EDEM
Escuela de Empresarios

Bases de Datos NOSQL con Cassandra

GFT

Esteban Chiner

Curso 2021/2022

Fecha 26/11/2021

Agenda

1. Introduction

- Storage formats
- NOSQL databases
- CAP theorem and why NOSQL

2. Cassandra

- Columnar databases or key-value stores
- Introduction to Cassandra & Features
- Cassandra internals
- Cassandra Query Language (CQL)
- SMACK Architecture

3. Success Stories

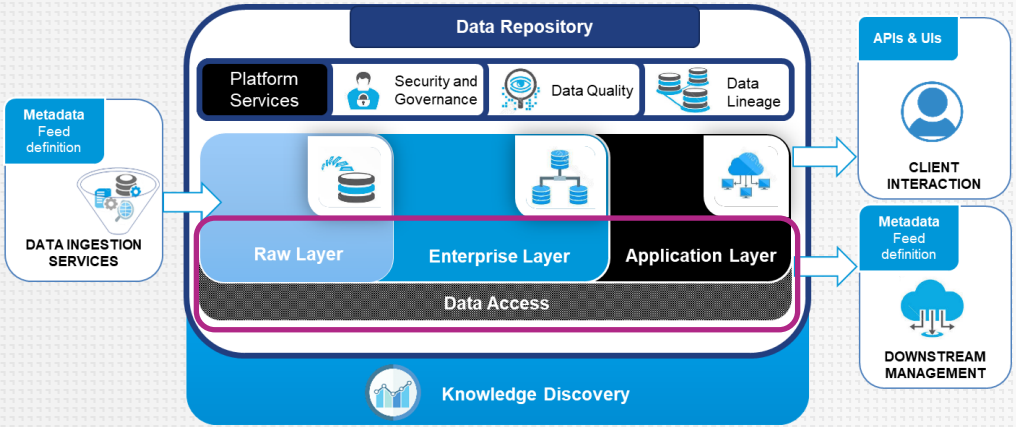
- SDM & Digital eXperience as a Platform
- IoT Data Lake in the Cloud
- Market Data Lake

Introduction

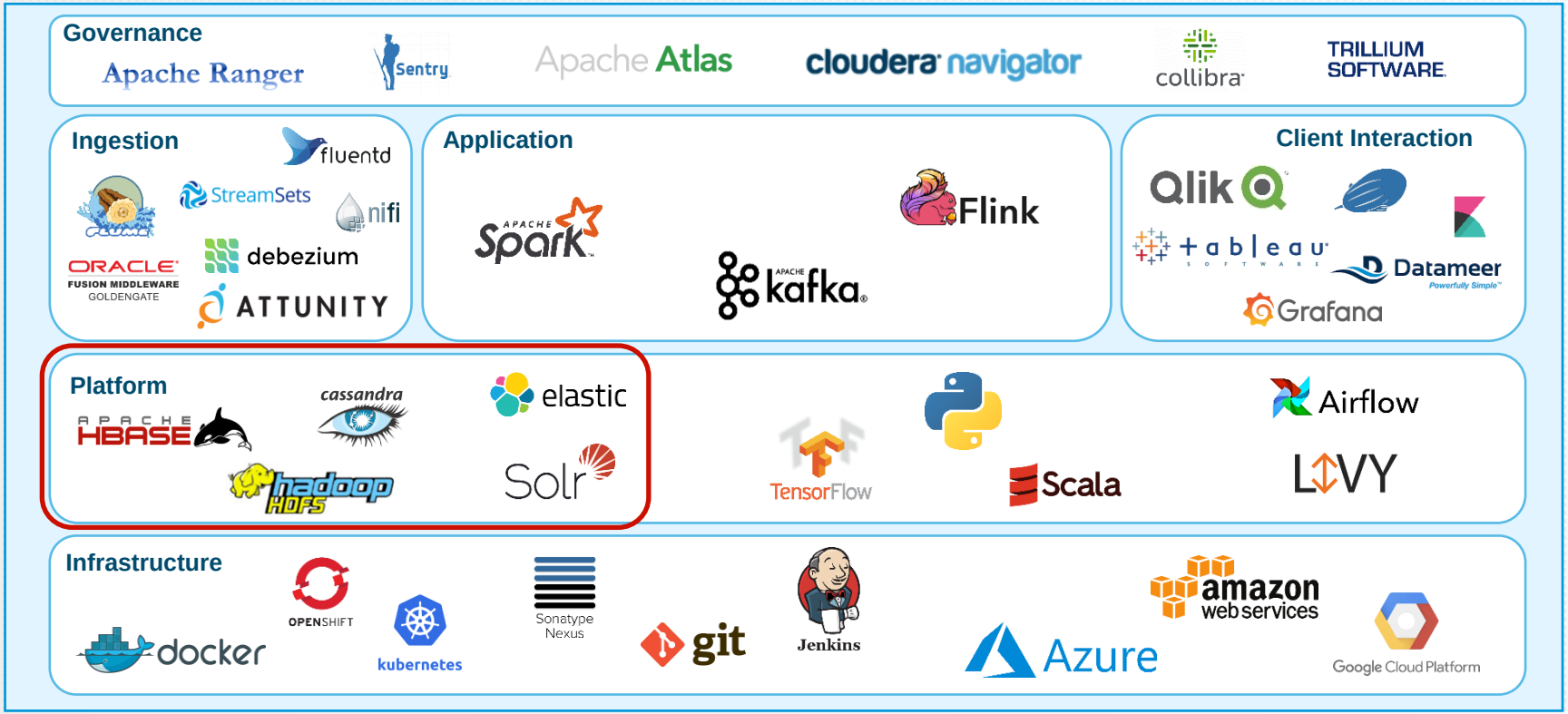
Big Data Storage

- Data Storage importance in Big Data Architectures is paramount given the **huge impact** it has on the following aspects:
  - **Ingestion**: How the data is ingested
  - **Volumes**: How much data we can store
  - **Accessibility**: How the data is accessed

- The **NOSQL types** available are the following, each having its pros and cons:
  - File
  - Key-Value
  - Document
  - Graph
  - In-Memory
  - Indexed



Technology Landscape



File Storage

Description

File Storage *is a distributed, fault-tolerant and usually POSIX compatible file storage. It behaves similar to a regular* ***file system*** *and can store large datasets, which are under the hood divided in blocks and replicated.*

Key aspects

- Simplest and cheapest way to store data
- Useful for sequential reading
- Storage can be
  - Text o Binary (Sequence Files)
  - Structured (Avro) or Unstructured (Images)
  - Column oriented (Parquet)

Technologies

File Storage Formats

	Avro	Parquet	ORC
Schema Evolution Support			
Compression			
Splitability			
Most Compatible Platforms	Kafka, Druid	Impala, Arrow Drill, Spark	Hive, Presto
Row or Column	Row	Column	Column
Read or Write	Write	Read	Read

Source: Nexla analysis, April 2018

Key-Value Store

Description

Key-Value Store *(or column-oriented databases) is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a* ***dictionary*** *or* ***hash table****.*

Key aspects

- Suitable for
  - Very well-defined access patterns
  - Either random read or write access
- Scales very well
- Dynamic columns
- Schema on read

Technologies

Document Store

Description

*A **Document Store** (or document-oriented database), or, is a computer program designed for storing, retrieving and managing **document-oriented information**, also known as **semi-structured data** (e.g. JSON or XML).*

Key aspects

- Own query language
- Easy to store and retrieve documents
- REST APIs (for JSON stores)

Technologies

Graph Database

Description

*A **Graph Database** is a database that uses graph structures for semantic queries with **nodes**, **edges**, and **properties** to represent and store data.*

Key aspects

- Good to represent relationships (networks)
  - Own query language
  - Performance
  - Presentation
- Does not scale well

Technologies

neo4j

Amazon Neptune

In-Memory Database

Description

*An **In-Memory database** (IMDB, also main memory database system or MMDB or memory resident database) is a database management system that primarily **relies on main memory** for computer data storage. It is contrasted with database management systems that employ a disk storage mechanism.*

Key aspects

- Great performance
- Hard to manage
  - Data state
  - Distribution (scalability)
- Suitable as
  - Cache
  - Data store for streaming applications

Technologies

hazelcast

apache Ignite

ALLUXIO

redis

Indexed

Description

*A **Indexed Database** can be considered as a **sub-type of a document store**, providing a distributed, multitenant-capable **full-text search engine** with an HTTP web interface and schema-free JSON documents.*

Key aspects

- Shares many features of document stores (own QL, storage of documents, REST API)
- Text search features such as
  - Faceting
  - Free text search
  - Synonyms
  - Autocomplete

Technologies

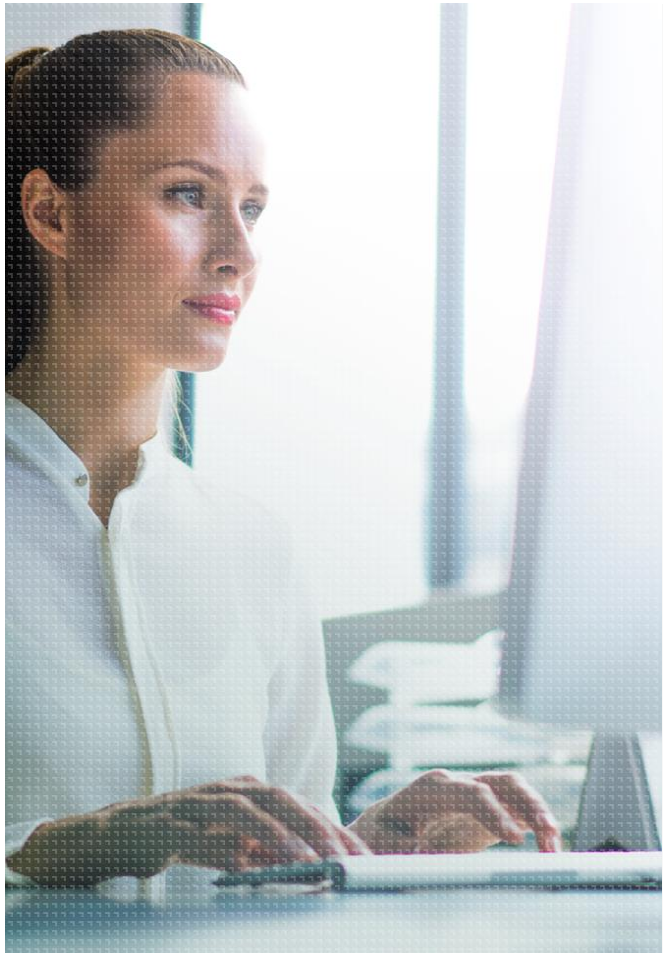
elasticsearch

Solr

DB-Engines Ranking

Rank Mar 2020	Rank Feb 2020	Rank Mar 2019	DBMS	Database Model	Score Mar 2020	Score Feb 2020	Score Mar 2019
1.	1.	1.	Oracle	Relational, Multi-model	1340.64	-4.11	+61.50
2.	2.	2.	MySQL	Relational, Multi-model	1259.73	-7.92	+61.48
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1097.86	+4.11	+50.01
4.	4.	4.	PostgreSQL	Relational, Multi-model	513.92	+6.98	+44.11
5.	5.	5.	MongoDB	Document, Multi-model	437.61	+4.28	+36.27
6.	6.	6.	IBM Db2	Relational, Multi-model	162.56	-2.99	-14.64
7.	7.	↑ 9.	Elasticsearch	Search engine, Multi-model	149.17	-2.98	+6.38
8.	8.	8.	Redis	Key-value, Multi-model	147.58	-3.84	+1.46
9.	9.	↓ 7.	Microsoft Access	Relational	125.14	-2.92	-21.07
10.	10.	10.	SQLite	Relational	121.95	-1.41	-2.92
11.	11.	11.	Cassandra	Wide column	120.95	+0.60	-1.84
12.	12.	↑ 13.	Splunk	Search engine	88.52	-0.26	+5.42
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	88.35	+1.01	+4.04
14.	14.	↑ 15.	Hive	Relational	85.38	+1.85	+12.38
15.	15.	↓ 14.	Teradata	Relational, Multi-model	77.84	+1.03	+2.63
16.	16.	↑ 21.	Amazon DynamoDB	Multi-model	62.51	+0.38	+8.02
17.	17.	↓ 16.	Solr	Search engine	55.09	-1.07	-4.92
18.	18.	↑ 20.	SAP HANA	Relational, Multi-model	54.27	-0.70	-1.24
19.	19.	↓ 18.	FileMaker	Relational	54.16	-0.72	-3.97
20.	↑ 21.	↓ 19.	SAP Adaptive Server	Relational	52.77	+0.04	-3.27
21.	↑ 22.	↑ 22.	Neo4j	Graph	51.78	+0.57	+3.20
22.	↓ 20.	↓ 17.	HBase	Wide column	51.15	-1.80	-7.64
23.	↑ 25.	↑ 25.	Microsoft Azure SQL Database	Relational, Multi-model	35.44	+4.04	+7.51
24.	↓ 23.	↓ 23.	Couchbase	Document, Multi-model	32.08	-0.08	-1.72
25.	↓ 24.	↑ 27.	Microsoft Azure Cosmos DB	Multi-model	31.63	-0.32	+6.81

Source: https://db-engines.com/en/ranking



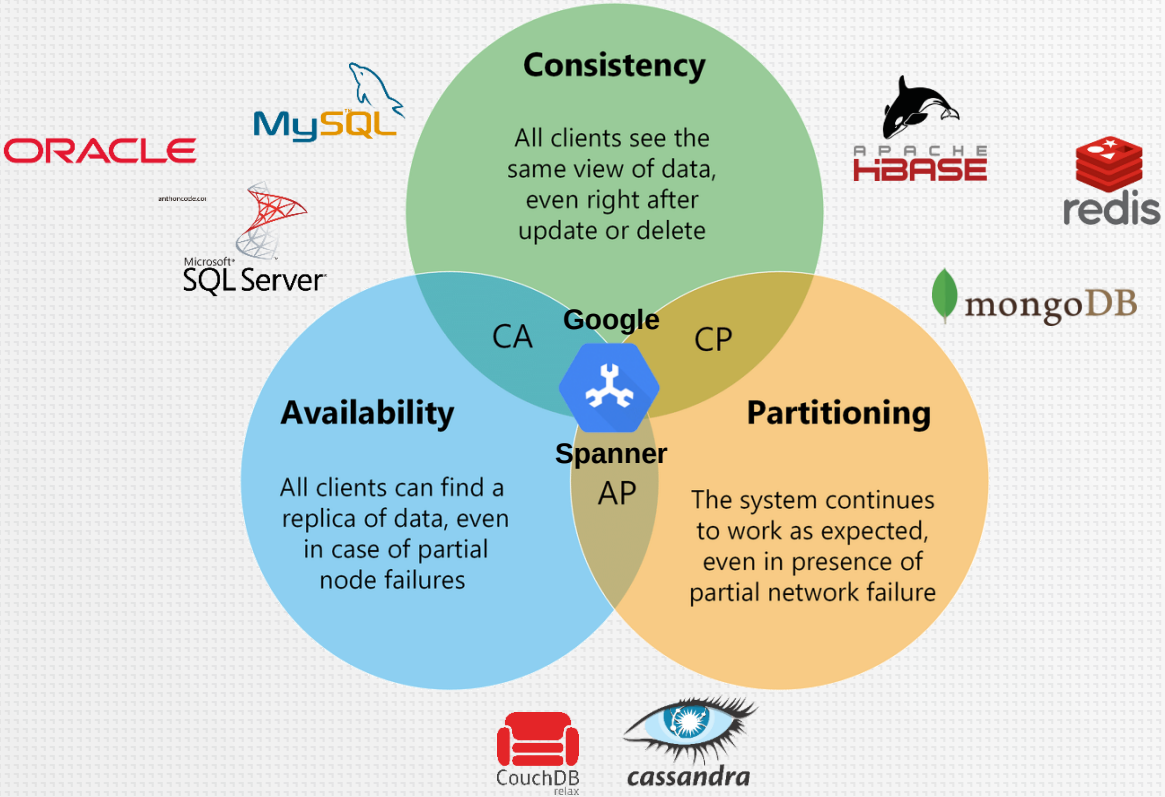
Exercise 0: Setup

Let’s do the initial setup:

https://github.com/echiner/edem-mda-nosql-cassandra

This will launch the main components, and we will run the first query to confirm the installation.

CAP Theorem


Consistency
All clients see the same view of data, even right after update or delete
Availability
All clients can find a replica of data, even in case of partial node failures
Partitioning
The system continues to work as expected, even in presence of partial network failure
CA
CP
AP
Google
Spanner
ORACLE
MySQL
Microsoft SQL Server
APACHE HBASE
redis
mongoDB
CouchDB
cassandra

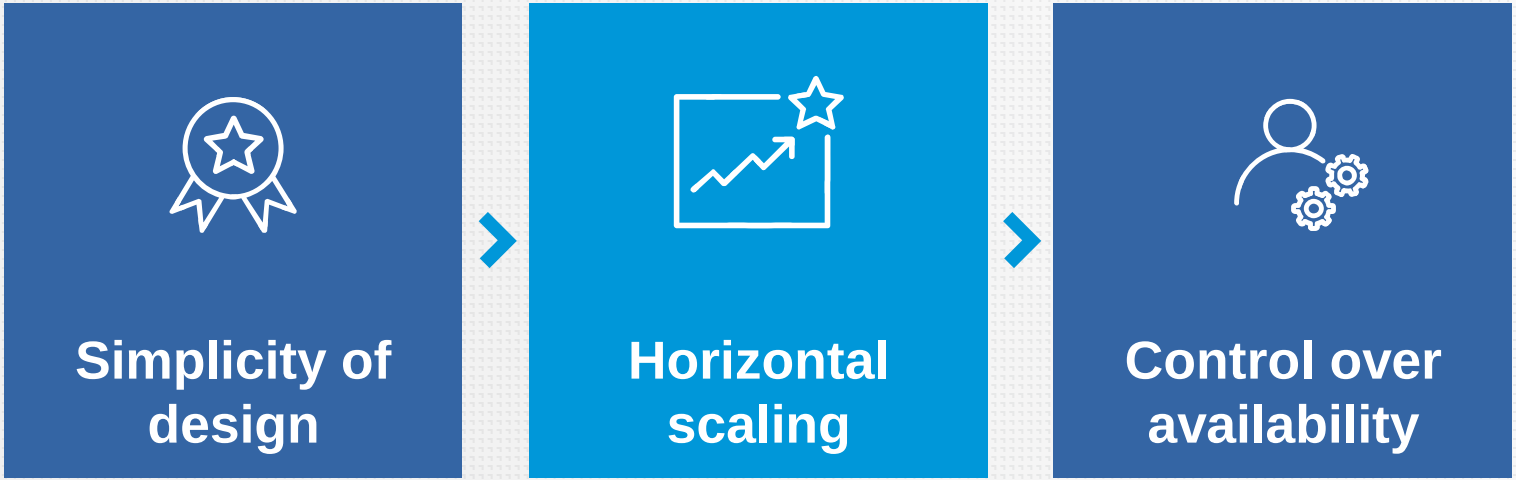
NOSQL evolution

SQL

NoSQL

NOSQL

Motivations of NOSQL databases



Cassandra

Introduction to columnar databases

- Also known as **key-value stores**
- Rows are “**self contained**” (include both the column name, the value and, usually, the timestamp)
- Columns can be added or removed, per row (**no static columns**)
- There can be many (thousands) of **columns per table**

7b976c48...	name: Bill Watterson	state: DC	birth_date: 1953
7c8f33e2...	name: Howard Tayler	state: UT	birth_date: 1968
7d2a3630...	name: Randall Monroe	state: PA	
7da30d76...	name: Dave Kellett	state: CA	

Differences between SQL and NOSQL

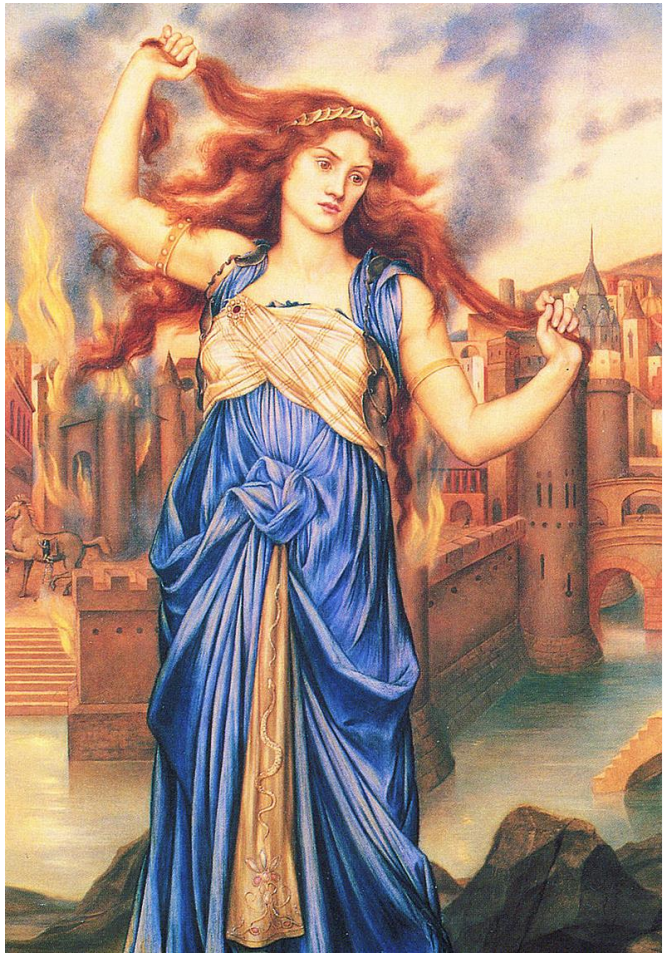
Relational (SQL) Database	NOSQL Database
Supports powerful query language	Supports very simple query language
It has a fixed schema	No fixed schema
Follows ACID (Atomicity, Consistency, Isolation, and Durability)	It is only “eventually consistent”
Supports transactions	Does not support transactions

Introduction to Cassandra

- Apache Cassandra is a **free** and **open-source**, **distributed**, **wide column store**, NOSQL database management system designed to **handle large amounts of data** across many commodity servers, providing **high availability** with **no single point of failure**.

- Cassandra offers robust support for clusters spanning **multiple datacenters**, with asynchronous **masterless** replication allowing **low latency** operations for all clients.

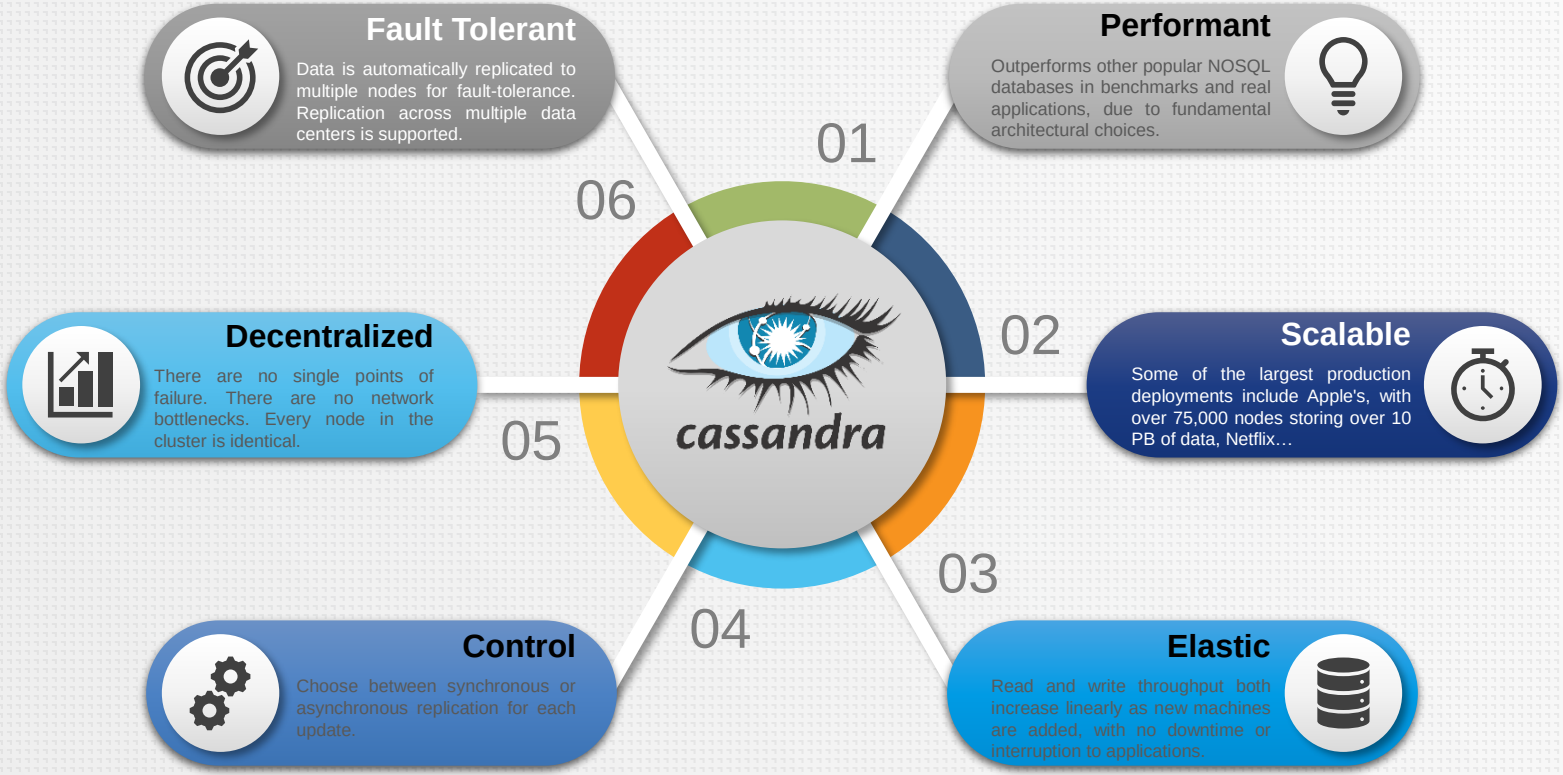
BigTable & Dynamo → Facebook creates Cassandra → **2008**: Open-sourced → **2009**: Apache Incubator → **2010**: Apache Top Level → **2011**: 1.0 released → **2013**: 2.0 released → **2015**: 3.0 released



Fun (mythology) fact

- Who is **Cassandra**?
- Was a priestess of Apollo in Greek mythology, and was given the gift of prophecy (**Oracle**)
- Upon refusing Apollo, she was **cursed** so that her prophecies would not be believed
- Sounds familiar?

Features


Fault Tolerant
Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported.
01
Performant
Outperforms other popular NOSQL databases in benchmarks and real applications, due to fundamental architectural choices.
02
Scalable
Some of the largest production deployments include Apple's, with over 75,000 nodes storing over 10 PB of data, Netflix…
03
Elastic
Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.
04
Control
Choose between synchronous or asynchronous replication for each update.
05
Decentralized
There are no single points of failure. There are no network bottlenecks. Every node in the cluster is identical.
06
cassandra

When to use Cassandra

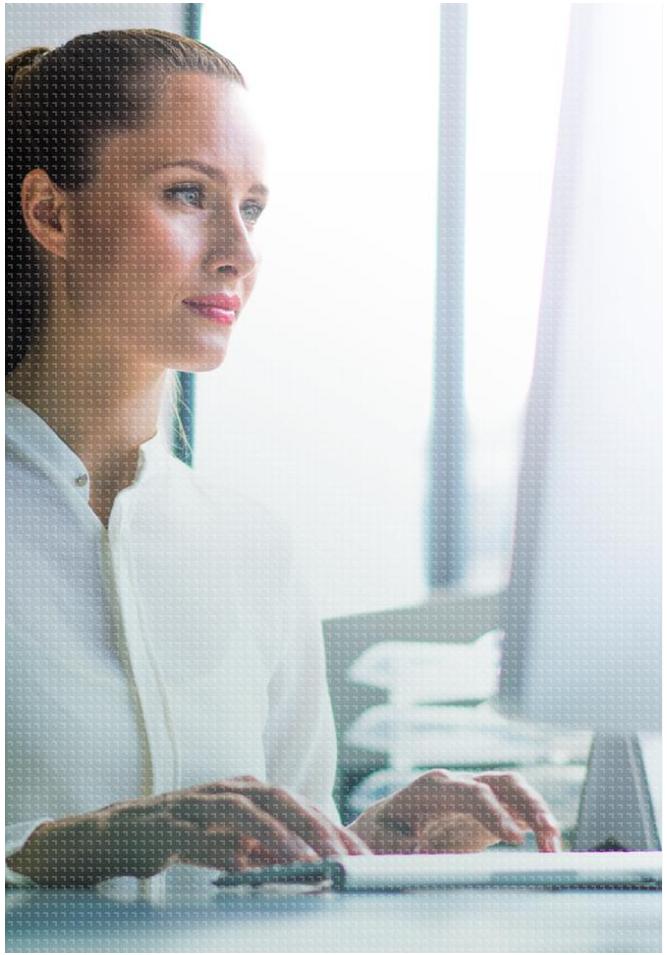
The **ideal Cassandra application** has the following characteristics:

- **Writes exceed reads** by a large margin
- **Data is rarely updated** and when updates are made they are idempotent
- Read Access is by a **known primary key**
- **Data can be partitioned** via a key that allows the database to be spread evenly across multiple nodes
- There is **no need for joins or aggregates**

Typical use cases

Some **use cases** for Cassandra would be:

- **Transaction logging**: Purchases, test scores, movies watched and user latest location
- Storing **time series** data (as long as you do your own aggregates)
- **Tracking** pretty much anything including order status, packages, etc.
- Storing **health tracker** data
- Weather service **history**
- Internet of things (**IoT**) status and event history



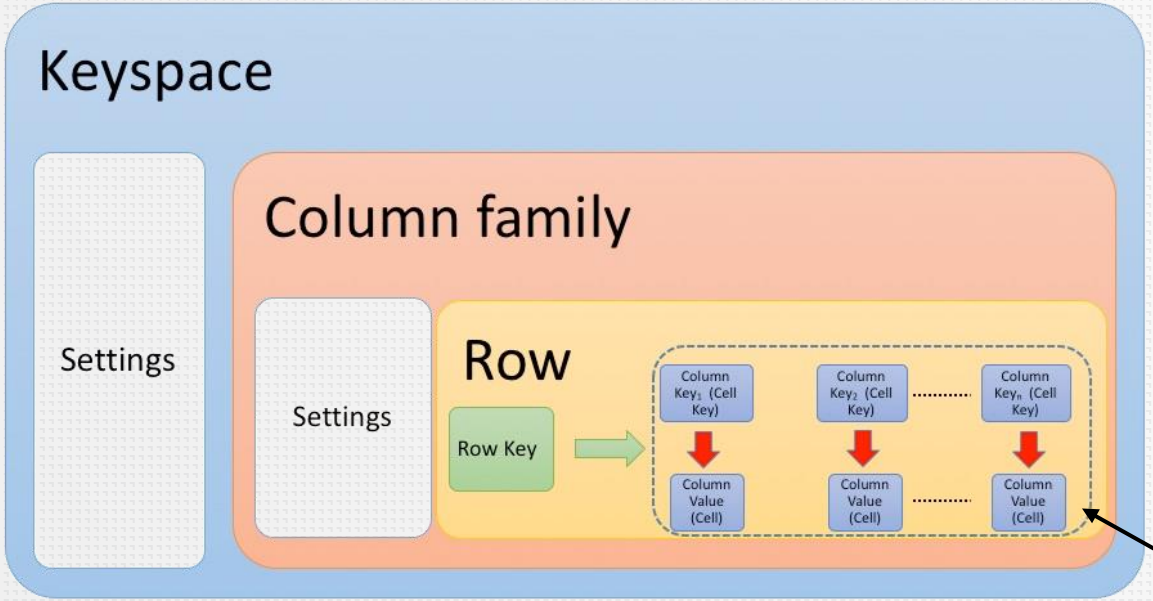
Exercise 1: Basic syntax

In **CQL console** (cqlsh):

- Help
- Describe tables
- Select system tables
- Create a new table
- Insert data
- Query

Follow the steps in “Exercise 1” in the GIT repository.

Basic concepts



Primary keys

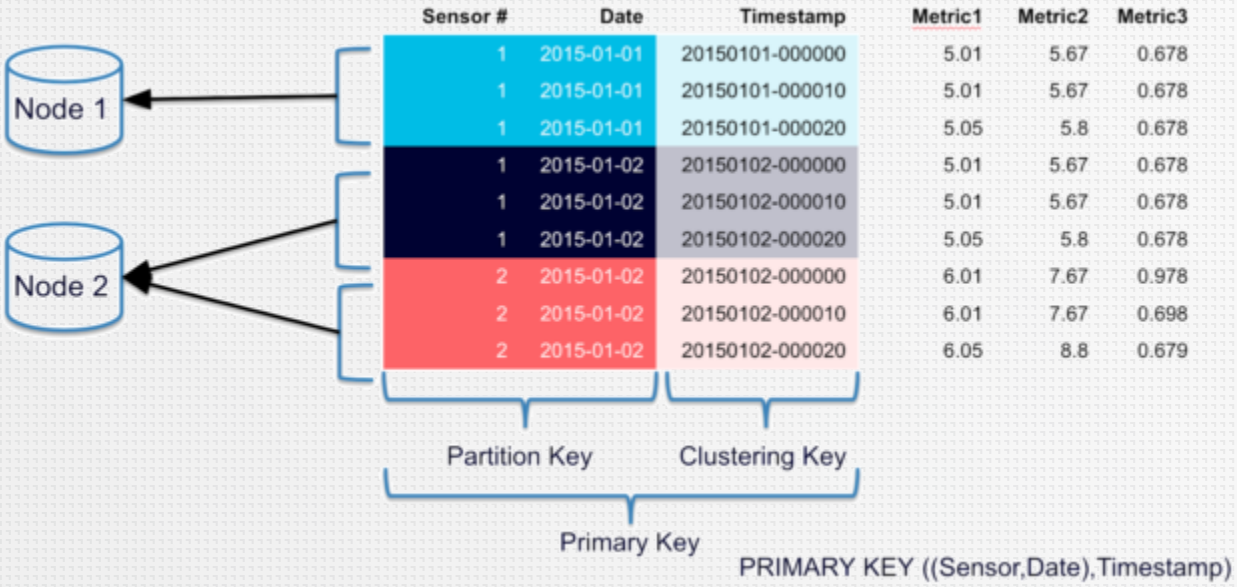
Key concepts in Cassandra are different from Relational databases.

- **Primary key**: It is what it's called a partition key. A part from uniqueness of the record in the database, it determines data locality or in which node must be stored.
- **Compound Primary Key**: A compound primary key is comprised of one or more columns that are referenced in the primary key. It is formed liked that:

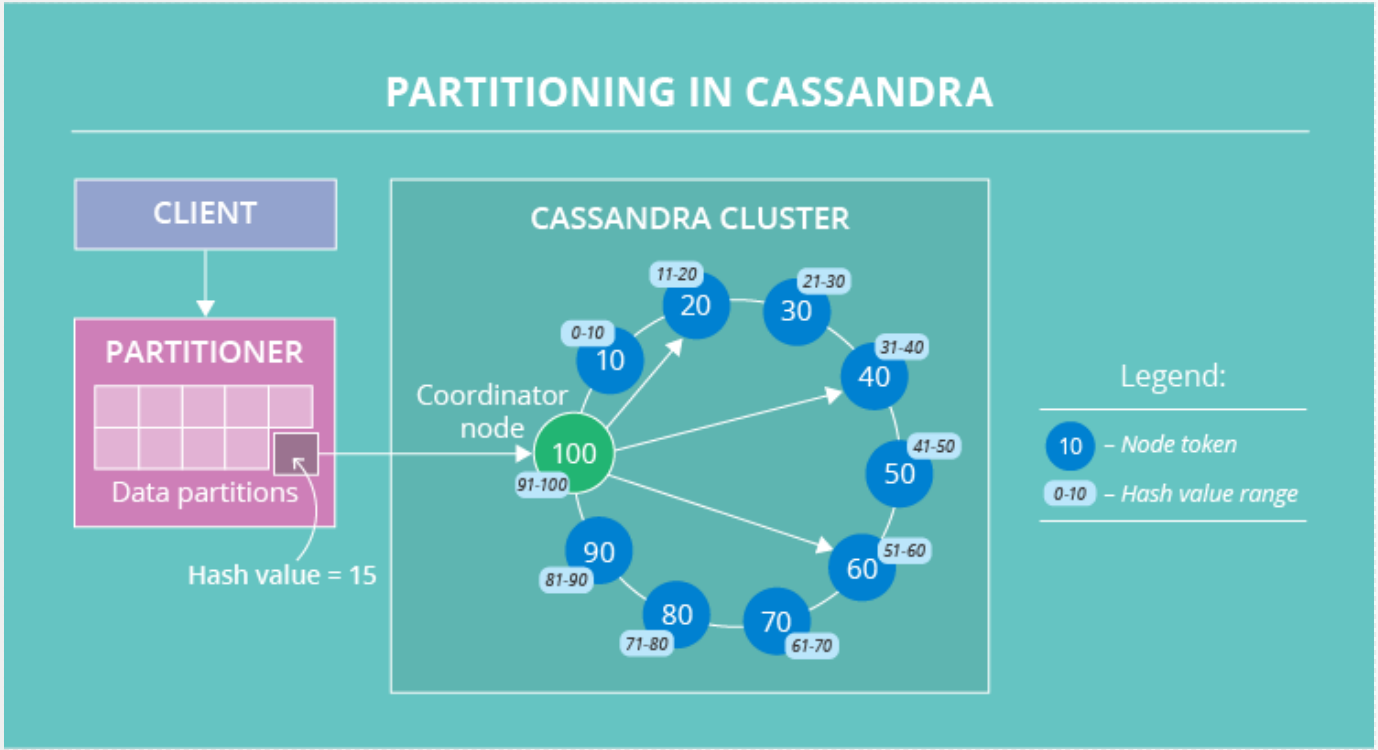
  ((Partition_Key_1,...Partition_Key_N), (Cluster_Key_1,...Cluster_Key_N))

  - **Partition Key**: The purpose of a partition key is to identify the partition or node in the cluster that stores that row.
  - **Clustering Key**: The purpose of the clustering key is to store row data in a sorted order.

Primary Keys: Partitioning and clustering



Primary Keys: Partitioning and clustering



Choosing the Primary Key

- Always take into account the **data access pattern**, i.e. how the data is going to be read. That will determine the Primary Key composition.

- For **example**:

```
Create table MusicPlaylist
    (
        SongId int,
        SongName text,
        Year int,
        Singer text,
        Primary key(SongId, SongName)
    );
```

Partitioning Key **Clustering Column**

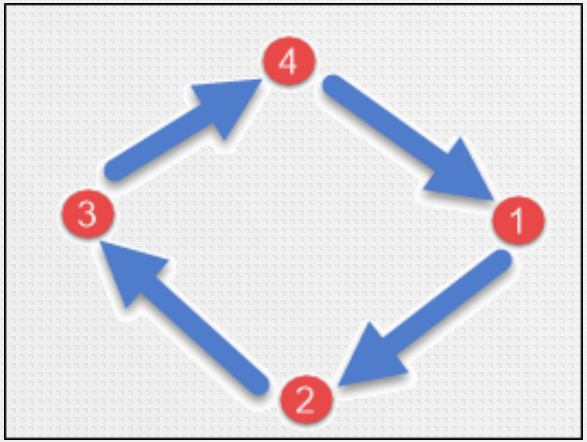
```
Create table MusicPlaylist
    (
        SongId int,
        SongName text,
        Year int,
        Singer text,
        Primary key((SongId, Year), SongName)
    );
```

Partitioning (Composite) Key **Clustering Column**

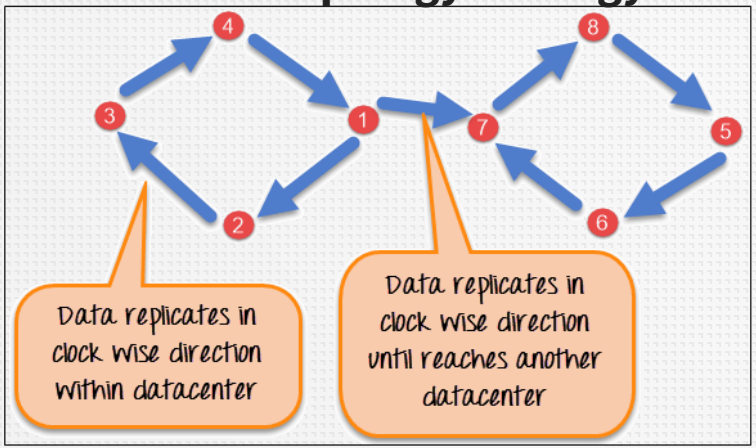
Replication

- The **replication factor** defines how many times the data is replicated
- The **replication strategy** defines how the data is replicated across the cluster:

SimpleStrategy

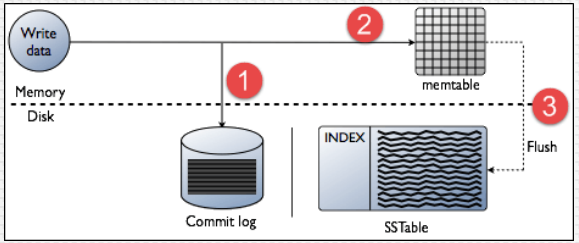


NetworkTopologyStrategy

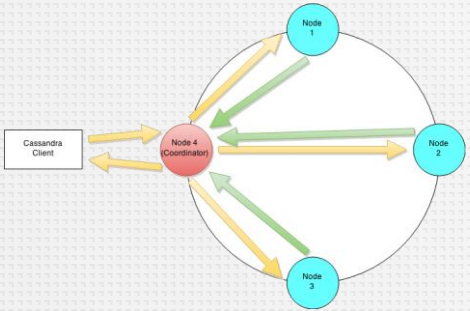


Writing data

- The coordinator sends a **write request** to replicas. If all the replicas are up, they will receive write request regardless of their consistency level.

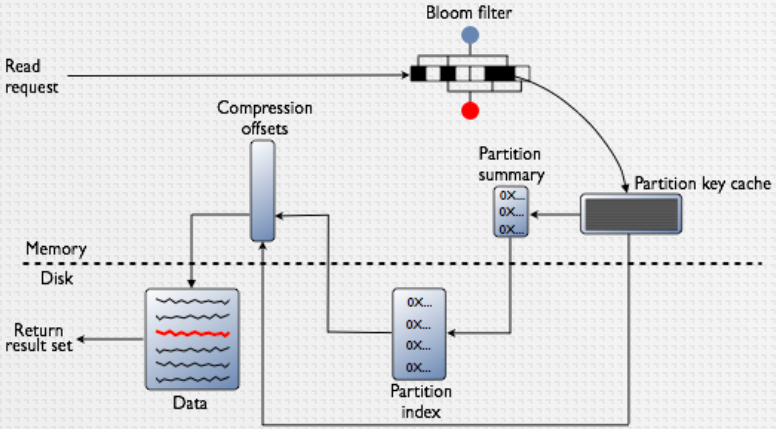


- **Consistency level** determines how many nodes will respond back with the success acknowledgment.



Reading data

- Reading data is "**complex**" (though transparent) in Cassandra, but understanding it allows us to build better data models.



Summarizing:

1. Check the memtable
2. Check row cache, if enabled
3. Checks Bloom filter
4. Checks partition key cache, if enabled
5. Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not
6. If the partition summary is checked, then the partition index is accessed
7. Locates the data on disk using the compression offset map
8. Fetches the data from the SSTable on disk

***TIP**: We need to balance between having everthing together in a single partition and spread the data across the cluster*

Cassandra Query Language (CQL)

- CQL (Cassandra Query Language) offers a **model close to SQL** in the sense that data is put in *tables* containing *rows* of *columns*. For that reason, when used in this training, these terms (tables, rows and columns) have the same definition than they have in SQL.

- They do **not** refer to the concept of rows and columns found in the deprecated thrift API (and earlier version 1 and 2 of CQL).

```
SELECT *
FROM parts
WHERE part_type='alloy' AND part_name='hubcap'
AND part_num=1249 AND part_year IN ('2010', '2015');
```

CQL – Data Types (Native)

type	constants supported	description
ascii	string	ASCII character string
bigint	integer	64-bit signed long
blob	blob	Arbitrary bytes (no validation)
boolean	boolean	Either true or false
counter	integer	Counter column (64-bit signed value). See Counters for details
date	integer , string	A date (with no corresponding time value). See Working with dates below for details
decimal	integer , float	Variable-precision decimal
double	integer float	64-bit IEEE-754 floating point
duration	duration ,	A duration with nanosecond precision. See Working with durations below for details
float	integer , float	32-bit IEEE-754 floating point
inet	string	An IP address, either IPv4 (4 bytes long) or IPv6 (16 bytes long). Note that there is no inet constant, IP address should be input as strings
int	integer	32-bit signed int
smallint	integer	16-bit signed int
text	string	UTF8 encoded string
time	integer , string	A time (with no corresponding date value) with nanosecond precision. See Working with times below for details
timestamp	integer , string	A timestamp (date and time) with millisecond precision. See Working with timestamps below for details
timeuuid	uuid	Version 1 UUID, generally used as a “conflict-free” timestamp. Also see Timeuuid functions
tinyint	integer	8-bit signed int
uuid	uuid	A UUID (of any version)
varchar	string	UTF8 encoded string
varint	integer	Arbitrary-precision integer

CQL – Data Types (Collections)

Maps

```
CREATE TABLE users (
    id text PRIMARY KEY,
    name text,
    favs map<text, text> // A map of text keys, and text values
);

INSERT INTO users (id, name, favs)
           VALUES ('jsmith', 'John Smith', { 'fruit' : 'Apple', 'band' : 'Beatles' });

// Replace the existing map entirely.
UPDATE users SET favs = { 'fruit' : 'Banana' } WHERE id = 'jsmith';
```

Sets

```
CREATE TABLE images (
    name text PRIMARY KEY,
    owner text,
    tags set<text> // A set of text values
);

INSERT INTO images (name, owner, tags)
            VALUES ('cat.jpg', 'jsmith', { 'pet', 'cute' });

// Replace the existing set entirely
UPDATE images SET tags = { 'kitten', 'cat', 'lol' } WHERE name = 'cat.jpg';
```

CQL – Data Types (Collections)

Lists

```
CREATE TABLE plays (
    id text PRIMARY KEY,
    game text,
    players int,
    scores list<int> // A list of integers
)

INSERT INTO plays (id, game, players, scores)
           VALUES ('123-afde', 'quake', 3, [17, 4, 2]);

// Replace the existing list entirely
UPDATE plays SET scores = [ 3, 9, 4] WHERE id = '123-afde';
```

User Defined Types

```
CREATE TYPE phone (
    country_code int,
    number text,
)

CREATE TYPE address (
    street text,
    city text,
    zip text,
    phones map<text, phone>
)

CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, frozen<address>>
)
```

CQL – Data Definition

Command	Description
CREATE KEYSPACE	A keyspace is created using a CREATE KEYSPACE statement, setting parameters such as the **replication strategy**, **replication factor**, etc.
USE	The USE statement allows to change the current keyspace (for the connection on which it is executed).
ALTER KEYSPACE	An ALTER KEYSPACE statement allows to modify the options of a keyspace. The same options as in CREATE KEYSPACE apply here.
DROP KEYSPACE	Results in the immediate, irreversible removal of that keyspace, including all the tables, UTD and functions in it, and all the data contained in those tables.
CREATE TABLE	Creates a new table (similar to SQL) and the following properties can be configured: **partitioning**, **clustering columns**, **clustering order**, etc.
ALTER TABLE	Altering an existing table uses the ALTER TABLE statement, allowing to add new columns (not the PK), remove columns and alter some options.
DROP TABLE	Dropping a table results in the immediate, irreversible removal of the table, including all data it contains.
TRUNCATE	Truncating a table permanently removes all existing data from the table, but without removing the table itself.

CQL – Data Definition (CREATE TABLE)

```
CREATE TABLE monkeySpecies (
    species text PRIMARY KEY,
    common_name text,
    population varint,
    average_size int
) WITH comment='Important biological records'
   AND read_repair_chance = 1.0;

CREATE TABLE timeline (
    userid uuid,
    posted_month int,
    posted_time uuid,
    body text,
    posted_by text,
    PRIMARY KEY (userid, posted_month, posted_time)
) WITH compaction = { 'class' : 'LeveledCompactionStrategy' };

CREATE TABLE loads (
    machine inet,
    cpu int,
    mtime timeuuid,
    load float,
    PRIMARY KEY ((machine, cpu), mtime)
) WITH CLUSTERING ORDER BY (mtime DESC);
```

- Primary keys (simple and composite)
- Data types (text, int, uuid, inet…)
- Compaction configuration
- Clustering order
- Comments

CQL – Data Manipulation

Command	Description
SELECT	The SELECT statements reads one or more columns for one or more rows in a table. It returns a result-set of the rows matching the request, where each row contains the values for the selection corresponding to the query. Additionally, functions including aggregation ones can be applied to the result.
INSERT	The INSERT statement writes one or more columns for a given row in a table. Note that since a row is identified by its PRIMARY KEY, at least the columns composing it must be specified.
UPDATE	The UPDATE statement writes one or more columns for a given row in a table. The where_clause is used to select the row to update and must include all columns composing the PRIMARY KEY. Non primary key columns are then set using the SET keyword.
DELETE	The DELETE statement deletes columns and rows. If column names are provided directly after the DELETE keyword, only those columns are deleted from the row indicated by the WHERE clause. Otherwise, whole rows are removed.
BATCH	Multiple INSERT, UPDATE and DELETE can be executed in a single statement by grouping them through a BATCH statement. This can be done to improve performance (avoid round trips, batch changes in a single partition, etc.)

CQL – Data Manipulation (SELECT & BATCH)

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
  AND time > '2011-02-03'
  AND time <= '2012-01-01'

SELECT COUNT (*) AS user_count FROM users;
```

```
BEGIN BATCH
   INSERT INTO users (userid, password, name) VALUES ('user2', 'ch@ngem3b', 'second user');
   UPDATE users SET password = 'ps22dhds' WHERE userid = 'user3';
   INSERT INTO users (userid, password) VALUES ('user4', 'ch@ngem3c');
   DELETE name FROM users WHERE userid = 'user1';
APPLY BATCH;
```

CQL – Others

- **Materialized views**
- **Security**: Users & Roles
- **Functions**: cast, token, now, time conversion, UDFs…
- **Secondary indexes**: Indexing a column not part of the PK
- **Aggregations**: count, max, min, sum and avg
- **JSON support**: inserts and selects
- **Triggers**: defined using Java and running outside de database

CQL – Limitations

There are following **limitations** in Cassandra query language (CQL):

- ~~CQL does not support aggregation queries like max, min, avg~~
- ~~CQL does not support group by, having queries~~
- CQL does not support joins
- CQL does not support OR queries
- CQL does not support wildcard queries
- CQL does not support Union, Intersection queries
- Table columns cannot be filtered without creating the index
- Greater than (>) and less than (<) query is only supported on clustering column

Cassandra query language is **not suitable for analytics purposes** because it has so many limitations.

Data Modeling

MAXIMIZE WRITES

In Cassandra, writes are very cheap. Maximize your writes for better read performance and data availability. There is a tradeoff between data write and data read.

A

DUPLICATE DATA

Data denormalization and data duplication are defacto of Cassandra. Disk space is not more expensive than memory, CPU processing and IOs operation.

C

SPREAD DATA

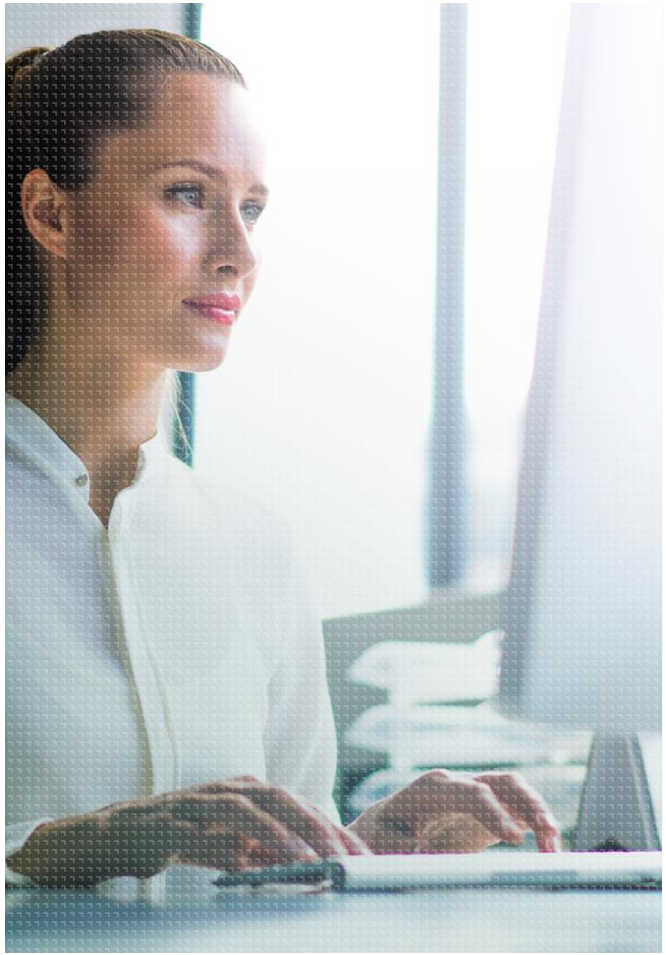
You want an equal amount of data on each node of Cassandra cluster. Data is spread to different nodes based on partition keys that is the first part of the primary key.

B

MINIMIZE PARTITIONS

Partition are a group of records with the same partition key. When the read query is issued, it collects data from different nodes from different partitions.

D



Exercise 2: Creating a Data Model

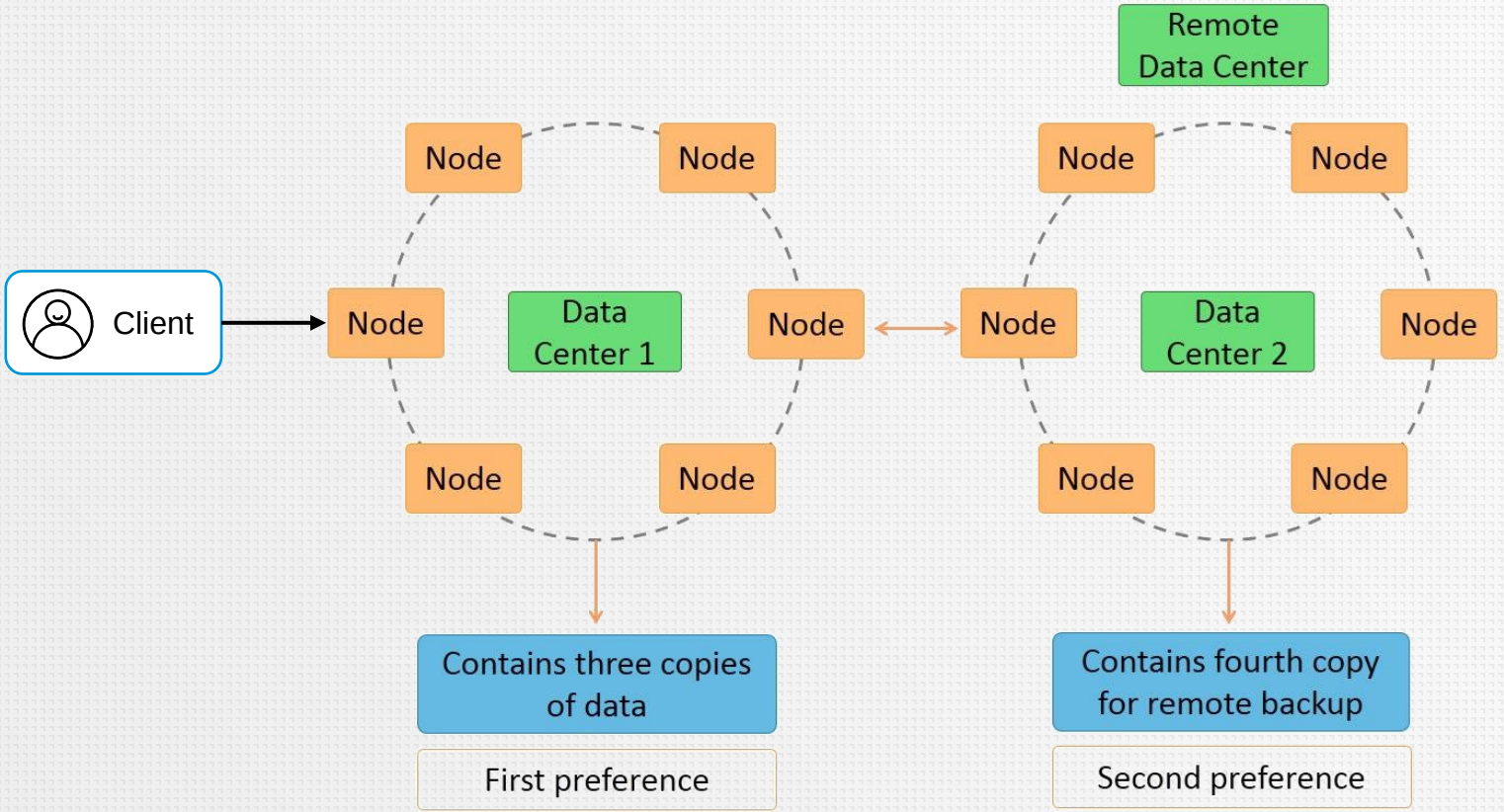
Things to take into account:

- Maximize the number of **writes**
- Maximize **data duplication**
- **Spread data** evenly around the cluster
- Minimize number of **partitions** read while querying data
- **Denormalize** data

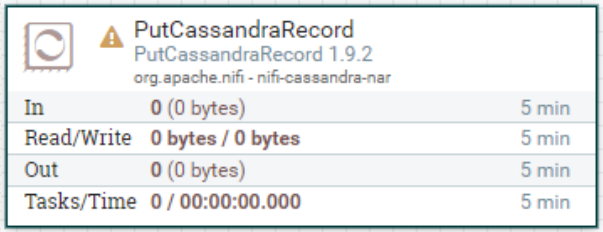
In this exercise you will create your first tables, in this case using **Zeppelin**.

Follow the steps in “Exercise 2” in the GIT repository.

Architecture

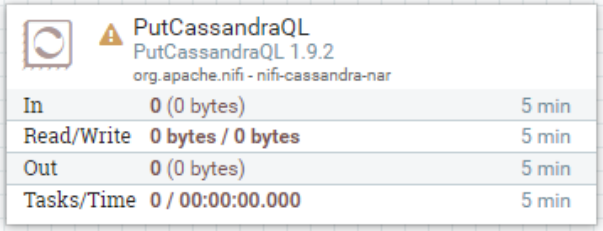


Using NiFi to read/write data



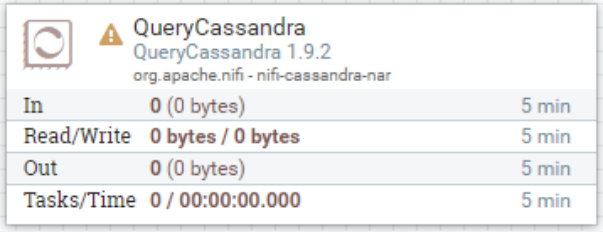
PutCassandraRecord

Reads the content of the incoming FlowFile as individual records using the configured 'Record Reader' and writes them to Cassandra.



PutCassandraQL

Executes the provided CQL statement on Cassandra. The FlowFile content is the CQL and can use '?' for parameters coming from the FlowFile attributes (cql.args.N.type and cql.args.N.value).



QueryCassandra

Executes the provided CQL select query on Cassandra. Result may be converted to Avro or JSON format. Streaming is used so arbitrarily large result sets are supported.

Reading data using the Java API

- There are **two different ways** to access Cassandra using Java:
  - **JDBC Driver** (several available)
  - DataStax **Java Driver** (v4.6)

```
<dependency>
  <groupId>com.datastax.oss</groupId>
  <artifactId>java-driver-core</artifactId>
  <version>${driver.version}</version>
</dependency>

<dependency>
  <groupId>com.datastax.oss</groupId>
  <artifactId>java-driver-query-builder</artifactId>
  <version>${driver.version}</version>
</dependency>

<dependency>
  <groupId>com.datastax.oss</groupId>
  <artifactId>java-driver-mapper-runtime</artifactId>
  <version>${driver.version}</version>
</dependency>
```

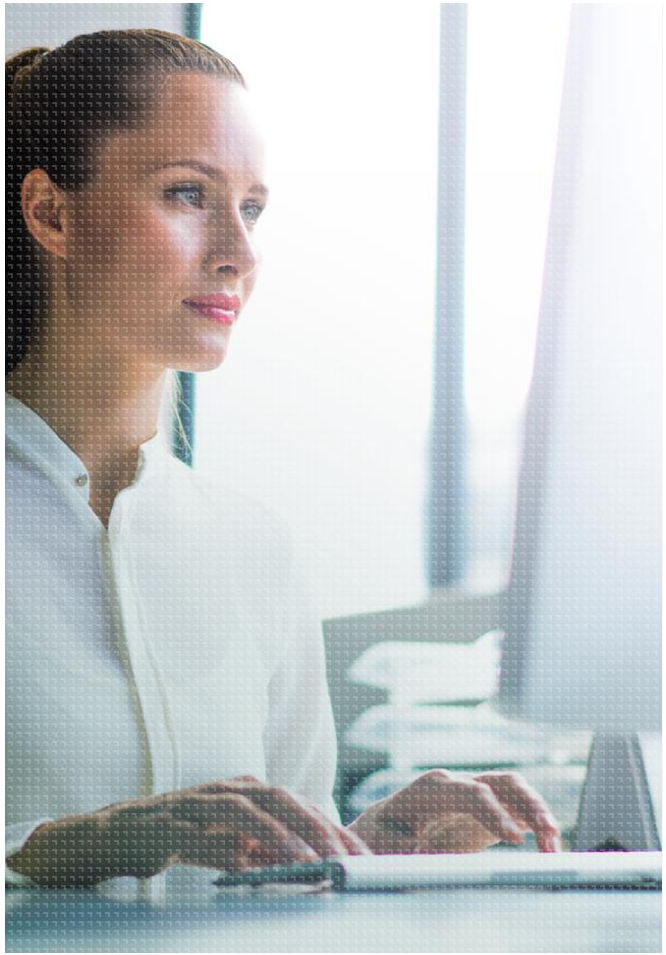
https://docs.datastax.com/en/developer/java-driver/4.6/

Reading data using the Java API

```java
import com.datastax.oss.driver.api.core.CqlSession;
import com.datastax.oss.driver.api.core.cql.*;

try (CqlSession session = CqlSession.builder().build()) {
  ResultSet rs = session.execute("select release_version from system.local");
  Row row = rs.one();
  System.out.println(row.getString("release_version"));
}
```

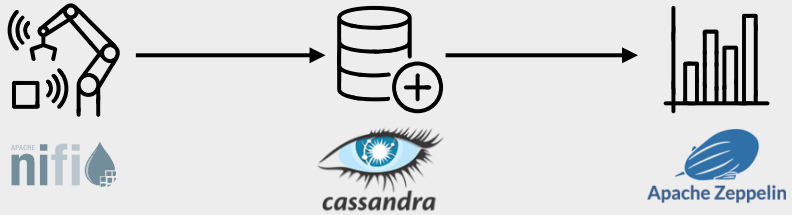
```java
import static com.datastax.oss.driver.api.querybuilder.QueryBuilder.*;

try (CqlSession session = CqlSession.builder().build()) {

  Select query = selectFrom("system", "local").column("release_version"); // SELECT release_version FROM system.local
  SimpleStatement statement = query.build();

  ResultSet rs = session.execute(statement);
  Row row = rs.one();
  System.out.println(row.getString("release_version"));
}
```

https://docs.datastax.com/en/developer/java-driver/4.6/



Exercise 3: Inserting and reading data

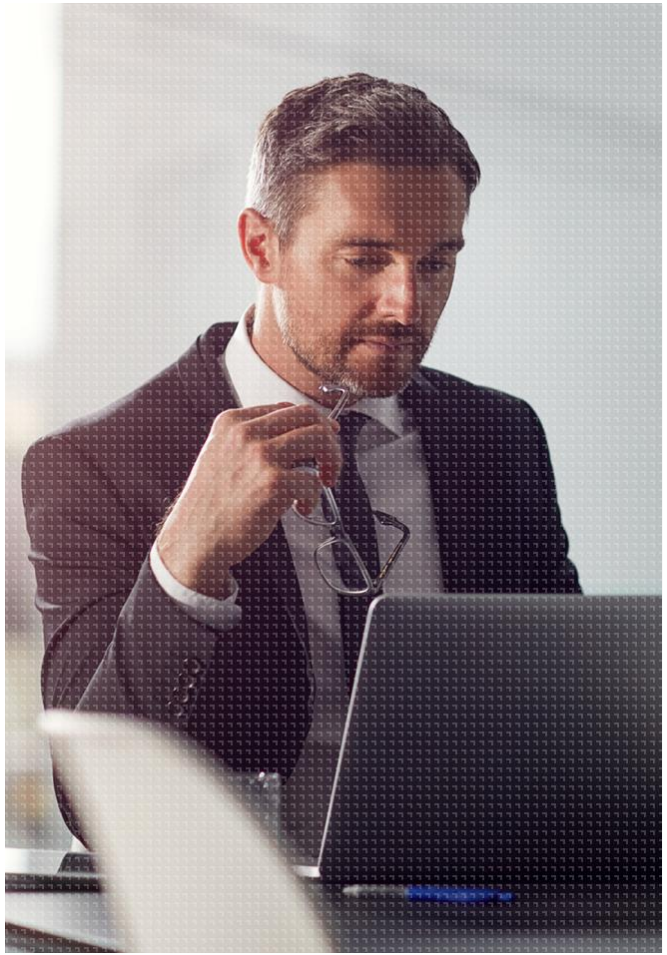
In this exercise we will simulate a simple **IoT project**, gathering data using **NiFi**, storing it in **Cassandra** and run some basic queries in **Zeppelin**.



Task we will perform:

- Create a new table to store sensor data
- Create a NiFi workflow to generate and insert data
- Create a Zeppelin notebook for dashboarding

Follow the steps in “Exercise 3” in the GIT repository.



Quiz Time!

- Name three examples of **NOSQL databases** and what type they are
- Describe two **use cases** when Cassandra makes sense as the storage system
- List two or three **features** of Cassandra
- Difference between **Replication Strategy** and **Replication Factor**.
- What is the difference between **partition key** and **clustering key**?
- What **collections** can be used in CQL?

SMACK

SMACK is the acronym for

Spark + **M**esos + **A**kka + **C**assandra + **K**afka

Processing & Analytics

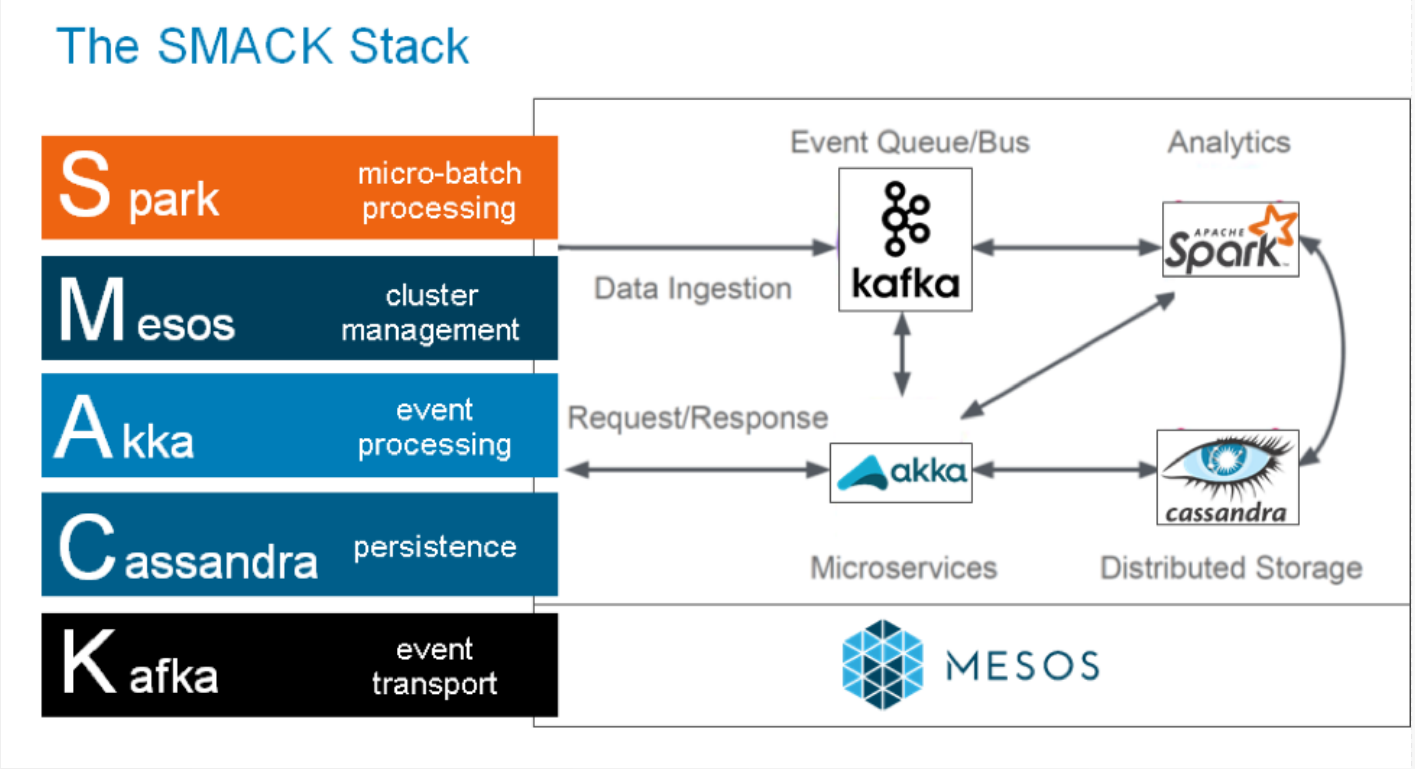
Resource Management

Reactive Programming

Distributed Database

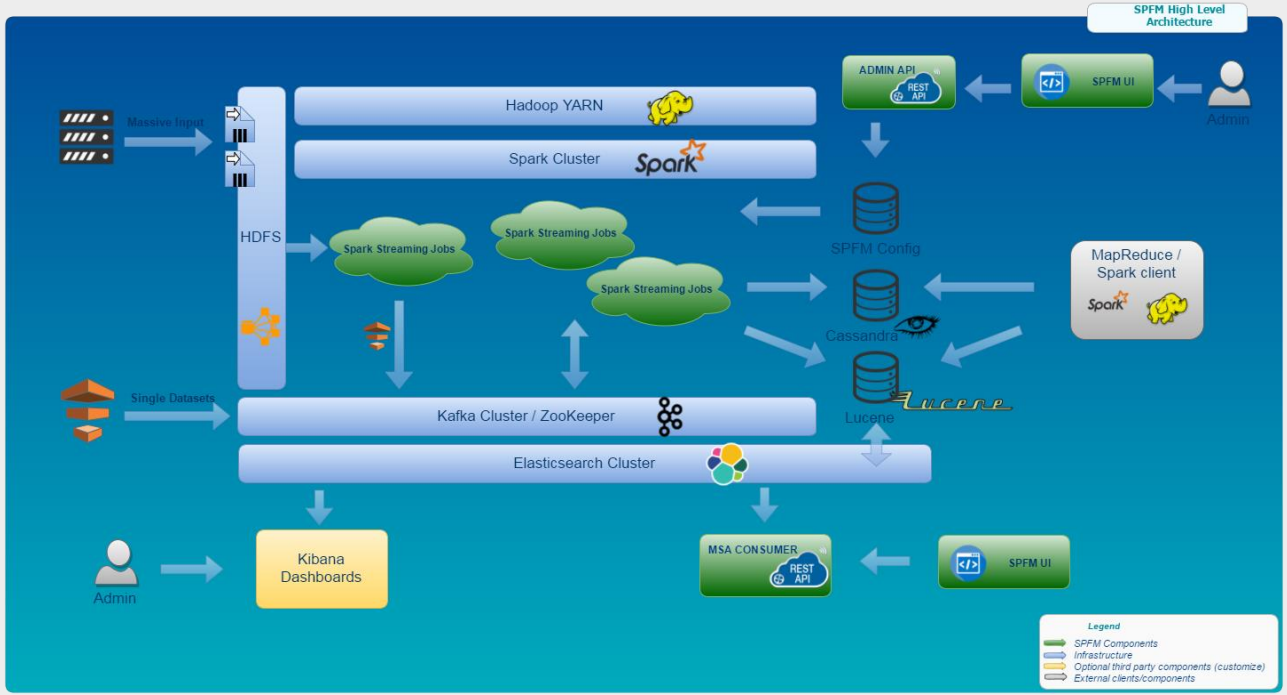
Message Broker

SMACK


The SMACK Stack
Spark
micro-batch processing
Mesos
cluster management
Akka
event processing
Cassandra
persistence
Kafka
event transport
Event Queue/Bus
Analytics
kafka
Spark
Data Ingestion
Request/Response
akka
cassandra
Microservices
Distributed Storage
MESOS

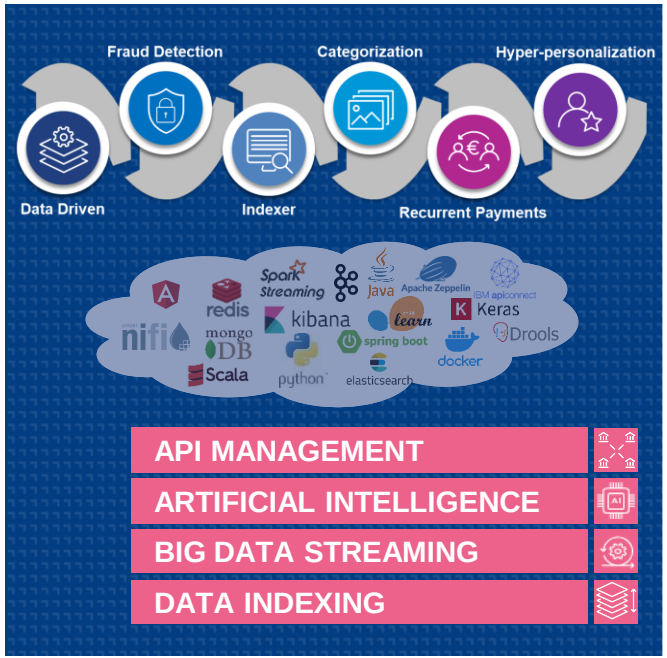
Success Stories

SDM: High Level Architecture



Digital eXperience real-time platform for tier 1 Mexican bank

Success story



THE CHALLENGE

Analyse data to offer an ultra-personalized experience to the user while providing valuable information to the bank

- Create a 360º view with all data available for each customer
- Categorize movements of all clients and expose this information to the user and internally
- Identify recurrent payments and increase conversion ratio from manual to direct debit payments.
- Index all movements for las 2 years to enable contextual search
- Contribute to fraud detection by alerting of unusual transactions and suspicious ……

THE ENGAGEMENT

Define a platform using cutting-edge technologies and AI tendencies capable of analysing huge volumes of data in real-time and that can be easily integrated with Fintech community

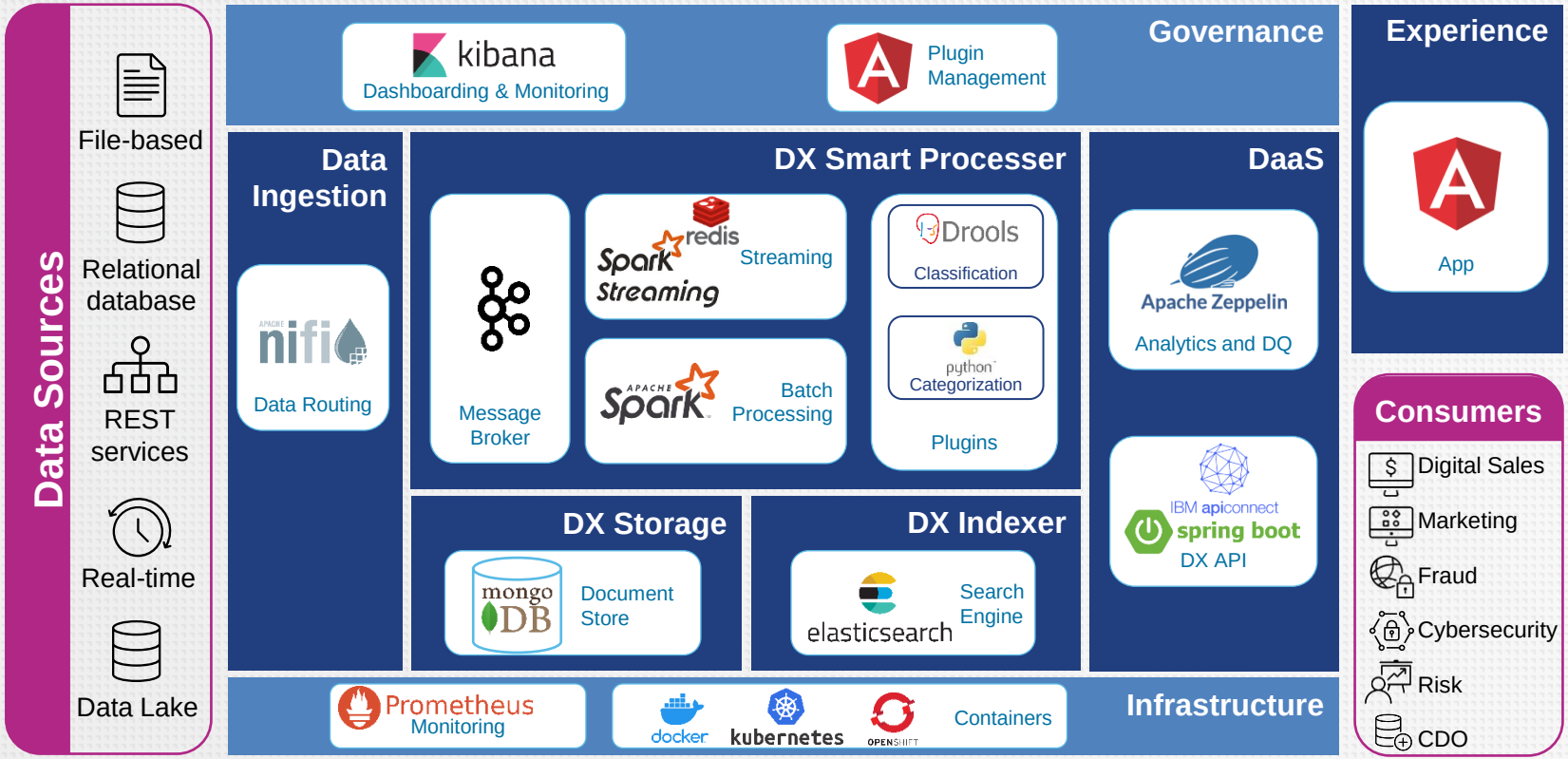
- Build a modular, flexible and easy-to-scale system that can absorb business demand from different sources (marketing, fraud, cybersecurity, digital banking...)
- Implement Machine Learning-based categorizator able categorize transactions in real-time and allow customers to personalize it creating their own categories.
- Detect potential fraud by using majority algorithms
- Contribute to data democratization by exposing all information generated through an open API
- Detect recurrent payments and offer personalized products for each customer
- Create a new type of transaction “flexible direct debit” to convert clients from manual payments
- Introduce elasticSearch for indexing all transactions and expose this info internally and externally

THE BENEFIT

Harnessing the power of digital experience information

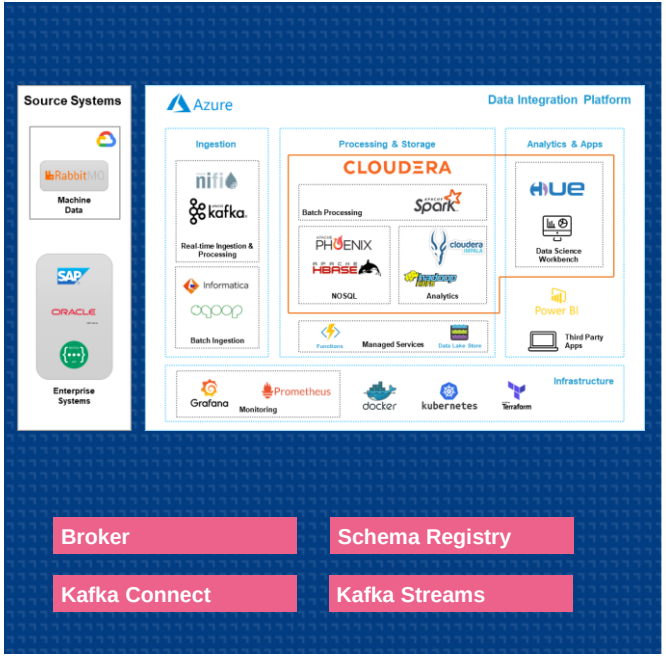
- Huge information increasement which allows better data-driven decisions
- Real-time analytics available for both customers and business
- User satisfaction and customer loyalty significantly increased
- Enabled a new way for offering hyper-personalized products directly to clients (Innovation)
- Fraud detection capabilities extended
- Cloud-ready platform, step forward on the strategic architecture path

Technical Architecture



IoT Data Lake in the Cloud

Success story



THE CHALLENGE

Run a complete data lake and analytics platform in the Cloud, for real-time IoT data

- Create a Data Lake able to handle and store data in big volumes, both in real-time and batch
- Analyze and apply machine learning models to enterprise data in an easy manner
- Real-time monitoring capabilities for all components and applications, at all levels
- Include state of the art support services for the data lake, such as Data Catalog or Monitoring
- All infrastructure should be cloud based, fully automated and defined as IaC (Infrastructure as Code)

THE ENGAGEMENT

Development of different work packages for the platform

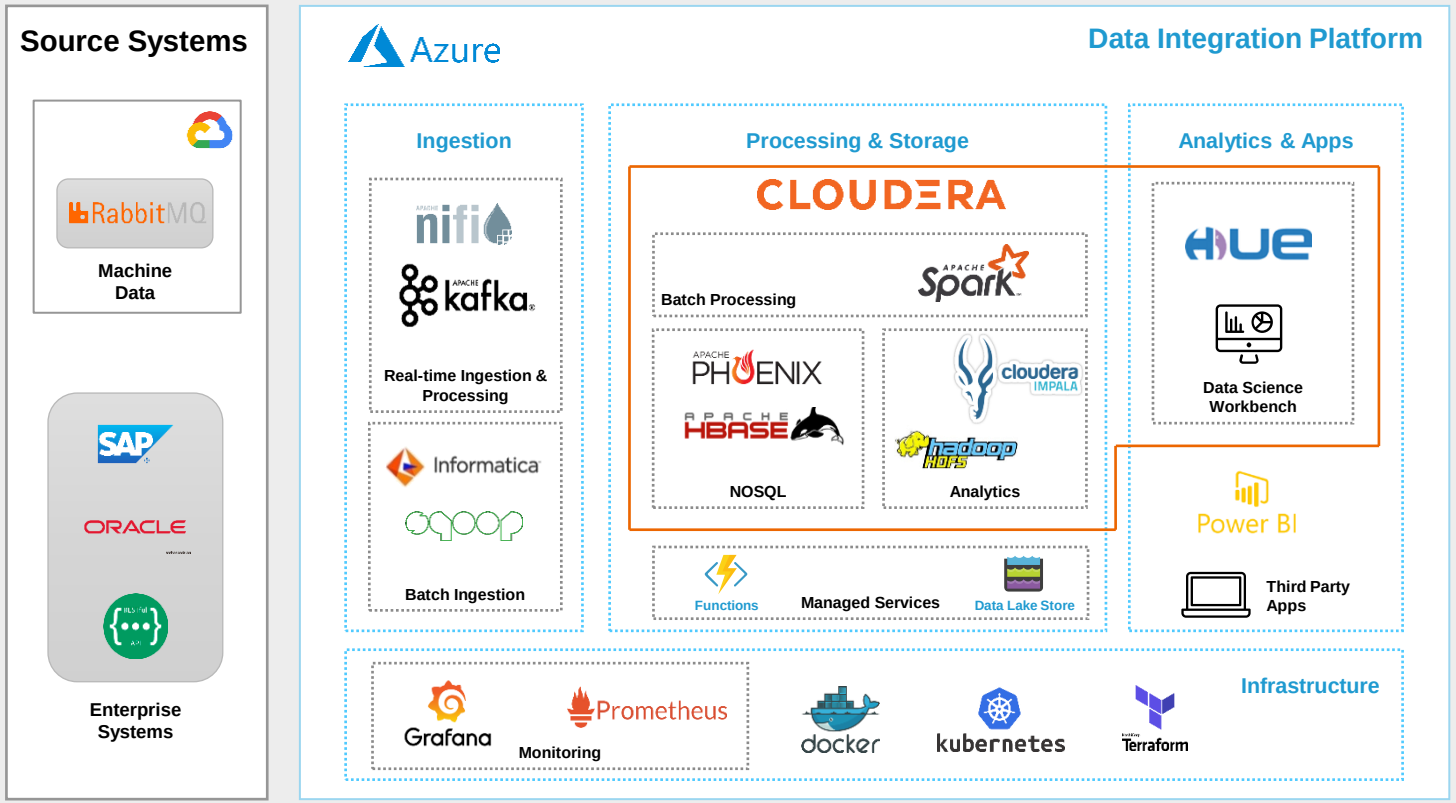
- Development of real-time data feeds, leveraging Kafka components and Azure managed services
- Design and deployment of a complete end-to-end monitoring framework based on Prometheus, Loki, Grafana and Jaeger
- Development of reporting capabilities using PowerBI against Big Data storage with Impala
- Definition of the SDLC (branching strategy, CI/CD pipelines, testing, etc.), based on Gitlab
- Infrastructure scripted and automated using Terraform and Ansible

THE BENEFIT

Full operational analytics platform with real-time data

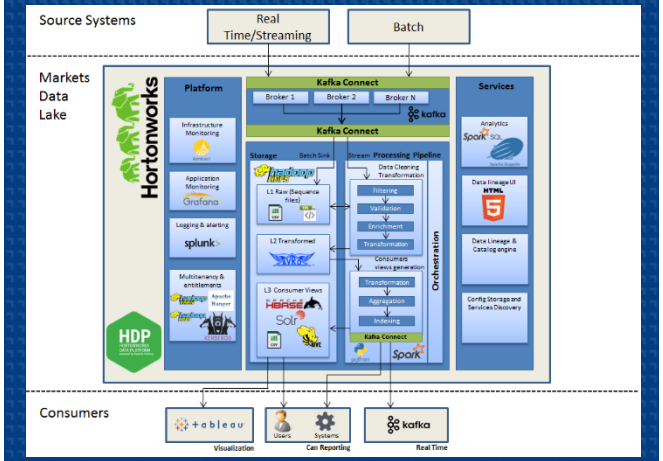
- Analytics on all corporate and machine data with always up to date data
- Scalable solutions, able to cope not only with existing volumes but ready for future ones
- Infrastructure flexibility by leveraging Cloud services and IaC

DIP Architecture



Market Data Lake

Success story



Broker

Schema Registry

Kafka Connect

THE CHALLENGE

Data Lake with batch and real time processing

- Design an implement an Strategic Data Lake using big data technologies covering all trading markets (FX, EQ, CM, MM, etc)
- Layered data storage having 3 levels: L1 for Raw data, L2 for transformed/enriched data, L3 for visualization/extraction data
- Multitenancy for different users and applications
- Authentication and authorization (entitlements) mechanism
- Ingestion of both real-time and batch data

THE ENGAGEMENT

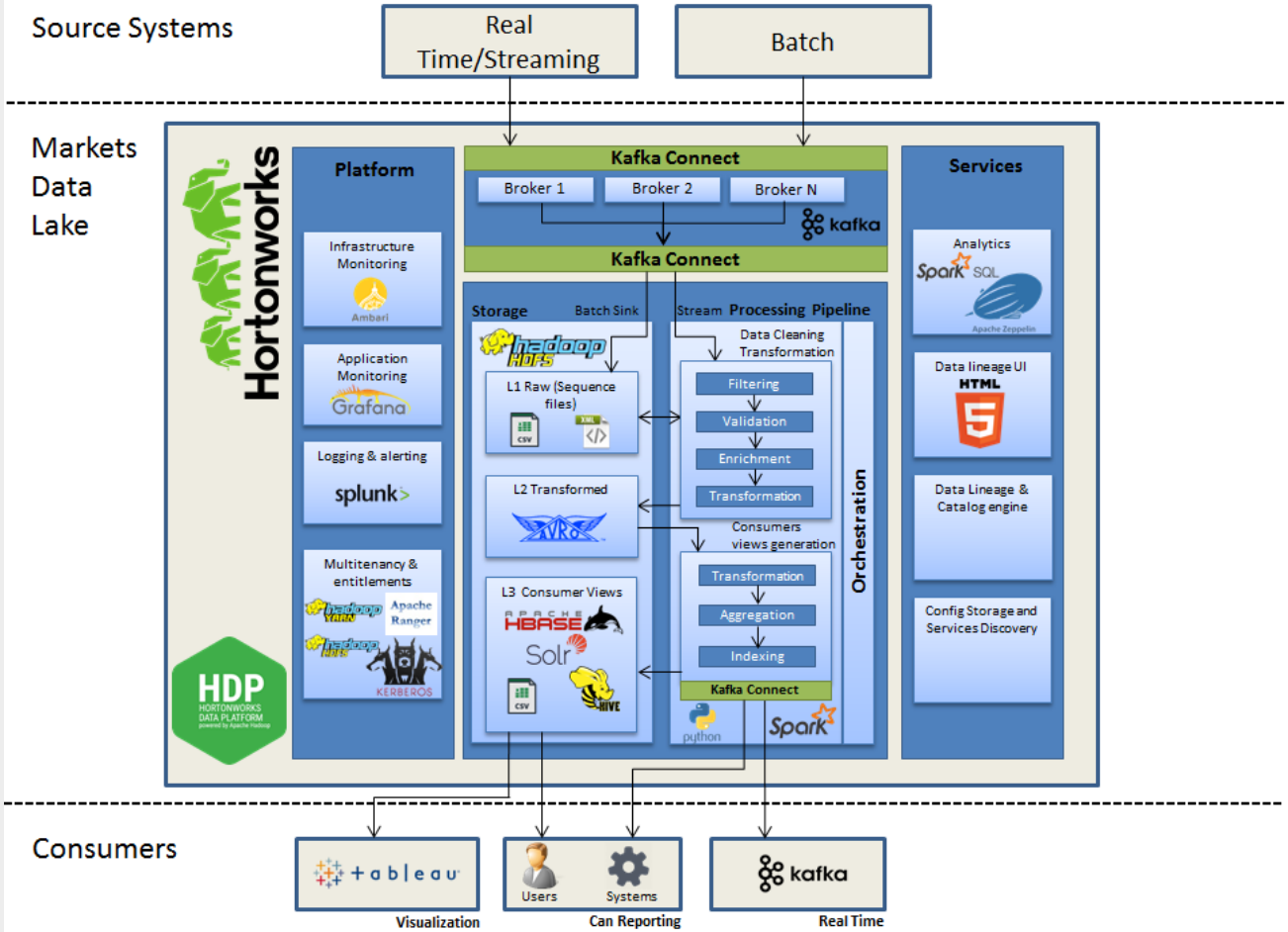
Design and implement a solution based on Kafka, Spark, Grafana, Splunk and HBase.

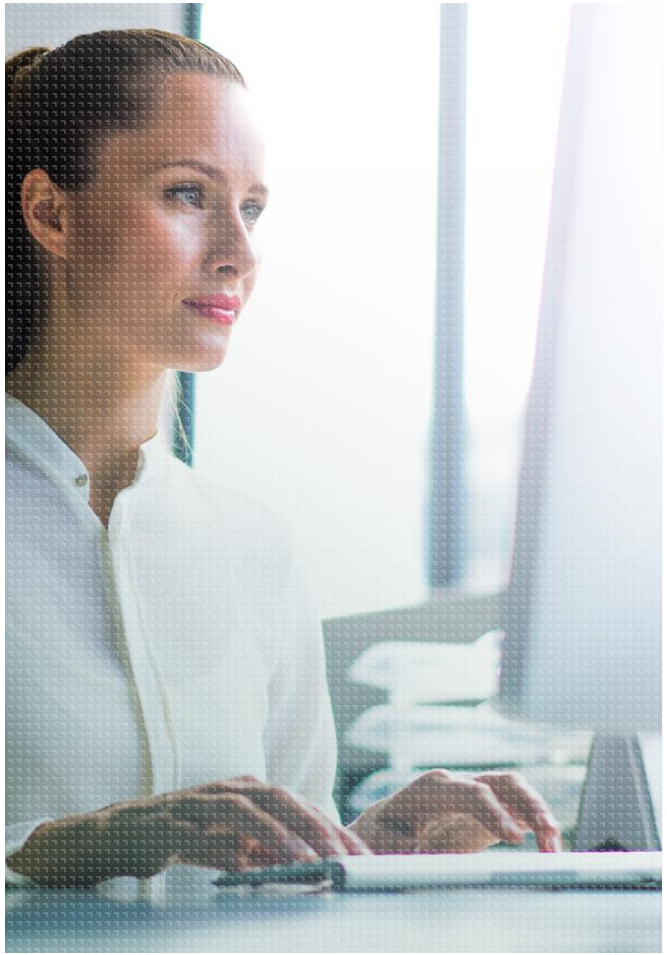
- Kafka is used as the message broker
- Kafka Connect to ingest data from any source (SFTP, JMS, etc)
- HDFS is used for the data storage in a 3-layers architecture
- Spark as the engine for the transformation pipelines
- HBase is the repository for the data catalog as well as to track data lineage
- Platform metrics collection and monitoring with Grafana
- Splunk for application logging and alerting

THE BENEFIT

Common repository horizontally scalable

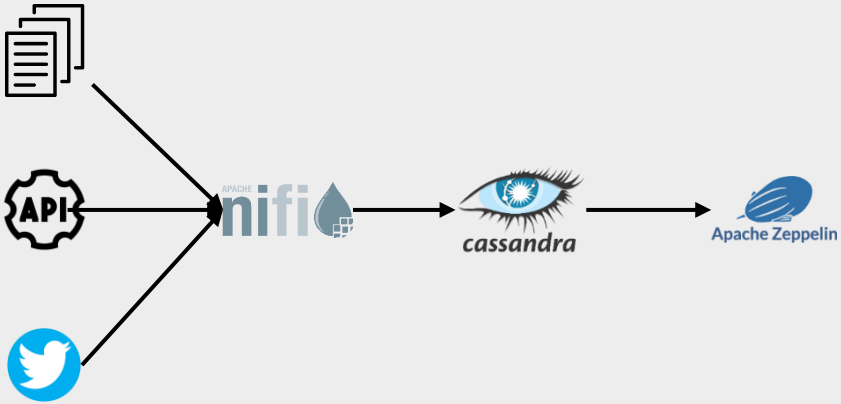
- Unique data repository where Quants teams (data scientists) perform market abuse insights
- Fully horizontally scalable solution
- Self-service ingestion, landing, and enrichment
- Extraction APIs
- Integration of Apache Zeppelin notebook for ad-hoc analysis


Source Systems
Real Time/Streaming
Batch
Markets Data Lake
Hortonworks
Platform
Infrastructure Monitoring
Ambari
Application Monitoring
Grafana
Logging & alerting
splunk>
Multitenancy & entitlements
Apache Ranger
KERBEROS
HDP
HORTONWORKS DATA PLATFORM
Kafka Connect
Broker 1
Broker 2
Broker N
kafka
Kafka Connect
Storage
Batch Sink
Stream Processing Pipeline
L1 Raw (Sequence files)
L2 Transformed
L3 Consumer Views
HBASE
Solr
HIVE
Data Cleaning Transformation
Filtering
Validation
Enrichment
Transformation
Consumers views generation
Transformation
Aggregation
Indexing
Kafka Connect
python
Spark
Orchestration
Services
Analytics
Spark SQL
Apache Zeppelin
Data lineage UI
HTML
Data Lineage & Catalog engine
Config Storage and Services Discovery
Consumers
tableau
Visualization
Users
Systems
Can Reporting
kafka
Real Time



Exercise 4: Inserting and analyzing data in real-time

Ingest data (ideally in real-time) and analyze using Zeppelin.



Optionally, if you prefer, you could read the data from Java, simulating a REST service.

Use “Typical use cases” slide (24) as a reference.

GFT

Shaping the future of digital business

Esteban Chiner

Senior Architect, specialized in Big Data & Blockchain