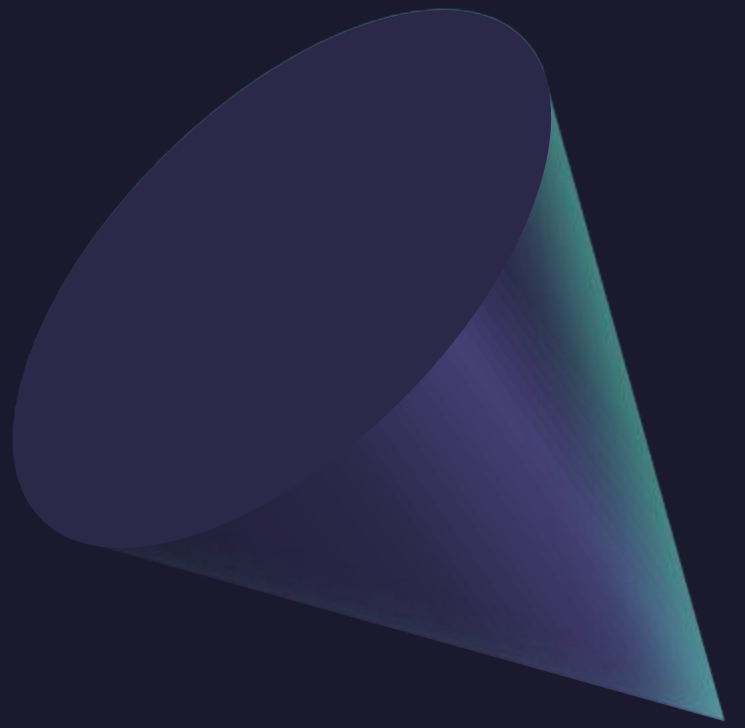
System Design - MapReduce

Software Systems Development

IIIT Hyderabad



Content

Distributed Computing

- Distributed Computing involves :
  - Clusters of machines connected over network
  - Distributed Storage
    - Network Attached Storage
    - Disks attached to clusters of machines
- *How can we make effective use of multiple machines?*
- Commodity clusters :
  - Commodity: Available off the shelf at large volumes
  - Lower Cost of Acquisition
  - Cost vs. Performance
  - Low disk bandwidth, and high network latency
  - CPUs typically comparable
- *How can we use many of such machines of modest capability?*

Distributed Computing

- Challenges of Distributed Computing involves :
  - Divide a job into multiple tasks
  - Understand dependencies between tasks: Control, Data
  - Coordinate and synchronize execution of tasks
  - Pass information between tasks
  - Avoid race conditions, deadlocks
- Parallel and distributed programming model makes it easy
  - Message Passing Interface (MPI) : It isn't a programming language but a library of functions that programmers can call from C, C++, or Fortran code to write parallel programs in distributed-memory systems.
  - MapReduce by Google.

MapReduce

"A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs."

Dean and Ghermawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI, 2004

MapReduce

- MapReduce is a distributed data-parallel programming model from Google.
- It works best with a distributed file system, called Google File System (GFS).
- Hadoop is the open source framework implementation from Apache that can execute the MapReduce programming model.
- Amazon's PaaS is yet another implementation of Map-Reduce
- It provides :
  - Clean abstraction for programmers
  - Automatic parallelization & distribution
  - Fault-tolerance
  - A batch data processing system
  - Provides status and monitoring tools
- MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion, particularly for ad hoc analysis.

MapReduce Vs RDBMS

- An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data.
- MapReduce suits applications where the data is written once, and read many times, whereas a relational database is good for datasets that are continually updated.
- MapReduce works well on unstructured or semi-structured data.

MapReduce: Data-parallel Programming Model

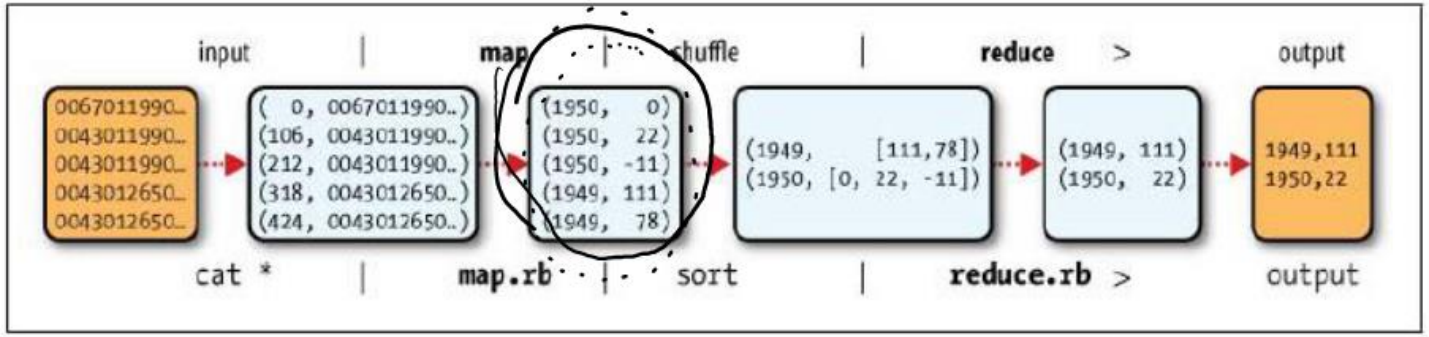
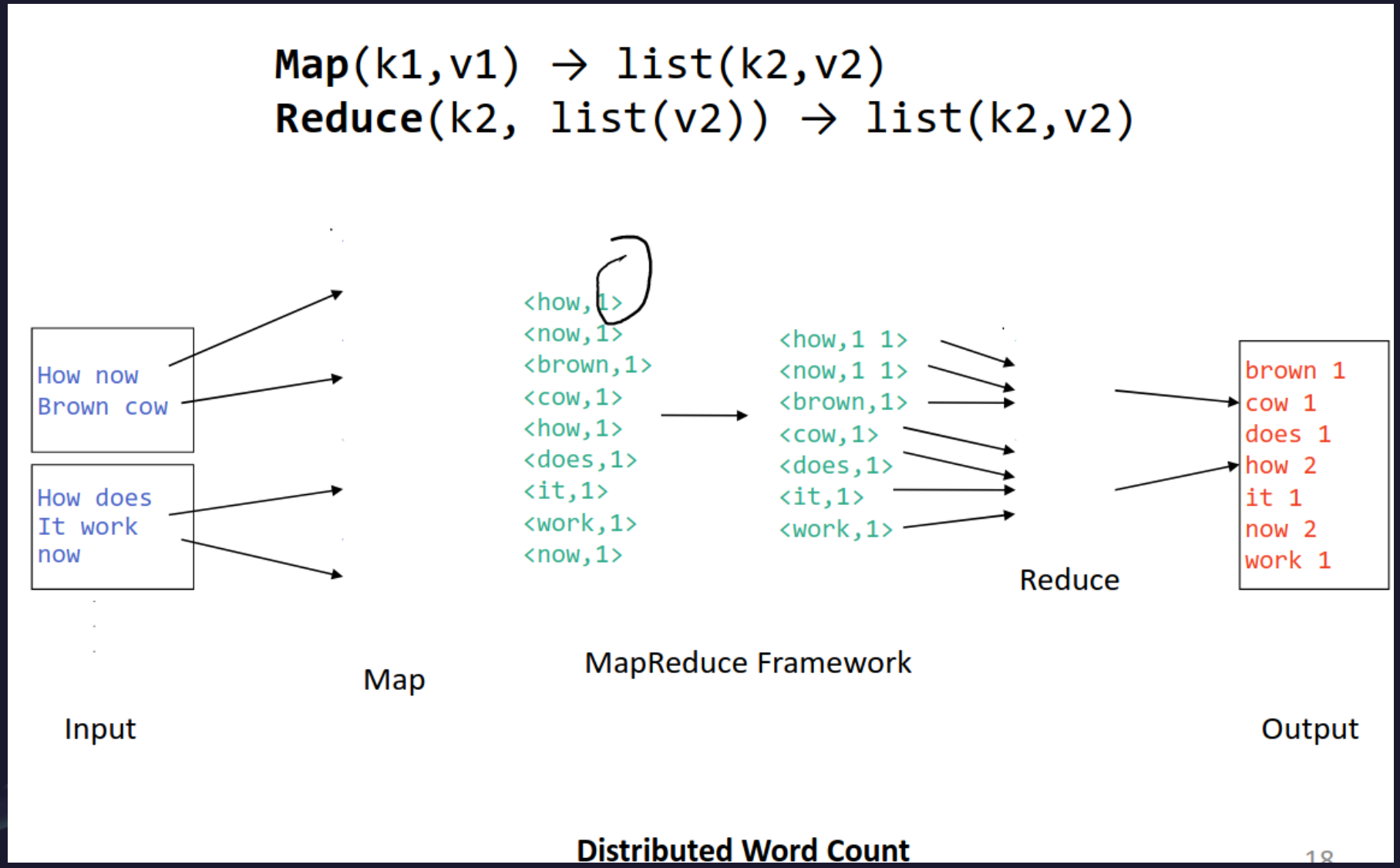


Figure 2-1. MapReduce logical data flow

- Process data using map & reduce functions
- map(k_i, v_i) → List<k_m, v_m>[]
  - ***map*** is called on every input item
  - Emits a series of intermediate key/value pairs
- All values with a given key are **grouped** together
- reduce(k_m, List<v_m>[]) → List<k_r, v_r>[]
  - ***reduce*** is called on every unique key & all its values
  - Emits a value that is added to the output

Source : Distributed Systems, Prof. Kishore Kothapalli, Monsoon 2022

MapReduce: Word Count



Source : Distributed Systems, Prof. Kishore Kothapalli, Monsoon 2022

Map

- Input records from the data source
  - lines out of files, rows of a database, etc
- Passed to map function as key-value pairs
  - Line number, line value
- map() produces zero or more intermediate values, each associated with an output key.

Map

- **Example: Upper-case Mapper**

```
map(k, v) { emit(k.toUpper(), v.toUpper()); }

("foo", "bar") → ("FOO", "BAR")
("Foo", "other") → ("FOO", "OTHER")
("key2", "data") → ("KEY2", "DATA")
```

- **Example: Filter Mapper**

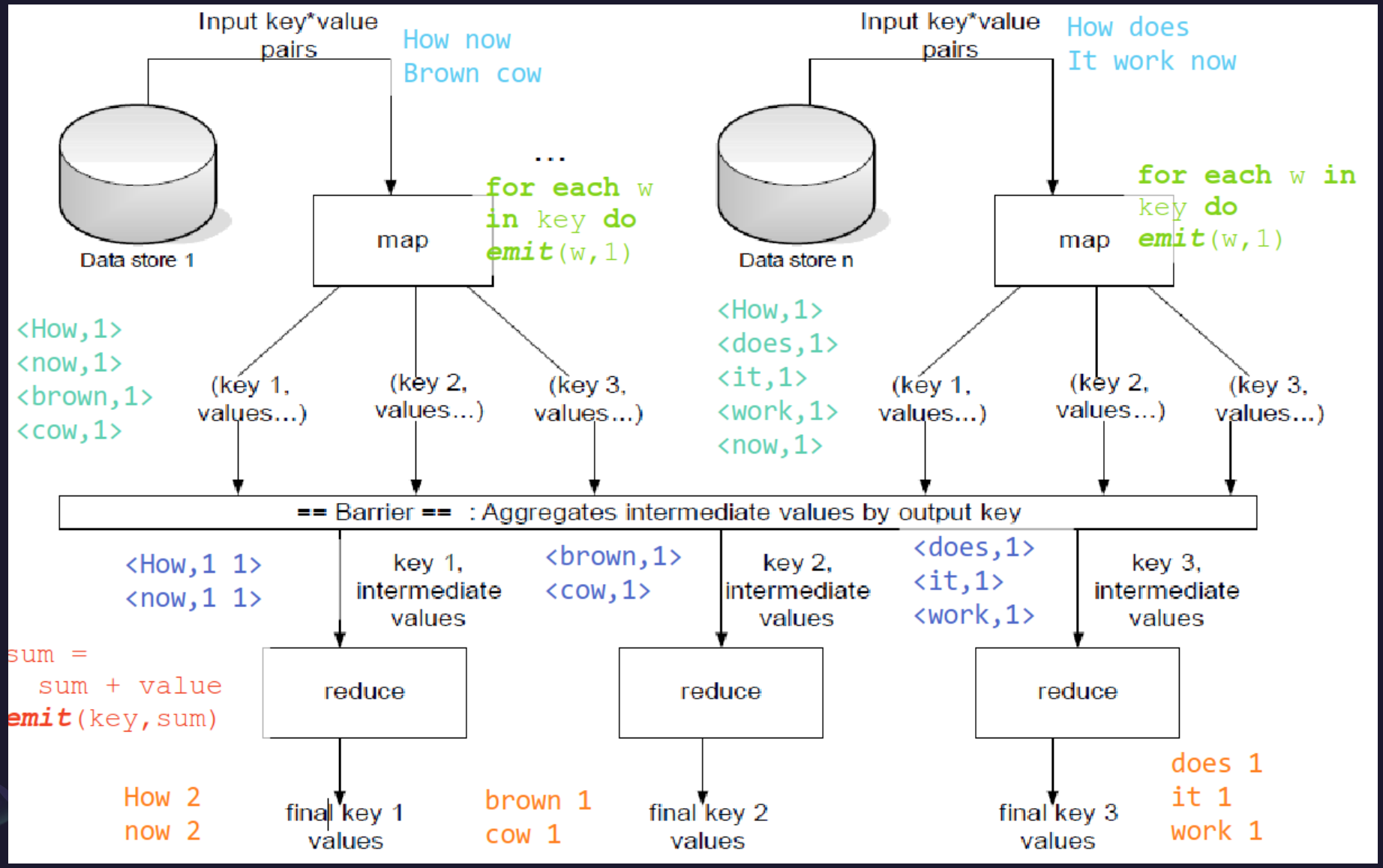
```
map(k, v) { if (isPrime(v)) then emit(k, v); }

("foo", 7) → ("foo", 7)
("test", 10) → () //nothing emitted
```

- Input records from the data source
  - lines out of files, rows of a database, etc
- Passed to map function as key-value pairs
  - Line number, line value
- map() produces zero or more intermediate values, each associated with an output key.

Reduce

- All the intermediate values from map for a given output key are combined together into a list.
- reduce() combines these intermediate values into one or more final values for that same output key ... Usually one final value per key.
- One output “file” per reducer

Word count Detailed

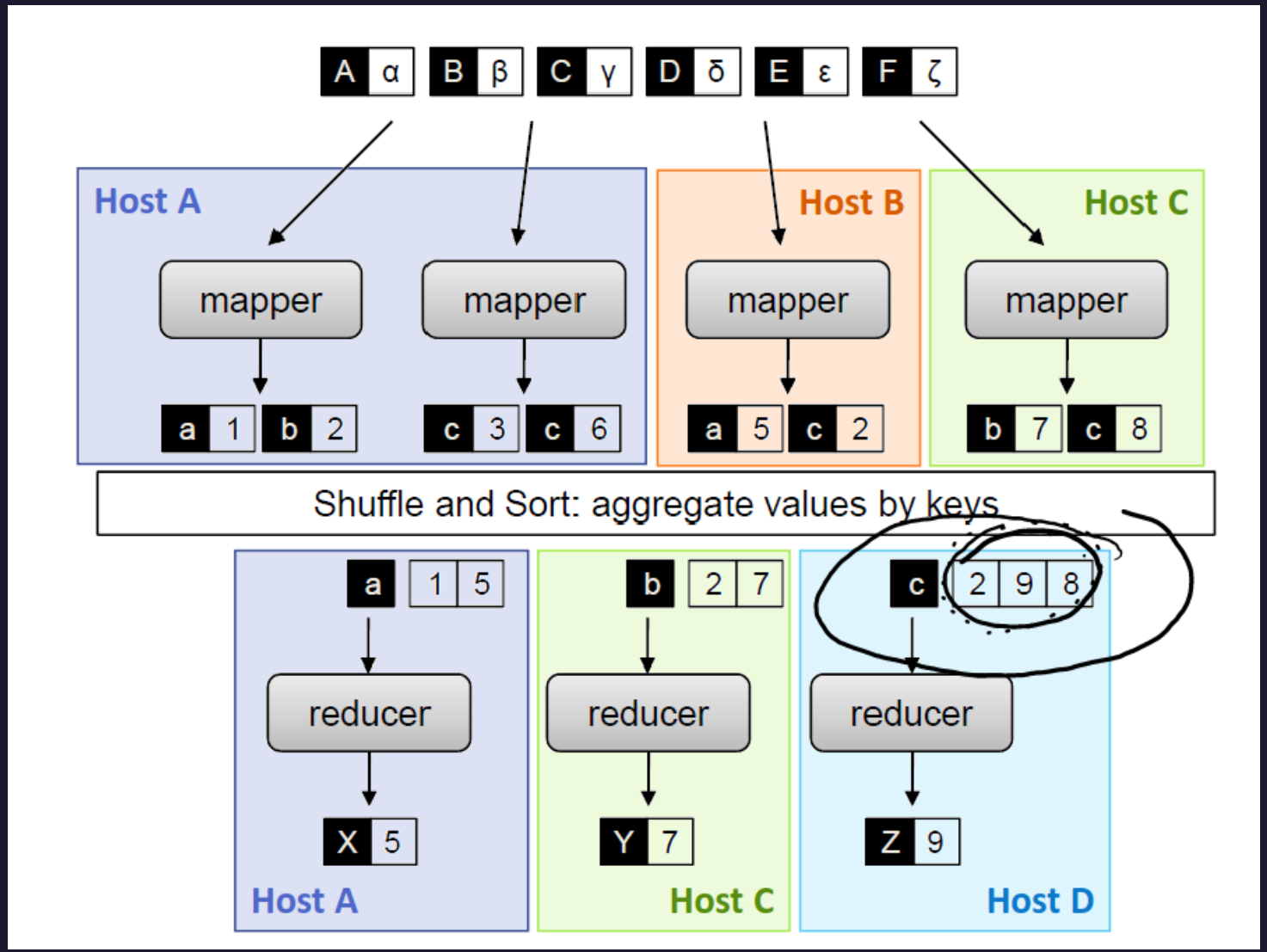


Source : Distributed Systems, Prof. Kishore Kothapalli, Monsoon 2022

Shuffle & Sort

- Shuffle does a “group by” of keys from all mappers
  - Similar to SQL groupBy operation
- Sort of local keys to Reducer task performed
  - Keys arriving at each reducer are sorted
  - No sorting guarantee of keys across reducer tasks
- No ordering guarantees of values for a key
  - Implementation dependent
- Shuffle and Sort implemented efficiently by framework

Map-Shuffle-Sort-Reduce



Source : Distributed Systems, Prof. Kishore Kothapalli, Monsoon 2022

Message Queues

- Nowadays, due to the wide adoption of microservice-based architecture, enterprise-grade applications are built as decoupled modules/services with specific functionalities.
- Queuing systems provide a sort of mechanism for these services to communicate by exchanging or transferring data in the form of buffers from one point (a source/output) to another (a destination).
- This can either be within the same application/process or different services, as the case may be.
- Message brokers are tools that allow applications to communicate via a queuing mechanism.
- These systems may act as a publisher/subscriber kind of system, where one application or process is the publisher or producer of messages/data and the other, the subscriber or consumer of same.
- **RabbitMQ** is a highly performant, open-source message broker with support for a variety of messaging protocols.

RabbitMQ

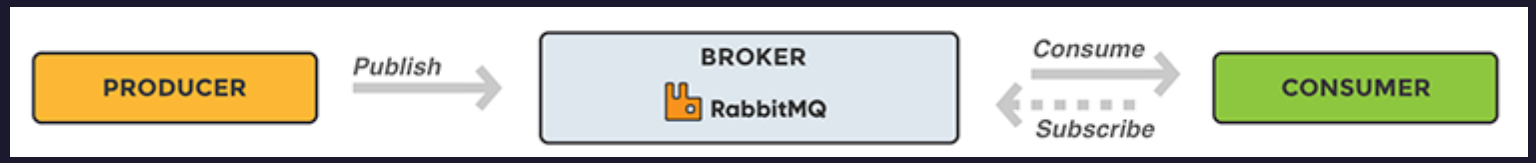
- **RabbitMQ** is one of the most popular open-source message brokers. With RabbitMQ, we can define queues, push messages to these queues, and subsequently consume messages from them. Message brokers are key here because they offer a point of contact or interface between the producing and consuming application or process.
- **Producer:** A producer sends or pushes messages to a queue based on a queue name
- **Queue:** A queue is a medium via which we can transfer and store messages or buffers
- **Consumer:** A consumer subscribes, receives, or consumes messages from the broker, and then processes or uses them in another process or application
- **Exchange:** An exchange is an entry point to the broker as it receives messages from a publisher and routes them to the appropriate queue
- **Broker:** A message broker basically offers a storage mechanism for data produced from one application. This data is usually meant to be consumed by another application that connects to the broker with the given parameters or connection strings
- **Channel:** Channels offer a sort of lightweight connection to a broker via a singular and shared TCP connection. This is due to the fact that creating multiple open connections to a broker is an expensive operation
- **Virtual host (Vhost):** Virtual hosts make it possible for a single broker to host a couple of isolated environments

RabbitMQ Example

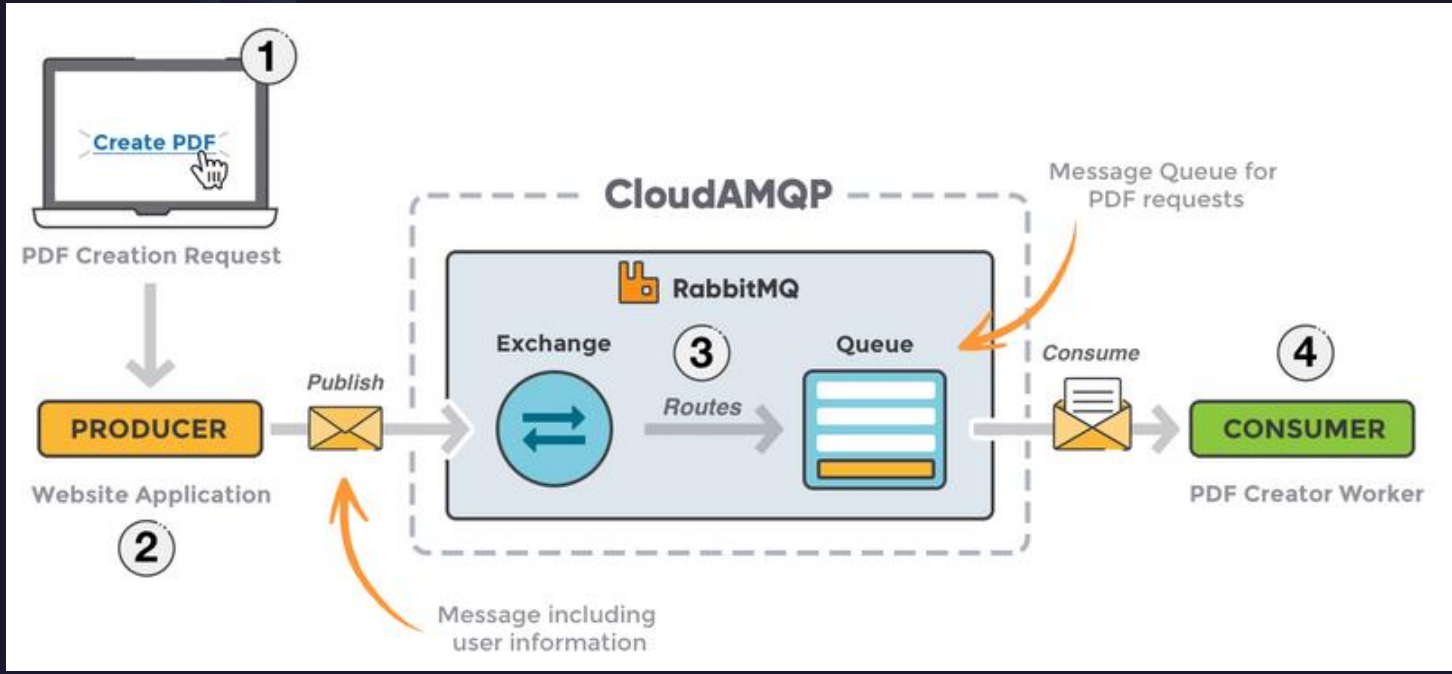
- Let's take a scenario where a web application allows users to upload information to a website. The site will handle this information, generate a PDF, and email it back to the user. Handling the information, generating the PDF, and sending the email will, in this example case, take several seconds. That is one of the reasons why a message queue will be used to perform the task.

- When the user has entered user information into the web interface, the web application will create a "PDF processing" message that includes all of the important information the user needs into a message and place it onto a queue defined in RabbitMQ.

- The basic architecture of a message queue is simple -
    - there are client applications called producers that create messages and deliver them to the broker (the message queue).
    - Other applications, called consumers, connect to the queue and subscribe to the messages to be processed. Software may act as a producer, or consumer, or both a consumer and a producer of messages.
    - Messages placed onto the queue are stored until the consumer retrieves them.



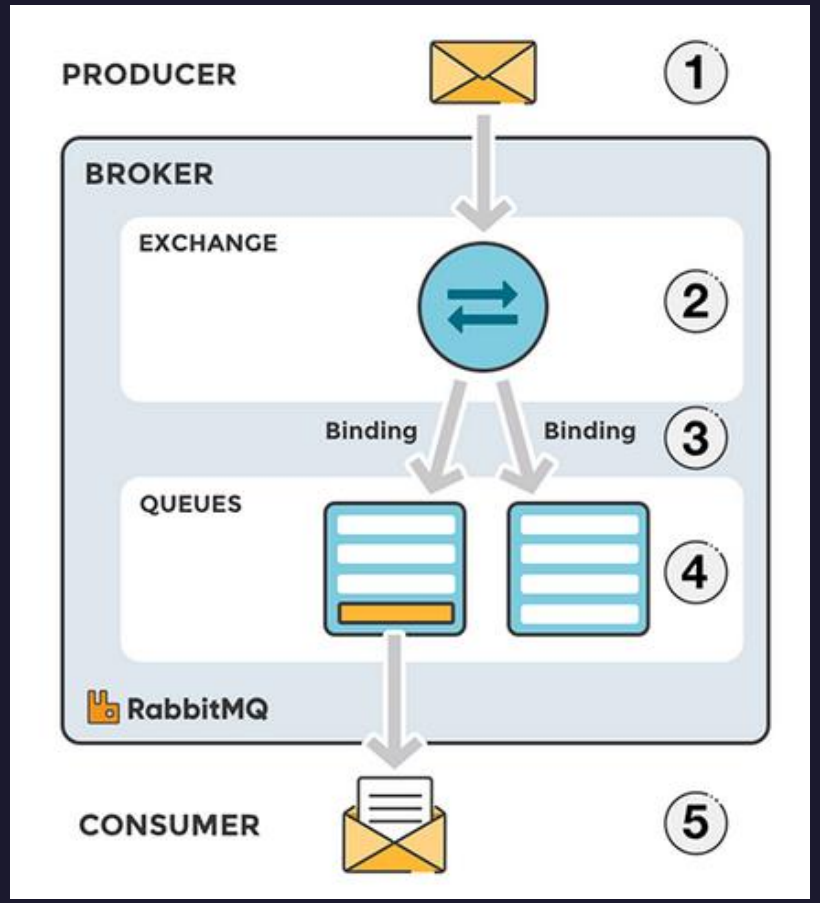
RabbitMQ Example



- The user sends a PDF creation request to the web application.
- The web application (the producer) sends a message to RabbitMQ that includes data from the request such as name and email.
- An exchange accepts the messages from the producer and routes them to correct message queues for PDF creation.
- The PDF processing worker (the consumer) receives the task message and starts processing the PDF.

Exchanges

- Messages are not published directly to a queue; instead, the producer sends messages to an exchange. An exchange is responsible for routing the messages to different queues with the help of bindings and routing keys. A binding is a link between a queue and an exchange.
- Message flow in RabbitMQ
  - The producer publishes a message to an exchange. When creating an exchange, the type must be specified. This topic will be covered later on.
  - The exchange receives the message and is now responsible for routing the message. The exchange takes different message attributes into account, such as the routing key, depending on the exchange type.
  - Bindings must be created from the exchange to queues. In this case, there are two bindings to two different queues from the exchange. The exchange routes the message into the queues depending on message attributes.
  - The messages stay in the queue until they are handled by a consumer
  - The consumer handles the message.



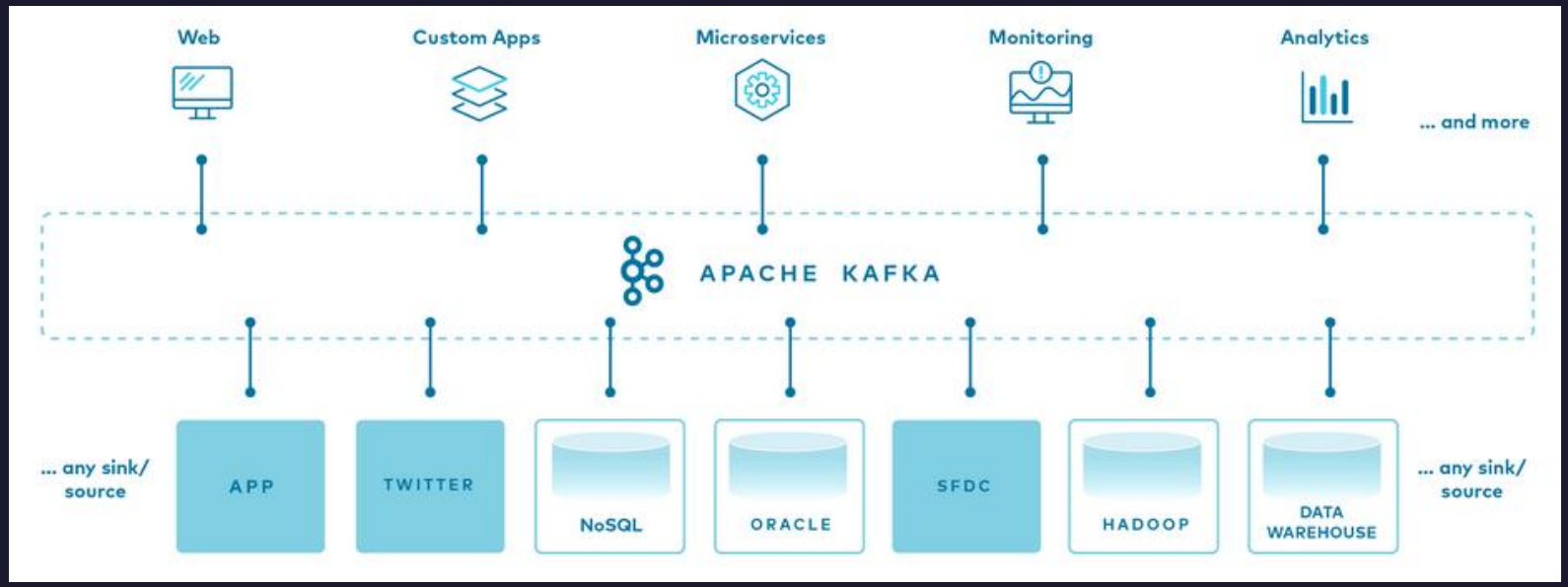
Event Streaming

- **Event streaming** is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed.

- Event Streaming Use cases :
  - To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
  - To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
  - To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
  - To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.
  - To serve as the foundation for data platforms, event-driven architectures, and microservices.
  - To connect, store, and make available data produced by different divisions of a company.

Kafka

- **Apache Kafka** is an event streaming platform used to collect, process, store, and integrate data at scale. It has numerous use cases including distributed streaming, stream processing, data integration, and pub/sub messaging.
- Benefits :
  - Data Integration
  - Metrics and Monitoring
  - Log Aggregation
  - Stream Processing
  - Publish-Subscribe Messaging



Kafka Semantics

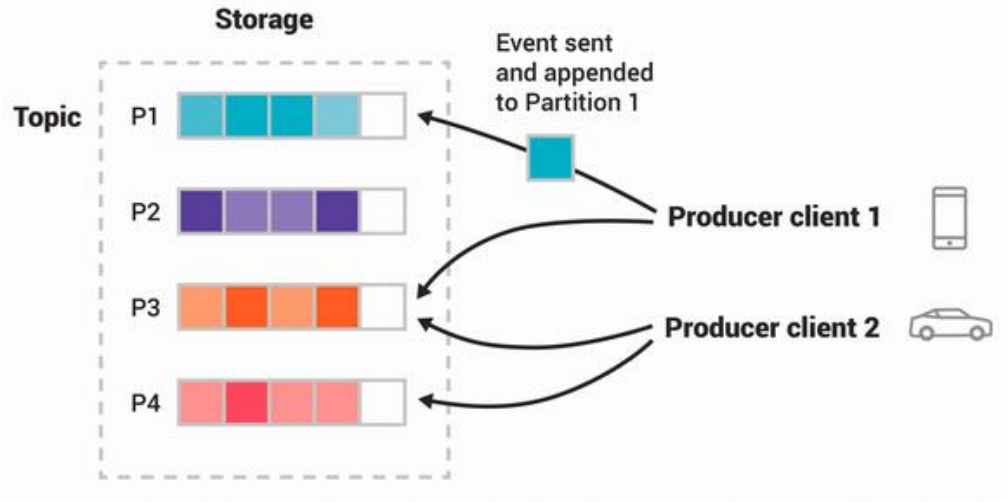
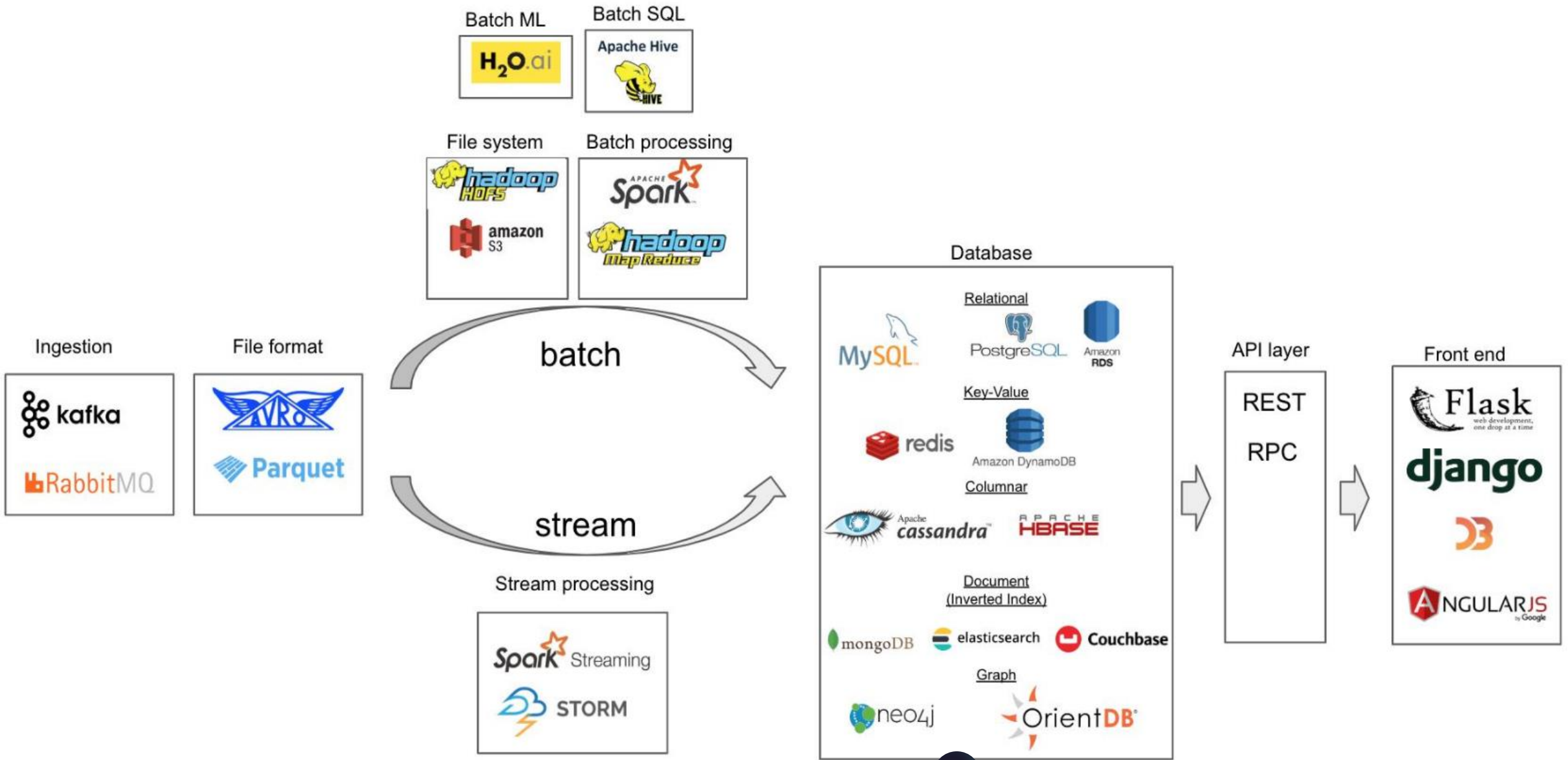


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

- An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. For ex. Event key: "Alice" , Event value: "Made a payment of $200 to Bob" , Event timestamp: "Jun. 25, 2020 at 2:06 p.m."
- **Producers** are those client applications that publish (write) events to Kafka.
- **consumers** are those that subscribe to (read and process) these events.
- Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be "payments".
- Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers.


Batch ML
H2O.ai
Batch SQL
Apache Hive
HIVE
File system
hadoop HDFS
amazon S3
Batch processing
APACHE Spark
hadoop MapReduce
Ingestion
kafka
RabbitMQ
File format
AVRO
Parquet
batch
stream
Stream processing
Spark Streaming
STORM
Database
Relational
MySQL
PostgreSQL
Amazon RDS
Key-Value
redis
Amazon DynamoDB
Columnar
Apache cassandra
APACHE HBASE
Document (Inverted Index)
mongoDB
elasticsearch
Couchbase
Graph
neo4j
OrientDB
API layer
REST
RPC
Front end
Flask
web development, one drop at a time
django
D3
ANGULARJS
by Google

References

- https://hpc.nmsu.edu/discovery/mpi/introduction/
- https://www.ibm.com/topics/mapreduce
- https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html
- https://blog.logrocket.com/understanding-message-queuing-systems-using-rabbitmq/
- https://kafka.apache.org/intro
- https://developer.confluent.io/what-is-apache-kafka/

Thank you