ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Introduction to Big Data

May, 2023

Dávid Szabó

epam

AGENDA

1. Motivation
2. Hadoop
3. Spark
4. Workshop

MOTIVATION

WHY BIG DATA?

NoSQL

- How to store data so that
  - … some queries can execute efficiently
- Limited ad-hoc queries

- Realtime queries
  - Execution time: milliseconds, seconds

Big Data

- How to store ANY DATA as is
- How to run ANY QUERY (analytics)

- Non-realtime queries
  - Execution time: minutes, hours

BIG DATA – DEFINITION

"deals with data sets that are too large or complex to be handled by traditional systems"

Big Data is not only about size

Volume

- Size of the data

Velocity

- Speed of data creation, storage, analysis and visualization

Variety

- Data comes in different formats (90% of data is unstructured)

EXAMPLE – SELF-DRIVING CARS

Volume

- Petabytes of sensor data

Velocity

- High-speed collection of sensor data
- High-speed processing

Variety

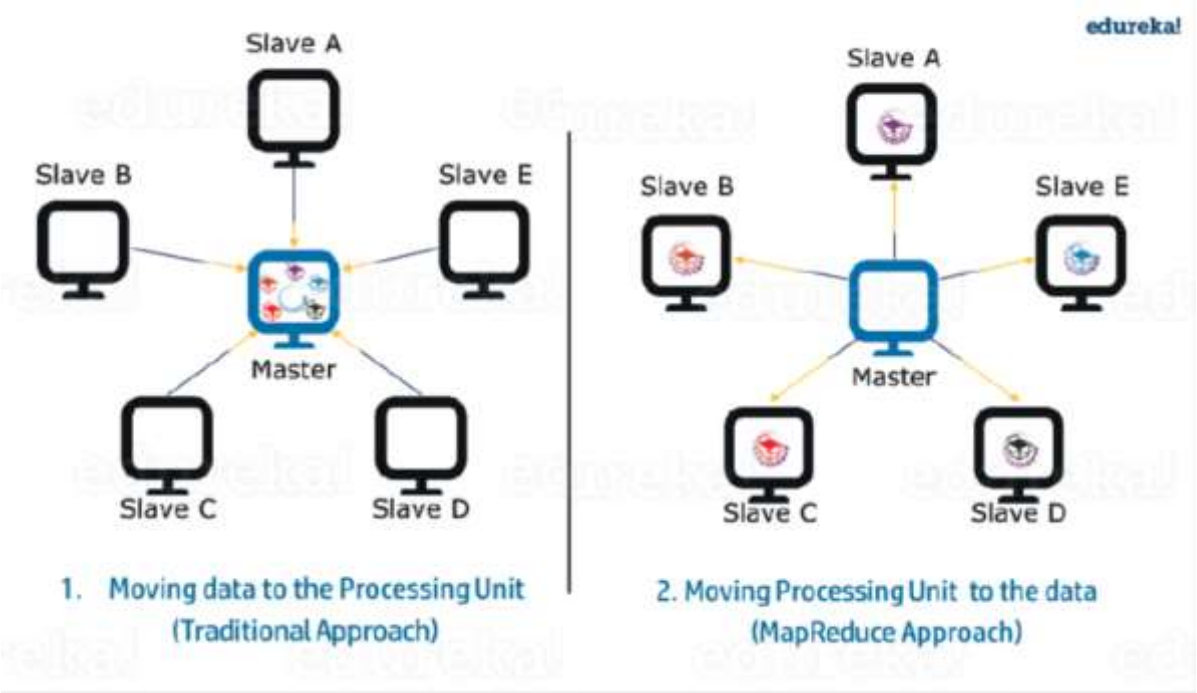
- Sensor data
- Image recognition
- Log files

Collect
store securely
and inexpensively

Analyze
find
connections

Understand
get
insights

IDEA – DATA LOCALITY



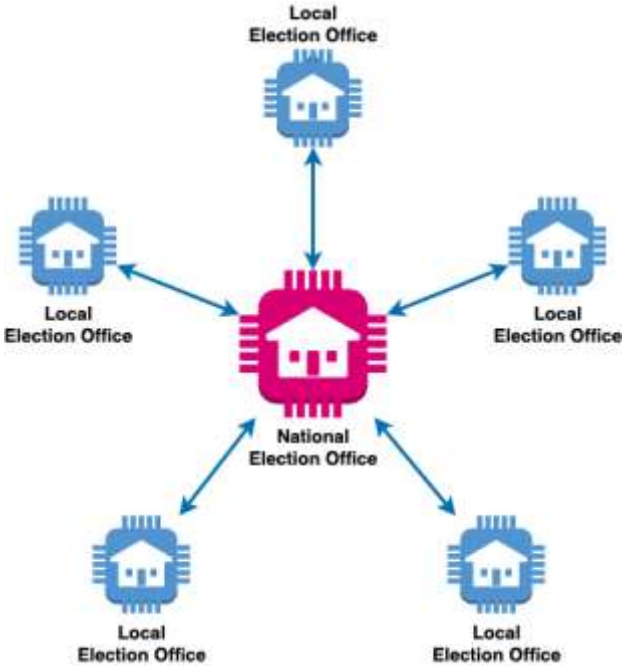
Idea

- Move the code instead of the data

Advantages

- Much less data to transfer
- Parallel processing

DATA LOCALITY EXAMPLE – NATIONAL ELECTIONS



- National office has the algorithm
- Local offices have the data

Traditional computing

- Move the data to the algorithm
- National office counts the votes

Distributed computing (map-reduce)

- Move the algorithm to the data
- Local offices process their data (map)
- National office summarizes the data (reduce)

HADOOP

WHAT IS HADOOP?

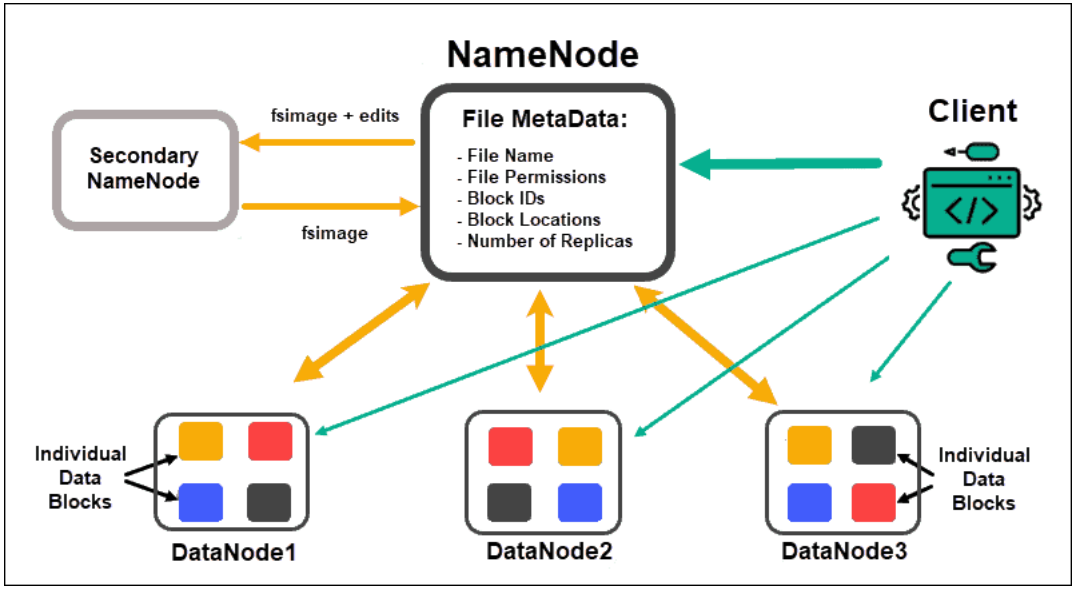
An **open-source** software platform

- … for **distributed storage**
- … and **distributed processing**
- … of very large datasets
- … on computer clusters built from **commodity hardware**

History:

- Based on GFS (Google File System) and MapReduce papers published by Google in 2003-2004
- Originally built by Yahoo! as an open-source web search engine
- Development driven by Doug Cutting & Tom White
- Initial release: 2006

HDFS – DISTRIBUTED FILE SYSTEM



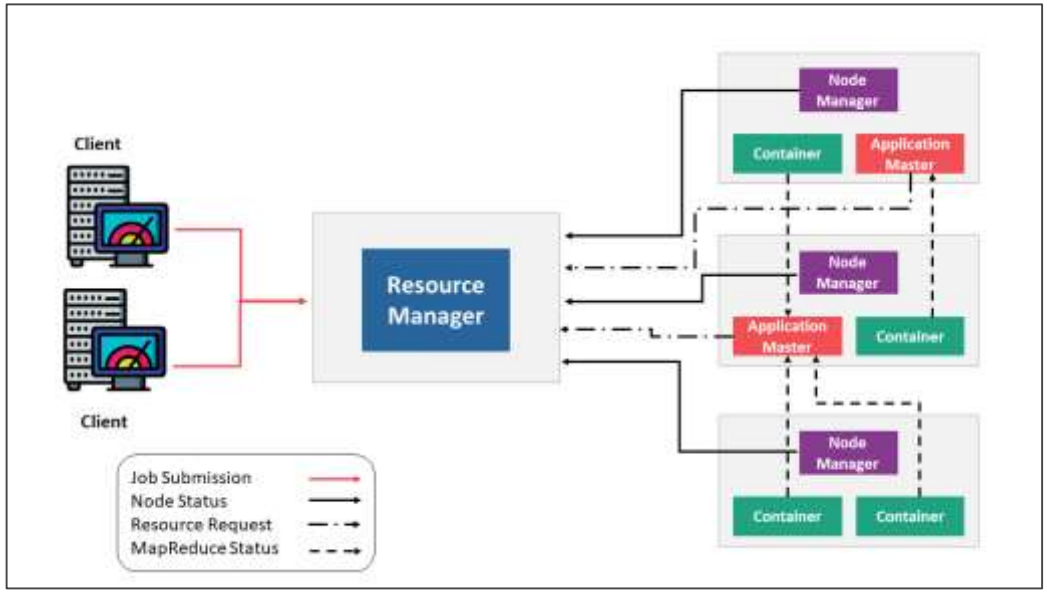
- Files are partitioned and replicated
- Optimized for large files
  - Breaks them into 128 Mb blocks
  - Many small files is an anti-pattern
- NameNode: table of contents
- DataNodes: contain the file chunks

```
hdfs dfs -mkdir /user/dir1
hdfs dfs -ls /user/dir1
hdfs dfs -put /home/localfile /user/dir1
hdfs dfs -cat /user/dir1/localfile
```

YARN – DISTRIBUTED COMPUTING

YARN: Yet **A**nother **R**esource **N**egotiator

- A framework for job scheduling and cluster resource management.



Resource Manager

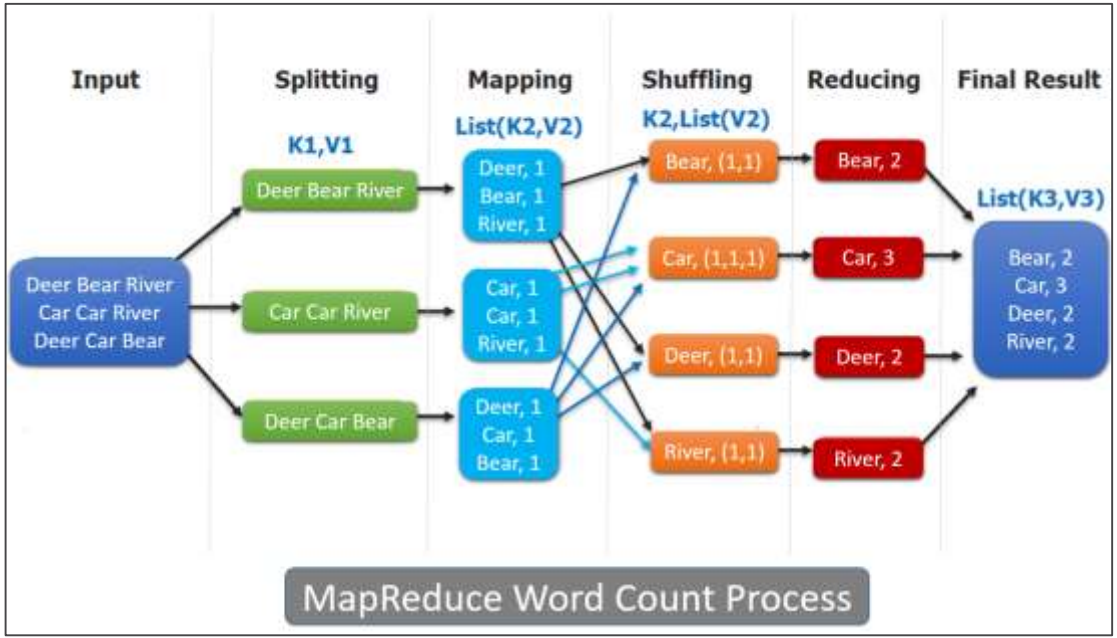
- Maintains a process overview
- Designates resources to applications
- Scheduling workloads

Node Manager

- Tracks resources on the node
- Acts as a slave for Resource Manager

MAPREDUCE – PROGRAMMING FRAMEWORK

MapReduce: A YARN-based system for parallel processing of large data sets.

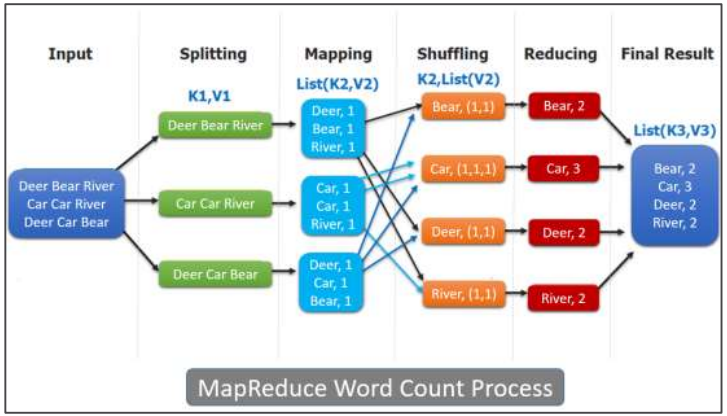



```java
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1));
        }
    }
}
```

```java
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,Context context)
            throws IOException,InterruptedException {
        int sum=0;
        for(IntWritable x: values) {
            sum += x.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

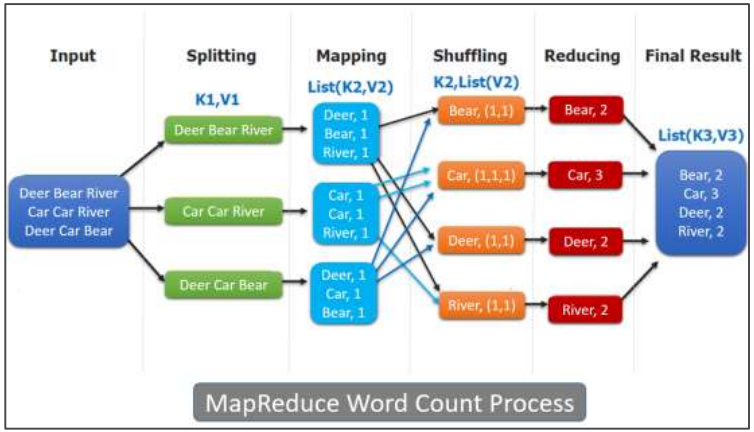
MAPREDUCE – MAP

```
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1));
        }
    }
}
```

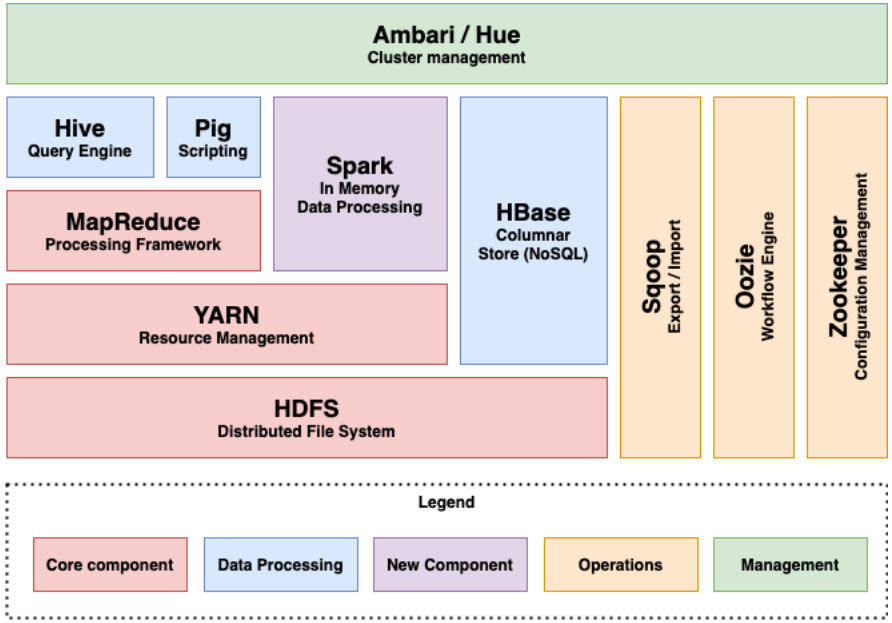


MAPREDUCE – REDUCE

```
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,Context context)
            throws IOException,InterruptedException {
        int sum=0;
        for(IntWritable x: values) {
            sum += x.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```



HADOOP COMPONENTS



Components are changing

- Some loosing relevance
  - (Pig, Oozie, Mahout, ...)
- Others are added
  - (Spark, Presto, ...)
- Many applications work together
  - (Cassandra, Airflow, ...)

Apache Hadoop Architecture Explained

HIVE – SQL INTERFACE

Data Warehouse software

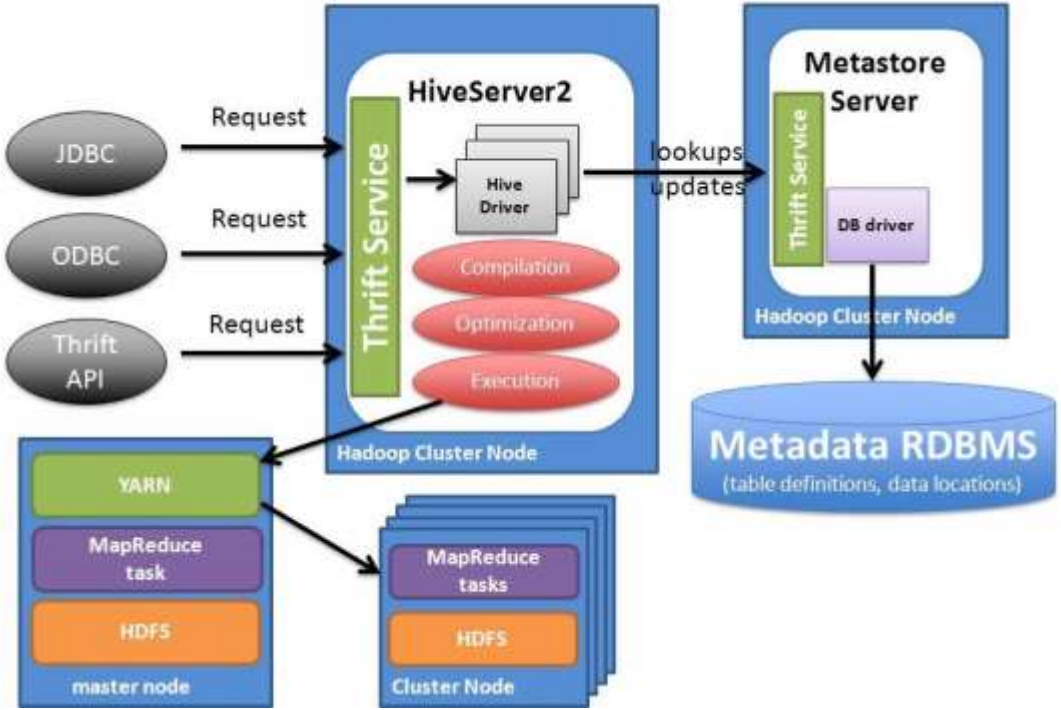
… for reading, writing, and managing large datasets

… residing in distributed storage

… using SQL

```
CREATE EXTERNAL TABLE User(
  first_name VARCHAR(64),
  last_name VARCHAR(64),
  company_name VARCHAR(64),
  address STRUCT<zip:INT, street:STRING>,
  country VARCHAR(64),
  city VARCHAR(32),
  state VARCHAR(32),
  post INT,
  phone_nos ARRAY<STRING>,
  mail MAP<STRING, STRING>,
  web_address VARCHAR(64)
)
COMMENT 'Temporary ORC table'
STORED AS ORCLOCATION '/user/hive/orc/user'
TBLPROPERTIES ("orc.compress"="SNAPPY");
```

HIVE – ARCHITECTURE



Hive Metastore

- Stores the schema info

Hive Server

- Translates SQL to MapReduce
- (or TEZ or Spark)

HIVE – PARTITIONS

Table definition

```
CREATE TABLE by_month (
  country string,
  iso_code string,
  day date,
  total_vaccinations float
)
PARTITIONED BY (
  year int, month int)
```

HDFS folders

```
by_month/year=2020
by_month/year=2020/month=12
by_month/year=2021
by_month/year=2021/month=1
by_month/year=2021/month=2
by_month/year=2021/month=3
```

HIVE – FEATURES

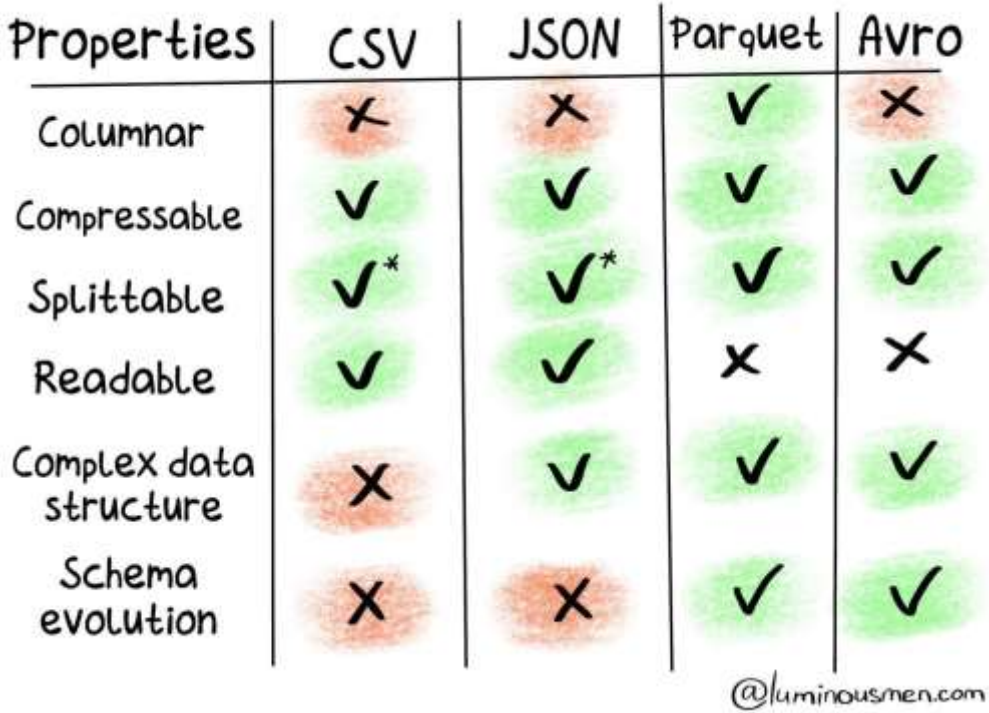
Supports

- A large subset of SQL
- Complex data types (arrays, structs, maps)
- Views
- Indexing
- Partitioning
- Bucketing
- User-defined functions
- Sampling
- Explain
- Optimization hints

Hive is

- NOT a relational database
- NOT designed for OLTP
  (on-line transaction processing)
- NOT real-time
  (not even with small amounts of data)

FILE FORMATS



Columnar format

- Data is stored by column, not row
- Read optimization
- Compression improvements

Schema evolution

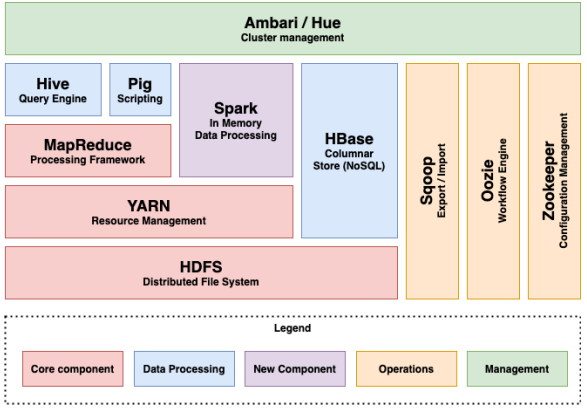
- Supports schema changes

Big Data File Formats

HADOOP DISTRIBUTIONS

Hadoop is a software package

- Components are open-source and free
- Distributions are tested packages with optional support



CLOUDERA

HORTONWORKS

MAPR

amazon EMR

Google Cloud DataProc

Microsoft Azure HDInsight

HADOOP – TAKE AWAY

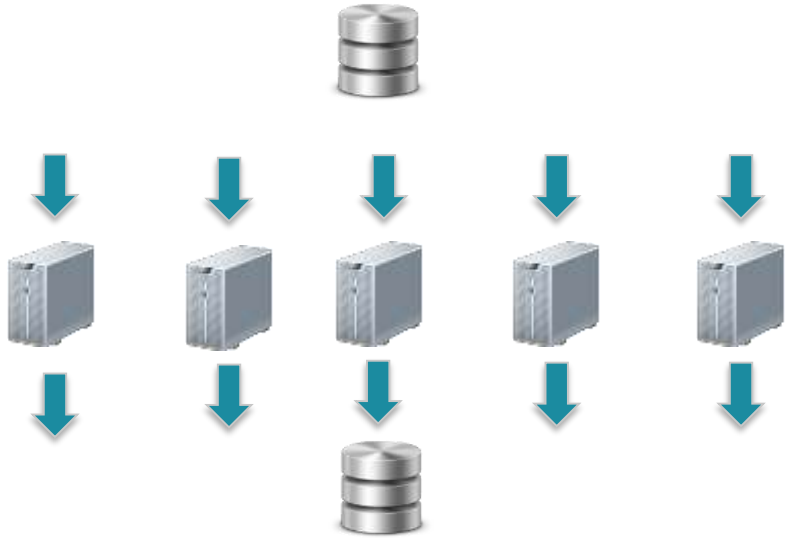
- Concept of MapReduce (Elections)
- Distributed Storage (HDFS)
- Distributed Computing (YARN + MapReduce)
- Hive: SQL on distributed files
- Hive metastore: Schema definition of files

SPARK

SPARK

Lightning-fast unified **analytics engine for large-scale data processing**

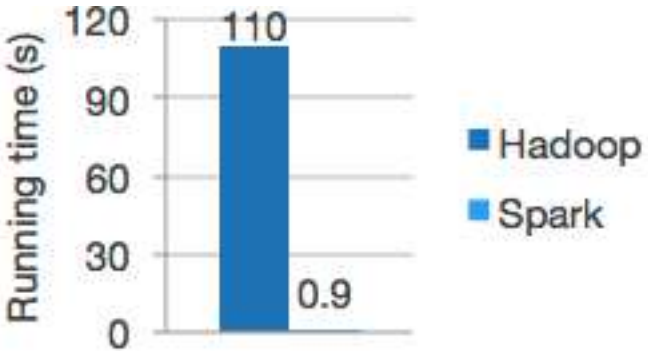
- Distributed computing
- Not a data store - but can integrate with them
- Not for OLTP



SPARK

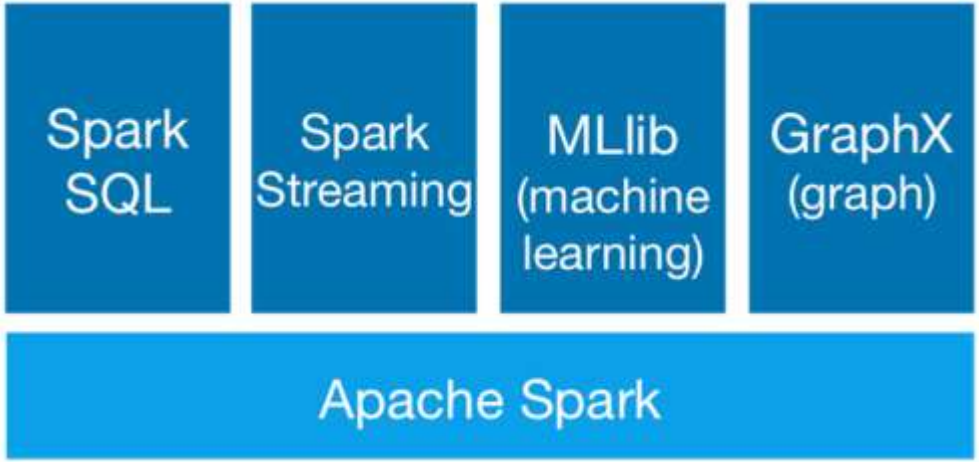
Lightning-fast unified analytics engine for large-scale data processing

- Distributed
- In-memory
  - On iterative processing
- High-level optimizations



SPARK

Lightning-fast **unified** analytics engine for large-scale data processing



SPARK

Easy to use

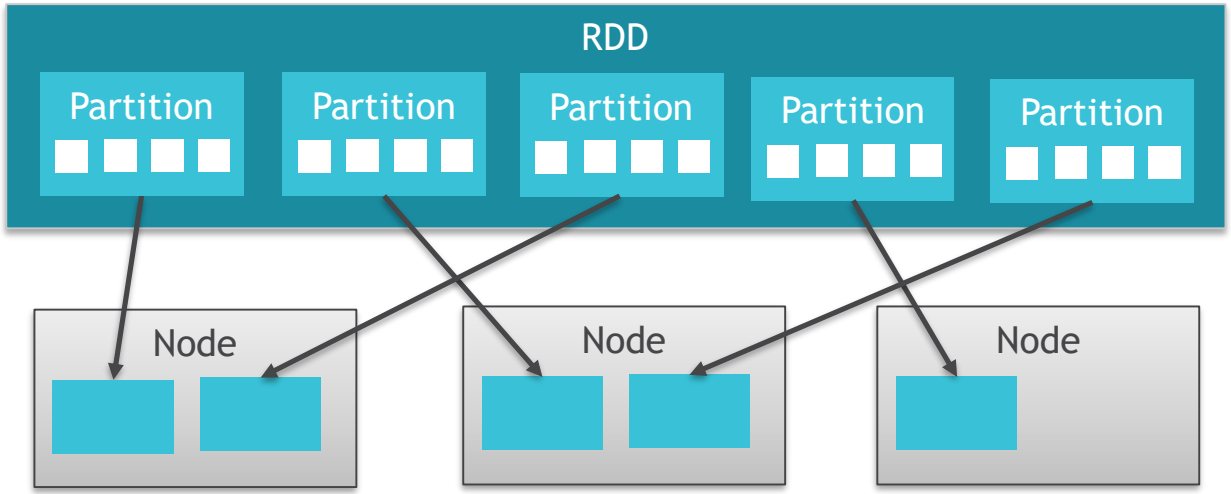
- High level operations
- Supports
  - several programming languages (Java, Scala, Python, R, SQL)
  - Many data stores (HDFS, Hive, HBase, Cassandra, ...)
- Interactive shell

```
df = spark.read.json("logs.json")
df.where("age > 21")
  .select("name.first").show()
```

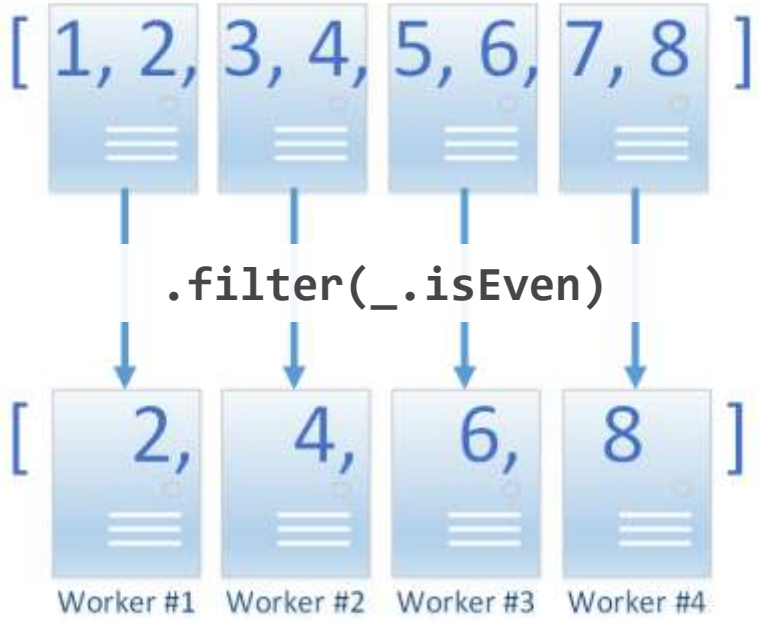
DATA COLLECTIONS & OPERATIONS

RDD (Resilient Distributed Dataset)

- Distributed collection of data
- Rebuild on failure
- Immutable
- Built via parallel transformations



RDD – OPERATIONS



DATA LOCALITY

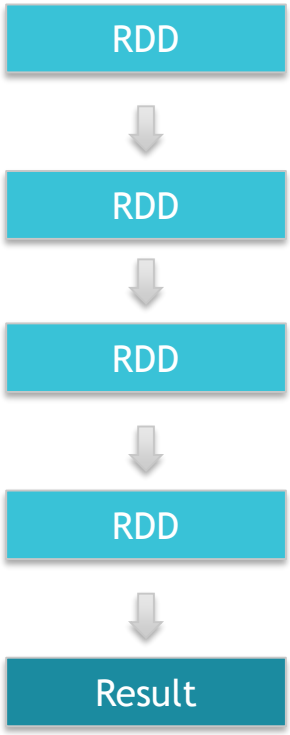
TRANSFORMATIONS & ACTIONS

Transformations

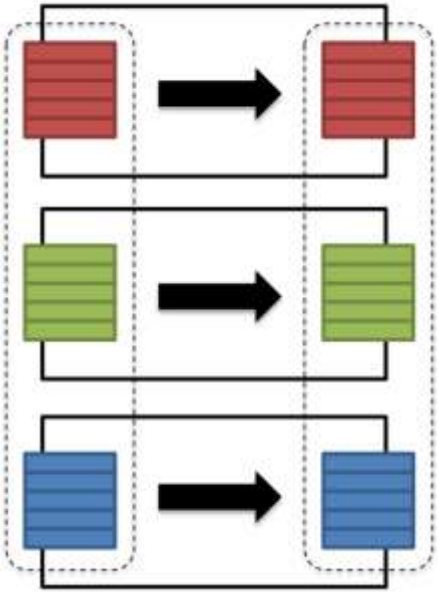
- Creates new data collection
- Lazy evaluation
- Example:
  - map
  - filter
  - group by
  - join

Actions

- Creates result
- Triggers execution
- Example
  - count
  - collect
  - write



TRANSFORMATIONS



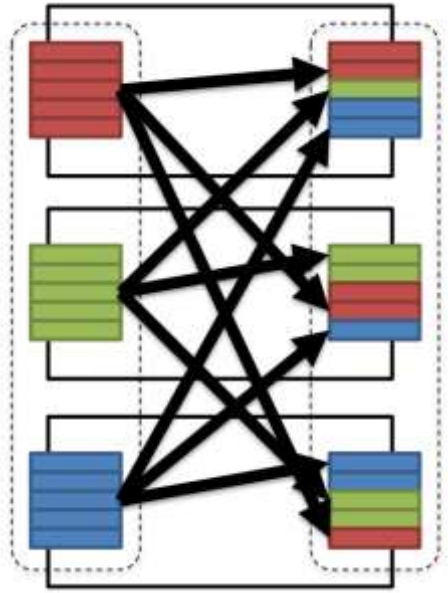
Narrow transformations

- Input and output stays in same partition
- Parallel execution
- Pipelining
- Example
    - Map
    - Filter

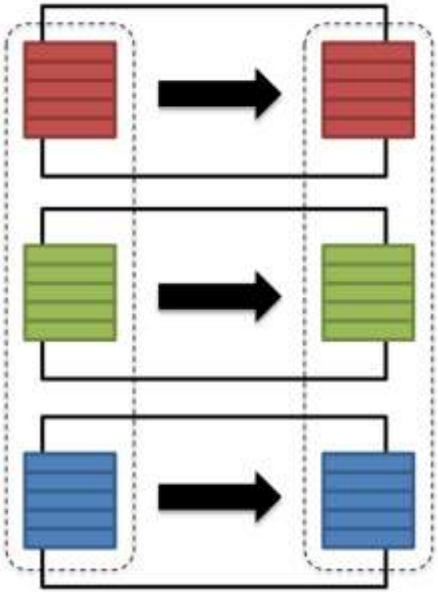
TRANSFORMATIONS

Wide transformations

- Input from other partitions required
- Needs expensive data shuffling
  - Serialization, disk & network I/O
- Examples
  - Group by
  - Repartition

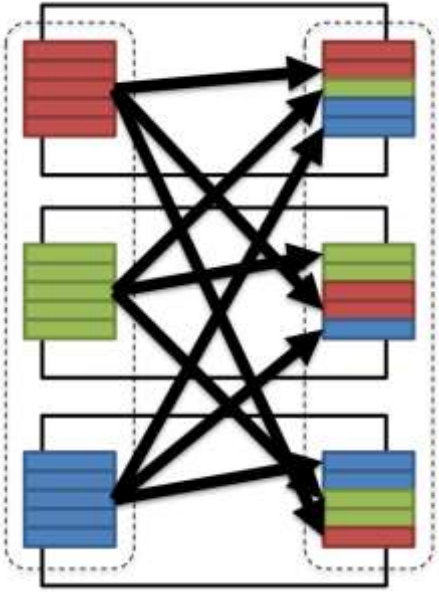


TRANSFORMATIONS



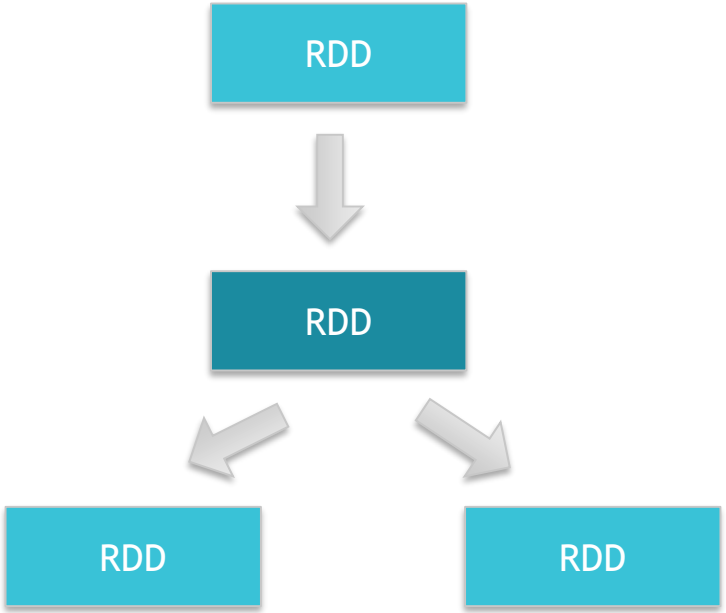
Joins

- Shuffle join
- Broadcast join



CACHE (PERSIST)

- Optimization technique
- Avoid recomputation on
  - Data reuse
  - Failure
- Key on iterative algorithms



APIS

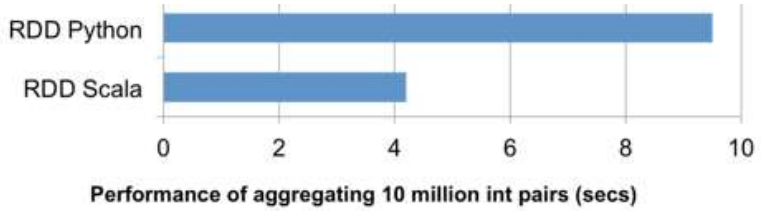
RDD

- Primary API in Spark 1.X
- Functional
- Records are raw objects
- Compile time type safety
- User controlled partitioning and storage

```
val lines = sc.textFile(@"hdfs://path/words.txt")
val processed = lines
    .filter(s => s.startsWith("a"))
    .map(s => s.reverse)
    .sortBy(s => s.length)
```

Limitations

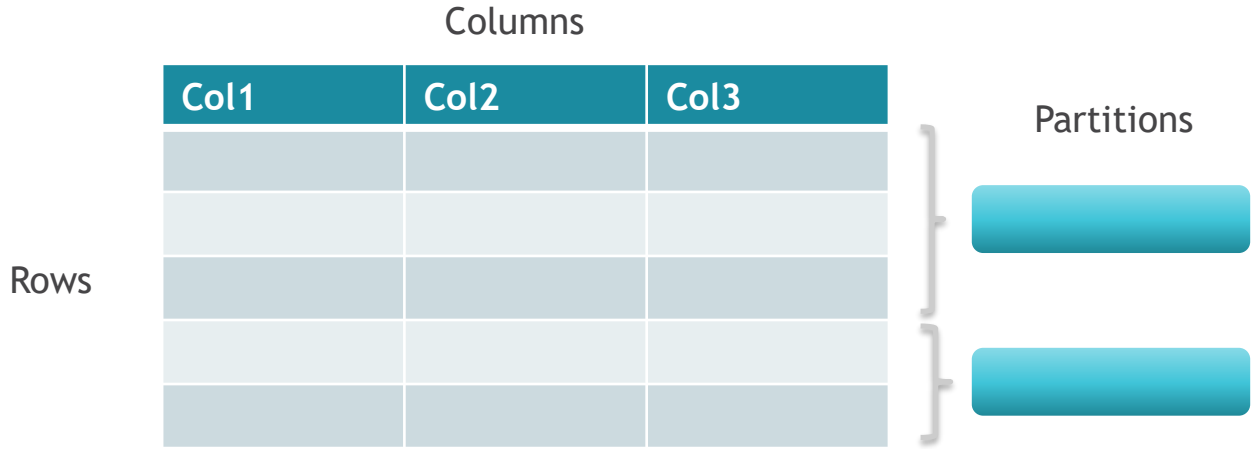
- Low level, non-expressive API
- No higher level optimization
- I/O handling is limited and can be cumbersome
- Low performance on non-JVM languages



DATAFRAMES

- Higher level abstraction on top of RDD
- Schema: named columns with types
- Conceptually like an RDMS table
- SQL-like declarative API

```
val persons = spark.read.json("test.json")
val names = lines
    .where("age > 21")
    .select("name.first")
names.show()
```



DATA TYPES

Primitive types

- ByteType, ShortType, IntegerType, LongType
- FloatType, DoubleType, DecimalType
- StringType, VarcharType, CharType
- BinaryType
- BooleanType
- TimestampType, DateType

Complex types

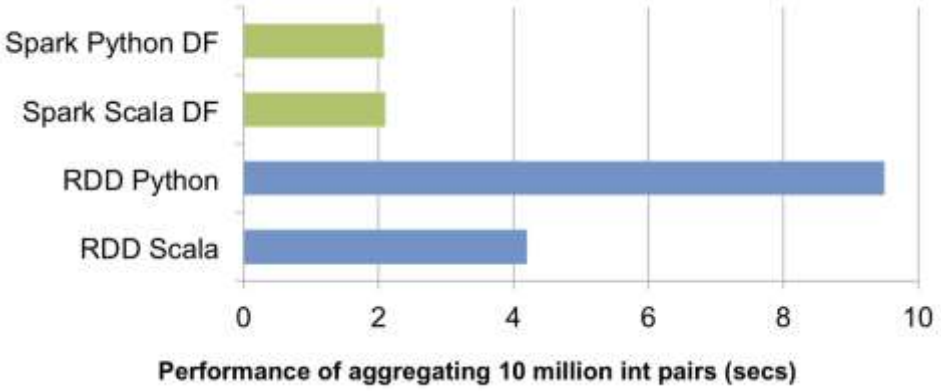
- ArrayType
- MapType
- StructType, StructField

own optimized types

DATAFRAMES

- Expressive high level operations and functions
- Language interoperability (no performance loss)
- Automatic optimizations
- Powerful I/O handling (Datasource API)
- No compile time type safety

```
val persons = spark.read
    .json("persons.json")
val names = persons
    .where("age > 21")
    .select("name.first")
names.show()
```



DATASETS

- Available in Scala and Java
- Strongly typed domain objects + encoders
- DataFrames are actually Datasets of type Row

```
case class Person(
    id: String,
    firstName: String,
    lastName: String,
    age: Int
)
```

```
val persons =
    spark.read.json("persons.json")
    .as[Person]
val names = persons
    .filter(_.age > 21)
    .select("firstName")
names.show()
```

- Both relational and functional operations
- High level optimizations
- Data source API
- Compile type safety
- Encoding has performance penalty

SPARK SQL

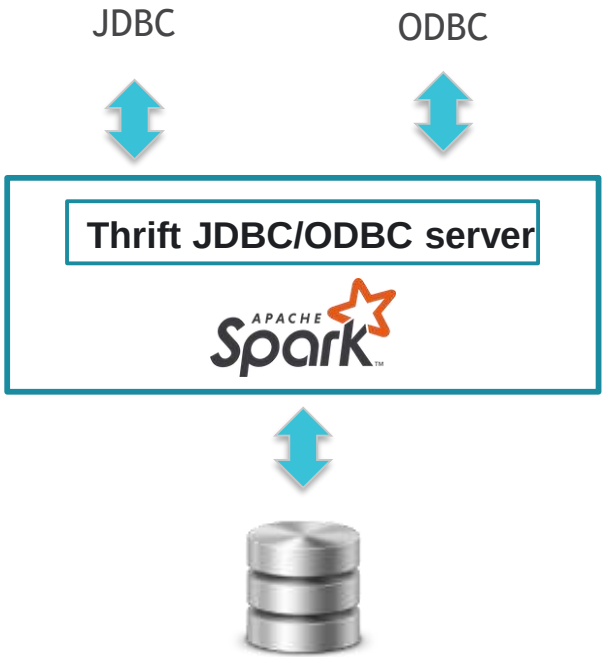
Programmatic SQL

```
persons.createOrReplaceTempView("people")
val names = spark.sql(
  "SELECT name.first FROM people WHERE age > 21"
)
names.show()
```

CLI

```
spark-sql> _
```

ThriftServer



SPARK SQL

Features

- Subset of ANSI SQL
- Managed & unmanaged tables
- Complex types
- Catalog
- Views
- Partitioning & bucketing
- User defined functions
- Cache tables
- Etc.

VS

SPARK – TAKE AWAY

- Distributed, fault-tolerant dataset
- Fast, mostly in-memory execution
- Four different API with pros and cons
- Structured APIs provide high level operations and optimizations

BREAK

WORKSHOP

ENVIRONMENT

Source available on

david-szabo-epam/spark-workshop

DATASOURCES

- Builder
- Default formats
  - Several file types
    (CSV, JSON, Partquet, Avro, etc.)
  - SQL (JDBC/ODBC)
  - Hive
- Available via Libraries:
  - Cassandra
  - MongoDB
  - Hbase
  - AWS Redshift
  - Many others
- Schema inference
- Smart sources
  - Query pushdown

```
spark.read
    .format("csv")
    .option("mode", "FAILFAST")
    .option("header", "true")
    .schema(someSchema)
    .load("/data/test-data2021.csv")
```

```
dataframe.write
    .format("csv")
    .option("mode", "OVERWRITE")
    .option("sep", "\t")
    .save("/data/test-output.tsv")
```

CHOOSING API

DataFrame

- Can be a good default choice
- Full compile time safety not needed
- For best general performance

SQL query

- When it is easier to express fully in SQL

Dataset

- Cannot be expressed in DataFrame manipulations
- Need or want compile time safety
- Performance penalty is acceptable

RDD

- Generally avoid
- Maintaining legacy applications
- Low level features (e.g. custom partitioning)

General considerations

- Personal, team and project preferences are factor
- You can change between APIs

WORKSHOP – TAKE AWAY

- Prefer structured APIs
- APIs are interoperable
- Datasource API
- Spark UI to monitor your jobs
- Execution plans can be checked via command and UI


Questions?

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR

Apache Cassandra
SQL in the NoSQL World

Attila Szűcs

March 2022


WHY?

RDBMS – RELATIONAL DATABASES

THEY ARE GREAT :)

- PROVEN
- RELIABLE
- GENERAL PURPOSE
- WELL KNOWN

THEY HAVE LIMITATIONS :(

- DESIGNED FOR SINGLE MACHINES
- RIGID SCHEMA
- LOCKING ISSUES
- EXPENSIVE

NOSQL MOVEMENT

Around 2006-2007 the two big companies presented their solutions

Amazon DynamoDB vs. Google BigTable

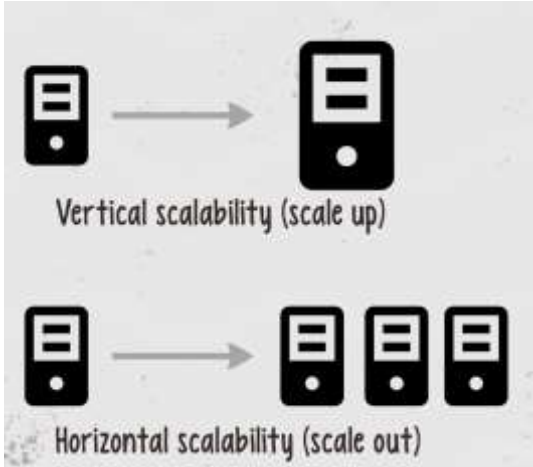
Cassandra inherited from both papers

WHAT IS NOSQL?

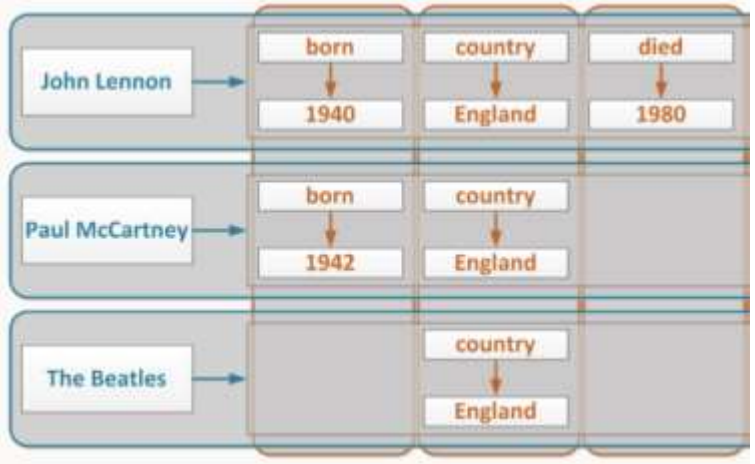
Terminology is wrong

- No SQL? – We do have SQL
- Not Only SQL? – Cassandra's only interface is SQL (since v3)
- Non Relational? – graph databases are all about relations

SCALABILITY IN MIND



FLEXIBLE SCHEMA



OVERVIEW

AGENDA

1. Data Model – Schema Flexibility
2. Architecture - Scalability
3. Deep Dive - CQL
4. Data Modeling

- Client Connectivity
- Operations and Monitoring
- Cassandra & Big Data

Source code available on

medvekoma/cassandra-workshop

FLEXIBLE SCHEMA – NOSQL LANDSCAPE

KEY-VALUE STORE
(value is an object)

redis riak

DOCUMENT STORE
(value is a hierarchy)

CouchDB mongoDB

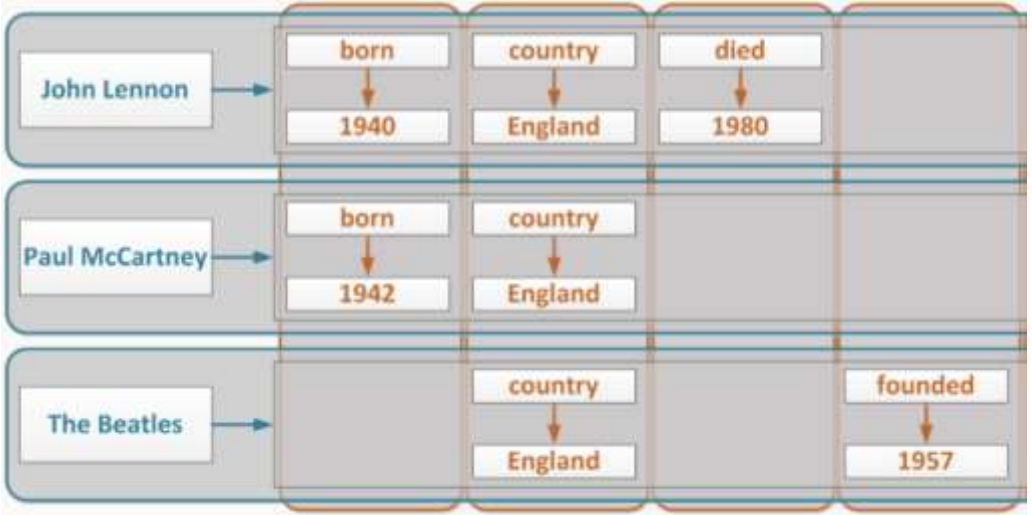
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

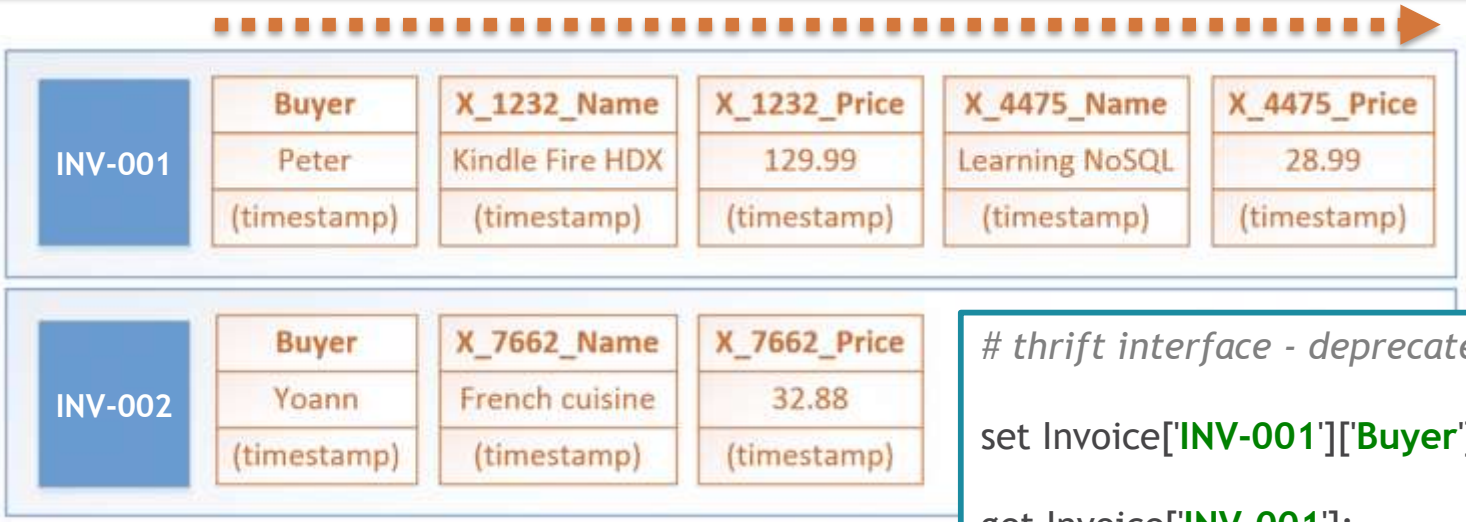
Embedded sub-document

COLUMN STORE
(value is a map/dictionary)

APACHE HBASE cassandra



COLUMN STORE



- Column names can vary by row (map of maps)
- Wide rows (up to ~2 billion columns)
- Column names are data (byte array)
- Columns are sorted by name

```
# thrift interface - deprecated

set Invoice['INV-001']['Buyer'] = 'Peter';

get Invoice['INV-001'];
get Invoice['INV-001']['X_1232_Name'];

list Invoice;

del Invoice['INV-001'];
del Invoice['INV-002']['X_7662_Price'];
```

CQL – CASSANDRA'S SQL INTERFACE

```
CREATE TABLE nobel_laureates
(
  year int,
  category text,
  laureateid int,
  name text,
  borncountrycode text,
  borncity text,
  PRIMARY KEY (year, laureateid)
);

CREATE INDEX ON nobel_laureates (borncountrycode);

SELECT * FROM nobel_laureates WHERE year = 2010;

UPDATE nobel_laureates
SET borncountrycode = 'HU'
WHERE year = 2017 AND laureateid = 9999;
```

Standard SQL interface

- Built on the top of internal data model (map of maps)
- Optimized for performance
- Primary interface to Cassandra

DEMO #1 – MEET CASSANDRA

Source code - Demo #1

- Environment & Toolset
- Inserting & Updating
- Collection types
- User defined types

CQL – INSERTS AND UPDATES

```
CREATE TABLE user
(
  id int,
  name text,
  PRIMARY KEY (id)
);

INSERT INTO user (id, name)
VALUES ( 1, 'Ada Lovelace');

-- Key violation ?
INSERT INTO user (id, name)
VALUES ( 1, 'Charles Babbage');

-- Nothing to update
UPDATE user
SET name = 'Linus Torvalds'
WHERE id = 2;
```

- No reads before writes
- Inserts and Updates are the same => UPSERT
- Only specified columns are written, not the whole row.

No, columns are “overwritten”.
Cassandra doesn’t verify key collisions.

```
-- ... equivalent to
INSERT INTO user (id, name)
VALUES (2, 'Linus Torvalds');
```

CQL – DATA TYPES

Primitive types

- int, tinyint, smallint, bigint
- decimal, float, double
- text (varchar), ascii
- timestamp, date, time
- uuid, timeuuid
- blob
- inet

Collection types

- list<T>
- set<T>
- map<TKey, Tvalue>

Each collection element is mapped to a distinct cell in the underlying data structure.

Frozen types are mapped to a single cell (serialized)

User defined types

```
CREATE TYPE article
(
  id int,
  name text,
  price decimal
);

CREATE TABLE invoice
(
  id int PRIMARY KEY,
  buyer text,
  items list<frozen<article>>
);
```

Source code - Demo #2

DEMO #2 – DATA TYPES

Source code - Demo #2

- Collection types
- User defined types

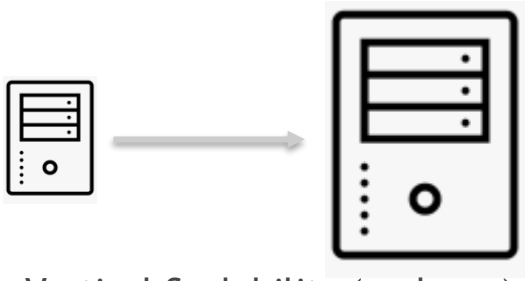
ARCHITECTURE

AGENDA – ARCHITECTURE

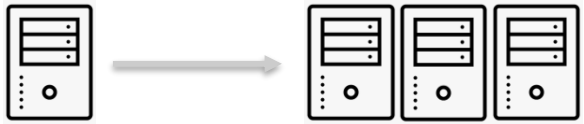
1. Data Model – Schema Flexibility
2. **Architecture - Scalability**
3. CQL – Cassandra Query Language
4. Data Modeling

- Partitioning
- Replication
- Consistency
- Durability & Performance

SCALABILITY – HORIZONTAL PARTITIONING



Vertical Scalability (scale up)
Limited



Horizontal Scalability (scale out)
Unbounded

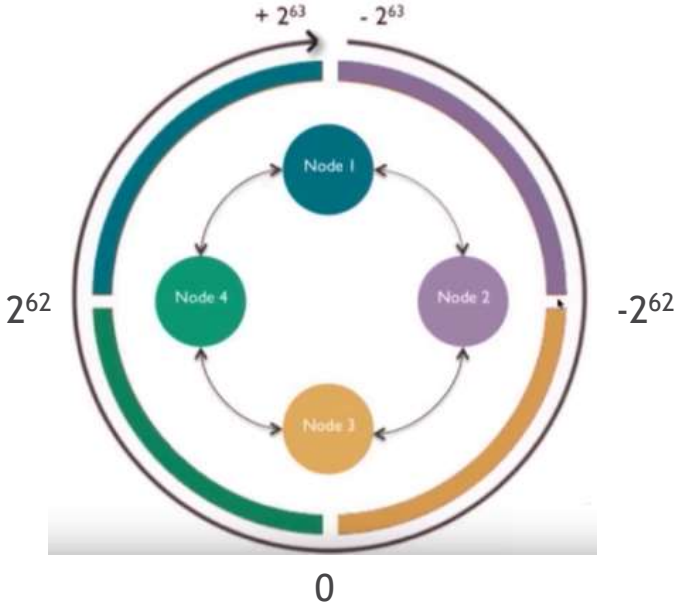
City	Country	Region
Lisbon	Portugal	Europe
London	United Kingdom	Europe
Seattle	United States	North America
Los Angeles	United States	North America

City	Country	Region
Lisbon	Portugal	Europe
London	United Kingdom	Europe

City	Country	Region
Seattle	United States	North America
Los Angeles	United States	North America

Horizontal Partitioning – based on a column

PARTITIONING – HASH BASED



Key	Hash value	Node
Ada Lovelace	~ 2^{30}	Node 4
Charles Babbage	~ -2^{28}	Node 3

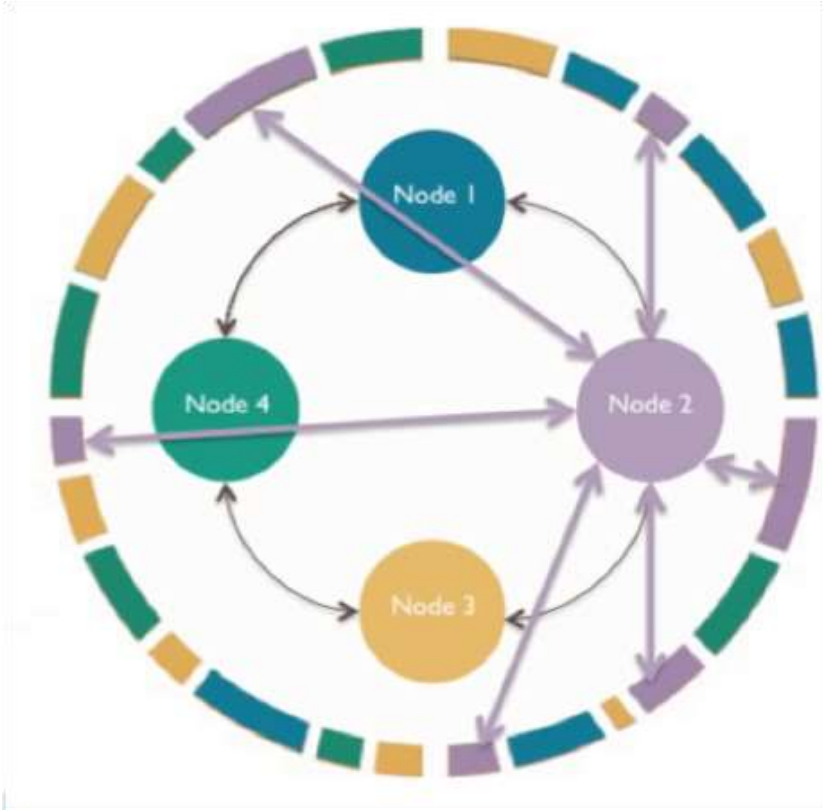
Configurable Partitioner

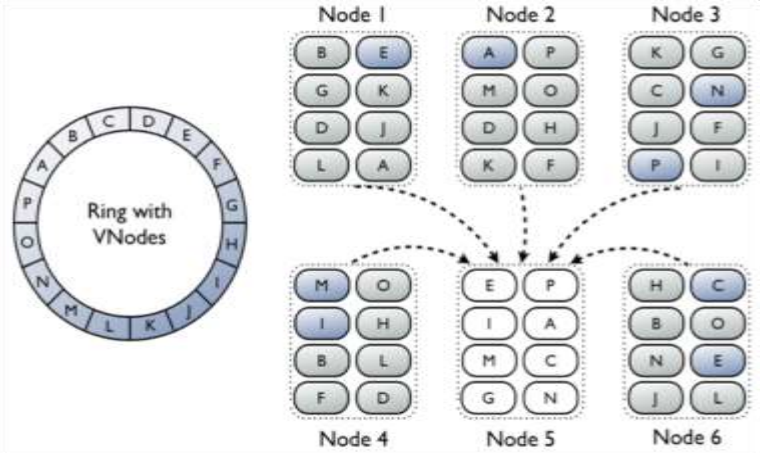
- Murmur3Partitioner
  default, uniform distribution
- ByteOrderedPartitoner
  enables range queries, but creates „hotspots”: not recommended
- RandomPartitioner
  old default

Hash algorithm

- *Fast (constant time)*
- *Consistent*

PARTITIONING – VIRTUAL NODES





- Multiple token ranges by node
- Default setting (256 tokens)
- Easier redistibution of data

REPLICATION

Why do we need replication?

- **Resilience**
  - the more nodes we have the more likely some will fail
- **Throughput**
  - More replicas can serve more requests

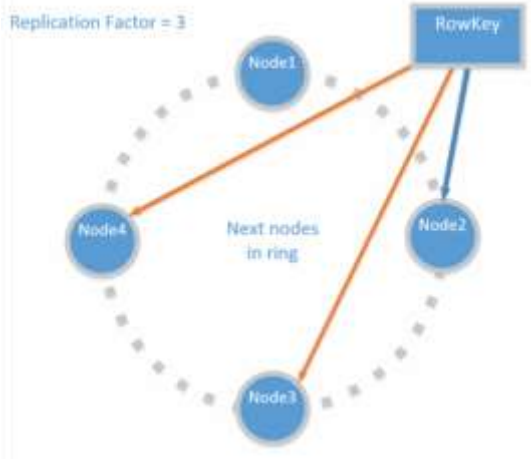
Replication Factor

- # of nodes that store the data
- 3 is a typical value in production

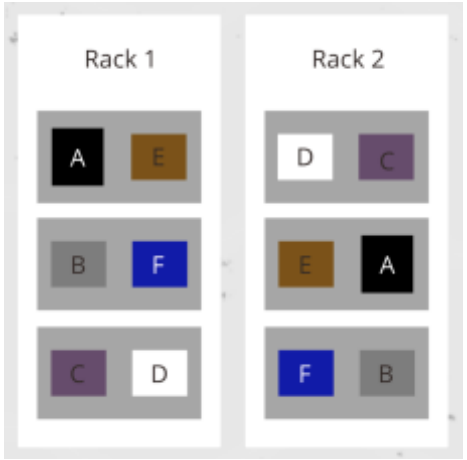
Replication Strategy

- SimpleStrategy
- NetworkTopologyStrategy (Rack & DC aware)

SIMPLE STRATEGY



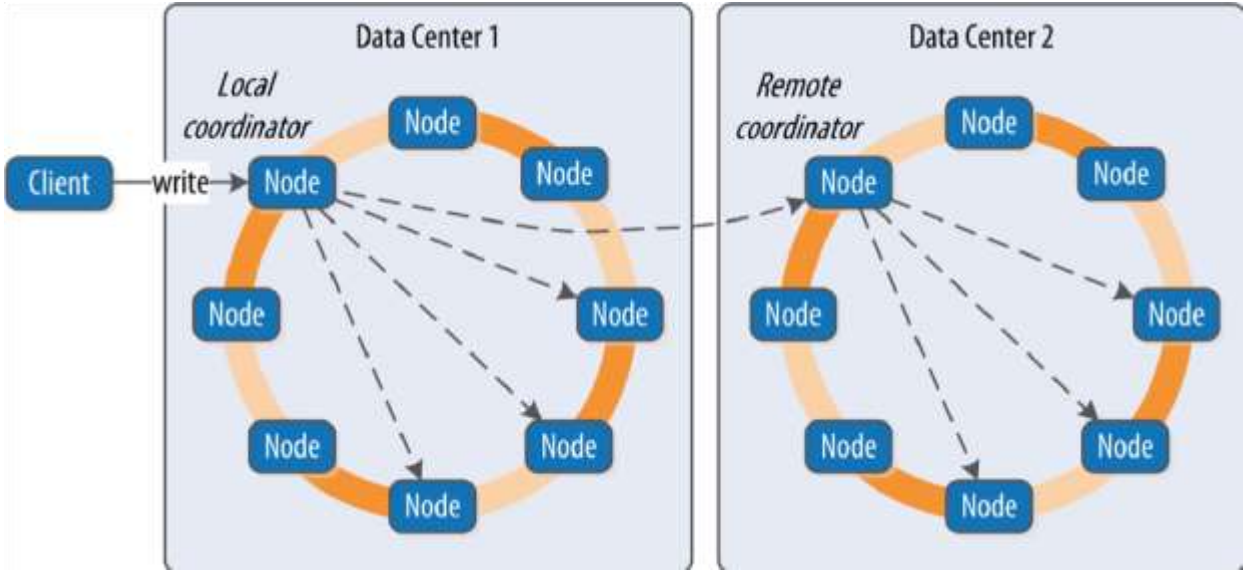
NETWORK TOPOLOGY STRATEGY



No „master" copy, active everywhere

„Last" write wins (timestamp)

REPLICATION – MULTI-DATACENTER



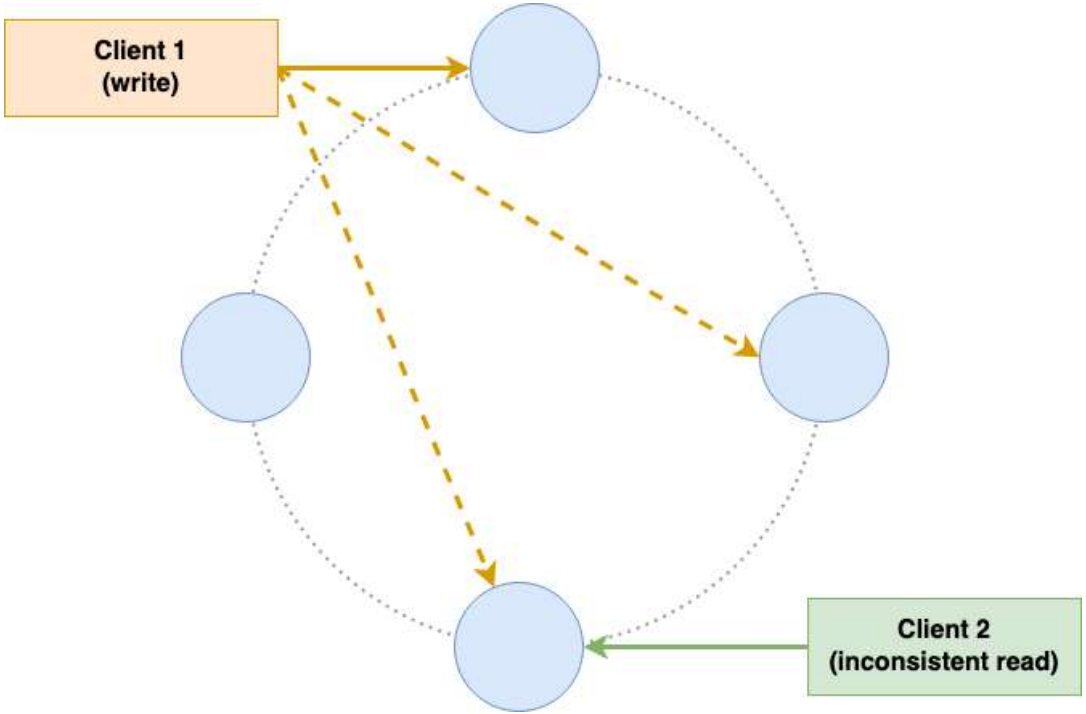
Data centers can have different

- Number of nodes
- Replication factor

Active everywhere

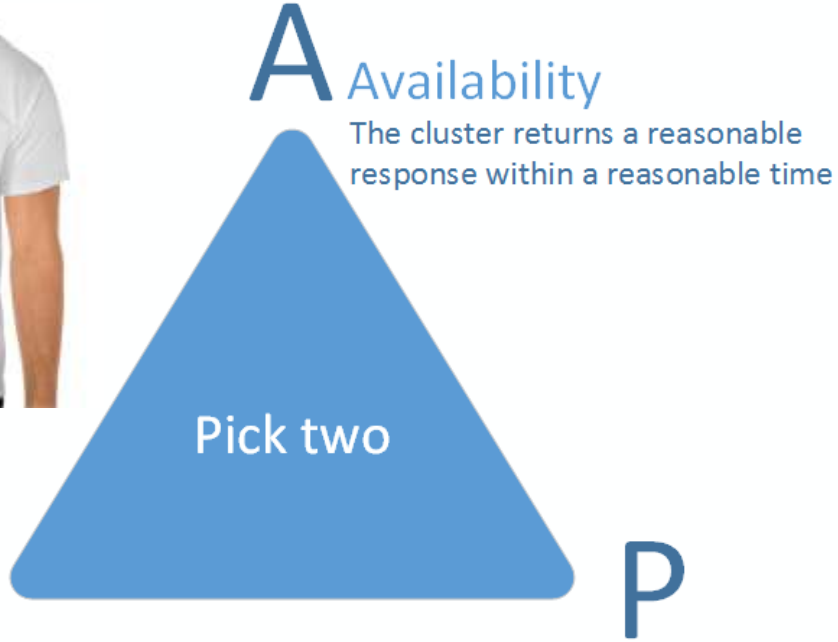
- Any DC and any node can receive reads and writes
- Cluster is operational even if the whole DC fails

REPLICATION – IMPLICATIONS



- Replication is slow
  - usually a background process
- All nodes are equal
  - (no master node)
- Inconsistent read
  - stale data (milliseconds)

CONSISTENCY – CAP THEOREM



Not Partition Tolerant (AC)

- Relational DBs

Primarily Available (AP)

- Cassandra
- Riak
- Dynamo

Primarily Consistent (CP)

- MongoDB
- Hbase
- Redis

NOT CONSISTENT ???

When is consistency mandatory?

Social media post

Webshop Price and Stock

Debit Card Payment

Consistency is overrated

- Most of the times we don’t need full consistency…
- ... but sometimes we do.

TUNABLE CONSISTENCY

Consistency Level

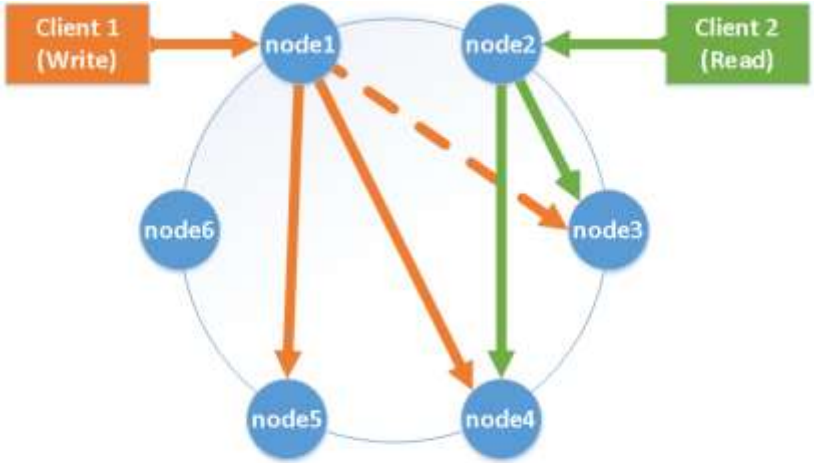
- The # of nodes the client waits for
- Can be specified for every operation

Possible values

- **ONE** (default), TWO, THREE
- ALL
- QUORUM
- LOCAL_ONE, LOCAL_QUORUM, EACH_QUORUM
- ANY

Consistency is achieved if

- Write CL + Read CL > RF



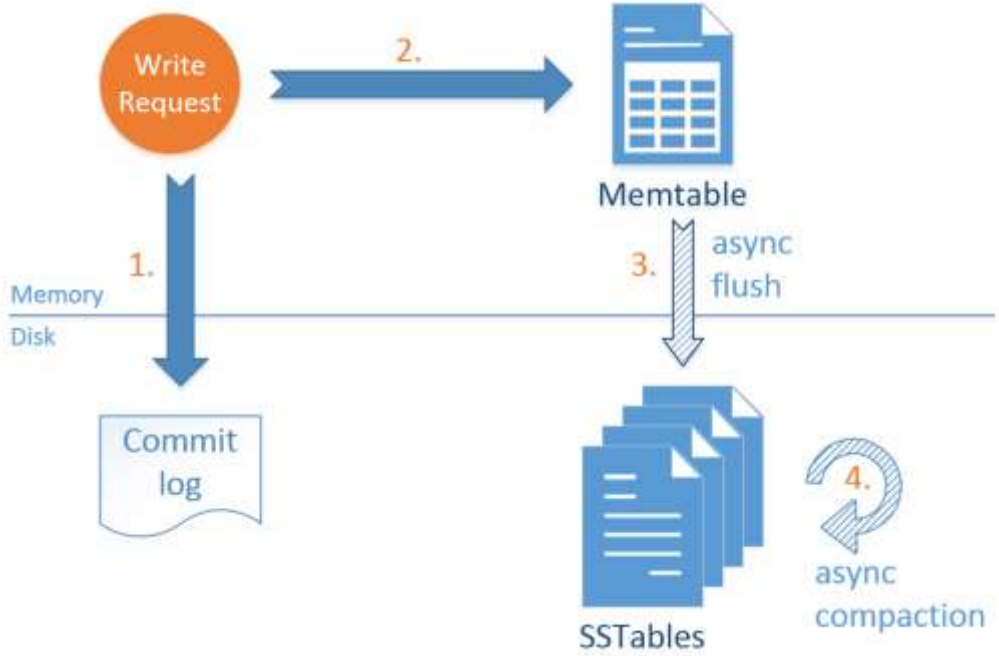
Replication Factor: 3
Write Consistency Level: QUORUM
Read Consistency Level: QUORUM

DEMO #3 – ARCHITECTURE

Source code - Demo #3

- Partitioning
- Replication
- Consistency

WRITE PATH – DURABILITY & PERFORMANCE



Commit Log

- Append-only file on disk, for disaster recovery
- Batch or Periodic mode

Memtable

- Has the structure of the table

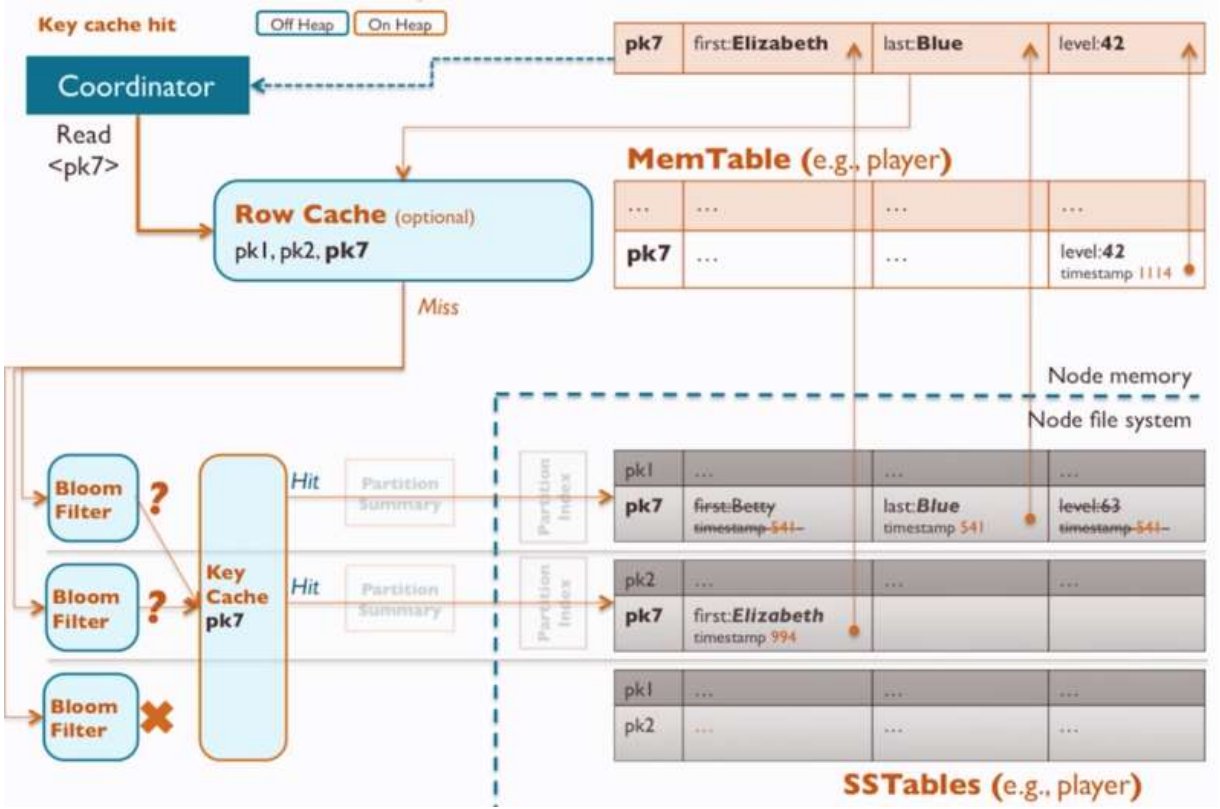
SSTables

- Memtables are flushed when large or old
- Sorted String Table, copy of the memtable
- Immutable files

Compaction Strategies

- Size tiered (default)
- Date tiered
- Leveled

READ PATH



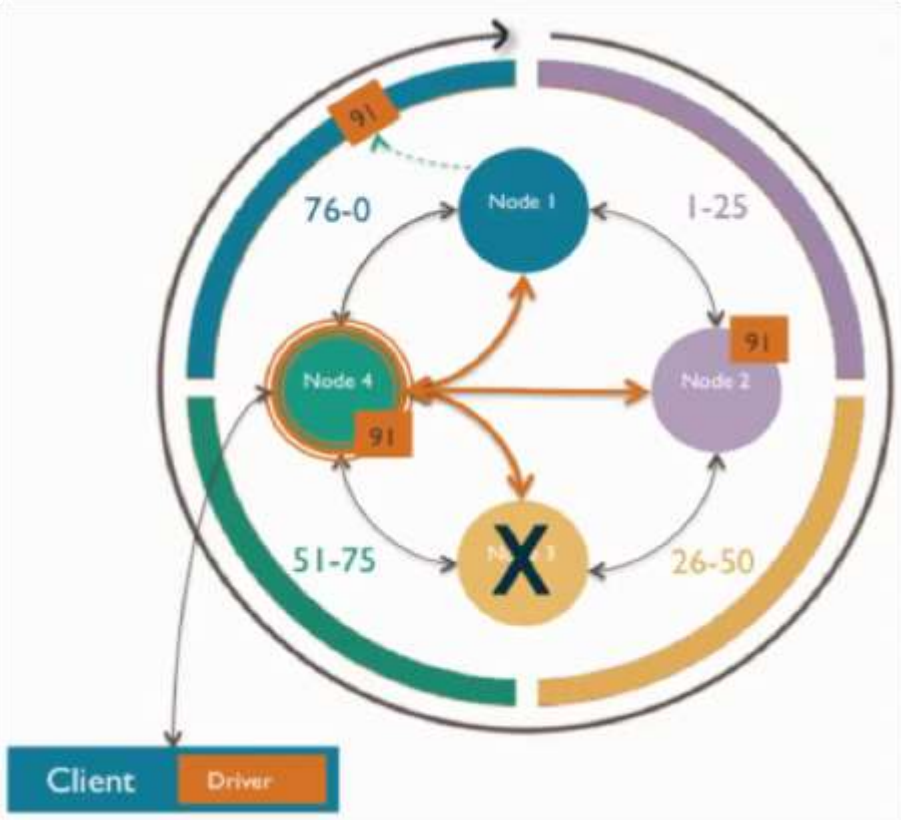
- Row Cache is optional
- Bloom filters
  - Are fast
  - Allow false positives

CLUSTER FORMATION – GOSSIP PROTOCOL

(image copyright by Márta Kiss)

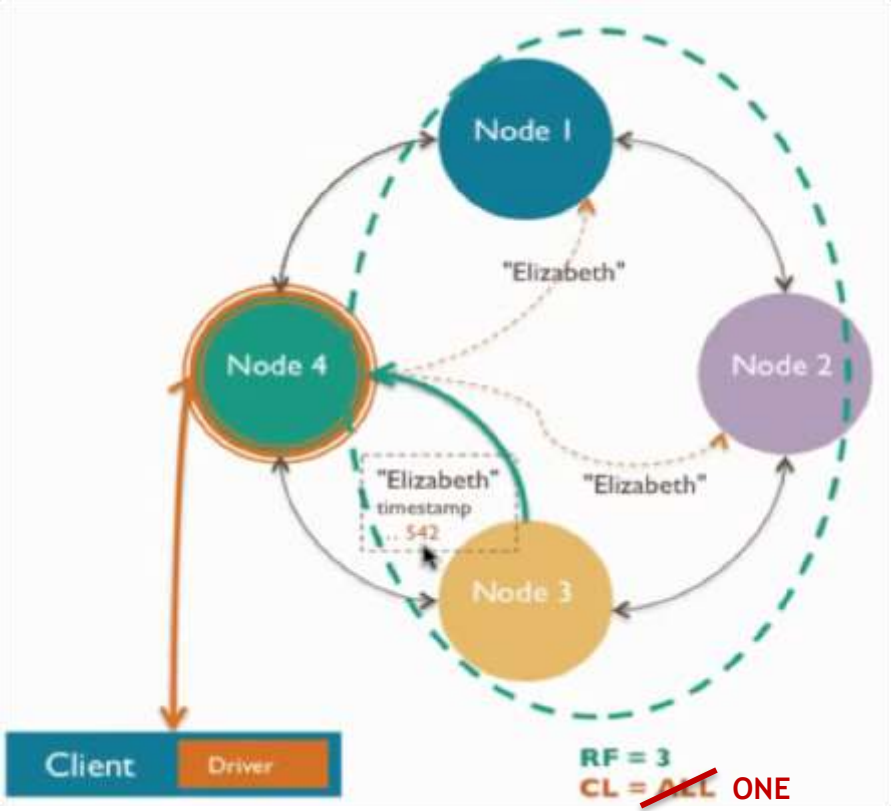
- Nodes are in constant communication.
- It is enough to know a few nodes; you will hear about everything.

HINTED HANDOFF



- Recovery mechanism for writes targeting offline nodes.
- Coordinator stores the failed writes temporarily
  - In the system.hints table (2.x)
  - In its own file system (3.x)
- Hints are delivered when the node comes back.

READ REPAIR



Background check of consistency

- read_repair_chance (0.2 is a typical value)
- Remaining nodes send a digest of the data
- Mismatches are corrected in the background

ARCHITECTURE – TAKE AWAY

- Master-less architecture
- Multi-DC support, active anywhere
- Eventual consistency
- Tunable consistency
- Durability & Performance

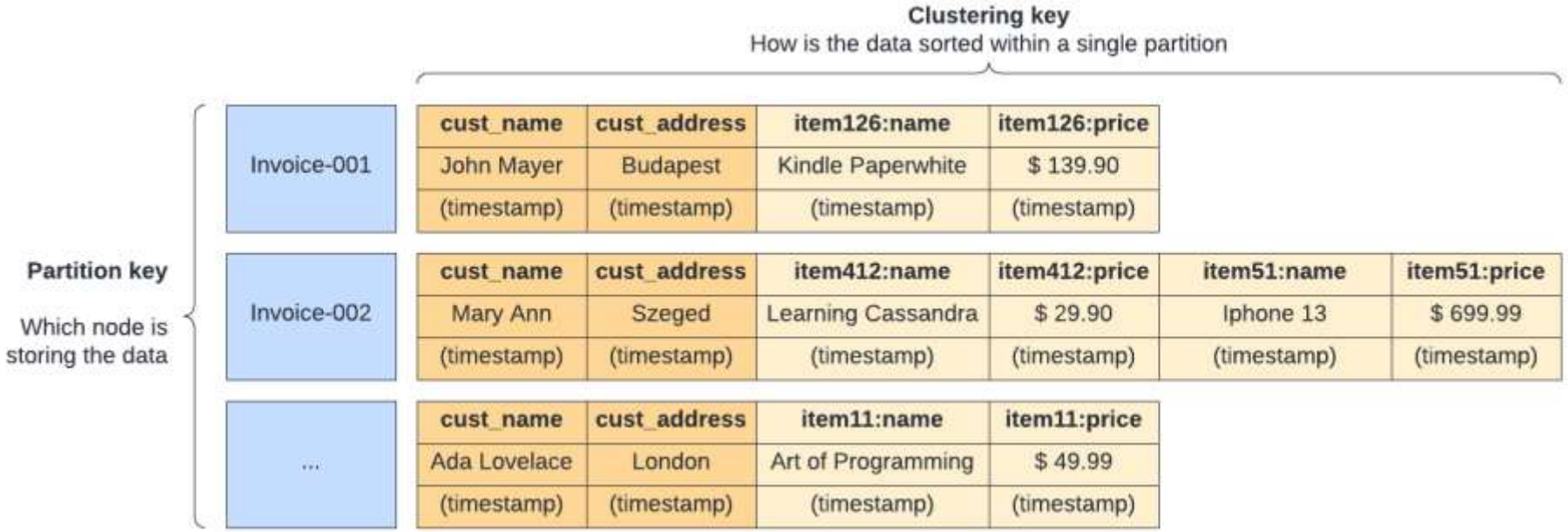
CQL – CASSANDRA QUERY LANGUAGE

AGENDA – CQL

1. Data Model
2. Architecture
3. **Deep Dive - CQL**
4. Data Modeling

- Keys in Cassandra
- Filtering & Ordering
- Advanced CQL
- Limitations

KEYS IN CASSANDRA



- **Partition key** is mandatory
- **Clustering key** is optional

- In CQL, **Primary key** is used to express both partition key and clustering key

PRIMARY KEY & PARTITION KEY

Primary key

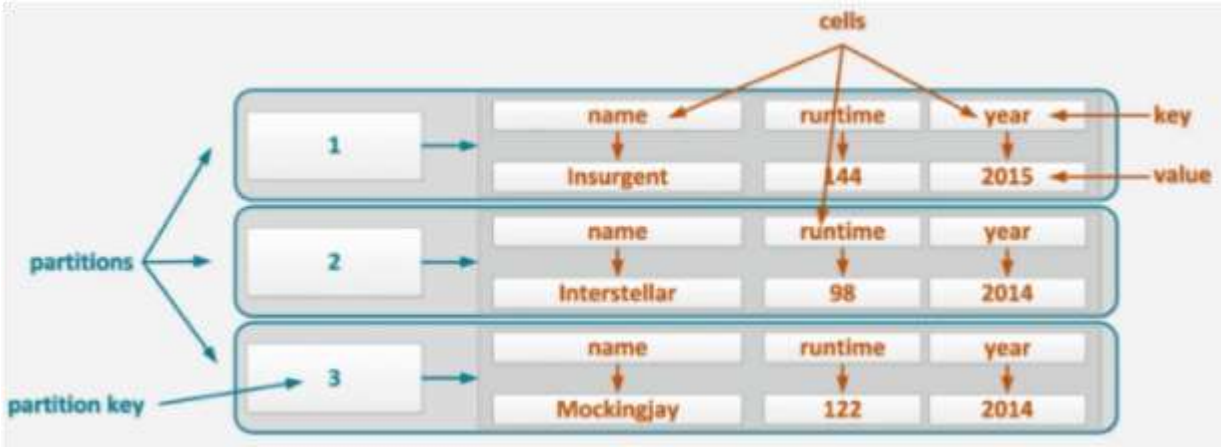
- is mandatory
- is unique
- determines partitioning

Logical view (SELECT * FROM car_registry_1)

id	name	runtime	year
1	Insurgent	119	2015
2	Interstellar	98	2014
3	Mockingjay	122	2014

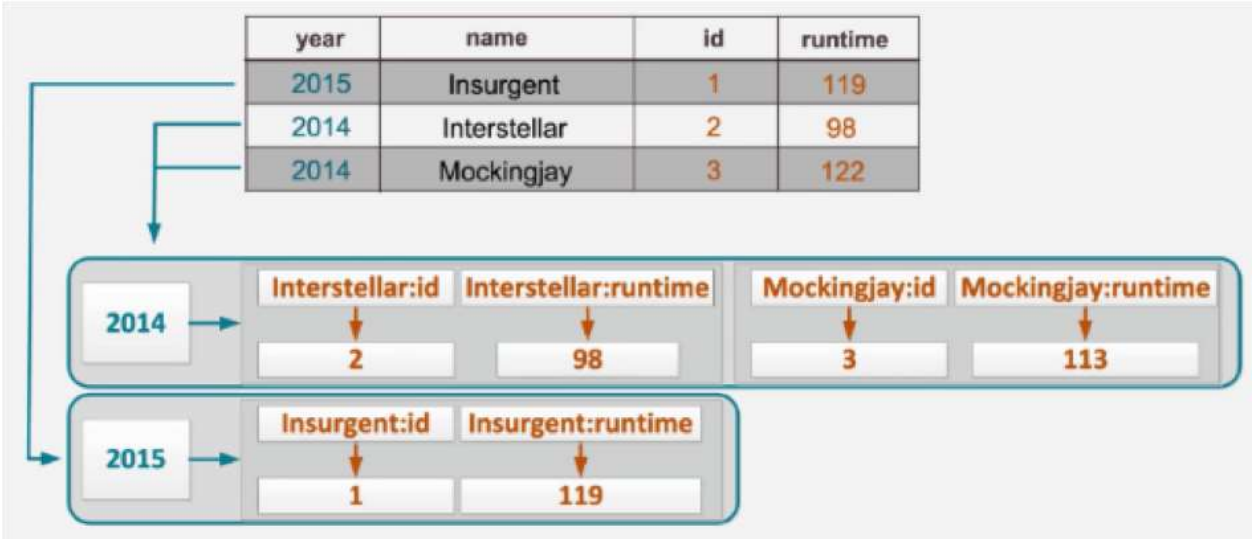
```
CREATE TABLE movies
(
  id int,
  name text,
  runtime int,
  year int,
  PRIMARY KEY (id)
);
```

Internal representation (row key, columns)



PRIMARY KEY = PARTITION KEY + CLUSTERING KEY

```
CREATE TABLE movies (
  id int,
  name text,
  runtime int,
  year int,
  PRIMARY KEY (year, name)
);
```



Clustering key: Sorts fields within partition

Primary key: Partition key(s) + Clustering key(s)

Compound Partition Keys: **PRIMARY KEY ((pk1, pk2), ck1, ck2);**

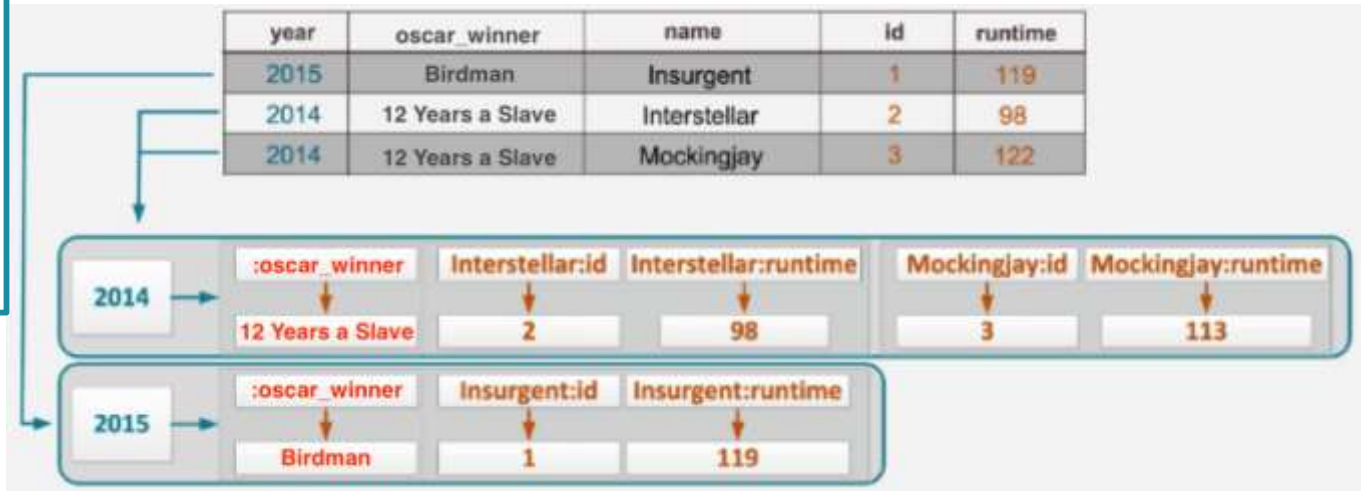
PRIMARY KEY CHEAT SHEET

PRIMARY KEY	PARTITION KEY	CLUSTERING KEY
-	ERROR: Primary key is mandatory	
PRIMARY KEY (field1)	field1	
PRIMARY KEY (field1, field2)	field1	field2
PRIMARY KEY (field1, field2, field3)	field1	field2, field3
PRIMARY KEY ((field1, field2))	field1, field2	
PRIMARY KEY ((field1, field2), field3, field4)	field1, field2	field3, field4

NOTE: Primary key must also be unique!

STATIC COLUMN

```
CREATE TABLE movies (
  id int,
  name text,
  runtime int,
  year int,
  oscar_winner text static,
  PRIMARY KEY (year, name)
);
```



oscar_winner is unique by year, we can store it once

Static fields are stored only once per partition key

DEMO #4 – PRIMARY KEYS

Source code - Demo #4

- Primary keys
- Partition keys
- Clustering keys

DEMO #5 – FILTERING & ORDERING

Source code - Demo #5

- Filtering
- Ordering

FILTERING BY KEY

By Partition Key

```
CREATE TABLE table1 (
  pk int,
  ck1 int,
  ck2 text,
  PRIMARY KEY (pk, ck1, ck2)
);
```

```
-- When filtering by partition key
-- values must be specified exactly
-- (inequality operators are not supported)
SELECT * FROM table1 WHERE pk = 2;

-- IN operator is also supported
--  CAUTION: it can overload the cluster!
SELECT * FROM table1 WHERE pk IN (1, 3, 5);
```

By Partition Key and Clustering Key

```
-- Inequality operators for clustering columns
SELECT * FROM table1 WHERE pk = v1
AND ck1 >= v2;

-- ERROR! ck1 is not restricted
SELECT * FROM table1 WHERE pk = v1
AND ck2 = v2;

-- ERROR! Must restrict ck1 by equality
SELECT * FROM table1 WHERE pk = v1
AND ck1 >= v2 AND ck2 <= v3;

-- Slicing. can combine clustering columns
SELECT * FROM table1 WHERE pk = v1
AND (ck1, ck2) >= (v1, v2);
```

SECONDARY INDEXES

```
CREATE TABLE track (
  track_id int,
  title text,
  album text,
  artist text,
  PRIMARY KEY(album, track_id)
);

-- ERROR!
-- Cannot filter on non-key, non-indexed fields
SELECT * FROM track WHERE artist = 'U2';

CREATE INDEX ON track(artist);

-- OK. filter by secondary index
SELECT * FROM track WHERE artist = 'U2';
```

THE BAD

- Secondary indexes are much slower than keys
- Indexes are maintained on each node, thus queries are affecting the whole cluster

THE GOOD

- Performance could be enough. Measure.
- Fast, if previously filtered by partition key:

```
SELECT * FROM track
WHERE album = 'Joshua Tree'
AND artist = 'U2';
```

ORDERING

- Records within a partition are ALWAYS sorted by clustering key (ASC by default)
- ORDER BY is only valid on clustering keys and can only change direction
    - WITH CLUSTERING ORDER BY – can change the default ordering to DESC

```
-- Can change the default clustering order (ASC)
CREATE TABLE transaction (
  accountnr text,
  transactiondate timestamp,
  recipient text,
  amount decimal,
  PRIMARY KEY (accountnr, transactiondate)
)
WITH CLUSTERING ORDER BY
(transactiondate DESC);
```

```
-- Records are sorted in DESC order
SELECT * FROM transaction WHERE accountnr = '001';

-- Records are sorted in ASC order
-- (slower if the clustering order doesn't match)
SELECT * FROM transaction WHERE accountnr = '001'
ORDER BY transactiondate;

-- Records are sorted in DESC order
SELECT * FROM transaction WHERE accountnr = '001'
ORDER BY transactiondate DESC;
```

DELETIONS

```
-- Delete a row
DELETE FROM table1 WHERE pk = value;

-- Time to live feature (seconds)
-- Row is deleted on expiry
INSERT INTO table1 (pk, f1)
VALUES (v1, v2)
USING TTL 60;
```

Tombstone

Deleted data is replaced with "tombstones" in order to avoid its reappearance from another replica.

- Tombstones are kept for `gc_grace_seconds`
  (default value is 10 days)
- Expired tombstones are removed
  during compaction
- Resurrecting deleted data is possible
  if a node is down for more than `gc_grace_seconds`
- Too many tombstones hinder performance

LIMITATIONS OF CQL

- NO JOINS
  - Expensive, don’t scale
- Limited WHERE (as seen before)
  - By keys & secondary index
- Limited ORDER BY (as seen before)
  - By clustering key, can change direction
- Limited GROUP BY and aggregates (since 2.2)
  - Aggregates: COUNT, SUM, MIN, MAX, AVG
  - Only by fields of the primary key (in order)
- Limited SELECT DISTINCT
  - For partition keys and static columns

Despite the SQL interface,
it is not an RDBMS

- Distributed System
- Built for Scalability and Performance
- SQL is only an interface for convenience

DEMO #6 – ADVANCED CQL

Source code - Demo #6

- Counter tables
- Lightweight transactions
- Materialized views
- JSON support

COUNTER TABLES

AUTONUMBER is difficult in a distributed system

```
-- Only key and counter columns are allowed
CREATE TABLE employee_counter (
  employeeid bigint,
  year int,
  vacation_days counter,
  badges counter,
  PRIMARY KEY (employeeid, year)
);
```

```
-- Update counter is the only operation allowed
UPDATE employee_counter
SET badges = badges + 1
WHERE employeeid = 1 AND year = 2016;

-- ERROR: Cannot set, just increment/decrement
UPDATE employee_counter
SET badges = 17
WHERE employeeid = 1 AND year = 2016;
```

TRANSACTIONS – NO ACID

Batches

```
BEGIN BATCH
  INSERT INTO track_by_artist ...
  INSERT INTO track_by_album  ...
APPLY BATCH;
```

- **ATOMIC**
  - all operations are reverted on failure
- **NOT ISOLATED**
  - other operations can see the un-applied changes

Supports INSERT, UPDATE and DELETE

(SELECT is not supported)

Execution order is not deterministic!

Lightweight Transactions

```
-- Verify by primary key
INSERT INTO users (username, name, passwordhash)
VALUES ('ada', 'Ada Lovelace', '20a46ee0')
IF NOT EXISTS;

-- Verify by any field
UPDATE users
SET passwordhash = '20a5f580'
WHERE username = 'ada'
IF passwordhash = 'FFa46ee0';
```

[applied]	username	name	passwordhash
False	ada	Ada Lovelace	20a46ee0

MATERIALIZED VIEWS

```
CREATE MATERIALIZED VIEW laureates_by_category AS
    SELECT * FROM nobel_laureates
    -- primary key fields must be non-null
    WHERE category IS NOT NULL
    AND laureateid IS NOT NULL
PRIMARY KEY (category, year, laureateid)
WITH CLUSTERING ORDER BY
(year DESC, laureateid ASC);
```

- Available only from version 3.x
- Uses batchlog to maintain consistency
- Can delete data if primary key fields change

Restrictions for materialized views:

- Include all source primary keys in the materialized view's primary key.
- Only one new column can be added to the materialized view's primary key. Static columns are not allowed.
- Exclude rows with null values in the materialized view primary key column.

JSON SUPPORT

```
INSERT INTO contacts JSON
'{
  "id": 2,
  "name": "epam",
  "phones":
  {
    "Budapest": "+36 1 3277400",
    "Szeged": "+36 62 550656"
  }
}';
```

- Json string must match the fields and data types of the table
- Supports collections and user defined types

- Allows easy JSON processing from client code.
- Drivers don't need any special support
  (it is a feature of the engine)

```
String cql = "INSERT INTO contacts JSON ?";
String json = "{ … }";

preparedStatement = session.prepare(cql);
statement = preparedStatement.bind(json);

session.execute(statement);
```

CQL – TAKE AWAY

- CQL is built on top of internal data structure
- Primary Key = Partition Key + Clustering Key
- Upsert: Inserts and Updates are equivalent
- Writes without read
- Collections are stored by element
- Think performance:
    - Filter by key (and secondary index)
    - Limited Aggregations
- Lightweight Transactions

DATA MODELING

AGENDA – DATA MODELING

1. Why Cassandra?
2. Data Layout
3. Architecture
4. CQL – Cassandra Query Language
5. **Data Modeling**

- Rules
- 1-to-many relations
- Exercise

MODELING RULES

DESIGN BY QUERY

- Enumerate all possible queries; and design the tables to support them.

DE-NORMALIZE TABLES

- Embed 1-to-1 relationships as extra fields or used defined types.
- Embed 1-to-n relationships as collections or clustering keys.

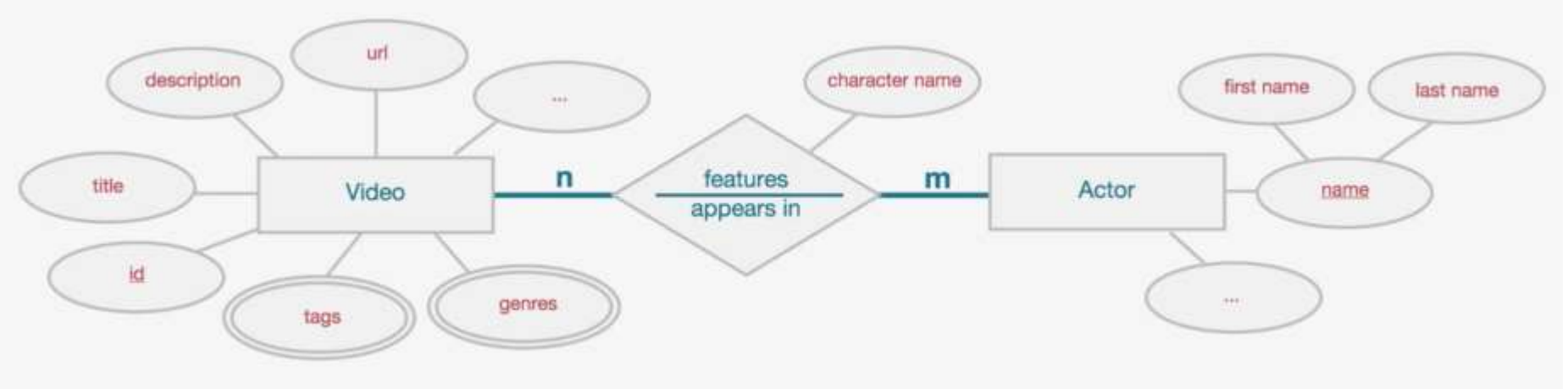
SECONDARY INDEXES

- Facilitate search by non-key columns.
- WARNING: they are much slower than lookups by key. Measure performance if needed.
- Secondary indexes can be FAST when data is already filtered by partition key.

DUPLICATE TABLES

- Create more tables with the same content, but different partition key, and update both.
- Consider using index tables to preserve space (only store the primary key of the main table)
- Use Materialized Views.

DESIGN BY QUERY



- Design a conceptual model for your domain
- Go through your queries and design a table for each
- Consider reusing a table for a different query only if you can do it efficiently

1-TO-MANY RELATIONSHIPS

CLUSTERING KEYS, STATIC FIELDS

```
CREATE TABLE invoice (
  -- header fields
  invoice_id bigint,
  issuer text static,
  address text static,
  total_price decimal static,

 -- detail fields
  article_id int,
  article_name text,
  price decimal,

  -- primary key
  PRIMARY KEY (invoice_id, article_id)
);
```

HEADER

```
-- retrieve header information only
SELECT DISTINCT
issuer, address, total_price
FROM invoice
WHERE invoice_id = 1111;
```

DETAIL

```
-- retrieve whole invoice
SELECT * FROM invoice
WHERE invoice_id = 1111;
```

DATA DUPLICATION

ORIGINAL TABLE

```
CREATE TABLE track_by_album (
  track_id int,
  title text, -- other fields
  album text,
  artist text,
  PRIMARY KEY(album, track_id)
);
```

DUPLICATE TABLE

```
CREATE TABLE track_by_artist (
  track_id int,
  title text, -- other fields
  album text,
  artist text,
  PRIMARY KEY(artist, track_id)
);
```

INDEX TABLE

```
CREATE TABLE album_by_artist (
  album text,
  artist text,
  PRIMARY KEY(artist, album)
);
-- first select albums by artist
-- then select tracks from original table by album
```

MATERIALIZED VIEW

```
CREATE MATERIALIZED VIEW track_by_artist AS
  SELECT * FROM track_by_album
  WHERE artist IS NOT NULL
  AND track_id IS NOT NULL
PRIMARY KEY (artist, album, track_id);
```


epam
Lab work

EXERCISE – DATA MODELING

- Design a Cassandra database and associated queries based on the description

EXERCISE – DATA MODELING

CONTEXT

We are creating the datastore for the **European Traffic Incidents Office**. All incidents arrive with the following information:

- *Registration number*
- *Registration country code*
- *Driver PID*
- *Driver name*
- *Incident's date and time*
- *Incident's location*
- *Penalty points (if any)*

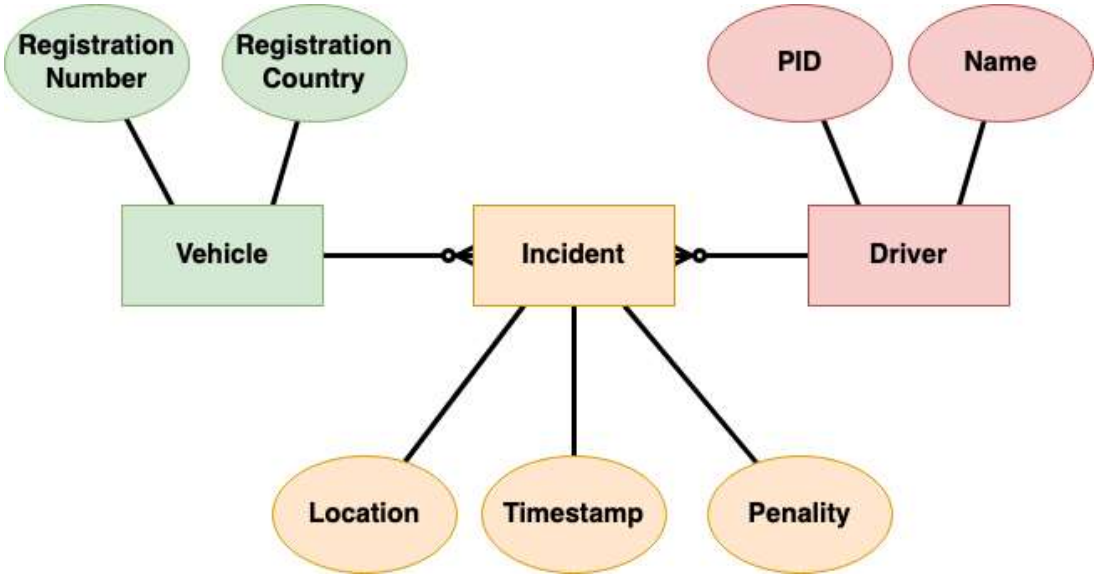
REQUIREMENTS

1. Provide a list of incidents of a vehicle for a specific date range, sorted by incident's date and time (newest first).
2. Provide the same list for a driver (identified by Driver PID), sorted by vehicle, then by date and time (newest first).
3. Display the penalty points collected by a driver.

TO PREPARE

- Table definitions
- CQL statements for data ingestion
- CQL statements for data retrieval

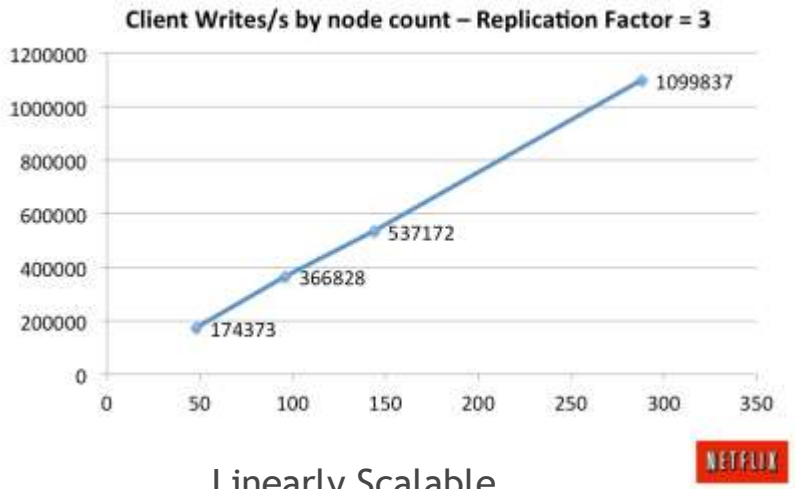
DATA MODELING EXERCISE



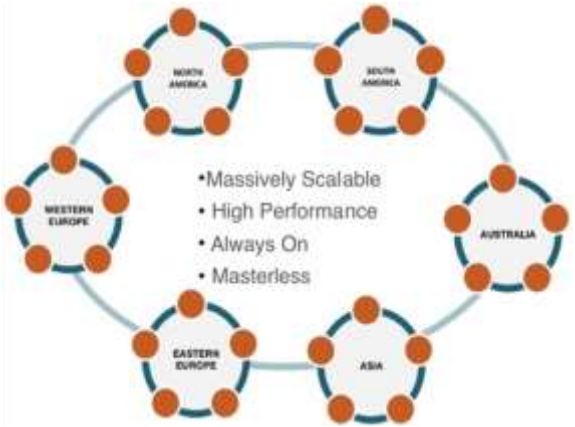
1. Create a conceptual model
2. Design your tables by query
3. Validate your design

CONCLUSIONS

WHAT IS CASSANDRA?



Linearly Scalable
High Throughput



No Single Point of Failure
Supports multiple Data Centers

SQL Support
(primary interface)

Used by

...and
many more

TYPICAL USE CASES – GOOD FOR

HIGH-SPEED DATA INGESTION

- IoT applications
- User activity tracking
- Results of big-data processing pipelines

SCALABLE STRUCTURED DATA

- Product catalogs & retail apps

MULTI-DATACENTER, ACTIVE ANYWHERE

- Geographically distributed apps
- Social media analytics

TIME-SERIES DATA

- IoT and sensor data
- Can benefit from TTL feature

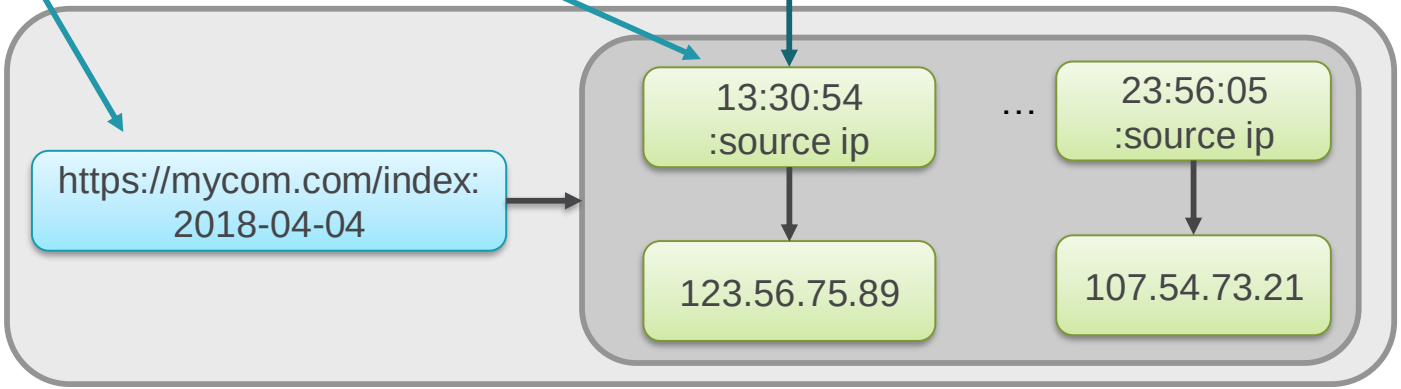
TIMESERIES EXAMPLE

Track the timestamp and source ip of page hits by url and date

```
create table if not exists page_hits_by_url_and_date(
    url text,
    capture_date text,
    capture_time time,
    source_ip inet,
    primary key((url,capture_date), capture_time)
);
```

Clustering column

Composite row key



CASSANDRA IS NOT IDEAL WHEN…

- You don’t need scalability
  - *use a RDMBS in such cases*
- Too many or unknown search conditions
  - *might combine with Solr & Spark though*
- Consistency is more important than availability
  - *tunable consistency & lightweight transactions can be considered*
- Difficult to map data to a columnar format
  - *JSON support & user defined types could help*
- Read-heavy workloads
  - *Primarily optimized for writes*
- Frequent, wide-range updates
  - *Background compaction has to keep up; use SSD*

CASSANDRA – SELF CHECK QUESTIONS

- What is the difference between the Partition key and the Clustering key?
- Why is it a bad idea to use the country code as partition key for a table of all European vehicles?
- In what circumstances could the following query be correct?
    - `SELECT * FROM table1 WHERE field1 < 1000;`
- What is the difference between the set, list and map collection types?
- How many nodes are we waiting for in a 10-node cluster, when the consistency level is QUORUM and the replication factor is 3? And when the RF is 4?
- When would you use a lightweight transaction?
- Is normalization or de-normalization the preferred approach in Cassandra data modeling?
- What kind of JOIN operations exist in Cassandra?

CASSANDRA – TAKE AWAY

- Master-less architecture
- Multi-DC support, active anywhere
- CAP, Eventual consistency & Tunable consistency
- Durability & Performance
- Design by Query

- Internal Data Model - map of maps
- CQL is just a convenience on top of that (limitations)
- Primary Key = Partition Key + Clustering Key


Questions?

About EPAM MEP

The Mentoring Program is our in-house upskilling program that allows you to **learn from our EPAM experts**.

The program will give you the chance to acquire the right knowledge **to become one of EPAM's technologists.**

- Java
- .NET
- Test Automation
- DevOps
- Data Engineering

Interested? Apply here: https://epa.ms/edu-hungary
Meetup tomorrow (HU): https://epa.ms/mep-meetup



CLIENT CONNECTIVITY

LANGUAGES & DRIVERS

Wide range of languages

- C#, Java,
- C / C++,
- Node.js, PHP, Python, Ruby

Popular drivers

- DataStax,
- Astyanax,
- Kundera,
- ODBC

DataStax driver features

- Async and sync requests
- Paging
- Node discovery
- Configurable load balancing
- Configurable retry policy
- Transparent failover handling

DATASTAX DRIVER

NODE DISCOVERY

```
cluster = Cluster.builder()
    .addContactPoint(node1)
    .build();

metadata = cluster.getMetadata();
hosts = metadata.getAllHosts();
```

LOAD BALANCING POLICIES

- DCAwareRoundRobinPolicy
- RoundRobinPolicy
- TokenAwarePolicy
- LatencyAwarePolicy

RECONNECTION POLICIES

- ConstantReconnectionPolicy
- ExponentialReconnectionPolicy

RETRY POLICIES

- DefaultRetryPolicy
- DowngradingConsistencyRetryPolicy
- FallthroughRetryPolicy
- LoggingRetryPolicy

DEVELOPMENT – OBJECT MAPPING

```java
@Table(name = "laureates")
public class Laureate {

    @PartitionKey
    private int year;

    @ClusteringColumn
    @Column(name = "laureateid")
    private int id;

    private String country;

    // other fields
}
```

```java
@Accessor
public interface LaureateAccessor {

    @Query("SELECT * FROM laureates WHERE year=2002")
    Result<Laureate> getYear2002();

    // parameters by order
    @Query("SELECT * FROM laureates WHERE year=?")
    Result<Laureate> getByYear(int year);

    // parameters by name
    @Query("SELECT * FROM laureates WHERE country=:cntr")
    Result<Laureate> getByCountry(@Param("cntr") String country);
}
```

See also: Spring Data for Apache Cassandra

OPERATIONS & MONITORING

Operations – nodetool

Command-line tool for managing Cassandra, can perform many actions:

Cluster

- status – display cluster info
- repair – fix replication issues
- getendpoints – display nodes for a key

Server

- info – display server info
- tpstats – thread pools, dropped msgs
- tablehistograms – latencies

Network

- proxyhistograms
- netstats

Storage

- cleanup – after bootstrapping a new node
- flush – dump memtable to an sstable

Compaction

- compact – DON'T DO IT IN PROD !
- compactionstats

Backup

- snapshot – create backup
- clearsnapshot

Operations – other command line tools

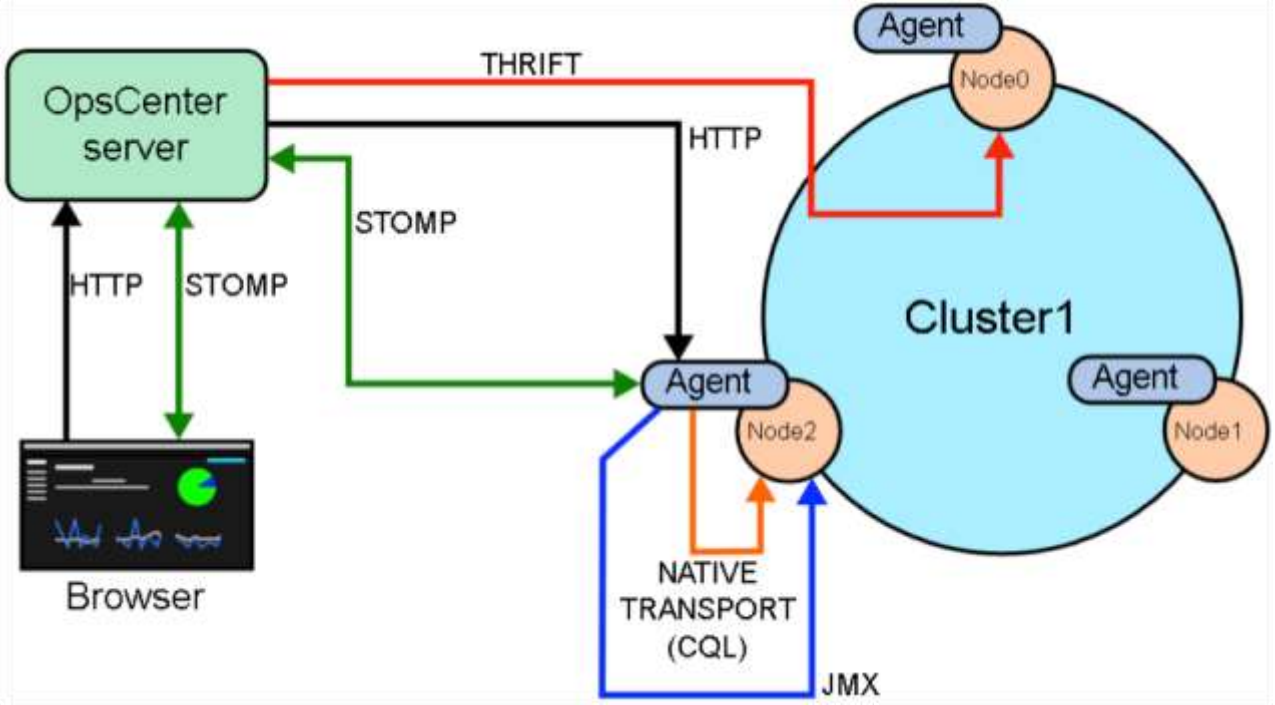
cassandra-stress

- java-based stress testing utility for benchmarking and load testing a Cassandra cluster.

sstable utilities

- sstable2json – display content
- sstablekeys – display row keys
- json2sstable – create sstable
- sstablesplit – oposite of compact
- sstableloader – bulk-load data

Operations – OpsCenter overview



Features

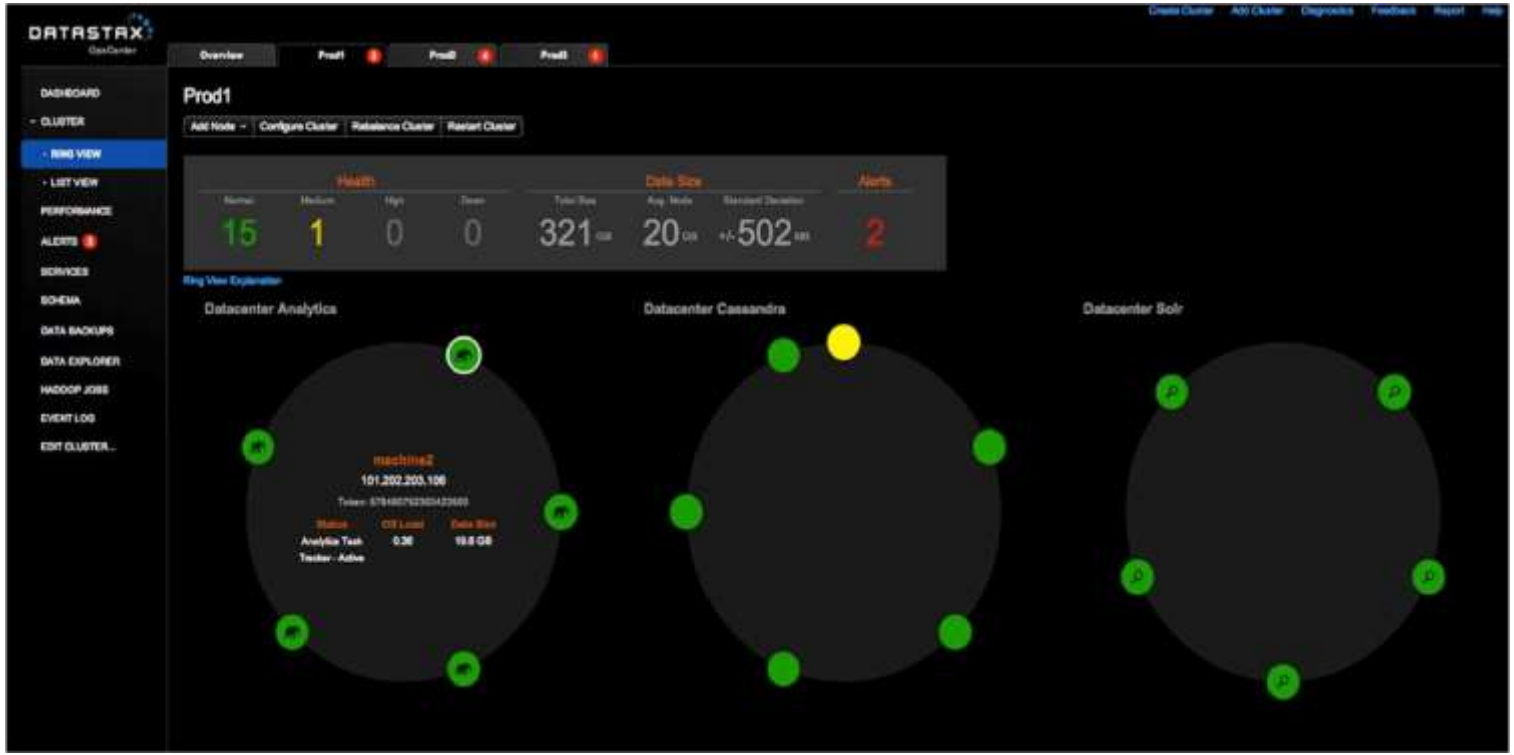
- Display performance metrics
- Simplify cluster management tasks
- Monitoring and alerting

Operations – OpsCenter performance metrics

Displays

- CPU usage
- Latencies
- Throughput

Operations – OpsCenter monitoring and alerts

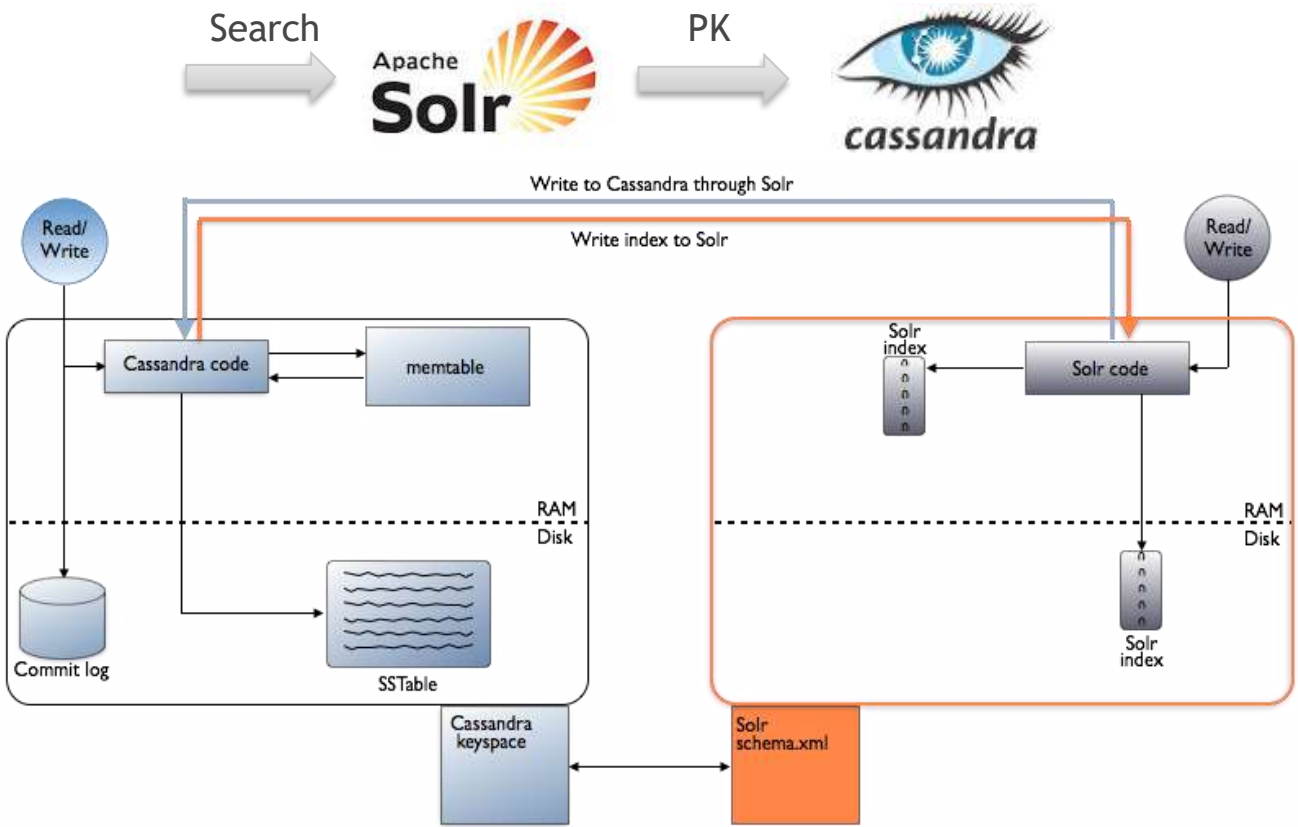


CASSANDRA & BIG DATA

Cassandra Integrations

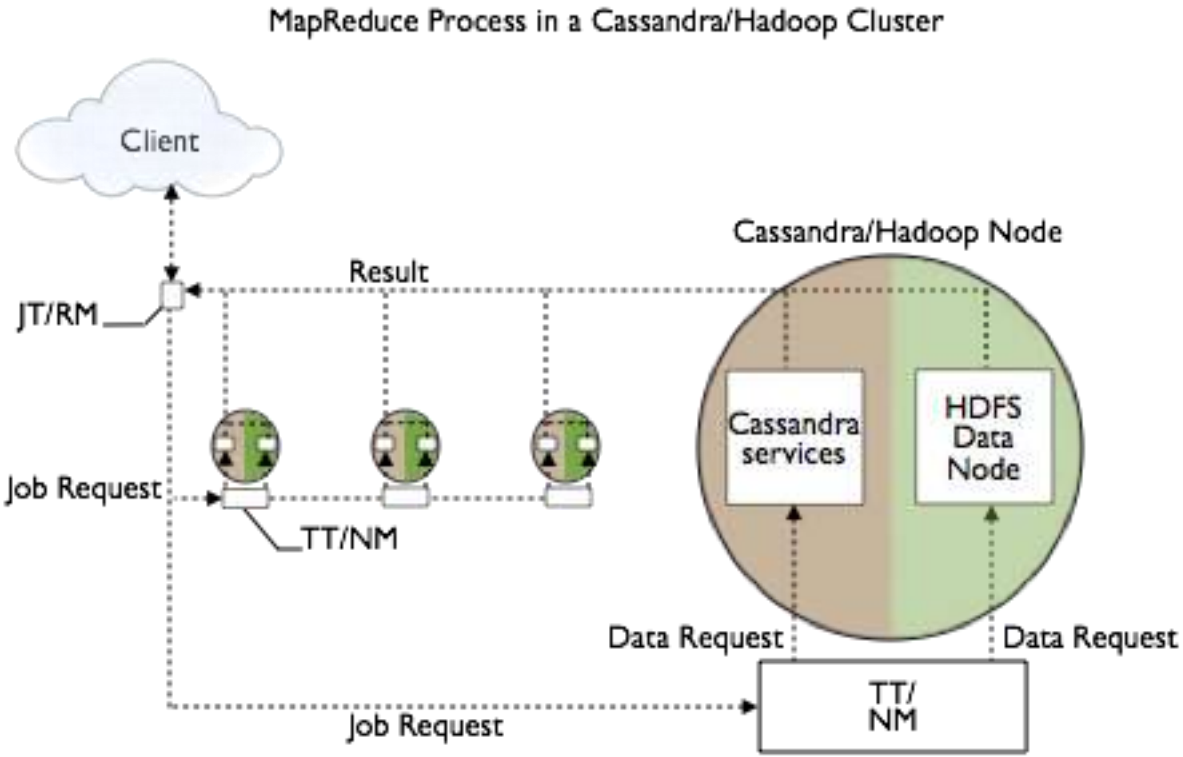
- Although it is not part of Hadoop, it integrates well with many components of the Hadoop ecosystem

Cassandra & Solr



- DataStax Enterprise provides seamless integration of Cassandra and Solr.
- Solr can perform searches that return the primary key for Cassandra.
- Cassandra can then return the actual data.

Cassandra & Hadoop



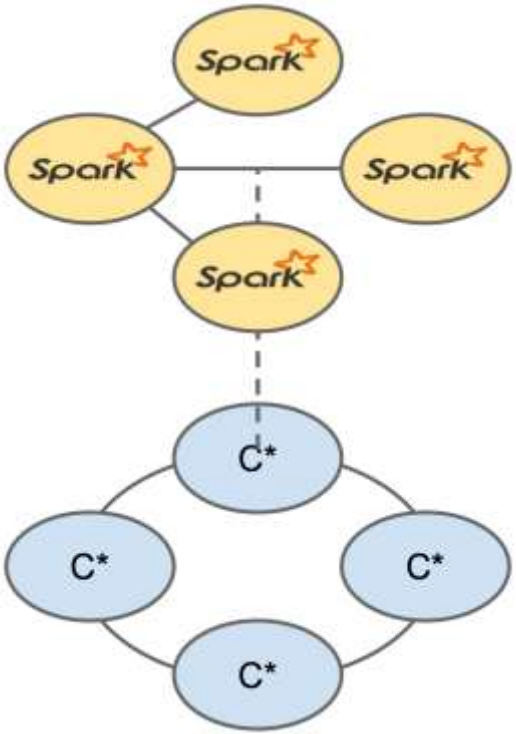
Cassandra comes with

- CqlInputFormat
- CqlOutputFormat and
- CqlBulkOutputFormat

classes.

These act as Hadoop specific InputFormat and OutputFormat implementations.

Cassandra & Spark



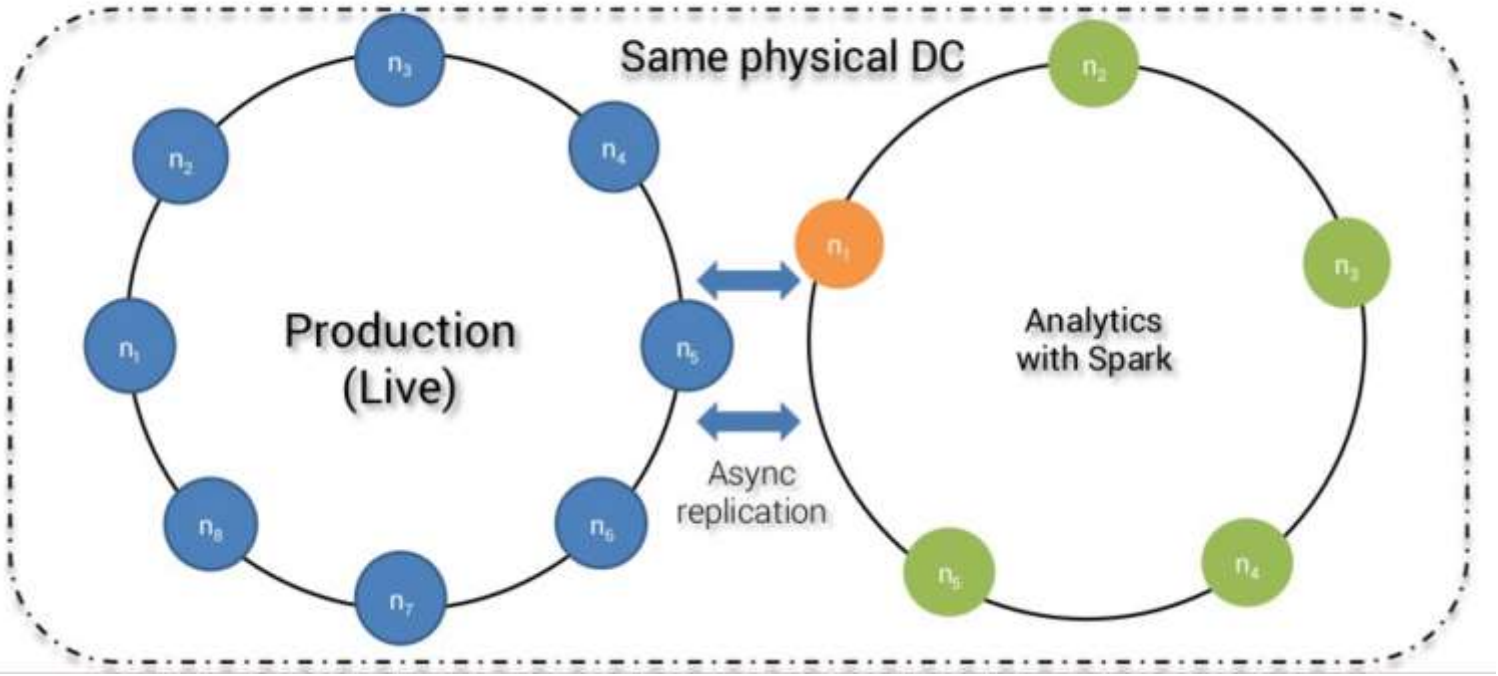
- Install Spark Workers on all Cassandra nodes to benefit from Data Locality.
- Spark can use Cassandra secondary indexes on every node efficiently.

DataStax Spark Cassandra connector

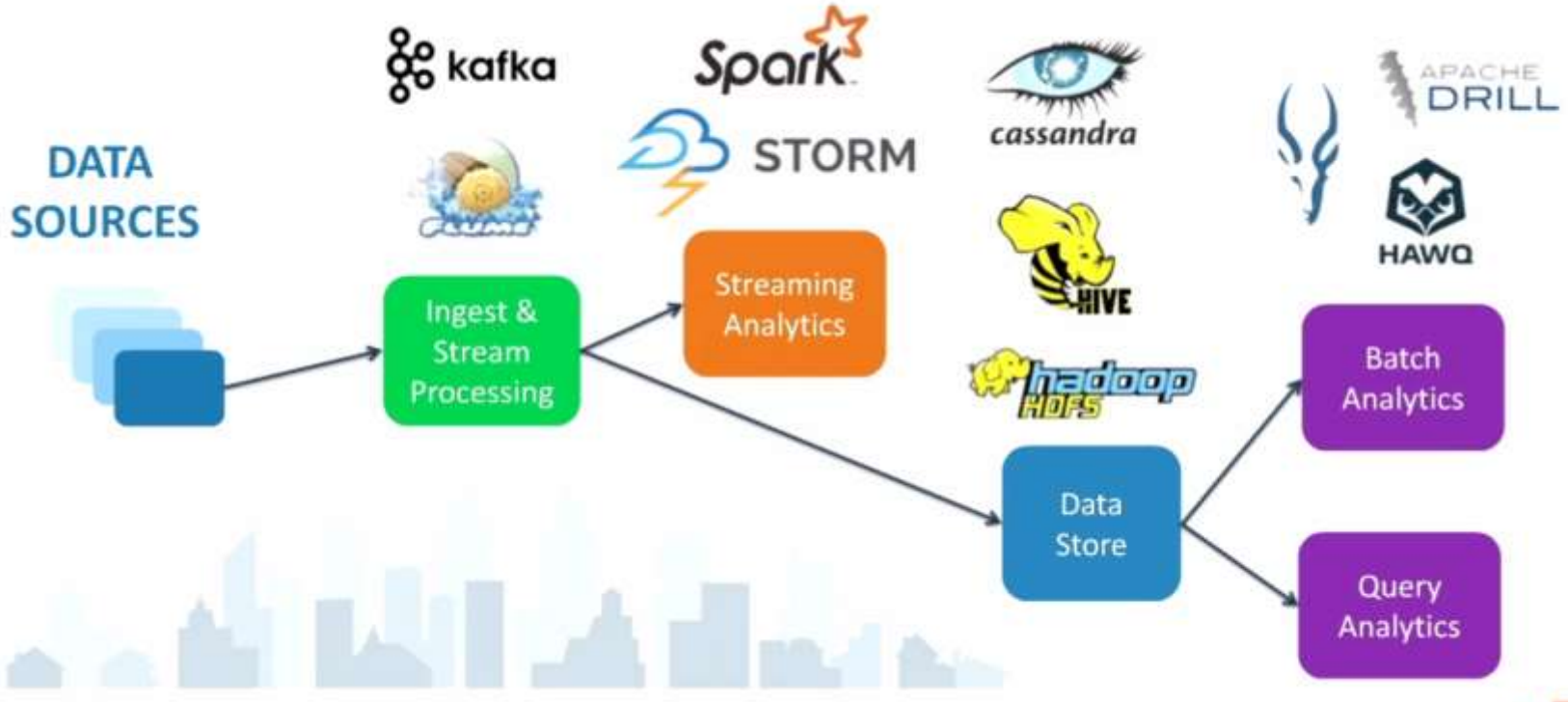
```
import com.datastax.spark.connector._

// saving to Cassandra
predictionsRdd.join(music)
  .saveToCassandra("keyspace", "table1")

// reading from Cassandra
val rdd = ssc.cassandraTable[MyEntity]("keyspace", "table1")
val results = rdd.where("key = ?", value).collect
```

Cassandra & Spark – Datacenter for Analytics

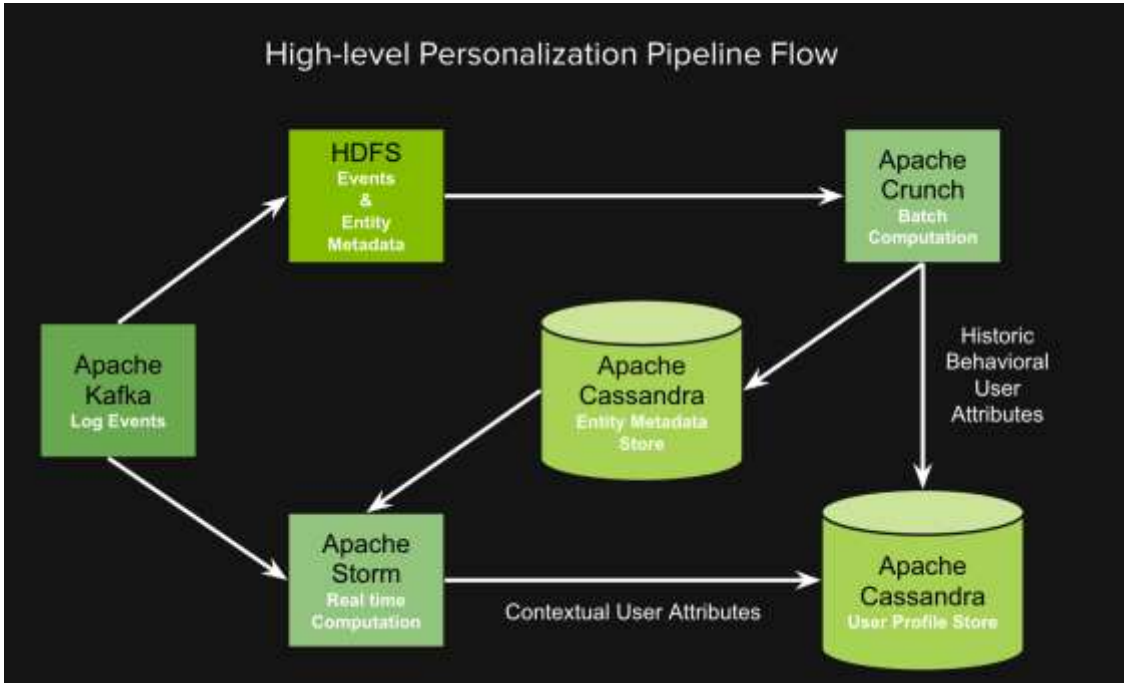
Workload segregation with virtual DC



Use cases – A Typical Analytics Model



Use cases – Spotify

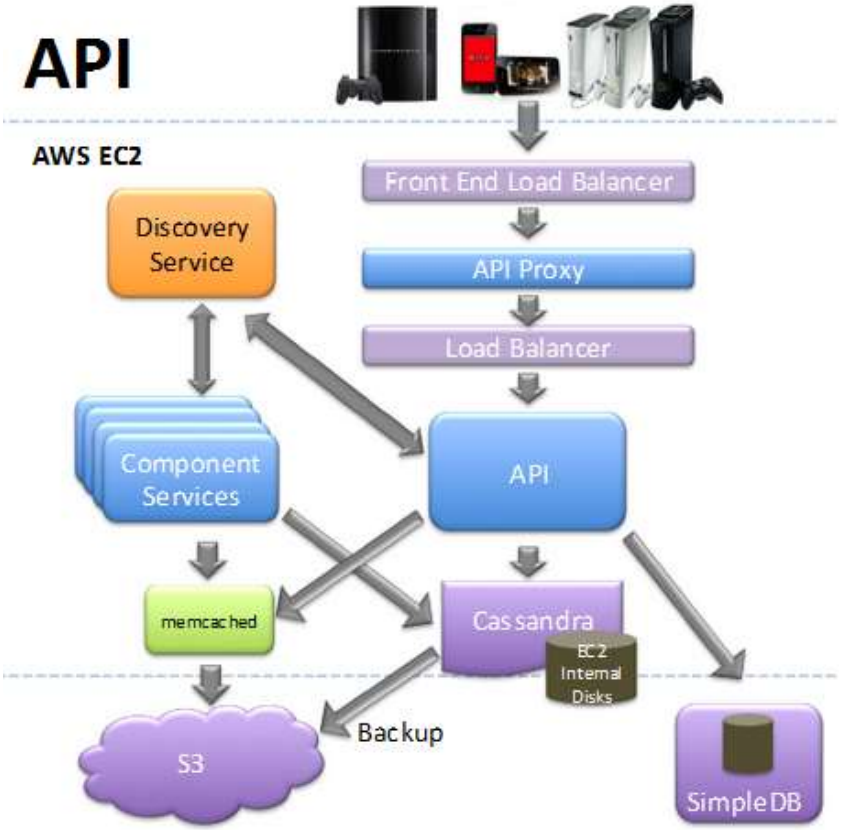


- **Cassandra** is used to store user profile attributes and metadata about entities like playlists, artists, etc.

Why Cassandra?

- **Scale** horizontally
- Support **cross-site replication**
- Have **low latency** even at the cost of consistency since we aren't performing transactions
- Have the ability to **load bulk** and streaming data from Crunch and Storm respectively

Use cases – Netflix



Why Cassandra?

- Fast
  - low latency, low latency variance
- Scalable
  - Runs on Amazon EC2
  - High and scalable throughput
  - Multi-region clusters
- Available
  - Amazon availability zones
  - Data integrity checks and repairs
  - Online backup / restore

MongoDB

NoSQL contender for SQL's place

Dávid Szabó

April, 2023

WHAT IS MONGODB

NoSQL

- Non-relational model
- Horizontally scalability
- Flexible schema

WHAT IS MONGODB

NoSQL

- Non-relational model
- Horizontally scalability
- Flexible schema

- Document-oriented
- General purpose
- Free and open-source
  - Enterprise version exists
  - Managed: MongoDB Atlas

```
{
    _id: 1,
    name: { first: „John”, last: „Doe” },
    birthDate: Date(„1988-10-10”),
    contacts: {
        email: „jdoe@example.com”,
        phone: „999-999-999”,
        skype: „JohnDoe88”,
        twitter: „realJohnDoe”
    }
}
```

WHAT IS MONGODB

Rank Apr 2023	Rank Mar 2023	Rank Apr 2022	DBMS	Database Model	Score Apr 2023	Score Mar 2023	Score Apr 2022
1.	1.	1.	Oracle	Relational, Multi-model	1228.28	-33.01	-26.54
2.	2.	2.	MySQL	Relational, Multi-model	1157.78	-25.00	-46.38
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	918.52	-3.49	-19.94
4.	4.	4.	PostgreSQL	Relational, Multi-model	608.41	-5.41	-6.05
5.	5.	5.	MongoDB	Document, Multi-model	441.90	-16.89	-41.48
6.	6.	6.	Redis	Key-value, Multi-model	173.55	+1.10	-4.05
7.	7.	↑ 8.	IBM Db2	Relational, Multi-model	145.49	+2.57	-14.97
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	141.08	+2.01	-19.76
9.	9.	↑ 10.	SQLite	Relational	134.54	+0.72	+1.75
10.	10.	↓ 9.	Microsoft Access	Relational	131.37	-0.69	-11.41

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

WHO USES MONGODB

Single View

eCommerce

Internet of Things

Digital Transformation

Mobile

Travel Graph &
Recommendation System

Product Catalog

Drug Sequencing

Database as a Service

Analytics

Artificial Intelligence

Gaming

More on https://www.mongodb.com/who-uses-mongodb

OVERVIEW

AGENDA

1. Data Model
2. Architecture
3. Workshop
4. Exercise

DATA MODEL

AGENDA

1. **Data Model**
2. Architecture
3. Workshop
4. Exercise

- Documents
- Data modelling principles

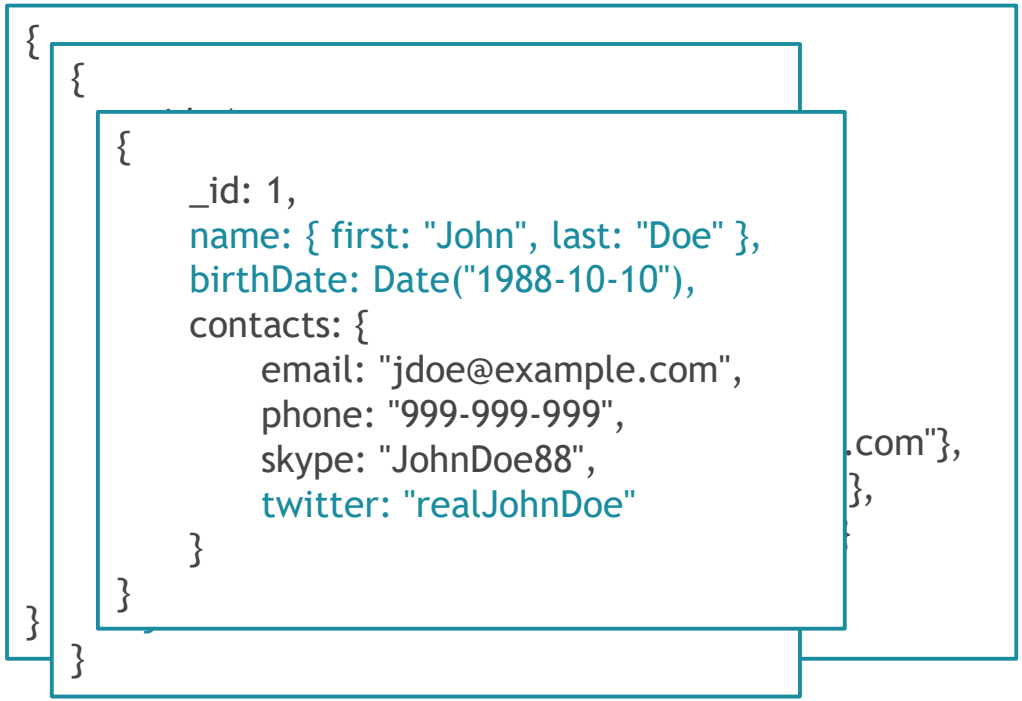
RELATIONAL MODEL

Person		
Id	**Name**	**Age**
1	John Doe	32
2	Jane Doe	47

Address		
Person_Id	**City**	**Street**
1	Neverwhere	Void Avenue 11
2	Othertown	Main street 9

Contact		
Person_Id	**Type**	**Contact**
1	Email	jdoe@example.com
1	Phone	999-999-999
1	Skype	JohnDoe88
2	Phone	123-456-789

DOCUMENT MODEL



- BSON = binary JSON
- Single entity
- Embedded struture
- No schema

...unless you want one

WHY DOCUMENTS?

- Support for unstructured data
- Embedded structure avoids need for JOINS
  - Horizontal scalability
- Flexible schema
  - Ease of development
- Map well to language structures like objects
  - No need for awkward mapping
  - Supports polymorphism

```
{
      _id: 1,
      name: { first: "John", last: "Doe" },
      birthDate: Date("1988-10-10"),
      contacts: {
            email: "jdoe@example.com",
            phone: "999-999-999",
            skype: "JohnDoe88",
            twitter: "realJohnDoe"
      }
      likes: ["sports", "food"]
}
```

QUERY SAMPLES

Query filter

```
> db.coll.find(
      { birthDate: { $gt: ISODate("1985-01-01") }}
)
```

Query on embedded field

```
> db.coll.find(
      { "contacts.email": "jdoe@example.com" }
)
```

Query on array element

```
> db.test.find(
      { "likes": "food" }
)
```

Add to array

```
> db.test.updateOne(
      { _id: 1},
      { $push: { likes: "music" }}
)
```

```
{
      _id: 1,
      name: { first: "John", last: "Doe" },
      birthDate: Date("1988-10-10"),
      contacts: {
            email: "jdoe@example.com",
            phone: "999-999-999",
            skype: "JohnDoe88",
            twitter: "realJohnDoe"
      }
      likes: ["sports", "food"]
}
```

DATA TYPES

BSON - binary JSON

- Converts to Extended JSON

Primitive types

- bool
- int, long
- double, decimal
- string
- regex
- date, timestamp
- binData

Embedded structures

- object
- array

Special types

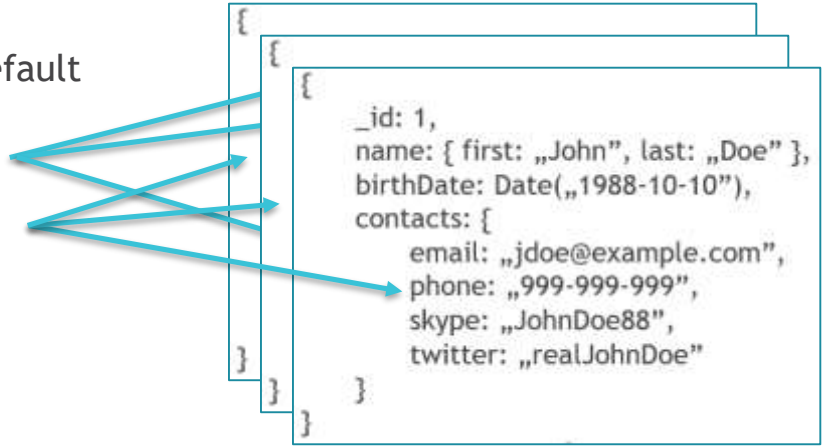
- objectId
- javascript
- null
- minKey
- maxKey

WHAT IS WHAT IN MONGODB

RDBMS	MongoDB
database	database
table	collection
row	document
column	field
index	index
primary key	primary key (_id)
table joins	embedded documents (or $lookup)
aggregation (e.g. group by)	aggregation pipeline

ATOMICITY & TRANSACTIONS

- Atomicity on document level
  - Including embedded subdocuments and arrays
- Multi-document operations are not atomic by default

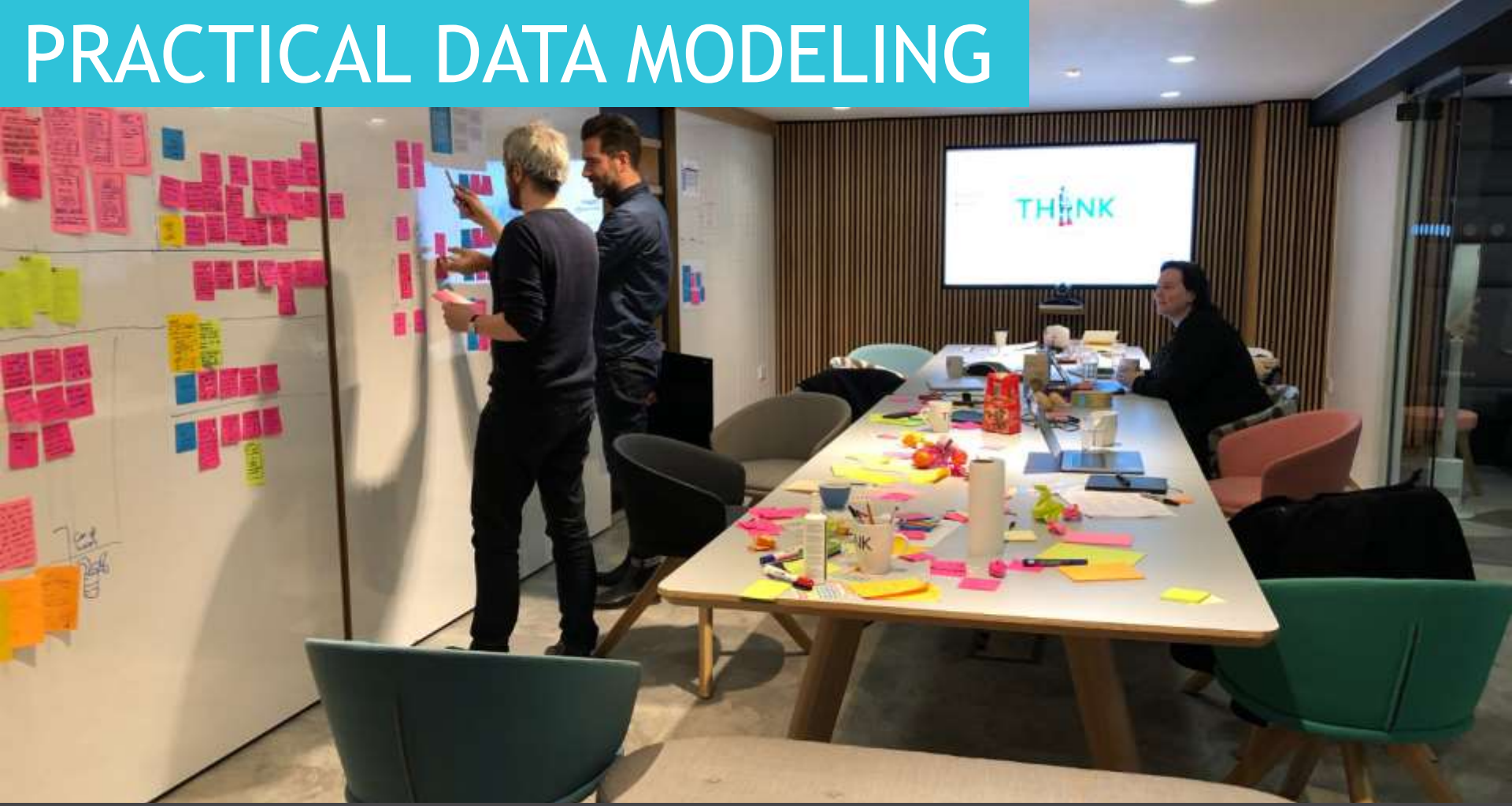


Multi document transactions

- Provide distributed ACID
- Transaction level consistency settings
- Has significant performance impact
- Has limitations
- Not as mature as in relational DBs

Avoid need for transactions with good data modeling if possible

PRACTICAL DATA MODELING



DENORMALIZED MODEL

- Avoids JOIN
- Atomic

Use

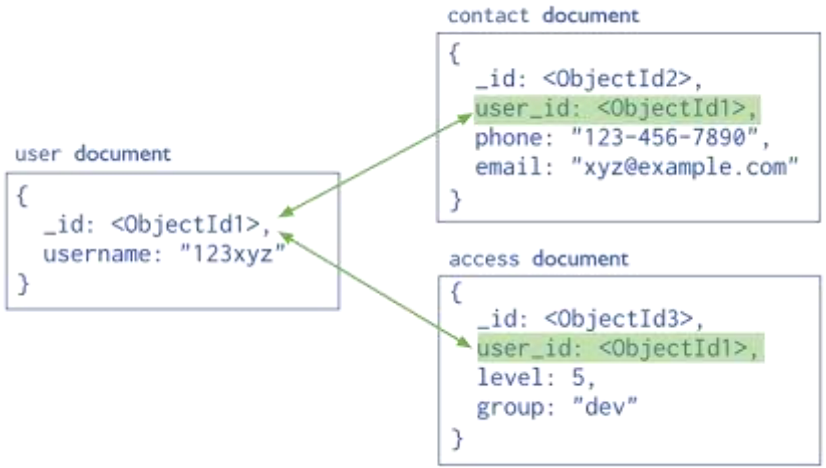
- Whenever you can ☺
- Contains relationship
  - No identity of its own
- Modified together

NORMALIZED MODEL

- Can JOIN via $lookup stage
- Needs transaction to be atomic

Use

- Complex many-to-many relationship
- Unbounded
- Embedding would lead to duplication without benefits
- Large, rarely needed data



SUBSET PATTERN

movie_details
_id
movie_id
full_plot
poster
full_cast
...

movie
_id
title
director
year
genre
plot_summary
reviews (top 3)
rating
comment

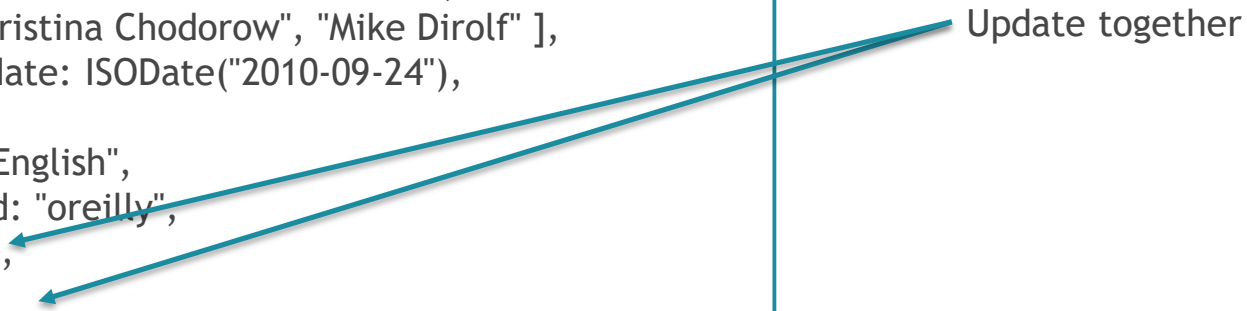
movie_review
_id
movie_id
rating
comment

DESIGN FOR ATOMIC UPDATE

```
{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly",
    available: 3,
    checkout: [
        { by: "userId5", date: ISODate("2021-02-15") }
    ]
}
```

DESIGN FOR ATOMIC UPDATE

```
{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly",
    available: 2,
    checkout: [
        { by: "userId5", date: ISODate("2021-02-15") },
        { by: "userId12", date: ISODate("2021-03-23") }
    ]
}
```



SCHEMA VALIDATION

```
db.createCollection("people", {
  validator: {
     $jsonSchema: {
        bsonType: "object",
        required: [ "name" ],
        properties: {
           name: {
              bsonType: "string"
           },
           year: {
              bsonType: "int",
              minimum: 2017,
              maximum: 3017
           },
...
```

DATA MODEL – TAKE AWAY

- Document model
- Embedded structures
  - Prefer denormalized structures
  - Relations are possible
- Flexible schema
  - Schema restrictions are optional
- Lends itself to scalability

ARCHITECTURE

AGENDA – ARCHITECTURE

1. Data Model
2. **Architecture**
3. Workshop
4. Exercise

- Replication
- Sharding
- Availability
- Consistency
- Durability

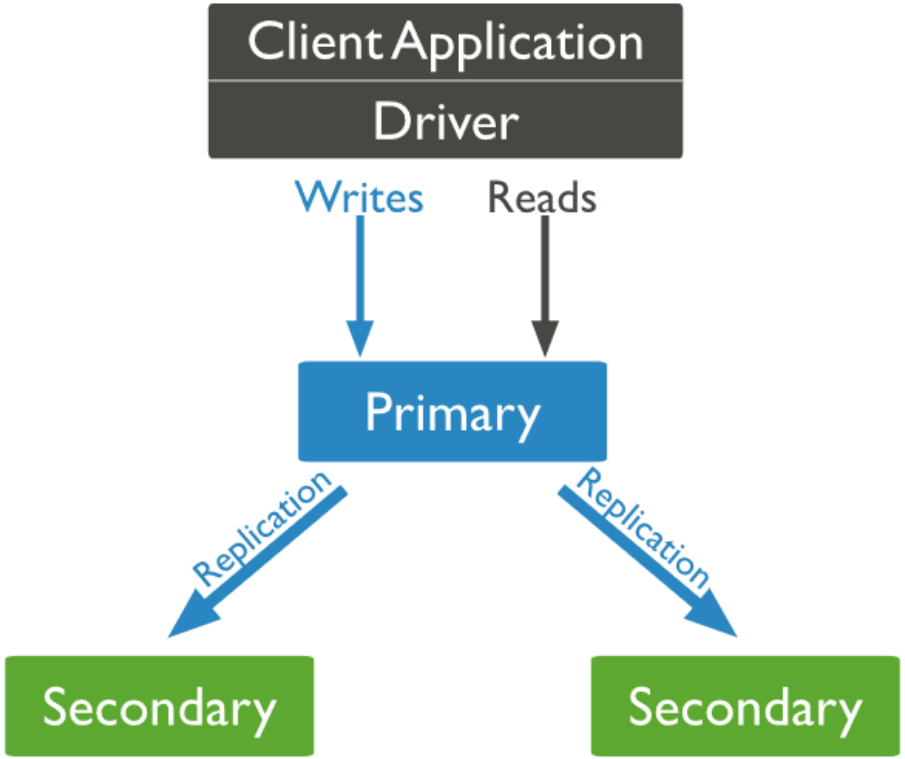
REPLICATION

Provides

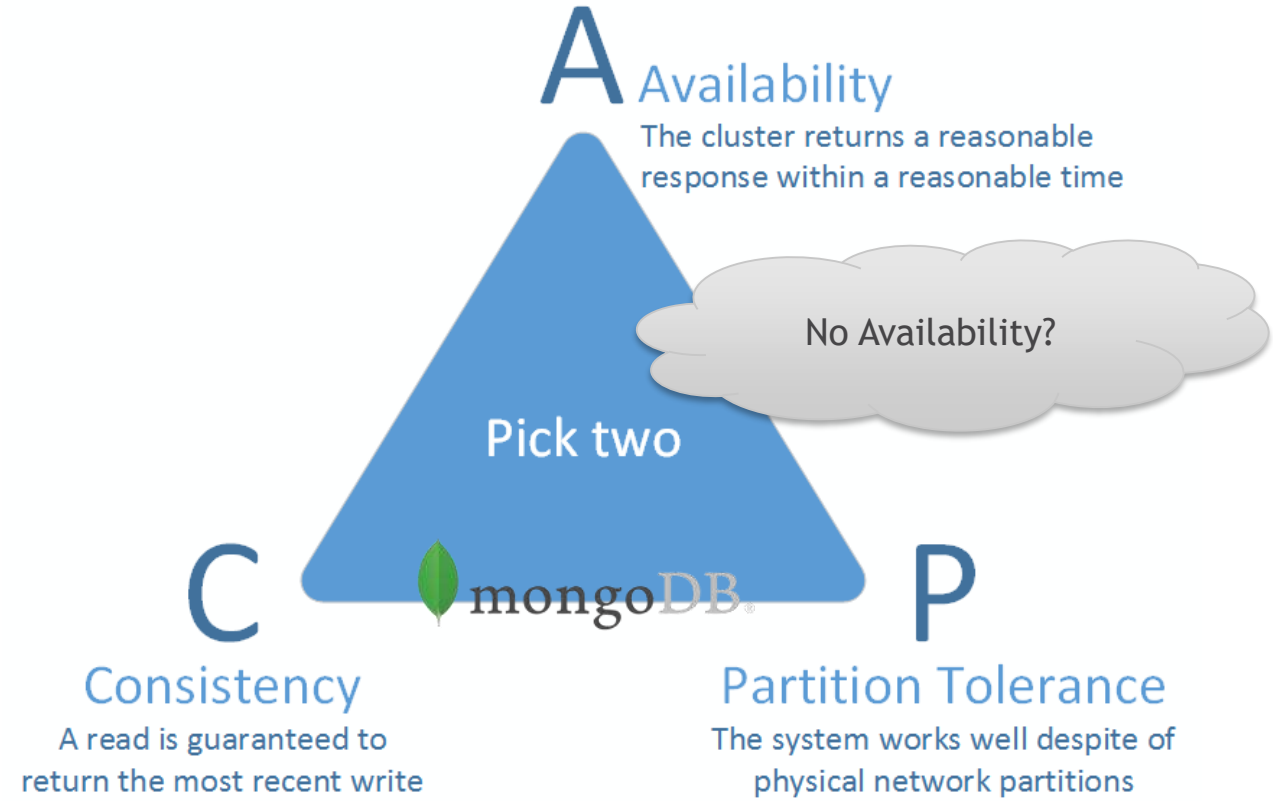
- Redundancy
- High availability

Replica sets

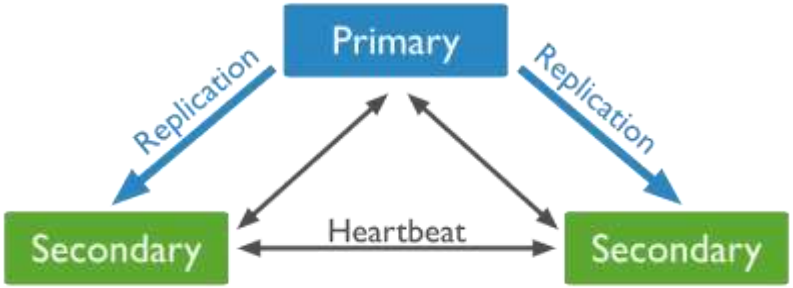
- Single Primary
- Several Secondaries
- All writes go to Primary
- Eventually all changes are replicated to secondaries

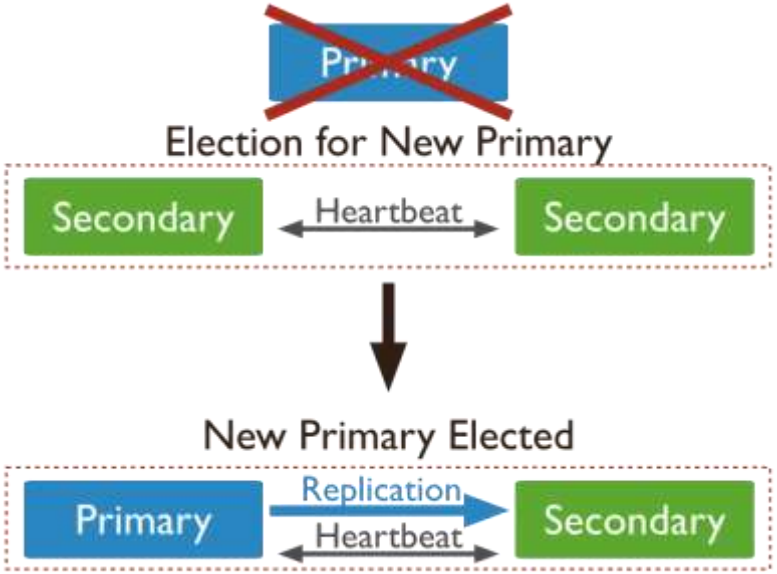


CONSISTENCY – CAP THEOREM



AUTOMATIC FAILOVER



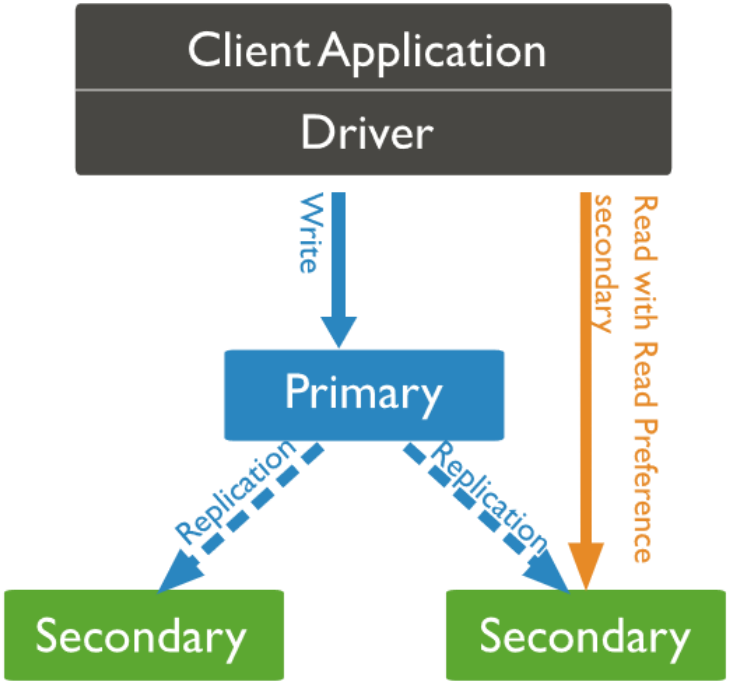


- Election takes ~12 seconds (median with default settings)
- No writes until new primary elected
- What about reads?

READ PREFERENCE

- **primary (default)**
- allow non-primary reads
  - primaryPreferred
  - secondary
  - secondaryPreferred
  - nearest

Main goal is better availability, not throughput scaling.



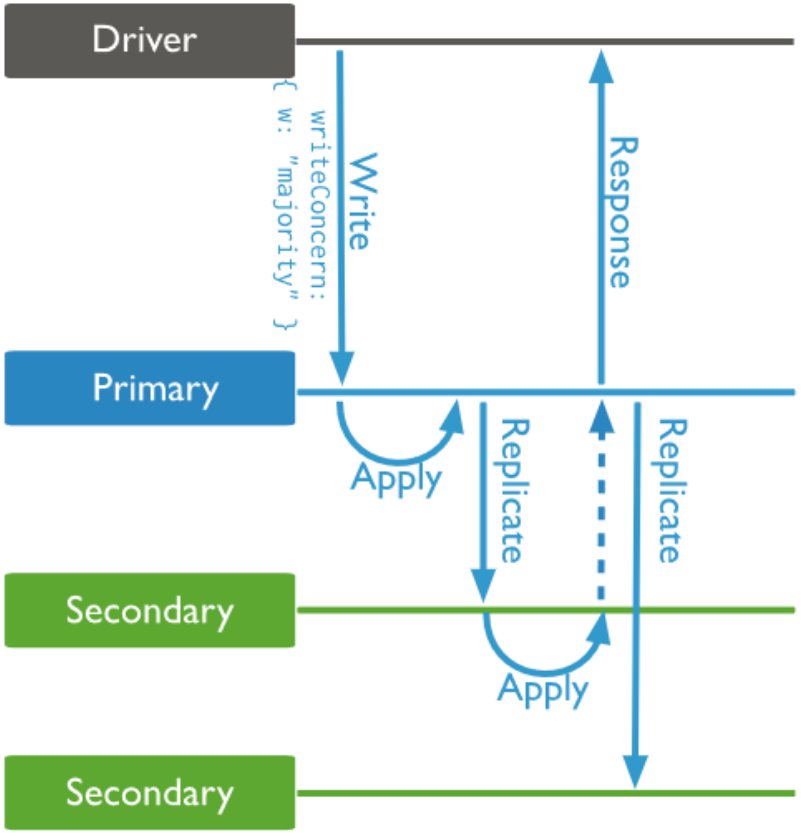
READ AND WRITE CONCERN

Write concern

- **w: 1 (default)**
- **w: majority**
- w: 0
- w: N

Read concern

- **local**
- **majority**
- available
- linearizable
- snapshot



DURABILITY

Single machine

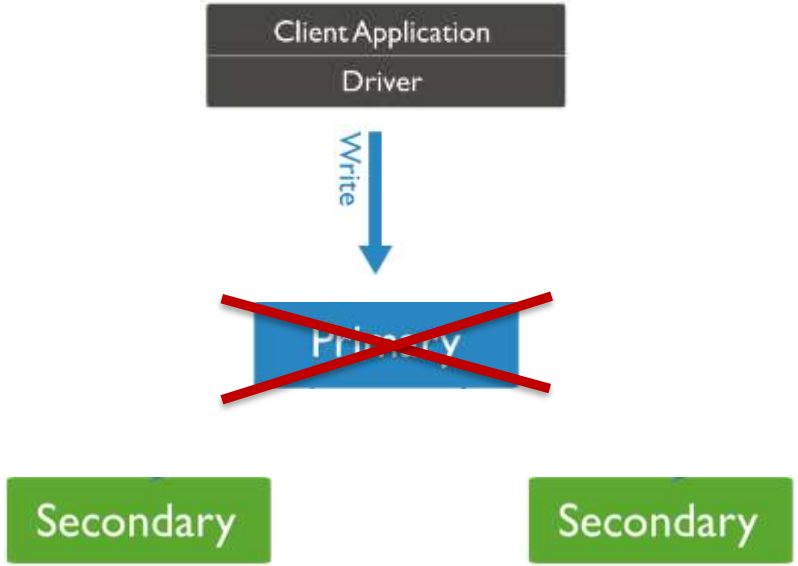
- Works on documents in memory
- Periodic checkpointing
- Journal
  - write ahead log

Replication provides durability

...except in some scenarios

e.g. primary failover with w:1

Use w:majority if want to avoid write loss



SHARDING

Shard

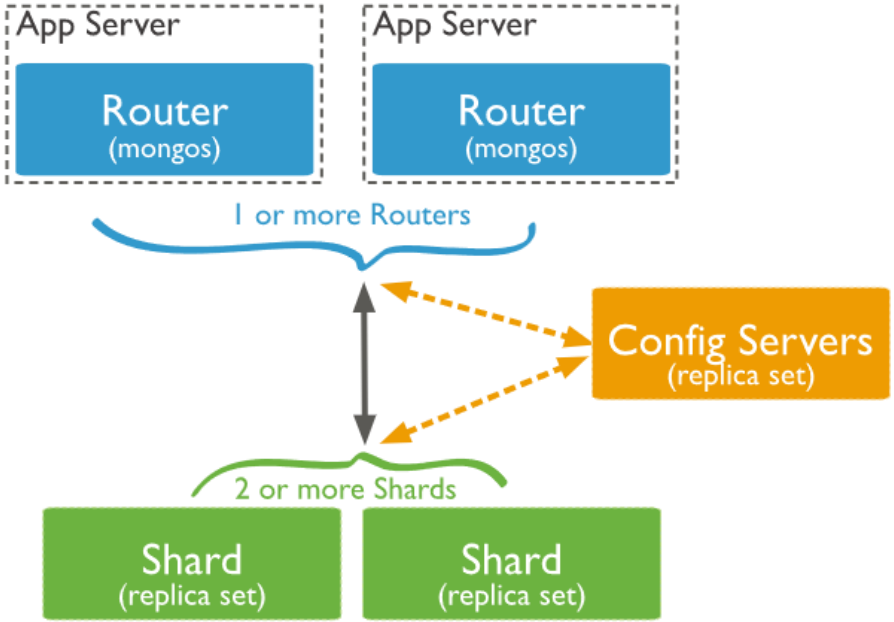
- Stores a subset of sharded data
- No direct client connection

Config server

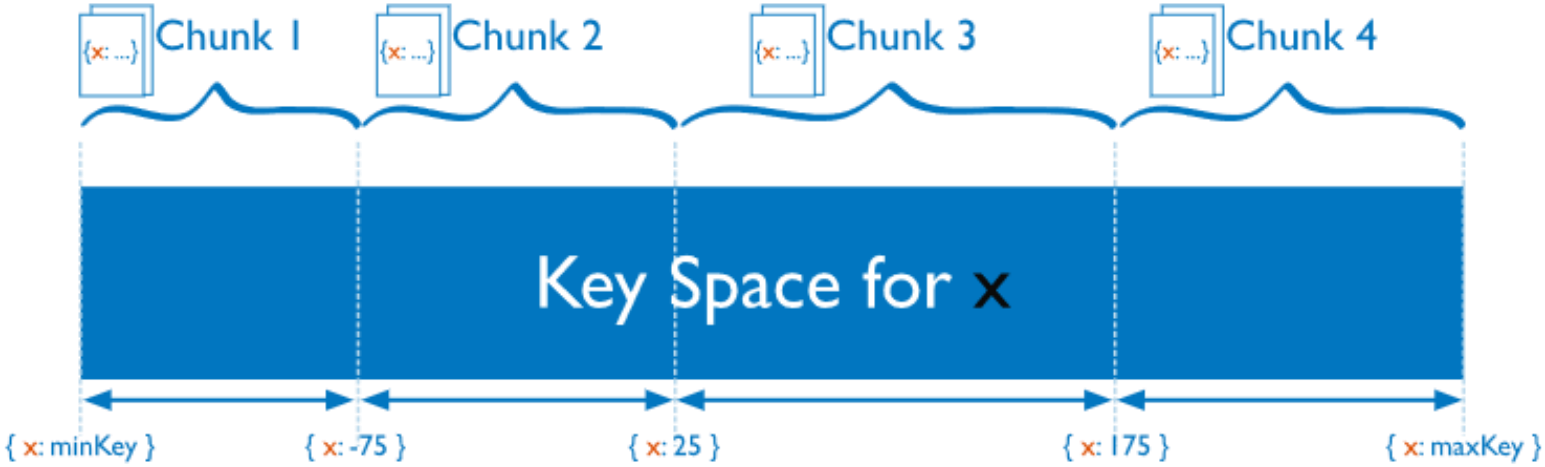
- Holds metadata for sharded cluster

Router

- Clients connect to routers
- Distributes queries to shards
- Caches metadata from config server
- Stateless



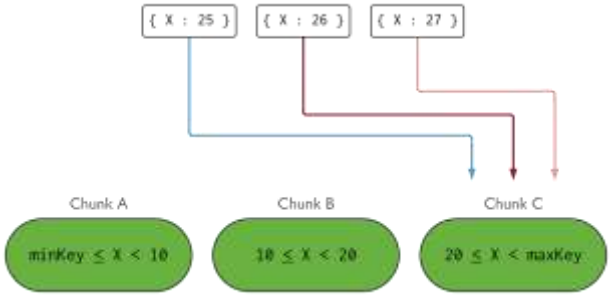
SHARD KEY


{x: ...}
Chunk 1
{x: ...}
Chunk 2
{x: ...}
Chunk 3
{x: ...}
Chunk 4
Key Space for x
{ x: minKey }
{ x: -75 }
{ x: 25 }
{ x: 175 }
{ x: maxKey }

CHOOSING THE SHARD KEY

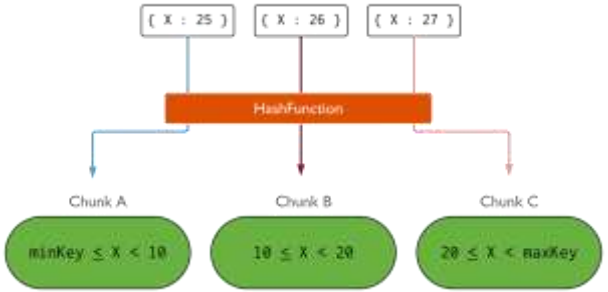
Data distribution

- Cardinality
- Frequency
- Monotonicity

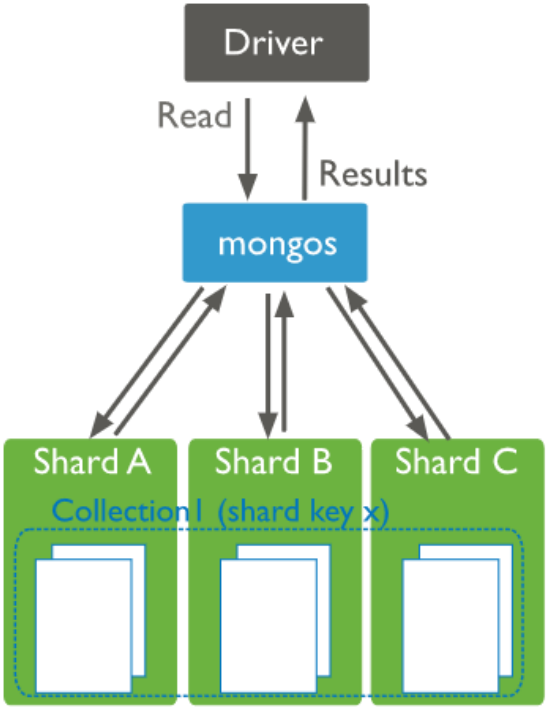


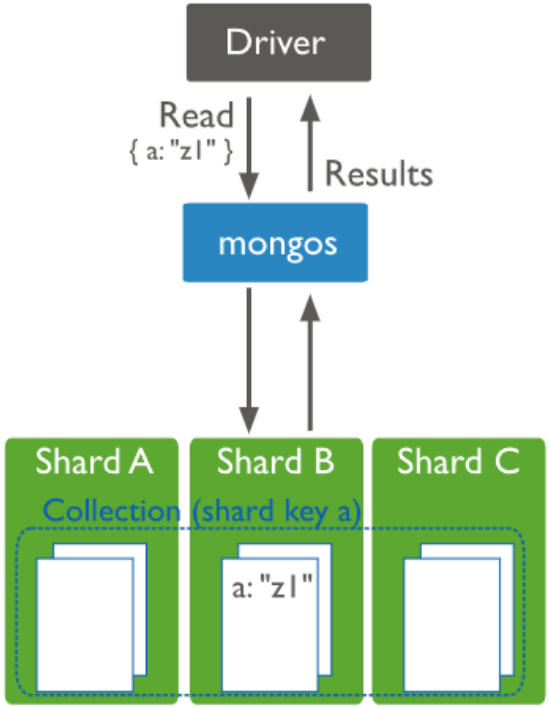
Sharding strategies

- Ranged sharding
- Hashed sharding

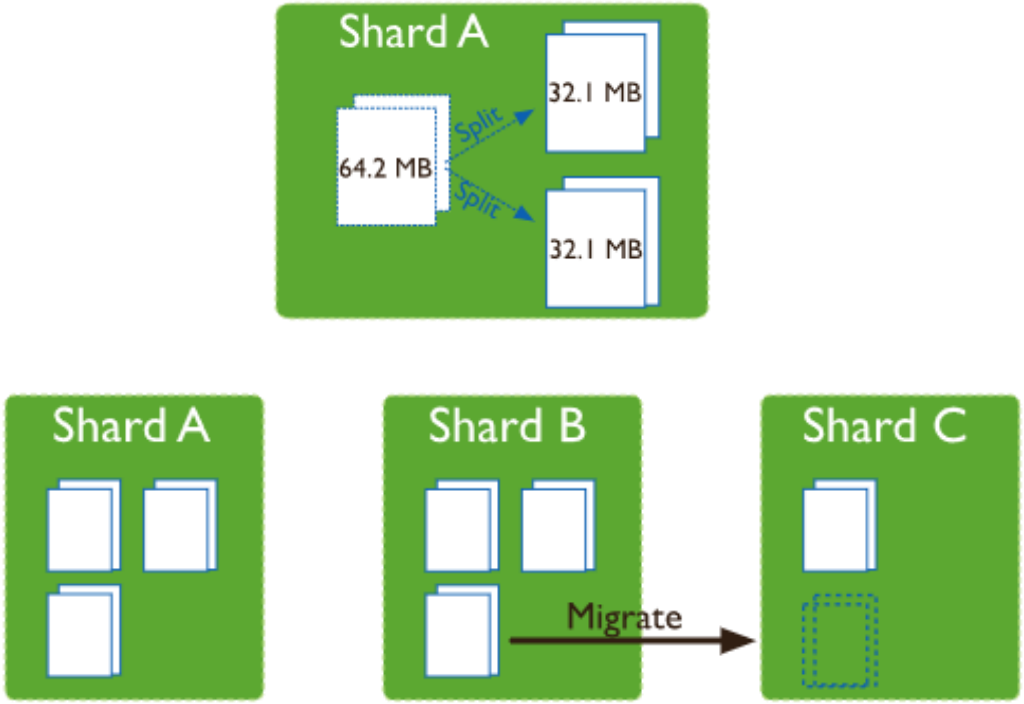


BROADCAST VS TARGETED OPERATIONS





CHUNKS SPLITTING AND BALANCER



ARCHITECTURE – TAKE AWAY

- Single Primary
- Consistency first, but tunable
- High availability (with short downtimes)
- Scalable through sharding

BREAK

WORKSHOP

AGENDA

1. Data Model
2. Architecture
3. **Workshop**
4. Exercise

- Introduction
- CRUD operations
- Aggregations
- Advanced

ENVIRONMENT

Source available on

david-szabo-epam/mongodb-presentation

DEMO #2 – CRUD OPERATIONS

- Insertion
- Queries
- Updates
- Deletion

PRIMARY KEY & COLLECTIONS

Primary key

- is name _id
- is mandatory
- is created when omitted
- is the first column
- is unique
- is indexed

Collections

- No need for DDL operations
- Explicit creation command for special purpose collections

```
db.people.insertOne({
      name: { first: "Jim", last: "Jones" },
      birthDate: new Date("1977-10-10"),
      likes: ["food", "sports"]
})
```

```
{
      "acknowledged" : true,
      "insertedId" : ObjectId("6057b272bfc40396c19d6db4")
}
```

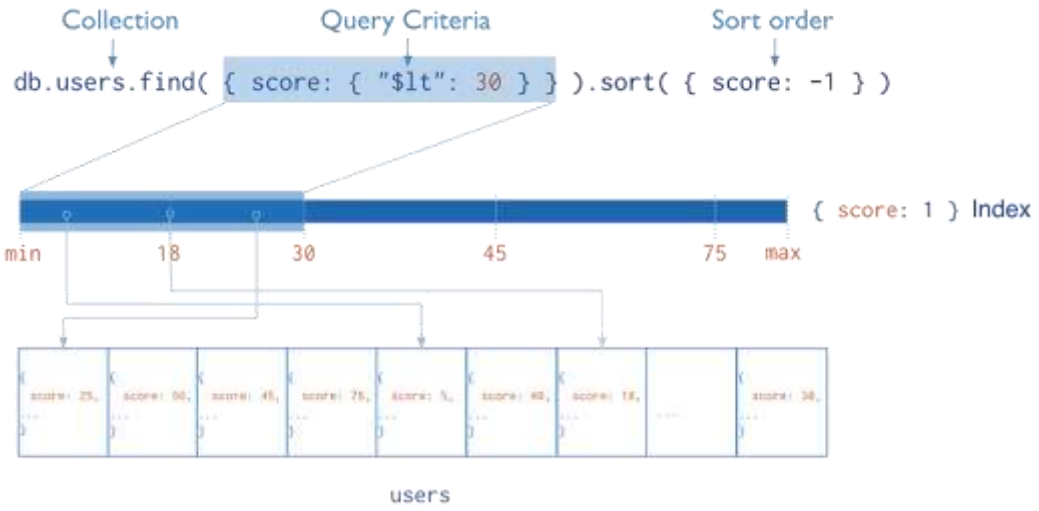
INDEXES

Considerations

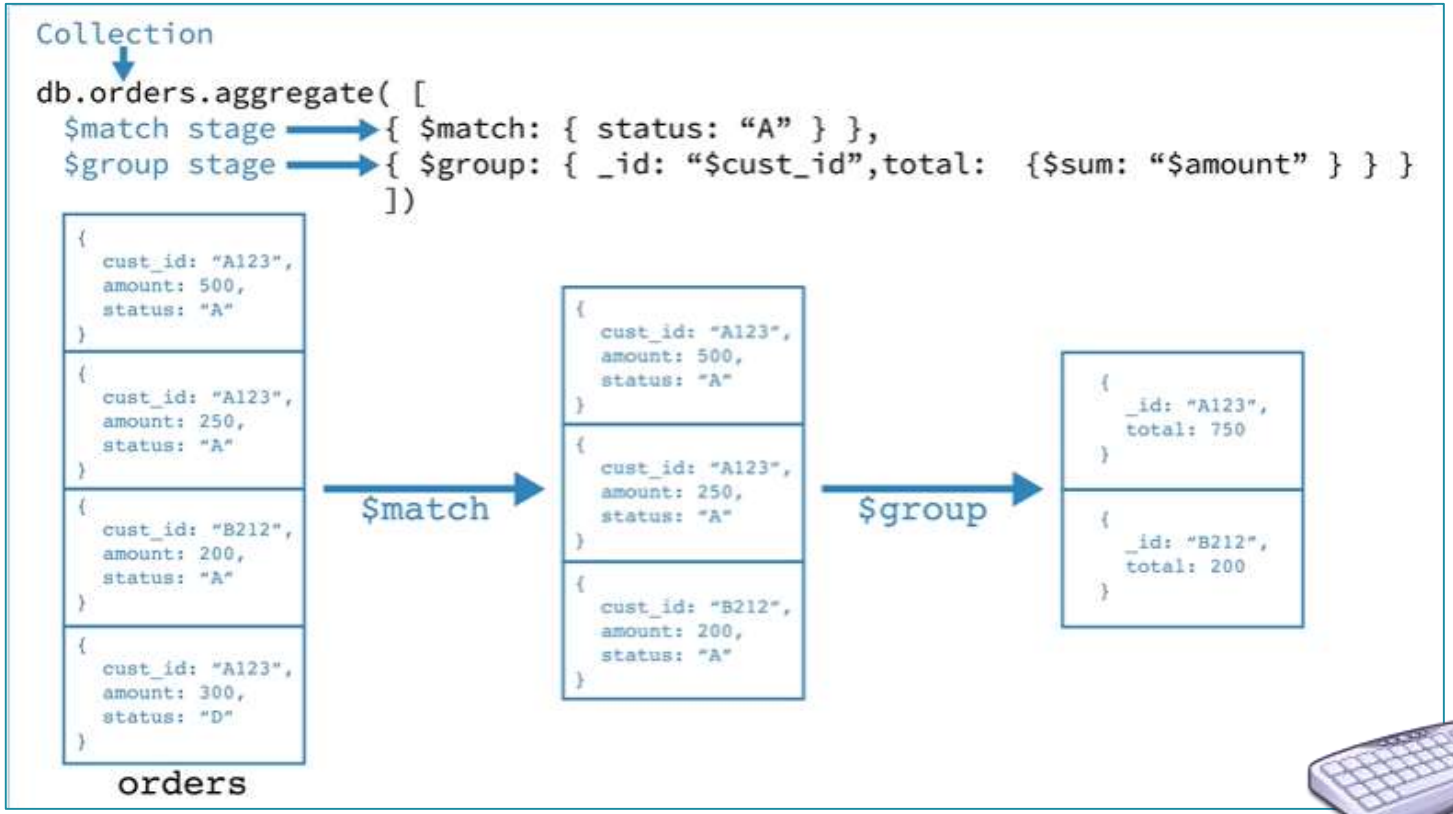
- Improves read performance
- Decreases write performace
- Memory

Used by

- Filters
- Sorting
- Aggregation pipelines
- Sharding



AGGREGATION PIPELINES



AGGREGATION PIPELINE – MAPPING TO SQL

SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit

SQL	MongoDB
SUM()	$sum
COUNT()	$sum, $sortByCount
join	$lookup
SELECT INTO TABLE	$out
MERGE INTO TABLE	$merge
UNION ALL	$unionWith

AGGREGATION PIPELINE - OPTIMIZATION

Considerations

- Some automated optimization
- Optimize yourself
- Early filtering
- Indexes
- Memory
- Sharding

```
{ $addFields: {
    maxTime: { $max: "$times" },
    minTime: { $min: "$times" }
} },
{ $project: {
    _id: 1, name: 1, times: 1,
    maxTime: 1, minTime: 1,
    avgTime: {
       $avg: ["$maxTime",
              "$minTime"] }
} },
{ $match: {
    name: "Joe Schmoe",
    maxTime: { $lt: 20 },
    minTime: { $gt: 5 },
    avgTime: { $gt: 7 }
} }
```

```
{ $match: { name: "Joe Schmoe" } },
{ $addFields: {
    maxTime: { $max: "$times" },
    minTime: { $min: "$times" }
} },
{ $match: {
    maxTime: { $lt: 20 },
    minTime: { $gt: 5 }
} },
{ $project: {
    _id: 1, name: 1, times: 1,
    maxTime: 1, minTime: 1,
    avgTime: {
       $avg: ["$maxTime",
              "$minTime"] }
} },
{ $match: { avgTime: { $gt: 7 } } }
```

VIEWS AND MATERIALIZED VIEWS

Views

- Defined by aggregation pipeline
- Computed on query
- Read only
- Inherits the characteristics of the underlying collections

Materialized views

- Regular collections
- Aggregation pipeline with $merge
- Can be run regularly

```
db.createCollection(
  "<viewName>",
  {
    "viewOn" : "<source>",
    "pipeline" : [<pipeline>],
    "collation" : { <collation> }
  }
)
```

```
db.orders.aggregate( [
   { $group: { _id: "$quarter", purchased: { $sum: "$qty" } } },
   { $merge : {
      into: "quarterlyreport",
      on: "_id",
      whenMatched: "merge",
      whenNotMatched: "insert"
   } }
])
```

ADVANCED QUERIES

Text searches

- Tokenization
- Stemming
- Stopwords
- Ranking

Geospatial searches

- 2dsphere and 2d
- Distance
- Intersection & containment

```
db.stores.find( { $text: { $search: „database book" } } )
```

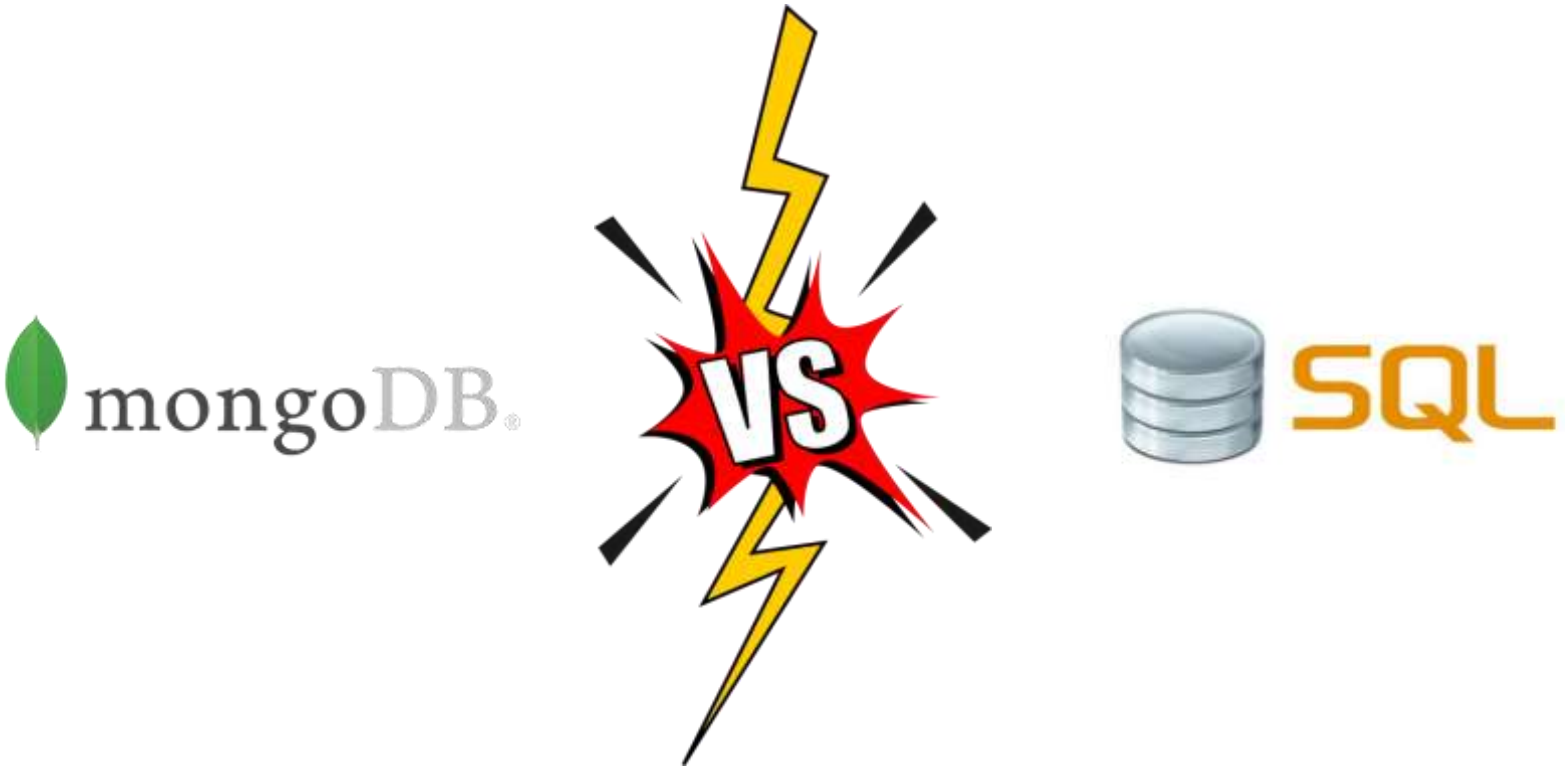
```
db.places.find({
    location: { $near: {
        $geometry: { type: "Point",
            coordinates: [ -73.9667, 40.78 ]
        },
        $minDistance: 1000, $maxDistance: 5000
    }}
})
```

For more advanced features consider a dedicated search engine (Elasticsearch, Solr)

EXERCISE

CONCLUSIONS

MONGODB VS RDBMS



MONGODB VS RDBMS – CLOSING THE GAPS

- Schema validation
- $lookup for JOINS
- Multi-document transactions

- JSON support

Still they are built for different purposes

MONGODB VS RDBMS

- Unstructured data
- Unstable schema
- Fits the document model
- Need scalability

- Schema is known and unchanging
- Have a lot of complex queries spanning across tables
- Strong ACID guarantees across documents

MONGODB VS CASSANDRA

- Document data model
- Flexible schema
- Better consistency
- Better ad-hoc query support

- Columnar data model
- High availability and distributed writes
- Scales better
- Write-heavy load

MONGODB – TAKE AWAY

- Embedded structures with rich query language
- Flexible schema
- Replica sets with single primary and secondaries
- Configurable consistency and availability
- Scalable through sharding
- Support for RDBMS features with tradeoffs


Questions?

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR

Streaming & Cloud

May 2023

Attila Szűcs

attila_szucs@epam.com
Data Engineering, Architecture

WHERE WE ARE?

NoSQL

Big Data (Batch)

Both use distributed computing to store & process large amount of data

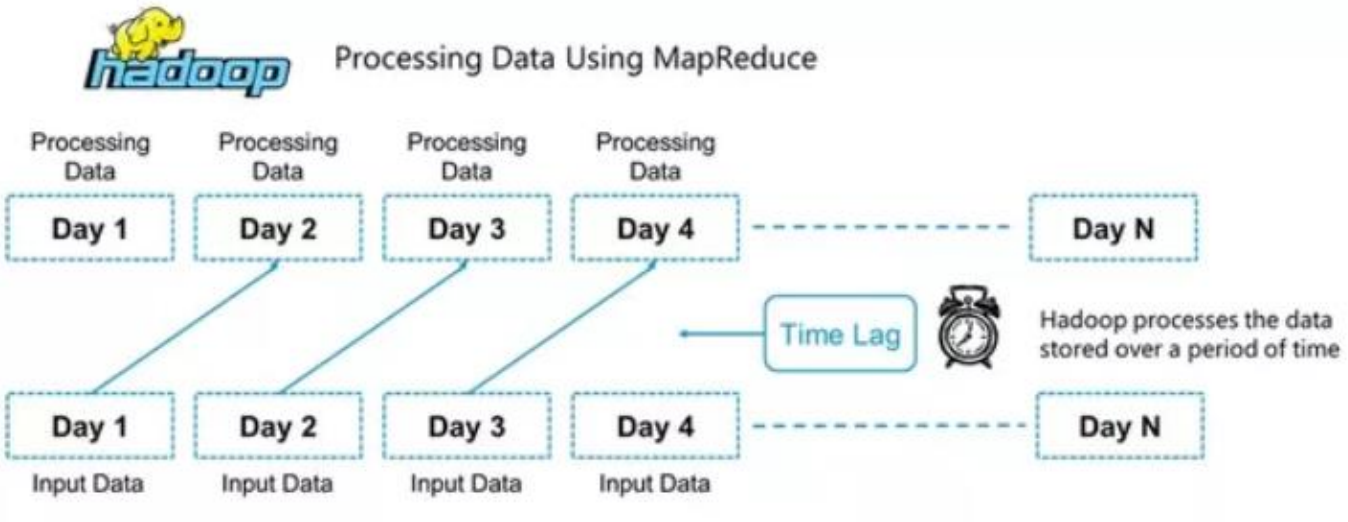
- **Solution for:** How to store the data so that we can run real-time queries
- **Latency:** milliseconds (real-time)

- **Solution for:** How to run ad-hoc queries on any (semi-structured) data
- **Latency:** minutes, potentially hours

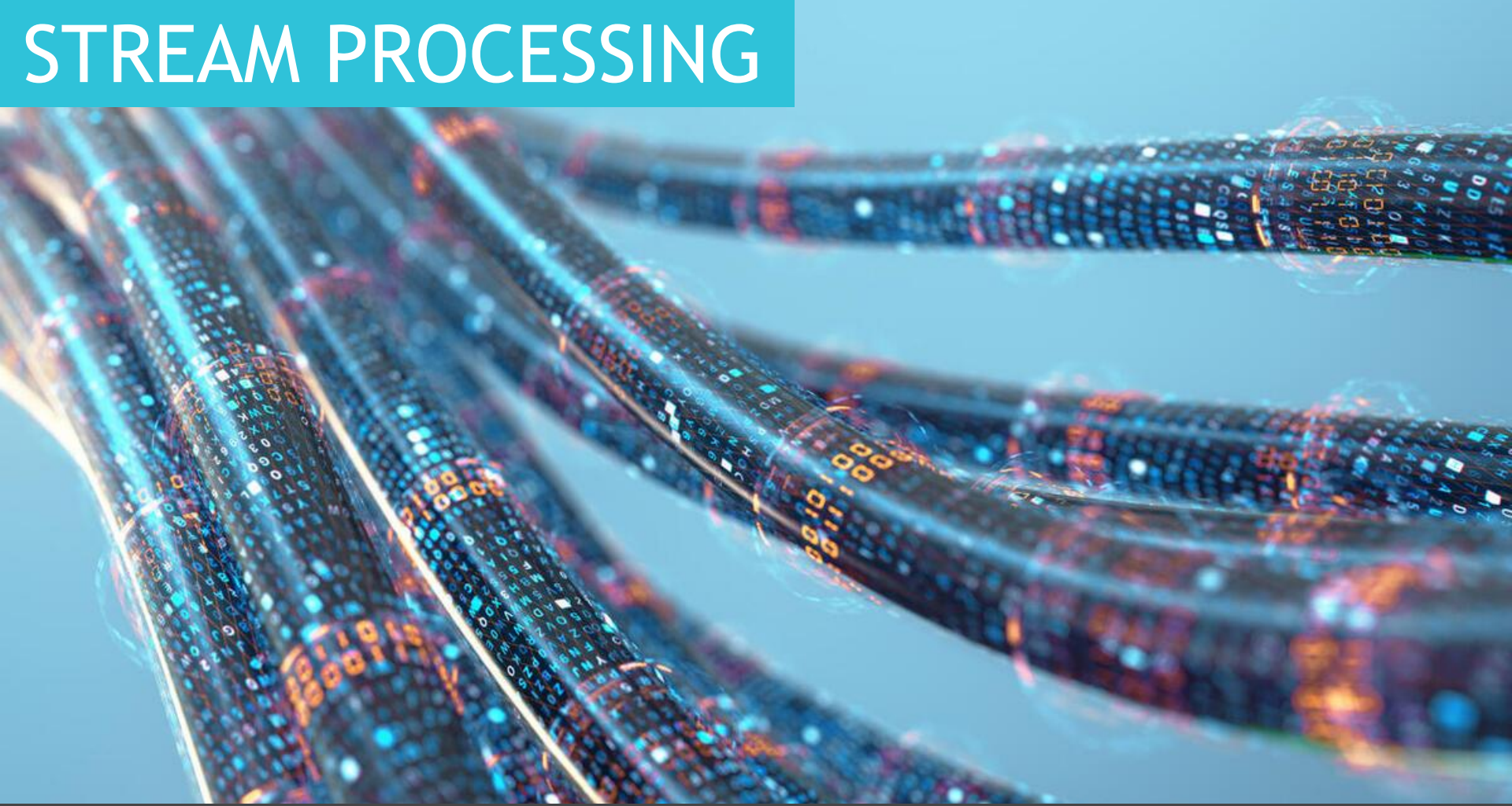
BATCH PROCESSING

Processing the data collected over a time period

- Traditional Big Data is batch processing
- Results are not real-time; we need to wait for the next “batch”



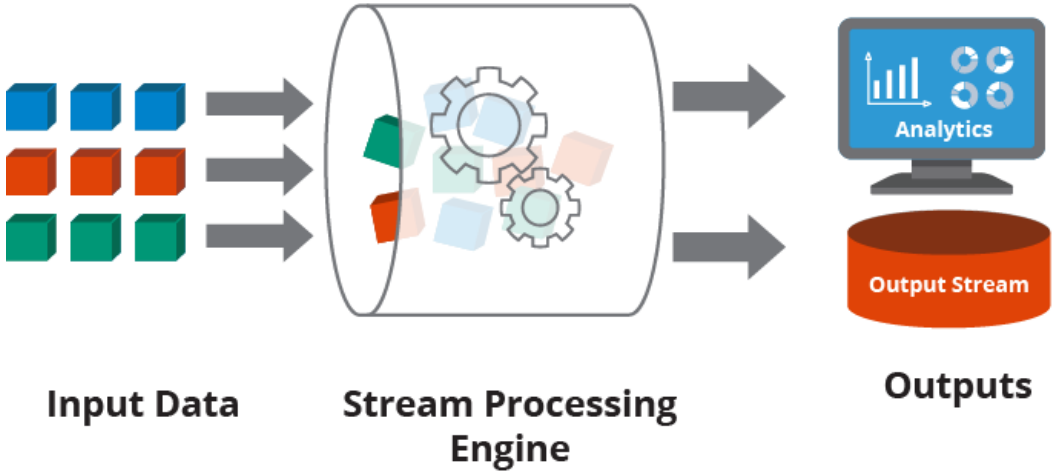
STREAM PROCESSING



STREAM PROCESSING

Definition:

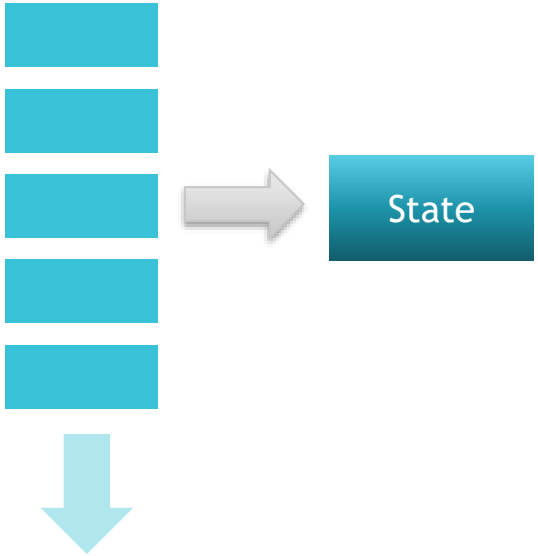
Stream processing is the practice of taking action on a series of data at the time the data is created.



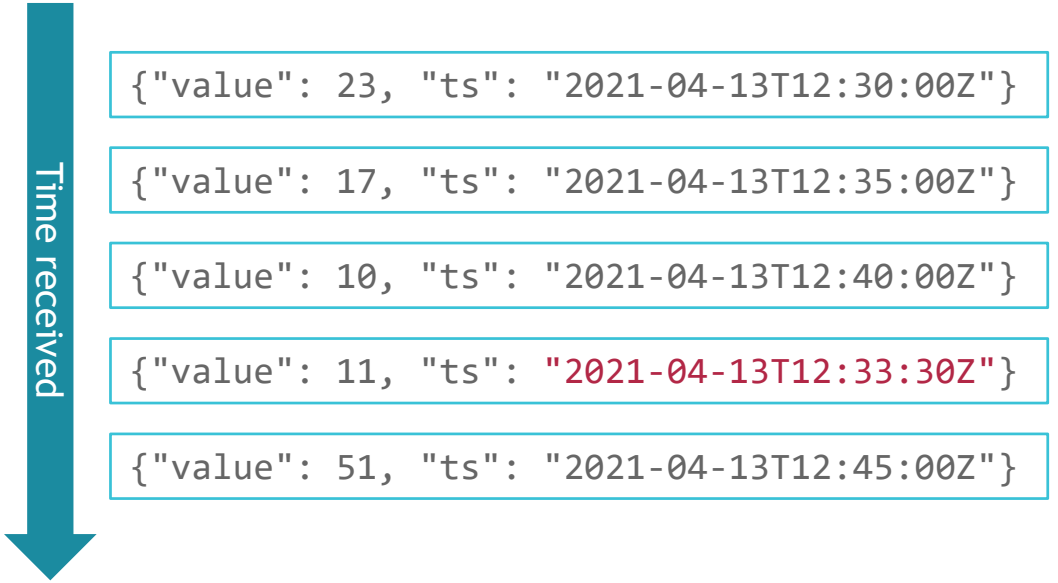
An overview on Hazelcast

STREAMING CHALLENGES

State management



Event time vs processing time; late events

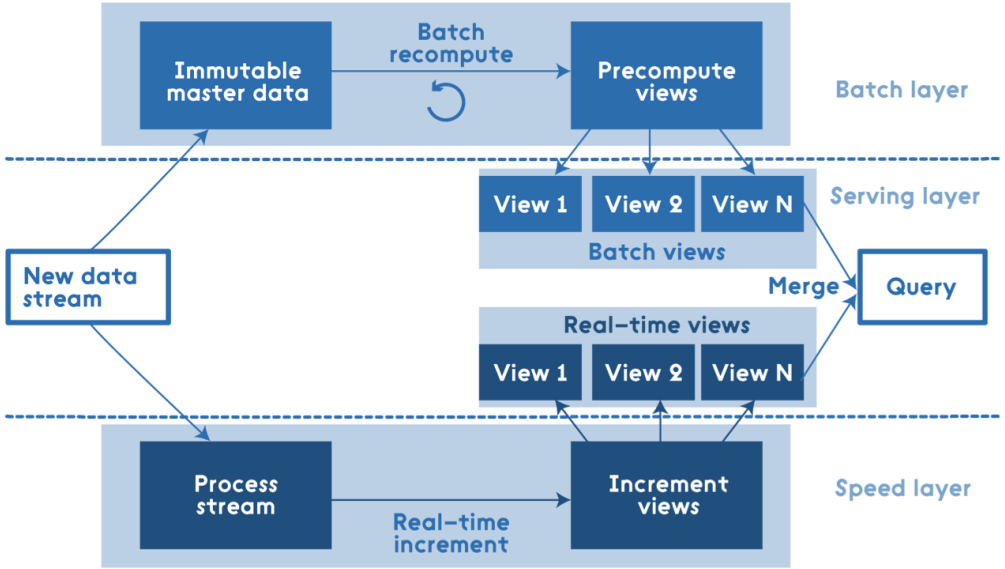


Traditionally it was considered in-accurate and unreliable

LAMBDA ARCHITECTURE

Attempts to balance latency, throughput, and fault-tolerance by

- using **batch processing** to provide comprehensive and accurate views of batch data,
- while simultaneously using real-time **stream processing** to provide views of online data.



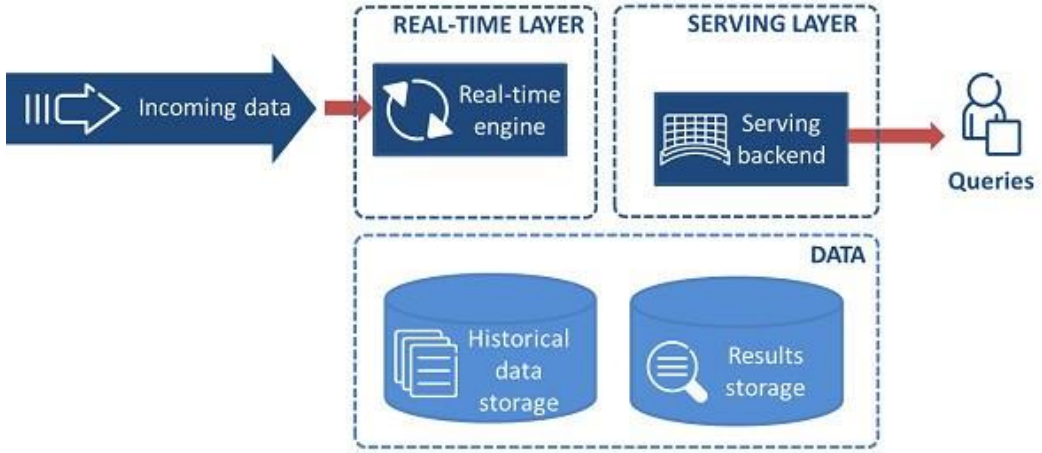
Disadvantages

- Two codebases, need to be in sync
- Complex architecture, specialized tools

KAPPA ARCHITECTURE

Is a software architecture used for processing streaming data. The main premise is that you can

- perform both real-time and batch processing with a single technology stack.



- **Everything is a stream**
  Batch operations become a subset of streaming
- **Keep it short and simple** (KISS) principle.
  A single analytics engine is required
  Coding and maintenance are simpler
- **Immutable data sources**
  Data source is persisted, and views are derived
  State can always be recomputed from source
- **Replay functionality**
  Computations and results can evolve by replaying the historical data from a stream

AGENDA

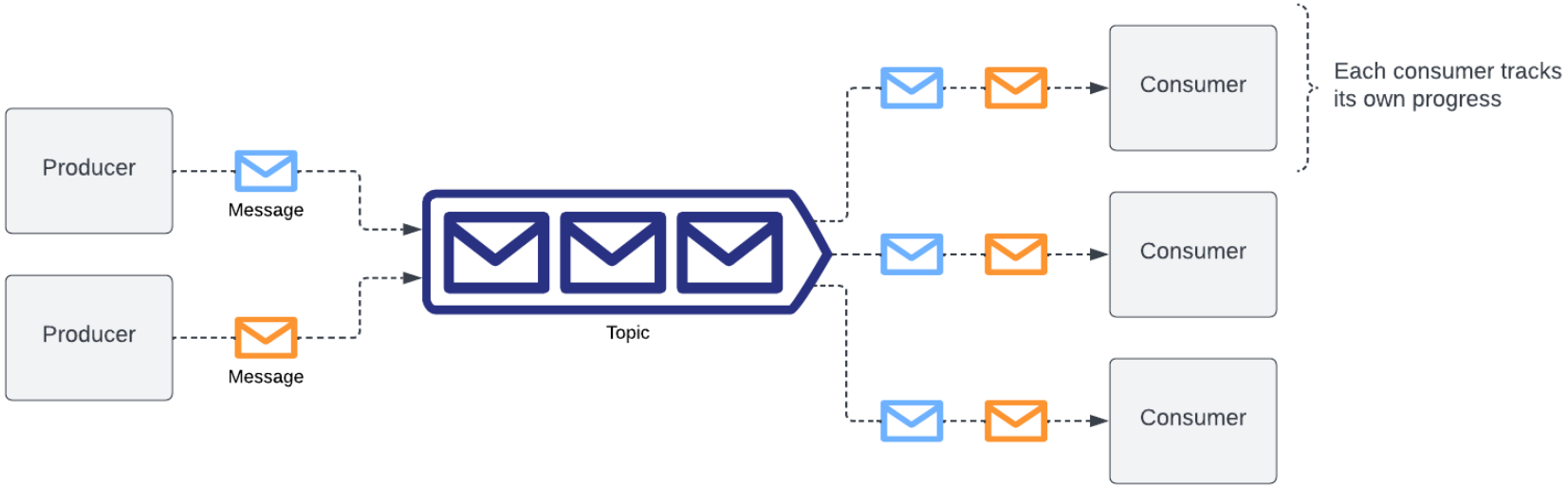
Handling events

- **Apache Kafka** is a distributed data store optimized for ingesting and processing streaming data in real-time.
- **Simplified definition:**
  Persistent & scalable messaging platform

Processing events

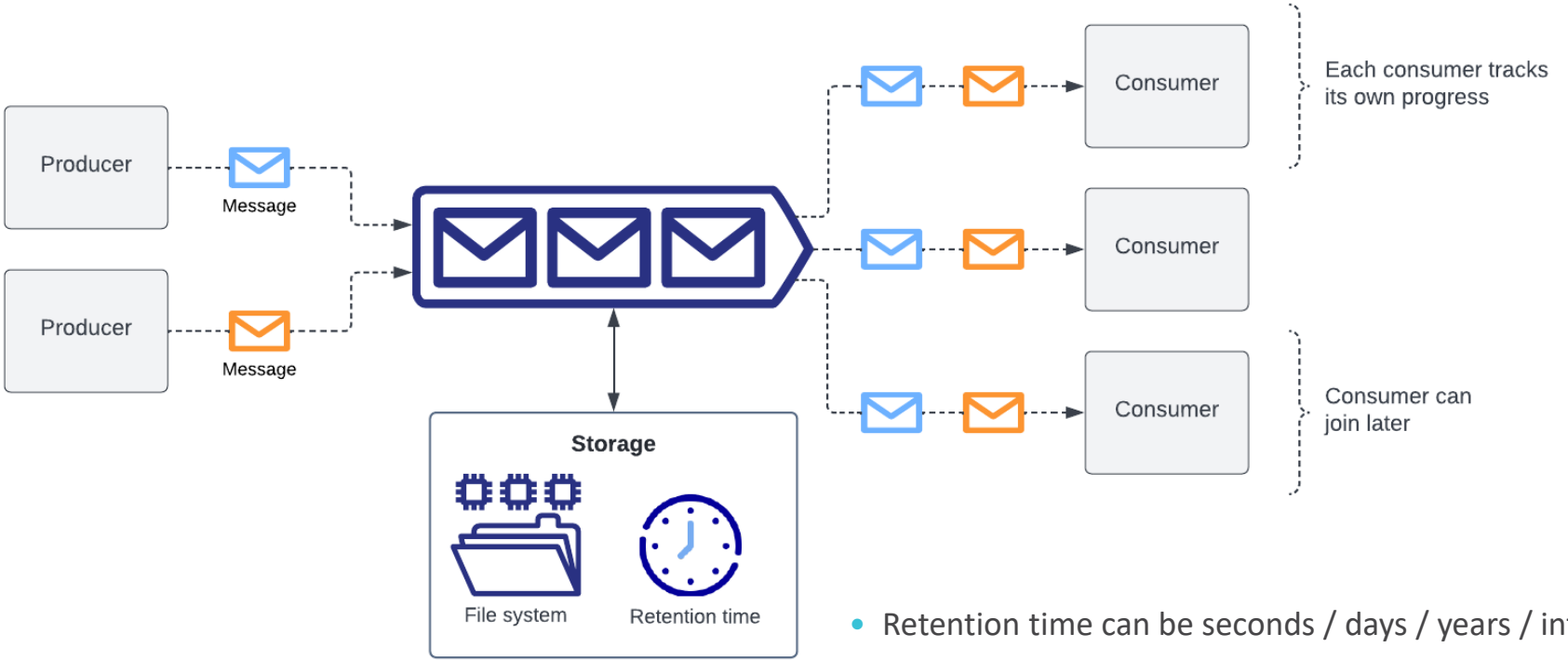
- **Spark streaming** is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- **Spark streaming workshop**

KAFKA – MESSAGING PLATFORM



- Decouples source and target systems
- Reduces integration complexity
- Each consumer receives all messages (by default)

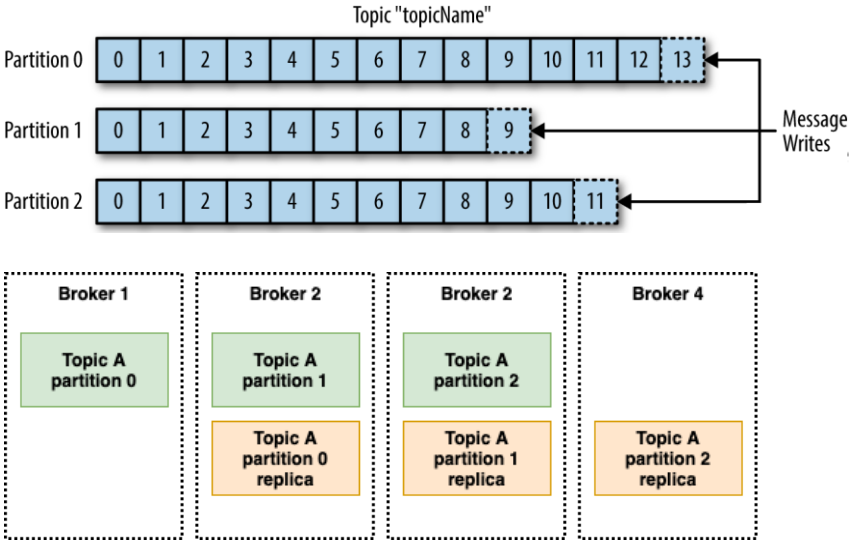
KAFKA – PERSISTENT STORAGE



- Retention time can be seconds / days / years / infinite
- New consumers can be added later

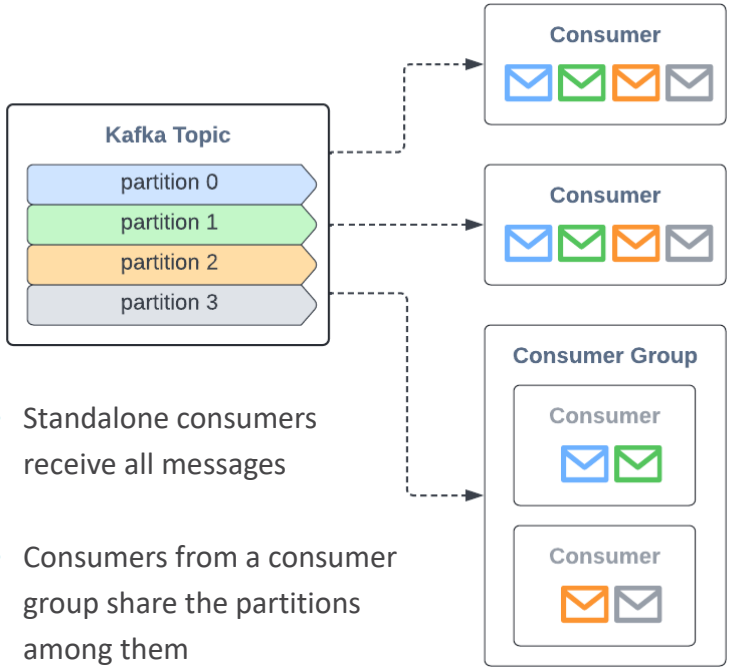
KAFKA – SCALABILITY

Scaling Storage



- Topics can be partitioned and replicated just like NoSQL tables
- Kafka cluster nodes are called “brokers”

Scaling Consumers



- Standalone consumers receive all messages
- Consumers from a consumer group share the partitions among them

AGENDA

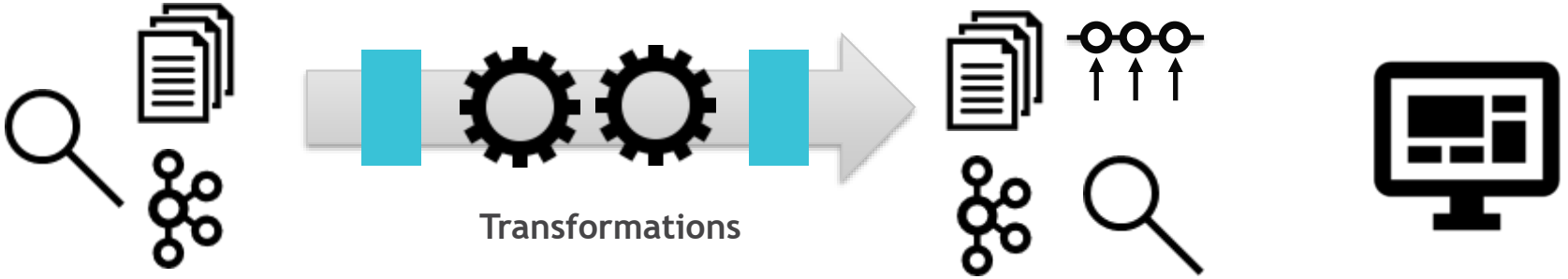
Handling events

- **Apache Kafka** is a distributed data store optimized for ingesting and processing streaming data in real-time.
- **Simplified definition:**
  Persistent & scalable messaging platform

Processing events

- **Spark streaming** is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- **Spark streaming workshop**

STREAM PROCESSING – CONCEPTS



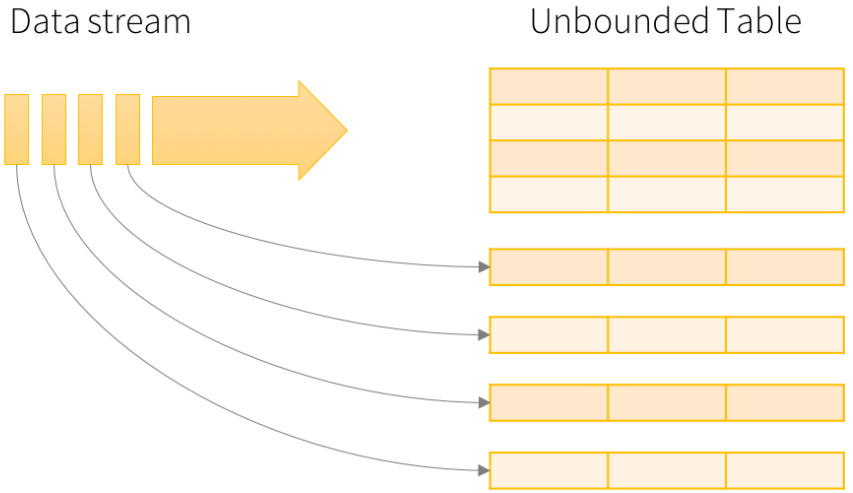
Source
- Files
- Kafka
- Socket (testing)
- Rate (testing)
- …

Triggers
- Default (microbatch)
- Fixed interval
- Once
- Continuous (experimental)

Sink
- Files
- Kafka
- Foreach
- Console (testing)
- **…**

Output modes
- Append (default)
- Update
- Complete

SPARK – STRUCTURED STREAMING



new data in the
data stream
=
new rows appended
to a unbounded table

Data stream as an unbounded table

- Spark's DataFrame API can be used
- Streams are (unbounded) data frames
- Operations on unbounded data frames result in unbounded data frames
- The same API can be used for both batch and stream processing

STRUCTURED STREAMING

Dataframe API

Transformations

- Simple transformations
- Aggregations
- Joins
  - Stream-Static
  - Stream-Stream

Limitations apply

```
// input stream
val input = spark.readStream
    .schema(schema)
    .json("/path/to/folder")

// transformation
val countStream = input
    .where("size > 10")
    .groupBy("category")
    .count()

// output stream
val outputStream = countStream.writeStream
  .outputMode("complete")
  .format("console")

// start streaming app
outputStream.start()
```

DEMO – SPARK STRUCTURED STREAMING

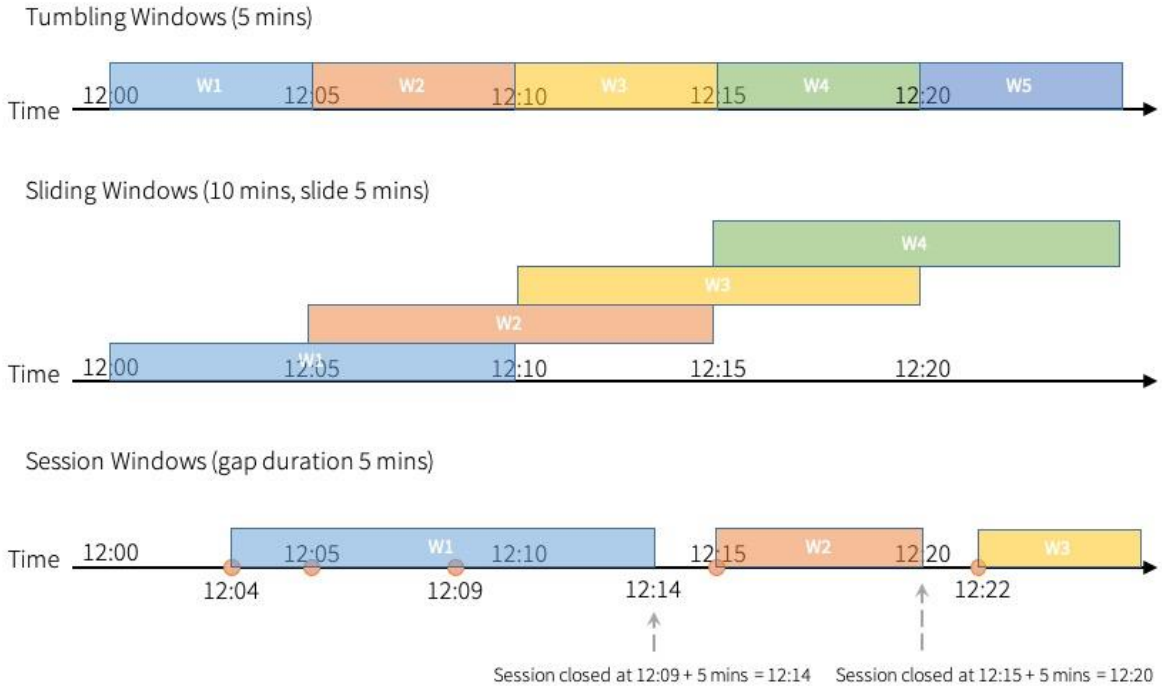
Source code available on

GitHub

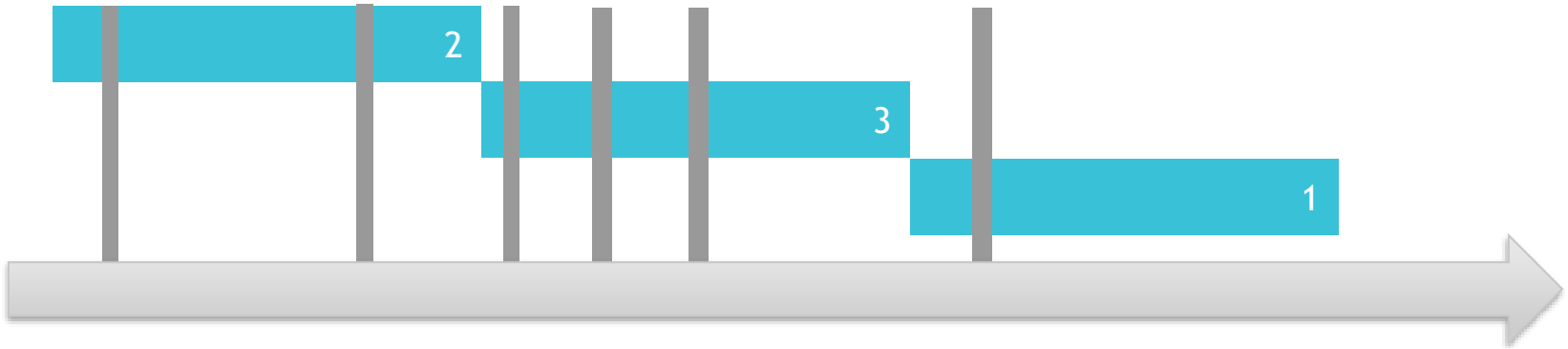
medvekoma/streaming-workshop

EVENT TIME & TIME WINDOWS

- Event time != processing time
- Event time is usually a field in the data
- Often, we group the data by event time windows
- Window types
  - Tumbling windows
  - Sliding windows
  - Session windows



EVENT TIME WINDOWS – TUMBLING WINDOWS

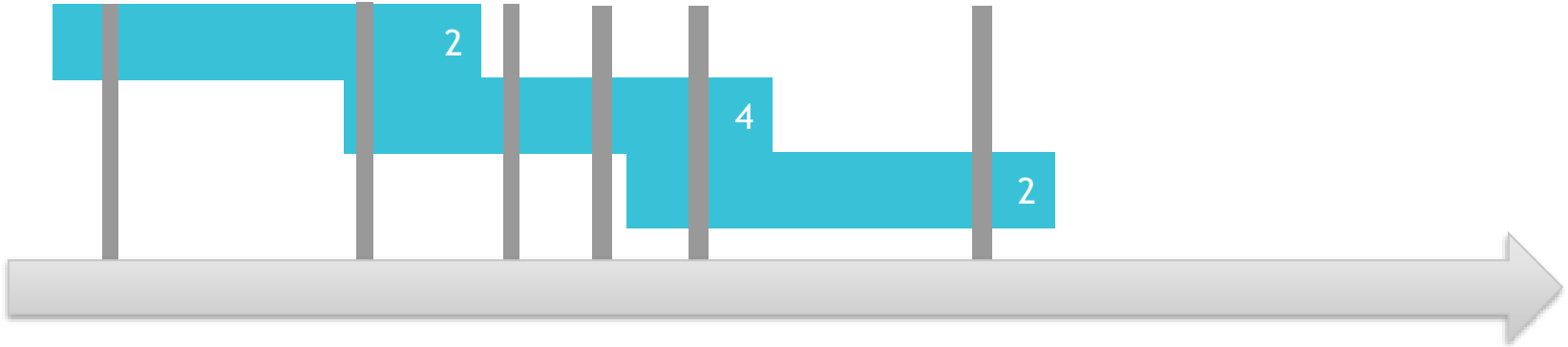


Window types

- Tumbling window

```
streamingDF
  .groupBy(
    window(col("event_time"), "15 minutes"))
  .count()
```

EVENT TIME WINDOWS – SLIDING WINDOWS

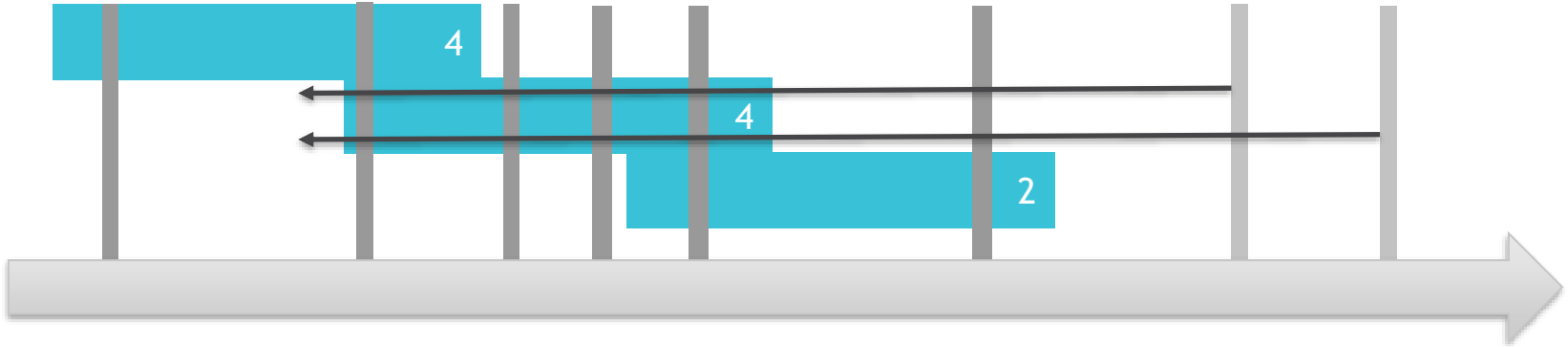


Window types

- Tumbling window
- Sliding window

```
streamingDF
  .groupBy(
    window(col("event_time"), "15 minutes", "10 minutes"))
  .count()
```

EVENT TIME WINDOWS – LATE EVENTS



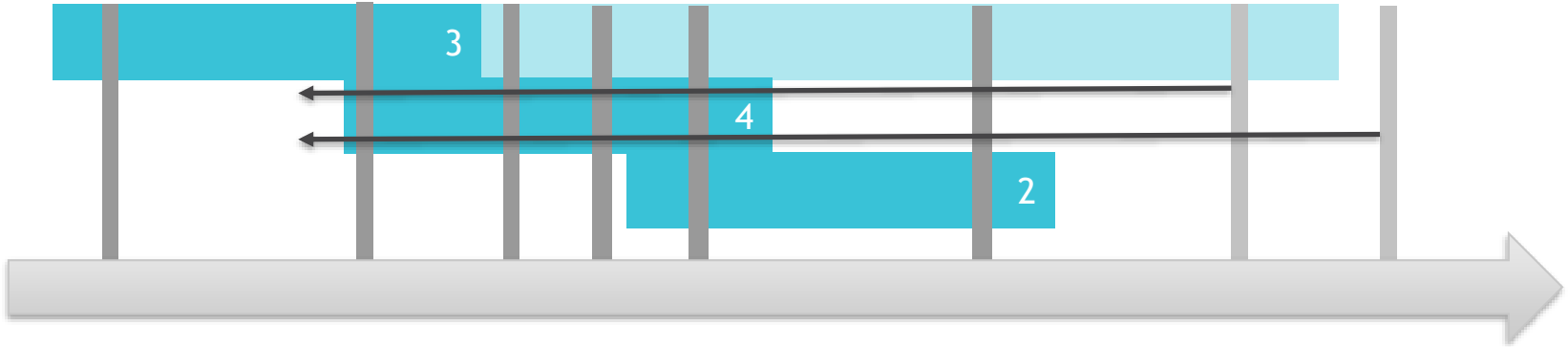
Window types

- Tumbling window
- Sliding window

Late events

```
streamingDF
  .groupBy(
    window(col("event_time"), "15 minutes", "10 minutes"))
  .count()
```

EVENT TIME WINDOWS – WATERMARKS



Window types

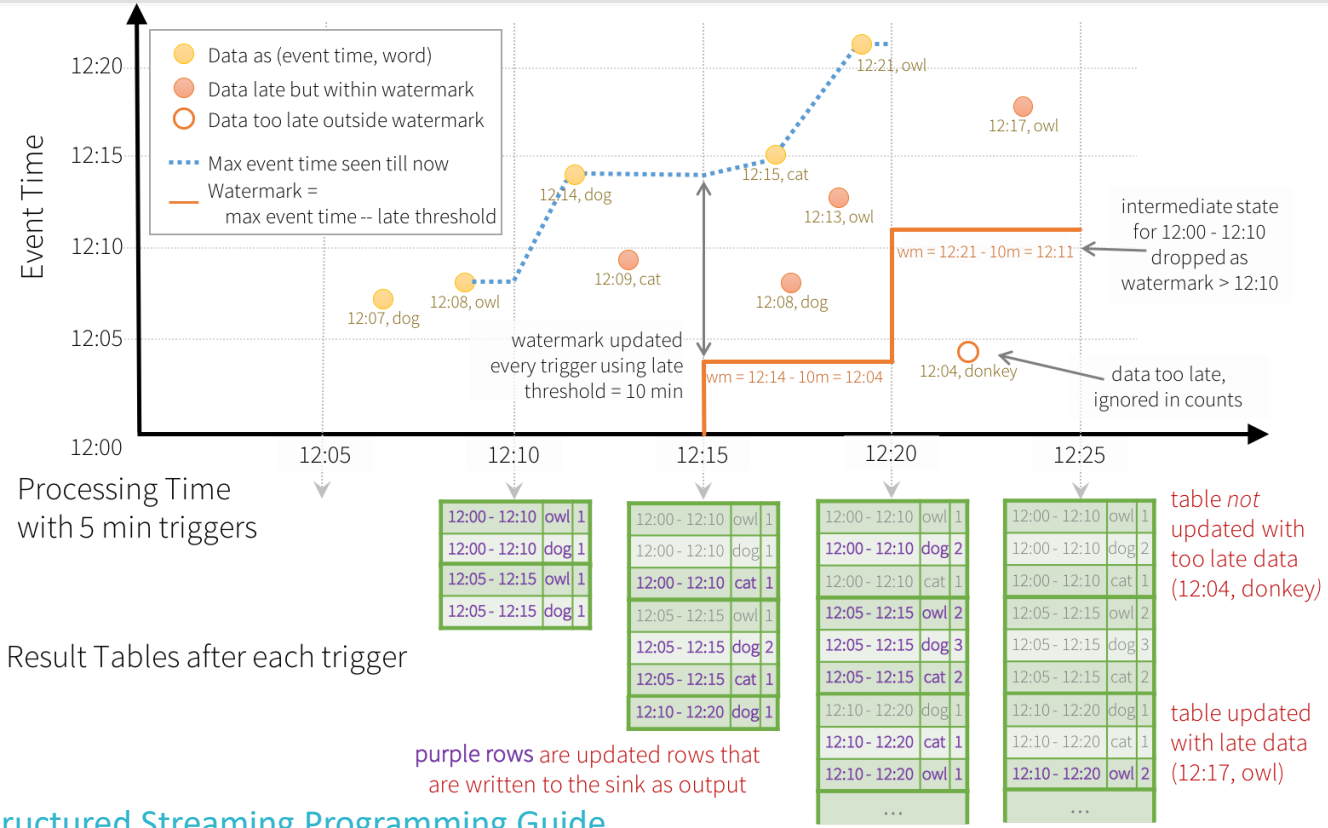
- Tumbling window
- Sliding window

Late events

- Watermarks

```
streamingDF
  .withWatermarks("event_time", "30 minutes")
  .groupBy(
    window(col("event_time"), "15 minutes", "10 minutes"))
  .count()
```

LATE EVENTS AND WATERMARKS



More info: [Structured Streaming Programming Guide]

STREAMING – TAKE AWAY

- Apache Kafka is a persistent, scalable messaging platform
- Structured Streaming is „Dataframe but streaming”
- A stream is an unbounded table
- The same API for batch and streaming
- Event time != processing time
- Late events & watermarks


Questions?

CLOUD COMPUTING

CLOUD COMPUTING

Cloud computing:

- Delivery of computing services over the internet (servers, storage, databases, software, etc.)
- ”Renting” IT services over the internet

Types

- **Infrastructure as a Service (IaaS):** Rent resources like servers, networking, data centers
- **Software as a Service (SaaS):** Rent software applications
- **Serverless:** Allows developers to create applications without thinking about infrastructure

Advantages

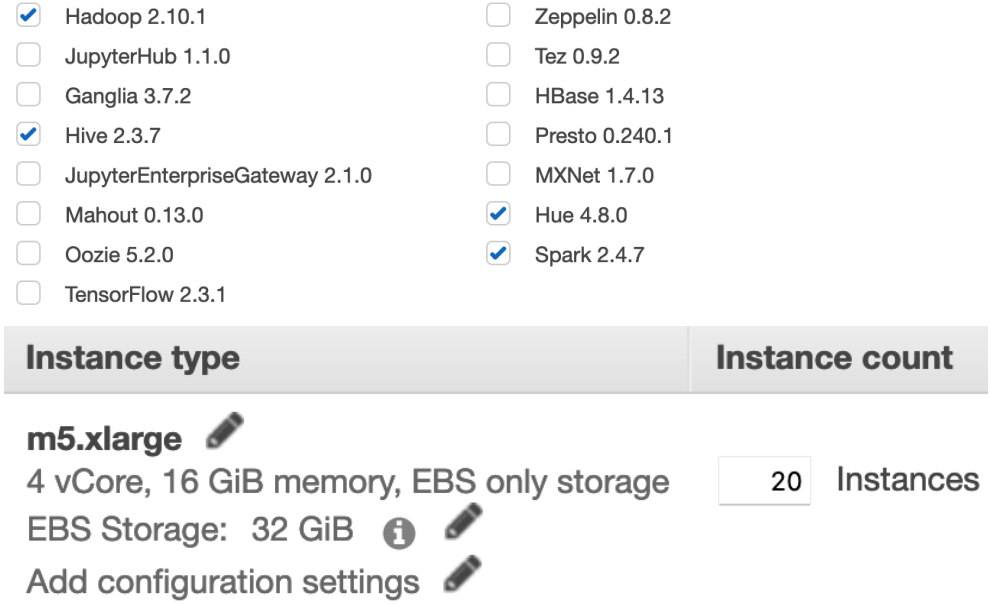
- Scalable infrastructure
- Globally accessible resources
- Reliable IT services
- Robust security

CLOUD – HADOOP AS A SERVICE

amazon EMR

Google Cloud DataProc

Microsoft Azure HDInsight



Advantages

- Managed service, less operational costs
- Quick provisioning, on demand clusters
- No up-front costs, pay as you go
- Auto-scaling

Things to consider

- Limited configuration
- Country regulations
- Cost (+10-25% on top of infrastructure cost)

CLOUD STORAGE

Object Storage in the Cloud

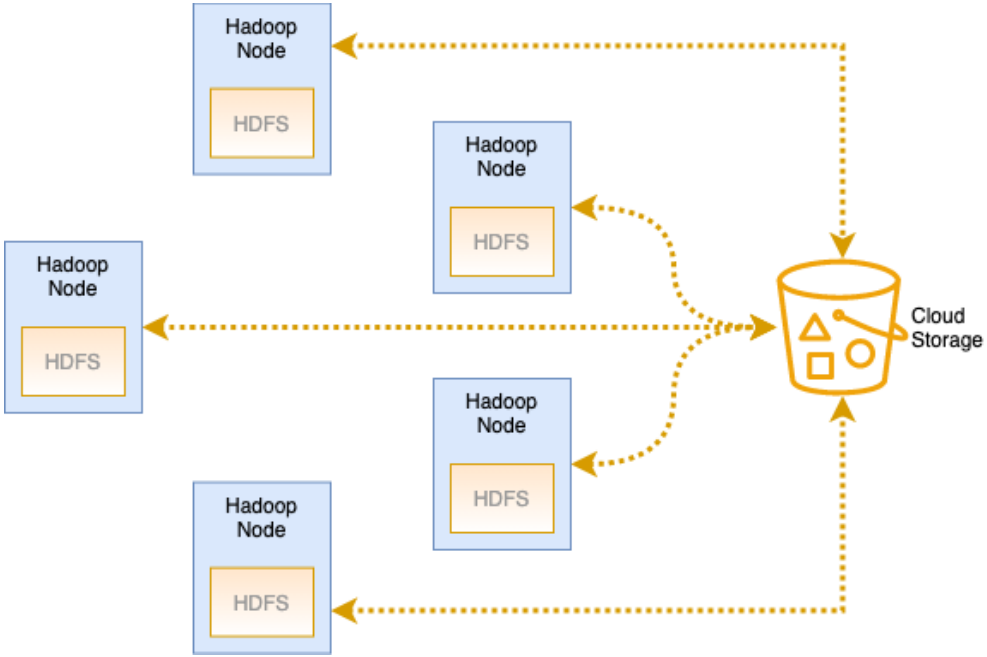
- looks like a file system
- distributed and replicated
- inexpensive
- 99.999999999% durability

Can be used instead of HDFS

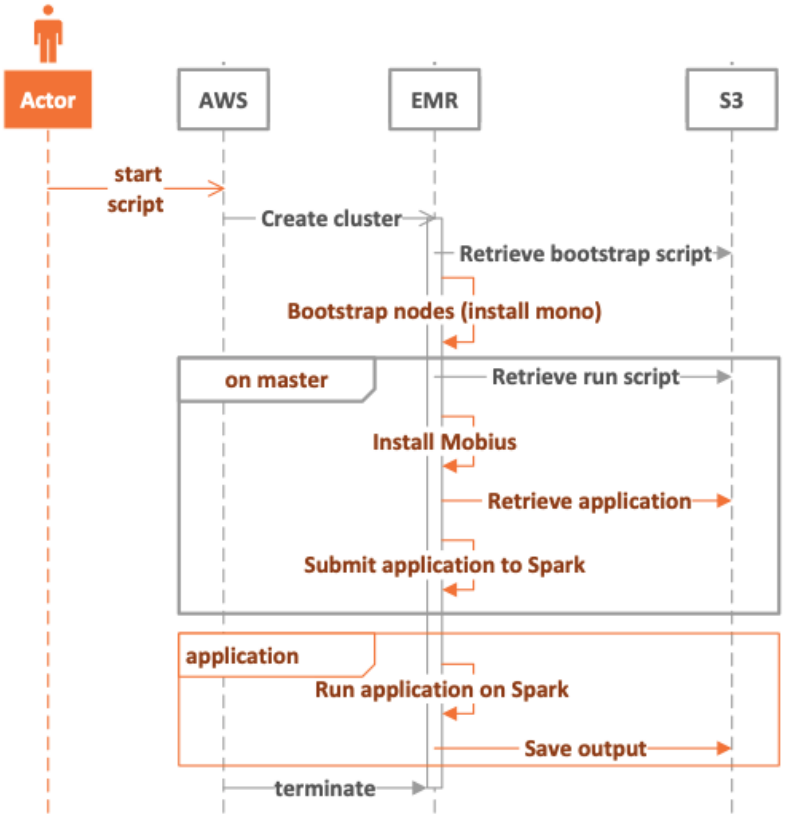
- separate storage from compute
- ephemeral clusters
- more clusters can work on the same data

Think about it: end of data locality?

- Partially
- High-speed networking
- Benefits outweigh drawbacks



CLOUD – EPHEMERAL CLUSTERS



Infrastructure as Code

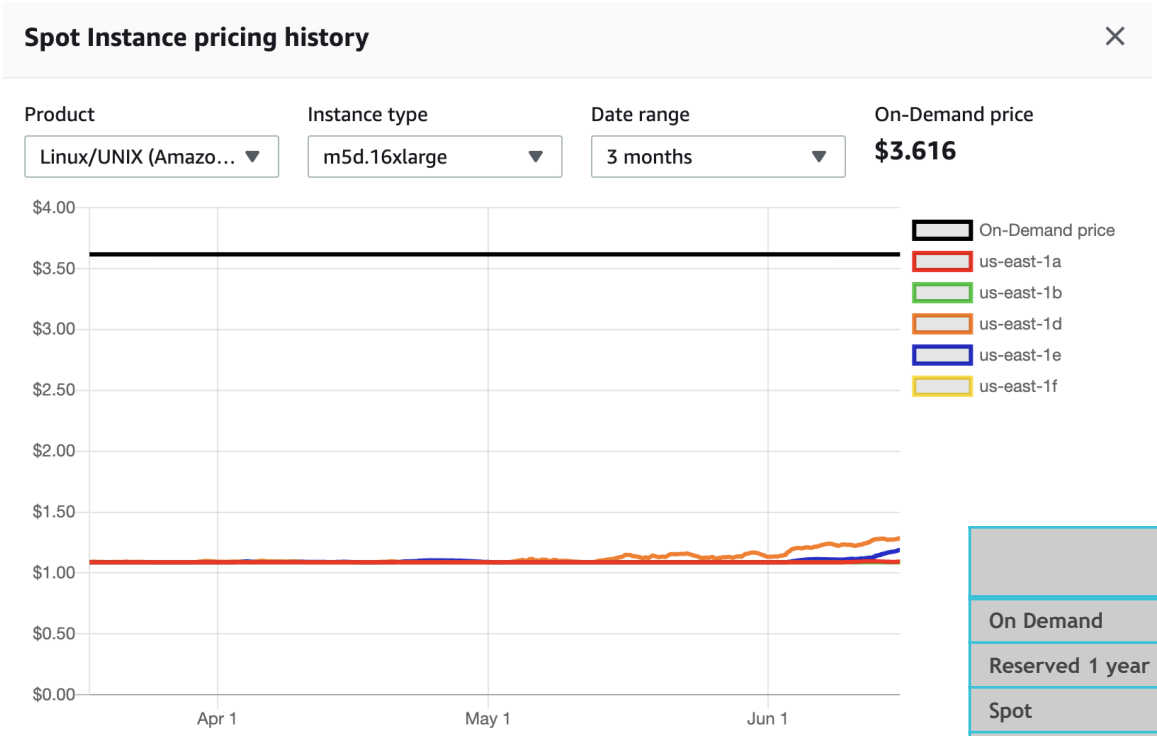
- Provision cluster
- Install additional software (if needed)
- Start application
- Terminate cluster on completion

Example: scalable workload – larger cluster

- Reduce execution time
- (almost) the same costs

Cluster size	Hourly cost per node	Execution time	Total Cost
5 nodes	$1	4 hours	$20
20 nodes	$1	1.1 hours	$22

SPOT INSTANCES – SPARE RESOURCES CHEAPER



	EC2	EMR	Total	Relative Cost
On Demand	$ 3.616	$ 0.27	$ 3.886	100 %
Reserved 1 year	$ 2.278	$ 0.27	$ 2.547	65 %
Spot	$ 1.300	$ 0.27	$ 1.570	40 %
~~Spot Block 1 hour~~	~~$ 1.989~~	~~$ 0.27~~	~~$ 2.259~~	~~58 %~~
~~Spot Block 6 hours~~	~~$ 2.531~~	~~$ 0.27~~	~~$ 2.801~~	~~72 %~~

CLOUD – SERVERLESS

Serverless Computing

- an execution model in which the **cloud provider**
- … **dynamically** manages the **allocation of resources**

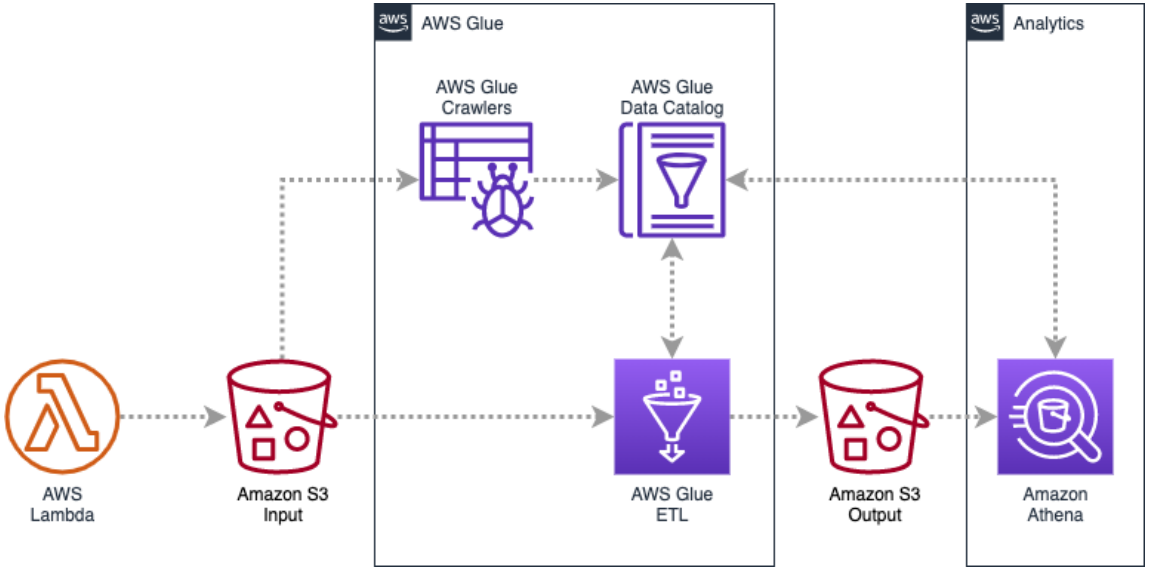
Servers are unimportant

- The cloud provider manages them internally
- Resource details are hidden from us
- We only care about code, performance and cost

Costs

- **Traditional:** pay for the resources + services
- **Serverless:** pay for the usage (execution time, amount of data, etc.)

CLOUD – SERVERLESS IN BIG DATA



AWS Lambda

- Function as a service
- Triggers execution

AWS Glue Data Catalog

- Metastore as a service

AWS Glue ETL

- ETL as a service (e.g., Spark)

Amazon Athena

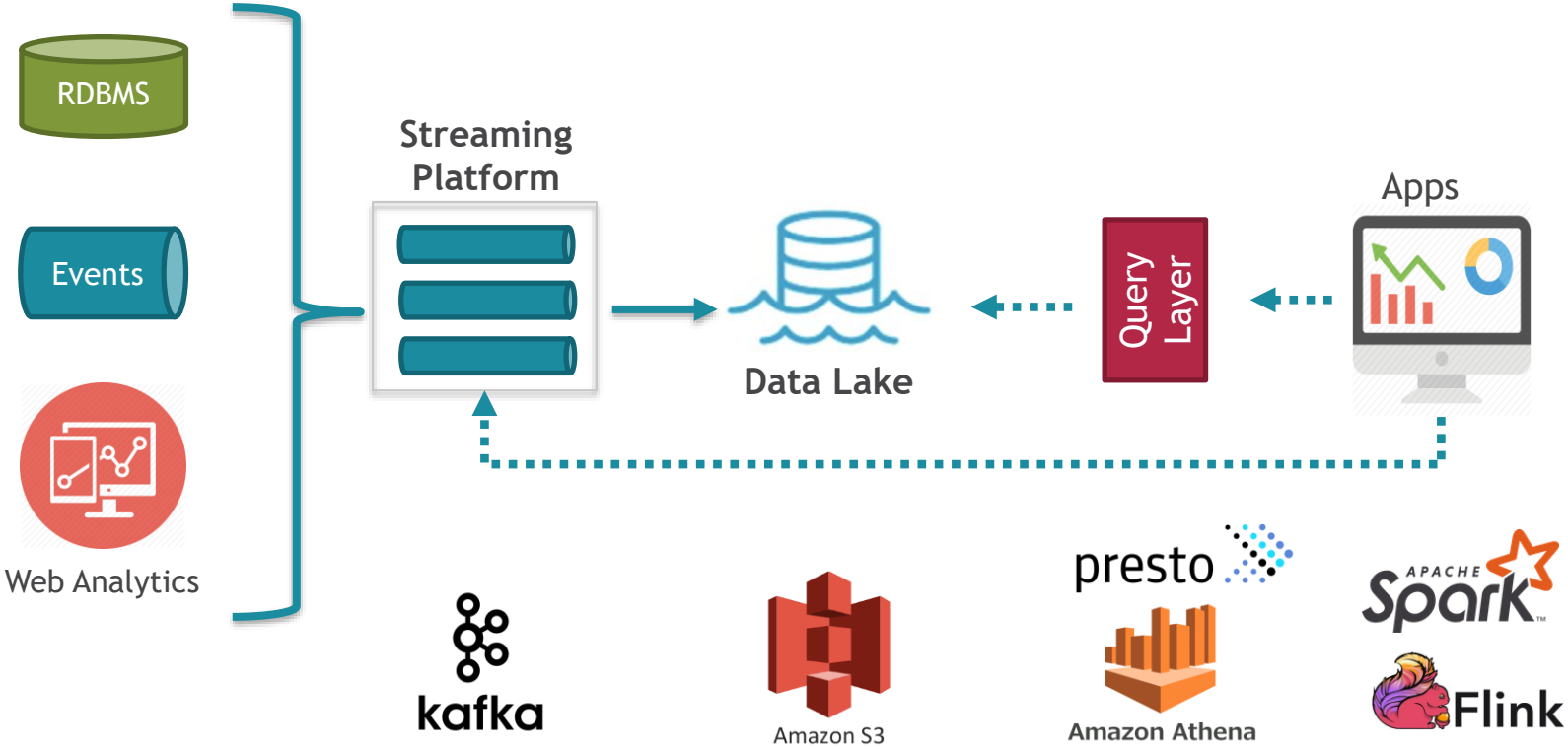
- Analytics as a service (presto)

CLOUD – TAKE AWAY

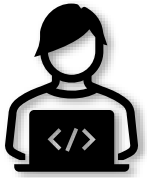
- Hadoop as a Service
- Infrastructure as Code
- Cloud storage: separate storage from compute
- Dynamic cluster sizes: Autoscaling
- Dynamic clusters: Ephemeral clusters
- Serverless: Focus on code, forget infrastructure

CONCLUSIONS

EXAMPLE PROJECT – DATA PLATFORM



CAREER PATHS IN DATA



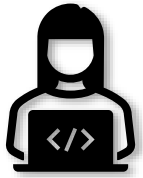
Data Engineer

Activities

- develops, tests & maintains data pipelines
- prepares data for Data Scientist
- builds infrastructure

Skills

- development (Java, Scala, Python)
- linux, docker, kubernetes
- cloud infrastructure (terraform)
- big data (Spark, Kafka, ...)



Data Scientist

Activities

- cleans & organizes data
- analyses data to develop insights
- builds models and solves business needs

Skills

- math, statistics
- development (Python, R)
- ML, statistics libraries
- some big data (Spark)

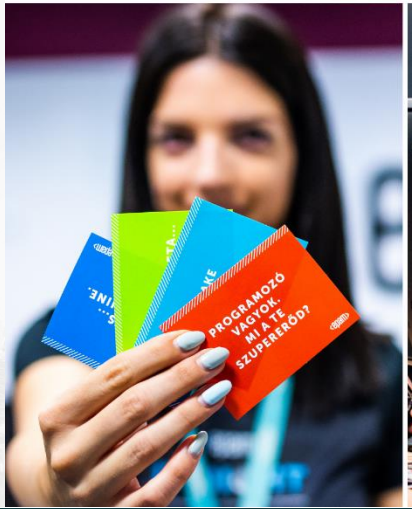
About EPAM MEP

The Mentoring Program is our in-house upskilling program that allows you to **learn from our EPAM experts**.

The program will give you the chance to acquire the right knowledge **to become one of EPAM's technologists.**

- Java
- .NET
- Test Automation
- DevOps
- Data Engineering

Interested? Apply here: https://epa.ms/edu-hungary




epam
Questions?