Modern Data Pipelines

Ryan Knight
James Ward

@TODO
@_JamesWard

Ryan Knight

Architect at Starbucks

- Distributed Systems guru
- Scala, Akka, Cassandra Expert & Trainer
- Skis with his 5 boys in Park City, UT
- First time to jFokus

James Ward

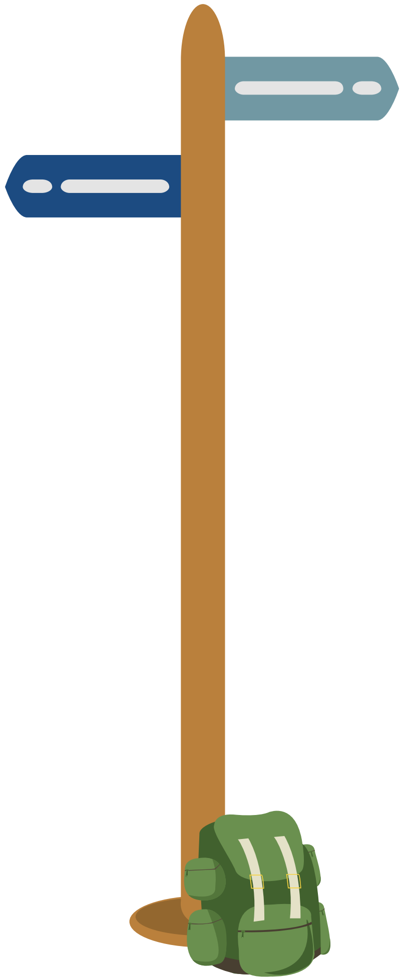
Developer at Salesforce

- Back-end Developer
- Creator of WebJars
- Blog: www.jamesward.com
- Not a JavaScript Fan
- In love with FP

salesforce

Agenda

- Modern Data Pipeline Overview
- Kafka
- Akka Streams
- Play Framework
- Flink
- Cassandra
- Spark Streaming



Code

github.com/jamesward/koober

Modern Data Pipelines

Real-Time, Distributed, Decoupled

Why Streaming Pipelines

Real Time Value

- Allow business to react to data in real-time instead of batch

Real Time Intelligence

- Provide real-time information so that the apps can use the information and adapt their user interactions

Distributed data processing that is both scalable and resilient

Clickstream analysis

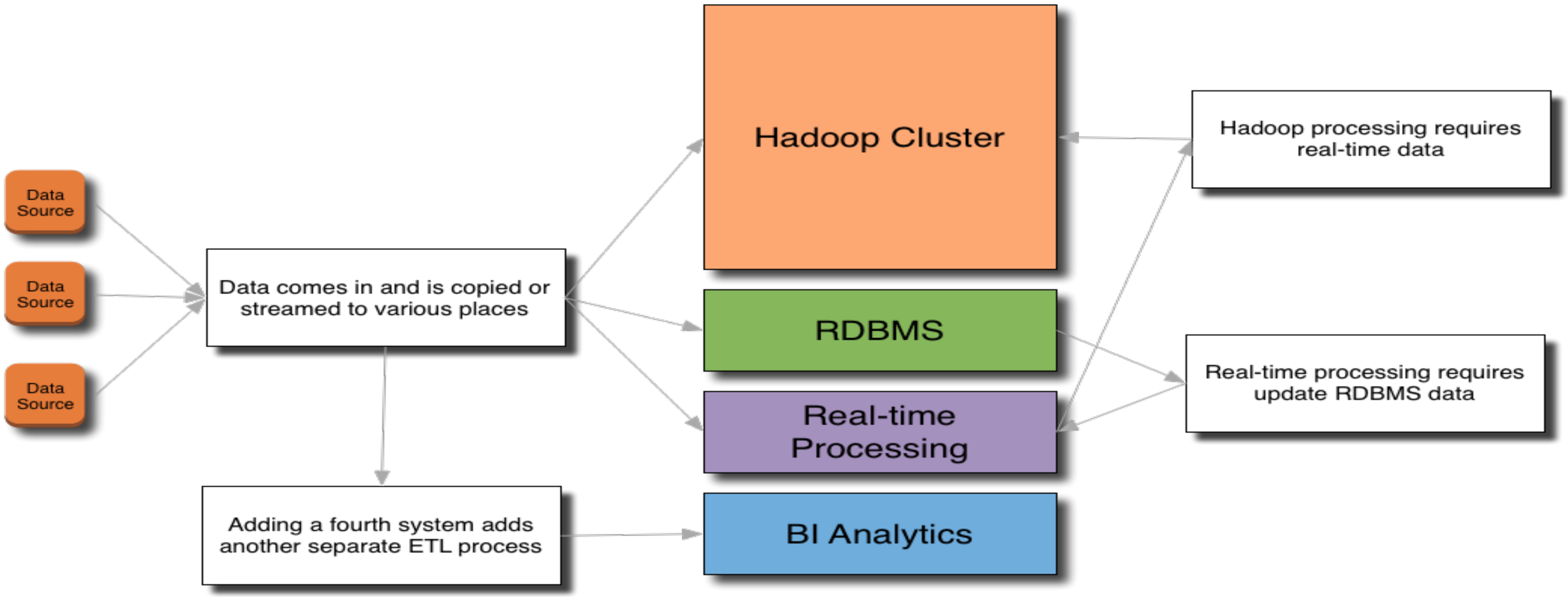
Real-time anomaly detection

Instant (< 10 s) feedback - ex. real time concurrent video viewers / page views

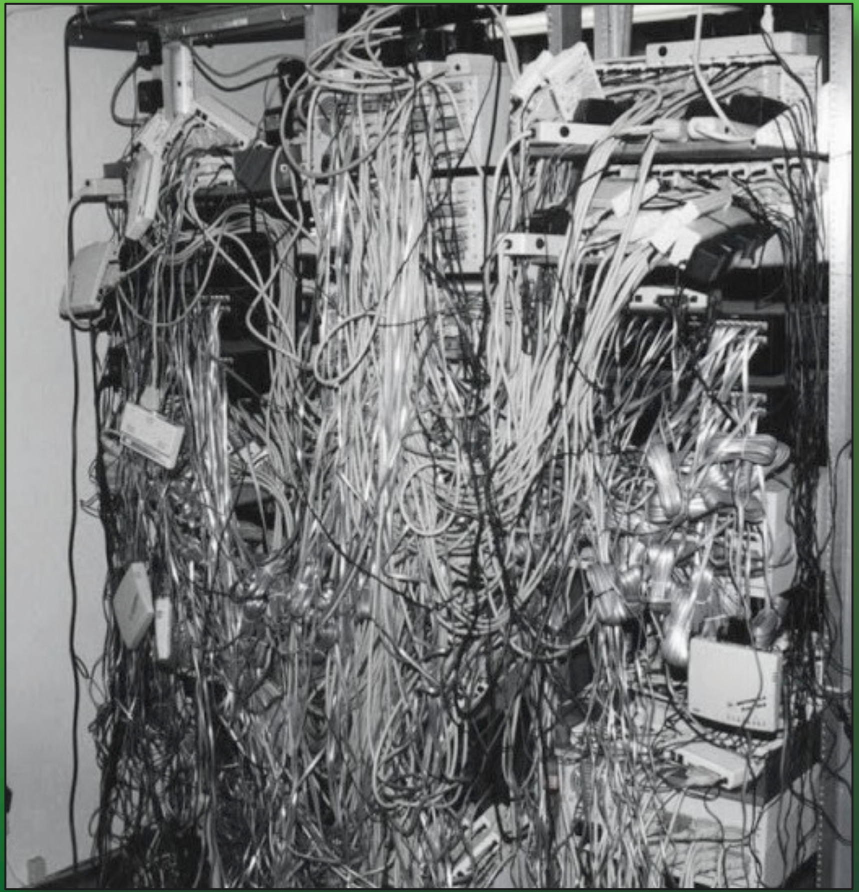
Data Pipeline Requirements

- Ability to process massive amounts of data
- Handle data from a wider variety of sources
- Highly Available
- Resilient - not just fault tolerant
- Distributed for Scale of Data and Transactions
- Elastic
- Uniformity - all-JVM based for easy deployment and management

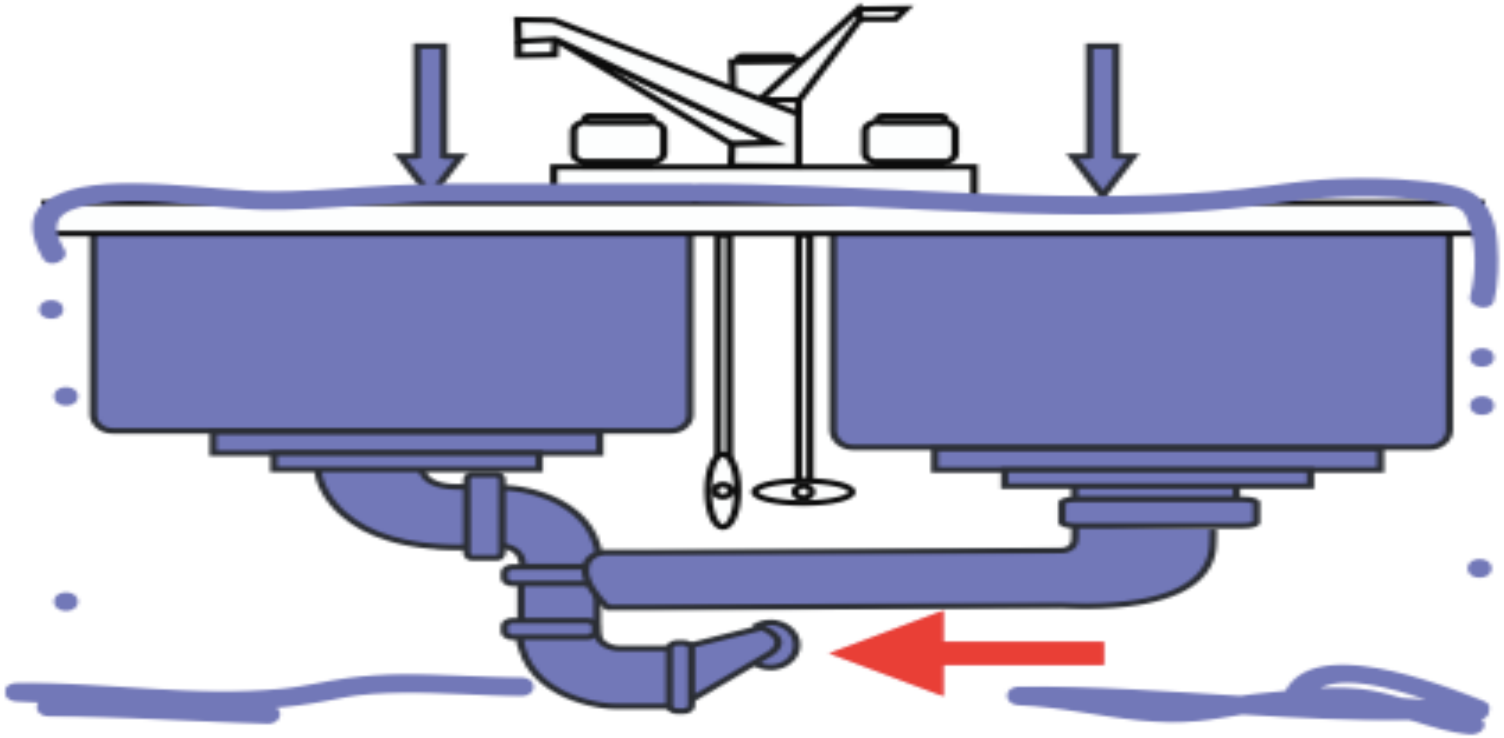
Traditional ETL



Data Integration Today

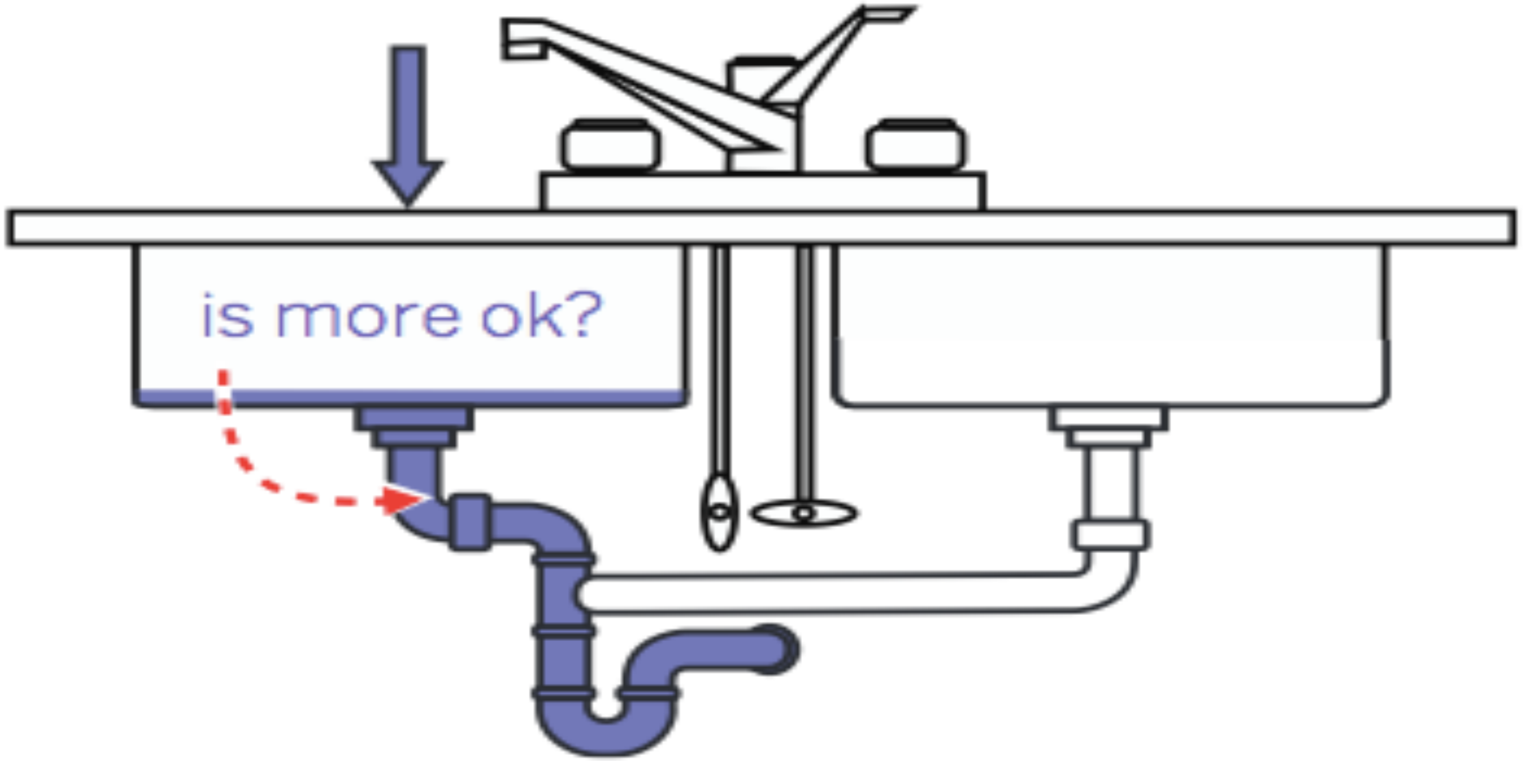


Data Pipelines today



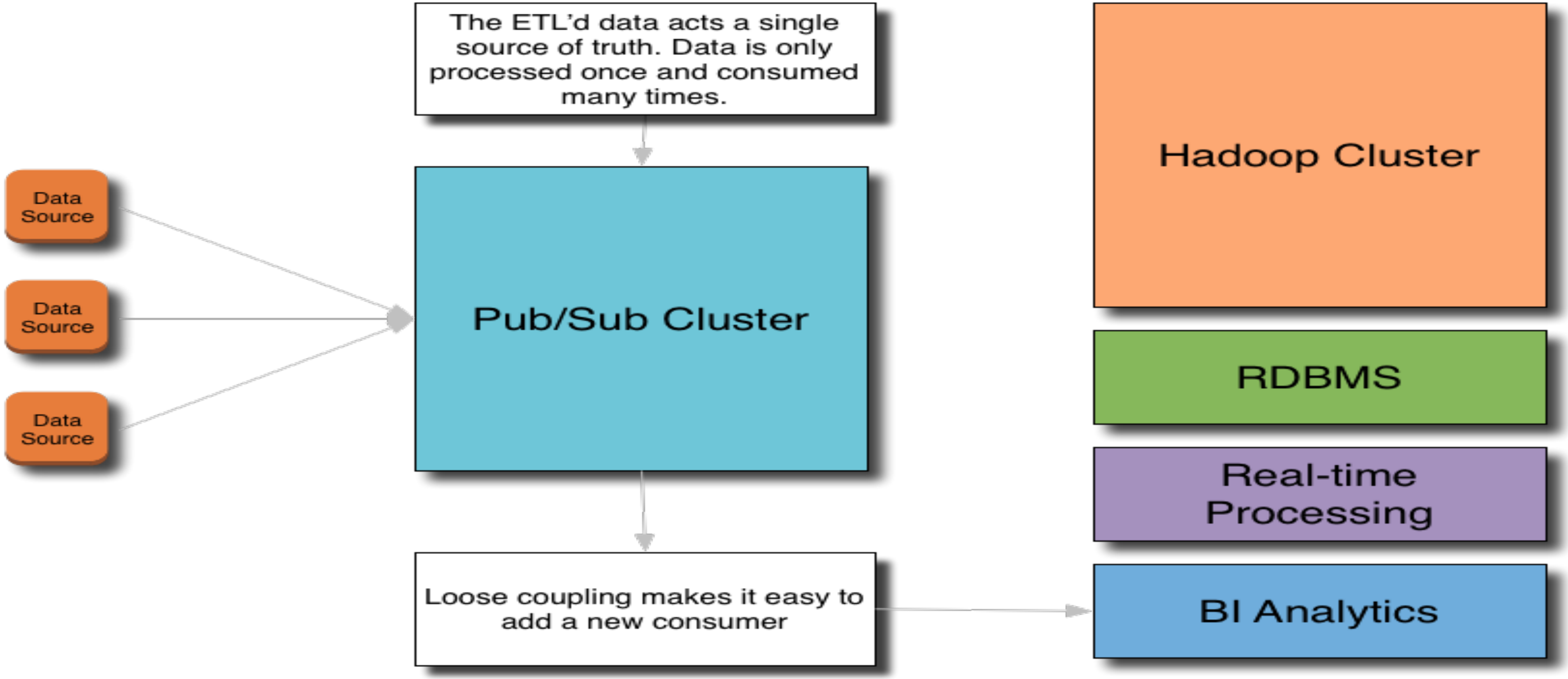
http://ferd.ca/queues-don-t-fix-overload.html

Backpressure

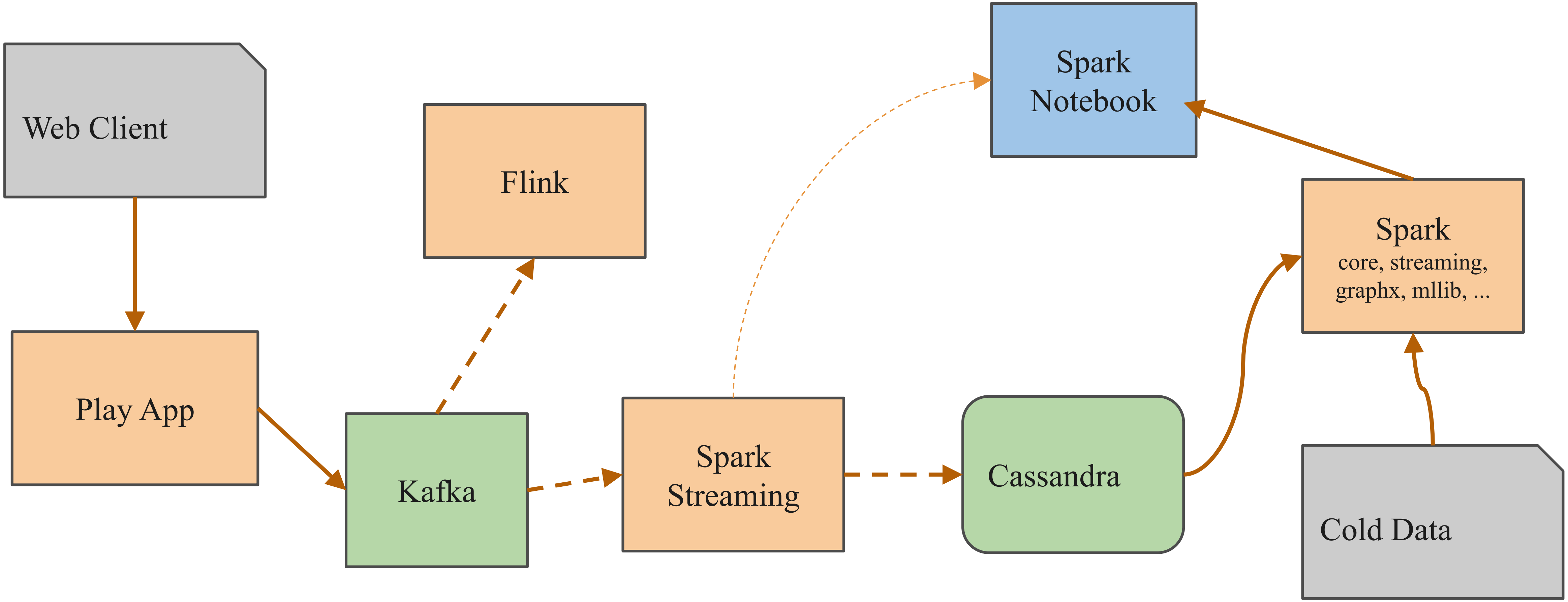


http://ferd.ca/queues-don-t-fix-overload.html

Data Hub / Stream Processing



Pipeline Architecture



Koober

github.com/jamesward/koober

Kafka

Distributed Commit Logs

What is Kafka?

Kafka is a distributed and partitioned commit log
Replacement for traditional message queues and publish subscribe systems
Central Data Backbone or Hub
Designed to scale transparently with replication across the cluster

Core Principles

1. One pipeline to rule them all
2. Stream processing >> messaging
3. Clusters not servers
4. Pull Not Push

Kafka Characteristics

Scalability of a filesystem

- Hundreds of MB/sec/server throughput
- Many TB per server

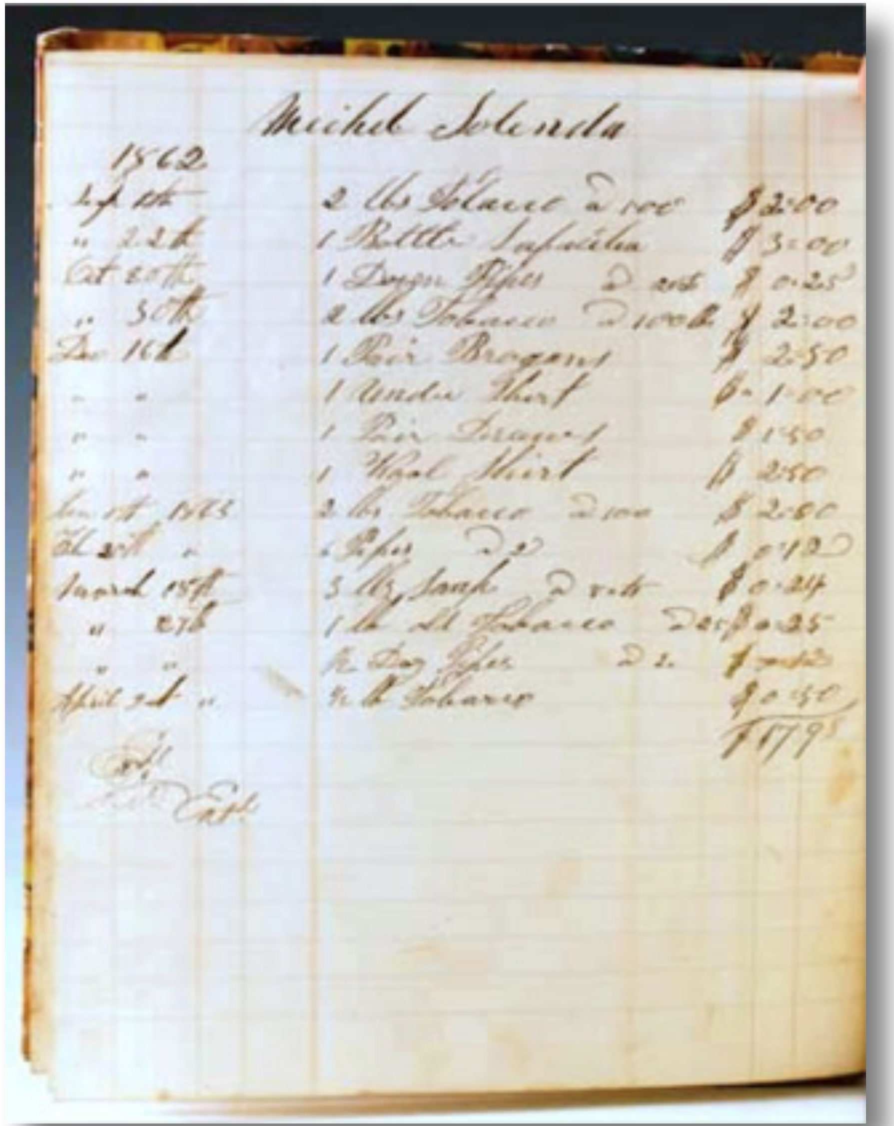
Durable - Guarantees of a database

- Messages strictly ordered
- All data persistent

Distributed by default

- Replication
- Partitioning model

Kafka is about logs



The Event Log

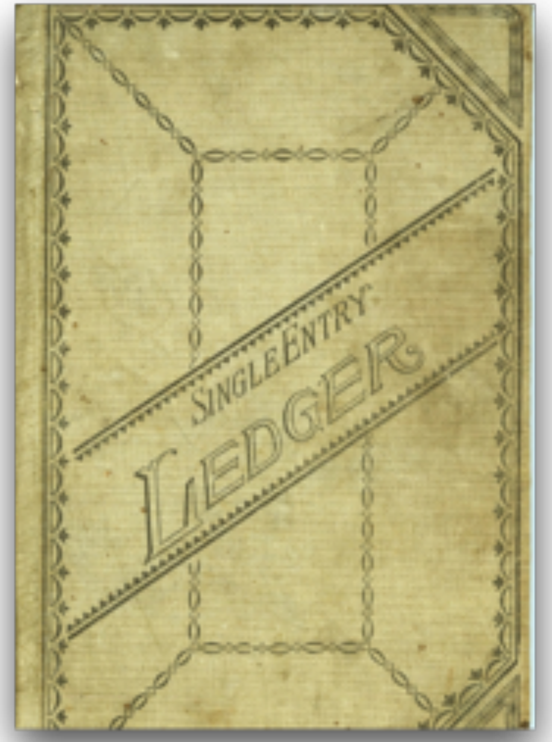
Append-Only Logging

Database of Facts

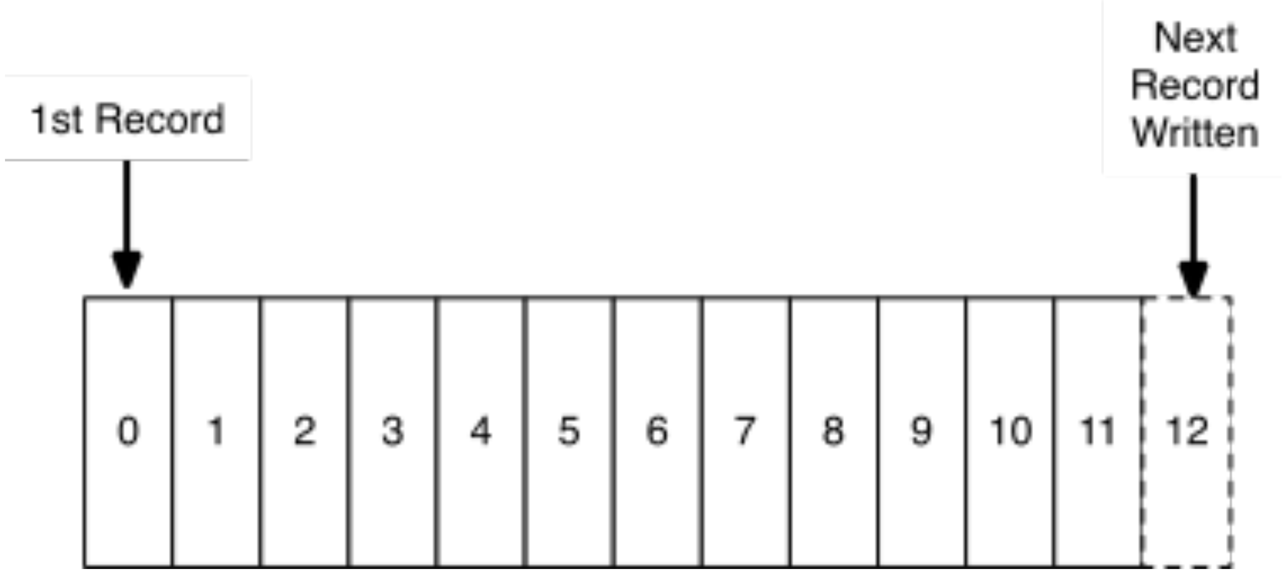
Disks are Cheap

Why Delete Data any more?

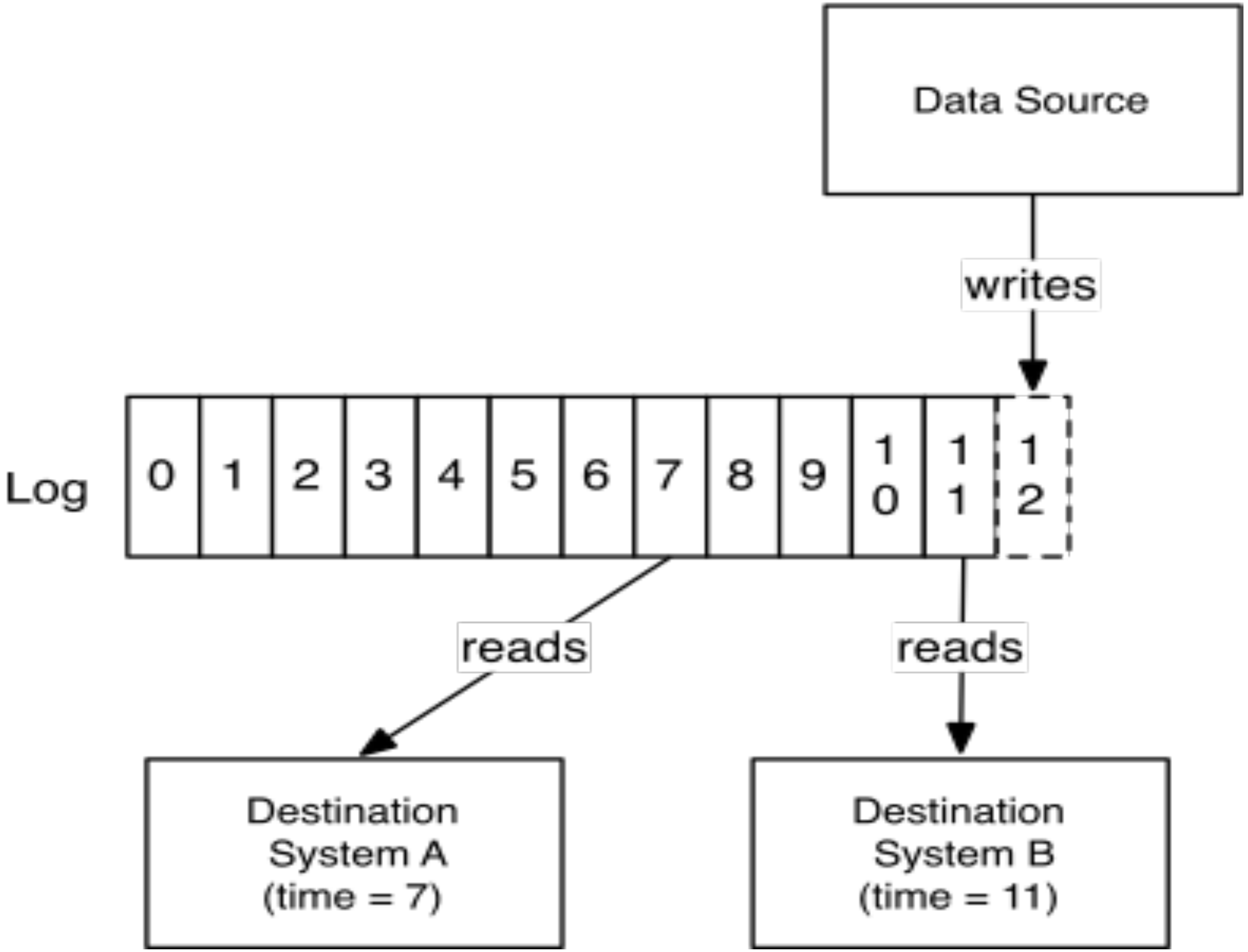
Replay Events



Append Only Logging

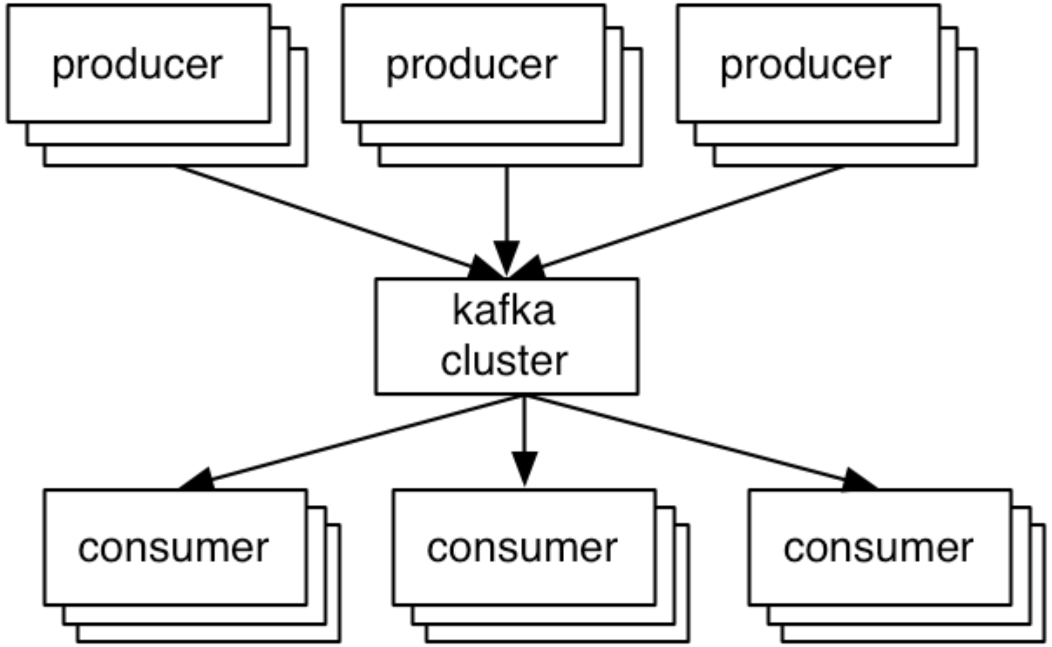


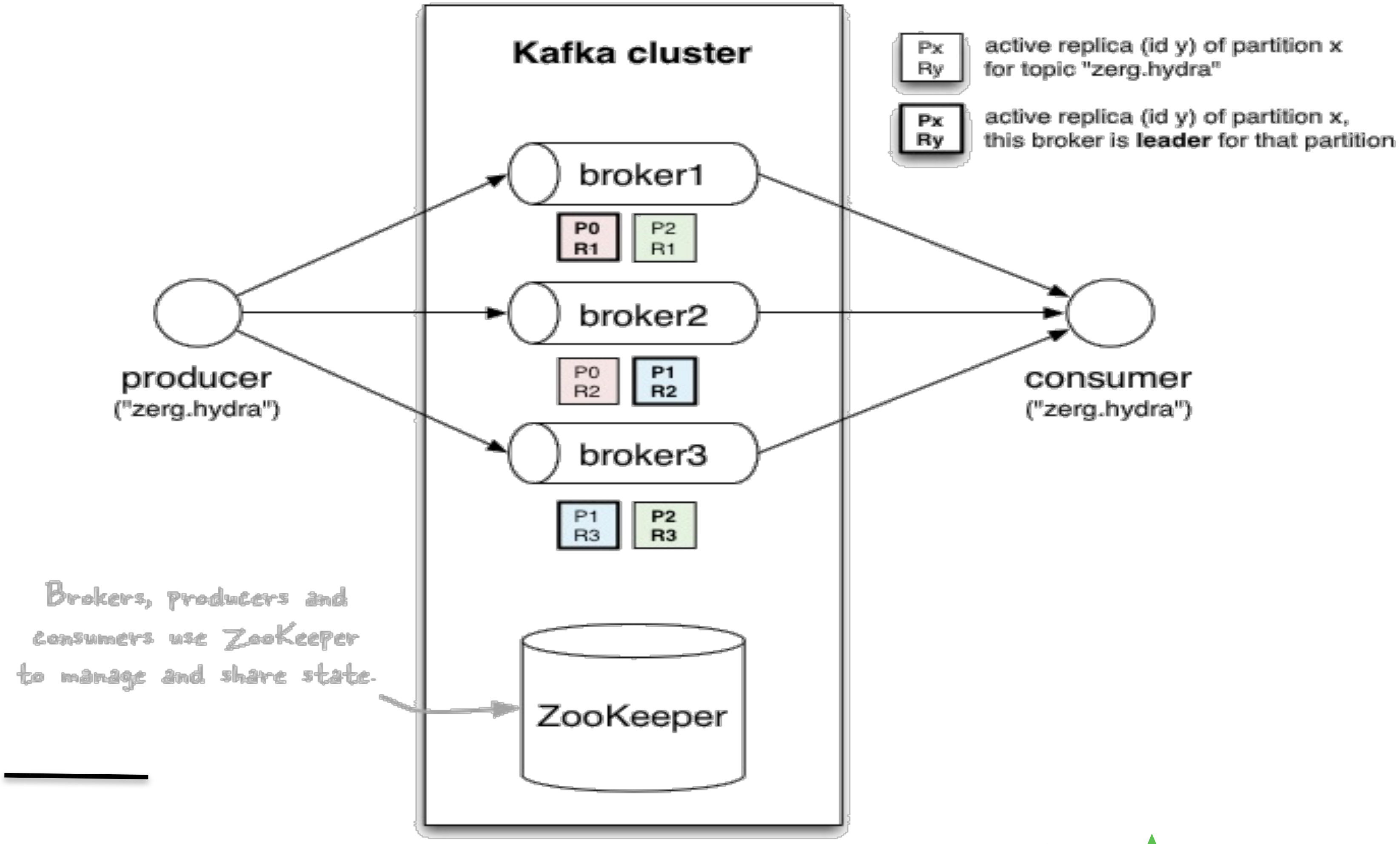
Logs: pub/sub done right



Kafka Overview

- **Producers** write data to **brokers**.
- **Consumers** read data from **brokers**.
- **Brokers** - Each server running Kafka is called a broker.
- All this is distributed.
- Data
  - Data is stored in **topics**.
  - **Topics** are split into **partitions**, which are **replicated**.
- **Built in Parallelism and Scale**



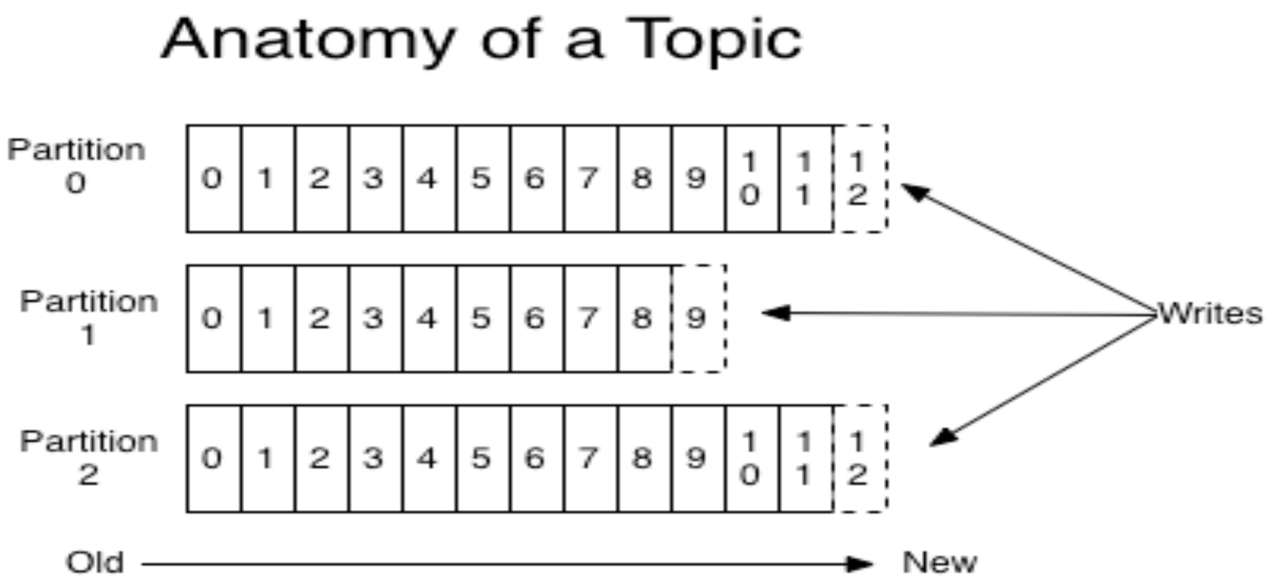


http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

Partitions

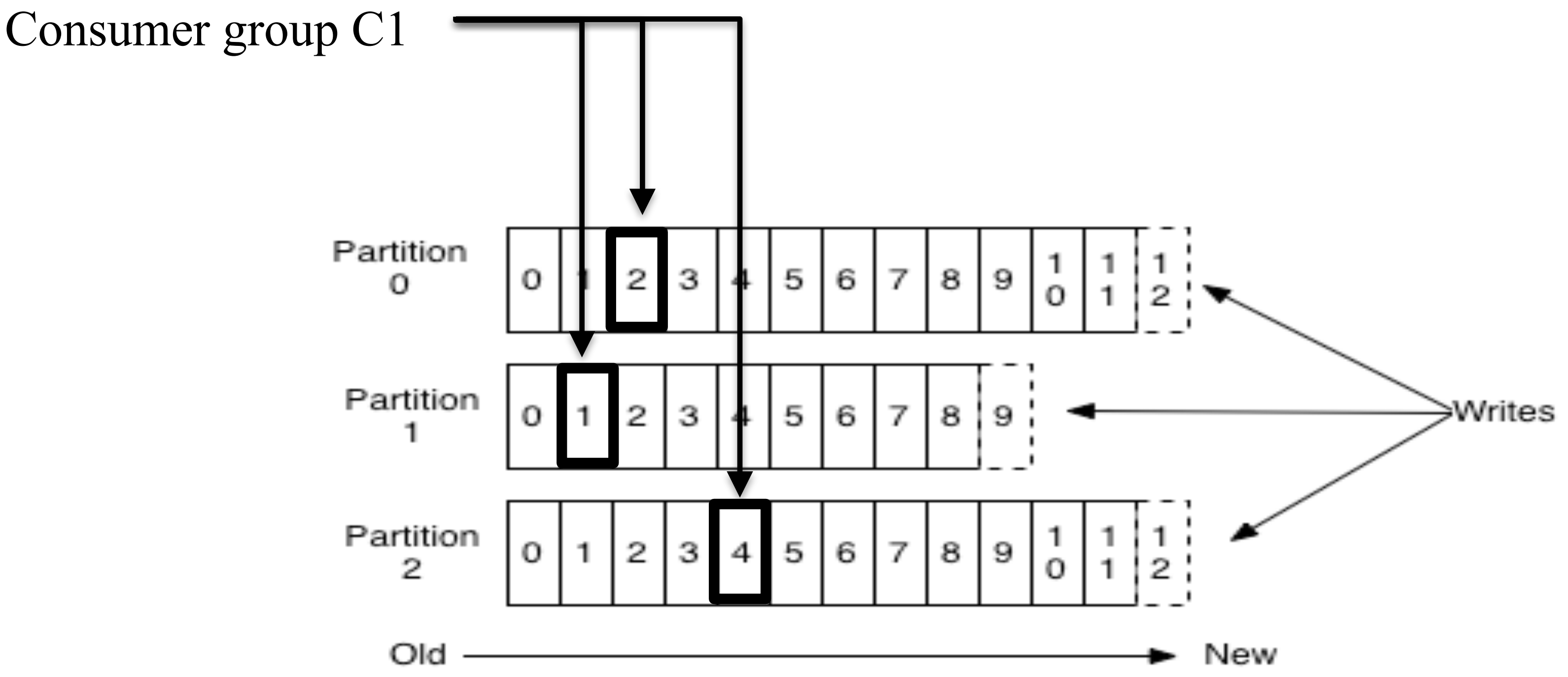
A topic consists of **partitions.**

Partition: **ordered + immutable** sequence of messages that is continually appended to



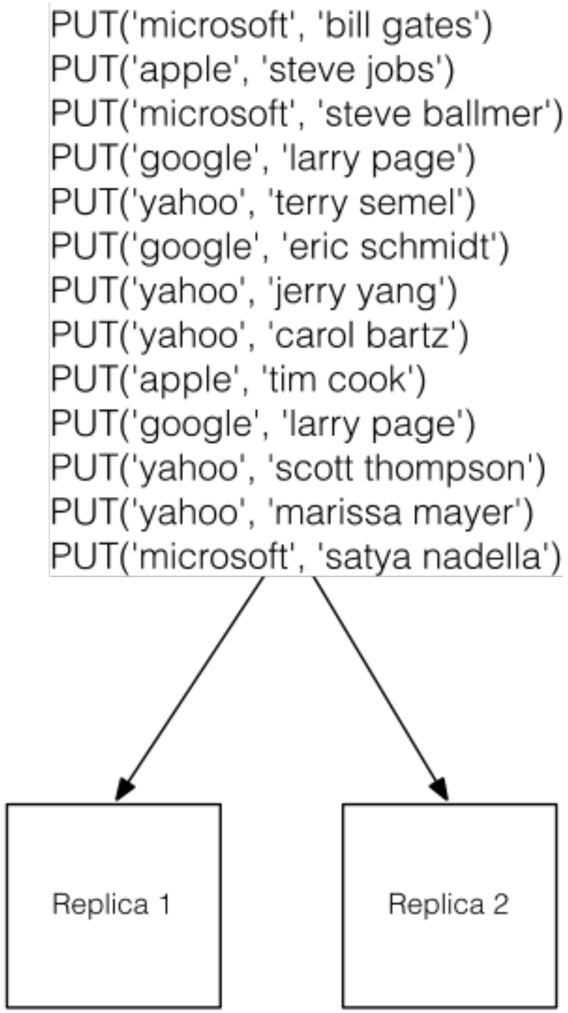
Partition offsets

- **Offset**: messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via *(offset, partition, topic)* tuples



Example:
A Fault-tolerant CEO Hash Table

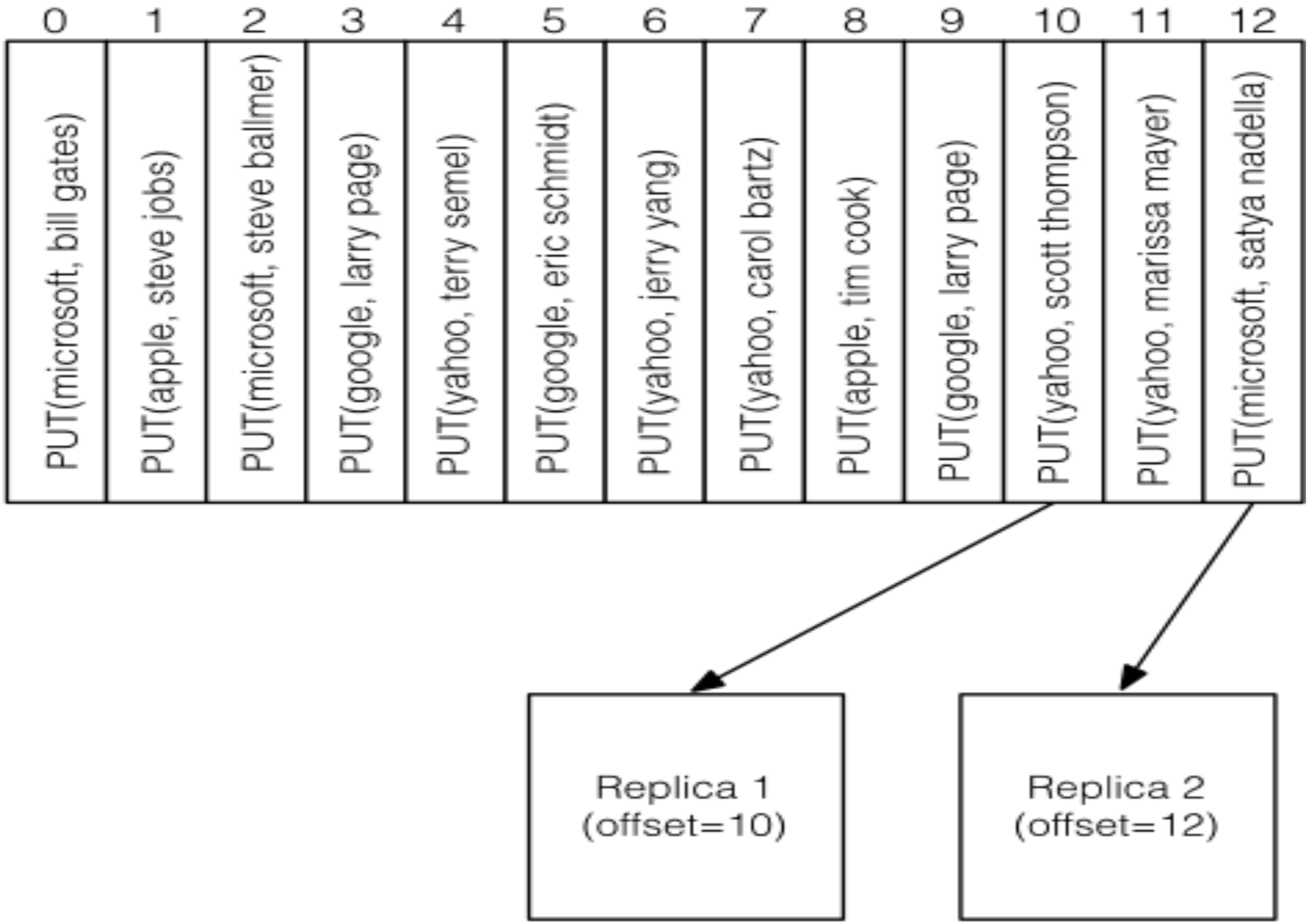
Operations



Final State

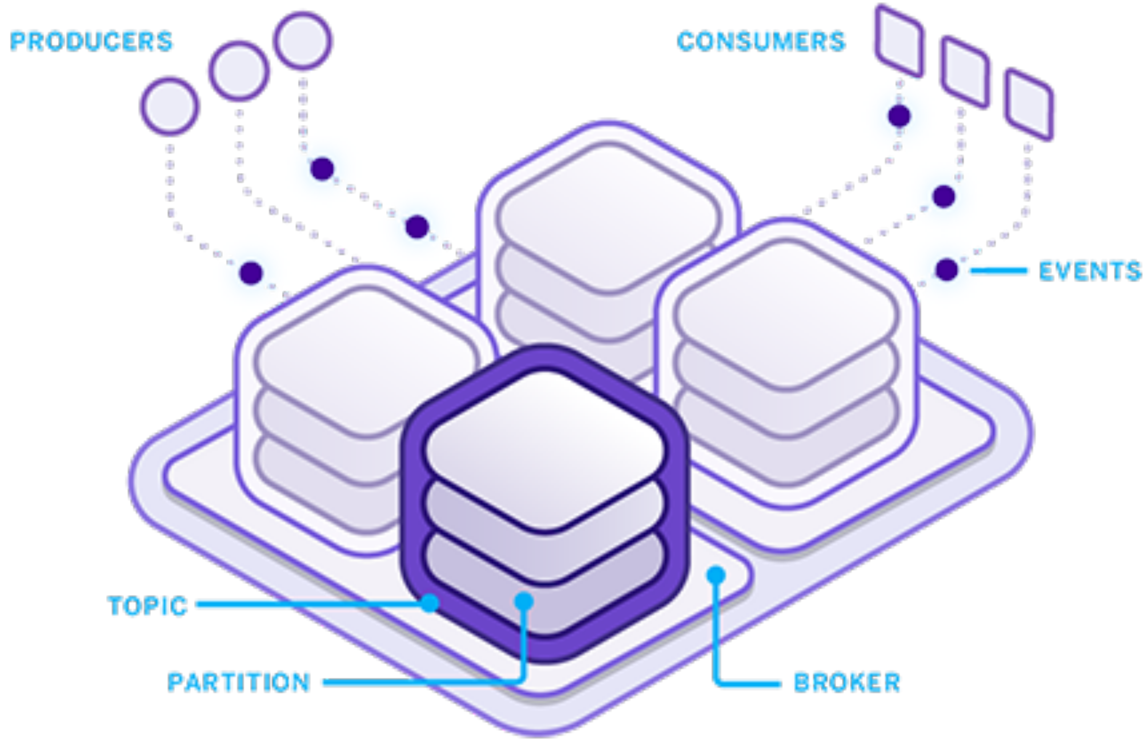
```
{
  'microsoft': 'satya nadella',
  'apple': 'tim cook',
  'google': 'larry page',
  'yahoo': 'marissa mayer'
}
```

Kafka Log


0
1
2
3
4
5
6
7
8
9
10
11
12
PUT(microsoft, bill gates)
PUT(apple, steve jobs)
PUT(microsoft, steve ballmer)
PUT(google, larry page)
PUT(yahoo, terry semel)
PUT(google, eric schmidt)
PUT(yahoo, jerry yang)
PUT(yahoo, carol bartz)
PUT(apple, tim cook)
PUT(google, larry page)
PUT(yahoo, scott thompson)
PUT(yahoo, marissa mayer)
PUT(microsoft, satya nadella)
Replica 1
(offset=10)
Replica 2
(offset=12)

Heroku Kafka

- Managed Kafka Cloud Service
- https://www.heroku.com/kafka



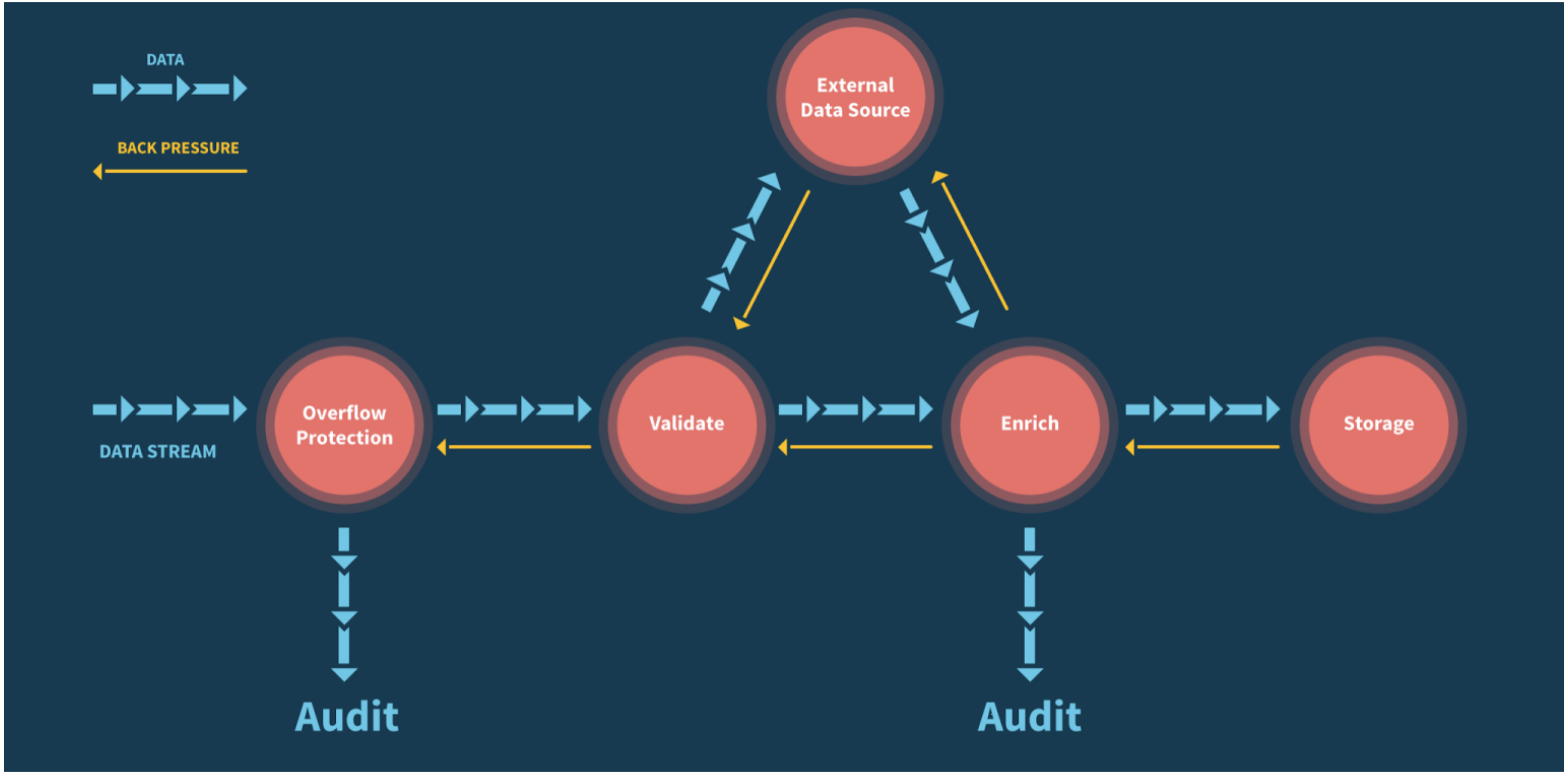
Code

Akka Streams

Reactive Streams Built on Akka

Reactive Streams

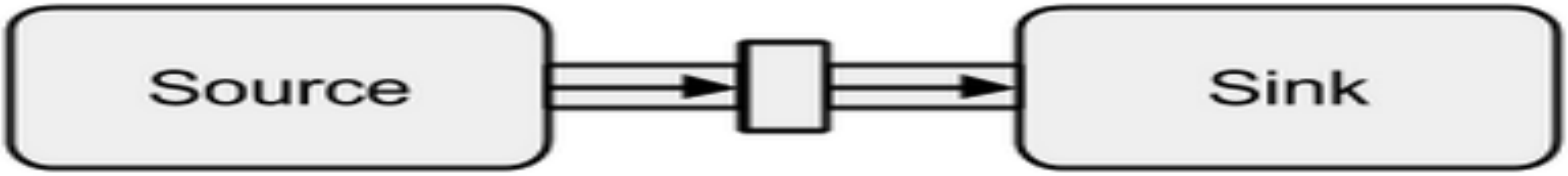
A JVM standard for asynchronous stream processing with non-blocking back pressure



Akka Streams

- Powered by Akka Actors
- Impl of Reactive Streams
- Actors can be used directly or just internally
- Stream processing functions: map, filter, fold, etc

Sink & Source




```
val source = Source.repeat("hello, world")
val sink = Sink.foreach(println)
val flow = source to sink
flow.run()
```

Code

Play Framework

Web Framework Built on Akka Streams

Play Framework

Scala & Java – Built on Akka Streams

Declarative Routing:

```
GET   /foo   controllers.Foo.do
```

Controllers Hold Stateless Functions:

```
class Foo {
  def do() = Action {
    Ok("hello, world")
  }
}
```

Reactive Requests

Don't block in wait states!

```
def doLater = Action.async {
  Promise.timeout(Ok("hello, world"), 5.seconds)
}


def reactiveRest = Action.async {
  ws.url("http://api.foo.com/bar").get().map { response =>
    Ok(response.json)
  }
}
```

WebSockets

Built on Akka Streams

```
def ws = WebSocket.accept { request =>
    val sink = ...
    val source = ...
    Flow.fromSinkAndSource(Sink.ignore, source)
}
```

Views

Serverside Templating with a Subset of Scala

```
app/views/blah.scala.html


@(foo: String)
<html>
<body>
  @foo
</body>
</html>
```

```
Action {
  Ok(views.html.blah("bar"))
}

<html>
<body>
  bar
</body>
</html>
```

Demo & Code

Flink

Real-time Data Analytics

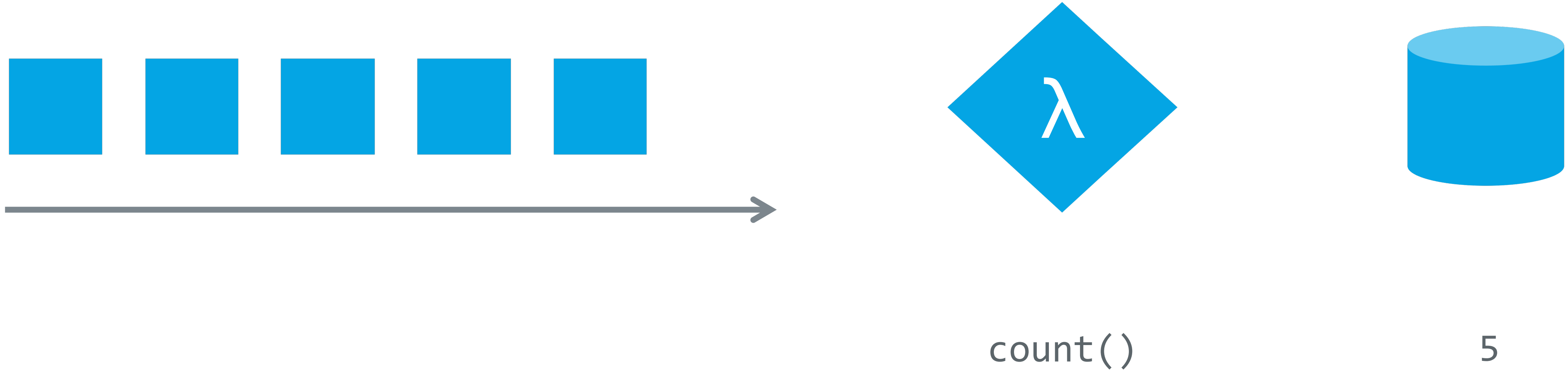
Flink

Real-time Data Analytics

- Bounded & Unbounded Data Sets
- Stream processing
- Distributed Core
  - Fault Tolerant
  - Clustered
- Flexible Windowing

Apache Flink

Continuous Processing for Unbounded Datasets



Windowing

Bounding with Time, Count, Session, or Data

Batch Processing

Stream Processing on Finite Streams

Data Processing

What can we do?

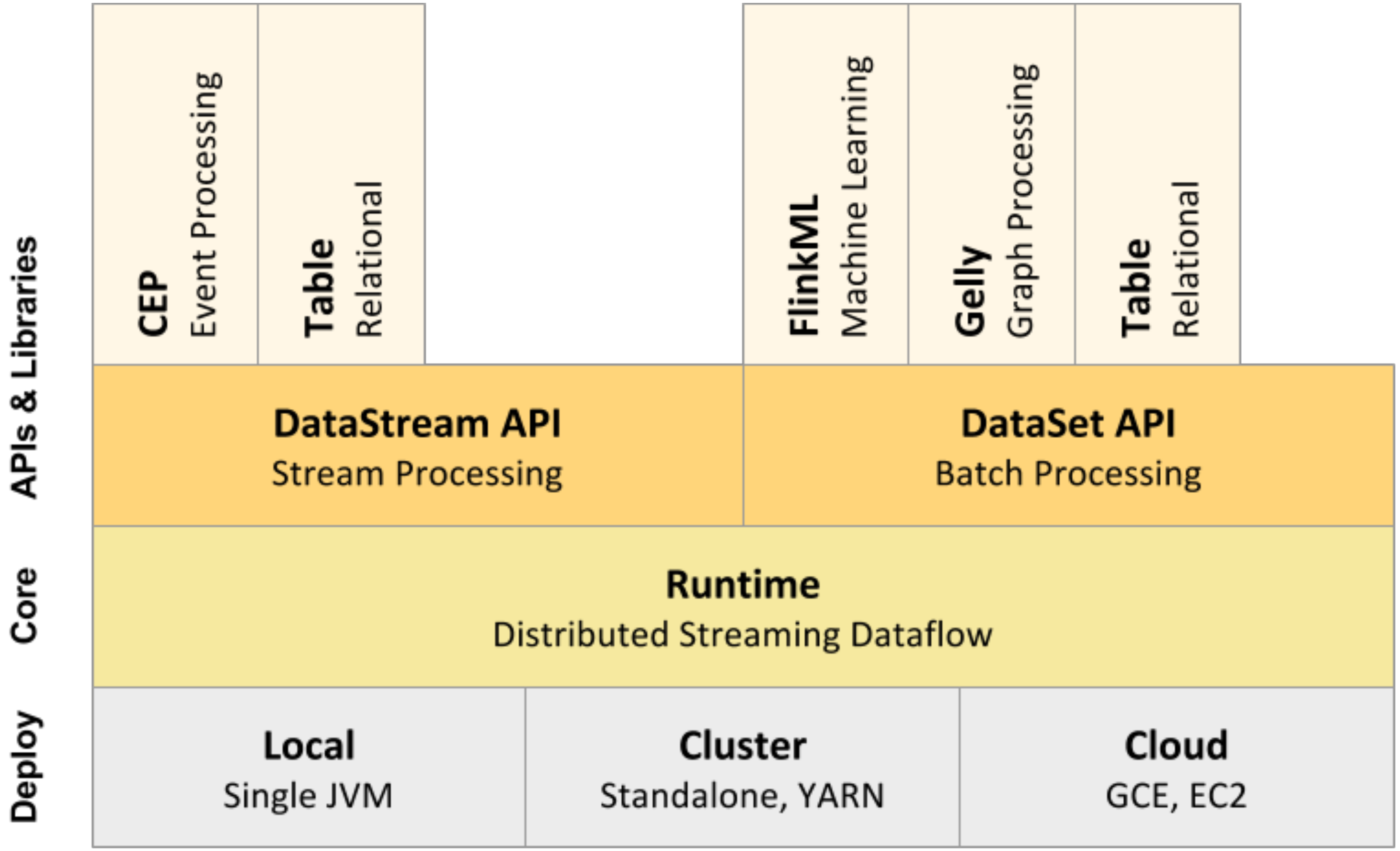
- Aggregate / Accumulate
- Transform
- Filter
- Sort

λ

```
fold(), reduce(), sum(), min()
map(), flatMap()
filter(), distinct()
sortGroup(), sortPartition()
```

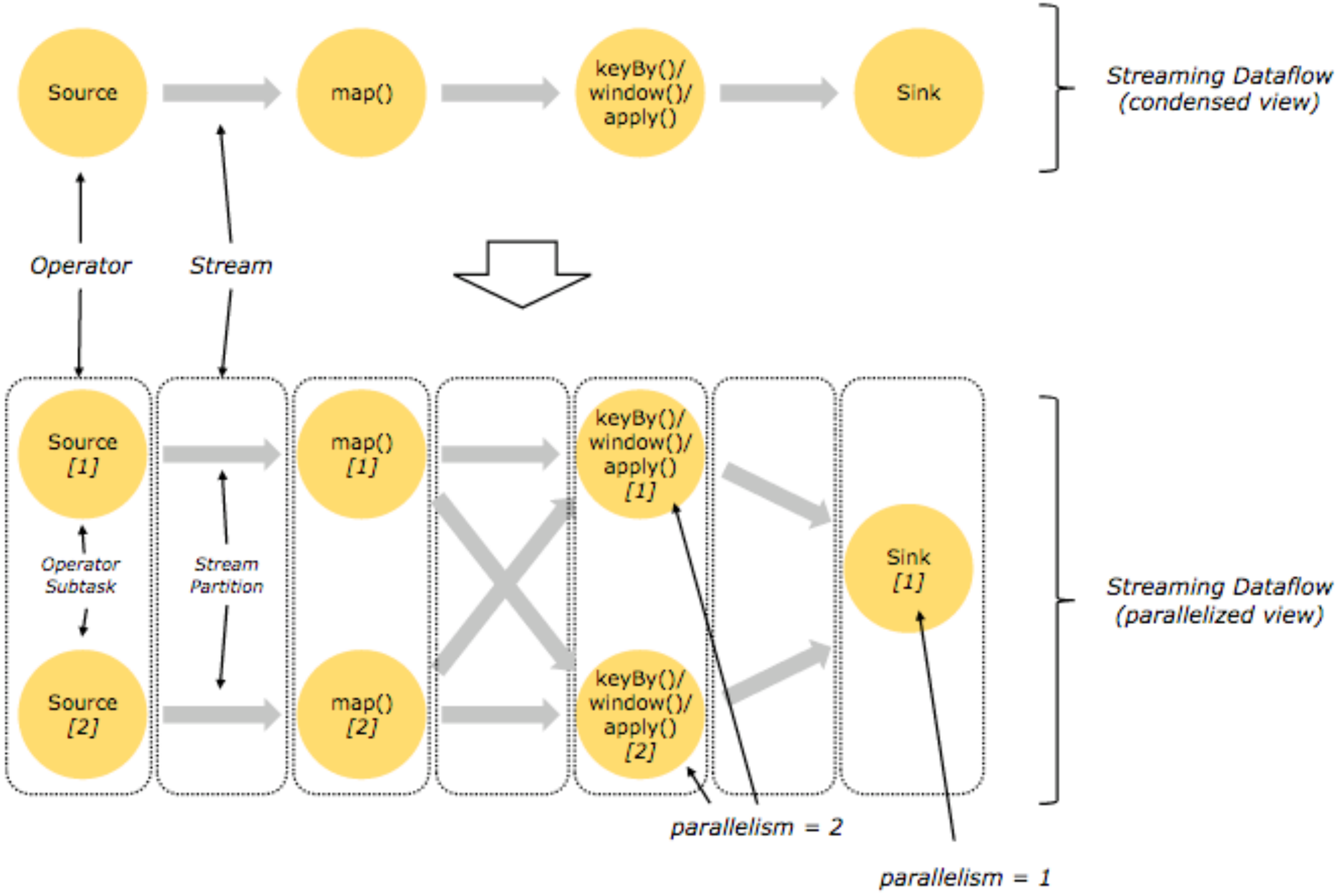
Apache Flink

Architecture



Partitioning

Network Distribution



Demo & Code

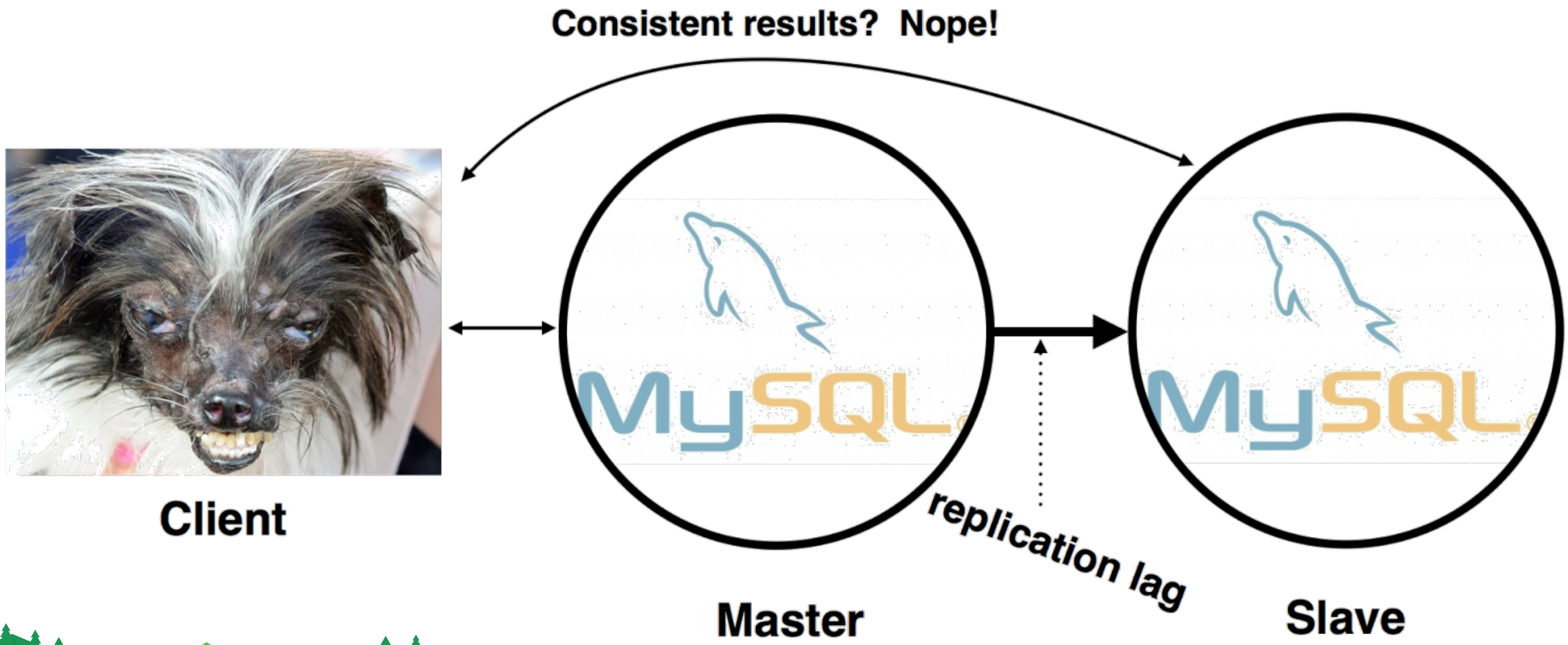
Cassandra

Distributed NoSQL Database

Challenges with Relational Databases

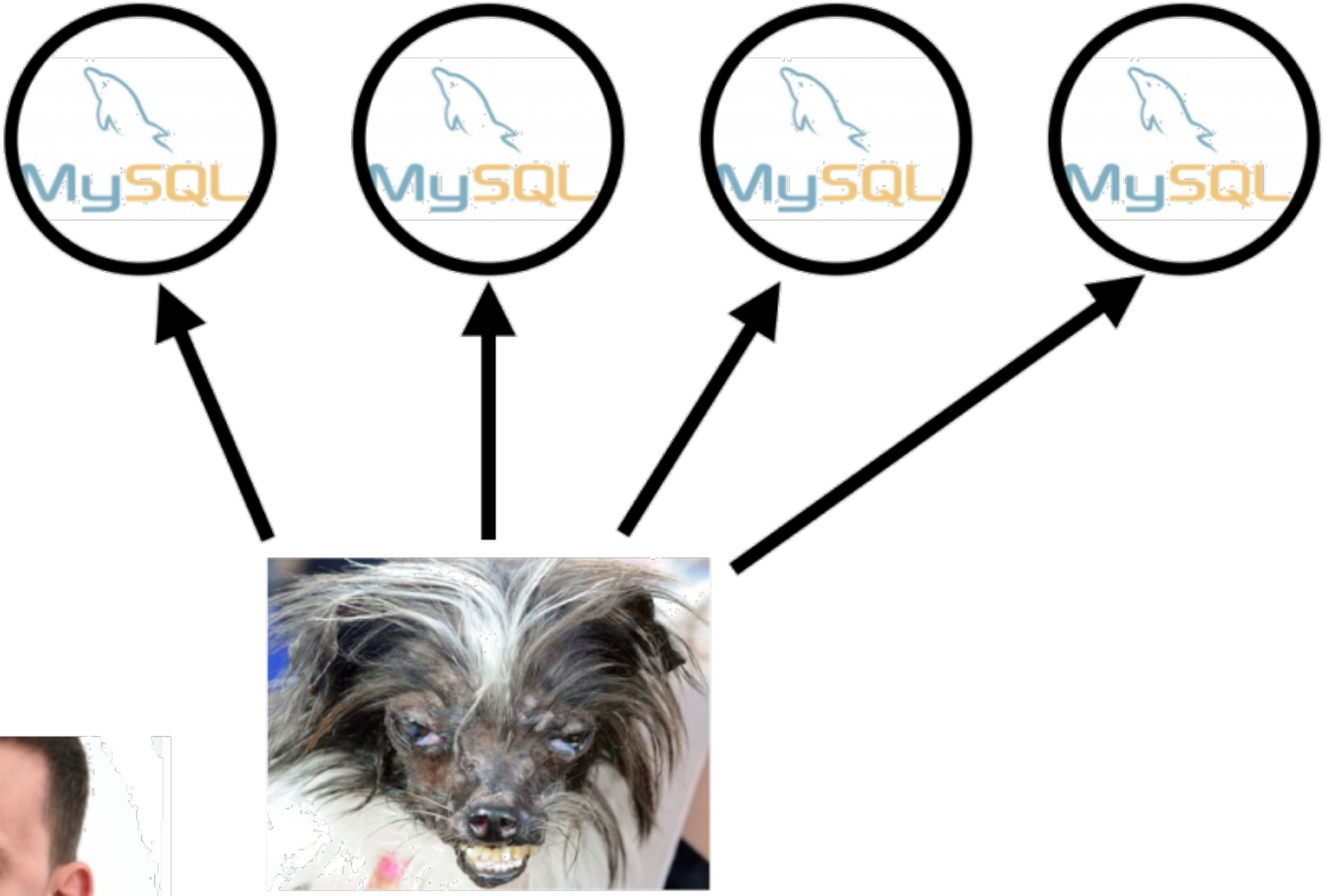
- How do you scale and maintain high-availability with a monolithic database?
- Is it possible to have ACID compliant distributed transactions?
- How can I synchronize a distributed data store?
- How do I resolve differing views of data?

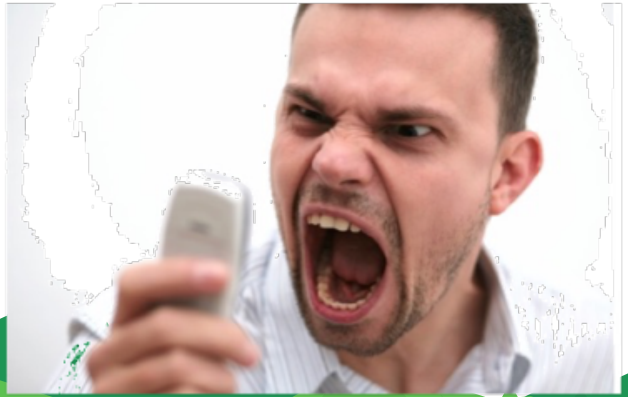
Replication: ACID is a lie



Sharding is a Nightmare

- Data is all over the place
- No more joins
- No more aggregations
- Denormalize all the things
- Querying secondary indexes requires hitting every shard
- Adding shards requires manually moving data
- Schema changes





High Availability.. not really

- Master failover... who's responsible?
  - Another moving part...
  - Bolted on hack
- Multi-DC is a mess
- Downtime is frequent
  - Change database settings (innodb buffer pool, etc)
  - Drive, power supply failures
  - OS updates

Goals of a Distributed Database

- Consistency is not practical - **give it up!**
- Manual sharding & rebalancing is hard - **Automatic Sharding!**
- Every moving part makes systems more complex
- Master / slave creates a Single Point of Failure / Bottleneck - **Simplify Architecture**!
- Scaling up is expensive - **Reduce Cost**
- Leverage cloud / commodity hardware

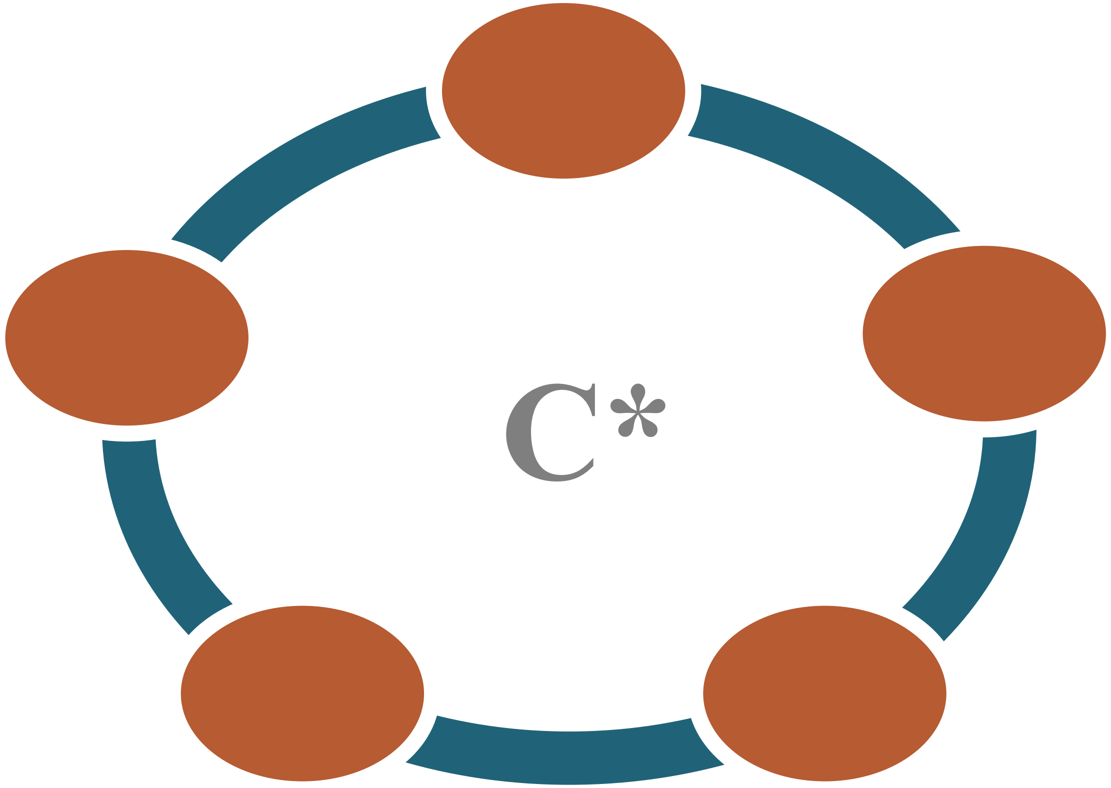
What is Cassandra?

Distributed Database

✓ Individual DBs (nodes)

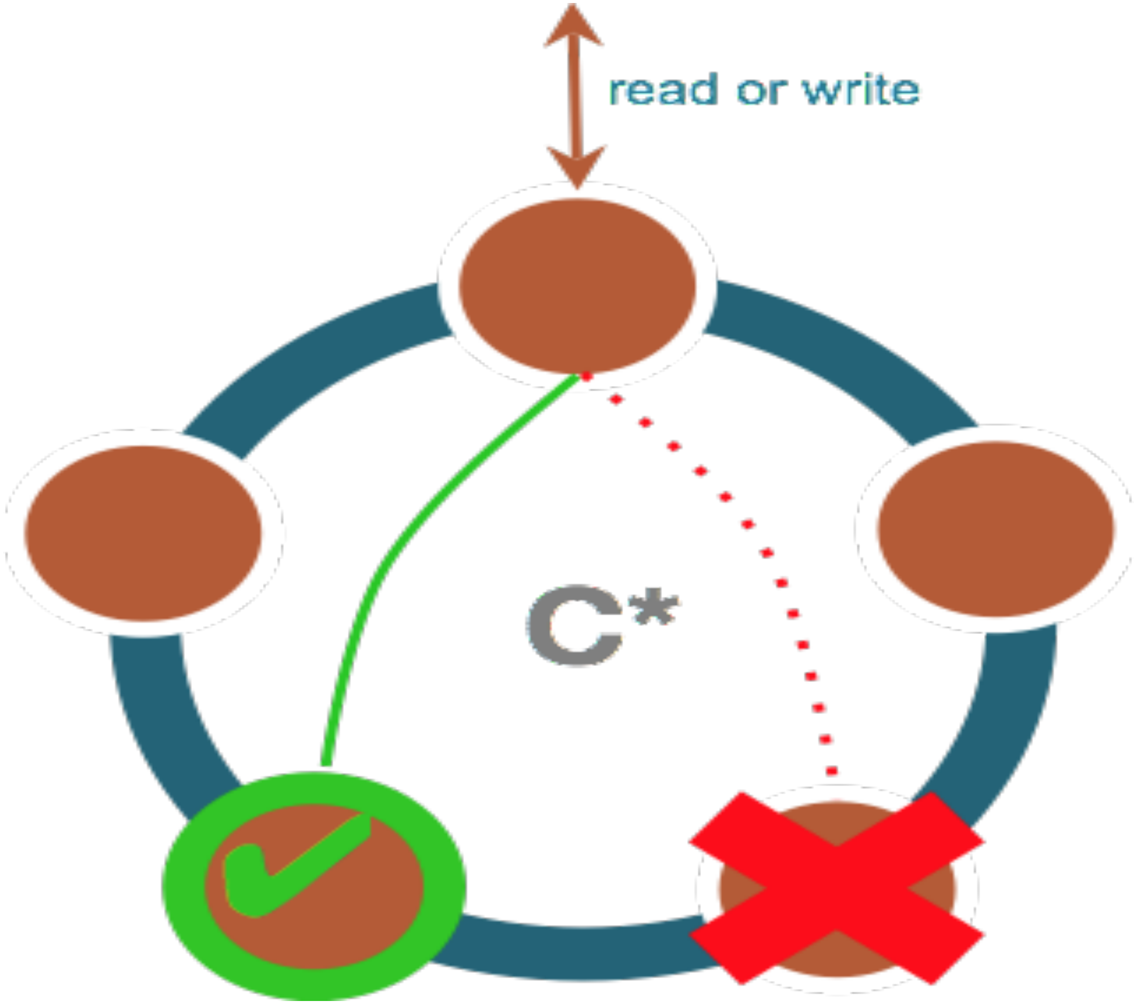
✓ Working in a cluster

✓ Nothing is shared

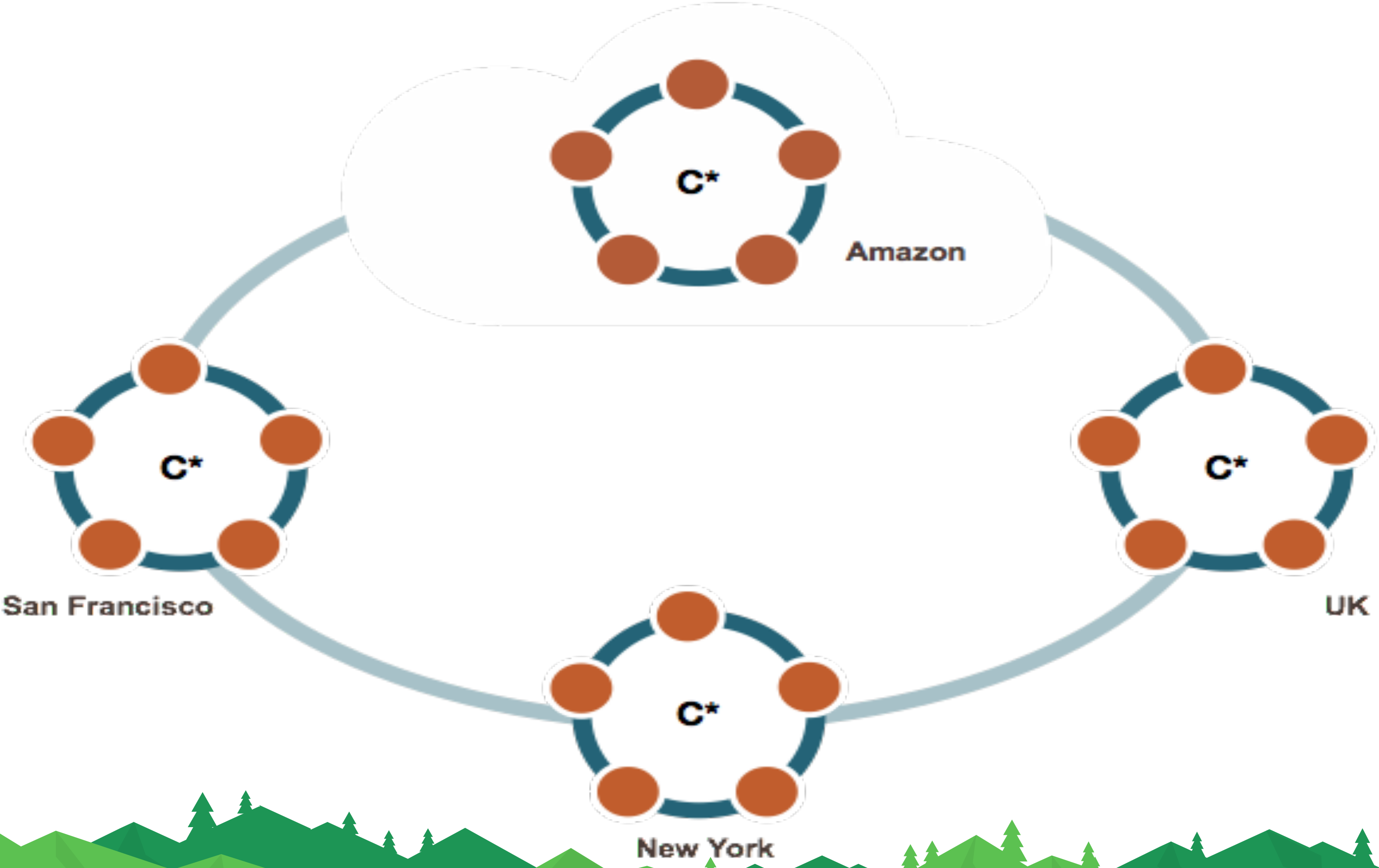


Cassandra Cluster

- Nodes in a peer-to-peer cluster
    - No single point of failure
- Built in data replication
    - Data is always available
    - 100% Uptime
- Across data centers
    - Failure avoidance



Multi-Data Center Design



Why Cassandra?

It has a flexible data model

Tables, wide rows, partitioned and distributed

- ✓ Data
- ✓ Blobs (documents, files, images)
- ✓ Collections (Sets, Lists, Maps)
- ✓ UDTs

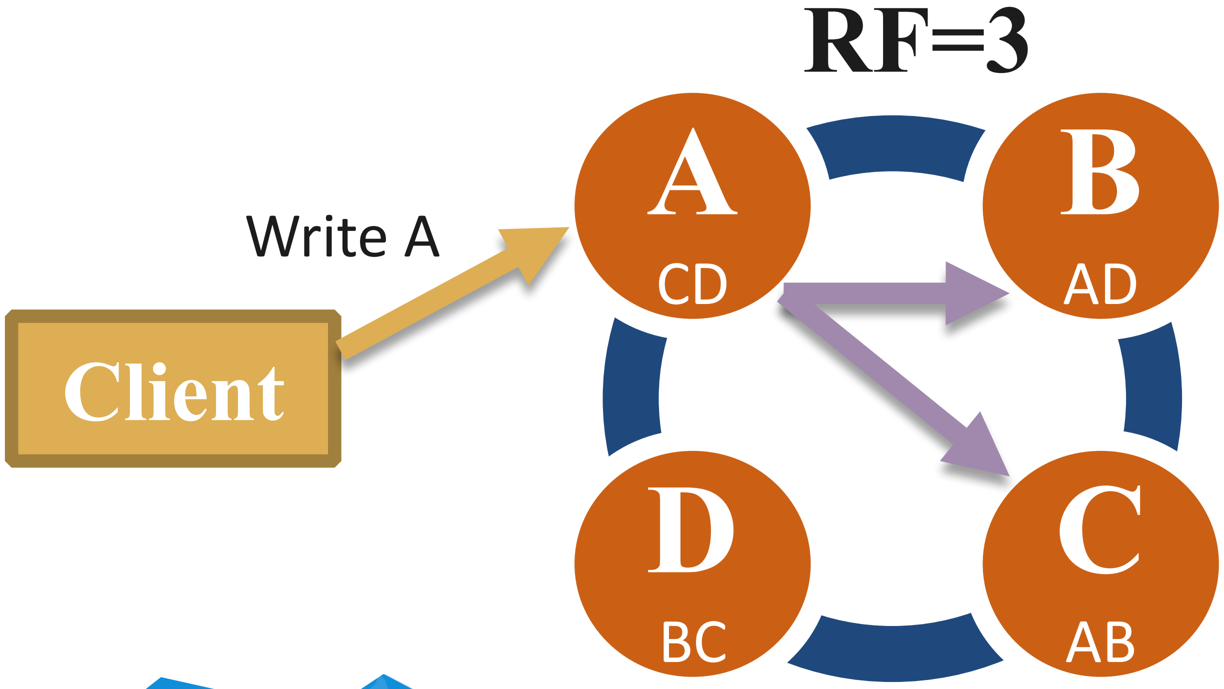
Access it with CQL ← familiar syntax to SQL

Row Key1	Column Key1	Column Key2	Column Key3	...
	Column Value1	Column Value2	Column Value3	...
⋮				

Two knobs control Cassandra fault tolerance

Replication Factor (server side)

How many copies of the data should exist?

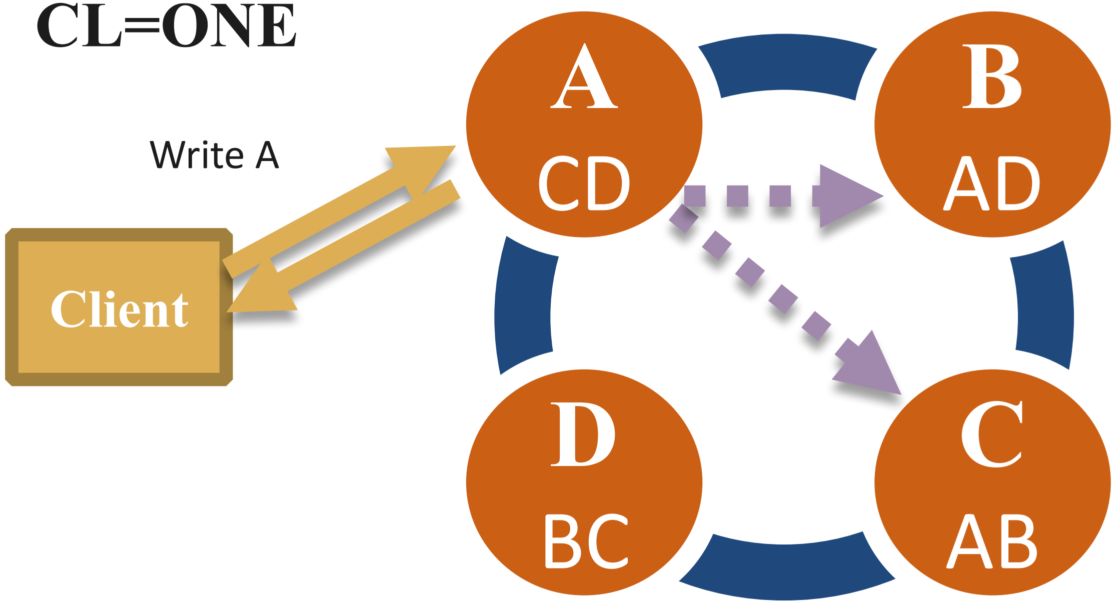


Two knobs control Cassandra fault tolerance

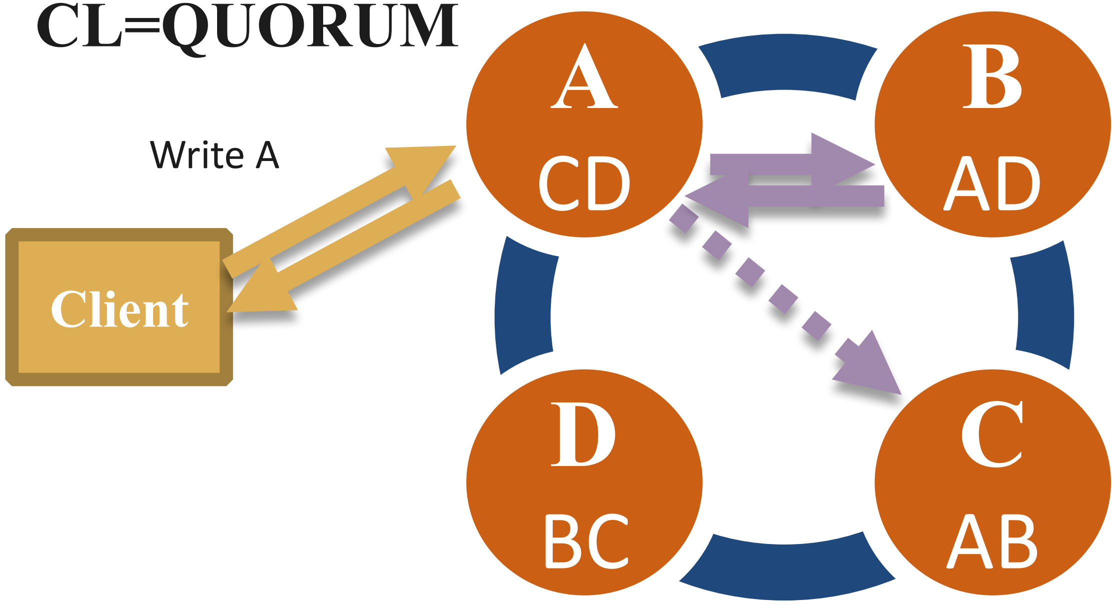
Consistency Level (client side)

How many replicas do we need to hear from before we acknowledge?

CL=ONE



CL=QUORUM



Consistency Levels

Applies to both Reads and Writes (i.e. is set on each query)

ONE – one replica from any DC

LOCAL_ONE – one replica from local DC

QUORUM – 51% of replicas from any DC

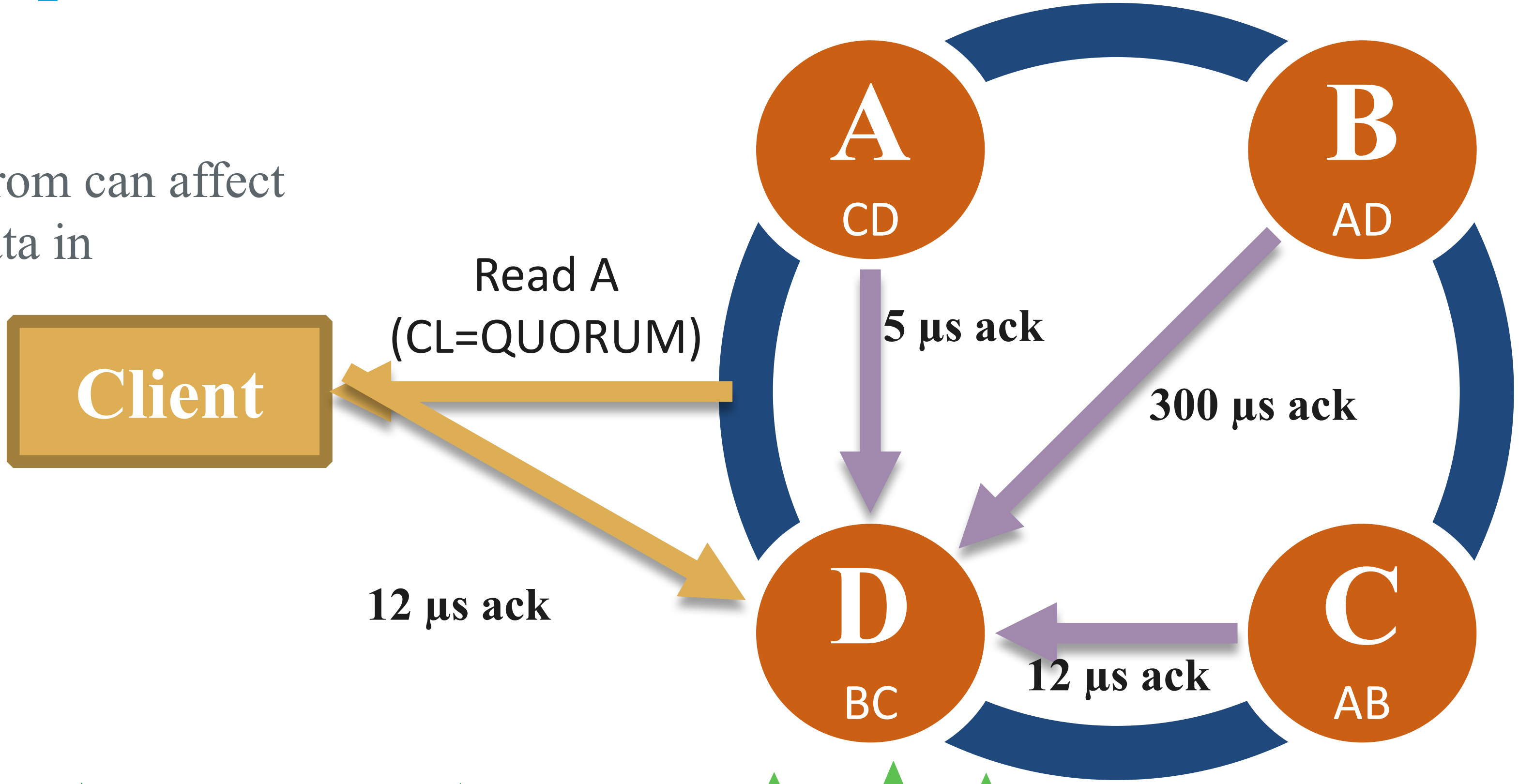
LOCAL_QUORUM – 51% of replicas from local DC

ALL – all replicas

TWO

Consistency Level and Speed

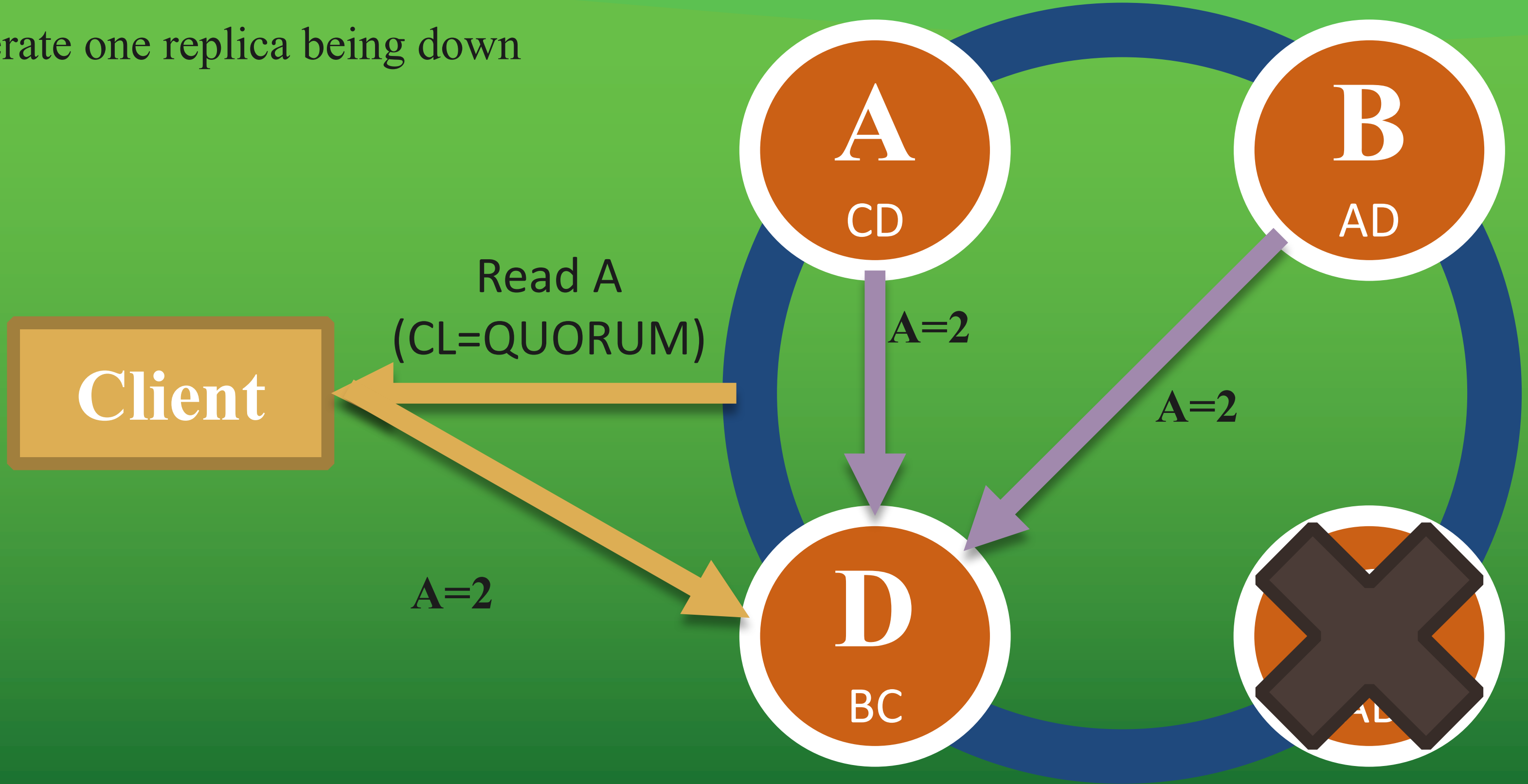
How many replicas we need to hear from can affect how quickly we can read and write data in Cassandra?



Consistency Level and Availability

Consistency Level choice affects availability

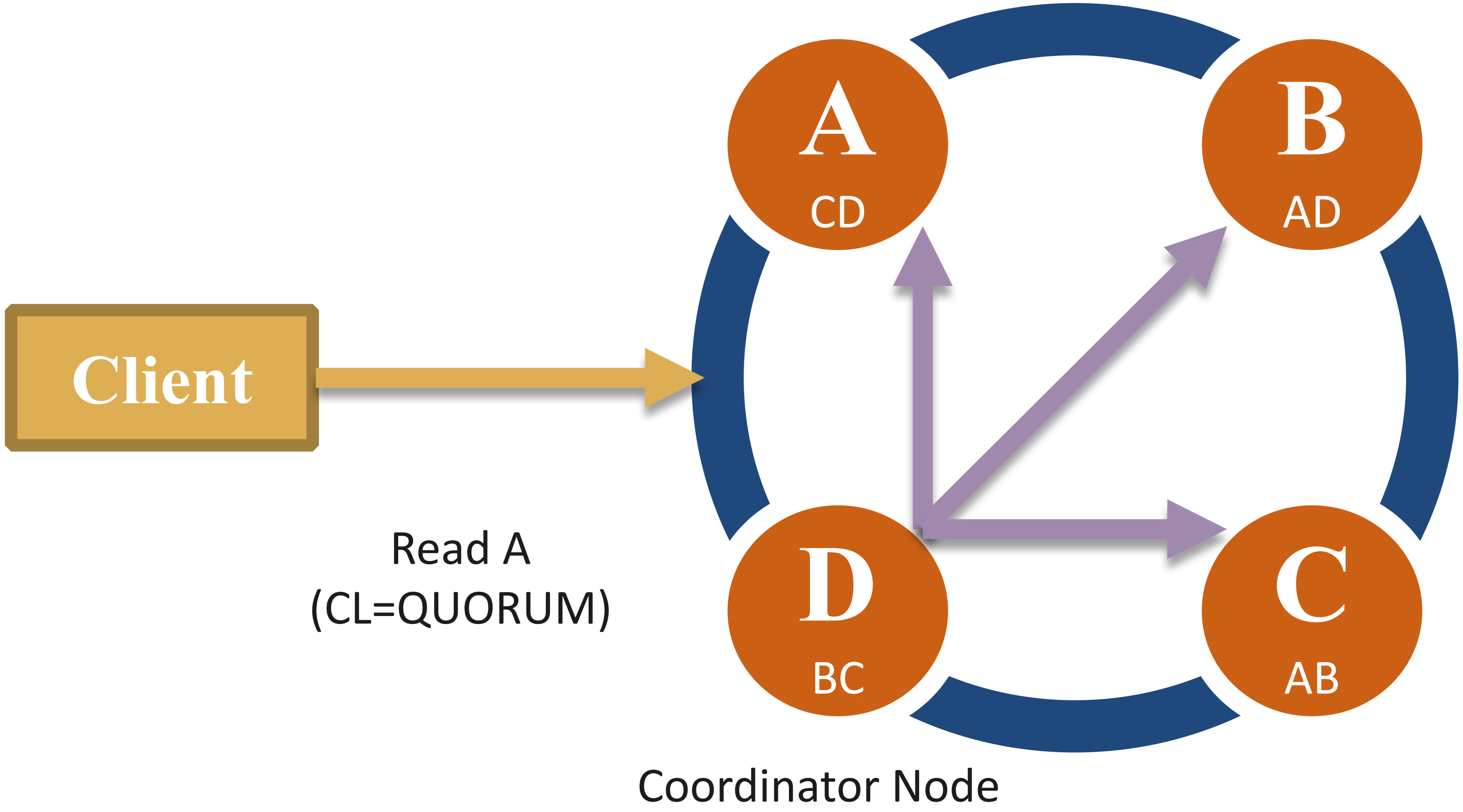
For example, QUORUM can tolerate one replica being down and still be available (in RF=3)



Reads in the cluster

Same as writes in the cluster, reads are coordinated

Any node can be the Coordinator Node



Spark Cassandra Connector

Spark Cassandra Connector

Data locality-aware (speed)

Read from and Write to Cassandra

Cassandra Tables Exposed as RDD and DataFrames

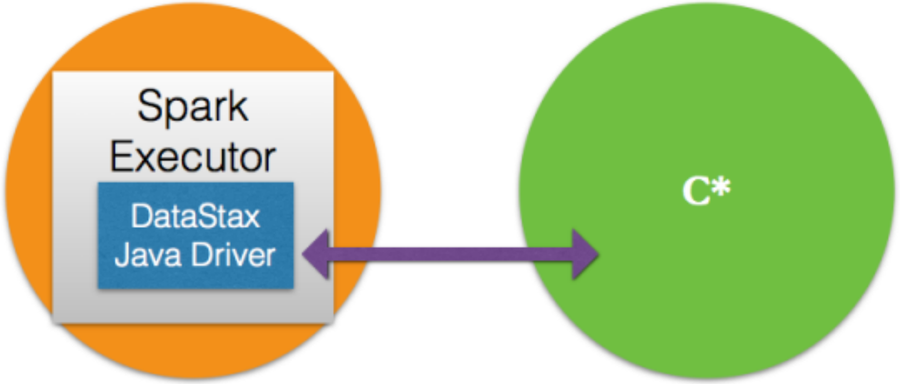
Server-Side filters (where clauses)

Cross-table operations (JOIN, UNION, etc.)

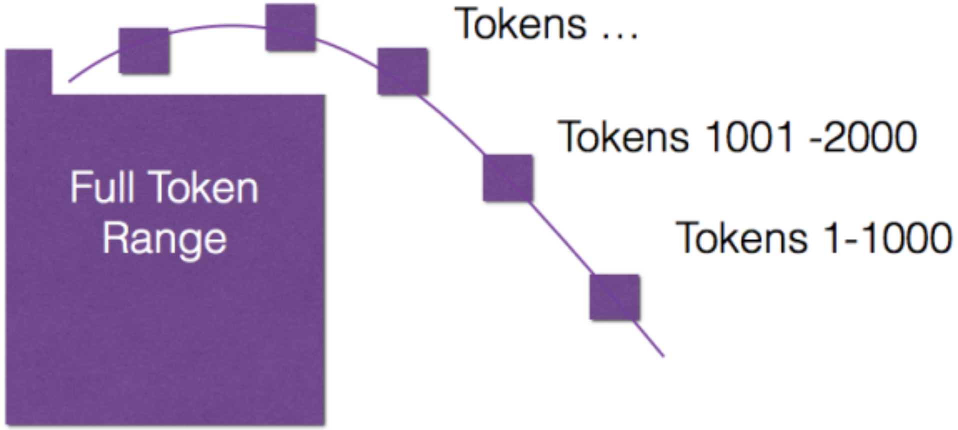
Mapping of Java Types to Cassandra Types

Spark Cassandra Connector

Spark Cassandra Connector uses the DataStax Java Driver to Read from and Write to C*



Each Executor Maintains a connection to the C* Cluster



RDD's read into different splits based on sets of tokens

Code

Spark Streaming

Stream Processing Built on Spark

Hadoop?

Hadoop Limitations

- Master / Slave Architecture
- Every Processing Step requires Disk IO
- Difficult API and Programming Model
- Designed for batch-mode jobs
- No even-streaming / real-time
- Complex Ecosystem

What is Spark?

Fast and general compute engine for large-scale data processing

Fault Tolerant Distributed Datasets

Distributed Transformation on Datasets

Integrated Batch, Iterative and Streaming Analysis

In Memory Storage with Spill-over to Disk

Advantages of Spark

- Improves efficiency through:
  - In-memory data sharing
  - General computation graphs - Lazy Evaluates Data
  - 10x faster on disk, 100x faster in memory than Hadoop MR
- Improves usability through:
  - Rich APIs in Java, Scala, Py..??
  - 2 to 5x less code
  - Interactive shell

Spark Streaming
real-time

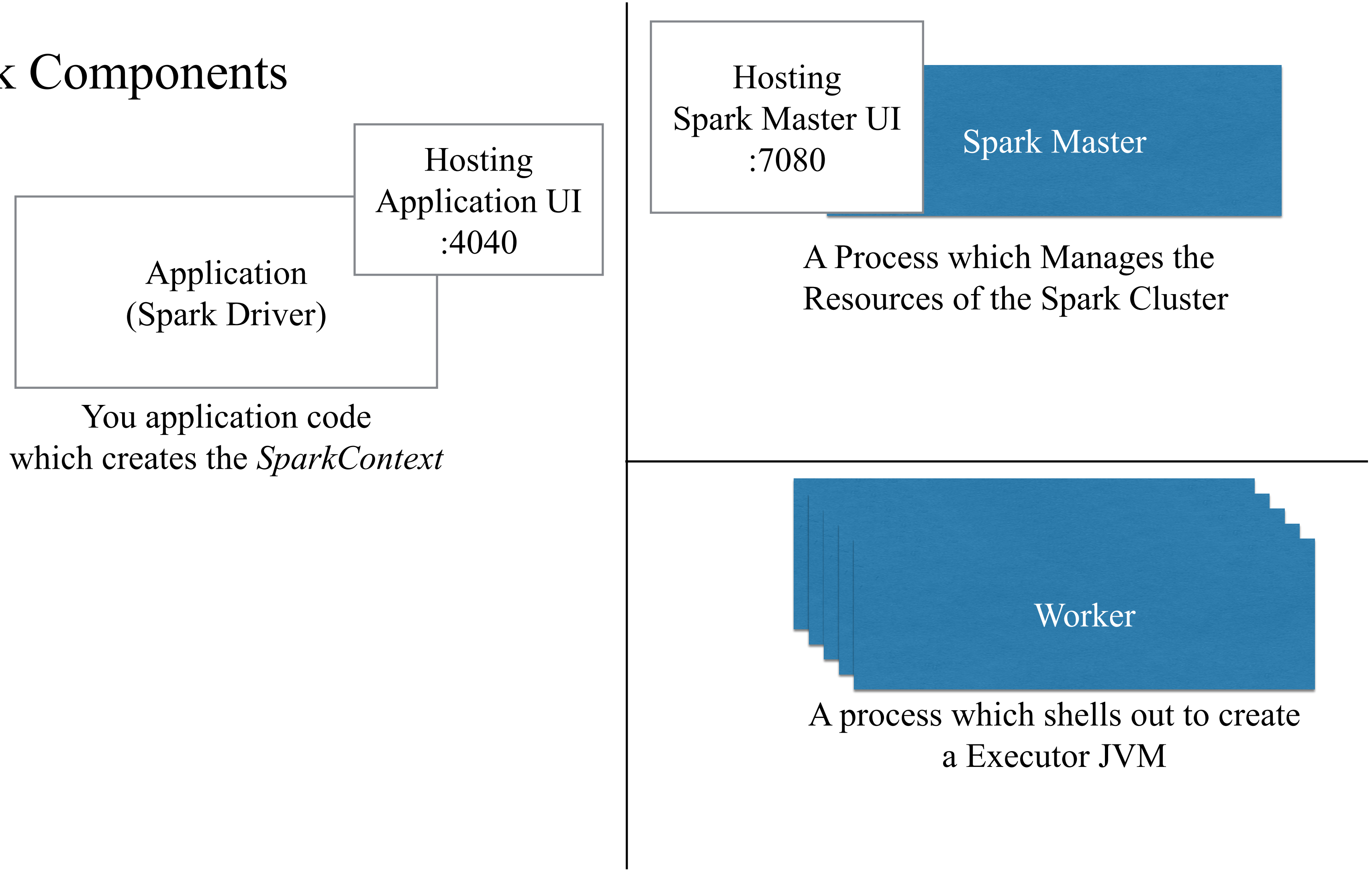
Spark SQL
structured

MLlib
machine learning

GraphX
graph

Spark Core

Spark Components



These processes are all separate and require networking to communicate

Resilient Distributed Datasets (RDD)

- The primary abstraction in Spark
- Collection of data stored in the Spark Cluster
- Fault-tolerant
- Enables parallel processing on data sets
- In-Memory or On-Disk

RDD Operations

Transformations - Similar to scala collections API

Produce new RDDs:

`filter, flatmap, map, distinct, groupBy, union, zip, reduceByKey, subtract`

Actions - Require materialization of the records to generate a value

`collect: Array[T], count, fold, reduce..`

DataFrame

- Distributed collection of data
- Similar to a Table in a RDBMS
- Common API for reading/writing data
- API for selecting, filtering, aggregating and plotting structured data

DataFrame Part 2

- Sources such as Cassandra, structured data files, tables in Hive, external databases, or existing RDDs.
- Optimization and code generation through the Spark SQL Catalyst optimizer
- Decorator around RDD - Previously SchemaRDD

Spark Versus Spark Streaming

zillions of bytes

Spark

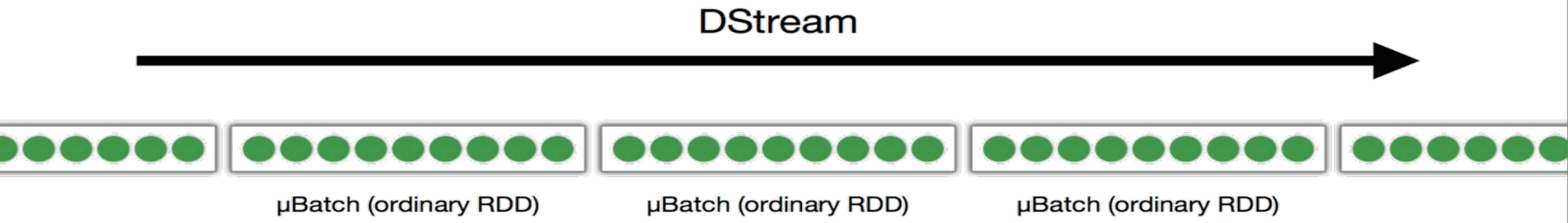
gigabytes per second

Spark Streaming

Spark Streaming Data Sources

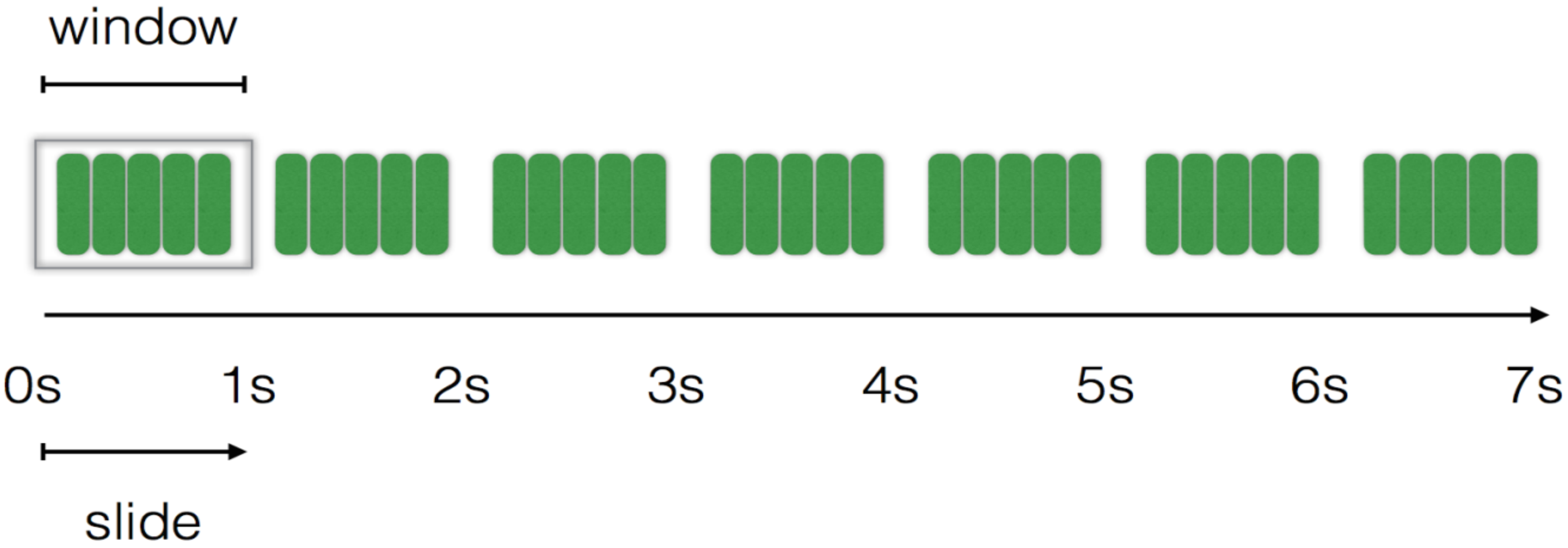
Spark Streaming General Architecture

DStream Micro Batches



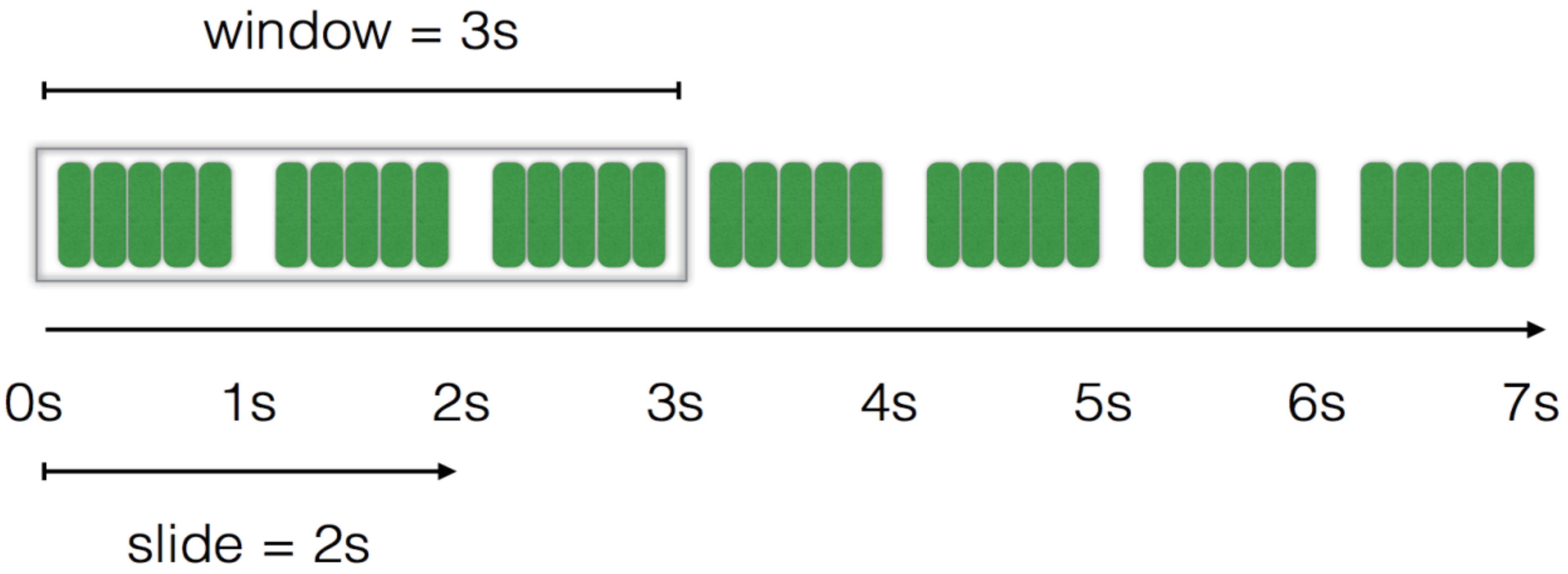
Processing of DStream = Processing of μBatches, RDDs

Windowing



By default:
window = slide = batch duration

Windowing



The resulting DStream consists of 3 seconds micro-batches
Each resulting micro-batch overlaps the preceding one by 1 second

Streaming Resiliency without Kafka

- Streaming uses aggressive checkpointing and in-memory data replication to improve resiliency.
- Frequent checkpointing keeps RDD lineages down to a reasonable size.
- Checkpointing and replication mandatory since streams don't have source data files to reconstruct lost RDD partitions (except for the directory ingest case).
- Write Ahead Logging to prevent Data Loss

Direct Kafka Streaming w/ Kafka Direct API

- Use Kafka Direct Approach (No Receivers)
  - Queries Kafka Directly
  - Automatically Parallelizes based on Kafka Partitions
  - (Mostly) Exactly Once Processing - Only Move Offset after Processing
  - Resiliency without copying data

Demo & Code