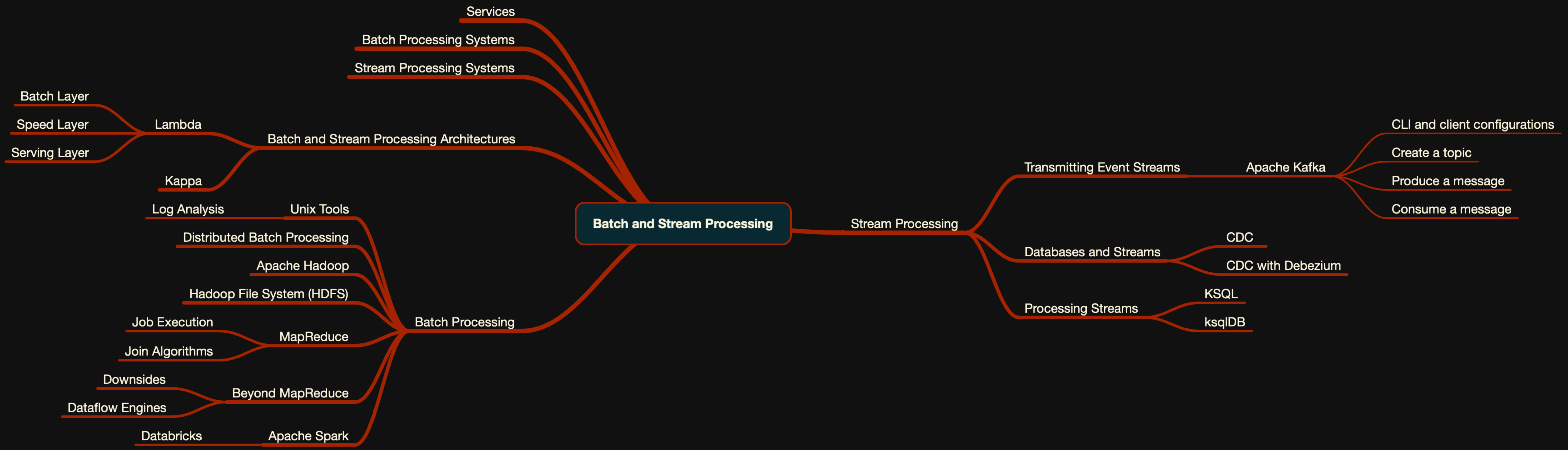
Batch & Stream Processing

by Fatih Nayebi, Ph.D.

Master of Management Analytics, Desautels Faculty of Management, McGill

Agenda



Batch and Stream Processing
Services
Batch Processing Systems
Stream Processing Systems
Batch and Stream Processing Architectures
Lambda
Batch Layer
Speed Layer
Serving Layer
Kappa
Batch Processing
Unix Tools
Log Analysis
Distributed Batch Processing
Apache Hadoop
Hadoop File System (HDFS)
MapReduce
Job Execution
Join Algorithms
Beyond MapReduce
Downsides
Dataflow Engines
Apache Spark
Databricks
Stream Processing
Transmitting Event Streams
Apache Kafka
CLI and client configurations
Create a topic
Produce a message
Consume a message
Databases and Streams
CDC
CDC with Debezium
Processing Streams
KSQL
ksqlDB

Agenda

1. Introduction

- *Machine Learning (ML)* and *Deep Learning (DL)* systems need to deal with data (Big or small)
- ML and DL algorithms / models are required to run with different frequencies
    - Real-time, every n minutes, every day, every week,...
- There are different approaches and architectures to achieve the best results

1.1. How to run algorithms / models?

- **Transformation**
  - Batch / Offline Transformation
  - Near-Real-Time Transformation
- **Training**
  - Offline Training
  - Near-Real-Time Training
- **Prediction**
  - Offline Prediction
  - Near-Real-Time Prediction
  - Real-Time Prediction

1.2. Approaches

- HTTP/REST based APIs
- Request / Response approach
- Other approaches
  - Services (Online Systems)
  - Many algorithms can be categorized by whether they operate on all data points at once or on one or a few times in mini-batches.
    - Batch Processing Systems (Offline Systems)
    - Stream Processing Systems (Near-Real-Time Systems)

2. Services

1. Online Systems
2. Requirements

2.1. Online Systems

- A stand-alone unit of a larger system that provides some particular function
- A service waits for a request or instruction from a client to arrive.
- When one is received, the service tries to handle it as quickly as possible and sends a response back.
- Response time is usually the primary measure of performance of a service, and availability is often very important (if the client can't reach the service, the user will probably get an error message).

2.2. Requirements

- Services are black boxes, in that their logic is hidden from their users.
- Services are autonomous, in that they are responsible for the function they serve.
- Services are stateless and return either their value or an error. You should be able to restart a service without it affecting the system.
- Services should be reusable. This is especially useful with Machine Learning systems, where you might have a random forest fitting service that can serve many systems.
- 12-Factor Apps

3. Batch Processing Systems - Offline Systems

- A batch processing system takes a large amount of input data, runs a job to process it, and produces some output data.
- Jobs often take a while (from a few minutes to several days), so there normally isn't a user waiting for the job to finish.
- Instead, batch jobs are often scheduled to run periodically (for example, once a day).
- The primary performance measure of a batch job is usually throughput (the time it takes to crunch through an input dataset of a certain size).

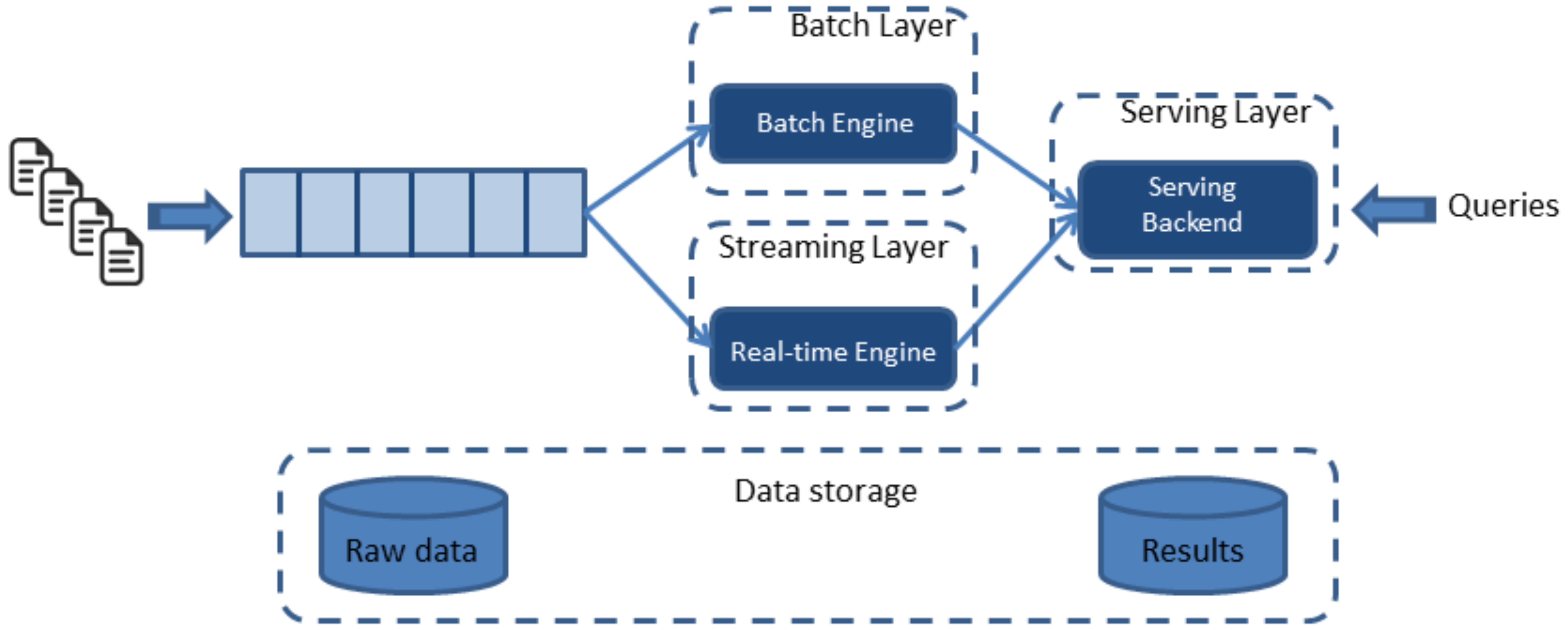
4. Stream Processing Systems - Near-Real-Time Systems

- Stream processing is somewhere between online and offline/batch processing (so it is sometimes called near-real-time or near-line processing).
- Like a batch processing system, a stream processor consumes inputs and produces outputs (rather than responding to requests).
- However, a stream job operates on events shortly after they happen, whereas a batch job operates on a fixed set of input data.
- This difference allows stream processing systems to have lower latency than the equivalent batch systems.

5. Batch and Stream processing architectures

1. Lambda (λ)
2. Kappa (κ)

5.1. Lambda Architecture



5.1. Lambda (λ) Architecture

- λ-architecture is a way of processing massive quantities of data (i.e. "Big Data") that provides access to batch processing and stream processing methods with a hybrid approach.
- λ-architecture is used to solve the problem of computing arbitrary functions.
- The λ-architecture itself is composed of 3 layers:
  - Batch layer
  - Speed (Stream) layer
  - Serving layer

5.1.1. Lambda (λ) Architecture - Batch Layer

- New data comes continuously, as a feed to the data system. It gets fed to the batch layer and the speed layer simultaneously.
- Here we can find lots of ETL and a traditional data warehouse.
- This layer is built using a predefined schedule, usually once or twice a day.
- The batch layer has two very important functions:
  - To manage the master dataset
  - To pre-compute the batch views.

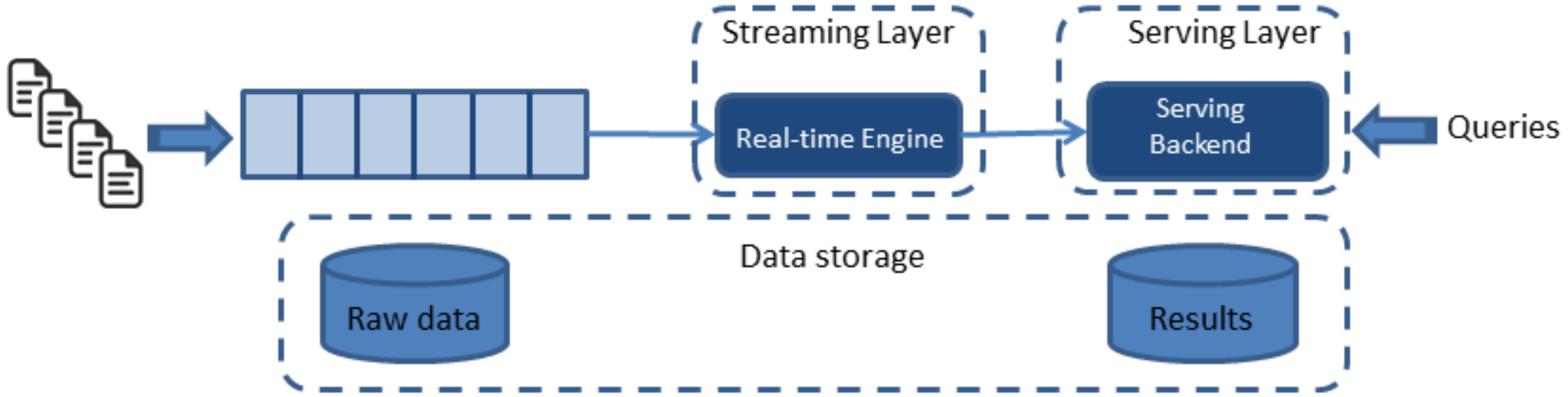
5.1.2. Lambda (λ) Architecture - Speed Layer

- This layer handles the data that are not already delivered in the batch view due to the latency of the batch layer.
- In addition, it only deals with recent data in order to provide a complete view of the data to the user by creating real-time views.

5.1.3. Lambda (λ) Architecture - Serving layer

- The outputs from the batch layer in the form of batch views and those coming from the speed layer in the form of near real-time views get forwarded to the serving.
- This layer indexes the batch views so that they can be queried in low-latency on an ad-hoc basis.

5.2. Kappa Architecture



5.2. Kappa (κ) Architecture

- κ-Architecture is a simplification of λ-Architecture.
- A κ-Architecture system is like a λ-Architecture system with the batch processing system removed.
- To replace batch processing, data is simply fed through the streaming system quickly.

6. Batch Processing

- Batch processing is an important building block in our quest to build reliable, scalable, and maintainable applications.
    1. Unix Tools
    2. Distributed Batch Processing
    3. Apache Hadoop
    4. Hadoop File System (HDFS)
    5. MapReduce
    6. Beyond MapReduce
    7. Apache Spark

6.1. Unix Tools

Suppose we have a web server that appends a line to a log file every time it serves a request. For instance, using the nginx default access log format, one line of the log might look like this:

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000]
"GET /css/typography.css HTTP/1.1"
200 3377 "http://martin.kleppmann.com/" "Mozilla/5.0
(Macintosh; Intel Mac OS X10_9_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115
Safari/537.36"
```

6.1. Unix Tools

In order to interpret it, we need to look at the definition of the log format, which is as follows:

```
$remote_addr - $remote_user [$time_local] "$request"
$status $body_bytes_sent "$http_referer" "$http_user_agent"
```

So, this one line of the log indicates that on February 27, 2015, at 17:55:11 UTC, the server received a request for the file /css/typography.css from the client IP address 216.58.210.78. The user was not authenticated, so $remote_user is set to a hyphen (-). The response status was 200 (i.e., the request was successful), and the response was 3,377 bytes in size. The web browser was Chrome 40, and it loaded the file because it was referenced in the page at the URL http://martin.kleppmann.com/.

6.1.1. Unix Tools - Log Analysis

For example, say you want to find the five most popular pages on the website. We can do this in a Unix shell as follows:

```
cat /var/log/nginx/access.log |
awk '{print $7}'              |
sort                          |
uniq -c                       |
sort -r -n                    |
head -n 5
```

6.1.1. Unix Tools - Log Analysis

- `cat /var/log/nginx/access.log` reads the log file.
- `awk '{print $7}'` splits each line into fields by whitespace, and output only the seventh such field from each line, which happens to be the requested URL. In our example line, this request URL is /css/typography.css.
- `sort` alphabetically sorts the list of requested URLs. If some URL has been requested n times, then after sorting, the file contains the same URL repeated n times in a row.
- `uniq -c` filters out repeated lines in its input by checking whether two adjacent lines are the same. The -c option tells it to also output a counter: for every distinct URL, it reports how many times that URL appeared in the input.
- `sort -r -n`sorts by the number (-n) at the start of each line, which is the number of times the URL was requested. It then returns the results in reverse (-r) order, i.e. with the largest number first.
- `head -n 5` outputs just the first five lines (-n 5) of input, and discards the rest.

6.1.1. Unix Tools - Log Analysis

The output of that series of commands looks something like this:

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol[…]”
```

6.2. Distributed Batch Processing

The two main problems that distributed batch processing frameworks need to solve are:

- **Partitioning**
  In MapReduce, mappers are partitioned according to input file blocks. The output of mappers is repartitioned, sorted, and merged into a configurable number of reducer partitions. The purpose of this process is to bring all the related data—e.g., all the records with the same key—together in the same place. Post-MapReduce data-flow engines try to avoid sorting unless it is required, but they otherwise take a broadly similar approach to partitioning.
- **Fault tolerance**
  MapReduce frequently writes to disk, which makes it easy to recover from an individual failed task without restarting the entire job but slows down execution in the failure-free case. Data-flow engines perform less materialization of intermediate state and keep more in memory, which means that they need to recompute more data if a node fails. Deterministic operators reduce the amount of data that needs to be recomputed.

6.3. Apache Hadoop

- Hadoop is an open-source data processing tool that was developed by the Apache Software Foundation.
- Hadoop is somewhat like a distributed version of Unix
- Hadoop is comprised of the following two components:
  - **A distributed processing framework - MapReduce**: Hadoop uses Hadoop MapReduce as its distributed processing framework where processing tasks are distributed across clusters of nodes so that large data volumes can be processed very quickly across the system as a whole.
    - MapReduce is a quirky implementation of a Unix process (which happens to always run the sort utility between the map phase and the reduce phase).
  - **A distributed file system - HDFS**: Hadoop uses the Hadoop Distributed File System (HDFS) as its distributed file system.

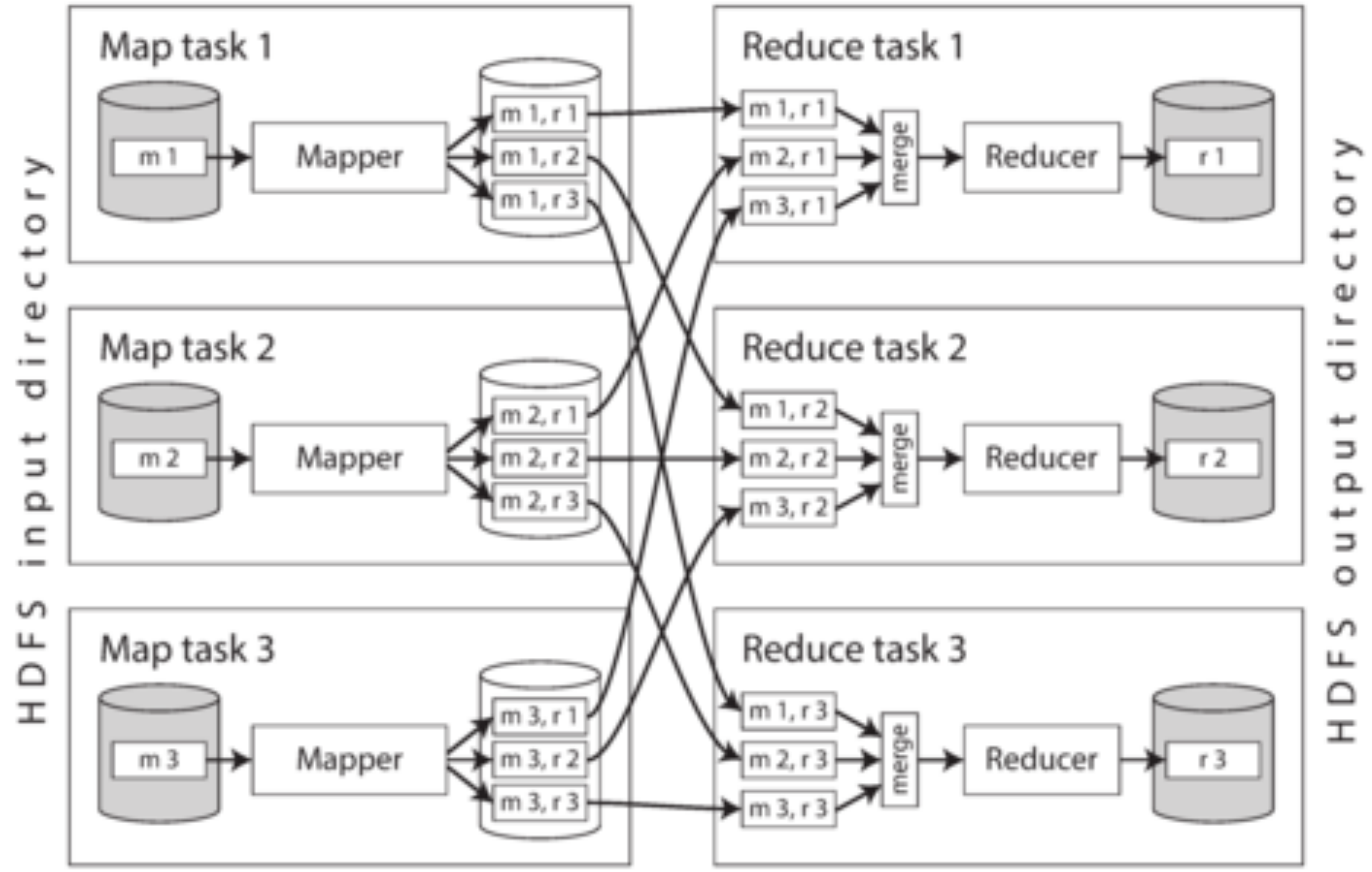
6.4. Hadoop File System (HDFS)

- While Unix tools use `stdin` and `stdout` as input and output, MapReduce jobs read and write files on a distributed filesystem.
- In Hadoop's implementation of MapReduce, that filesystem is called HDFS (Hadoop Distributed File System), an open source reimplementation of the Google File System (GFS).
- Various other distributed filesystems besides HDFS exist, such as GlusterFS and the Quantcast File System (QFS).
- Object storage services such as Amazon S3, Azure Blob Storage, and OpenStack Swift are similar in many ways.
- HDFS is based on the shared-nothing principle, in contrast to the shared-disk approach of Network Attached Storage (NAS) and Storage Area Network (SAN) architectures.
  - Shared-disk storage is implemented by a centralized storage appliance, often using custom hardware and special network infrastructure such as Fibre Channel.
  - On the other hand, the shared-nothing approach requires no special hardware, only computers connected by a conventional datacenter network.

6.4. Hadoop File System (HDFS)

- HDFS consists of
  - A daemon process running on each machine, exposing a network service that allows other nodes to access files stored on that machine (assuming that every general-purpose machine in a datacenter has some disks attached to it).
  - A central server called the NameNode keeps track of which file blocks are stored on which machine. Thus, HDFS conceptually creates one big filesystem that can use the space on the disks of all machines running the daemon.
- In order to tolerate machine and disk failures, file blocks are replicated on multiple machines. Replication may mean simply several copies of the same data on multiple machines, or an erasure coding scheme such as Reed–Solomon codes, which allows lost data to be recovered with lower storage overhead than full replication.
- The techniques are similar to RAID, which provides redundancy across several disks attached to the same machine; the difference is that in a distributed filesystem, file access and replication are done over a conventional datacenter network without special hardware.

6.5. Map Reduce



6.5. Map Reduce

- MapReduce is a batch processing algorithm published in 2004 and was (perhaps over-enthusiastically) called *“the algorithm that makes Google so massively scalable”*.
- It was subsequently implemented in various open source data systems, including Hadoop, CouchDB, and MongoDB.
- MapReduce is a fairly low-level programming model compared to the parallel processing systems that were developed for data warehouses many years previously but it was a major step forward in terms of the scale of processing that could be achieved on commodity hardware.

6.5. Map Reduce

- Although the importance of MapReduce is now declining, it is still worth understanding, because it provides a clear picture of why and how batch processing is useful.
- In fact, batch processing is a very old form of computing. Long before programmable digital computers were invented, punch card tabulating machines—such as the Hollerith machines used in the 1890 US Census —implemented a semi-mechanized form of batch processing to compute aggregate statistics from large inputs.
- MapReduce bears an uncanny resemblance to the electromechanical IBM card-sorting machines that were widely used for business data processing in the 1940s and 1950s. - As usual, history has a tendency of repeating itself.

6.5.1. Map Reduce - Job Execution

- MapReduce is a programming framework with which you can write code to process large datasets in a distributed filesystem like HDFS.
- The easiest way of understanding it is by referring back to the web server log analysis example in “Simple Log Analysis”.
- The pattern of data processing in MapReduce is very similar to this example:
    - Read a set of input files, and break it up into records. In the web server log example, each record is one line in the log (that is, \n is the record separator).
    - **Map** - Call the mapper function to extract a key and value from each input record. In the preceding example, the mapper function is awk '{print $7}': it extracts the URL ($7) as the key, and leaves the value empty.
    - Sort all of the key-value pairs by key. In the log example, this is done by the first sort command.
    - **Reduce** - Call the reducer function to iterate over the sorted key-value pairs. If there are multiple occurrences of the same key, the sorting has made them adjacent in the list, so it is easy to combine those values without having to keep a lot of state in memory. In the preceding example, the reducer is implemented by the command uniq -c, which counts the number of adjacent records with the same key.

6.5.2. Map Reduce - How to create a MapReduce job?

Need to implement two callback functions, the mapper and reducer, which behave as follows:

- **Mapper**:
  The mapper is called once for every input record, and its job is to extract the key and value from the input record. For each input, it may generate any number of key-value pairs (including none). It does not keep any state from one input record to the next, so each record is handled independently.
- **Reducer**:
  The MapReduce framework takes the key-value pairs produced by the mappers, collects all the values belonging to the same key, and calls the reducer with an iterator over that collection of values. The reducer can produce output records (such as the number of occurrences of the same URL).

6.5.3. Map Reduce - Join Algorithms

- **Sort-merge joins**
  Each of the inputs being joined goes through a mapper that extracts the join key. By partitioning, sorting, and merging, all the records with the same key end up going to the same call of the reducer. This function can then output the joined records.
- **Broadcast hash joins**
  One of the two join inputs is small, so it is not partitioned and it can be entirely loaded into a hash table. Thus, you can start a mapper for each partition of the large join input, load the hash table for the small input into each mapper, and then scan over the large input one record at a time, querying the hash table for each record.
- **Partitioned hash joins**
  If the two join inputs are partitioned in the same way (using the same key, same hash function, and same number of partitions), then the hash table approach can be used independently for each partition.

6.6. Beyond MapReduce

- MapReduce is a fairly clear and simple abstraction on top of a distributed filesystem.
- Simple to understand what it is doing.
- Not easy to use. Implementing a complex processing job using the raw MapReduce APIs is actually quite hard and laborious—for instance, you would need to implement any join algorithms from scratch.
- Various higher-level programming models (Pig, Hive, Cascading, Crunch) were created as abstractions on top of MapReduce.
- If you understand how MapReduce works, they are fairly easy to learn, and their higher-level constructs make many common batch processing tasks significantly easier to implement.
- There are also problems with the MapReduce execution model itself
- Other tools are sometimes orders of magnitude faster for some kinds of processing.

6.6. Beyond MapReduce

- Every MapReduce job is independent from every other job.
- The main contact points of a job with the rest of the world are its input and output directories on the distributed filesystem.
- If you want the output of one job to become the input to a second job, you need to configure the second job's input directory to be the same as the first job's output directory, and an external workflow scheduler must start the second job only once the first job has completed.
- This setup is reasonable if the output from the first job is a dataset that you want to publish widely within your organization. In that case, you need to be able to refer to it by name and reuse it as input to several different jobs (including jobs developed by other teams).
- Publishing data to a well-known location in the distributed filesystem allows loose coupling so that jobs don't need to know who is producing their input or consuming their output.
- In many cases, you know that the output of one job is only ever used as input to one other job, which is maintained by the same team. In this case, the files on the distributed filesystem are simply intermediate state: a means of passing data from one job to the next. In the complex workflows used to build recommendation systems consisting of 50 or 100 MapReduce jobs, there is a lot of such intermediate state.

6.6.1. - Beyond MapReduce - MapReduce downsides

- A MapReduce job can only start when all tasks in the preceding jobs (that generate its inputs) have completed, whereas processes connected by a Unix pipe are started at the same time, with output being consumed as soon as it is produced. Skew or varying load on different machines means that a job often has a few straggler tasks that take much longer to complete than the others. Having to wait until all of the preceding job's tasks have completed slows down the execution of the workflow as a whole.
- Mappers are often redundant: they just read back the same file that was just written by a reducer, and prepare it for the next stage of partitioning and sorting. In many cases, the mapper code could be part of the previous reducer: if the reducer output was partitioned and sorted in the same way as mapper output, then reducers could be chained together directly, without interleaving with mapper stages.
  Storing intermediate state in a distributed filesystem means those files are replicated across several nodes, which is often overkill for such temporary data.

6.6.2. Beyond MapReduce - Data-flow Engines

- In order to fix the problems with MapReduce, several new execution engines for distributed batch computations were developed.
- The most well known of data-flow engines are Spark, Tez, and Flink.
- There are various differences in the way they are designed, but they have one thing in common: they handle an entire workflow as one job, rather than breaking it up into independent sub-jobs.
- Since they explicitly model the flow of data through several processing stages, these systems are known as data-flow engines.
- Like MapReduce, they work by repeatedly calling a user-defined function to process one record at a time on a single thread. They parallelize work by partitioning inputs, and they copy the output of one function over the network to become the input to another function.
- Unlike in MapReduce, these functions need not take the strict roles of alternating map and reduce, but instead can be assembled in more flexible ways, called function operators.

6.6.2. Beyond MapReduce - Data-flow Engines

- The data-flow engine provides several different options for connecting one operator's output to another's input:
  - One option is to repartition and sort records by key, like in the shuffle stage of MapReduce. This feature enables sort-merge joins and grouping in the same way as in MapReduce.
  - Another possibility is to take several inputs and to partition them in the same way, but skip the sorting. This saves effort on partitioned hash joins, where the partitioning of records is important but the order is irrelevant because building the hash table randomizes the order anyway.
- Data-flow engines like Spark, Flink, and Tez typically arrange the operators in a job as a directed acyclic graph (DAG).

Summary of Distributed Batch processing

- Distributed batch processing engines have a deliberately restricted programming model:
    - callback functions (such as mappers and reducers)
    - are assumed to be stateless and to have no externally visible side effects besides their designated output.
- This restriction allows the framework to hide some of the hard distributed systems problems behind its abstraction:
    - In the face of crashes and network issues, tasks can be retried safely, and the output from any failed tasks is discarded.
    - If several tasks for a partition succeed, only one of them actually makes its output visible.

Summary of Distributed Batch processing

- Thanks to the framework, your code in a batch processing job does not need to worry about implementing fault-tolerance mechanisms:
    - The framework can guarantee that the final output of a job is the same as if no faults had occurred, even though in reality various tasks perhaps had to be retried.
    - These reliable semantics are much stronger than what you usually have in online services that handle user requests and that write to databases as a side effect of processing a request.
- The distinguishing feature of a batch processing job is that it reads some input data and produces some output data, without modifying the input—in other words, the output is derived from the input. Crucially, the input data is bounded: it has a known, fixed size (for example, it consists of a set of log files at some point in time, or a snapshot of a database's contents). Because it is bounded, a job knows when it has finished reading the entire input, and so a job eventually completes when it is done.

6.7. Apache Spark

- Apache Spark™ is a unified analytics engine for large-scale data processing.
- Run workloads 100x faster than Hadoop.
- Write applications quickly in Python, Java, Scala, and R.
- Combine SQL, streaming, and complex analytics
  - SQL and DataFrames
  - MLlib
  - GraphX
  - Spark Streaming
- Runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.

6.7.1. Apache Spark - Databricks

- Databricks Community Edition

https://community.cloud.databricks.com/login.html

- Fully managed Apache Spark cluster

7. Stream Processing

- Transmitting Event Streams
  - Apache Kafka
- Databases and Streams
  - CDC
  - CDC with Debezium
- Processing Streams
  - Apache Spark
  - KSQL
  - ksqlDB

7.1.1. Transmitting Event Streams - Apache Kafka

- **Quick Start**

https://docs.confluent.io/current/quickstart/cloud-quickstart/index.html

- **Client configuration for Python**

https://github.com/confluentinc/examples/tree/master/clients/cloud/python#overview

7.1.1.1. Transmitting Event Streams - Apache Kafka - CLI and client configuration for MacOS X

- Install the Confluent Cloud CLI
  - Run this command to install the Confluent Cloud CLI.

    ```
    curl -L https://cnfl.io/ccloud-cli | sh -s -- -b /usr/local/binCopy
    ```

  - Note: This script will install the CLI in the /usr/local/bin directory by default. If you want to install it somewhere else, add the path to the end of the command and to your your $PATH variable.
- Log in to your Confluent Cloud organization using the Confluent Cloud CLI
  - Run this command to log in to the Confluent Cloud CLI.

    ```
    ccloud loginCopy
    ```

  - When prompted for your username and password, enter the same credentials that you used to log in to Confluent Cloud.

7.1.1.1. Transmitting Event Streams - Apache Kafka - CLI and client configuration for MacOS X

- Select your environment
  - Run this command to set your environment. To view the available environments, use the ccloud environment list command.

    ```
    ccloud environment use t22760Copy
    ```

- Select your cluster
  - Run this command to set your cluster. To view the available clusters, use the ccloud kafka cluster list command.

    ```
    ccloud kafka cluster use lkc-xqzq1Copy
    ```

7.1.1.1. Transmitting Event Streams - Apache Kafka - CLI and client configuration for MacOS X

- Use the CLI to create an API key and secret
  - Run this command to create an API key and secret. This is required to produce or consume your topic.

    ~~~
    ccloud api-key create --resource lkc-xqzq1Copy```
    After you have created your API key pair, copy the API
    Key and paste it at the end of this command:
    ```ccloud api-key use <API Key> --resource lkc-
    xqzq1Copy
    ~~~

    Note: Save your API Key and Secret pair in a safe and secure place.

7.1.1.2. Transmitting Event Streams - Apache Kafka - Create a topic

Run this command to create a topic named test-topic in your cluster:
ccloud kafka topic create test-topicCopy
Now, verify that your topic has been created:

```
ccloud kafka topic list
```

7.1.1.3. Transmitting Event Streams - Apache Kafka - Produce messages to your topic

Now you're ready to produce and consume some messages.
Run this command to start a console producer, which you can use to manually produce messages to test-topic:

```
`ccloud kafka topic produce test-topicCopy
```

Run that command, then produce a few messages to your cluster in Confluent Cloud using the console producer. You can do this by typing anything into the console and pressing Enter.

When you're done, press `ctrl+c` to exit the producer.

7.1.1.4. Transmitting Event Streams - Apache Kafka - Consume the messages you produced to your topic

To consume all of the messages in test-topic and print them to the console, run this command:

```
ccloud kafka topic consume -b test-topicCopy
`
```

Note: You should see the messages being consumed in real time in the window where your consumer is running.

Press `ctrl+c` to stop the consumer.

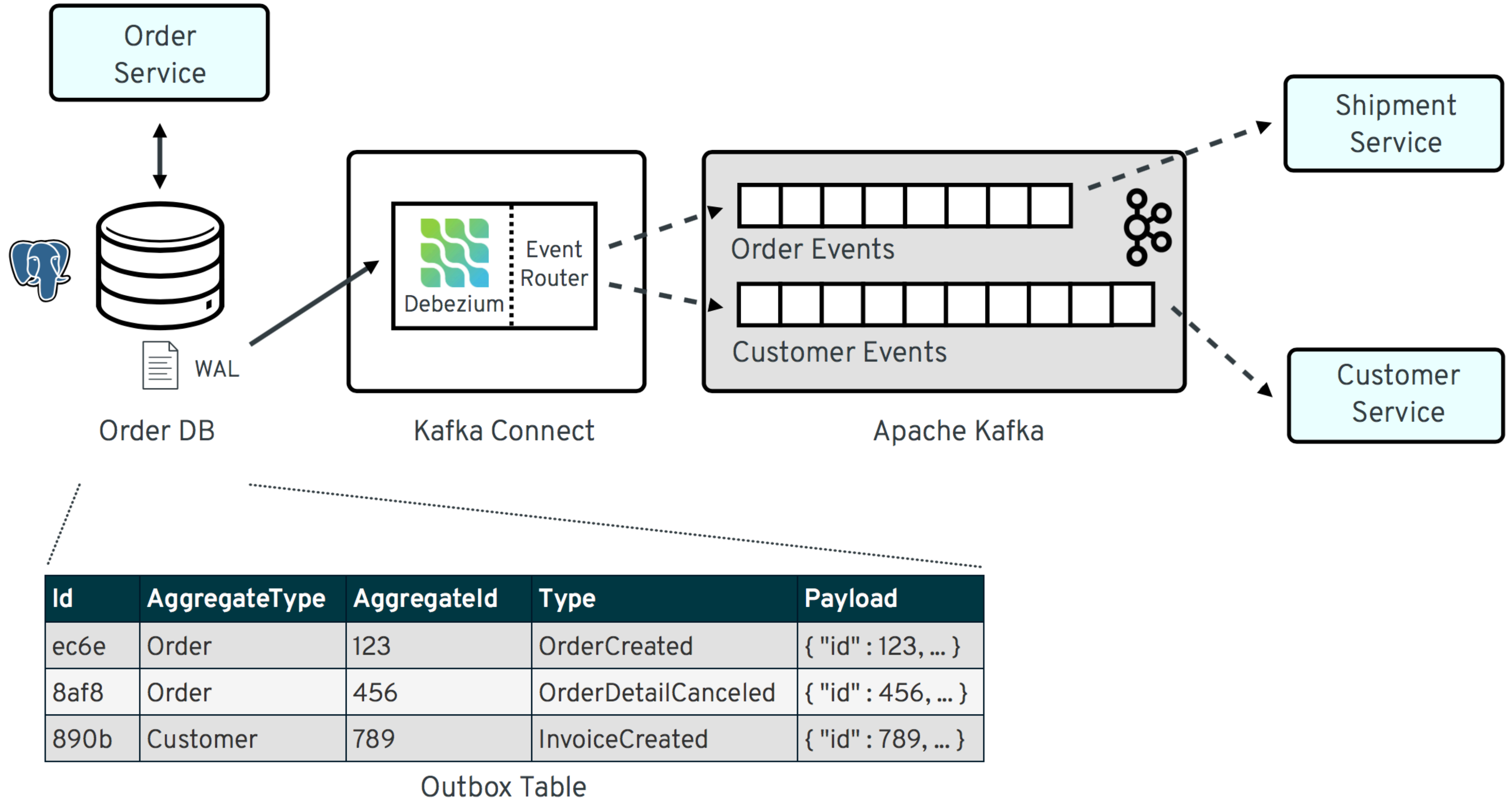
7.2. Databases and Streams

1. Change Data Capture (CDC)
2. CDC with debezium

7.2.1. Databases and Streams - Change Data Capture (CDC)

- In databases, change data capture (CDC) is a set of software design patterns used to determine (and track) the data that has changed so that action can be taken using the changed data.
- CDC is also an approach to data integration that is based on the identification, capture and delivery of the changes made to enterprise data sources.
- CDC solutions occur most often in data-warehouse environments since capturing and preserving the state of data across time is one of the core functions of a data warehouse, but CDC can be utilized in any database or data repository system.
- debezium.io

7.2.2. Databases and Streams - CDC with debezium



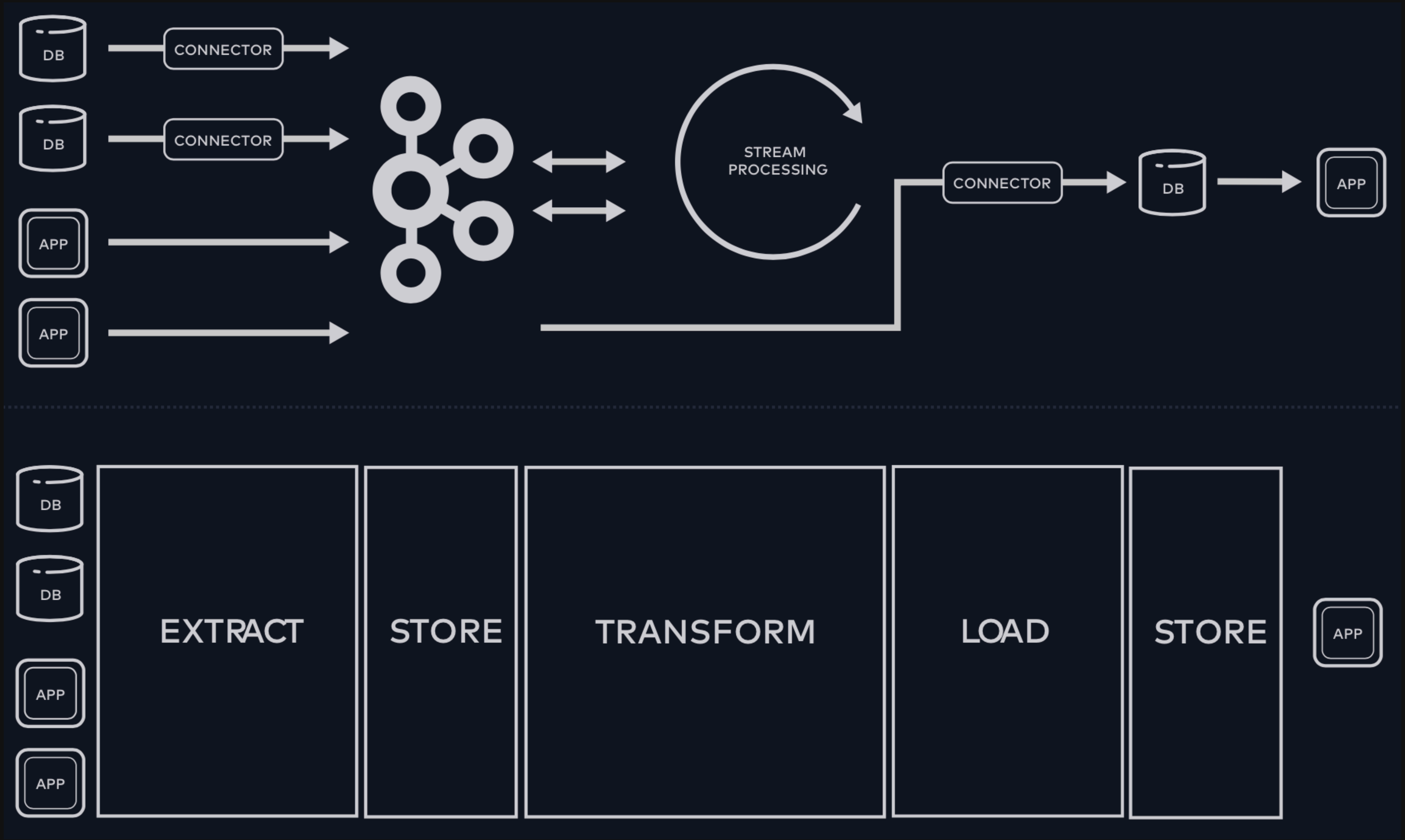
Id	AggregateType	AggregateId	Type	Payload
ec6e	Order	123	OrderCreated	{ "id" : 123, ... }
8af8	Order	456	OrderDetailCanceled	{ "id" : 456, ... }
890b	Customer	789	InvoiceCreated	{ "id" : 789, ... }

Outbox Table

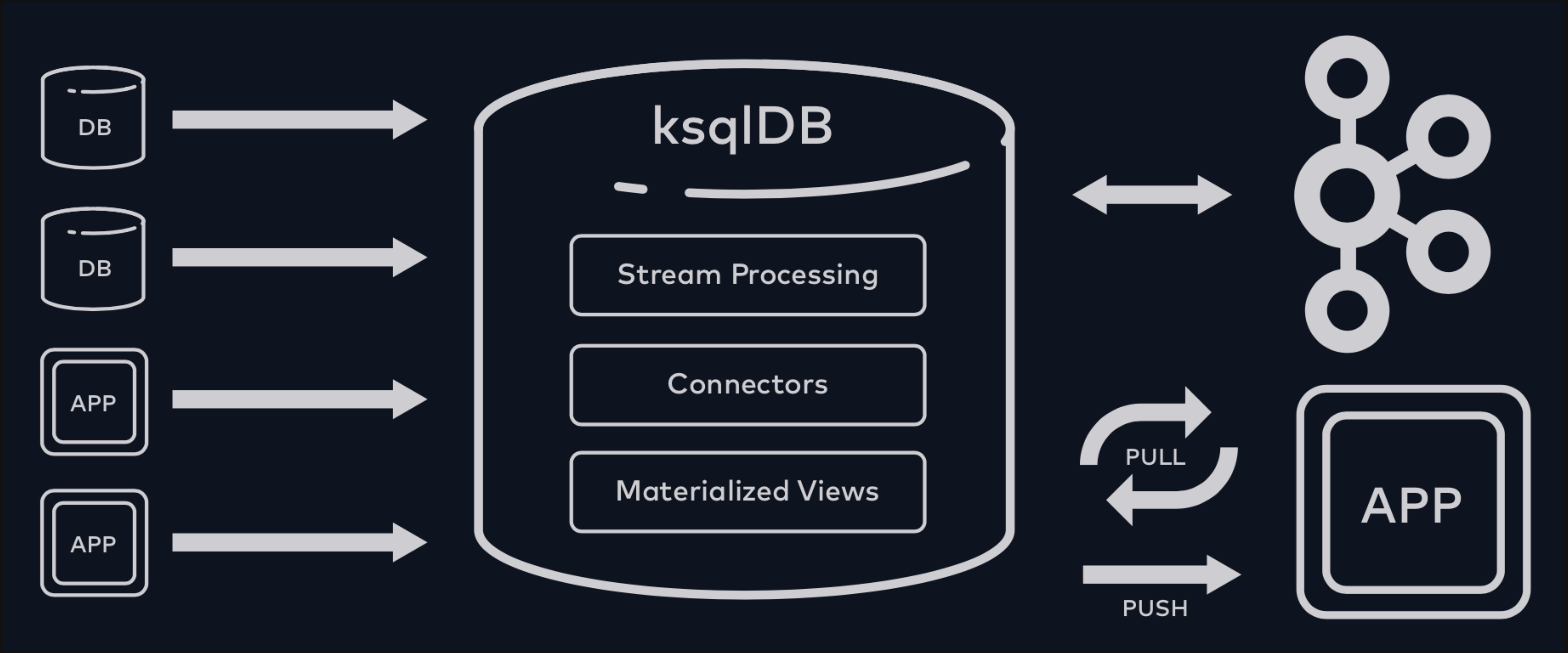
7.3. Processing Streams

1. KSQL
2. ksqlDB (Event Streaming Database)

7.3.1. Processing Streams - KSQL



7.3.2. Processing Streams - ksqlDB - Event Streaming Database



7.3.2. Processing Streams - ksqlDB - Event Streaming Database

- The event streaming database purpose-built for stream processing applications.
- Real-time
- Kafka-native
- High-level, lightweight syntax

7.3.2.1. Processing Streams - ksqlDB - Capture events

```
CREATE SOURCE CONNECTOR riders WITH (
  'connector.class' = 'JdbcSourceConnector',
  'connection.url'  = 'jdbc:postgresql:// ... ',
  'topic.prefix'    = 'rider',
  'table.whitelist' = 'geoEvents, profiles',
  'key'             = 'profile_id',
  ... );
```

7.3.2.2. Processing Streams - ksqlDB - Perform continuous transformations

```
CREATE STREAM locations AS
  SELECT rideId,
         latitude,
         longitude,
         GEO_DISTANCE(latitude,
                      longitude,
                      dstLatitude,
                      dstLongitude
                      ) AS kmToDst
  FROM geoEvents
  EMIT CHANGES;
```

7.3.2.3. Processing Streams - ksqlDB - Create materialized views

```
CREATE TABLE activePromotions AS
  SELECT rideId,
         qualifyPromotion(kmToDst) AS promotion
  FROM locations
  GROUP BY rideId
  EMIT CHANGES;
```

7.3.2.4. Processing Streams - ksqlDB - Serve lookups against materialized views

```
SELECT rideId, promotion
FROM activePromotions
WHERE ROWKEY = '6fd0fcdb';
```

7.3.2.5. Processing Streams - ksqlDB - Collections

- **Streams**:
  Streams are immutable, append-only sequences of events. They're useful for representing a series of historical facts.
- **Tables**:
  Tables are mutable collections of events. They let you represent the latest version of each value per key.

7.3.2.5. Processing Streams - ksqlDB - Collections - Create a stream

```
CREATE STREAM routeWaypoints (
       vehicleId VARCHAR,
       latitude DOUBLE(10, 2),
       longitude DOUBLE(10, 2)
) WITH (
       kafka_topic = 'locations',
       partitions = 3,
       key = 'vehicleId',
       value_format = 'json'
);
```

7.3.2.5. Processing Streams - ksqlDB - Collections - Create a table

```
CREATE TABLE currentCarLocations (
       vehicleId VARCHAR,
       latitude DOUBLE(10, 2),
       longitude DOUBLE(10, 2)
) WITH (
       kafka_topic = 'locations',
       partitions = 3,
       key = 'vehicleId',
       value_format = 'json'
);
```

7.3.2.6. Processing Streams - ksqlDB - Stream Processing

- Stream processing enables you to execute continuous computations over unbounded streams of events, ad infinitum.
- Transform, filter, aggregate, and join collections together to derive new collections or materialized views that are incrementally updated in real-time as new events arrive.
- Queries
    - Push
    - Pull

7.3.2.6. Processing Streams - ksqlDB - Stream Processing - Push queries

- Push queries let you subscribe to a query's result as it changes in real-time.
- When new events arrive, push queries emit refinements, which allow you to quickly react to new information.
- They’re a perfect fit for asynchronous application flows.

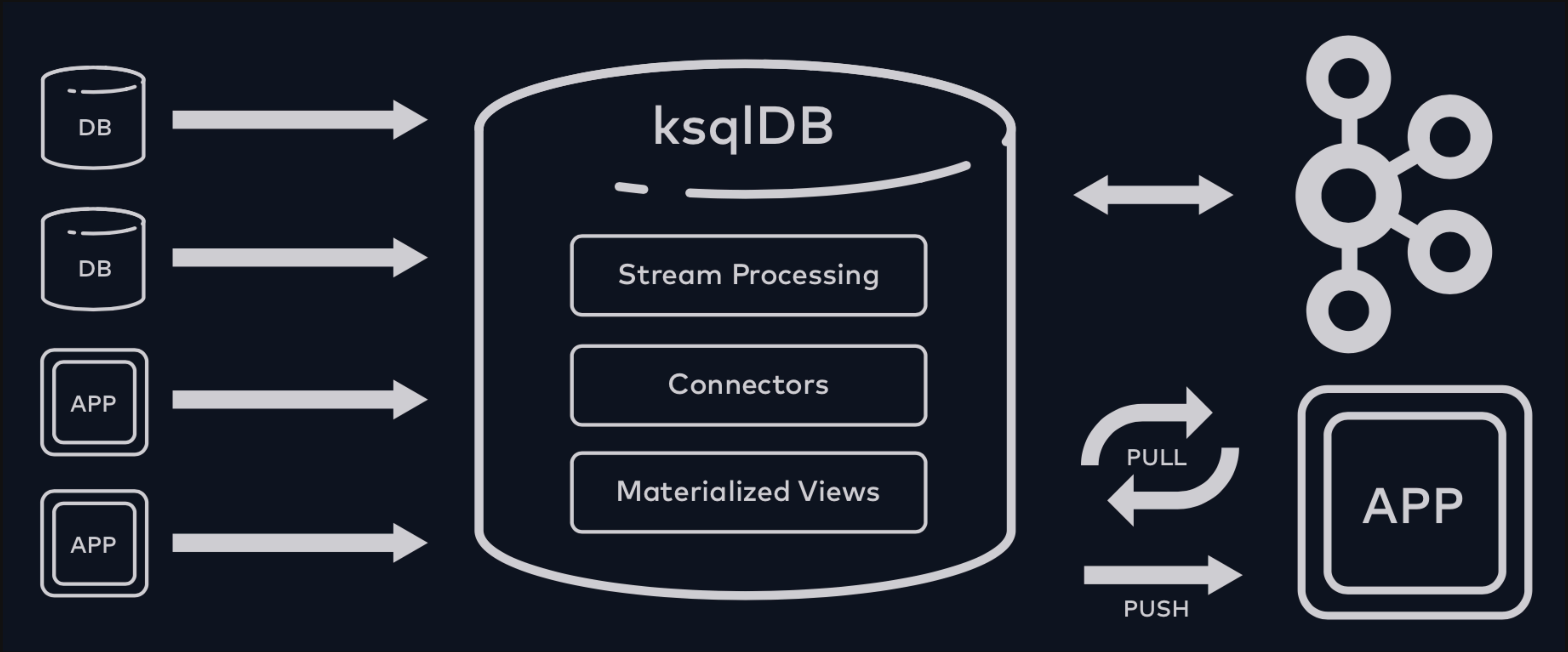
```
SELECT vehicleId,
       latitude,
       longitude
FROM currentCarLocations
WHERE ROWKEY = '6fd0fcdb'
EMIT CHANGES;
```

7.3.2.6. Processing Streams - ksqlDB - Stream Processing - Pull queries

- Pull queries allow you to fetch the current state of a materialized view.
- Because materialized views are incrementally updated as new events arrive, pull queries run with predictably low latency.
- They're a great match for request/response flows.

```
SELECT vehicleId,
       latitude,
       longitude
FROM currentCarLocations
WHERE ROWKEY = '6fd0fcdb';
```

7.3.2. Processing Streams - ksqlDB - Event Streaming Database



Thanks

Reference

- Designing Data-Intensive Applications by Martin Kleppmann - Chapters 10 and 11