The Stream Processor as a Database

The evolution of realtime analytics architecture

Jamie Grier
@jamiegrier
jamie@data-artisans.com

dataArtisans

Who am I?

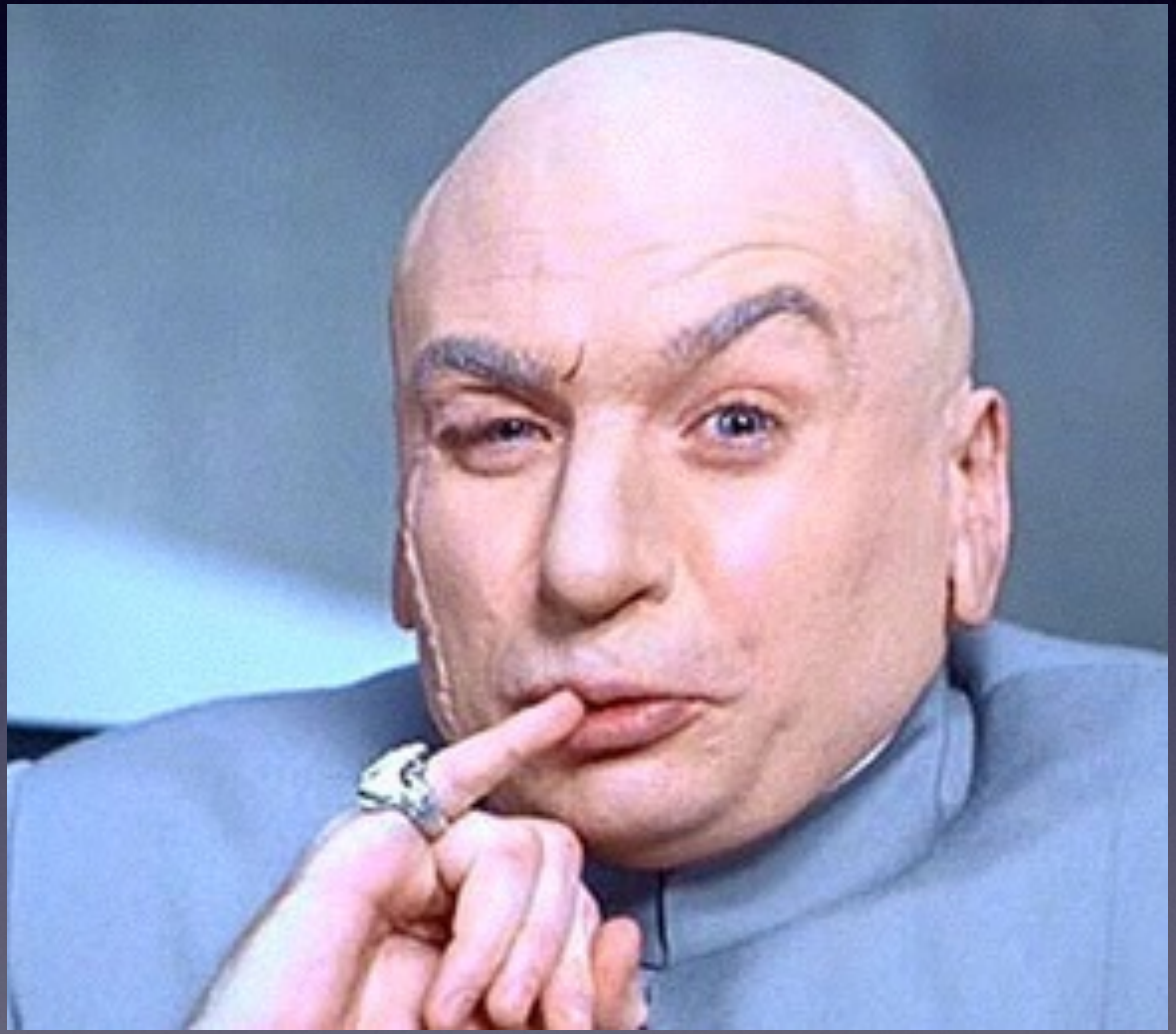
- Director of Applications Engineering at data Artisans
- Previously working on streaming computation at Twitter, Gnip and Boulder Imaging
- Involved in various kinds of stream processing for about a decade
- Now I spend my time helping people to be successful with Apache Flink in production applications

Introduction

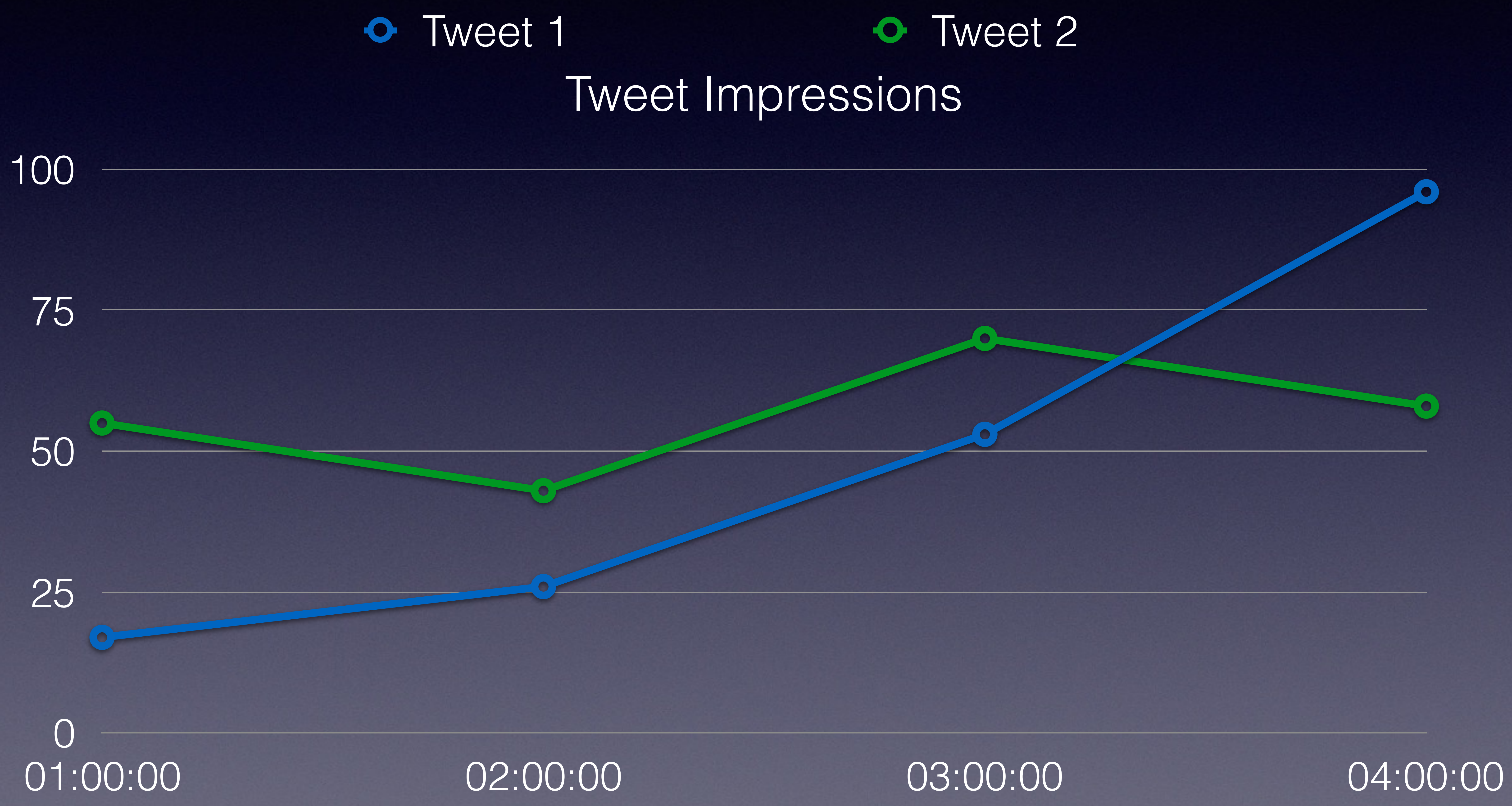
- Evolution of software architecture for real-time analytics at scale
- Pros and cons of each architecture
- New possibilities with robust stateful stream processing and **Queryable State!**
- Introduce the idea of using the stream processor itself as the DB
- **Demo** of Queryable State in Apache Flink 1.2-SNAPSHOT!

Motivating Example

- Tweet Impressions: 1 Million+ Impressions / Second
- 100 Million+ unique tweet impressions per hour
- Computing hourly aggegates for each tweet and storing in key/value store
- Low latency access to the current in-flight aggregates

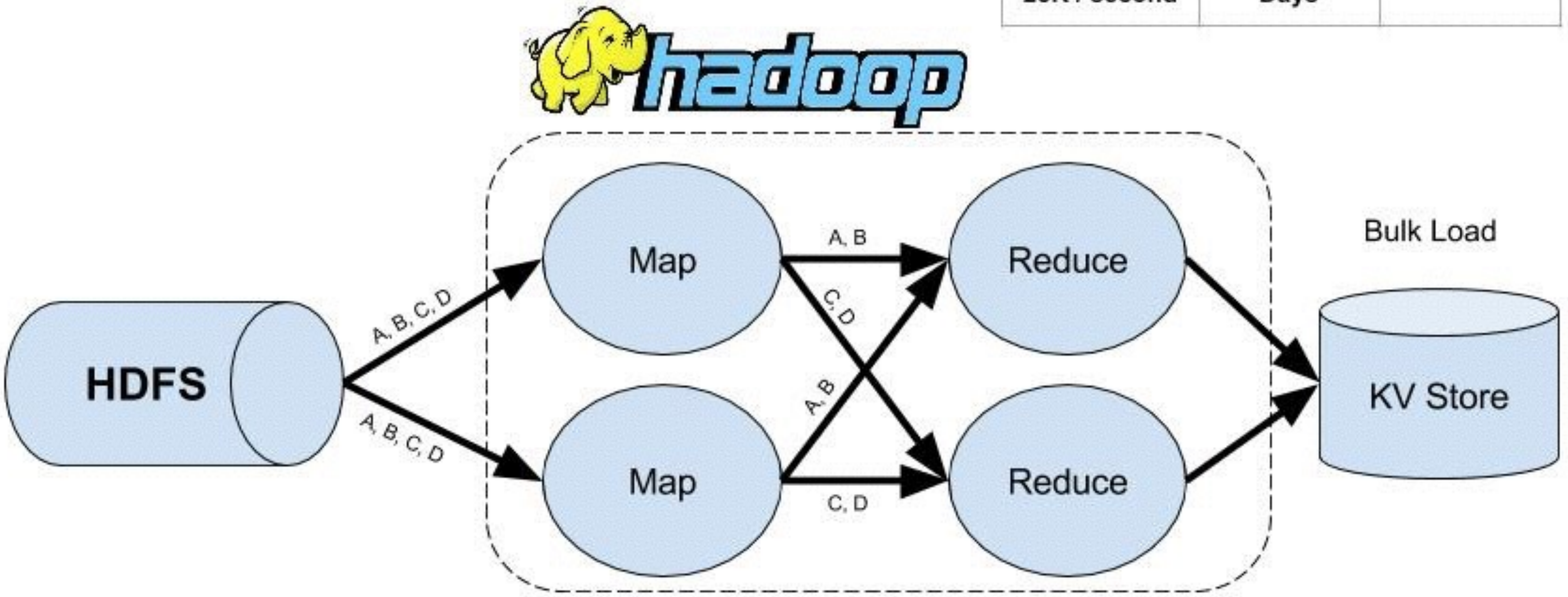


Tweet Impressions



Batch Architecture

KV Store QPS	Data Availability	Robust to Failures
Bulk Load 28K / second	Hours or Days	Yes



Batch Architecture

Pros

Bulk load of data into KV store can be very efficient

Robust against failures — just restart failed partitions

No resource usage between batches

Cons

Data Availability = Batch interval

Could be hours or more than a day

Not robust to out of order data issues

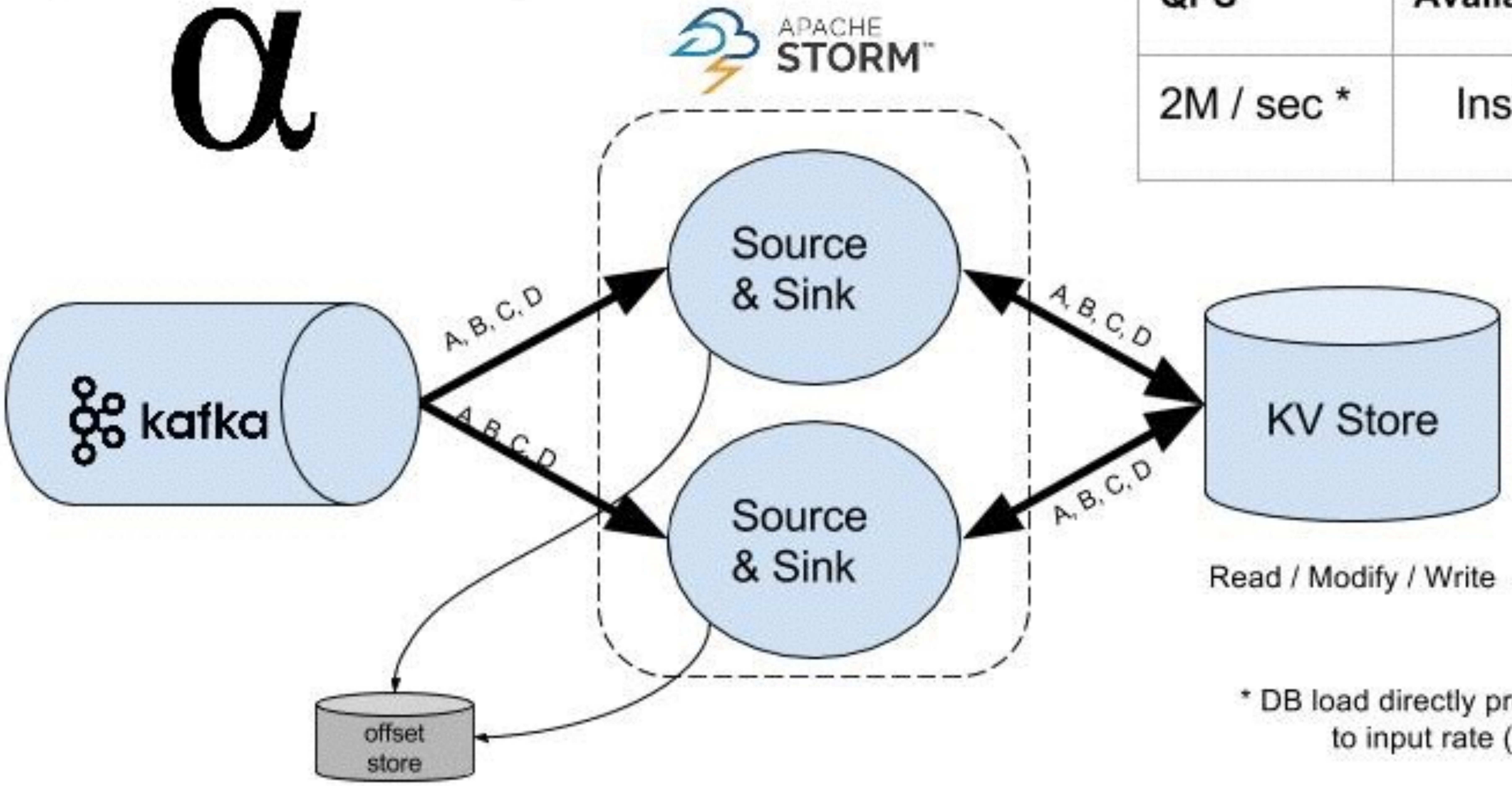
Batch boundary errors introduced

Hard to compute things like session windows

Alpha Architecture :)

α

KV STORE QPS	Data Availability	Robust to Failures
2M / sec *	Instant	No



* DB load directly proportional to input rate (2x)

Alpha Architecture (not really)

Alpha Architecture :)

Pros

Data is available for query instantly

Very simple architecture

Handles out of order data naturally

We can always get the best data so far for any given hour

Cons

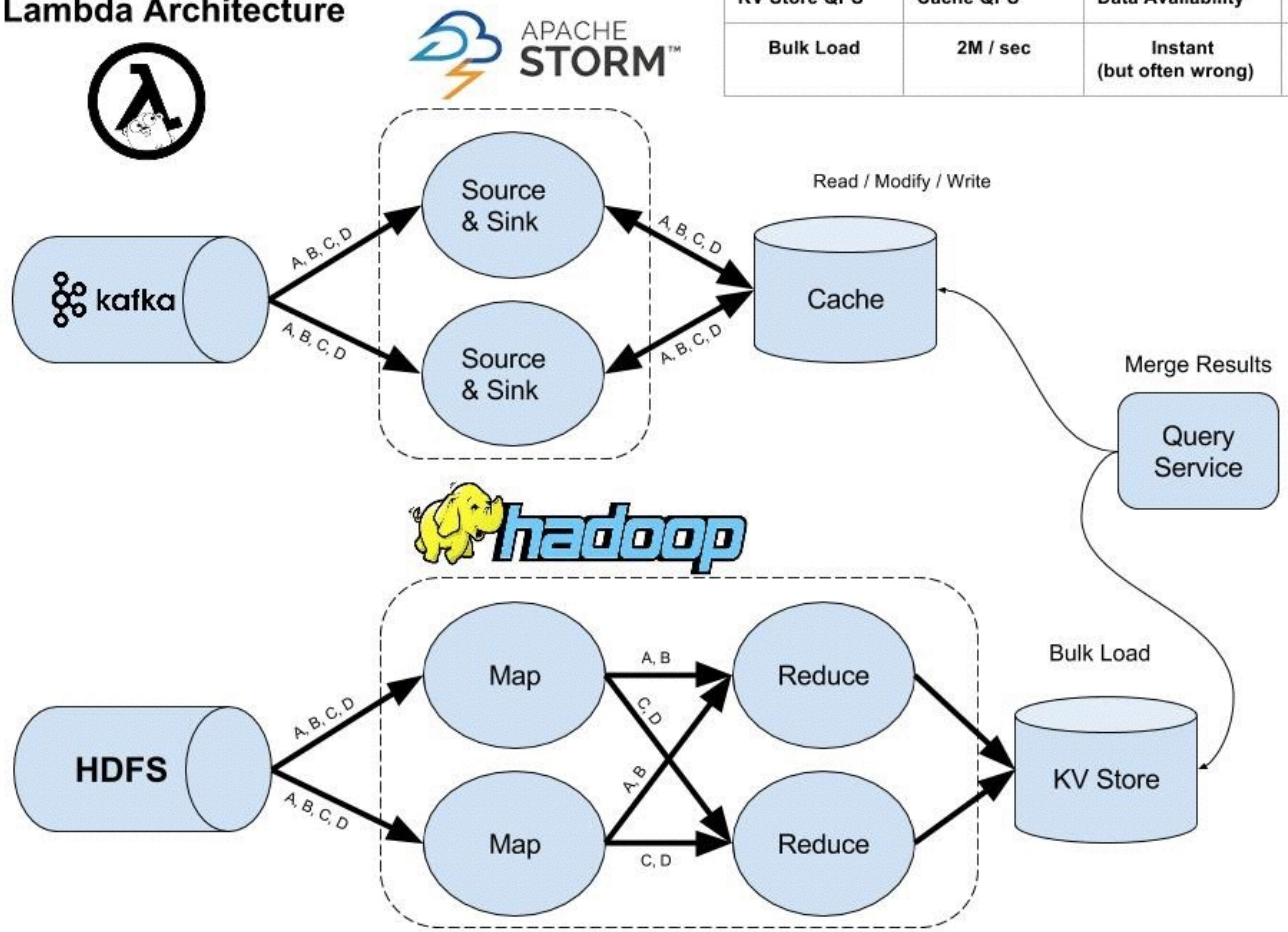
Key value store becomes the bottleneck very quickly

DB Load directly proportional to input rate (2x ?)

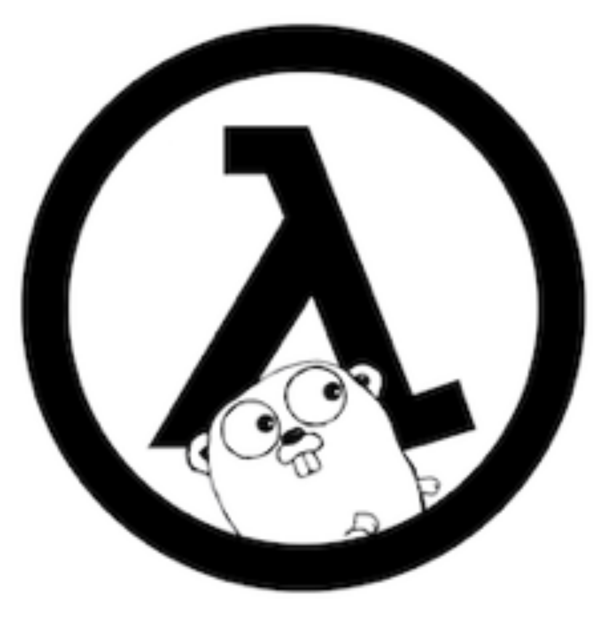
Not robust to failures - failures can lead to multiple counting, etc

Lambda Architecture

KV Store QPS	Cache QPS	Data Availability	Robust to Failures
Bulk Load	2M / sec	Instant (but often wrong)	Yes and No



Lamba Architecture



Lambda Architecture

Pros

Load on KV store is load, effectively bulk load of pre-computed aggregates

Instant access to data

Can be built by cobbling together various existing systems

Some of the best of two worlds

Cons

High load on speed layer cache, new bottleneck

Hard to reason about data correctness in speed layer

Correct data comes very late

Still subjest to batch boundary errors and hard to compute sessions, etc

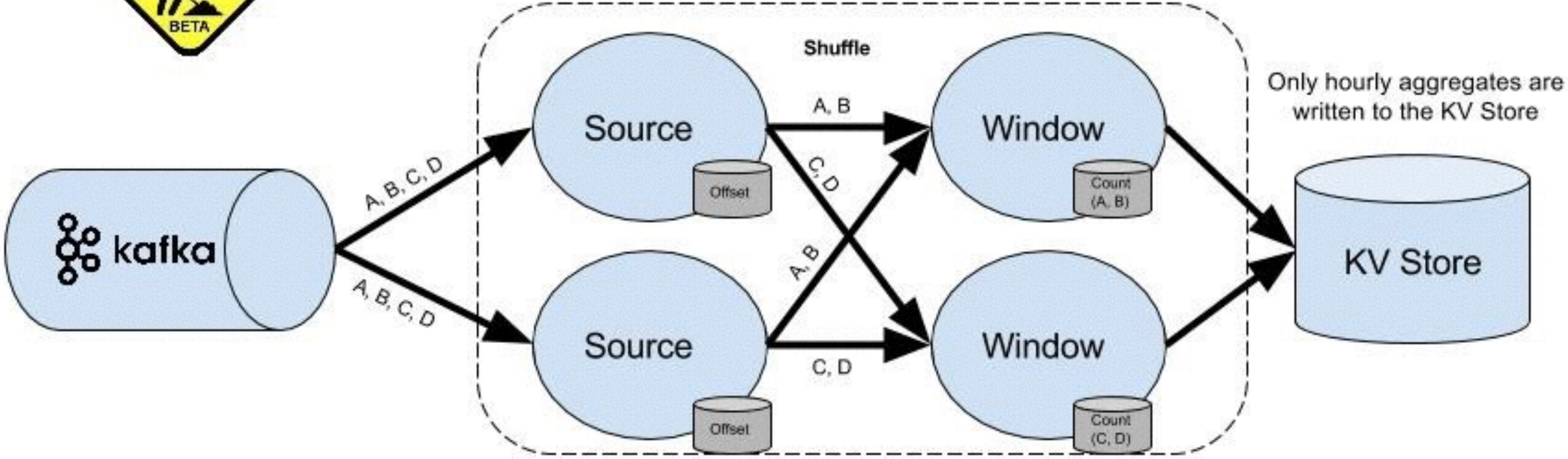
Complex and expensive!

Also the worst of two worlds

Beta Architecture :)

KV STORE QPS	Data Availability	Robust to Failures
100M / 1 hour = **28K /sec** *	**1 hour !**	**Yes**

Flink



Only hourly aggregates are written to the KV Store

= Fault tolerant local state

* KV store load proportional to key cardinality -- not input rate!

Beta Architecture (not really)

Beta Architecture :)

Pros

Dramatically reduced load on the KV Store

DB load is now relative to key cardinality not message input rate

Correct counts even in failure cases

Get’s rid of correctness issues caused by batch boundaries

We can tighten up our aggregate frequency as compared to a batch system

Cons

Time until data available increases - same as window size!

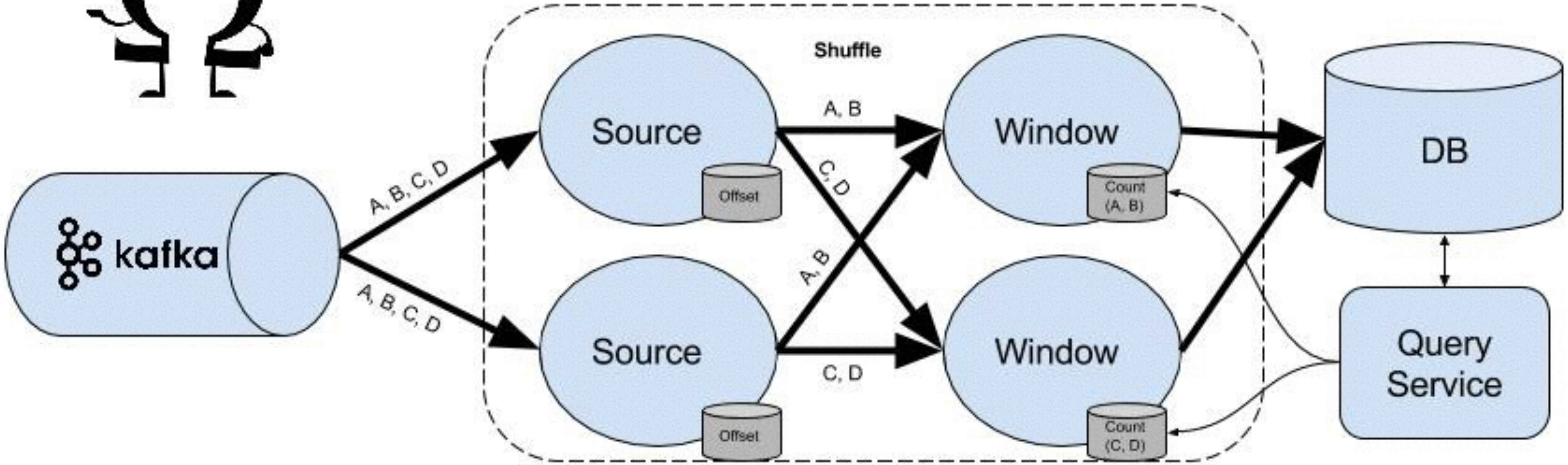
Writes to key value store must be idempotent to achieve robust semantics

The above is not always possible

Omega Architecture :)

Flink

DB QPS	Data Availability	Robust to Failures
100M / 1 hour = **28K /sec ***	**Instant**	**Yes**



= Fault tolerant local state

*** Queryable State!**

Omega Architecture (not really)

Omega Architecture :)

Pros

Dramatically less load on the KV store

KV Store load proportional to unique key rate not input rate

Correct data is available instantly!

Still very simple

Correct in failure cases

Only need current window state in Flink

Cons

Still need a separate KV Store and Query Service to merge results

In current implementation older data can be served in some failure scenarios

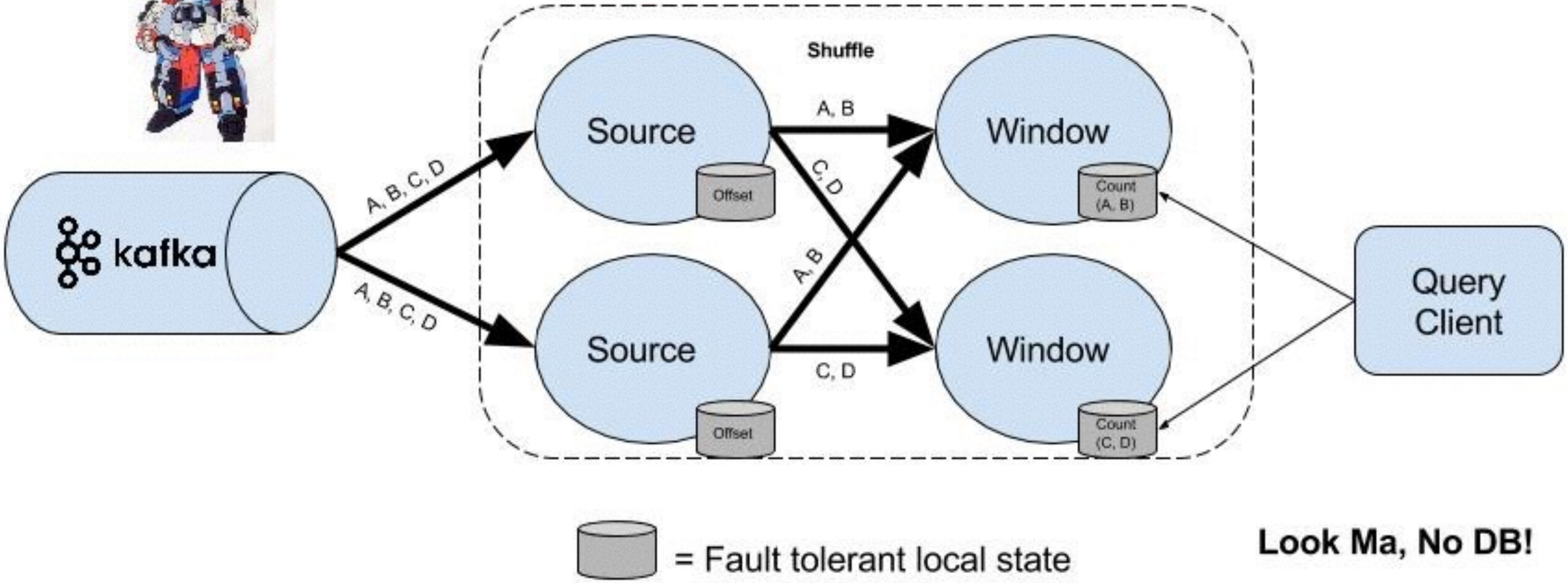
Omega Prime Architecture :)

"Optimus Prime was forced into a combination with his brother. The result of their combination is Omega Prime, a seemingly unique individual who not only combines the best of his component parts abilities but adds a considerable amount of power to the total. His importance in the fight against evil cannot be over-estimated, stretching beyond even his own universe." — Wikipedia

KV Store QPS	Data Availability	Robust to Failures
The Stream Processor is the Database	Instant	Yes

Omega Prime Architecture :)

Flink



Look Ma, No DB!

Omega Prime Architecture

Omega Prime Architecture :)

Pros

No external key value store

Easy to scale because all state is local

Data is available instantly

Simple

Correct in failure cases

Build whatever stateful stream applications you can think of and still have strong correctness guarantees

Cons

Total state size must fit in Flink State. This is a limitation — *for now*.

See *"very large state"* talk by Stephan Ewen

Are people ready to consider using the stream processor state as the only data store?

Demo!

- Flink 1.2-SNAPSHOT (master)
- Created a plug-in for Grafana to query Flink state directly
- Created a simple REST server to serve requests from Grafana
- Queries window state in Flink directly

dataArtisans

We're Hiring!

http://data-artisans.com/careers