CS5412 / Lecture 21 Apache Tools – Part 2

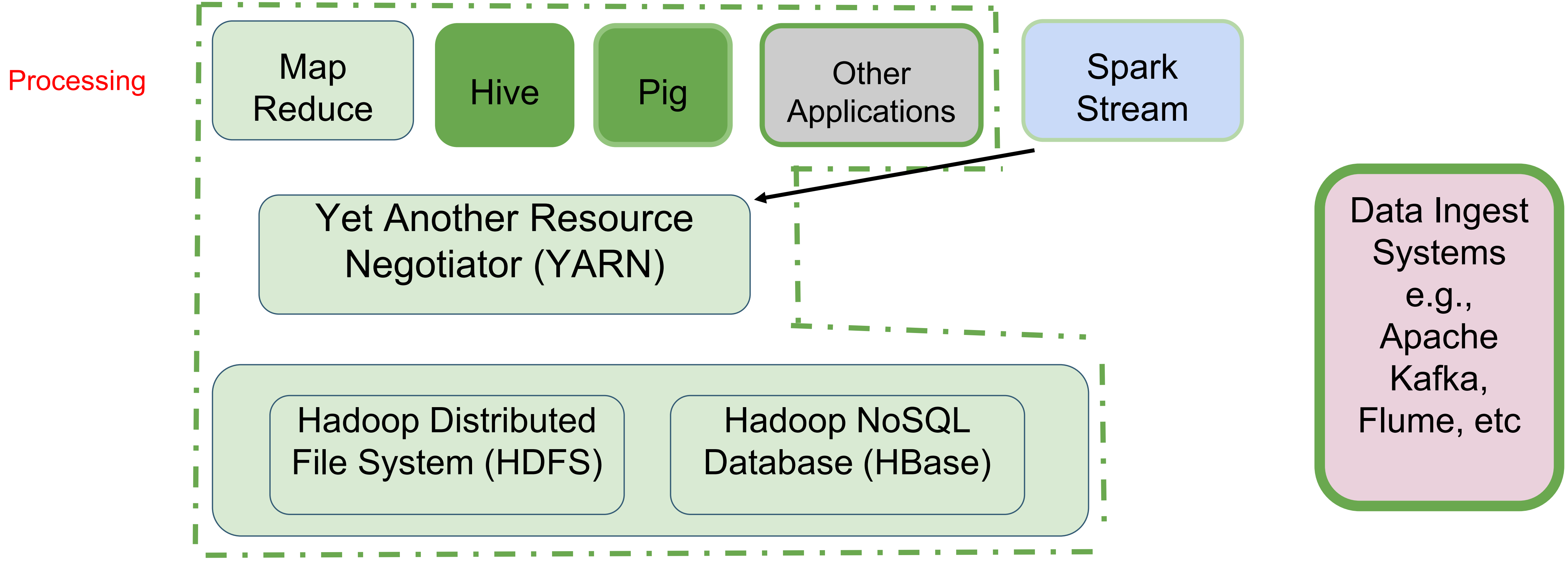
Ken Birman & Kishore Pusukuri, Spring 2019

PUTTING IT ALL TOGETHER

Reminder: Apache Hadoop Ecosystem

- HDFS (Distributed File System)
- HBase (Distributed NoSQL Database -- distributed map)
- YARN (Resource Manager)
- MapReduce (Data Processing Framework)

Hadoop Ecosystem: Processing



Apache Hive: SQL on MapReduce

Hive is an abstraction layer on top of Hadoop (MapReduce/Spark)

Use Cases:

- Data Preparation
- Extraction-Transformation-Loading Jobs (Data Warehousing)
- Data Mining

Apache Hive: SQL on MapReduce

Hive is an abstraction layer on top of Hadoop (MapReduce/Spark)

- ➢ Hive uses a SQL-like language called HiveQL
- ➢ Facilitates reading, writing, and managing large datasets residing in distributed storage using SQL-like queries
- ➢ Hive executes queries using MapReduce (*and also using Spark*)
  - HiveQL queries → Hive → MapReduce Jobs

Apache Hive

- Structure is applied to data at time of read → No need to worry about formatting the data at the time when it is stored in the Hadoop cluster
- Data can be read using any of a variety of formats:
  - Unstructured flat files with comma or space-separated text
  - Semi-structured JSON files (a web standard for event-oriented data such as news feeds, stock quotes, weather warnings, etc)
  - Structured HBase tables
- Hive is not designed for online transaction processing. Hive should be used for “data warehousing” tasks, not arbitrary transactions.

Apache Pig: Scripting on MapReduce

Pig is an abstraction layer on top of Hadoop (MapReduce/Spark)

- ➢ Use Cases:
  - Data Preparation
  - ETL Jobs (Data Warehousing)
  - Data Mining

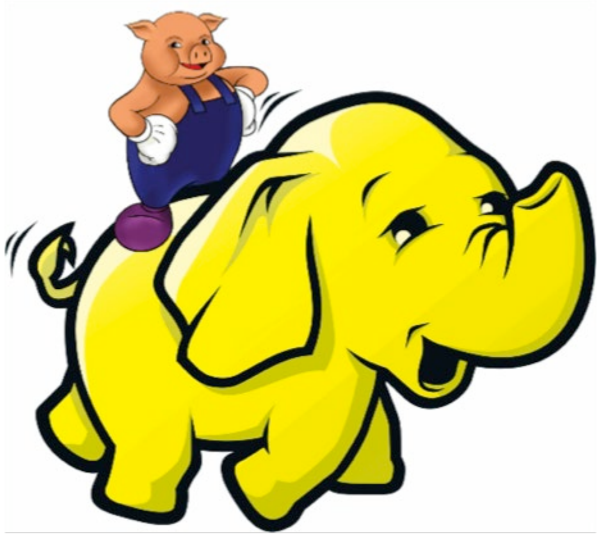
Apache Pig: Scripting on MapReduce

Pig is an abstraction layer on top of Hadoop (MapReduce/Spark)

- ➢ Code is written in Pig Latin "script" language (a data flow language)
- ➢ Facilitates reading, writing, and managing large datasets residing in distributed storage
- ➢ Pig executes queries using MapReduce (*and also using Spark*)
  - ○ Pig Latin scripts → Pig → MapReduce Jobs

Apache Hive & ApachePig

- Instead of writing Java code to implement MapReduce, one can opt between Pig Latin and Hive SQL to construct MapReduce programs
- Much fewer lines of code compared to MapReduce, which reduces the overall development and testing time



Apache Hive vs Apache Pig

- Declarative SQL-like language (HiveQL)
- Operates on the server side of any cluster
- Better for structured Data
- Easy to use, specifically for generating reports
- Data Warehousing tasks
- Facebook

- Procedural data flow language (Pig Latin)
- Runs on Client side of any cluster
- Best for semi structured data
- Better for creating data pipelines
  - allows developers to decide where to checkpoint data in the pipeline
- Incremental changes to large data sets and also better for streaming
- Yahoo

Apache Hive vs ApachePig: example

Job: Get data from sources *users* and *clicks* is to be joined and filtered, and then joined to data from a third source *geoinfo* and aggregated and finally stored into a table ValuableClicksPerDMA

```
insert into ValuableClicksPerDMA
select dma, count(*)
from geoinfo join (
select name, ipaddr
from users join clicks on
(users.name = clicks.user)
where value > 0;
) using ipaddr
group by dma;
```

```
Users     = load 'users' as (name, age, ipaddr);
Clicks    = load 'clicks' as (user, url, value);
ValuableClicks    = filter Clicks by value > 0;
UserClicks = join Users by name, ValuableClicks by
user;
Geoinfo   = load 'geoinfo' as (ipaddr, dma);
UserGeo   = join UserClicks by ipaddr, Geoinfo by
ipaddr;
ByDMA     = group UserGeo by dma;
ValuableClicksPerDMA    = foreach ByDMA generate group,
COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

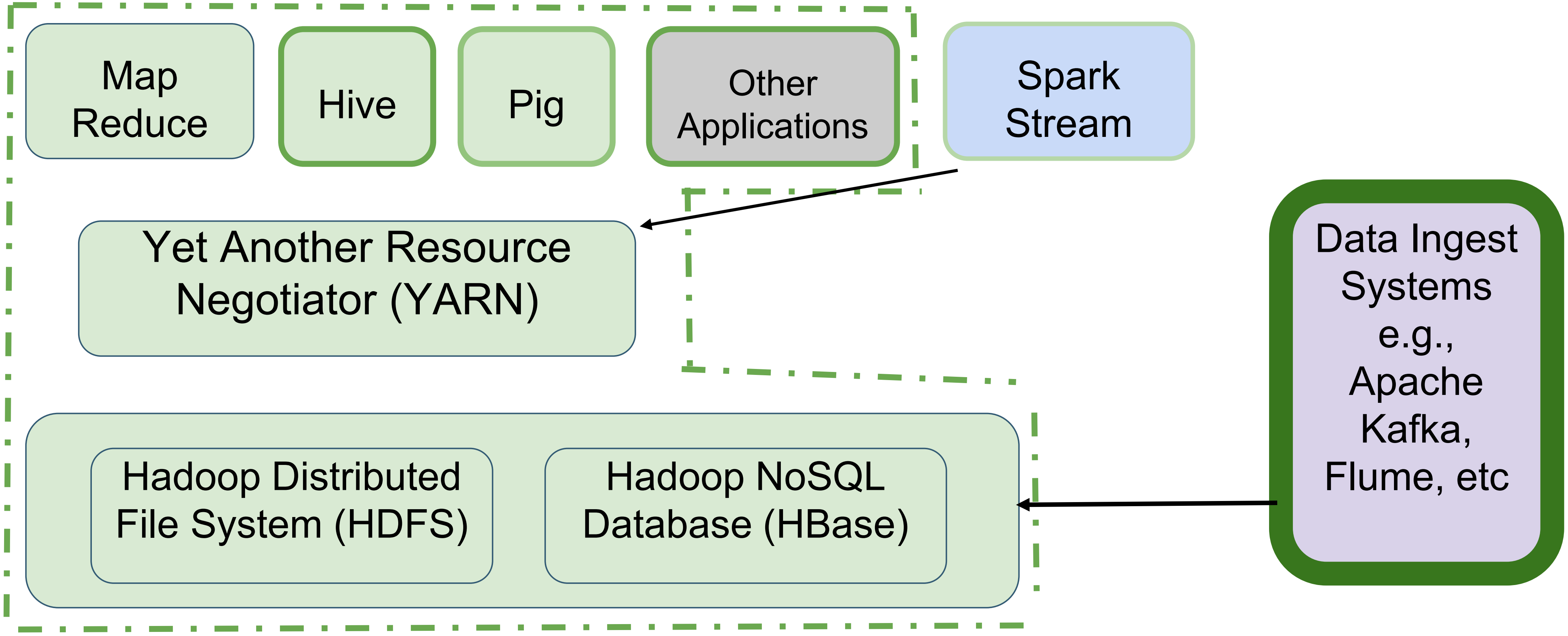
Comment: “Client side”??

When we say “runs on client side” we don’t mean “runs on the iPhone”. Here the client is any application using Hadoop.

So the “client side” is just “inside the code that consumes the Pig output”

In contrast, the “server side” lives “inside the Hive/HDFS layer”

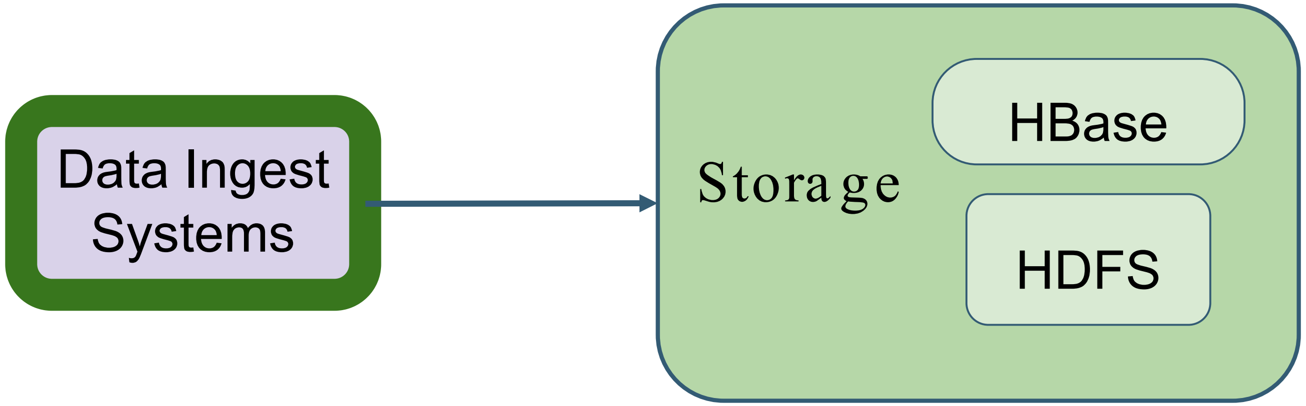
Hadoop Ecosystem: Data Ingestion



Data Ingestion Systems/Tools (1)

Hadoop typically ingests data from many sources and in many formats:

- Traditional data management systems, e.g. databases
- Logs and other machine generated data (event data)
- e.g., Apache Sqoop, Apache Fume, **Apache Kafka** (focus of this class)



Data Ingestion Systems/Tools (2)

- Apache Sqoop
  - High speed import to HDFS from Relational Database (and vice versa)
  - Supports many database systems, e.g. Mongo, MySQL, Teradata, Oracle

- Apache Flume
  - Distributed service for ingesting streaming data
  - Ideally suited for event data from multiple systems, for example, log files

Concept: “Publish-Subscribe” tool

The Apache ecosystem is pretty elaborate. It has many “tools”, and several are implemented as separate μ-services.

The μ-services run in pools: we configure the cloud to automatically add instances if the load rises, reduce if it drops

So how can individual instances belonging to a pool cooperate?

Models for cooperation

One can have explicit groups, the members know one-another, and the cooperation is scripted and deterministic as a function of a consistent view of the task pool and the membership (Zookeeper)

But this is a more complex model than needed. In some cases, we prefer more of a loose coordination, with members that take tasks from some kind of list, perform them, announce completion.

Concept: “Publish-Subscribe” tool

This is a model in which we provide middleware to glue requestors to workers, with much looser coupling.

The requests arrive as “published messages”, on “topics”

The workers monitor topics (“subscribe”) and then an idle worker can announce that it has taken on some task, and later, finished it.

Apache Kafka

- Functions like a distributed publish-subscribe messaging system (or a distributed streaming platform)
  - A high throughput, scalable messaging system
  - Distributed, reliable publish-subscribe system
  - Design as a message queue & Implementation as a distributed log service
- Originally developed by LinkedIn, now widely popular
- Features: Durability, Scalability, High Availability, High Throughput

What is Apache Kafka used for? (1)

- The original use case (@LinkedIn):
  - To track user behavior on websites.
  - Site activity (page views, searches, or other actions users might take) is published to central topics, with one topic per activity type.
- Effective for two broad classes of applications:
  - Building real-time streaming data pipelines that reliably get data between systems or applications
  - Building real-time streaming applications that transform or react to the streams of data

What is Apache Kafka used for? (2)

- Lets you publish and subscribe to streams of records, similar to a message queue or enterprise messaging system
- Lets you store streams of records in a fault-tolerant way
- Lets you process streams of records as they occur
- Lets you have both offline and online message consumption

Apache Kafka: Fundamentals

- Kafka is run as a cluster on one or more servers
- The Kafka cluster stores streams of *records* in categories called *topics*
- Each record (or message) consists of a key, a value, and a timestamp

- Point-to-Point: Messages persisted in a queue, a particular message is consumed by a maximum of one consumer only
- Publish-Subscribe: Messages are persisted in a topic, consumers can subscribe to one or more topics and consume all the messages in that topic

Apache Kafka: Components

Logical Components:

- ➢ Topic: The named destination of partition
- ➢ Partition: One Topic can have multiple partitions and it is an unit of parallelism
- ➢ Record or Message: Key/Value pair (+ Timestamp)

Physical Components:

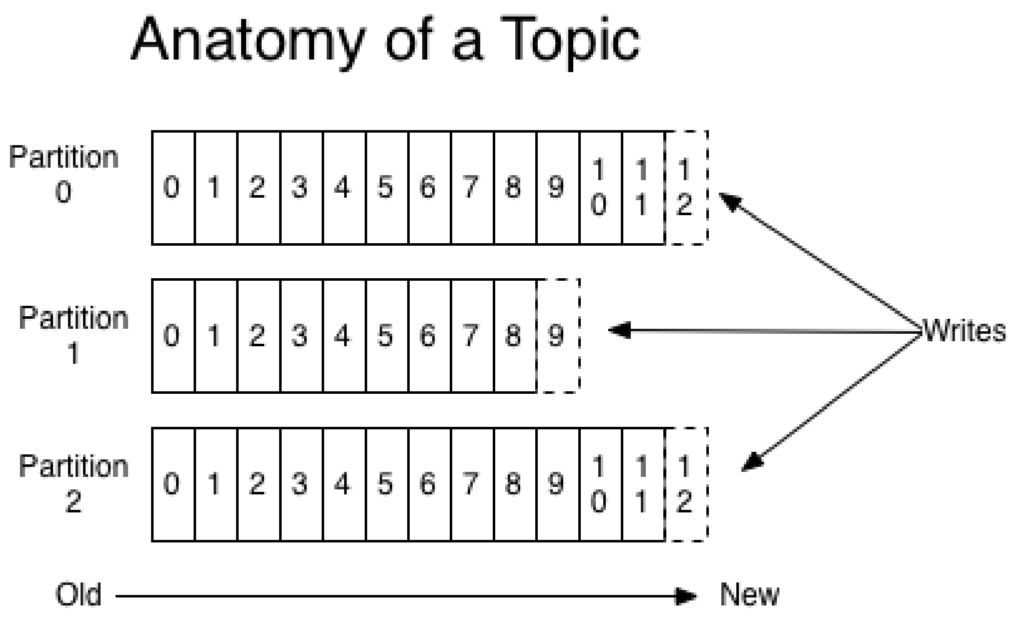
- ➢ Producer: The role to send message to broker
- ➢ Consumer: The role to receive message from broker
- ➢ Broker: One node of Kafka cluster
- ➢ ZooKeeper: Coordinator of Kafka cluster and consumer groups

Apache Kafka: Topics & Partitions (1)

- A stream of messages belonging to a particular category is called a topic (or a feed name to which records are published)
- Data is stored in topics.
- Topics in Kafka are always multi-subscriber -- a topic can have zero, one, or many consumers that subscribe to the data written to it
- Topics are split into partitions. Topics may have many partitions, so it can handle an arbitrary amount of data

Apache Kafka: Topics & Partitions (2)

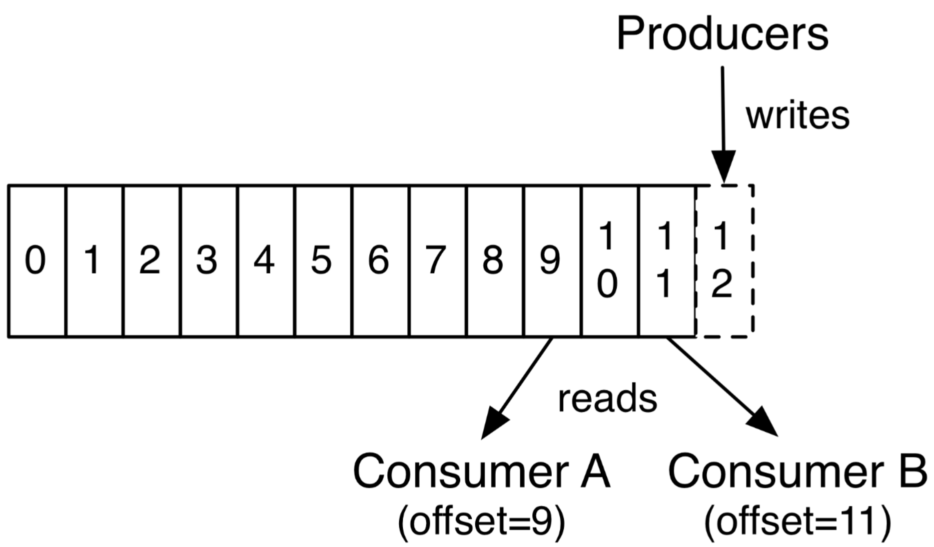
- ➢ For each topic, the Kafka cluster maintains a partitioned log that looks like this:



- ➢ Each partition is an ordered, immutable sequence of records that is continually appended to -- a structured commit log.
- ➢ *Partition offset*: The records in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each record within the partition.

Apache Kafka: Topics & Partitions (3)

- The only metadata retained on a per-consumer basis is the *offset* or position of that consumer in the log.
- This offset is controlled by the consumer -- normally a consumer will advance its offset linearly as it reads records (but it can also consume records in any order it likes)



Apache Kafka: Topics & Partitions (4)

The partitions in the log serve several purposes:

- Allow the log to scale beyond a size that will fit on a single server.
- Handles an arbitrary amount of data -- a topic may have many partitions
- Acts as the unit of parallelism

Apache Kafka: Distribution of Partitions(1)

- The partitions are distributed over the servers in the Kafka cluster and each partition is replicated for fault tolerance
- Each partition has one server acts as the “leader” (broker) and zero or more servers act as “followers” (brokers).
- The leader handles all read and write requests for the partition
- The followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader.
- Load Balancing: Each server acts as a leader for some of its partitions and a follower for others within the cluster.

Apache Kafka: Distribution of Partitions (2)

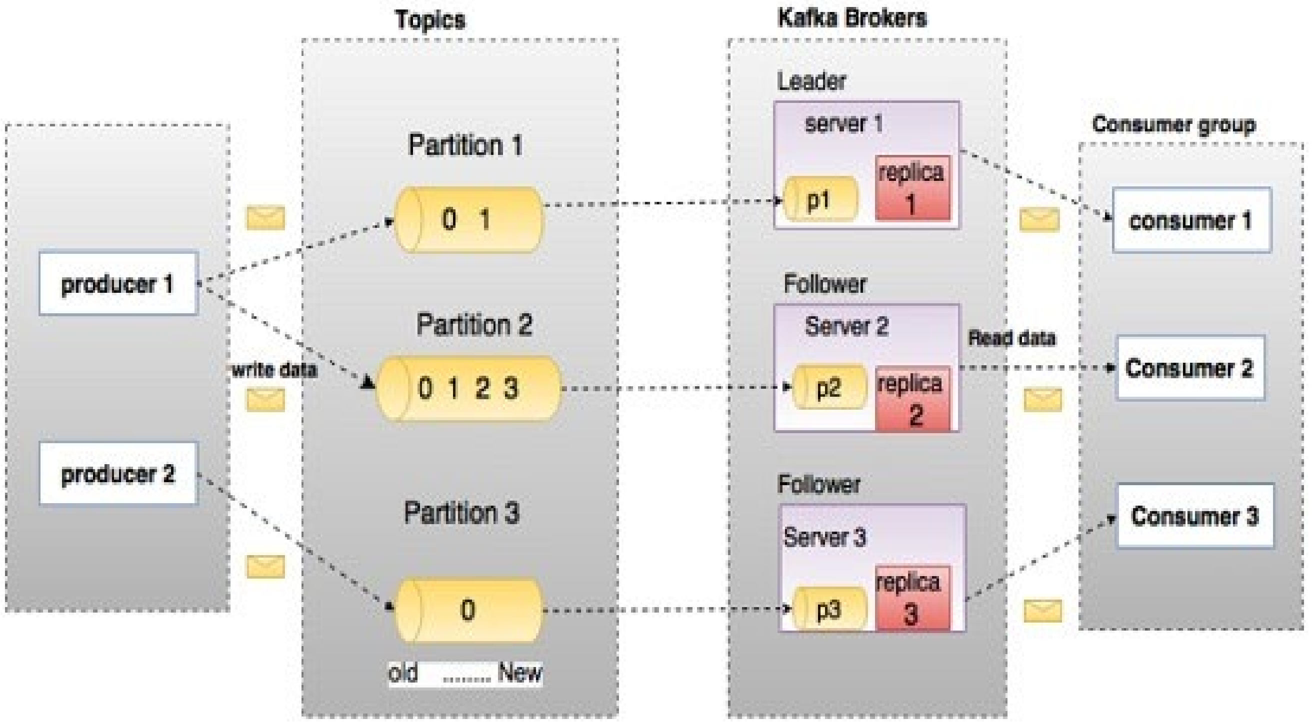
Here, a topic is configured into three partitions.

Partition 1 has two offset factors 0 and 1.

Partition 2 has four offset factors 0, 1, 2, and 3.

Partition 3 has one offset factor 0.

The id of the replica is same as the id of the server that hosts it.



Apache Kafka: Components

Logical Components:

- Topic: The named destination of partition
- Partition: One Topic can have multiple partitions and it is an unit of parallelism
- Record or Message: Key/Value pair (+ Timestamp)

Physical Components:

- Producer: The role to send message to broker
- Consumer: The role to receive message from broker
- Broker: One node of Kafka cluster
- ZooKeeper: Coordinator of Kafka cluster and consumer groups

Apache Kafka: Producers

- Producers publish data to the topics of their choice.
- The producer is responsible for choosing which record to assign to which partition within the topic.
- Record to Topic: In a round-robin fashion simply to balance load or can be done according to some semantic partition function

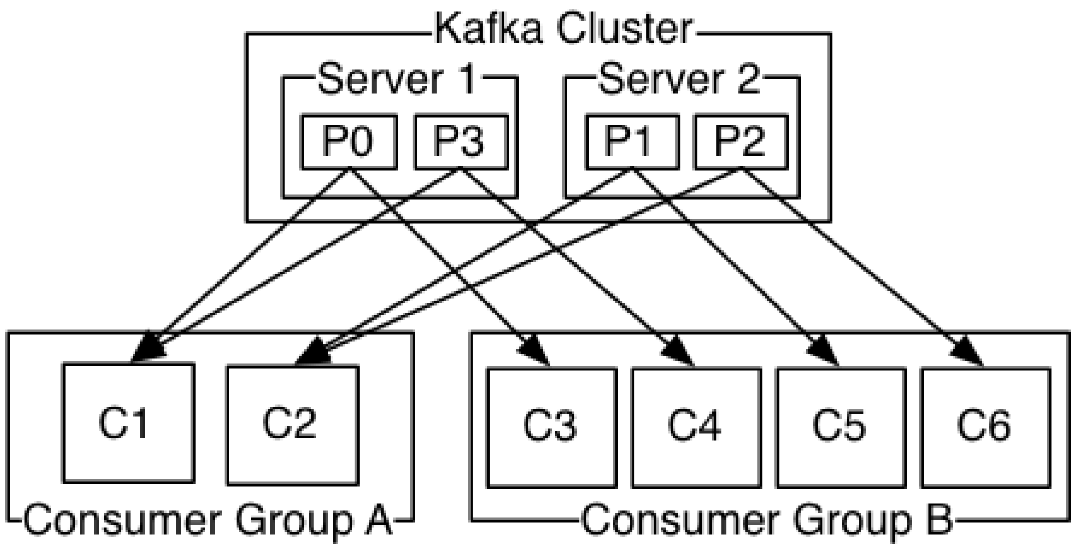
Apache Kafka: Consumers

- Consumer group: Balance consumers to partitions
- Consumers label themselves with a *consumer group* name
- Each record published to a topic is delivered to one consumer instance within each subscribing consumer group
- If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances.
- If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes.

Apache Kafka: Producers & Consumers

Example:

A two server Kafka cluster hosting four partitions (P0 to P3) with two consumer groups (A & B). Consumer group A has two consumer instances (C1 & C2) and group B has four (C3 to C6).



Apache Kafka: Design Guarantees (1)

- ➢ Records (or Messages) sent by a producer to a particular topic partition will be appended in the order they are sent.
- ➢ A consumer instance sees records in the order they are stored in the log.
- ➢ For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

Apache Kafka: Design Guarantees (2)

Message Delivery Semantics:

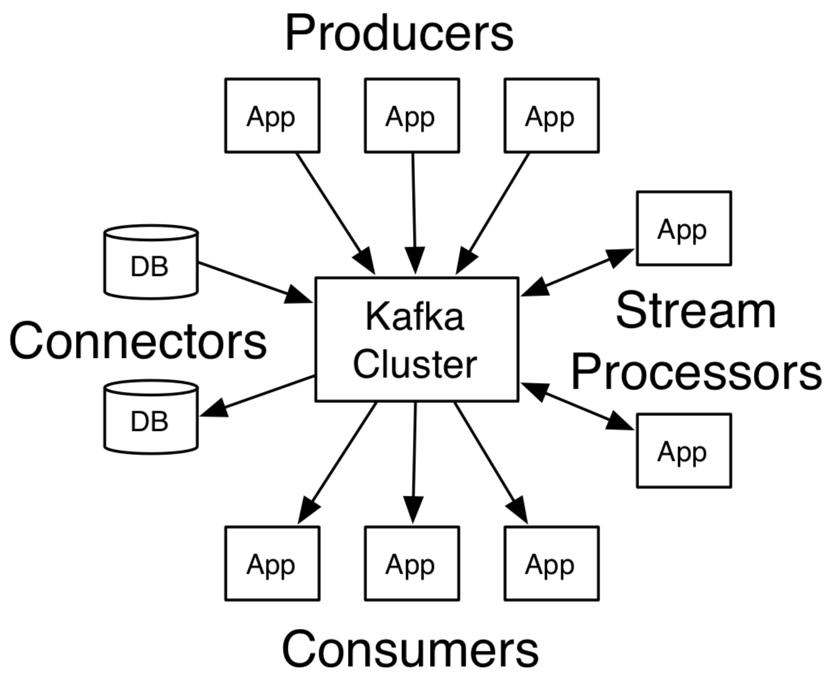
- At most once: Messages may be lost but are never redelivered.
- At least once: Messages are never lost but may be redelivered.
- Exactly once: Each message is delivered once and only once

Apache Kafka: Four Core APIs (1)

Producer API: Allows an application to publish a stream of records to one or more Kafka topics

Consumer API: Allows an application to subscribe to one or more topics and process the stream of records produced to them

Streams API: Allows an application to act as a *stream processor* -- consuming an input stream from one or more topics and producing an output stream to one or more output topics

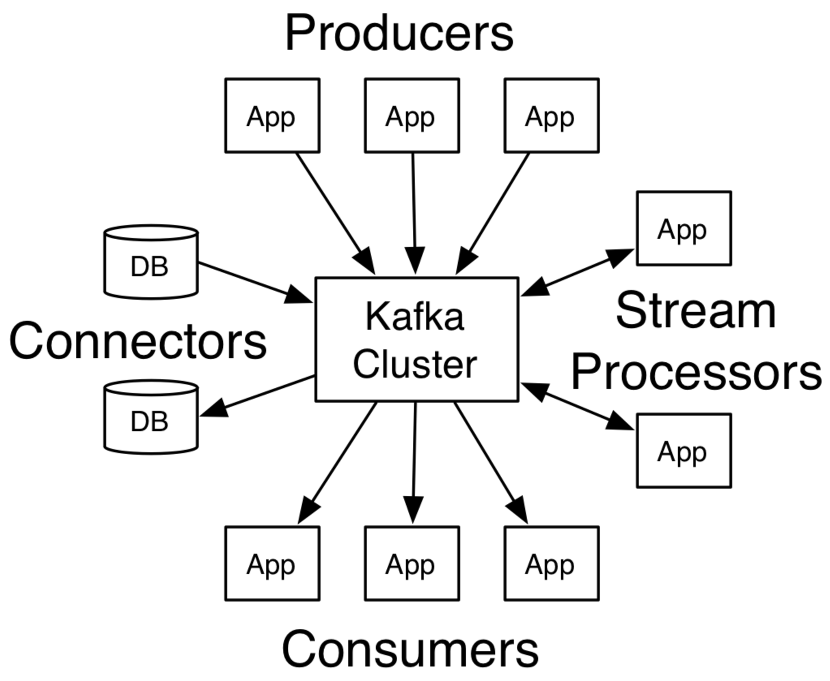


Apache Kafka: Four Core APIs (2)

Connector API:

Allows building and running producers or consumers that connect Kafka topics to existing applications or data systems.

For example, a connector to a relational database might capture every change to a table.



Kafka ← Messaging + Storage + Streaming

- Messaging:
  - The consumer group allows you to divide up processing over a collection of processes (as a queue)
  - Allows you to broadcast messages to multiple consumer groups (as with publish-subscribe).
- Storage: Data written to Kafka is written to disk and replicated for fault-tolerance.
- Streaming: Takes continuous streams of data from input topics → Processing → Produces continuous streams of data to output topics.

Tricky aspects?

Using the publish-subscribe model for fault-tolerant request-reply interactions is actually not so simple.

Someone posts a request (easy), but now a random member of the worker pool wants to grab the request: a race condition.

Tricky aspects? (cont)

… to break the tie, your server logic publishes an announcement:

Worker W has taken over task T.

Everyone trusts the *first* such announcement, ignores later ones.

Then when worker W finishes, it announces “Task T is complete”.

Tricky aspects? (cont)

Now, we wire in a failure detector service, usually Apache Zookeeper. We arrange to publish “Worker W has failed” or “Worker Z has joined the pool”.

With this, everyone will notice if W owned task T, but then crashed. T can be reactivated, or reissued.

So the problem is solved… but notice that it wasn’t transparent!

… a form of group membership!

We've previously seen group membership. Here we have another case for that model!

This example is basically using Zookeeper + Apache to create virtual-synchrony groups, and the resulting semantics are basically identical to what Derecho does for Paxos (but much slower).

There is a whole mathematical theory, which we won't cover, but it could be used to gain certainty that a solution is correct.

SUMMARY

Many big data systems are built using the standard Apache tools.

We've now seen a number of them.

The resulting systems are large and complex, often have many "moving parts", and manage themselves.