works

We turn your queries into pretty graphs



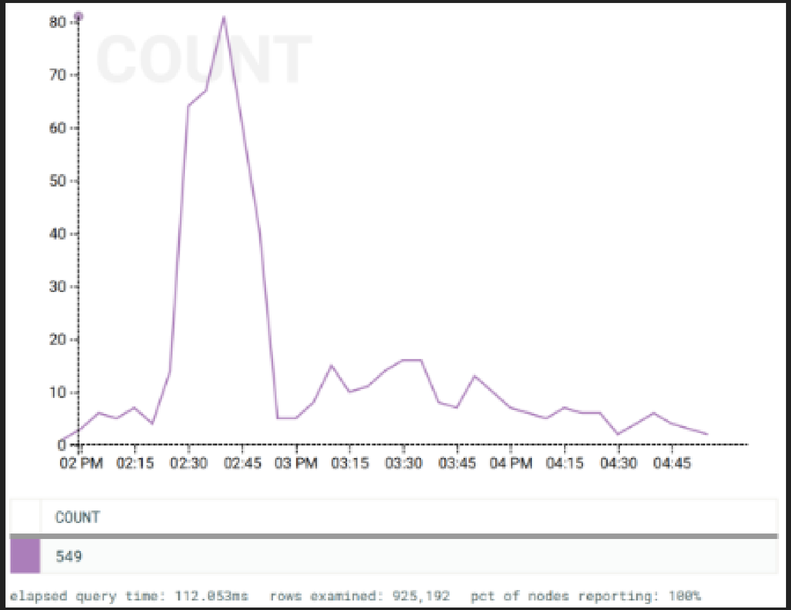
# Honeycomb - example

```
COUNT(*) WHERE status_code >= 500
```



# Honeycomb - example

```
COUNT(*) WHERE status_code >= 500
```

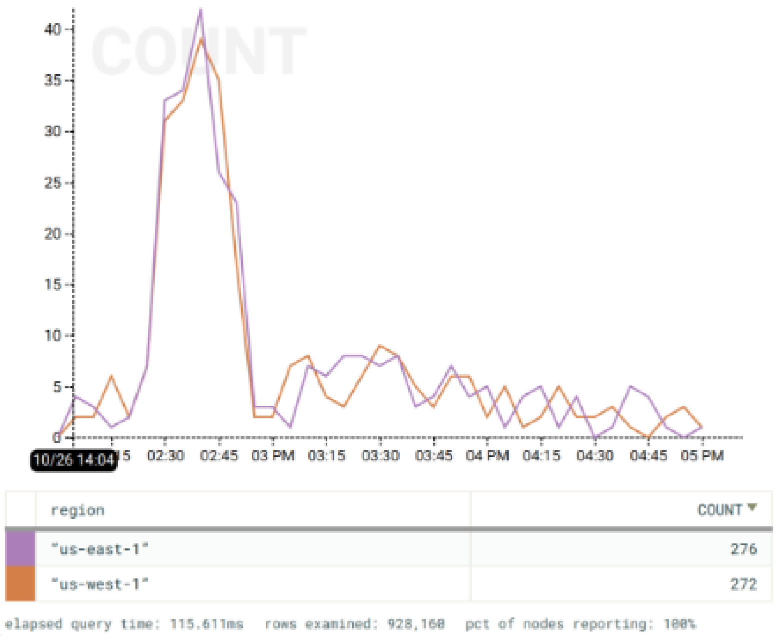



Hey, what's that error spike?

Maybe it's just one availability zone?

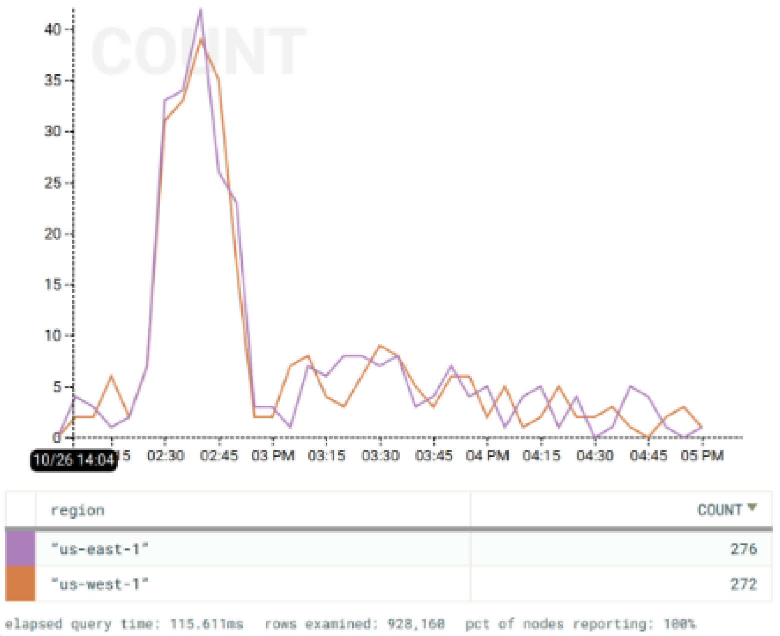
# Honeycomb - example

```
COUNT(*) WHERE status_code >= 500 GROUP BY region
```



# Honeycomb - example

```
COUNT(*) WHERE status_code >= 500 GROUP BY region
```




Across all availability zones...

Let's dig deeper

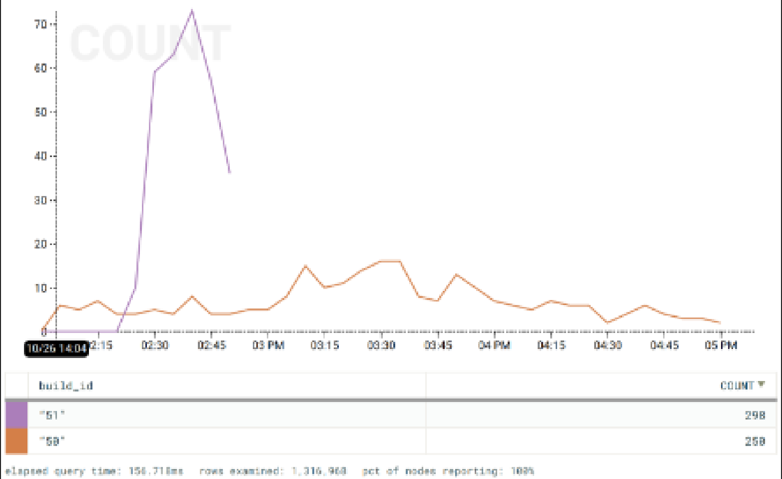
# Honeycomb - example

```
SELECT * WHERE status_code >= 500
```

| status_code | hostname | build_id |
|---|---|---|
| “500” | “app6” | “51” |
| “500” | “app9” | “51” |
| “500” | “app5” | “51” |
| “500” | “app1” | “51” |
| “500” | “app6” | “51” |
| “500” | “app1” | “51” |
| “500” | “app9” | “51” |
| “500” | “app8” | “51” |

# Honeycomb - example

```
COUNT(*) WHERE status_code >= 500 GROUP BY build_id
```

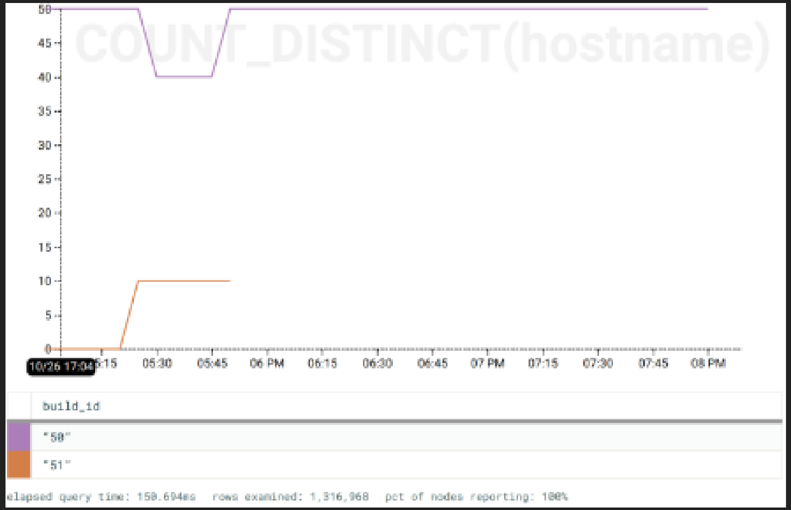



Looks like the spike came from the new build

How widely was the bad build deployed?

# Honeycomb - example

```
COUNT DISTINCT(hostname) GROUP BY build_id
```




Rolled out to 20% of the fleet

Then got rolled back

# Honeycomb - example

Other questions you might ask:

Which customers were affected by this error?

Which customers see the most errors?

Which microservice was causing the error?

# Our requirements

Store lots of events

Query them fast

# Our requirements

SQL-like queries

BREAK DOWN and FILTER

- *on any property of the data*
- *no fixed schema or predefined indices*

High cardinality

# Our requirements

Queries returning raw event data

... and returning time series

Operationally interesting calculations

- *percentiles, histograms*
- *COUNT_DISTINCT*

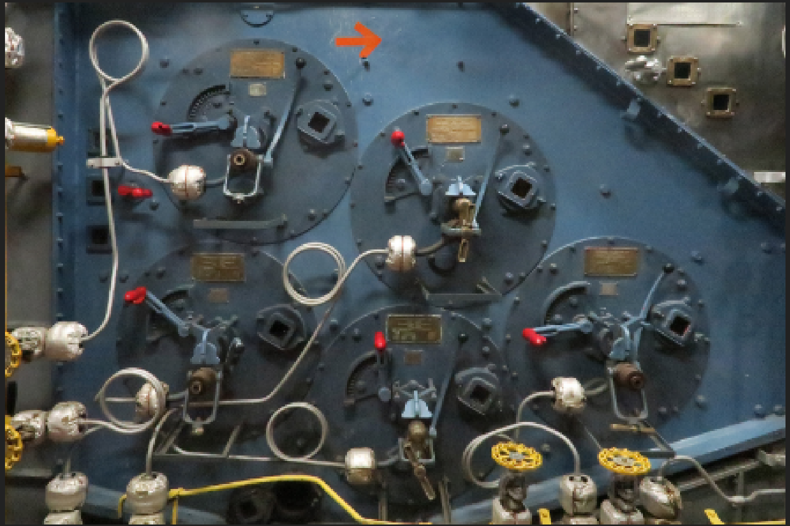
Fast!

# Our requirements

Maintain and operate with a startup budget :)

Simple!

- *Not a general purpose database*
- *Constrained access patterns*
- *No updates*
- *No joins, transactions, ACID*



# Where we're going

- Architecture Overview
- Column-oriented storage
- Distributed queries
- Operations

# Where we're going

**Architecture Overview**

Column-oriented storage

Distributed queries

Operations

# Scuba

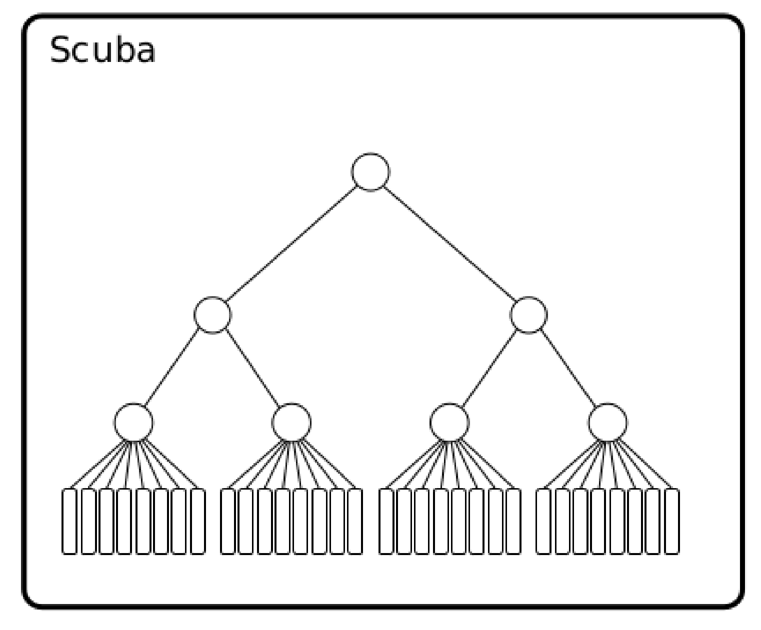
**Scuba: Diving into Data at Facebook**

Lior Abraham*
John Allen
Oleksandr Barykin
Vinayak Borkar
Bhuwan Chopra
Ciprian Gerea
Daniel Merl
Josh Metzler
David Reiss
Subbu Subramanian
Janet L. Wiener
Okay Zed
Facebook, Inc. Menlo Park, CA

Built to solve this problem at Facebook

Distributed event store

# Scuba




Ingest events at scale

Store them all

Distribute events across many nodes

Fast queries by fanning out to multiple nodes

Store everything in RAM for even faster queries

# Retriever at a glance

Distributed event store

Inspired by Facebook's Scuba

# Retriever at a glance

Storage on disk

- *Scuba uses RAM - $$$*
- *SSDs are fast*

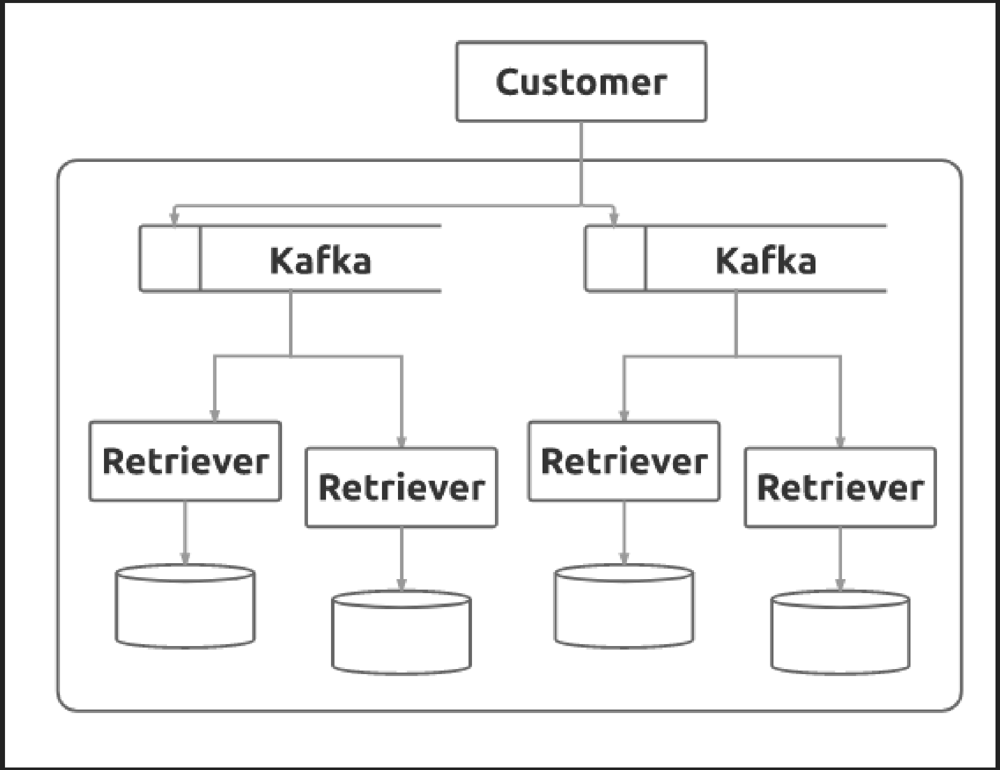
Column-oriented storage

Leverage filesystem features

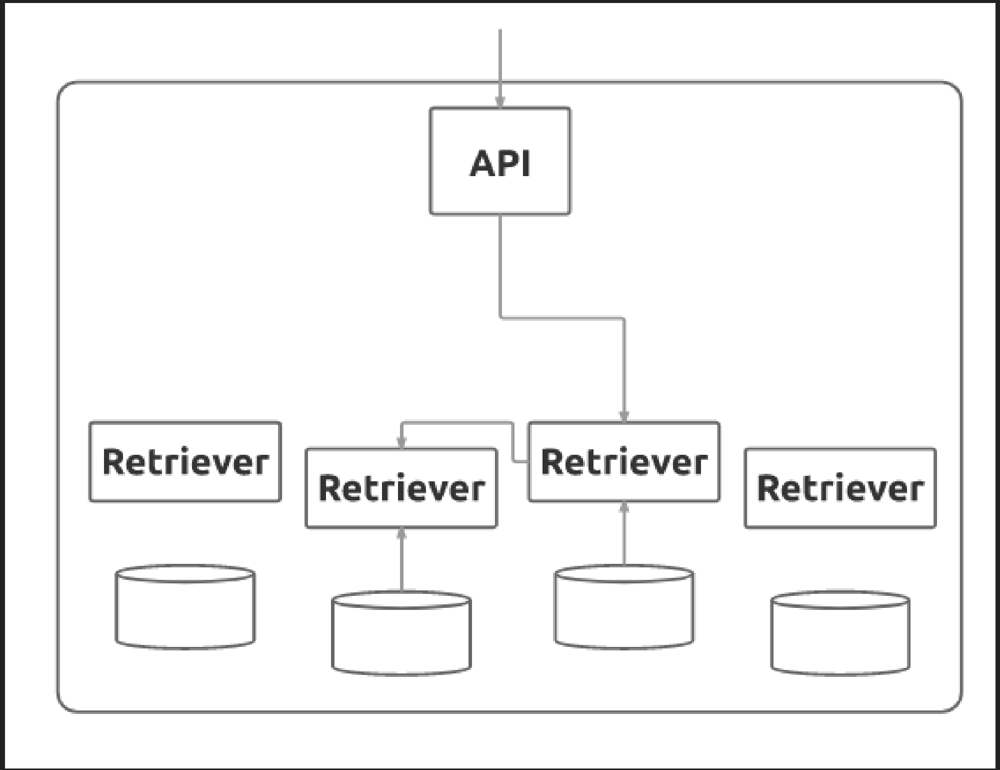
Uses Kafka for ingest

- *And for nice operational properties*

# Architecture - write path



# Architecture - read path



# Where we're going

Architecture Overview

**Column-oriented storage**

Distributed queries

Operations

# Data model - datasets

Customers have one or more datasets

- *analogous to tables*

Datasets are partitioned

- *each dataset is assigned to a number of partitions*
- *typically 3, up to 39*

Dataset partitions contain events

# Data model - events


```
{
  "path": "/foo",
  "response_time": 142.2,
  "status": 200,
}
```

```
{
  "path": "/foo",
  "response_time": 23,
  "status": 400,
  "error": "Bad request"
}
```

# Data model - events

| index | timestamp | path | response_time | status | error | message |
|---|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | | |
| 1 | 45085 | /foo | 23 | 400 | Bad request | |
| 2 | 45087 | /bar | 657 | 200 | | |
| 3 | 45107 | /foo | 105 | 200 | | |
| 4 | 45302 | | | | | Ground control to Major Tom |

No (fixed) schema

- *Arbitrary number of fields - e.g. hundreds*
- *All fields are nullable*

# Data model - events

| index | timestamp | path | response_time | status | error | message |
|---|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | | |
| 1 | 45085 | /foo | 23 | 400 | Bad request | |
| 2 | 45087 | /bar | 657 | 200 | | |
| 3 | 45107 | /foo | 105 | 200 | | |
| 4 | 45302 | | | | | Ground control to Major Tom |

Index is unique

- *assigned on ingest*

Timestamped

# Data model - events

| index | timestamp | path | response_time | status | error | message |
|---|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | | |
| 1 | 45085 | /foo | 23 | 400 | Bad request | |
| 2 | 45087 | /bar | 657 | 200 | | |
| 3 | 45107 | /foo | 105 | 200 | | |
| 4 | 45302 | | | | | Ground control to Major Tom |

How to store events?

- *Files on disk are just streams of bytes*
- *Row oriented?*
- *Column oriented?*

# Row oriented storage

| path | response_time | status | error |
| --- | --- | --- | --- |
| /foo | 142.2 | 200 | |
| . | | | |
| . | | | |

store all fields for a given record together

**record 0**

| | | | |
| --- | --- | --- | --- |
| /foo | 142.2 | 200 | |

# Row oriented storage

| path | response_time | status | error |
|---|---|---|---|
| /foo | 142.2 | 200 | |
| /foo | 23 | 400 | Bad request |
| . | | | |

store all fields for a given record together

| record 0 | | | | record 1 | | | |
|---|---|---|---|---|---|---|---|
| /foo | 142.2 | 200 | | /foo | 23 | 400 | Bad request |

# Row oriented storage

| path | response_time | status | error |
|---|---|---|---|
| /foo | 142.2 | 200 | |
| /foo | 23 | 400 | Bad request |
| /bar | 657 | 200 | |

store all fields for a given record together

| record 0 | | | | record 1 | | | | record 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| /foo | 142.2 | 200 | | /foo | 23 | 400 | Bad request | /bar | 657 | 200 | |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| . | | | | | |
| . | | | | | |

path.string

**record 0**

| | |
|---|---|
| 0 | /foo |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| . | | | | | |

path.string

| record 0 | | record 1 | |
|---|---|---|---|
| 0 | /foo | 1 | /foo |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
| --- | --- | --- | --- | --- | --- |
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| 2 | 45087 | /bar | 657 | 200 | |

path.string

| record 0 | | record 1 | | record 2 | |
| --- | --- | --- | --- | --- | --- |
| 0 | /foo | 1 | /foo | 2 | /bar |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| . | | | | | |
| . | | | | | |

response_time.float

**record 0**

| | |
|---|---|
| 0 | 142.2 |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| . | | | | | |

response_time.float

| record 0 | | record 1 | |
|---|---|---|---|
| 0 | 142.2 | 1 | 23 |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| 2 | 45087 | /bar | 657 | 200 | |

response_time.float

| record 0 | | record 1 | | record 2 | |
|---|---|---|---|---|---|
| 0 | 142.2 | 1 | 23 | 2 | 657 |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| . | | | | | |
| . | | | | | |

error.string

Don't write anything until we have a value!

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| . | | | | | |

error.string

| record 1 | |
|---|---|
| 1 | Bad request |

# Column oriented storage

| index | timestamp | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| 2 | 45087 | /bar | 657 | 200 | |

error.string

**record 1**

| | |
|---|---|
| 1 | Bad request |

# Storage format - timestamp column

| index | timestamp | path | response_time | status | error |
| --- | --- | --- | --- | --- | --- |
| 0 | 45080 | /foo | 142.2 | 200 | |
| 1 | 45085 | /foo | 23 | 400 | Bad request |
| 2 | 45087 | /bar | 657 | 200 | |

Special "timestamp" column always present

| record 0 | | record 1 | | record 2 | | record 3... |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 45808 | 1 | 45085 | 2 | 45087 | ... |

Tells us what index values exist

Let us filter by timestamp

# Storage format - reading

How do we read column-oriented data?

# Storage format - reading

Find out what columns exist

# Storage format - reading

Find out what columns exist

Columns are just files in a directory

- *just list the directory contents*

# Storage format - reading

Find out what columns exist

Columns are just files in a directory

- *just list the directory contents*

```
$ ls
path.string
response_time.float
status.int
error.string
```

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

open the column files we need

- *index (from timestamp column)*
- *status (for filter)*
- *response_time*

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

open the column files we need

index

```
*
```

status.int

```
*
```

response_time.float

```
*
```

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read an index

index

| | |
|---|---|
| 0 | * |

status.int

| |
|---|
| * |

response_time.float

| |
|---|
| * |

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read from status file until we hit index 0

index

| | |
|---|---|
| 0 | * |

status.int

| | | |
|---|---|---|
| 0 | 200 | * |

response_time.float

| |
|---|
| * |

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

status == 200!

index

| | |
|---|---|
| 0 | * |

status.int

| | | |
|---|---|---|
| 0 | 200 | * |

response_time.float

| |
|---|
| * |

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read from response_time file until we hit index 0

index

| | |
|---|---|
| 0 | * |

status.int

| | | |
|---|---|---|
| 0 | 200 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

collect response_time

index

| | |
|---|---|
| 0 | * |

status.int

| | | |
|---|---|---|
| 0 | 200 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read an index

index

| | | |
|---|---|---|
| 0 | 1 | * |

status.int

| | | |
|---|---|---|
| 0 | 200 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read from status file until we hit index 1

index

| | | |
|---|---|---|
| 0 | 1 | * |

status.int

| | | | | |
|---|---|---|---|---|
| 0 | 200 | 1 | 400 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

status ≠ 200, skip this event!

index

| | | |
|---|---|---|
| 0 | 1 | * |

status.int

| | | | | |
|---|---|---|---|---|
| 0 | 200 | 1 | 400 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read an index

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | |
|---|---|---|---|---|
| 0 | 200 | 1 | 400 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read from status file until we hit index 2

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 200 | 1 | 400 | 2 | 200 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

status == 200!

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 200 | 1 | 400 | 2 | 200 | * |

response_time.float

| | | |
|---|---|---|
| 0 | 142.2 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read from response_time file until we hit index 2

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 200 | 1 | 400 | 2 | 200 | * |

response_time.float

| | | | | |
|---|---|---|---|---|
| 0 | 142.2 | 1 | 23 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

read from response_time file until we hit index 2

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 200 | 1 | 400 | 2 | 200 | * |

response_time.float

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 142.2 | 1 | 23 | 2 | 657 | * |

response_times: [142.2]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

collect response_time

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 200 | 1 | 400 | 2 | 200 | * |

response_time.float

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 142.2 | 1 | 23 | 2 | 657 | * |

response_times: [142.2, 657]

# Storage format - reading

e.g. "AVG(response_time) WHERE status = 200"

etc

index

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | * |

status.int

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 200 | 1 | 400 | 2 | 200 | * |

response_time.float

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 142.2 | 1 | 23 | 2 | 657 | * |

response_times: [142.2, 657]

# Storage format - reading

ONLY VALUES IN **BOLD** GET READ

| **index** | path | **response_time** | **status** | error |
| --- | --- | --- | --- | --- |
| **0** | /foo | **142.2** | **200** | |
| **1** | /foo | 23 | **400** | Bad request |
| **2** | /bar | **657** | **200** | |

e.g. "AVG(response_time) WHERE status = 200"

Only read what you need!

- *didn't touch other columns*

# Dynamic sampling

| index | path | response_time | status | error |
|---|---|---|---|---|
| 0 | /foo | 142.2 | 200 | |
| 1 | /foo | 23 | 400 | Bad request |
| 2 | /bar | 657 | 200 | |

Not all events are equally interesting

Most fast, successful responses look the same

And they happen a lot more often

... hopefully

# Dynamic sampling

| index | **sample_rate** | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | **100** | /foo | 142.2 | 200 | |
| 1 | **1** | /foo | 23 | 400 | Bad request |
| 2 | **20** | /bar | 657 | 200 | |

Sample the events you send us

But sample *dynamically*

Tell us: "this event represents 100 just like it"

# Dynamic sampling

| index | **sample_rate** | path | response_time | status | error |
|---|---|---|---|---|---|
| 0 | **100** | /foo | 142.2 | 200 | |
| 1 | **1** | /foo | 23 | 400 | Bad request |
| 2 | **20** | /bar | 657 | 200 | |

Knowing sample rate, we can calculate on sampled data

e.g. COUNT per status

200: 100 + 20 = 120

400: 1 = 1

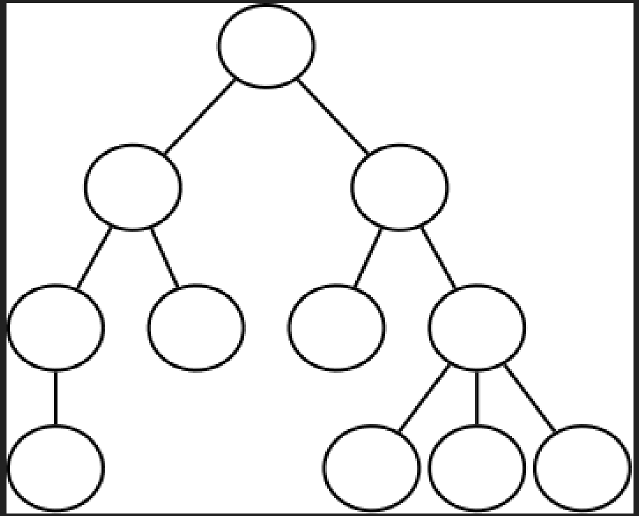
# Where we're going

Architecture Overview

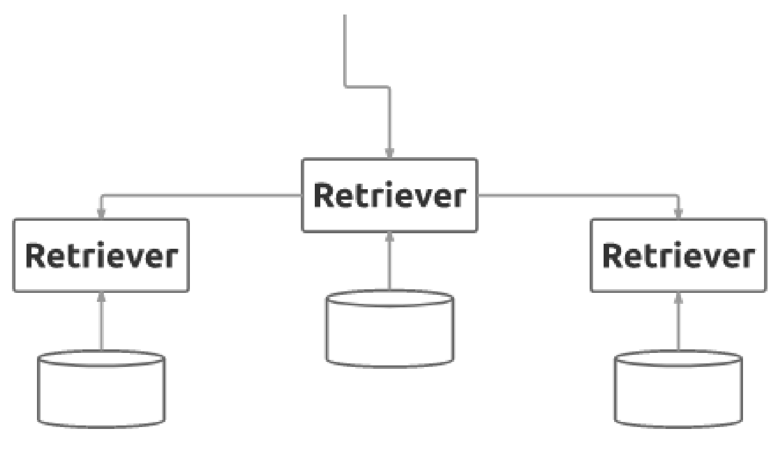
Column-oriented storage

**Distributed queries**

Operations



# Distributed queries




Client issues a query to a retriever root node

Root retriever forwards the query to retrievers on other partitions

- *All scan rows in parallel*
- *All perform local calculations*
- *All return calculations to root node*

Root retriever merges results and returns to client

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Need to be careful about combining results

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Need to be careful about combining results

```
# e.g. averaging two averages gives the wrong answer
AVG( 1, 2, 3, 3 ) # => 2.25
AVG( AVG( 1, 2, 3 ), AVG( 3 ) ) # => 2.5
```

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Need to be careful about combining results

```
# e.g. averaging two averages gives the wrong answer
AVG( 1, 2, 3, 3 ) # => 2.25
AVG( AVG( 1, 2, 3 ), AVG( 3 ) ) # => 2.5
```

Send back partial results that can be combined

```
# e.g. partial counts and sums can be combined correctly
SUM( 1, 2, 3, 3 ) / 4 # => 2.25
(SUM( 1, 2, 3 ) + SUM( 3 )) / (3 + 1) # => 2.25
```

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Other partial results that can be combined:

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Other partial results that can be combined:

Groups

```
{"/dashboard": 235, "/products/iphone": 454}
```

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Other partial results that can be combined:

Groups

```
{"/dashboard": 235, "/products/iphone": 454}
```

COUNT DISTINCT

- *HyperLogLog*

# Distributed reads - calculations

Data is partitioned across nodes

So each node can only do part of the calculation

Other partial results that can be combined:

Groups

```
{"/dashboard": 235, "/products/iphone": 454}
```

COUNT DISTINCT

- *HyperLogLog*

Percentiles

- *T-digest*

# Distributed reads - fanout

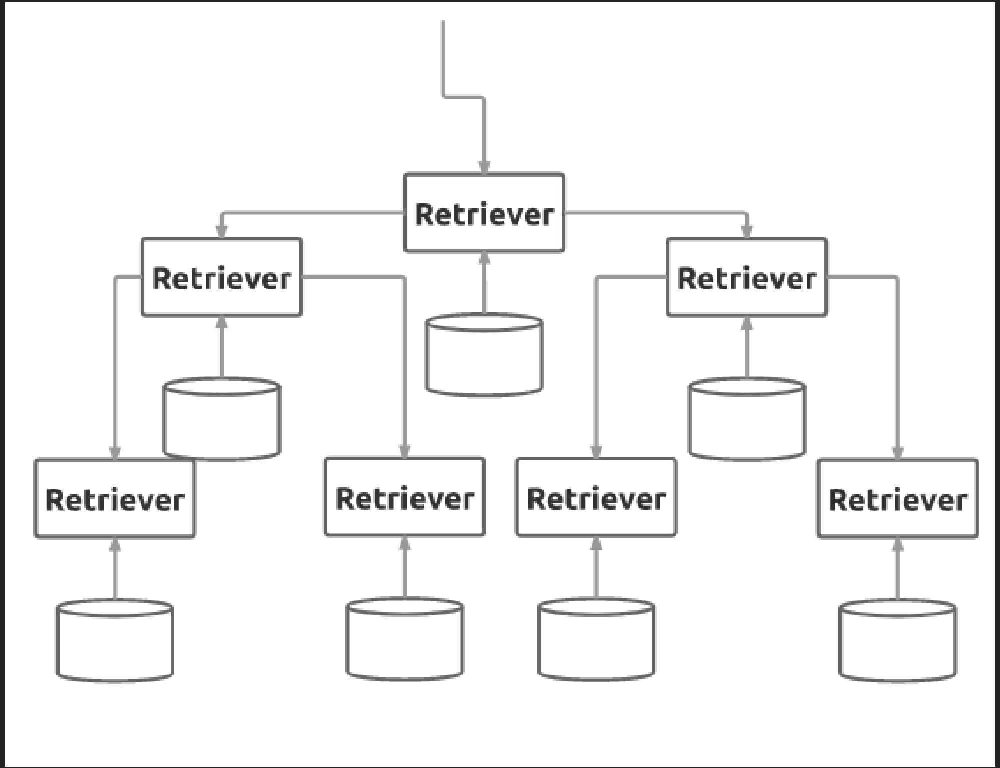
Root node merges the results

May still have to do a lot of work

- *e.g. merging large numbers of groups*

Don't want to overwhelm the root

# Distributed reads - fanout



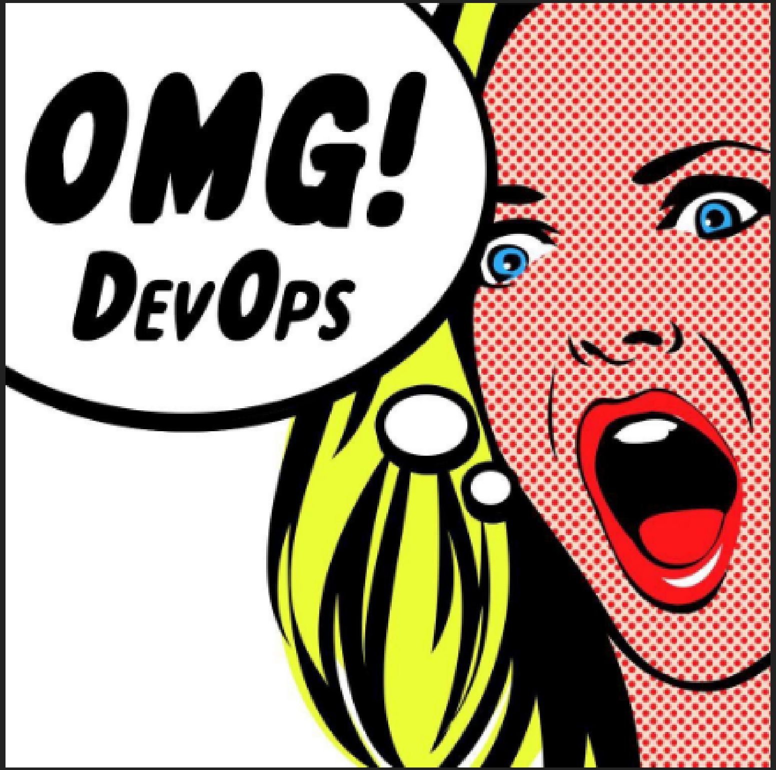
# Where we're going

Architecture Overview

Column-oriented storage

Distributed queries

**Operations**



# Detour - Kafka

Retriever relies on Kafka for ingesting events

Gives us:

- *Write distribution*
- *Replication*
- *Fault tolerance*
- *Disaster recovery*

# Detour - Kafka

Kafka is a distributed log

- *~ message queue*

Publish messages to topics

- *~ tables*

Topics are partitioned

- *horizontal scaling*

Messages *within a partition* are totally ordered

# Detour - Kafka

Kafka actually stores messages on disk

- *whether or not anyone is consuming them*
- *unlike most message queues*

Allows multiple consumers

- *aka pub-sub*

Allows replaying

# Ingestion

Clients publish events to a Kafka topic

- *Kafka topic is partitioned*
- *Datasets are assigned to partitions*

Client chooses which partition to write to

- *Client checks partition assignment for dataset*
- *Picks a partition (at random)*

Retriever on that partition consumes events from Kafka

- *and writes to disk*

All writes replicated to two nodes

- *Each partition of the Kafka topic has two retrievers consuming it*

# Quota management

Each customer gets a storage quota

Want to age out old data past quota

# Quota management

Split events into segments

- *Segments are just directories on disk*
- *Start a new segment when we've written enough events*

Calculate space occupied by each segment

- *Just stat the files!*

Background job periodically deletes oldest data

- *Just delete the directories!*

# Fault tolerance

What if retriever goes down?

- *Crash, network outage...*
- *Deploy / planned maintenance*

We have two replicas...

# Fault tolerance

What if retriever goes down?

- *Crash, network outage...*
- *Deploy / planned maintenance*

We have two replicas...

But we don't want to miss events coming in

# Failure recovery

Each retriever tracks Kafka offset

- *Events are totally ordered in Kafka (per partition)*

On boot, reconsume all events since last offset

# Failure recovery

Periodic checkpoints

- *Store Kafka offset of last-written message*
- *Store *index* of last-written message*

Determines where to reconsume from

# Failure recovery

Periodic checkpoints

- *Store Kafka offset of last-written message*
- *Store *index* of last-written message*

Determines where to reconsume from

Truncate written data to avoid duplicate writes

- *events up to checkpoint index was committed*
- *anything after that is suspect*

# Bootstrapping new nodes

What if a node disappears completely?

Find an existing node on the same partition

Copy over the data

- *just rsync the directory structure!*

... then consume Kafka from last checkpoint

# Operations - summary

Replication

- *via Kafka*

Fault tolerance

- *via Kafka*

Quota management

- *via filesystem*

Bootstrapping new nodes

- *via rsync*
- *and Kafka*

# Retriever

# Summary

Column-oriented storage is a cool trick

- *only read what you need*

# Summary

Column-oriented storage is a cool trick

- *only read what you need*

Kafka solves distributed systems problems for you

- *fault tolerance*
- *replication*

# Summary

Column-oriented storage is a cool trick

- *only read what you need*

Kafka solves distributed systems problems for you

- *fault tolerance*
- *replication*

Filesystems are actually pretty useful

- *read caching*
- *atomic renames*
- *rsync!*

# Summary

Column-oriented storage is a cool trick

- *only read what you need*

Kafka solves distributed systems problems for you

- *fault tolerance*
- *replication*

Filesystems are actually pretty useful

- *read caching*
- *atomic renames*
- *rsync!*

Look for ways to make hard problems easy

# Credits

- *retriever - rkleine (Flickr)*
- *record scratch dog - breadgirl (Twitter)*
- *architecture - barnyz (Flickr)*
- *Scuba paper - Facebook (various authors)*
- *columns - bcymet (Flickr)*
- *reading - triviaqueen (Flickr)*
- *dam - nevilleslens (Flickr)*
- *rube goldberg machine - agrinberg (Flickr)*