Syracuse University
School of Information Studies

Applications of Opensource in Big Data Management with Examples

Michael Fudge
Professor of Practice
Syracuse University
CASL Workshop, December 2022

About Me

- Professor of Practice in the School of Information Studies at Syracuse University
- Career Background
  - Data warehousing / Business intelligence
  - IT Systems Management
  - Software / Web / Mobile application development
- Academic Areas of Interest
  - Academic Computing Infrastructure (Kubernetes, Jupyterhub)
  - Learning Analytics / AI assisted learning
  - Internet of Things

About this talk: Two Parts

Presentation

- Overview of Open-Source tools for Big Data management.
- Conceptual and Foundational
- Gain an understanding of the landscape.

Pre-Recorded Lecture

- Demonstrations of the big data lab environment with examples of how to use it.
- In-depth and Experiential
- Try out Spark, Spark SQL, Object storage, MongoDb, Cassandra, and Neo4j.

https://github.com/mafudge/cas-library-bigdata-workshop-2022

Sections of This Talk

1. The Long Journey to Defining Big Data
2. Big Data Tools for Storage and Compute
    1. Hadoop: HDFS / MapReduce
    2. Spark: PySpark and SparkSQL
3. Common NoSQL Big Data Databases
    1. Document / MongoDb
    2. Big Table / Wide Column / Cassandra
    3. Graph / Neo4j

The Importance of Data

Questions?

- Why are US Social networking companies like Facebook, Google, and Twitter free services?
- Why do retail companies like Target, Starbucks, and Verizon have customer rewards programs?
- What makes rideshare services like U[illegible]r c[illegible]ft more efficient than taxis?

Two Kinds of Data

Reference-Oriented Data

- Business Entities; Data that references a person, place or thing
- Any row is updatable over time, Full CRUD
- Limited growth, can be frequent changes.
- Rows referenced by a natural or business key

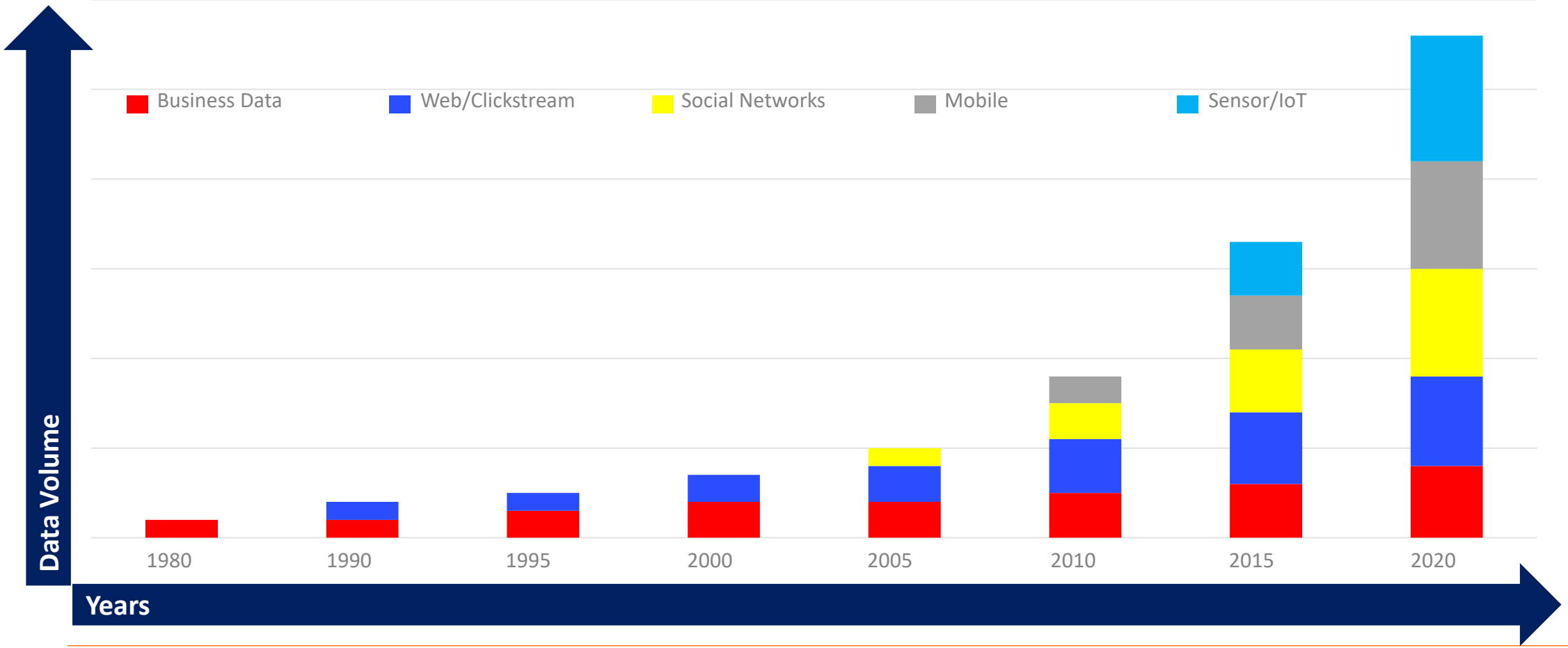
Process-Oriented Data

- Data Relationships; Data corresponding to an event or process
- Immutable – rows do not update, CR no UD
- Rapid growth
- Rows have no natural key; have a timestamp or composite key

The Prevalence of Data

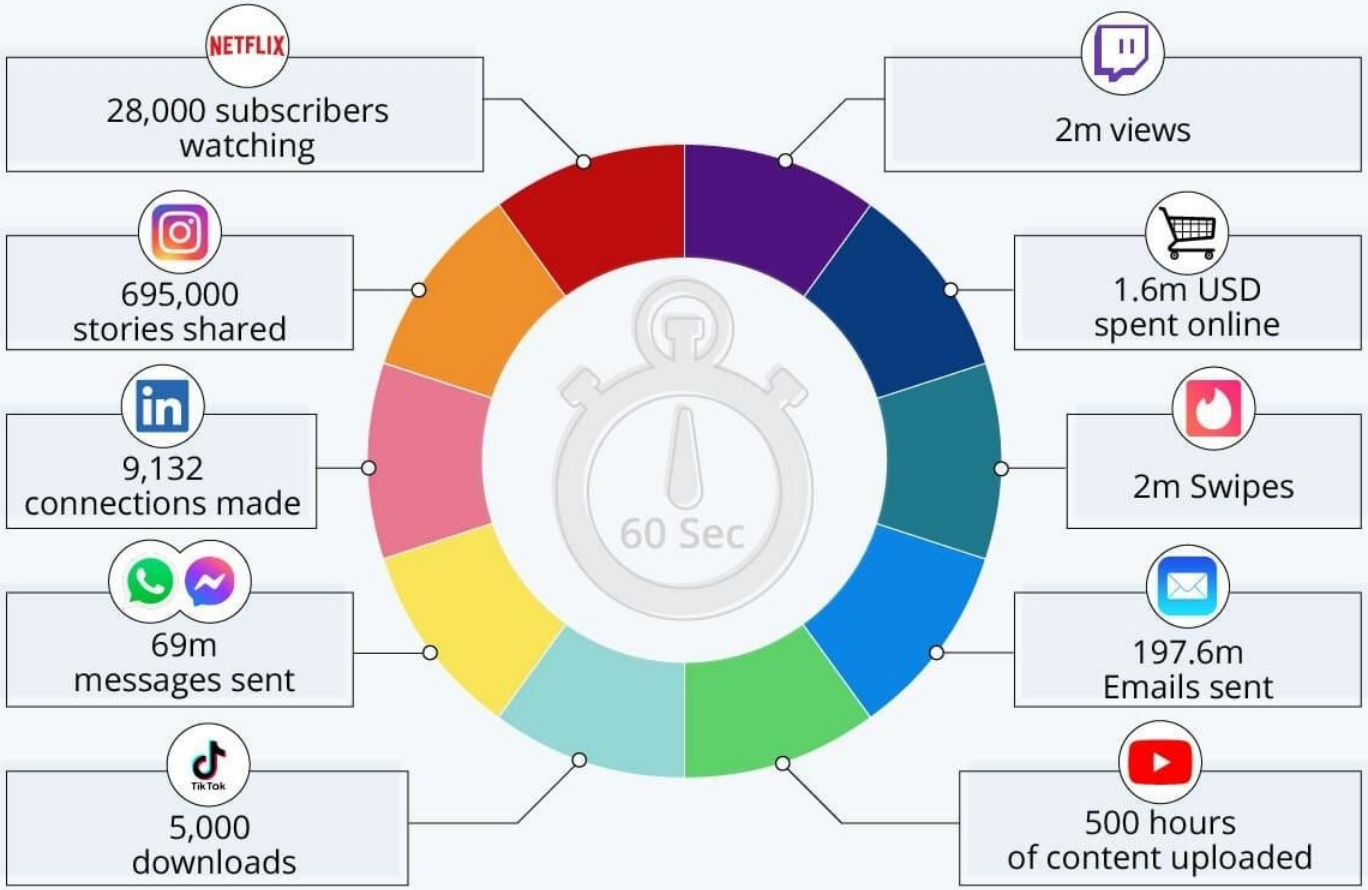
- Data are everywhere!
- There has been an explosion of data over the past few years
- The majority is process-oriented data. Based on sensors or events.
- There are new varieties of data.
  - Business data → user generated data → device generated data
- And there are different mediums than before.
  - Structured → semi-structured → unstructured text → multimedia

Example of how the volume of Organizational data has grown over time.



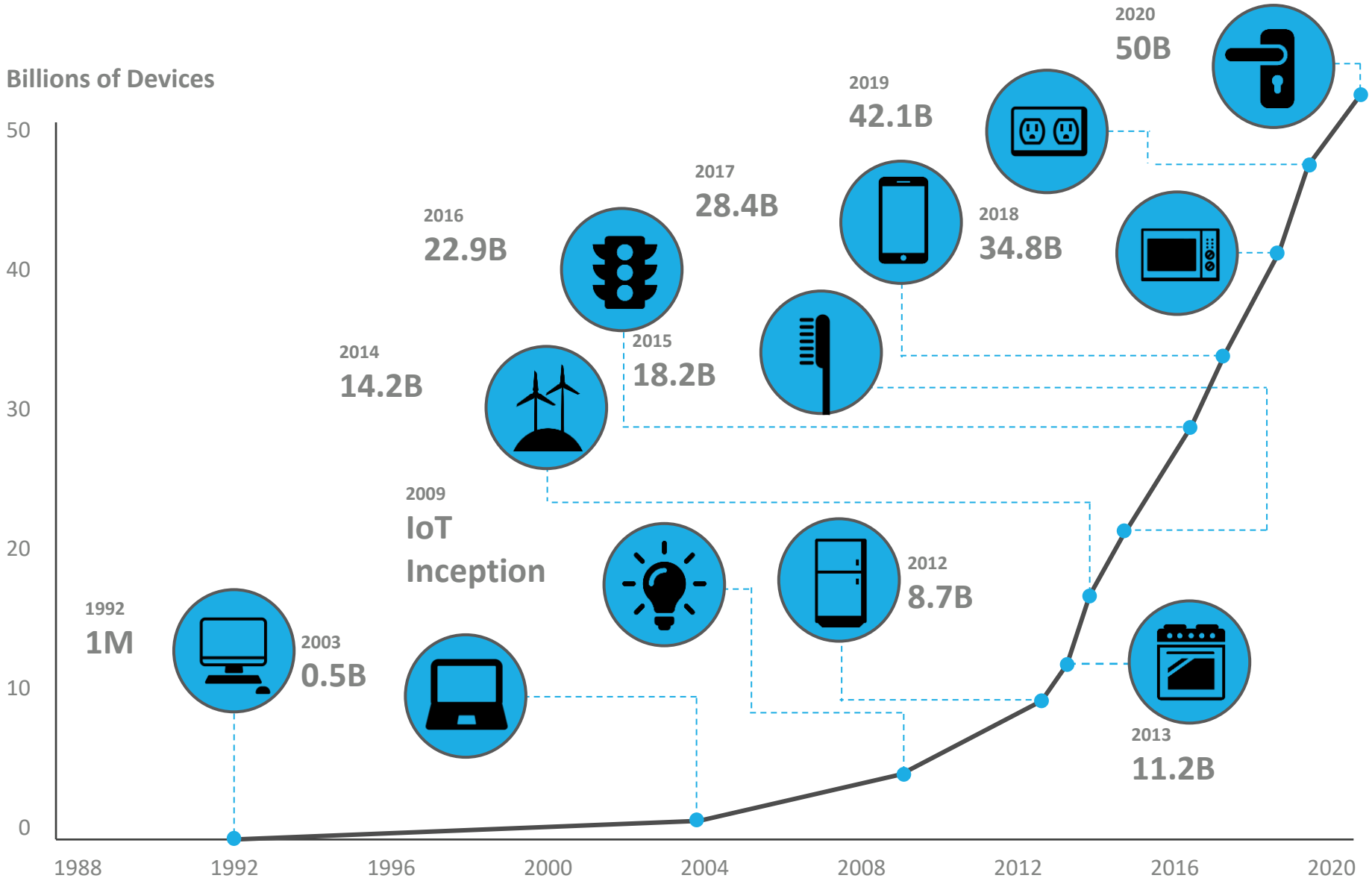
A Minute on the Internet in 2021

Estimated amount of data created on the internet in one minute



Source: Lori Lewis via AllAccess

How many devices are connected to the Internet?



More Data… Better Machine Learning!

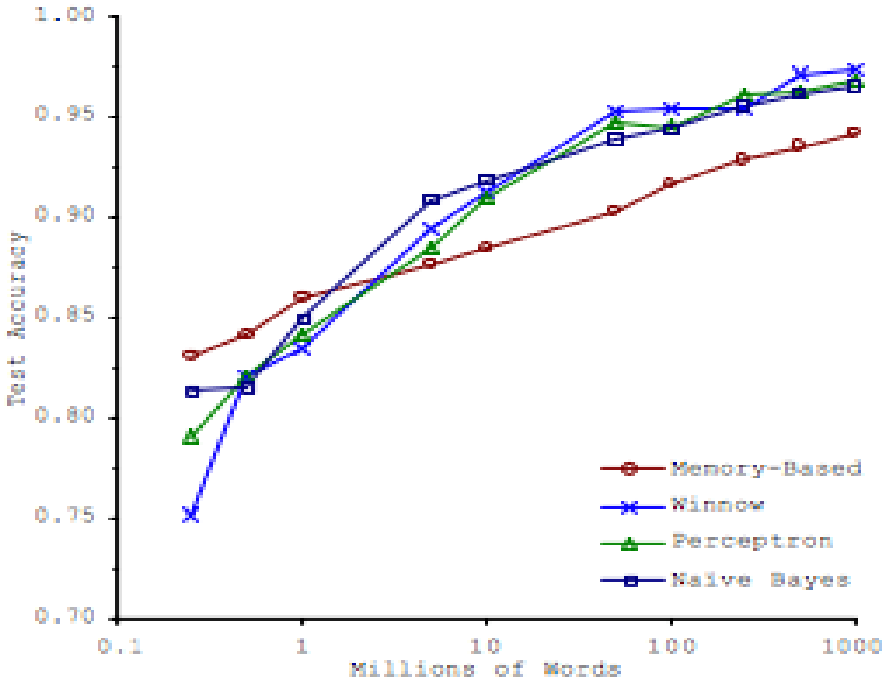


Figure 1. Learning Curves for Confusion Set Disambiguation

Banko & Brill, 2001:
"Algorithms Predict Better with Larger Data Sets"

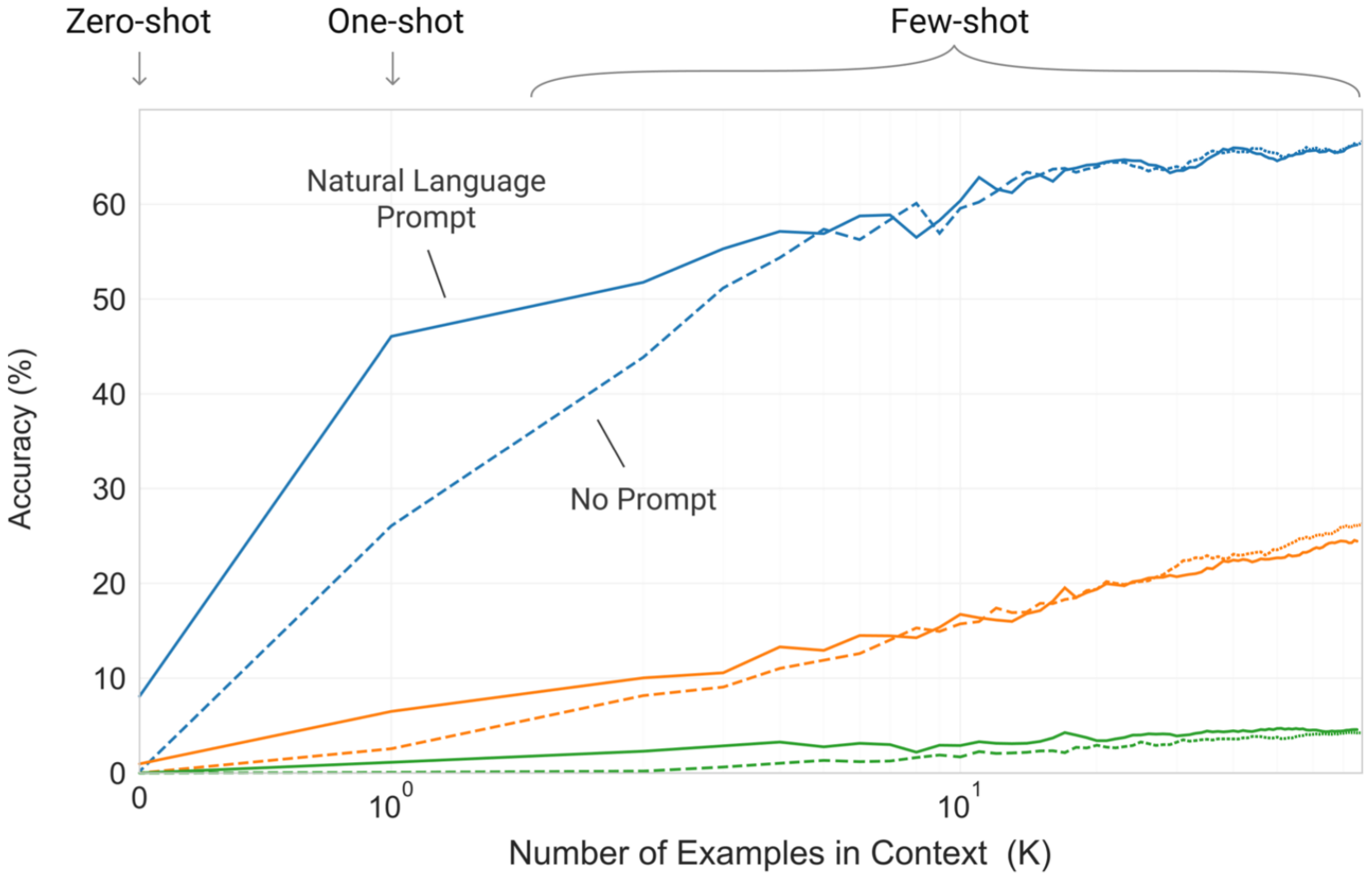
The Unreasonable Effectiveness of Data

Alon Halevy, Peter Norvig, and Fernando Pereira, *Google*

Halevy, Norvig & Pereira, 2009:
"Data set size matters
more than algorithms"

Big Data is important.
Size does matter!
Data size matters more than the algorithm!

GPT-3 – A Text Completion Model

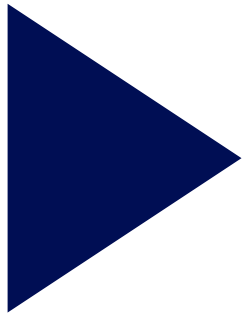


- Zero Shot Prompt: "List fruits"
- One Shot Prompt: "List Fruits
  For example:
  Apple"
- Few Shot Prompt: List Fruits
  For example:
  Apple
  Banana
  Orange"

https://medium.com/analytics-vidhya/openai-gpt-3-language-models-are-few-shot-learners-82531b3d3122

The extraction of value from data is not trivial

The more data, the greater the chance for value within



But the more data the more effort is required to extract that value!

Criticisms of the Relational Database Model

What Makes the RDBMS Great?

- Easy to understand tabular structure,
- Table meta-data and data integrity constraints help to ensure we don’t write “bad data”
- Data in tables are normalized to minimize redundancy and improve data consistency
- Wide adoption and an abundance of tools
- Support for SQL (Structured Query Language), which is easy to learn
- ACID: Writes are atomic, consistent, isolated, and durable.
- The de-facto standard—mature and proven

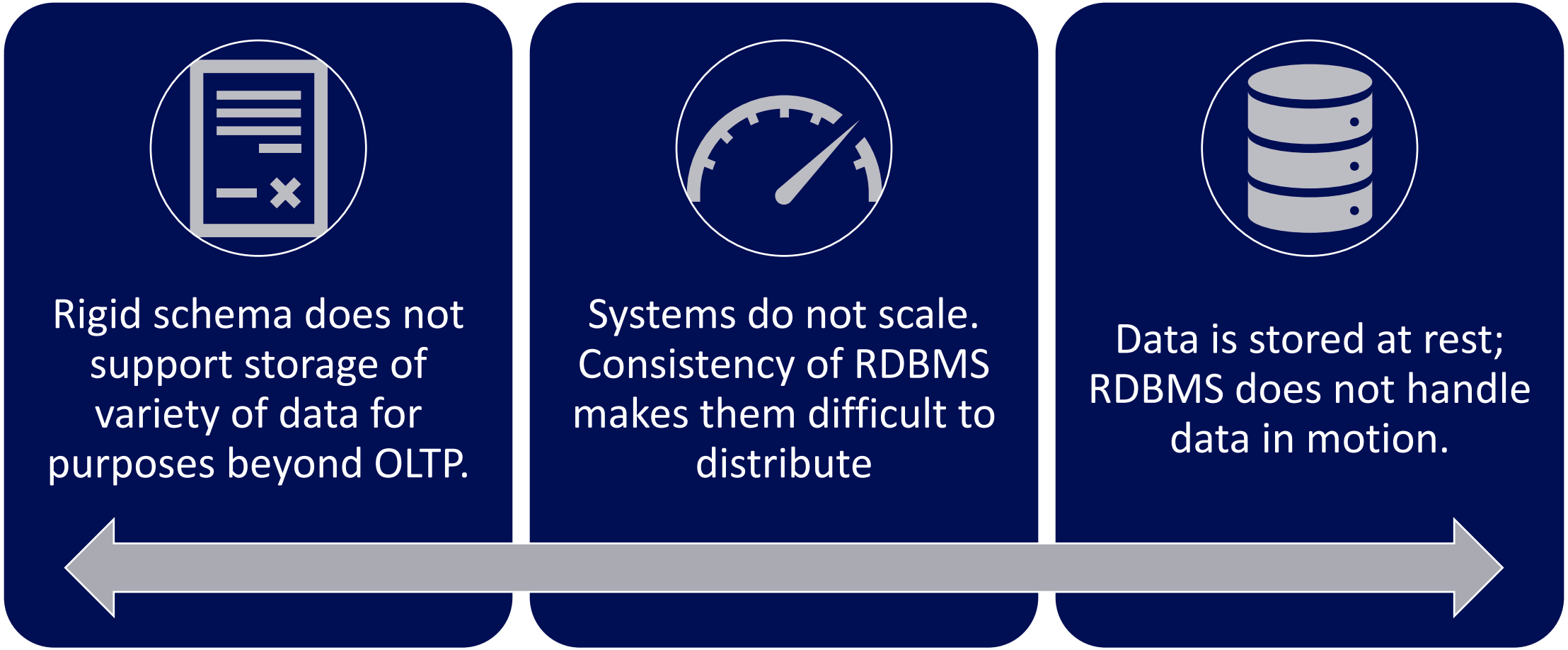
PROBLEM:

When all you have is a hammer (RDBMS), every data problem looks like a nail.

One Size Does Not Fit All !

- No one database management system is best suited for all of the complexity and variety of data found today.
- Therefore, different systems exist; to manage data that vary in:
  - Structure
  - Size
  - Rate of change
- We've learned the hard way that not all data problems are nails, and there is more than just the hammer.
- Most relational database vendors are keen to this and now offer hybrid options, that go beyond relational, but still….

Three Main Criticisms of the RDBMS

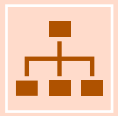


Scalability

RDBMS does not scale easily

Out of the box, Increases in data volumes and transactions per second lead to decreases in performance.

Companies throw hardware at the problem, buying datacenter in a box solutions like Oracle RAC, SQL Server MPP, or Vertica. This is expensive!



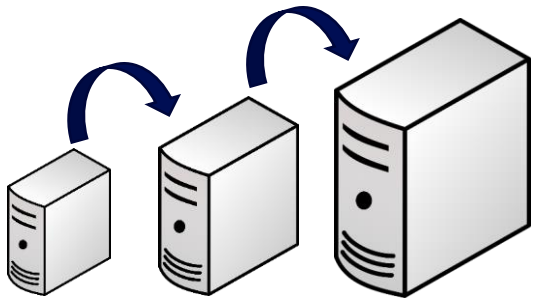
Companies hire specialists to set up clustering and improve performance through indexing, caching, and partitions. Requires money and expertise!

NewSQL: “We can scale that for you, but you will need to use the cloud!”
- Google Cloud Spanner, Amazon Aurora, or MemSQL.

Scaling: Up Versus Out

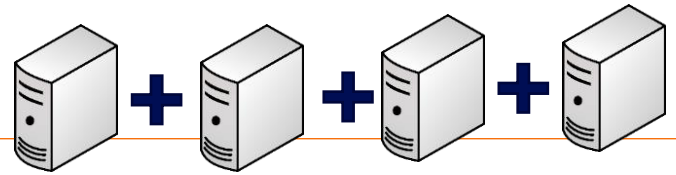
Vertical Scaling "Up"

- Add more resources to an existing system running the service
- Easier, but limited scale
- Single point of failure
- Any system can be scaled up

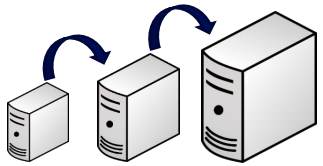


Horizontal Scaling "Out"

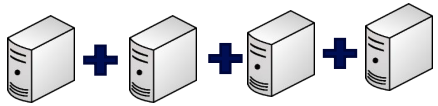
- Run the service over multiple systems, and orchestrate communication between them
- Harder, but can achieve unlimited scale
- Overhead resources needed to manage nodes
- System must be designed to do it



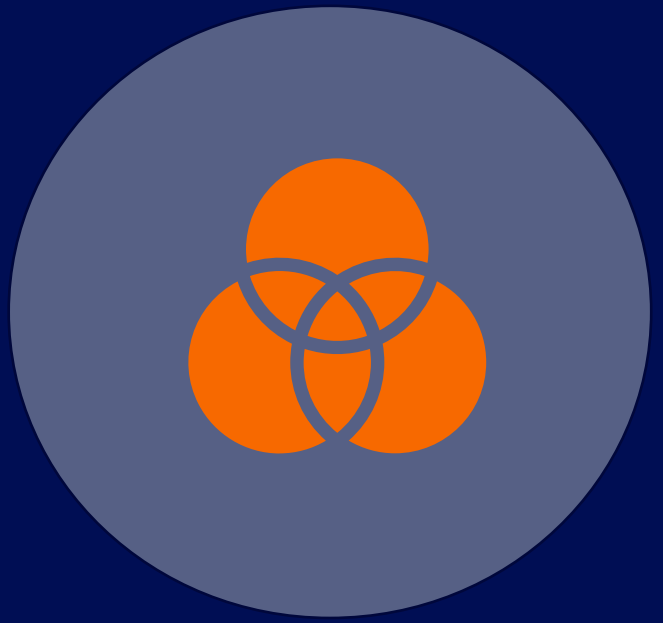
Over-Simplified Example: Some Website Hosted at AWS

Scale it up 

- It is hosted on amazon EC2.
- Upgrade the compute instance from
  - t2.micro 1 CPU 1 GB to
  - t2.xlarge 8CPU 32 GB
- Just pay more and you are done.
- When you update the website content, you do it once.

Scale it out 

- Setup AWS Elastic Load balancer in front of your site.
- Duplicate your t2.micro instance 7 times (you now have 8 copies)
- Configure ELB to route traffic to one of your 1 of 8 instances of your website.
- When you update the website, you need do it 8 times



CAP Theorem

When you scale out, you must concede data consistency or availability

WE OFFER 3 KINDS OF SERVICES

GOOD - CHEAP - FAST

BUT YOU CAN ONLY PICK TWO

GOOD & CHEAP WON'T BE FAST

FAST & GOOD WON'T BE CHEAP

CHEAP & FAST WON'T BE GOOD

The CAP Theorem of Distributed Data Stores

Eric Brewer[1]: You can only have two of the following three guarantees:

1. **Data consistency:** all nodes see the same data at the same time
2. **Data availability:** assurances that every request can be processed
3. **Partition tolerance:** network failures are tolerated, the system continues to operate

- Relational DBMS are designed to be ***consistent*** and ***available*** *and therefore cannot be* ***partition tolerant.***

1. Brewer, E. A. Towards Robust Distributed Systems. Portland, Oregon, July 2000. *CAP theorem*.

Fundamental Principle of Distributed Data

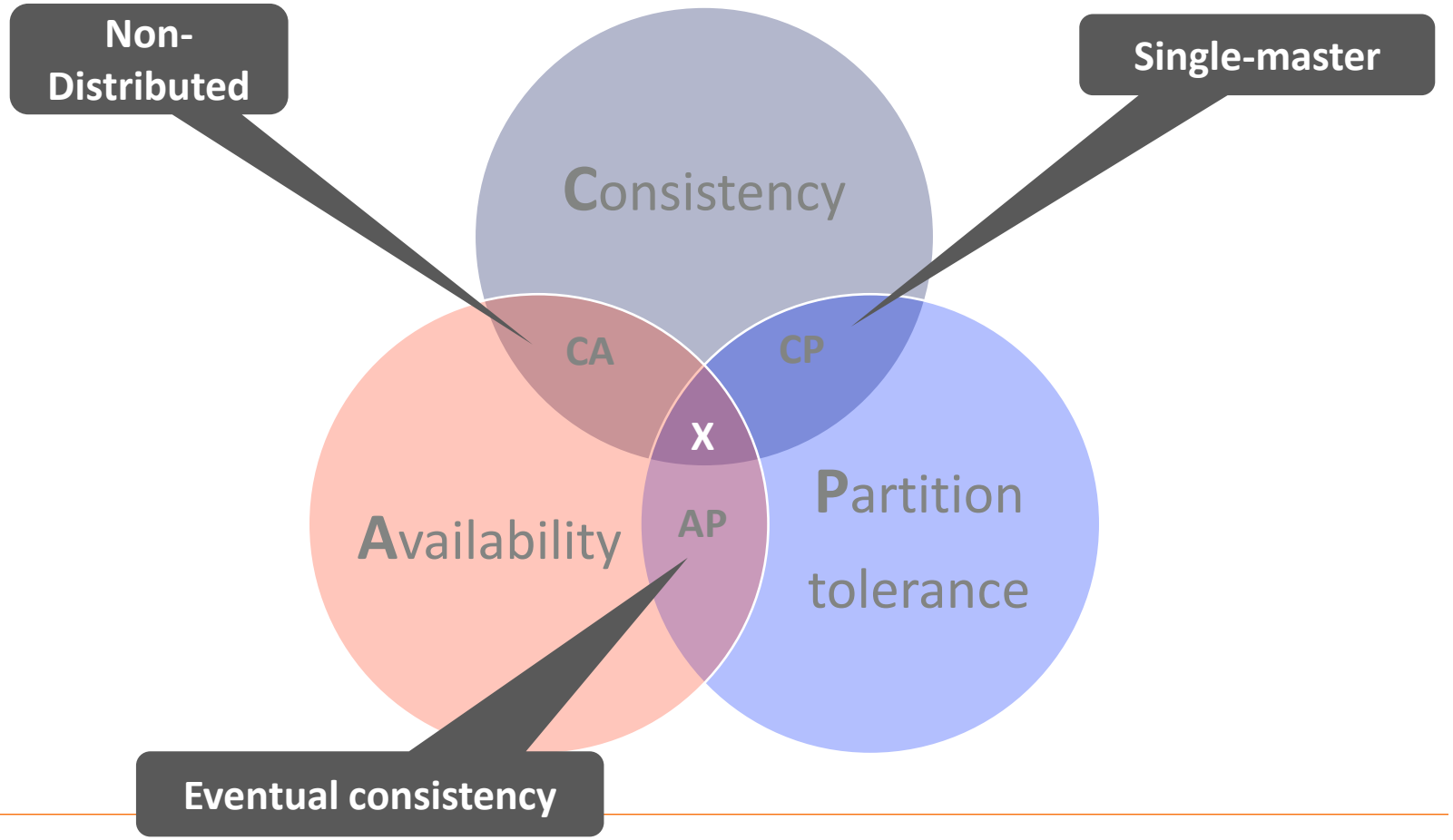
- IF you ***scale data out*** (so that data are distributed among multiple systems)
- THEN you must choose:
  - To have **data consistency** at the expense of ***data availability***
  - OR
  - To have ***data availability*** at the expense of ***data consistency***

Why Can’t We Have All 3?

- Suppose we lose **partition tolerance** between nodes. Then:
  - We must ignore any updates the nodes receive, or we sacrifice **consistency**,
  - Or we must deny changes until it becomes ***available*** again.
- If we choose to guarantee ***availability*** of requests, despite the failure:
  - We gain ***partition tolerance*** (the system still works) but lose ***consistency*** (nodes will get out of sync).
- If we choose to guarantee **consistency** of data, despite the failure:
  - We gain ***partition tolerance*** (again, system works) but lose ***availability*** (data on nodes cannot be changed, until the failure is resolved).

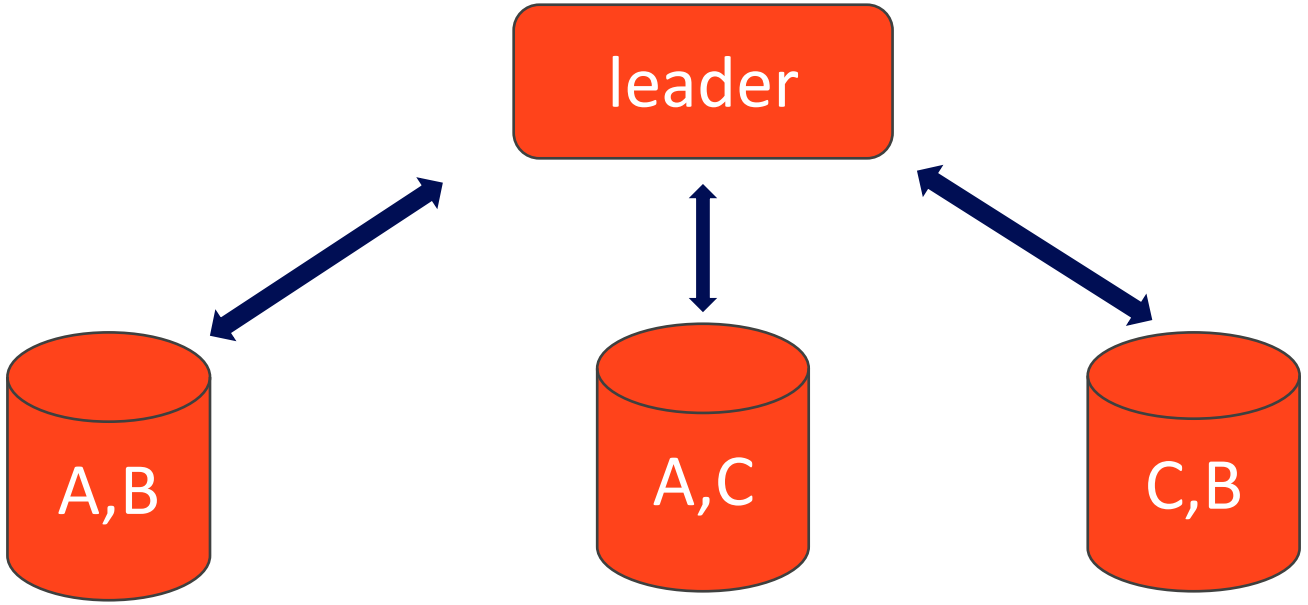
Database Systems and the CAP

What is guaranteed?



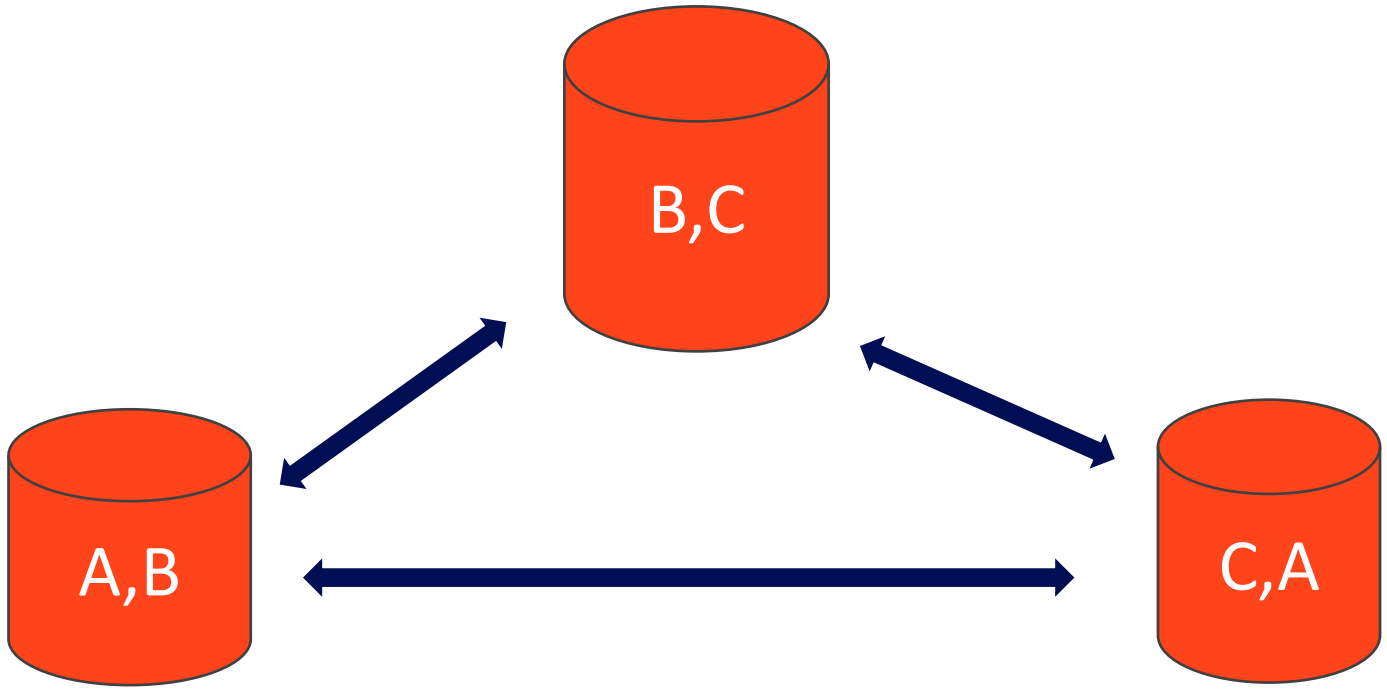
Single Master CP Systems

- A leader node is responsible for orchestration of writes. The leader stores no data
- If Leader goes down, we lose availability as we can no longer write.
- Even if the leader does not go down there are slight gaps in availability as the leader must propagate changes to other nodes.



Eventual Consistency AP Systems

- There is no leader node or single master. We read and write to any node.
- Any node can fail, and we can write still but our reads will be inconsistent.
- Even when we don't fail not all nodes will be consistent immediately, as changes must propagate to other nodes



All Types of CAP Databases

- CA: RDBMSs like Oracle, MySQL, and PostgreSQL:
  - Focus on consistency and availability (ACID principles), sacrificing partition tolerance (and thus they don't scale well horizontally)
  - Use cases: business data, when you don't need to scale out
- CP: Single-master systems like MongoDB, HBase, Redis, and HDFS:
  - Provide consistency at scale, but data availability runs through a single node
  - Use cases: read-heavy; caching, document storage, product catalogs
- AP: Eventual consistency systems like CouchDB, Cassandra, Redis and Dynamo:
  - Provide availability at scale but do not guarantee consistency
  - Use cases: write heavy, isolated activities: shopping carts, orders, social media

ACID vs BASE

ACID Databases (CP)

- **Atomic**: Everything in a transaction succeeds, or the entire transaction is rolled back.
- **Consistent**: A transaction cannot leave the database in an inconsistent state.
- **Isolated**: Transactions cannot interfere with each other.
- **Durable**: Completed transactions persist, even when servers restart and so on.

BASE Databases (AP)

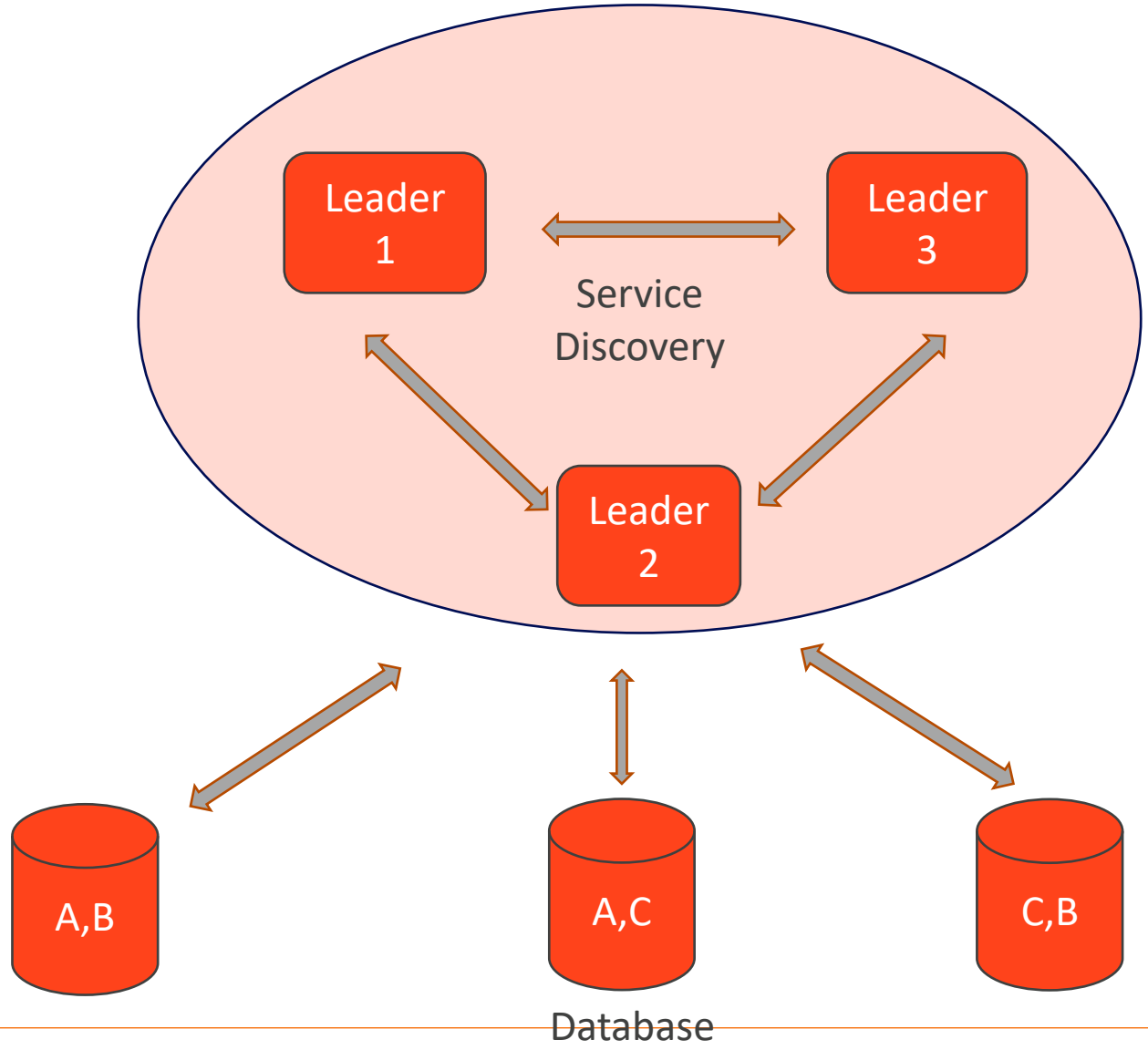
- **Basic availability**: Data can be read and written to any node.
- **Soft-state**: Nodes may change over time, even without direct updates.
- **Eventual consistency**: At some point all nodes will have the same data.

So, all Distributed Databases Are a Choice Between Availability and Consistency?

Well….

NewSQL Systems

- Circumvent the CAP theorem by combining AP and CP systems
- Leaders are AP and always available
- Data storage is CP and thus always consistent.
- Multi-layering ensures available and consistent data at scale.



NewSQL

- Typically, The Relational Model at Scale;
  - ACID Compliant
  - Horizontal Scalability
  - Why Relational? Its flexible and proven
- Complex system architecture thus cloud hosted or Kubernetes.
- Database as a service in the cloud:
  - Google cloud spanner, Amazon Aurora, Azure CosmosDb, MemSQL
- Open source:
  - CockroachDB, YugabyteDB

NoSQL

The Rise of NoSQL

NoSQL

- NoSQL stands for “not only SQL.”
- It generally refers to a wide range of database systems that address the scalability and other challenges of the relational database model.
- The entire point of NoSQL is that your need is not met by the relational model (really has nothing to do with SQL itself).
- It is not an anti-SQL or anti-relational database movement, although can be perceived that way.
- Not all problems are nails. Not all solutions are hammers!
- NoSQL databases use ***purpose built models*** whereas relational are more ***general purpose.***

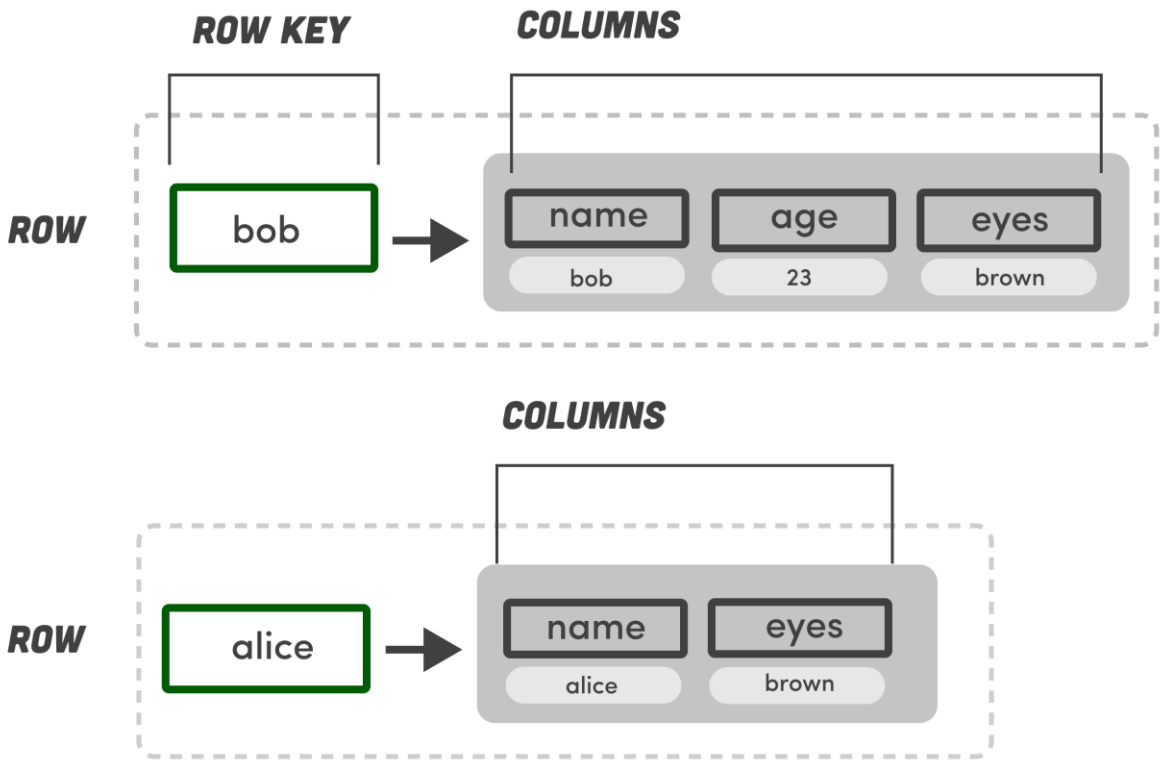
NoSQL Data Models

Of NoSQL Databases

Wide-Column

- Data stored under a row key.
- Value consists of columns of data in one big table.
- Every row does not need the same set of columns. Flexible schema.
- Common columns can be grouped and stored together.
- Not full CRUD, more CR.
- Designed to be BASE; Fast, guaranteed writes.
- SQL-Like query language

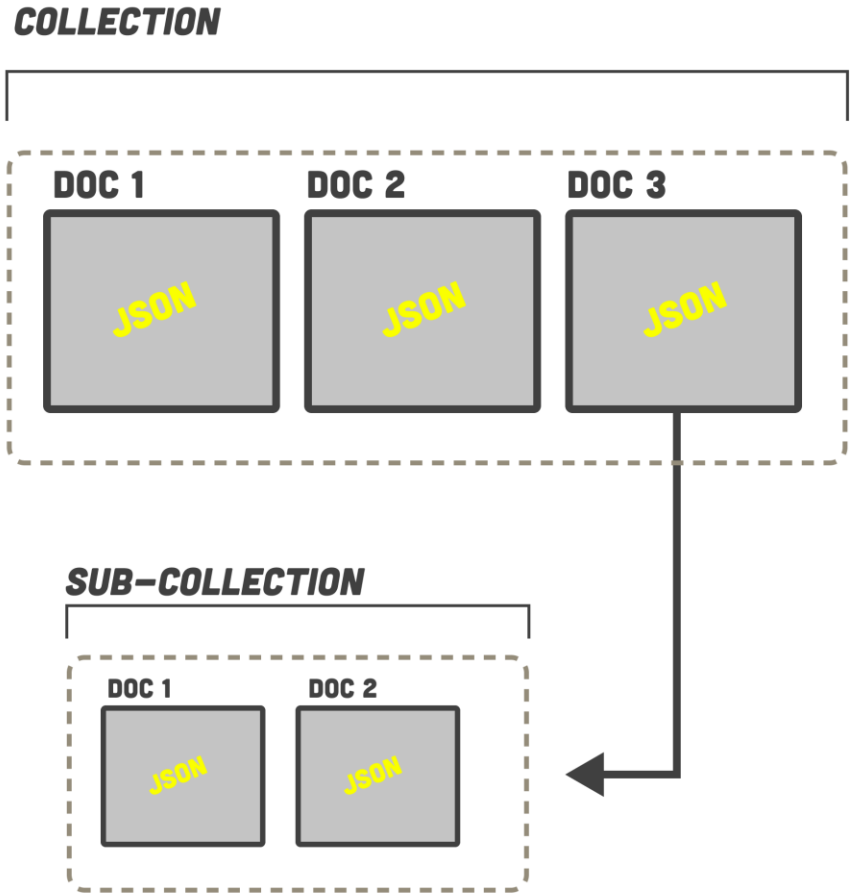
Open Source:
Cassandra, HBase



Document

- Keyed Data stored as documents in collections.
- Data is usually semi-structured, JSON format, consisting of key-value pairs.
- Documents in collections can have pointers to documents in other collections.
- Full CRUD.
- Systems can be ACID or BASE
- Custom Query Language

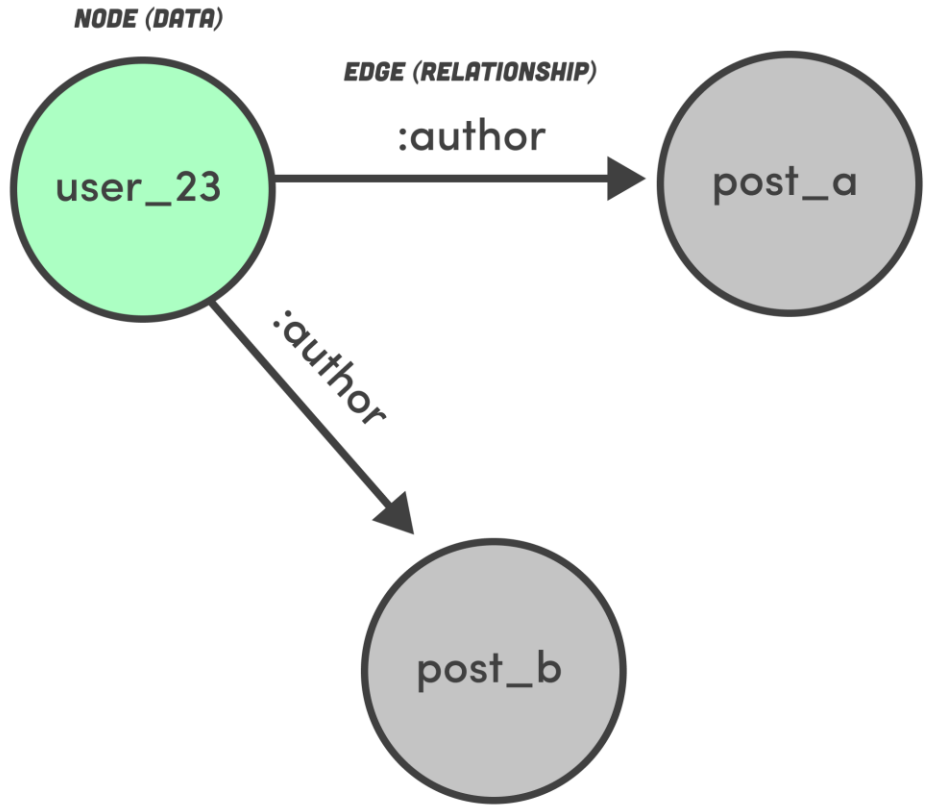
Open Source:
MongoDb, CouchDb



Graph

- Data model consists of node and edges.
- Structured and unstructured data may be stored in both the node and edge.
- Idea for data consisting of complex relationships.
- Supports full CRUD.
- Systems can be ACID or BASE
- Custom Query Language, very unlike SQL.

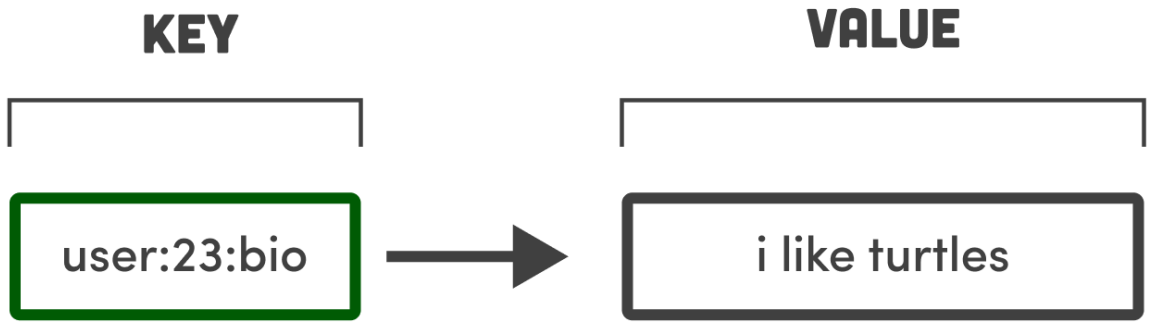
Open Source:
MongoDb, CouchDb



Key-Value

- Data stored under a key.
- Value can be anything.
- Keys can be name spaced.
- Very simple data model.
- Very fast and efficient.
- Supports full CRUD.
- Commonly used to store data you need to retrieve quickly, as if it were in memory.
- Simple Query Language
- ACID or BASE

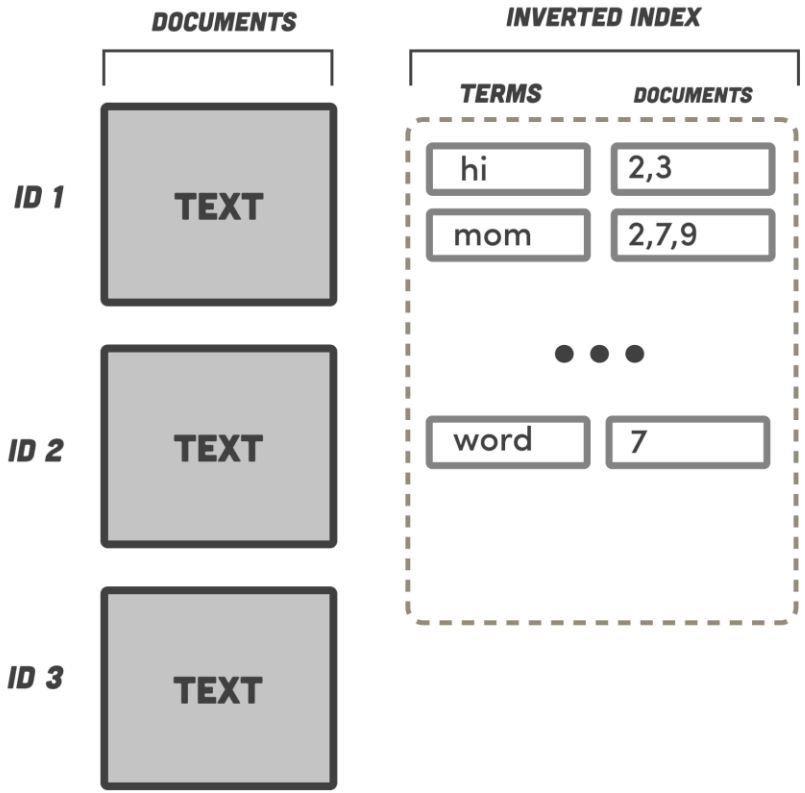
Open Source:
Redis, Memcached



Search Engine

- Documents can be structured, semi-structured, or unstructured.
- Indexes are created over the documents, and one can search for any content within the document.
- Search can find any data within the document; semi-structured data can be keyed.
- Not full CRUD, more CR.
- Systems are typically BASE
- Custom Query Language, non-SQL like.

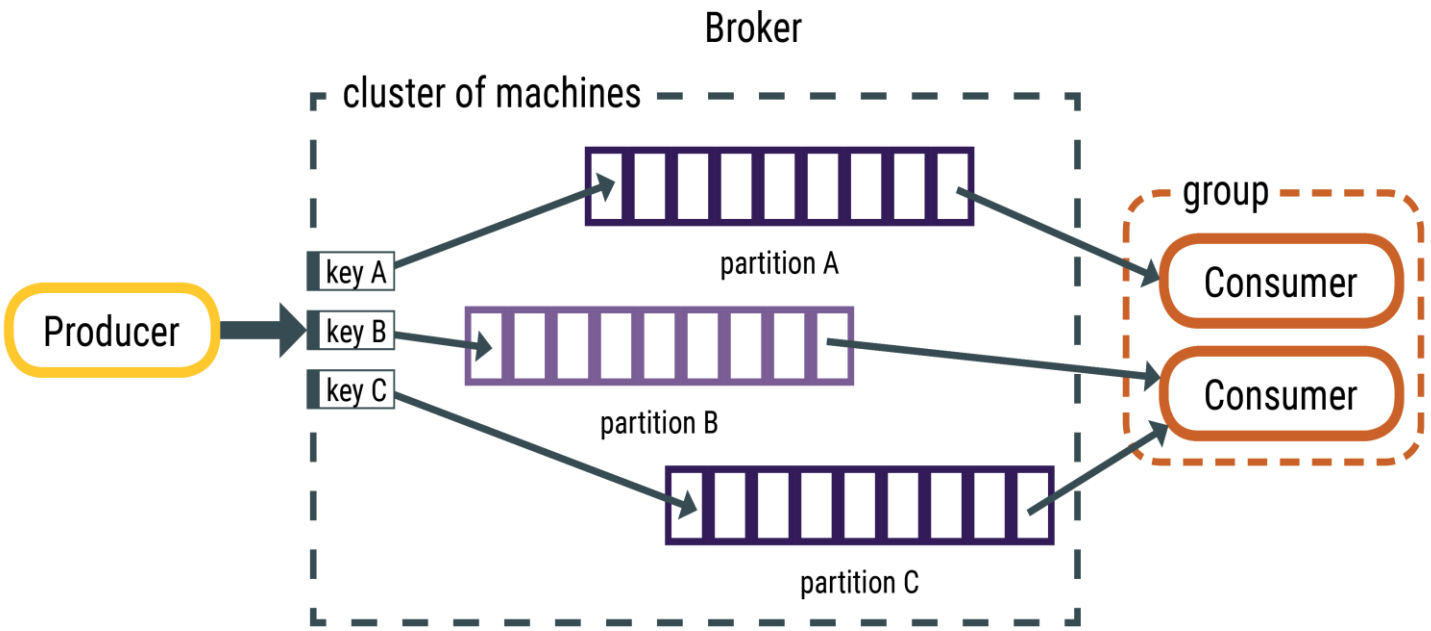
Open Source:
Elasticsearch, Solr



Streaming

- Supports structured, semi-structured, or unstructured data.
- Data in motion, not data at rest.
- Producers sends data, consumers receive it.
- Data is sent to a topic where it awaits consumption by consumers.
- Systems are typically BASE + ACID
- API's for publishing / subscribing to topics.
- Custom Query Language

Open Source:
Kafka



Multi Model

- Most RDBMS are now Multi-Model, including support for other models like Graph, Document, and Key-Value.
- This does not imply these systems will scale horizontally easily.
- Multi-Model databases are evolving in their features, performance and usability.
- Oracle, PostgreSQL, and Microsoft SQL Server are Multi-Model

- PostgreSQL is a relational database but has adopted many of the features found in NoSQL databases.
  - In memory tables for basic CRUD operations (Like Key-Value)
  - Document database for no-schema with JSON support
  - Graph support through Apache AGE
  - In-Memory support for Key-Value
  - Connectors to query unstructured data.

The NoSQL Guide

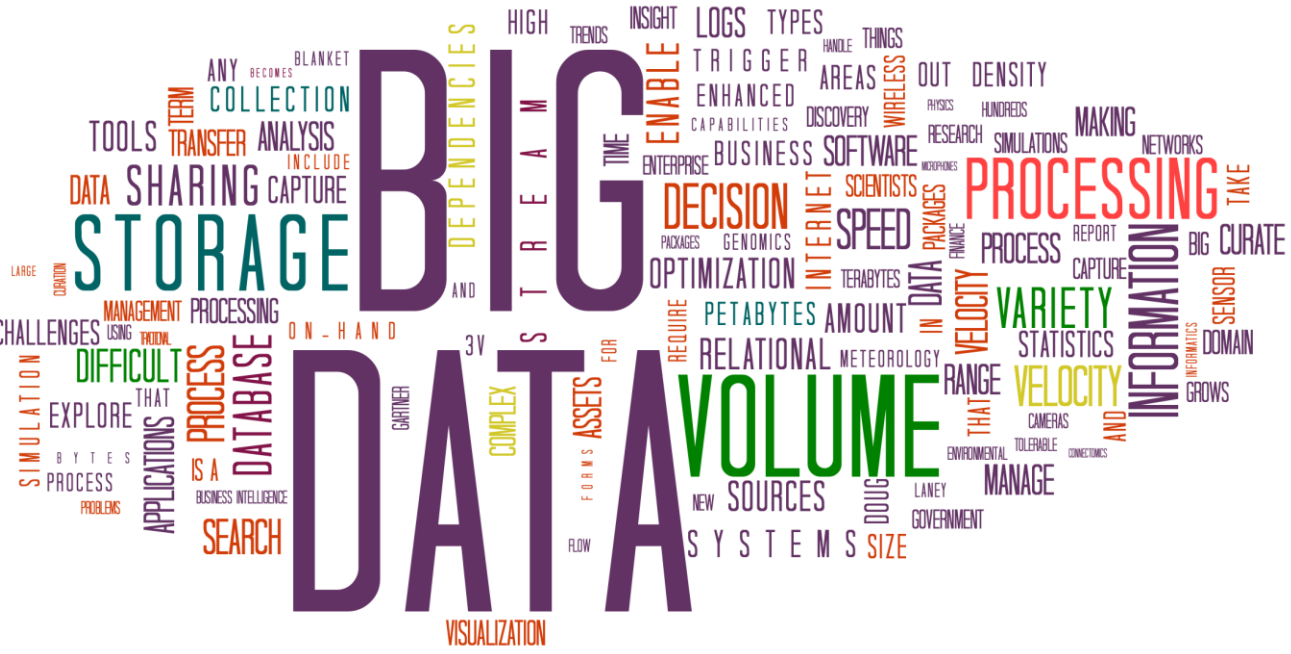
Database Model	Best In Class **	AWS	Azure	GCP
Relational	Oracle (rank:1)	Aurora	Azure SQL	Cloud SQL
Document	MongoDb (rank:5)	DocumentDB	CosmosDB (multi-model)	Firestore
Key-Value	Redis (rank:6)	DynamoDB	CosmosDB (multi-model)	Datastore
Wide Column	Cassandra (rank:12)	Keyspaces (Cassandra)	CosmosDB (multi-model)	Bigtable
Graph	Neo4j (rank:20)	Neptune	CosmosDB (multi-model)	Neo4j
Search	Elasticsearch (rank:8)	Elasticsearch	Elastic Stack	Elasticsearch
Object	Minio (rank:none)	S3	Blob	Cloud Storage
Streaming	Kafka (rank: none)	Kinesis	Event Hubs	Confluent cloud

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

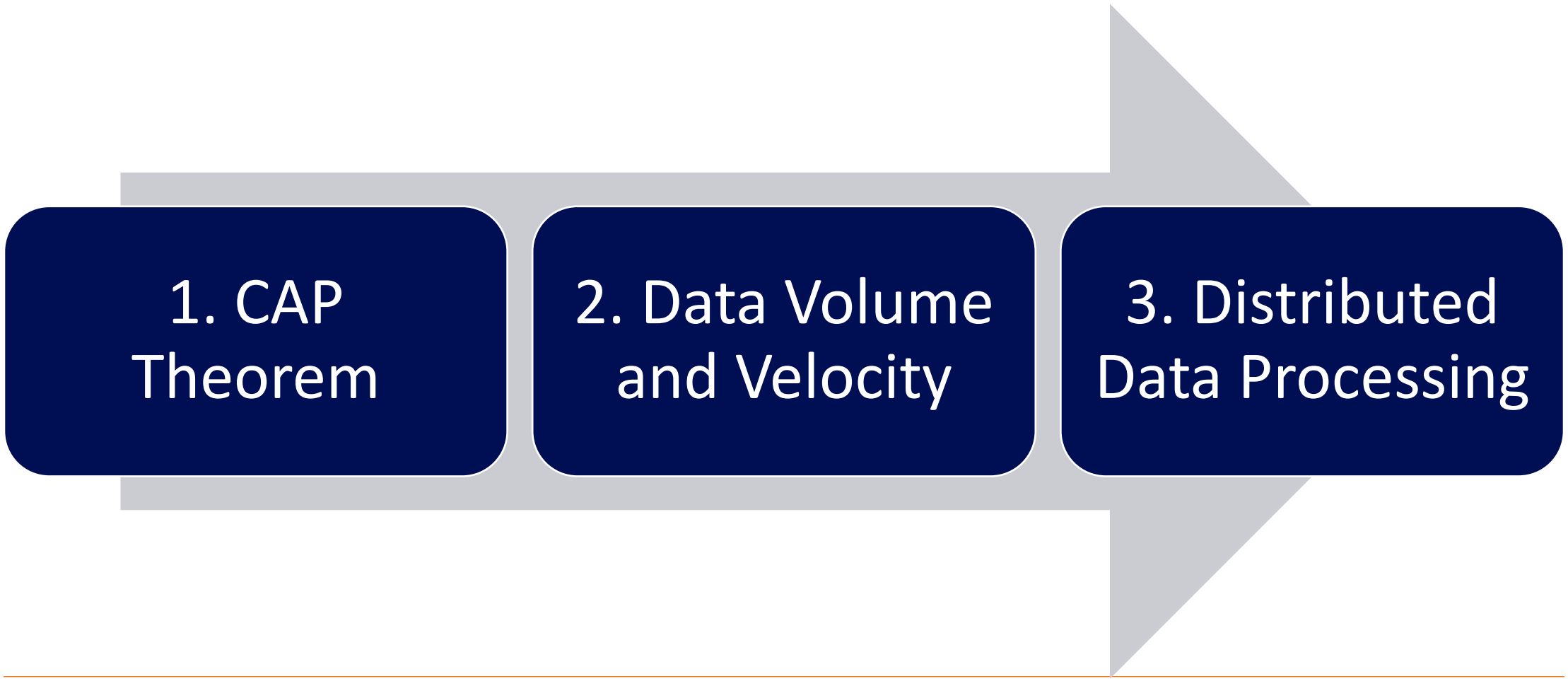
Big Data Defined

Finally, Defining Big Data

Any data set too large to be processed or stored by a single system and therefore, must be scaled horizontally.



3 Factors of Big Data

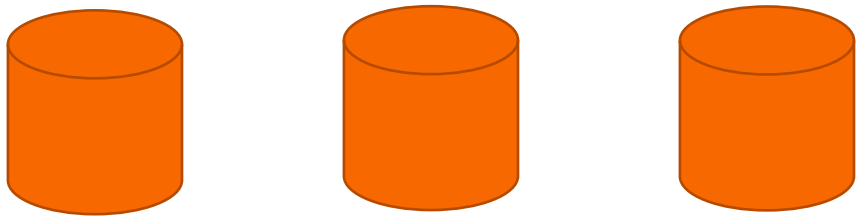


1. CAP Theorem

By definition, Big Data are distributed over more than one system.

Therefore, we require partition tolerance so will have three choices:

1. Concede consistency to guarantee availability?
2. Concede availability to guarantee consistency?
3. Use NewSQL layering of AP service discovery with a CP system?



2. Data Volume and Velocity

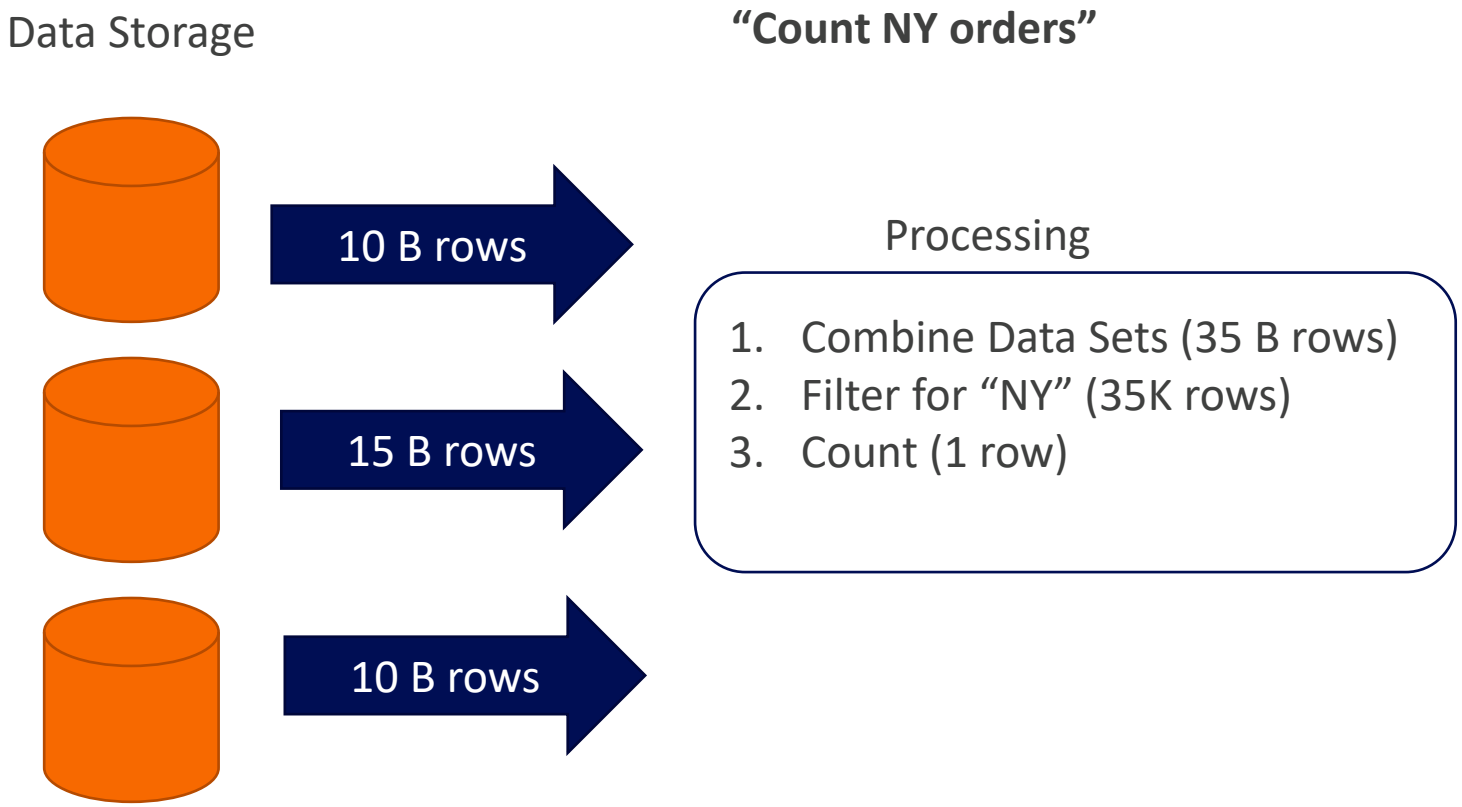
- Why are Big Data distributed?
- There are two reasons:

1. **Data Volume** The quantity of data is too large to : Diagram data set too large to fit on a single system
2. **Data Velocity.** Data is written at a rate causing a single system to be I/O bound.

3. Distributed Data Processing

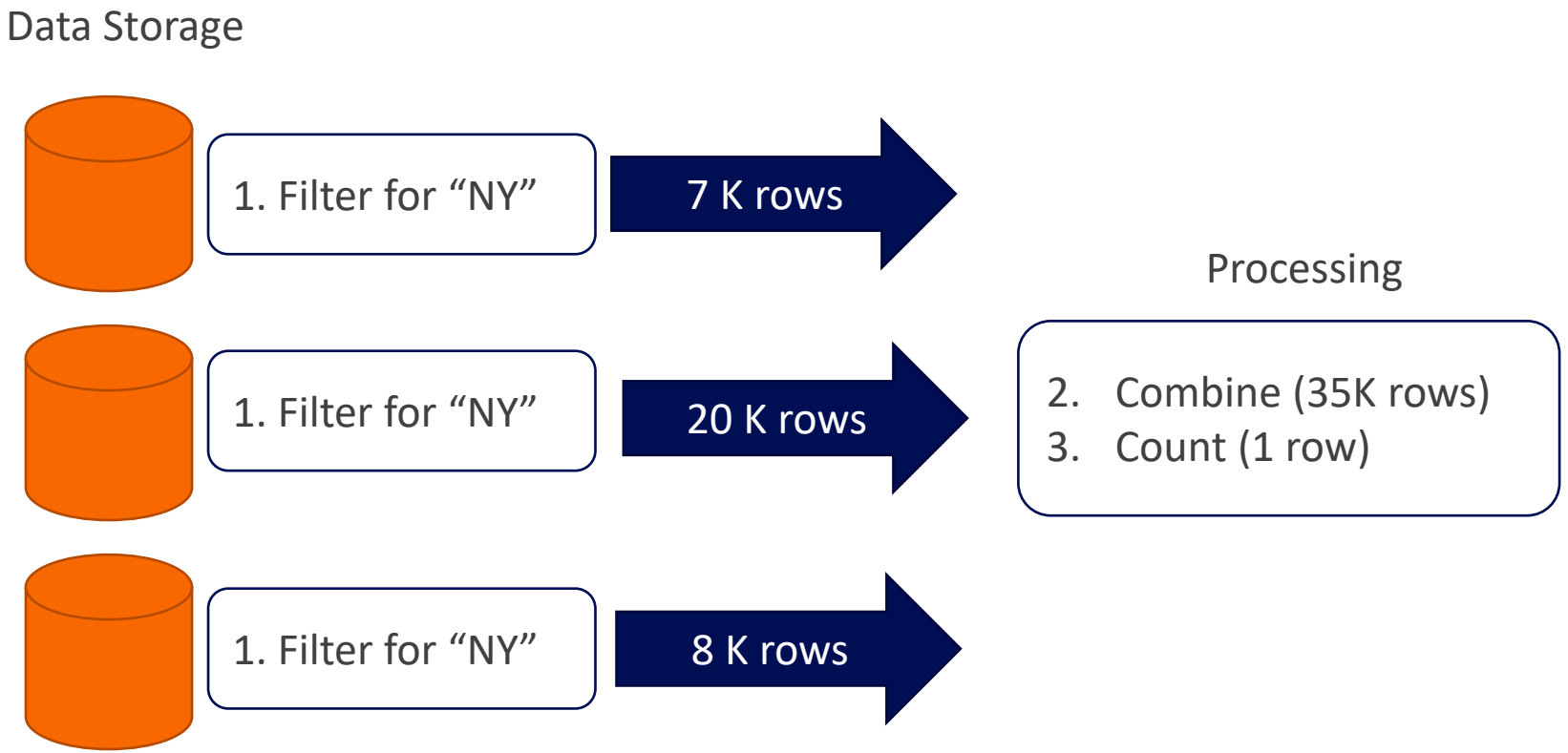
- Big Data are distributed; thus, data processing needs to be reconsidered.
- Cluster computing allows the processing to scale with the data.
- This requires different thinking from traditional data processing.
- When processing distributed data, we need to consider our actions

A Bottleneck Processing Distributed Data



- Lifting data off the data storage results in 35 B rows being processed in a single node
- This ends up being a choke point, negating the benefits of distributed data storage.

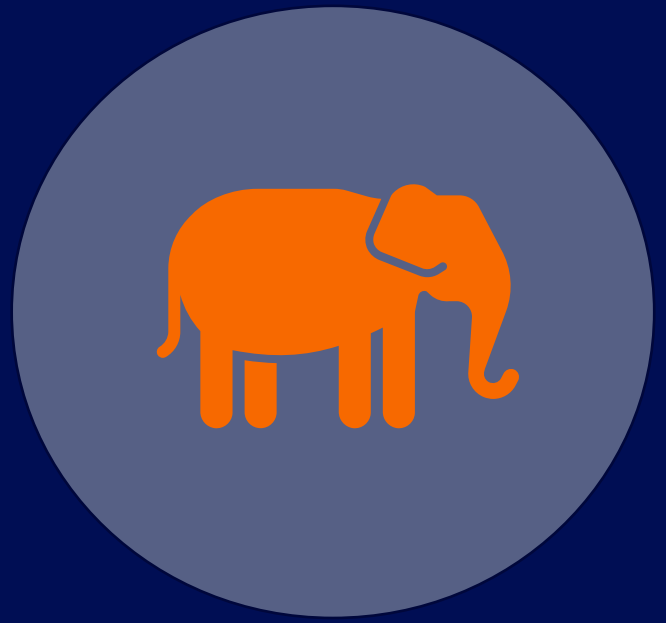
Distributed Data Processing Same Example



- Part of the processing is distributed to the data storage
- Data is filtered where its stored and now only thousands of rows are transferred over the network versus billions.

Is Your Data Big Enough?

- Don’t scale horizontally unless you must.
- Big Data systems are complex and costly to administer.
- How will you know its time? You can no longer scale up to improve performance.
- Leverage the cloud when possible, to avoid upfront hardware and administrative costs.



Hadoop

The What, Why and How

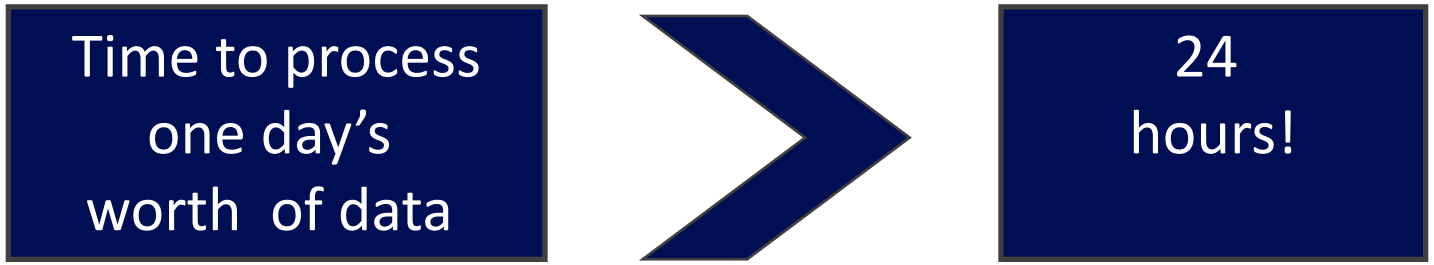
What is Hadoop?

- A System for Storing, Processing and Managing Data
- Just Like a DBMS only for “big data”
- What does *that* mean? Scales Out by design, both the data and compute to process it.

“It does look similar—but this one is powered by Hadoop”

Hadoop Origins

- The origins of Hadoop can be traced back to the internet companies of the 2000's
- Google, Facebook and Yahoo!
- These companies had a similar problem.

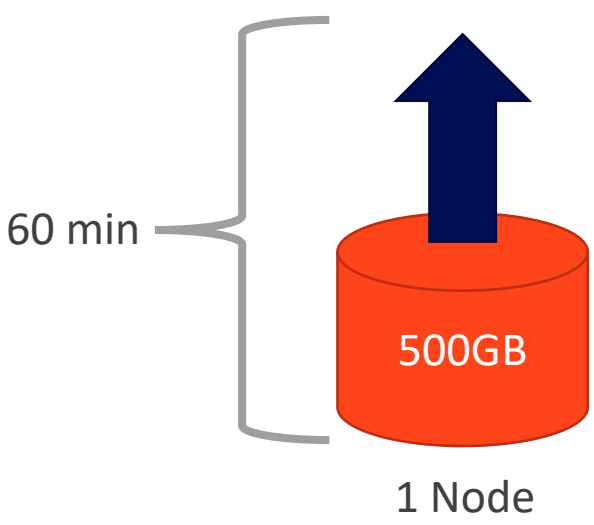


- The solution was to split the data and distribute chunks of it to be processed in parallel.

They did the math…

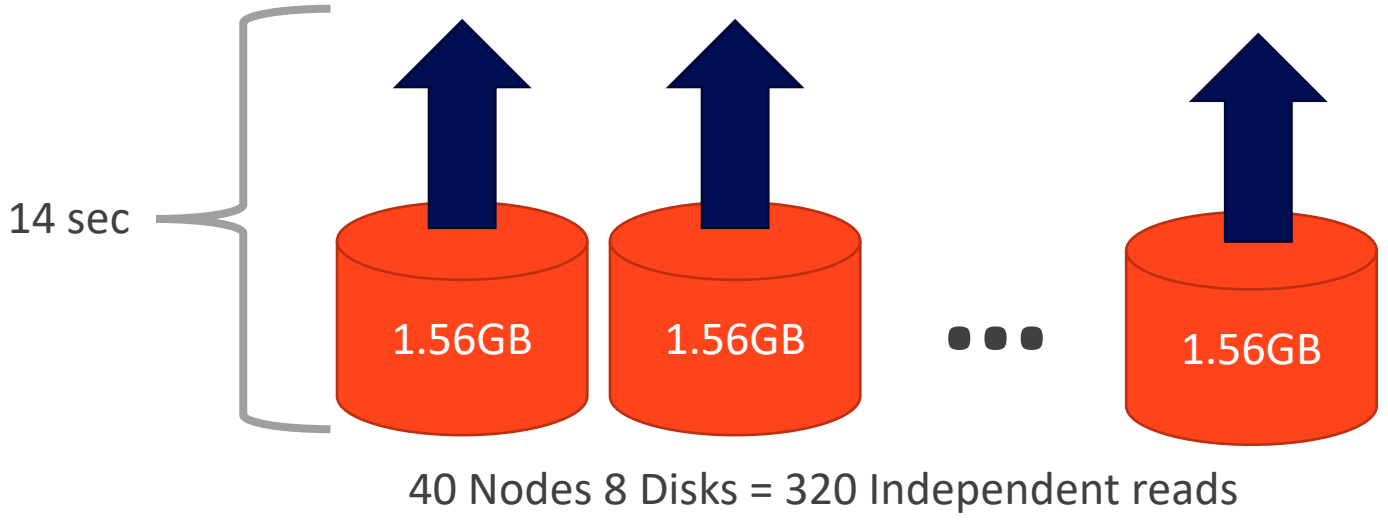
Relational DBMS

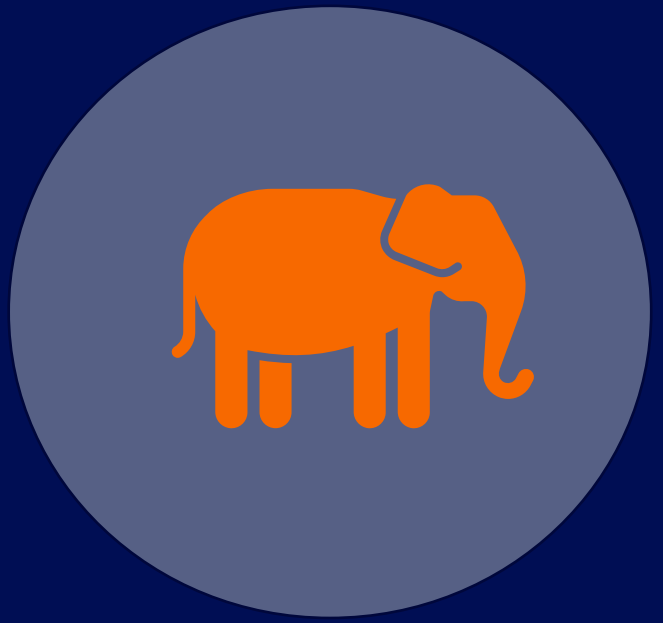
- 60 Minutes to read 500GB of data off the disk on 1 node (assuming 1Gbps)



Hadoop

- 14 seconds to read the same 500GB with 40 nodes x 8 disks per node.





How does Hadoop Work?

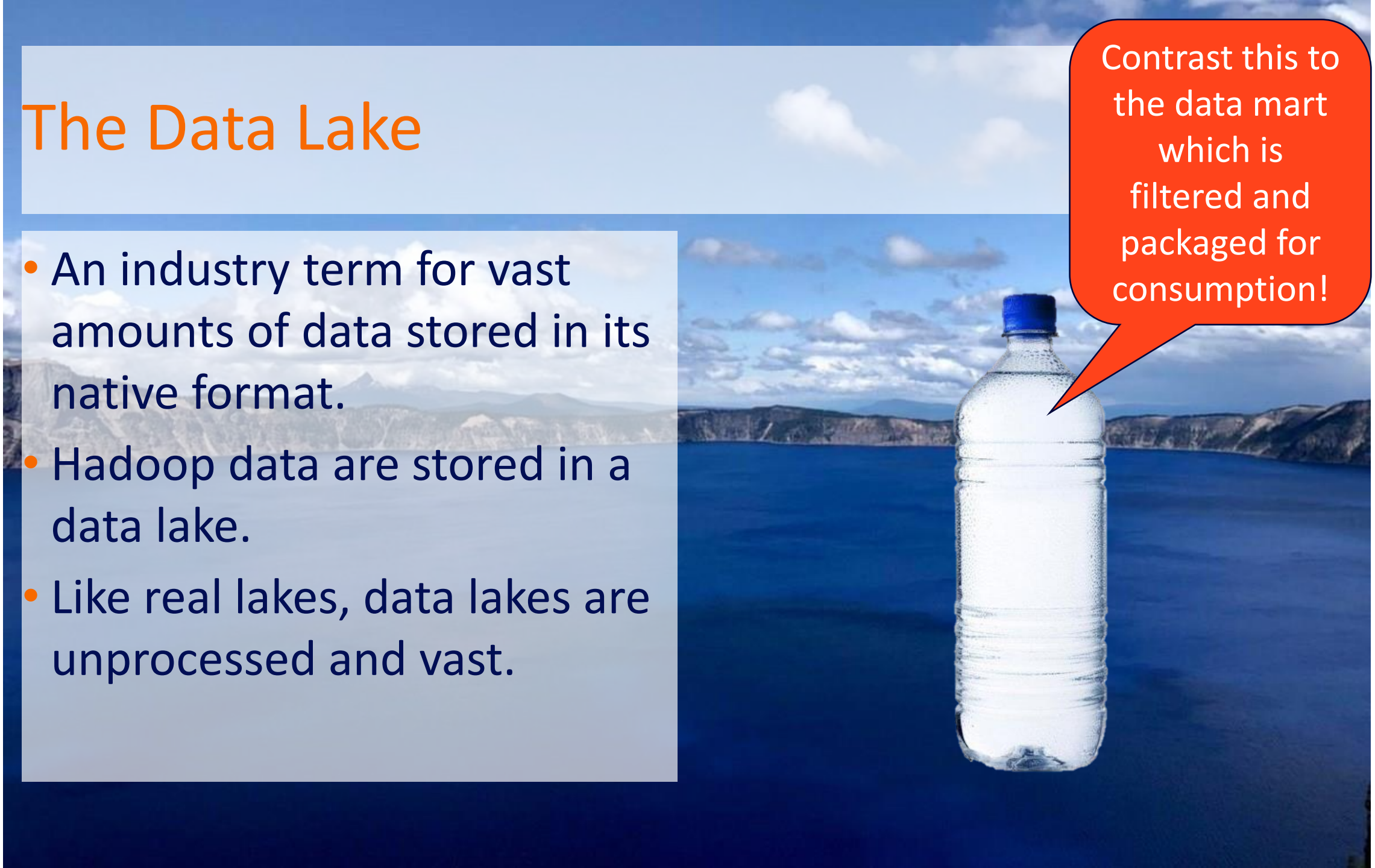
Let’s Explore the concepts of Hadoop (Data Lake, Schema On Read)

The Hadoop Philosophy is Simple

- Capture data as is and store it do not create a schema for it.
- Do not transform the data before storing it. We don't know what we plan to do with it yet!
- Keep everything. You never know what you will need when!
- When you retrieve the data, we then apply a schema and transform it to suit our needs. (Schema on Read)
- This is very different from relational modeling where we are not able to store data without creating a table first.

The Data Lake

- An industry term for vast amounts of data stored in its native format.
- Hadoop data are stored in a data lake.
- Like real lakes, data lakes are unprocessed and vast.



Collect and Store Data into the Data Lake

Request

```
GET https://www.googleapis.com/youtube/v3/commentThreads?part=snippet&maxResults=25&videoId=O4PXqpv8TAw&key={YOUR_API_KEY}
```

Response


```
200 OK

- Show headers -

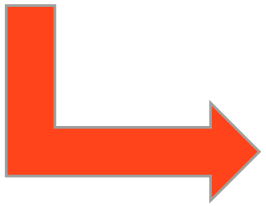
-{
  "kind": "youtube#commentThreadListResponse",
  "etag": "\"q5k97EMVGxODeKcDgp8gnMu79wM/lqCaY8zGX8gKVG6NRyNEbR3oSE8\"",
 -"pageInfo": {
   "totalResults": 16,
   "resultsPerPage": 25
  },
 -"items": [
  -{
    "kind": "youtube#commentThread",
    "etag": "\"q5k97EMVGxODeKcDgp8gnMu79wM/3uP8cTARSOm44LzKPV_LIHj5v8Q\"",
    "id": "z13si3gb1r3wdl0wi04ccjsoww2wetoovoc0k",
   -"snippet": {
     "videoId": "O4PXqpv8TAw",
    -"topLevelComment": {
      "kind": "youtube#comment",
      "etag": "\"q5k97EMVGxODeKcDgp8gnMu79wM/MZKledJ37NuMaz2UxwOIvaHqZgk\"",
      "id": "z13si3gb1r3wdl0wi04ccjsoww2wetoovoc0k",
     -"snippet": {
```


You know there is value in your Organization's YouTube channel, but you don't know what exactly that might be.

You extract the API output weekly & load the data into your Hadoop Data lake.

Apply a Schema on Read/Query


```
},
"videoId": "O4PXqpv8TAw",
"textDisplay": "excellent video!\ufeff",
"authorGoogleplusProfileUrl": "https://plus.google.com/100257256306278377832",
"canRate": true,
"viewerRating": "none",
"likeCount": 0,
"publishedAt": "2015-02-02T02:05:16.042Z",
"updatedAt": "2015-02-02T02:05:16.042Z"
```

```
create external table youtube.comments (
    videoId STRING,
    likes INT,
    text STRING,
    pub_date STRING
) location '/user/mafudge/youtubecomments/';
```

videoId	likes	text	pub_date
O4PXqpv8TAw	0	excellent video!\ufeff	2015-02-02T02:05:16.042Z
…	…	…	…

When we want to perform analysis on the data, we add a schema when the data are read.

This will create a table of data, like a relational table.

Analyze Your Data

videoId	likes	text	pub_date
O4PXqpv8TAw	0	excellent video!\ufeff	2015-02-02T02:05:16.042Z
…	…	…	…

```
select videoId, sum(likes)
    from youtube.comments
    group by videoId
    order by sum(likes) desc
    limit 10;
```

videoId	sum(likes)
O4PXqpv8TAw	56
V56Xqyy-2wq	14
…	…

Export/ Visualize

The table is not stored. It is a program that adds schema when we execute the actual analytics.
In this case, an SQL SELECT statement, but it could be any code.

What Exactly Is "Schema on Read"? Again?

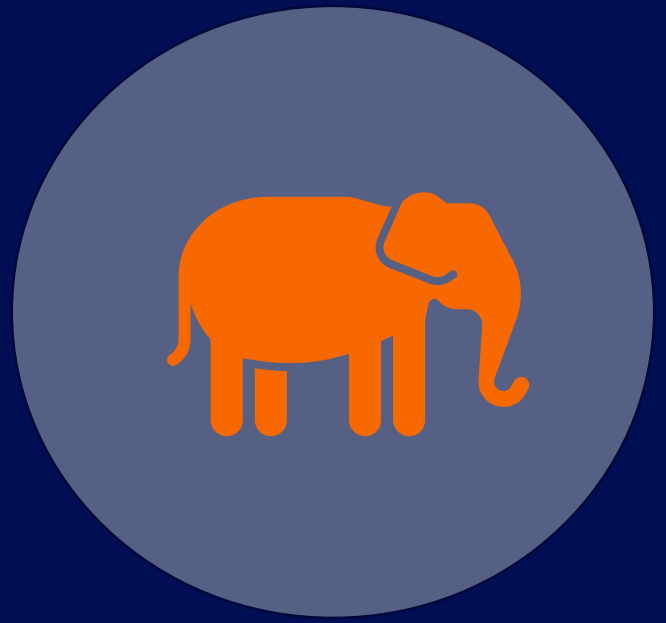
Traditional RDBMS

- You cannot write data without a table
- Cannot insert data unless data fit into table's *single design*
- Large up-front design costs
  - Conceptual models
  - Table design
- "Schema on write"

Hadoop

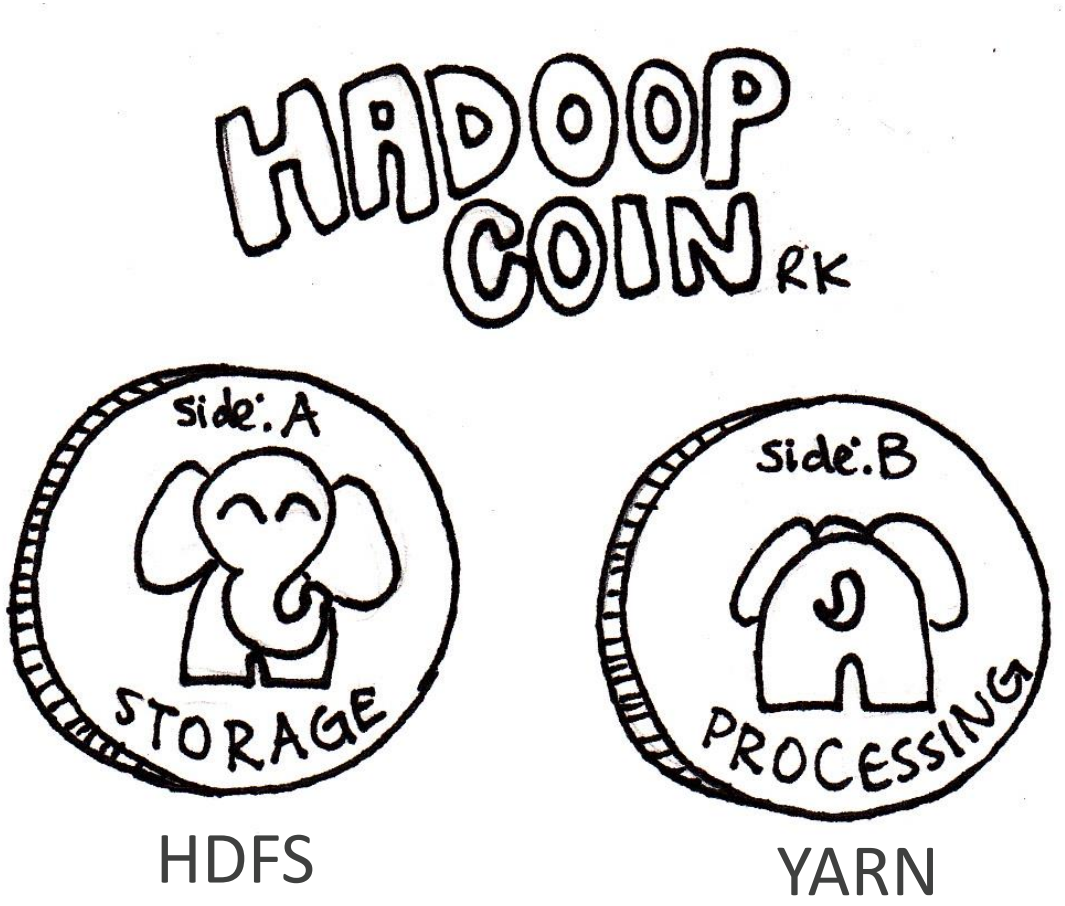
- You write the data "as they are" to the Data Lake
- Schema applied when data are read—*multiple designs*
- Very little up-front design costs
  - Just write to disk
  - Apply schema when you need it
- "Schema on read"

Hadoop Architecture



Two Essential Components of Hadoop

1. Distribute the data across nodes in the data lake by splitting it up.
   HDFS does this.
2. Move the compute to where the data lives.
   MapReduce/YARN does this.



HDFS YARN

Typical Hadoop Cluster

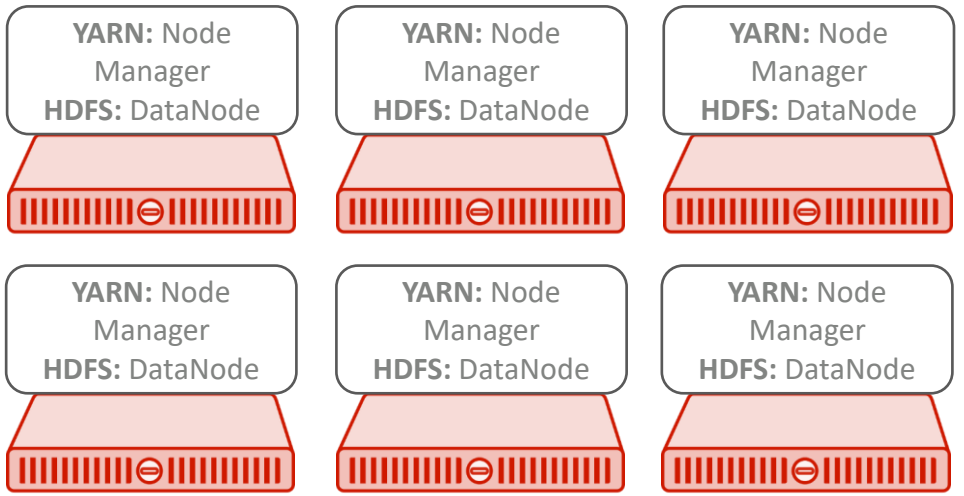
Master Nodes

- Manages the Hadoop infrastructure
- Runs *one* of each of these services per cluster, on a single server or many
- Should run on server-class hardware
- Very few of these nodes

YARN:
App Timeline Server,
Resource Manager,
History Server
HDFS:
NameNode

Worker Nodes

- Store data and perform processing over it
- Each node runs the same services
- Runs on commodity hardware
- Lots and Lots these nodes

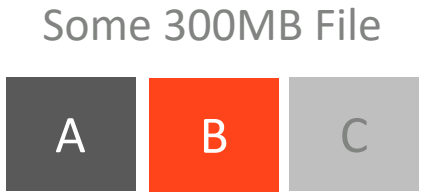


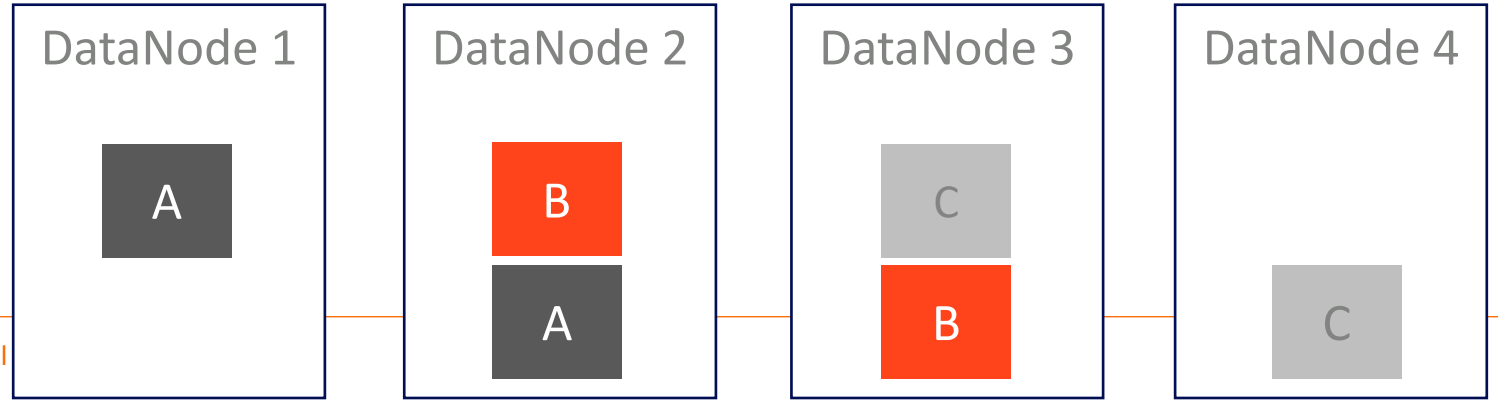
HDFS

Hadoop Distributed File System

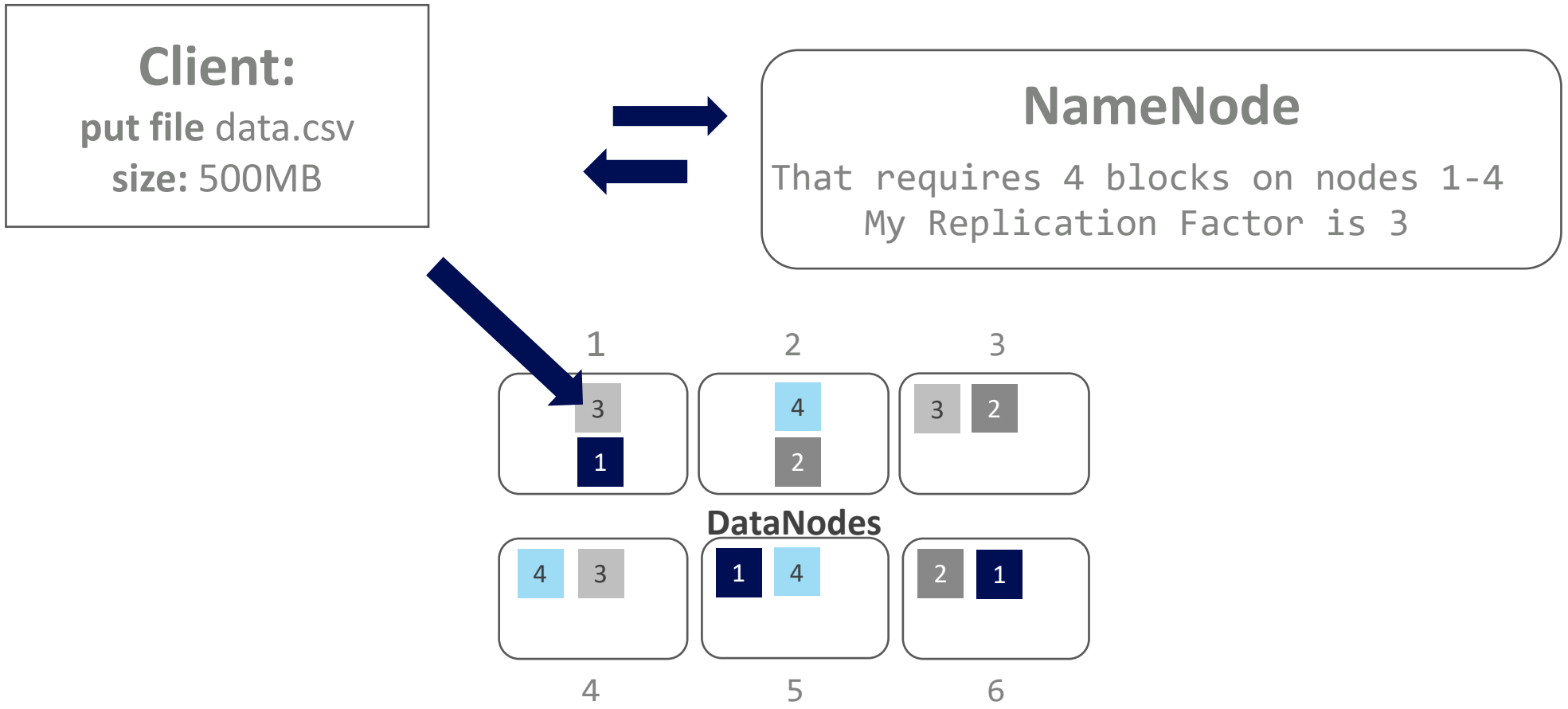
HDFS – Hadoop Distributed File System

- Based on Google's GFS – Used by search.
- CP System: NameNode master + several DataNode workers.
- Distributed data storage paradigm
- A single file is divided into 128MB blocks
- Blocks are spread across nodes
- Blocks are written multiple times for redundancy

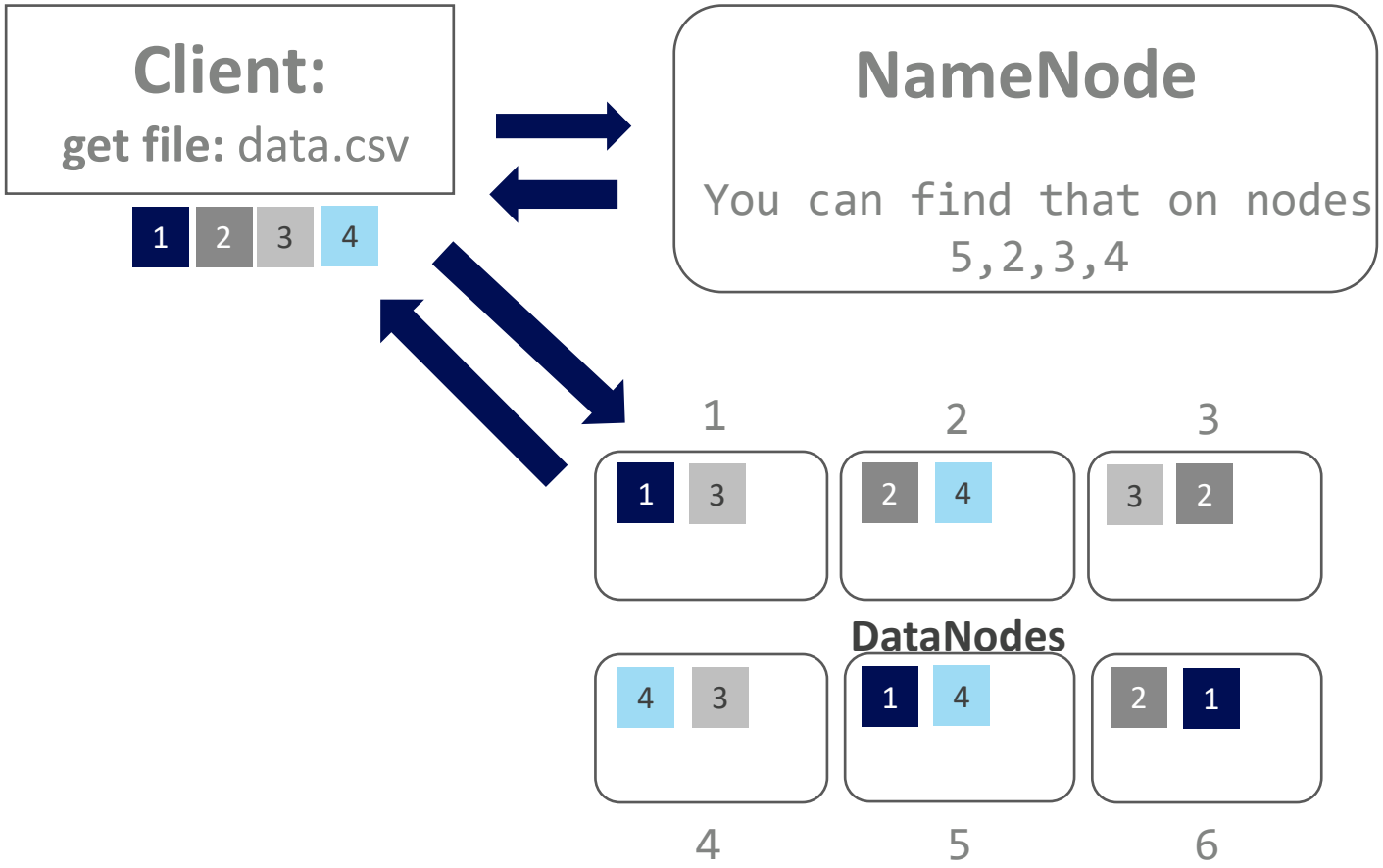




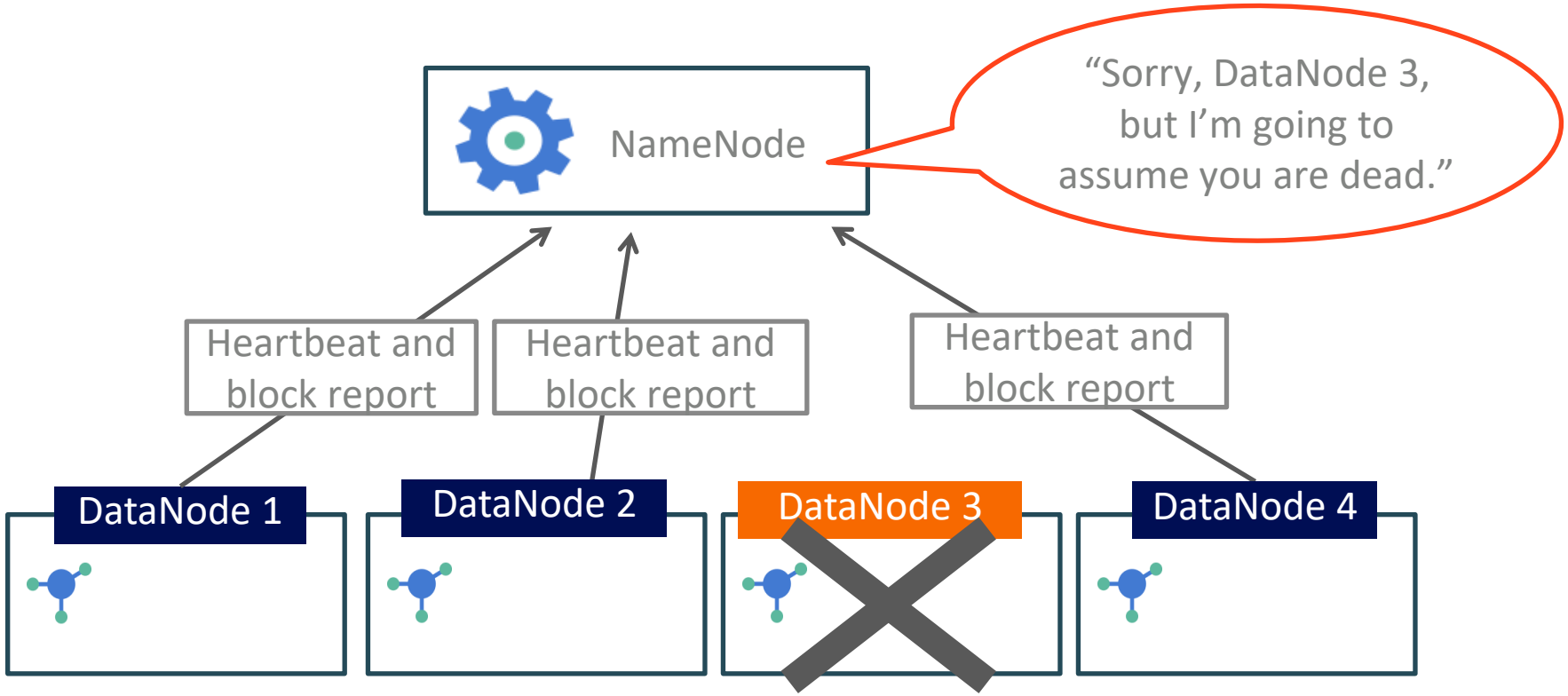
Example: HDFS Write



Example: HDFS Read



Example: HDFS Heartbeat and Block Report



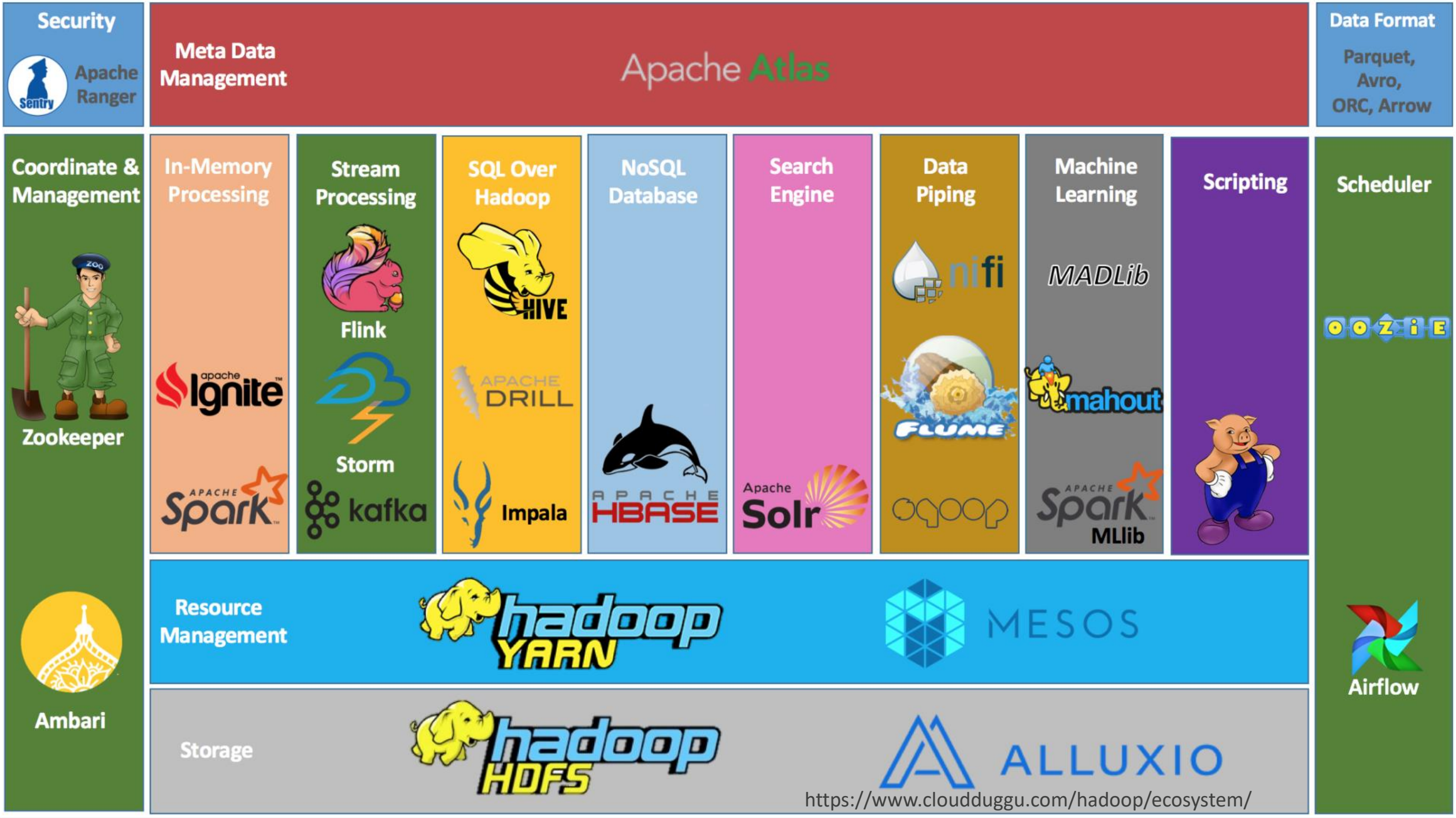
- HDFS DataNodes send routine heartbeats and block reports to the NameNode.
- This is how the NameNode knows where the blocks of a file are located.

YARN (and MapReduce)

Yet Another Resource Negotiator

YARN – Hadoop 2.0

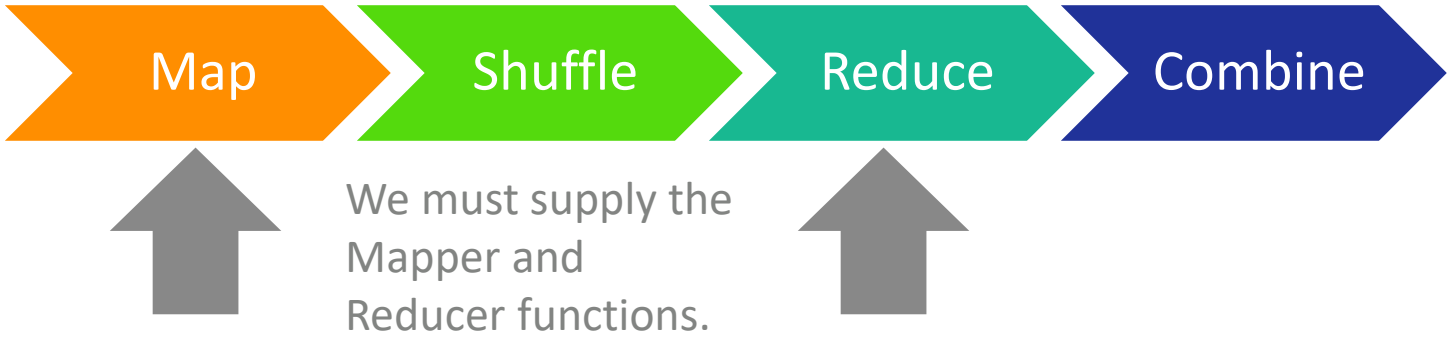
- YARN is a job executor and resource schedule for Hadoop.
- MapReduce is a distributed batch process algorithm
- Hadoop v1 could ONLY do Map-Reduce
- Hadoop v2 added YARN so any distributed application could run
- YARN provides an API so that you can build distributed applications on Hadoop.
- YARN made applications like Spark and Hive possible.
- Map-Reduce is now just one of the many applications which run on YARN



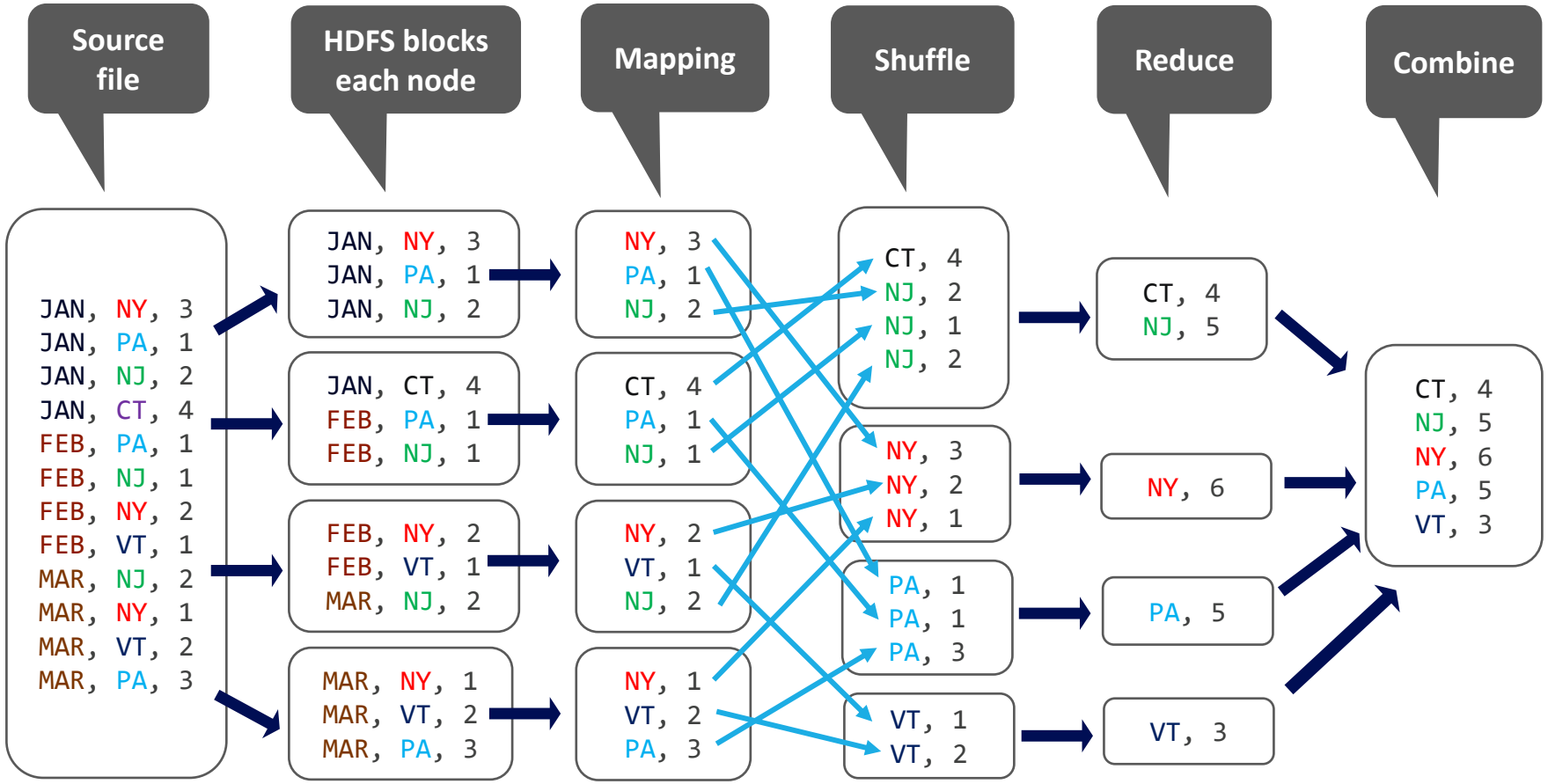
https://www.cloudduggu.com/hadoop/ecosystem/

MapReduce Algorithm

- Understanding MapReduce will help you understand distributed processing.
- Phases
  - Map → apply a transformation to a data set
  - Shuffle → transfer output from Mapper to Reducer nodes
  - Reduce → aggregate items into a single result
  - Combine → output of Reducer nodes into single output



Example MapReduce: “Total Orders by State”



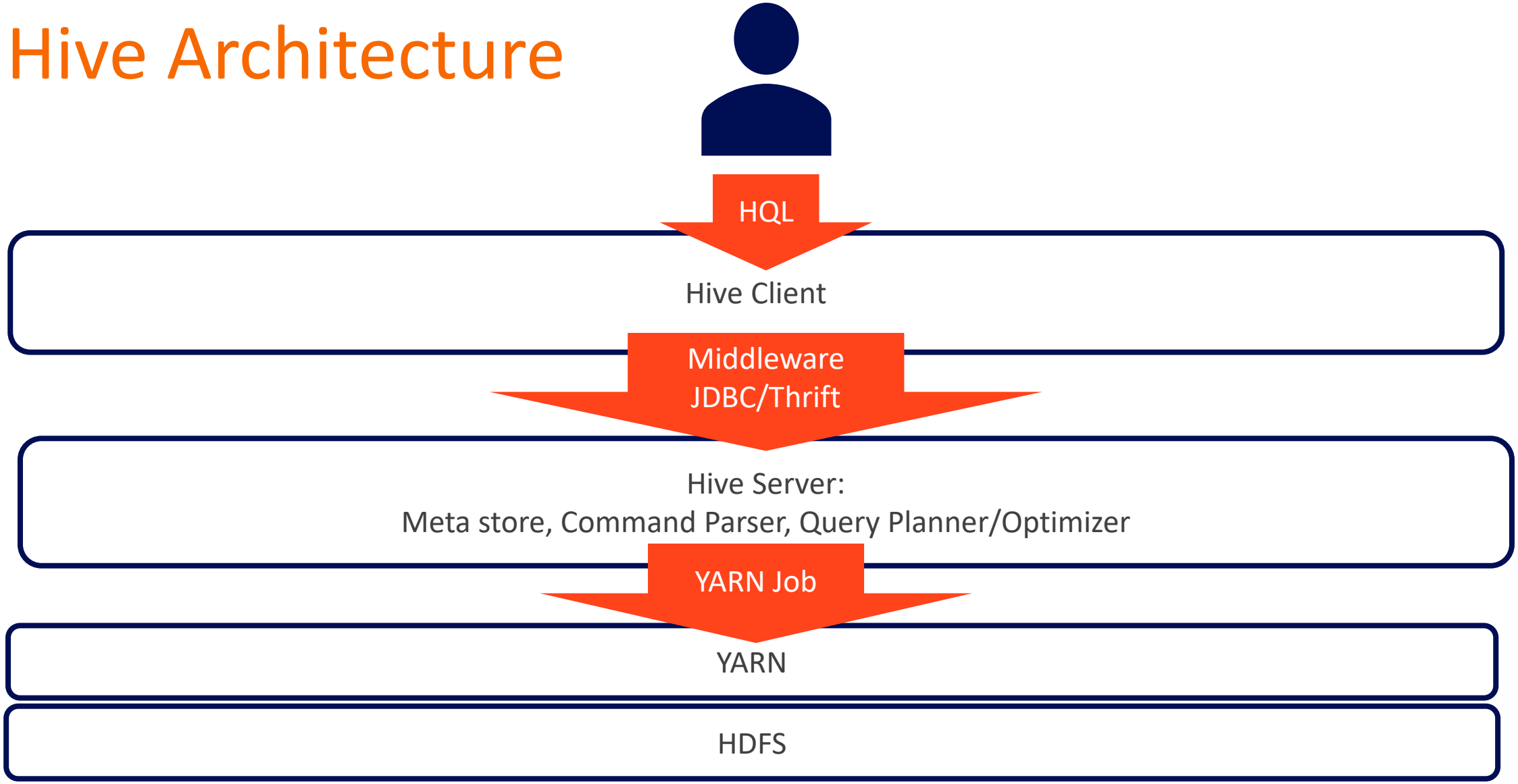
HIVE

SQL Over Hadoop HDFS

What is HIVE?

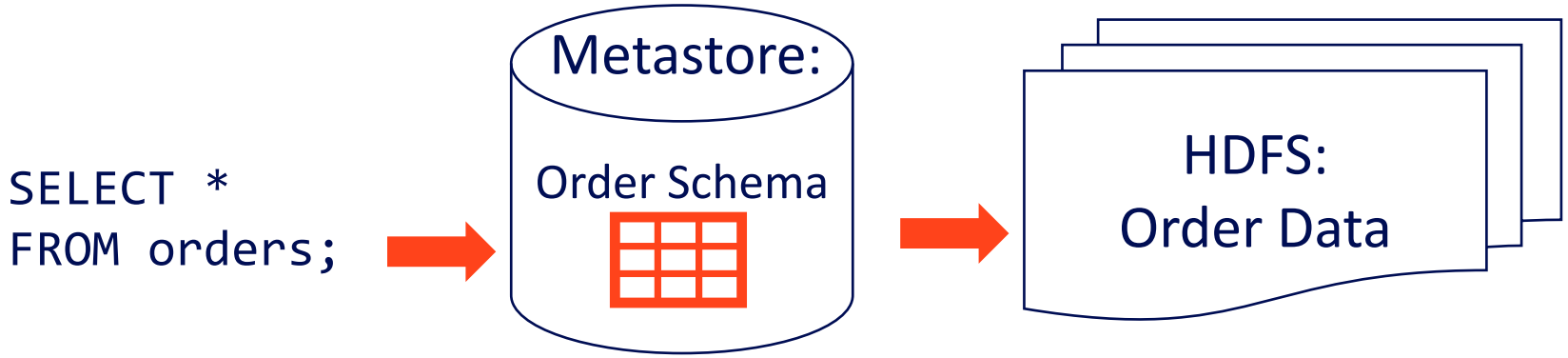
- SQL on Hadoop
- Apply a table schema over the data in HDFS
- Schema is not persistent – its “schema on read”
- Not standard SQL, but HiveQL – HQL
- Developed by Facebook, Now Apache Foundation
- Hcatalog / Hive metastore provides table access to other tools in the Hadoop ecosystem.

Hive Architecture



Hive Metastore

- The Hive metastore holds schema / metadata information for hive database objects:
  - Databases
  - Tables
  - Views
  - Indexes
- The metastore is commonly an RDBMS
- The actual data itself are stored on HDFS.



Two Types of Hive Tables

Internal (Hive-Managed)

- Data and Schema are managed by the Hive, like an RDBMS
- Data are moved into a special HDFS folder /user/hive/warehouse
- Drop table → deletes data
- Insert into table via Hive SQL

External

- Table over existing HDFS location
- Only the schema is managed by the Hive
- Drop table → deletes the schema only
- Insert into table by adding files to HDFS

Query Hive Like SQL!

- Once tables are created, they can be queried with SQL

```
SELECT * FROM customers;
```

```
FROM customers
  SELECT firstName, lastName, address, zip
  WHERE orderID > 0
  ORDER BY zip;
```

```
SELECT c.*, o.*
  FROM customers c
  JOIN orders o ON
     c.customerID = o.customerID;
```

Spark

Lightning-Fast Cluster Computing

What is Spark?

- General-purpose distributed data processing / computing engine for large-scale data analytics
- Developed at UC Berkeley AMP Lab, Now an Apache project, and heavily funded by Databricks.
- Designed from the ground-up to support distributed data processing.
- Stores working data in memory for fast processing.
- Best suited machine learning tasks and interactive / exploratory analytics

Spark: Key Features

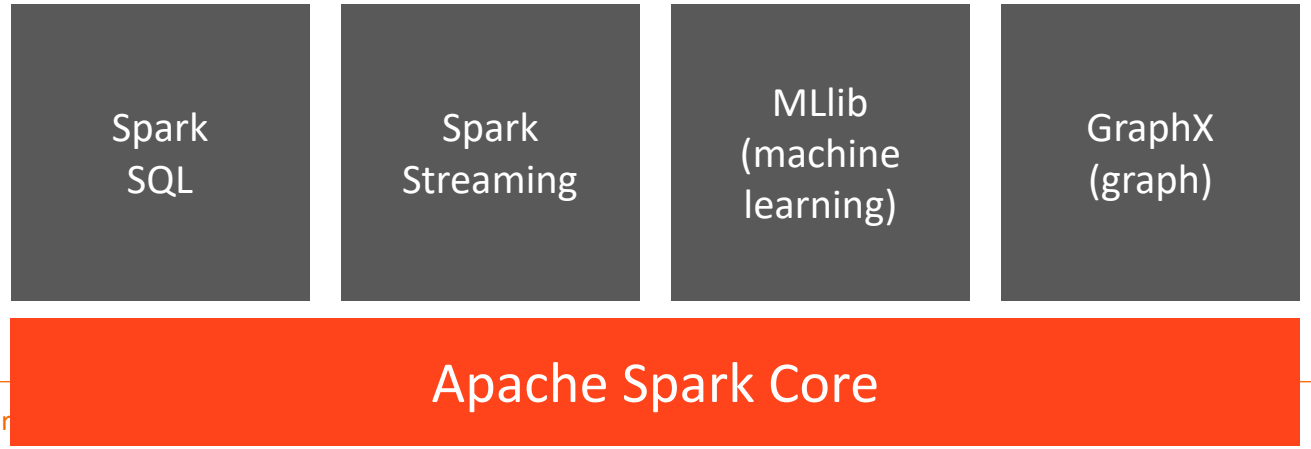
- Underlying API supports multiple language bindings
  - Scala, Java, Python, R and SQL.
  - You can use the APIs of these languages in Spark, provided the libraries are installed on the nodes in the cluster.
- Uniform data access.
  - Connect to any data source and manipulate it the same way courtesy of the RDD (Resilient-Distributed Data) and Data Frames APIs.
  - Makes data manipulations consistent regardless of where the data came from.

Spark Use Cases

- Interactive/exploratory analytics including machine learning tasks
- Real-time analytics and event processing
- Building data pipelines and Extract-Transform Load pipelines
- Machine learning on big data, creating ML pipelines

Spark Libraries

- Four libraries built on Spark Core
  1. **MLlib:** a machine learning library, similar to sklearn
  2. **Spark Streaming:** enables high-throughput, fault-tolerant stream processing of live data streams
  3. **Spark SQL:** runs SQL and HiveQL queries, Dataframes
  4. **GraphX:** an API for graphs and graph-parallel computation



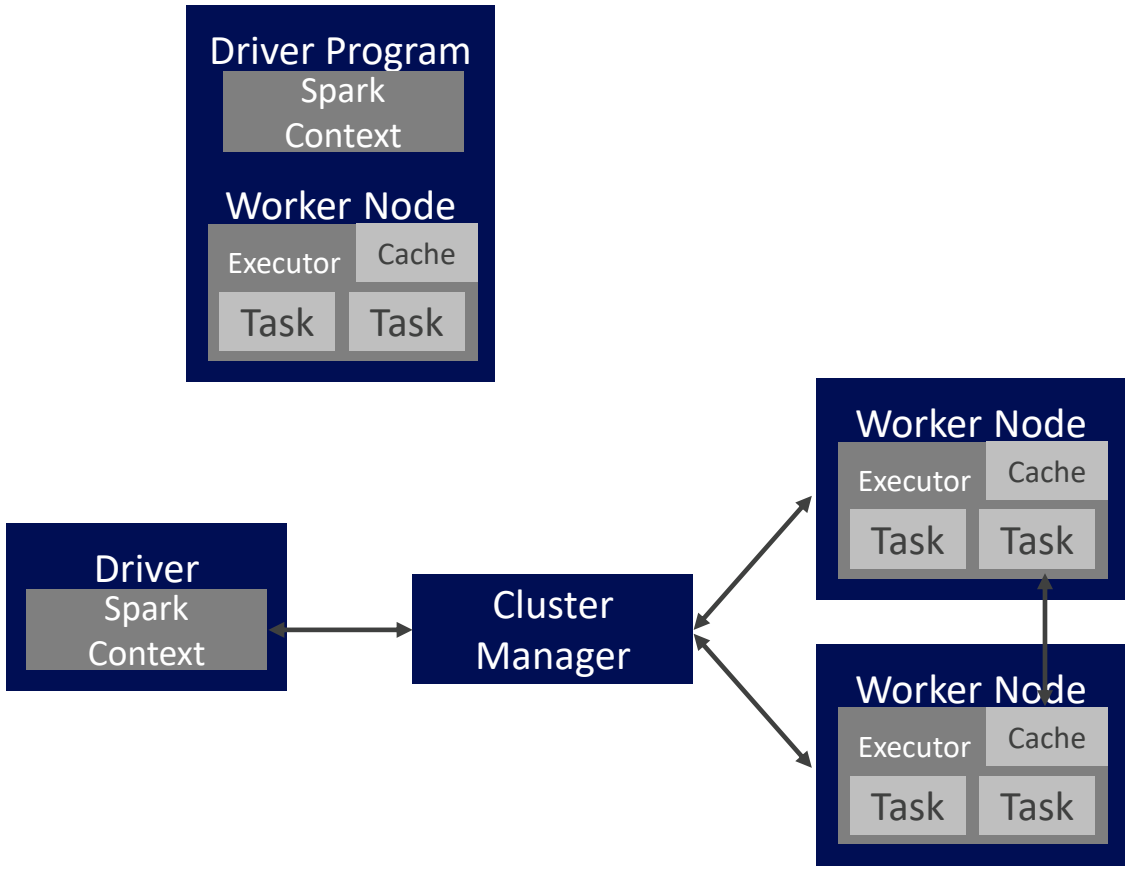
Spark Execution modes

Client

- Code executes locally
- Behaves identically to cluster mode
- Limited to the computing capacity of the running machine

Cluster

- Code executes on cluster.
- Supports YARN, Mesos, Kubernetes or standalone spark cluster
- Horizontally scalable.



PySpark Programming

Spark programming in Python

Essential Spark Concepts

- Resilient Distributed Datasets (RDDs)
  - Basic building block of all spark storage.
  - Abstracts away the need to worry about where data is and whether its available.
- Lazy Evaluation
  - No RDD transformations take place until we execute a command that requires output
- Directed Acyclic Graph (DAG)
  - The execution plan generated from the spark code is finite and deterministic.
  - Your code does not execute in the order written.

Creating A Spark Session

```
1  spark = SparkSession \
2      .builder \
3      .master("local") \
4      .appName('jupyter-pyspark') \
5          .config("hive.metastore.uris",
6                  "thrift://hive-metastore:9083") \
7          .enableHiveSupport() \
8      .getOrCreate()
```

"local" means client mode

appName is useful for cluster deployments

config() can be used multiple times

Spark Jars

- Spark Jars allow us to plug in additional spark libraries and integrations.
- Search here for Jars https://maven.org
- What you need to know:
  - Our Spark Version: 3.1.2
  - Our Scala Version: 2.1.2
  - Package String is the gradle DSL implementation
- Add package string to spark session:

```
.config("spark.jars.packages","gradle-package-string")\
```

Lazy Evaluation and the DAG

- Lazy evaluation assures nothing executes until a result it yielded
- Code is optimized to an execution plan. These two use the same plan.

```
a = grades.filter("year = 2016")\
    .filter(grades.Semester == "Fall")\
    .sort("Course") \
    .select("Course", grades.Credits, grades["Grade"])
```

```
a.explain()
```

```
== Physical Plan ==
*(2) Sort [Course#489 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(Course#489 ASC NULLS FIRST, 20
   +- *(1) Project [_c2#479 AS Course#489, _c3#480 AS Credit
      +- *(1) Filter (((isnotnull(_c0#477) AND isnotnull(_c1
         +- FileScan csv [_c0#477,_c1#478,_c2#479,_c3#480,_c
t: CSV, Location: InMemoryFileIndex[file:/home/jovyan/datase
ull(_c1), EqualTo(_c0,2016), EqualTo(_c1,Fall)], ReadSchema:
```

```
b = grades.sort("Course") \
    .filter(grades.Semester == "Fall")\
    .select("Course", grades.Credits, grades["Grade"])\
    .filter("year = 2016")
```

```
b.explain()
```

```
== Physical Plan ==
*(2) Sort [Course#489 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(Course#489 ASC NULLS FIRST, 20
   +- *(1) Project [_c2#479 AS Course#489, _c3#480 AS Credit
      +- *(1) Filter (((isnotnull(_c1#478) AND isnotnull(_c0
         +- FileScan csv [_c0#477,_c1#478,_c2#479,_c3#480,_c
t: CSV, Location: InMemoryFileIndex[file:/home/jovyan/datase
ull(_c0), EqualTo(_c1,Fall), EqualTo(_c0,2016)], ReadSchema:
```

Data Storage Options for Spark

- HDFS seems like the logical choice, but:
  - Requires Hadoop and YARN
  - Requires Spark cluster running on YARN
  - You want to use Spark but don't want to setup and manage Hadoop!
- Hive Integration
- Object Storage
- Distributed NoSQL databases
  - MongoDb, Elasticsearch, Cassandra, etc…

Hive Integration with Spark

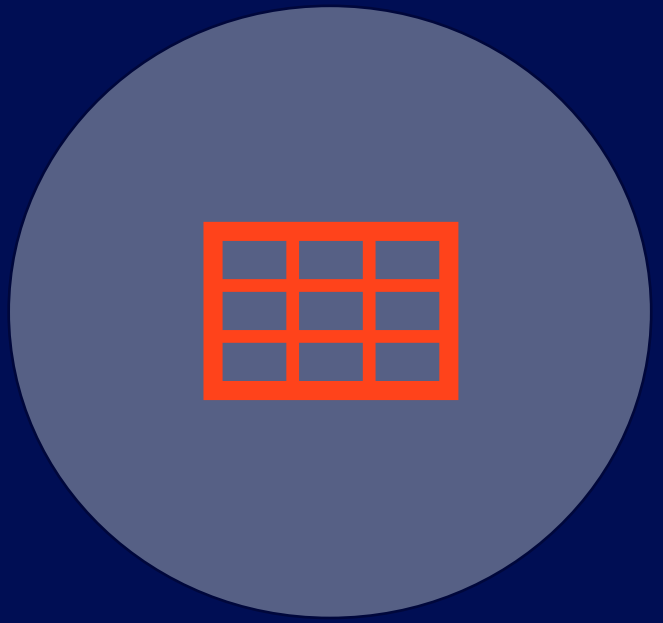
- We can use spark.sql to execute hive queries when enableHiveSupport() is configured.

```
        .config("hive.metastore.uris",
                "thrift://hive-metastore:9083") \
        .enableHiveSupport() \
```

- We Need to use .show() to force lazy evaluation of the SQL!
- We can read HDFS data through Hive but not directly. If its not a Hive table, its not available in Spark.

What is Object Storage?

- Cloud-based Network-Attached storage
- Client uses an API such as the AWS S3 API to get and put objects.
- Popular Object Storage Providers
  - AWS S3 (Simple Storage Service)
  - Azure Blob Storage
  - Google Cloud Storage
  - Minio (open source)
- Very popular storage option for spark in the cloud.
- Even easier with delta lake https://delta.io/



Spark DataFrames API

Declarative API for Working with Tables of Data

Understanding Spark File Paths

- Spark `can` read from/ write to files at variety of locations.
- Locations:
  - `file://` local (on spark client)
  - `s3a://` object storage (Minio / S3 / Azure Blob)
  - `hdfs://` Hadoop HDFS
  - `webhdfs://` Hadoop over Web HDFS
- The path can be a filename, folder, or wildcard
  - `stocks.csv` | `stocks/` | `stocks/jan*.csv`
- Example:
  - `hdfs://namenode/path/to/files`

Reading Data, the Spark Way

- It doesn’t make sense to load a local file with the client, especially when you are running spark in cluster mode.
- The **SparkFiles** module allows us to load the files into the temporary storage on the worker nodes of the spark cluster.
- We can also read data over HTTP using this method.

```
from pyspark import SparkFiles
spark.sparkContext.addFile("https://raw.githubusercontent.com/mafudge/datasets/master/stocks/stocks.csv")
file_on_spark = SparkFiles.get("stocks.csv")
```

Spark File Formats

- Spark can read / write data in a variety of formats.
  - **csv** delimited (comma, tab, etc) file
  - **text** generic text file, one row per line
  - **json** JSON format
  - **parquet** Parquet format (common big-data format with schema included)
  - **orc** Another common big-data format with schema.
- Each format has configurable options.
- https://spark.apache.org/docs/latest/sql-data-sources.html

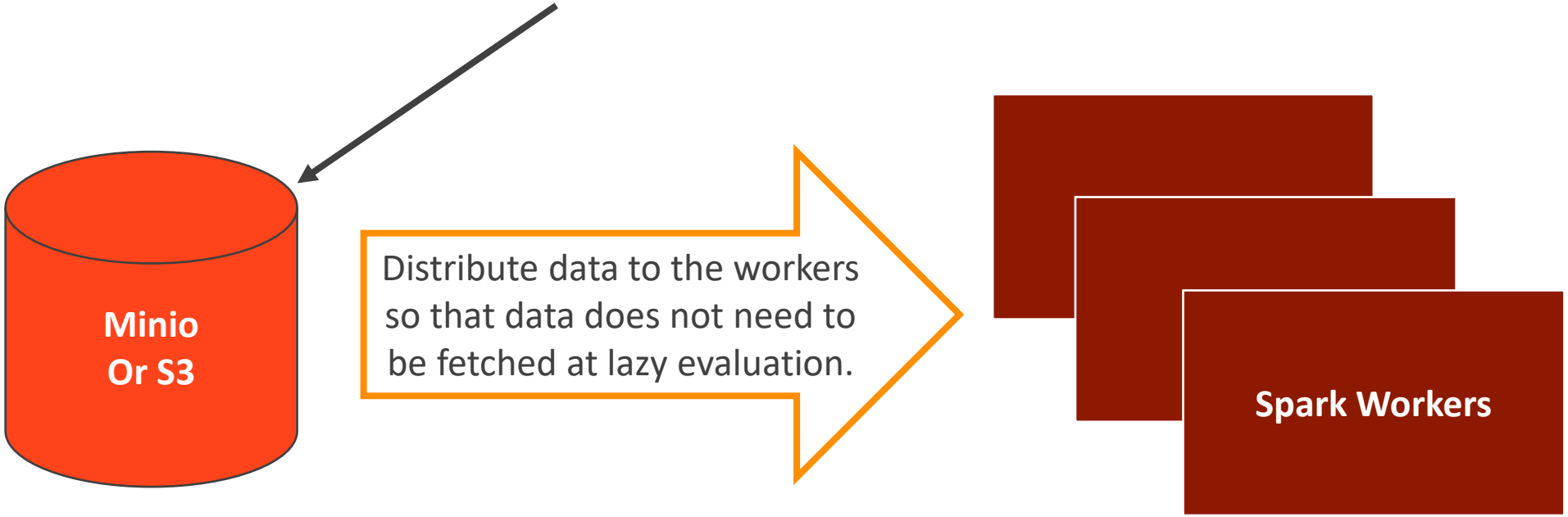
DataFrame Schemas

- Every spark dataframe has a schema, or collection of typed columns. The schema is stored in a **StructType**
- Each column is a **StructField** consisting of the field name and a specific **StructType**. Int, decimal, string, date, etc..
  - When you **spark.read** data, from the schema is always the most flexible type, String.
  - When you include the **inferSchema** option, and extra pass is made over the data to infer the `StructType` for each column.
- For formats that include a schema, like `parquet` or `orc` the schema in the file is loaded.

```
df = spark.read.csv("/home/jovyan/datasets/stocks/stocks.csv", inferSchema=true, header=True)
```

What does .cache() do?

```
spark.load.csv("s3a://bucket/file").cache()
```



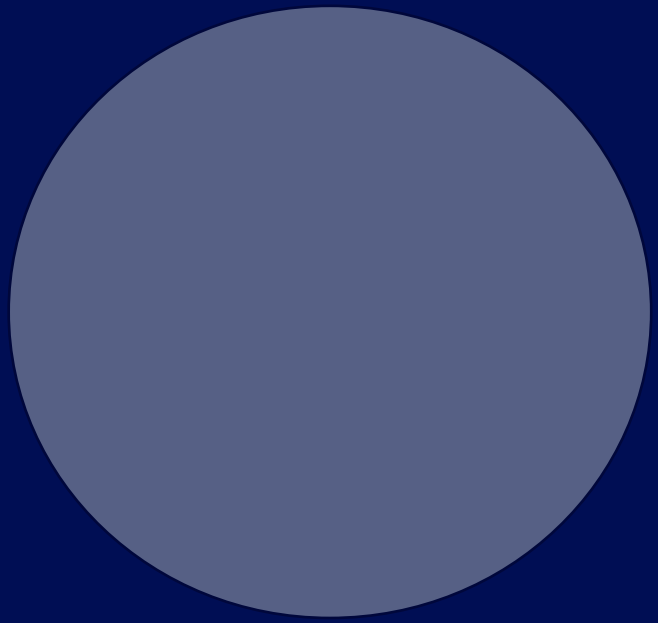
Example: Schema, inferSchema and Cache

```
df = spark.read.csv("file:///home/jovyan/datasets/ufo-sightings/*.csv",header=True, inferSchema=True).cache()
df.printSchema()
df.show(5)
```

```
root
 |-- Date / Time: string (nullable = true)
 |-- City: string (nullable = true)
 |-- State: string (nullable = true)
 |-- Shape: string (nullable = true)
 |-- Duration: string (nullable = true)
 |-- Summary: string (nullable = true)
 |-- Posted: string (nullable = true)
```

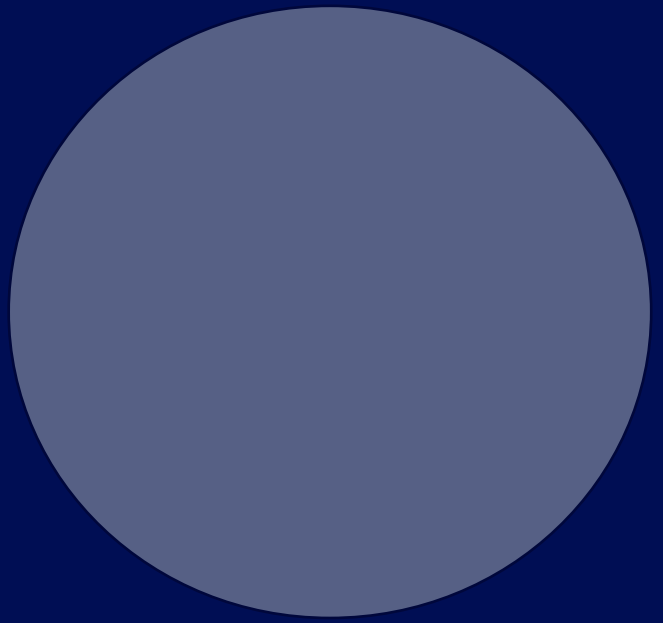
```
+-------------+--------------------+-----+--------+-------------+--------------------+--------+
|  Date / Time|                City|State|   Shape|     Duration|             Summary|  Posted|
+-------------+--------------------+-----+--------+-------------+--------------------+--------+
|2/29/16 23:45|        Harbor Beach|   MI|   Light|     1 minute|Yellow/white ball...|3/4/2016|
|2/29/16 23:30|           Sebastian|   FL|Triangle|20-40 minutes|6 low flying craf...|3/4/2016|
|2/29/16 23:00|Salunga/Landisvil...|   PA|Triangle| 5-15 minutes|Pennsylvania tria...|3/4/2016|
|2/29/16 22:00|                York|   PA|Triangle|   30 minutes|Myself and 2 frie...|3/4/2016|
|2/29/16 21:35|              Joliet|   IL| Unknown|   10 minutes|At approximately ...|3/4/2016|
+-------------+--------------------+-----+--------+-------------+--------------------+--------+
only showing top 5 rows
```

Spark Transformations



Too Many Transformations to Do Here!

- Examples are In the Pre-Recorded Video!
- For Reference:
  - /work/examples/PySparkCookbook.ipynb
  - /work/content/2-Object-Spark.ipynb
  - https://sparkbyexamples.com/pyspark-tutorial/

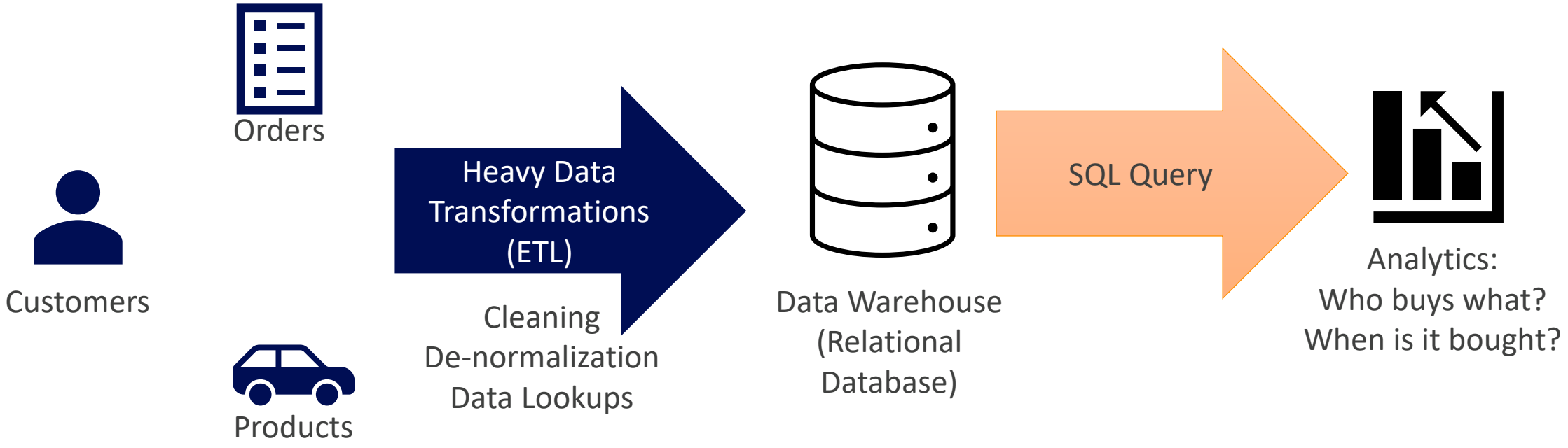


Evolution of SQL on Big Data

From Data Warehouses, To Data Lakes, to Serverless Data Warehouses

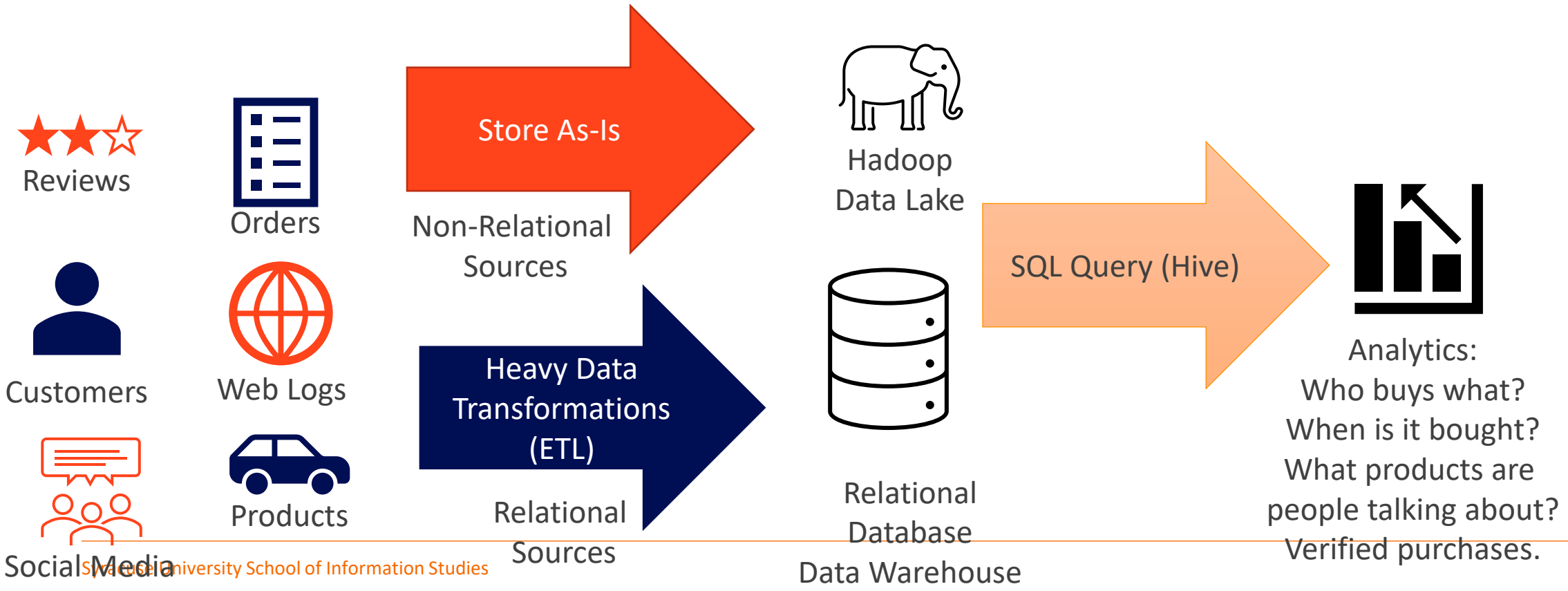
RDBMS Data Warehouses

- Organizations use data warehouses for business intelligence and data analytics.
- Data in the DW is heavily cleaned and transformed



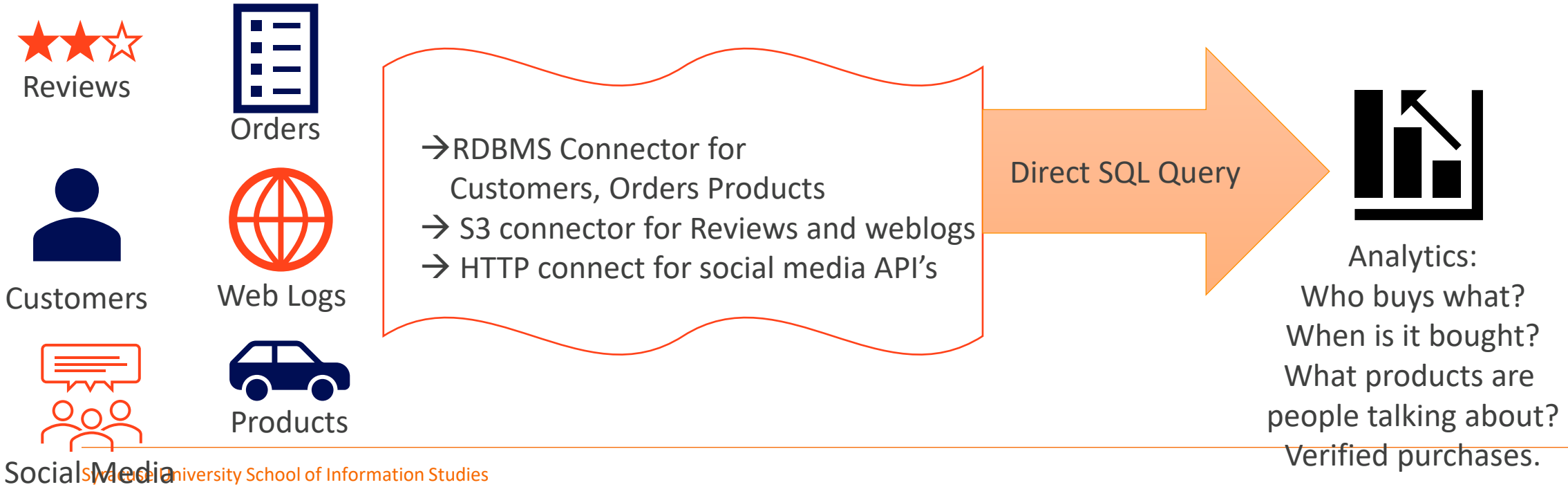
Hadoop Data Warehouse

- Modern data is complex and comes from a variety of sources (polyglot).
- Relational Db DW cannot keep up with the data volume, so we use Hadoop.



The Serverless Data Warehouse

- Large up-front investment make Hadoop costly and impractical for all but the largest of use cases.
- Agility is needed to query data from anywhere especially as cloud adoption increases.



Direct SQL Query

- Query any source of data with SQL
- Any Sources? Just about any!
  - Hadoop, Object storage, Local Files, Web API's
  - Relational Databases, NoSQL Databases, Streaming Databases
- Cloud Providers:
  - AWS Athena, Google BigQuery
- Open Source:
  - Apache Drill, Presto, Trino, Apache Spark SQL

Spark SQL

DataFrames to SQL Tables!

- Register the DataFrame as a SQL View
- Query it With SQL!!!
- ANSI Compliance https://spark.apache.org/docs/3.1.2/sql-ref-ansi-compliance.html
- SQL Syntax https://spark.apache.org/docs/3.1.2/sql-ref-syntax.html

Spark SQL

- Use `.createOrReplaceTempView()` to register a dataframe as a temp SQL view, valid for the **current session**.
- Then use `spark.sql()` to query it!
- The result is another DataFrame
- Example:

```
df.createOrReplaceTempView("sometable")
spark.sql("select * from sometable").show()
```

Example of Spark SQL

```
c = spark.read.csv("file:///home/jovyan/datasets/customers/customers.csv",
                   inferSchema=True, header=True)
c.createOrReplaceTempView("customers") # now its an SQL table in Spark!
query = '''
SELECT  Email, Gender, State, `Months Customer`
FROM customers
    WHERE State = 'NY'
'''
df = spark.sql(query)
df.toPandas()
```

	Email	Gender	State	Months Customer
0	afresco@dayrep.com	M	NY	1
1	cling@superrito.com	F	NY	6
2	etasomthin@superrito.com	M	NY	28
3	jpoole@dayrep.com	F	NY	12
4	ojouglad@einrot.com	M	NY	36
5	rovlight@dayrep.com	M	NY	42
6	sladd@superrito.com	M	NY	10
7	titupp@superrito.com	F	NY	42
8	tpani@superrito.com	M	NY	1

Document Database

What is a Document Database?

What Exactly is a Document Database?

- Designed to store semi-structured data, in JSON or XML format. This is the “Document”
- Full CRUD support.
- Documents Stored in Collections; Fetched by Key
  - Keys can be surrogate or assigned.
  - Keys are unique to the collection
- Popular Document Databases”
  - AWS Document Db
  - Google Firebase
  - MongoDb
  - CouchDb

Typical Document Db Data Model

Database

Boundary for one or more collections

Collection

A subject area for documents to be stored

Document

An individual subject in a collection

The Document (JSON Example)

- Tree-like structure
- Multiple Keys / Values Pairs
- A Document has composite data
- Self-Describing (Schema is with the data)
- No rigid schema. Anything can be saved in the collection.
  - Ask for something not there? → null

```
{
 "_id" :79687434756228,
 "name" : {
   "first" : "Michael",
   "last" : "Fudge"
 },
 "emails" : [
   { "email" : "mafudge@syr.edu",
     "type" : "work",
     "default" : True
   },
   { "email" : "mafudge@gmail.com",
     "type" : "personal"
   }
 ]
}
```

Document Database Use Cases

Good For ☺

- Collections of complex data attributes associated with a single entity.
- Items retrieved as a whole
- Examples:
  - Content Management
  - Product Catalogs
  - Immutable Data (Orders History)

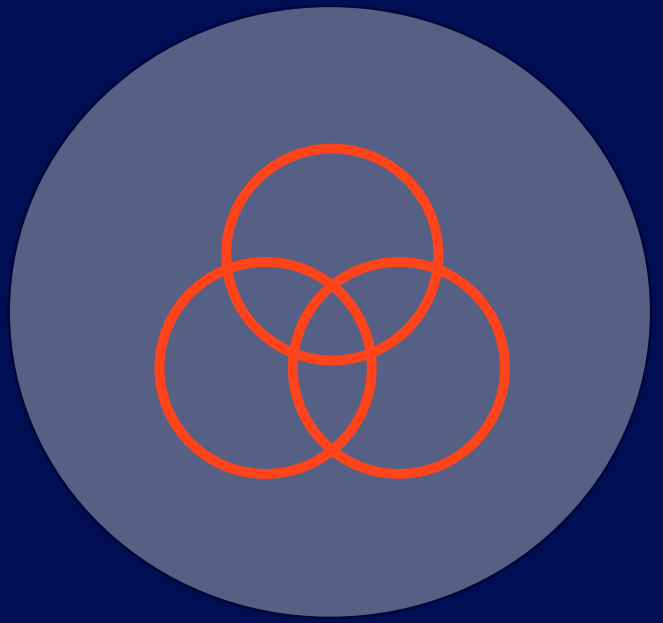
No So Good For ☹

- Complex Data Relationships
- Retrieving “Part” of the Document
- Data with a high rate of change

Document Databases are Programmer Friendly

- Read and Store Complex Data
- No Schema is Required, No SQL is Required
- Simple CRUD, Limited support for partial updates, usually you must retrieve the entire document.

```
1  from pymongo import MongoClient
2  client = MongoClient()
3  products = client.ecommercedb.products_collection
4  product = { "id" :3852972,
5      "name" : "Skis",
6      "categories" : ["sporting goods", "winter", "outdoors"],
7      "price" : 769.99
8  }
9  products.insert_one(product)
10 skis = products.get_one({"id" : 3852972 })
```



Relational vs Document

Comparison of the Two Models

Differences in design approaches

Relational vs. Document

Relational

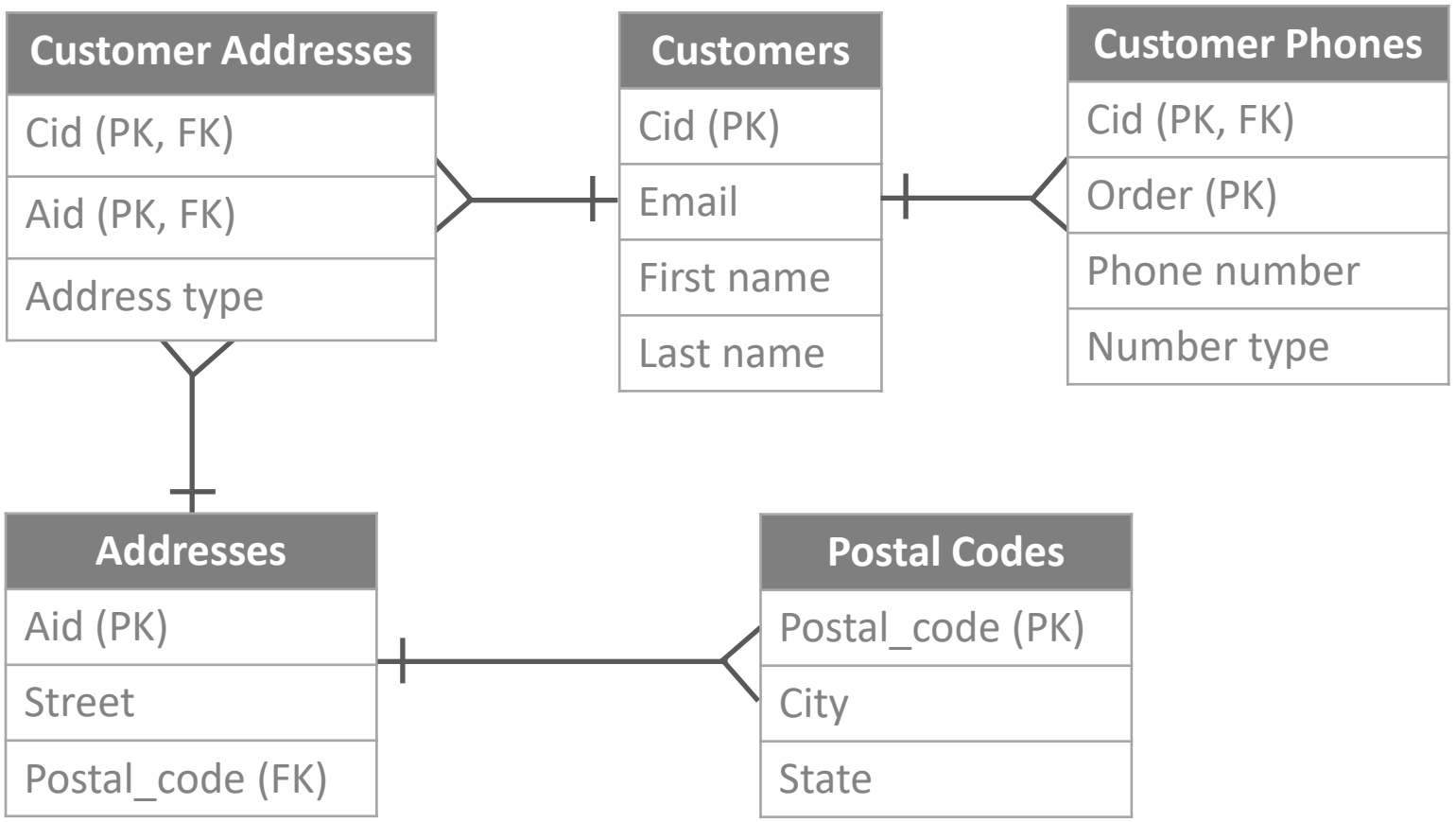
1. Designed for use by database administrators
2. Complex data model, normalized
3. Flexible, performant query operations
4. Related information stored in individual tables
5. Data redundancy is exception to the rule
6. Domain-Specific Query Language (SQL)

Document

1. Designed for use by programmers/developers
2. Simplified data model, denormalized
3. Query not as performant
4. Related information stored with document
5. Data redundancy is expected
6. Non-Domain Specific Query Language

Customer: The Relational Way

- Customer has many other entities like phones and addresses
- We can update phones / addresses independent from customer
- Data in 3rd Normal Form; Data Integrity
- Keys. Keys everywhere!



Customer: The Document Way

- Self-Describing / Schema-less
- Denormalized; all related data stored together.
- Data Integrity shifts from data logic to business logic.
- Easier to store, harder to query.
  - Do two customers have the same address?

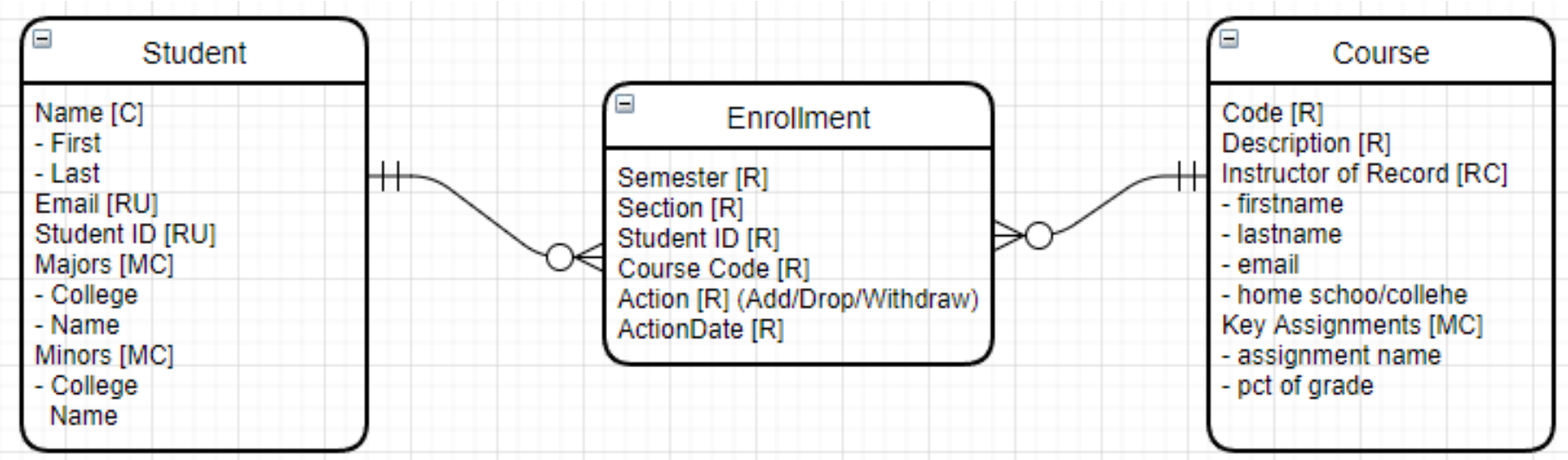
```
{       cid : 1,
        email: mafudge@syr.edu,
        first name: Michael,
        last name: Fudge,
        phones: [
                { type : home, number: 555-1234 },
                { type : cell, number: 555-9283 }
                ],
        addresses: [ { type: billing,
                street: 1313 Mockinbird Ln,
                city: Syracuse,
                state: NY,
                zip: 13244 }
        ]
}
```

Designing for Document Databases

- ER Modeling works. Entities are documents in the collection.
- Reference-Oriented Data
  - Store composite, Multi-Valued with main Entity
  - Ensure Slow Rate of Change. High rate of change is not a good use case.
  - Can support full CRUD
- Process-Oriented Data
  - Design to be Immutable or Historical
  - Should support only CR (no updates or deletes)
- Minimize the number Entities and Relationships.

Example: Student Enrollment

- Each entity is an example of a document in a collection. 3 Collections.
- Student and Course are reference-oriented. Composite data such as majors, minors get stored with student. Full CRUD support for these collections.
- Enrolment is process-oriented but designed to be document immutable. When a student drops another document is added.



Mongo Db

Architecture

MongoDB

- Document Database
  - Database → Collections → Documents
  - Documents are JSON Schemas
- Single Master Architecture
  - Scales well horizontally, but has a single point of failure
  - Supports Consistent reads
  - Supports sharding with data replication
- Data Model
  - Document data model—schema-less
  - Every document stored has a key
  - Uses JavaScript as a query language and JSON as a data format
  - There are only integrity constraints on the key; it must be unique per collection
- Custom query language MQL
  - Based on JavaScript

Mongo Db And Data Integrity.

- Document databases are incredibly flexible and do not complain about missing schema.
- No Database? I'll make that for you.
- No Collection? I'll make that for you.
- Add the same thing twice… no biggie I pick the key for you!
- Add people to the books collection. I'll gladly do that for you!
- **You are responsible for data integrity.**

Two Key Concepts of Distributed Data

Sharding

- Splits the data into partitions so that each node hosts part of it
- Distributes the I/O over several hosts; for performance
- Data distributed to nodes based on shard key strategy

https://docs.mongodb.com/manual/sharding/

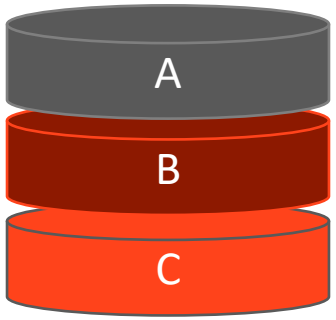
Replica

- Synced copies of shards on different nodes
- Each replica contains an exact mirror of the shard
- For Fault-Tolerance

https://docs.mongodb.com/manual/replication/

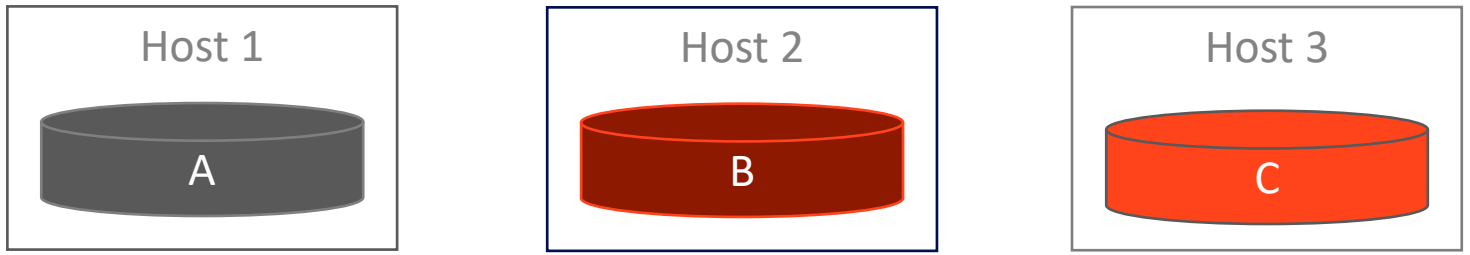
MongoDB Cluster Visualized

- Consider a mongo database, with a collection of documents mapped to sets A, B and C.



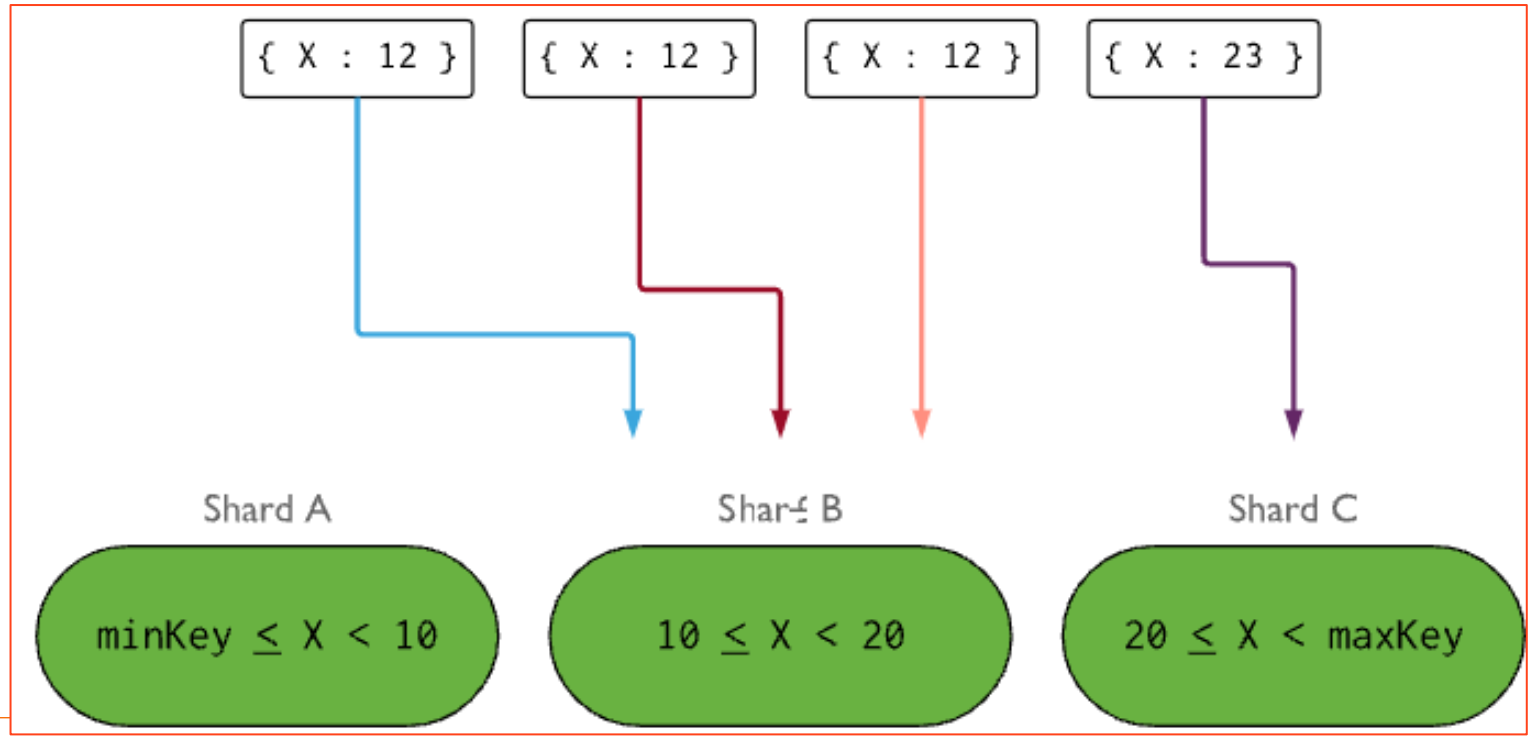
MongoDB Cluster Sharding

- It is ***sharded*** into three nodes. Each node (hosts 1–3) has different data—no single node has the same data.
- Each Host holds a single shard.



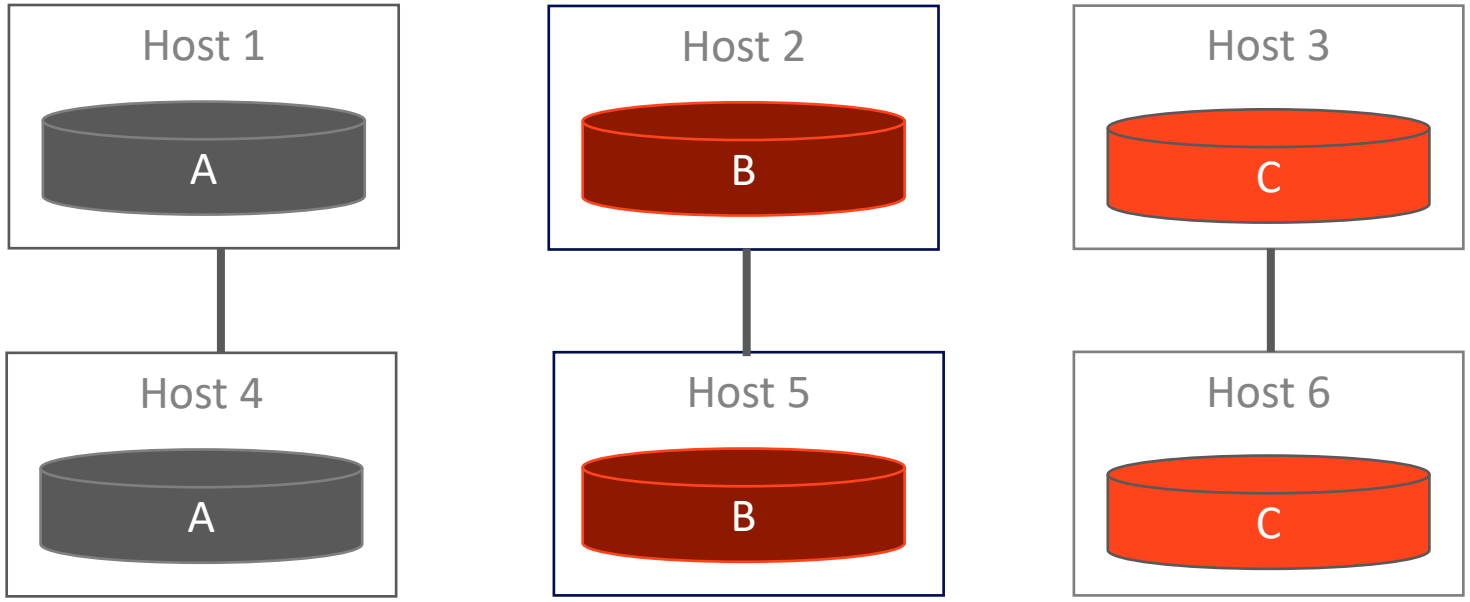
Shard Key

- The Shard Key determines how the data are distributed to the nodes.
- Two approaches:
  - Range key
  - Hash key
- We want **even distributions** for the best performance.
- Range key example →



MongoDB Cluster Replication

- Each of the three nodes has one replica. Hosts 4–6 are a mirror copy of hosts 1–3, respectively.
- For fault-tolerance we add nodes.



Mongo Db Query Language (MQL)

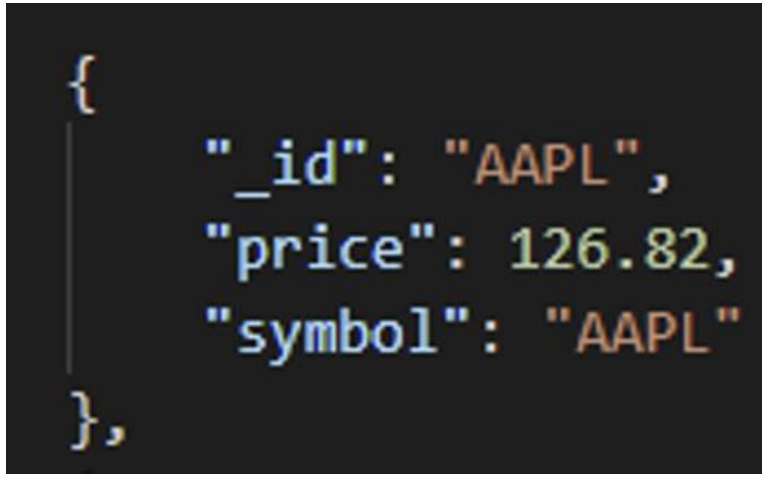
Commands in MQL
(Mongo Query Language)

MQL: Create and Read

- use *database*
- show collections
- db.*collection*.insertOne(*jsonData*)
- db.*collection*.insertMany(*arrayJsonData*)
- db.*collection*.find(*query*,*projection*)
- db.collection.replace(*query*,jsonData)

MQL Query Example

- *db.collection.find(query, projection)*
- db.stocks.find(
    {“symbol” : “AAPL”},
    { “symbol”:1, “price”:2}
)



https://docs.mongodb.com/manual/tutorial/query-documents/

MQL ←→ SQL

- https://docs.mongodb.com/manual/reference/sql-comparison/
- Good reference that provides the SQL equivalent of just about every MQL query.

Wide Column Database

What is a Wide Column Database?

What Exactly is a Wide Column Database?

- Also known as “Big Table” database or extensible record store.
- Model consists of:
  - Tables with columns where any value can be null.
  - Every table has a row key, value is retrieved by column key
  - No table joins
- Cloud Offerings
  - Google Bigtable
  - Azure Tables
  - AWS DynamoDb
  - Hadoop Hbase

Wide Column Visualized

- Columns are stored under a row-key, efficient storage for wide tables with many columns with nulls.

Row Key	Name	Year	GPA
1	Anne	Freshman	3.2
2	Bette	Freshman	
3	Chris		3.5
4	Dave		
5	Erin	Junior	
6	Fred		4.0

Row Key	Name
1	Anne
2	Bette
3	Chris
4	Dave
5	Erin
6	Fred

Row Key	Year
1	Freshman
2	Freshman
5	Junior

Row Key	GPA
1	3.2
3	3.5
6	4.0

Use cases

- Storing vast amounts of immutable, process-oriented data
  - Orders, Stadium Attendance, Tracking vehicle movement in real-time.
- Time-Series Data
  - IoT Telemetry (data created by IoT devices)
  - Health tracker data, Weather Data, User activity
- Applications with Specific Query Flow
  - Search a Region → Select Hotel in Region → Select available Room in Hotel
  - Find a student by name or id → Select course → view all submitted assignments

Wide Column Good And Bad

Good For

- Time Series Data
- Guaranteed Writes
- High Write Volume
- Process-Oriented Data
- Queries across a single subject
  - “Orders”
  - “Uber Trips”

Not Good For

- General Purpose Database design (Like Relational or Document)
- Ad Hoc Queries (querying it any way you want)
- Data Warehouses (reads are slow)
- Data that must be normalized
- Reference-Oriented Data

Typical Applications

Any time series data

IoT applications

User activity tracking

Performance monitoring

Social media analytics

E-commerce Transactions

Messaging

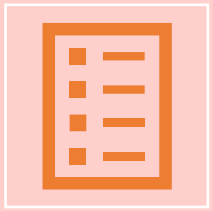
Cassandra

Open Source Wide-Column Database

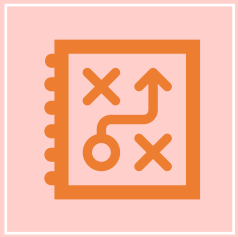
Cassandra

- Developed Originally by Facebook
  - Developed by Facebook, open sourced in 2008
  - Inspired by Google Bigtable and Amazon DynamoDB papers
- Eventual Consistency (BASE)
  - Can always write, but not guaranteed to read, the same thing
  - Scales well horizontally
  - AP System – no single point of failure
- Data Model
  - Distributed wide-column store database
  - Key maps to one or more columns
  - Has an SQL-like query language CQL
  - Tables must exist prior to data being stored
  - There are no data integrity constraints, no table joins.

Databases the Cassandra Way



Query design influences table design. Build your tables based on how you will query the data back out.

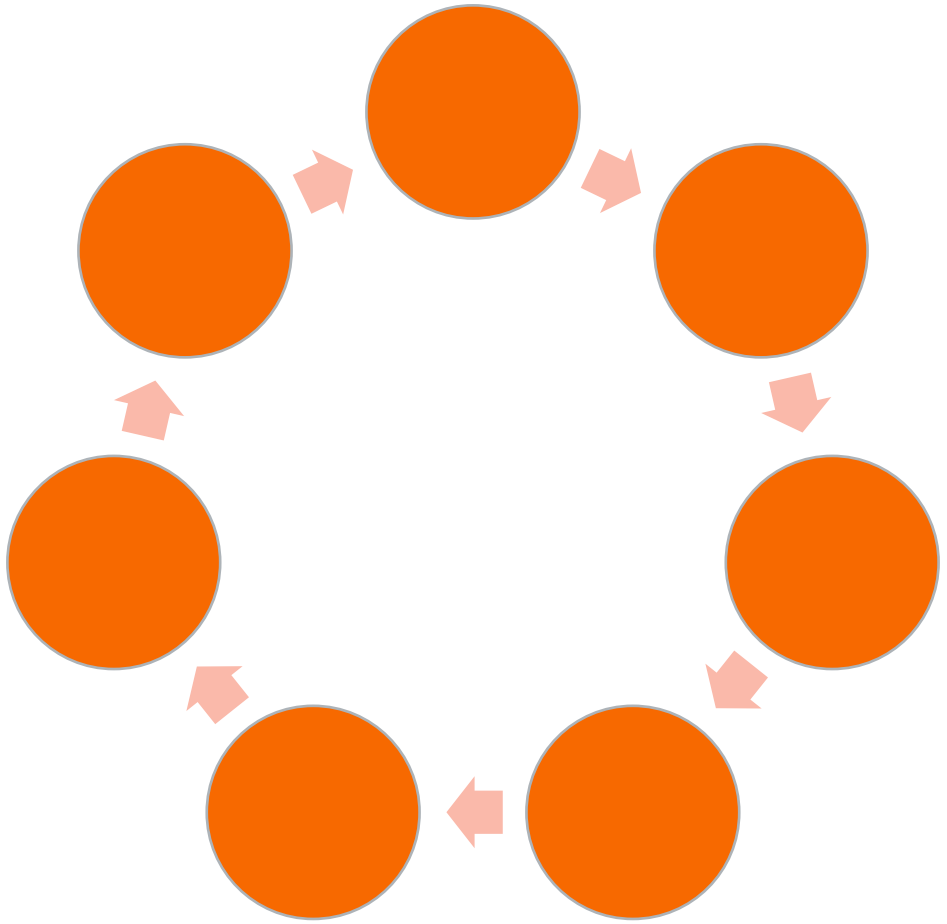


Redundant, de-normalized data, including different versions of the same conceptual entity

One large flat table per query.

Cassandra Ring Architecture

- All nodes play the same role
- All nodes communicate with each other
- There is no “single master” like MongoDB, Redis, and RDBMS solutions
- Data replicated to other nodes.
- No single point of failure
- To scale—add more nodes!
- You can configure replication and number of replicas



Cassandra Data Model

Cassandra Data Model at a Glance

- **Cluster:** a collection of Cassandra nodes
- **Keyspace:** a container for data tables and indexes; this is like a "database" in a RDBMS; defines the replication strategy
- **Table:** similar to a RDBMS table, but all columns are optional "wide column store" implementation
- **Row key:** used to uniquely identify a row in the table and to distribute the rows across the cluster
  - Consists of a Partition key and a Cluster key
- **Index:** similar function as an RDBMS index

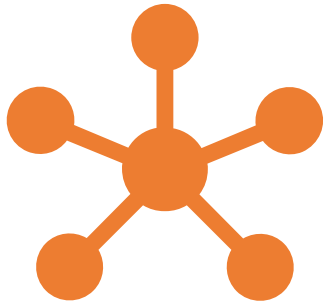
Tables

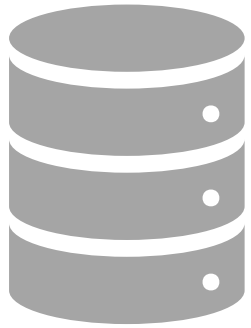
- Tables similar to RDBMS tables, except since there are no “joins” in Cassandra; the tables should be highly de-normalized and wide (many columns)
- Cassandra tables are best suited for the capture of events such as orders, sensor readings, and so on; these data are usually time series
- Tables are in wide-column store format; this means that all columns are NULLABLE, so there are no integrity constraints
- Use the CREATE TABLE statement to make tables
- https://cassandra.apache.org/doc/3.11/cassandra/cql/ddl.html#create-table-statement

Row Keys Are Important

- Must uniquely identify a row, but will also limit how we can retrieve rows.
- The first attribute in the row key list is the **partition key**. This value is hashed, and the value determines to which node in the ring the data are written. (similar to a MongoDb shard)
- Any additional parts of the row key are the **clustering key**, which determines the order by which the data are written to that node (similar to an RDBMS clustered index).
- To retrieve rows, you must include values from row key.

The Row Key: Partition + Cluster Key



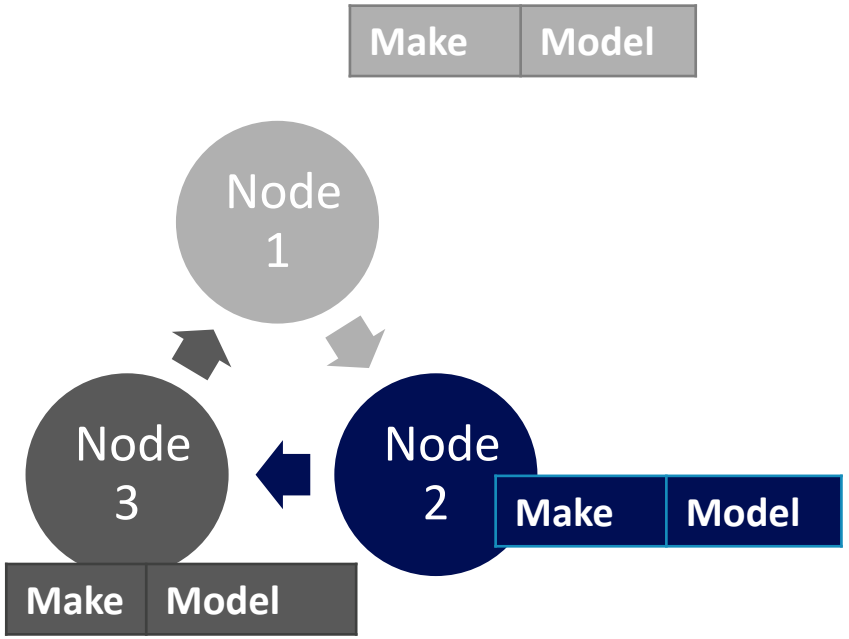


Partition key—distributes data across nodes

Clustering key—**sorts** data within a node

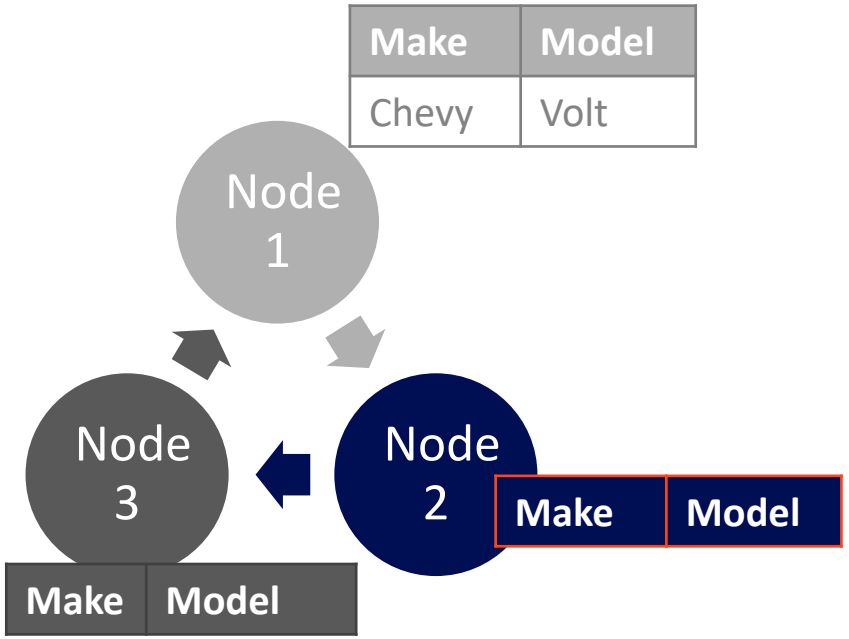
The Cassandra Cluster Data Visualized

Cassandra Cluster Visualized



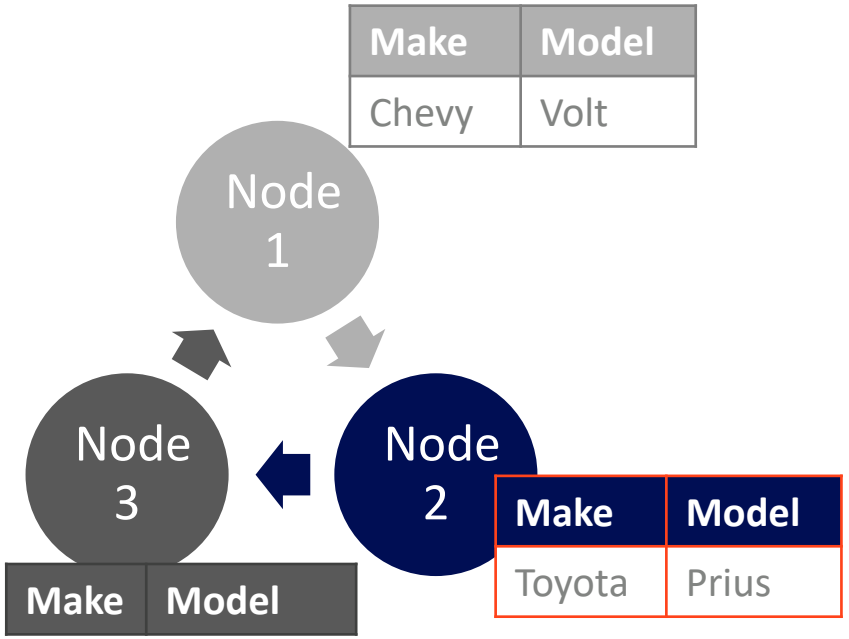
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:

Cassandra Cluster Visualized



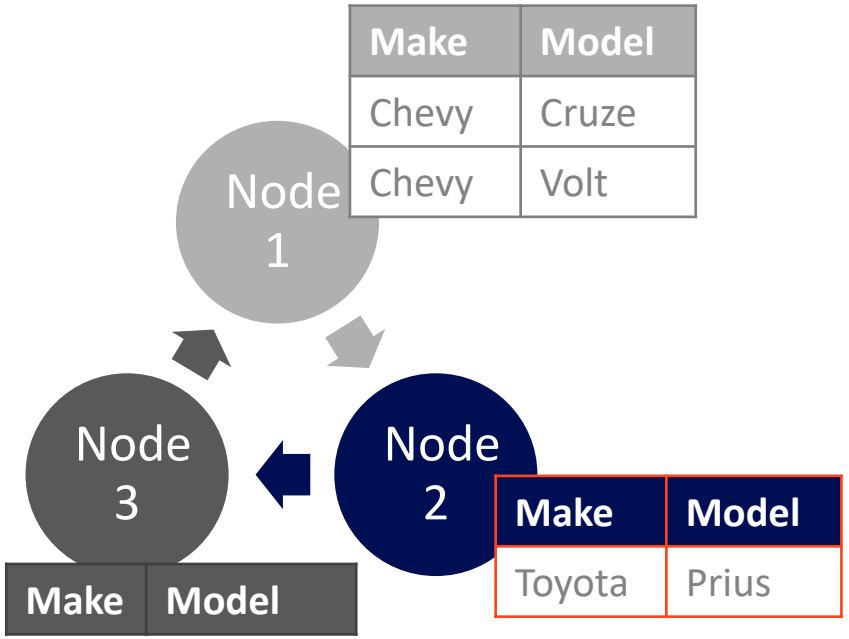
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt

Cassandra Cluster Visualized



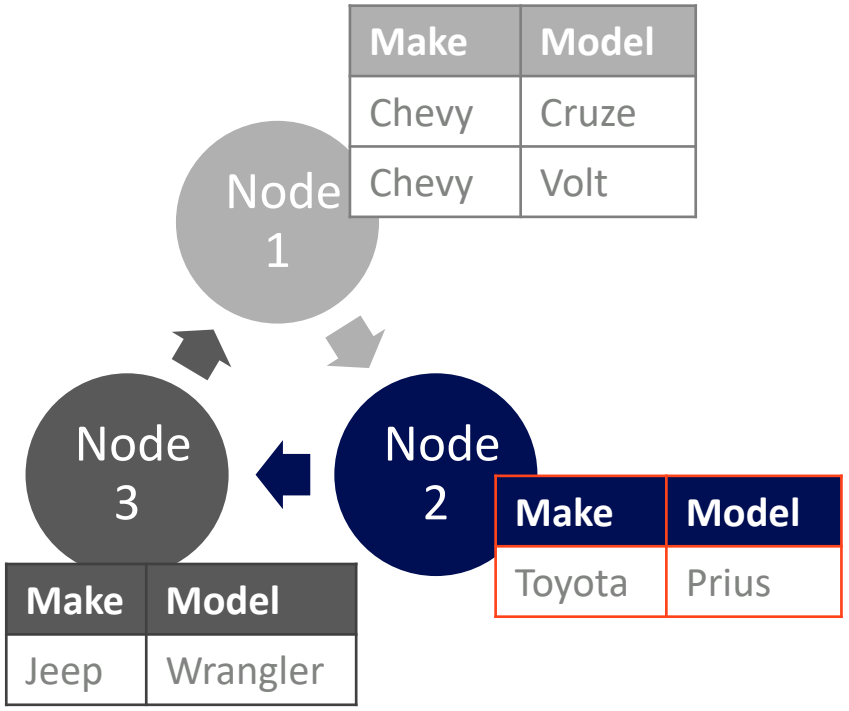
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt
  2. Toyota Prius

Cassandra Cluster Visualized



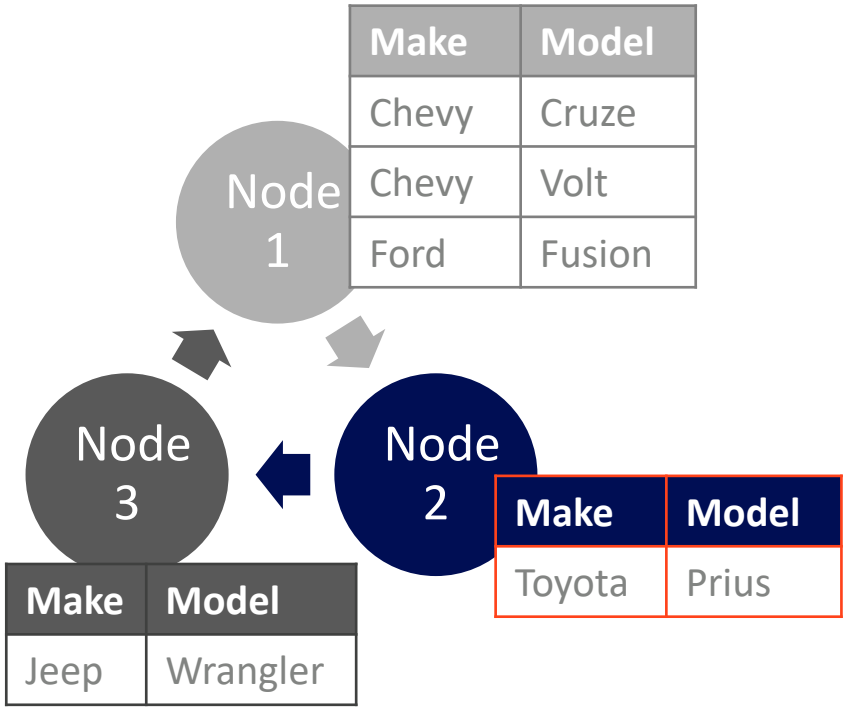
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt
  2. Toyota Prius
  3. Chevy Cruze

Cassandra Cluster Visualized



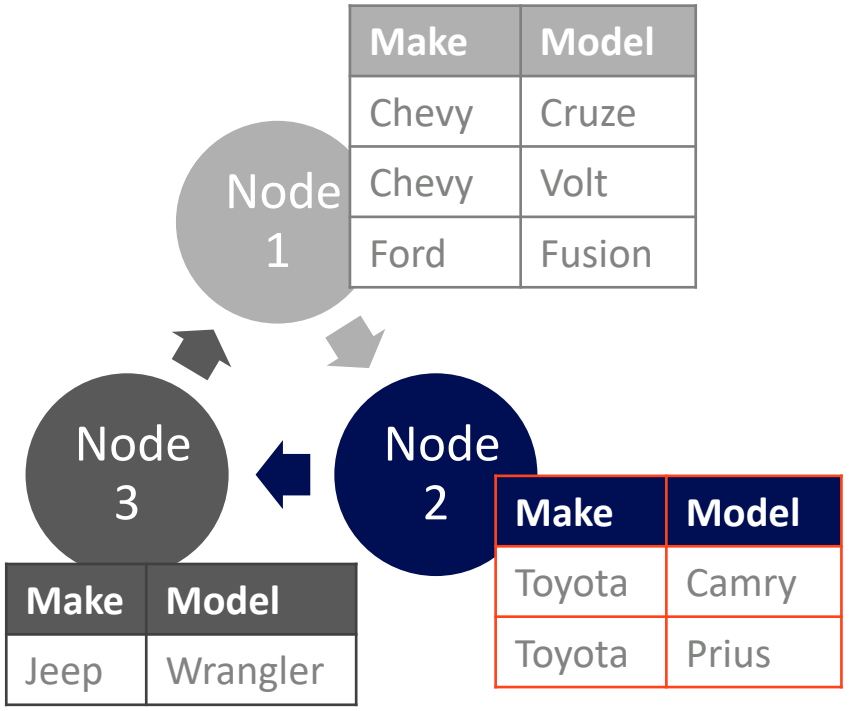
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt
  2. Toyota Prius
  3. Chevy Cruze
  4. Jeep Wrangler

Cassandra Cluster Visualized



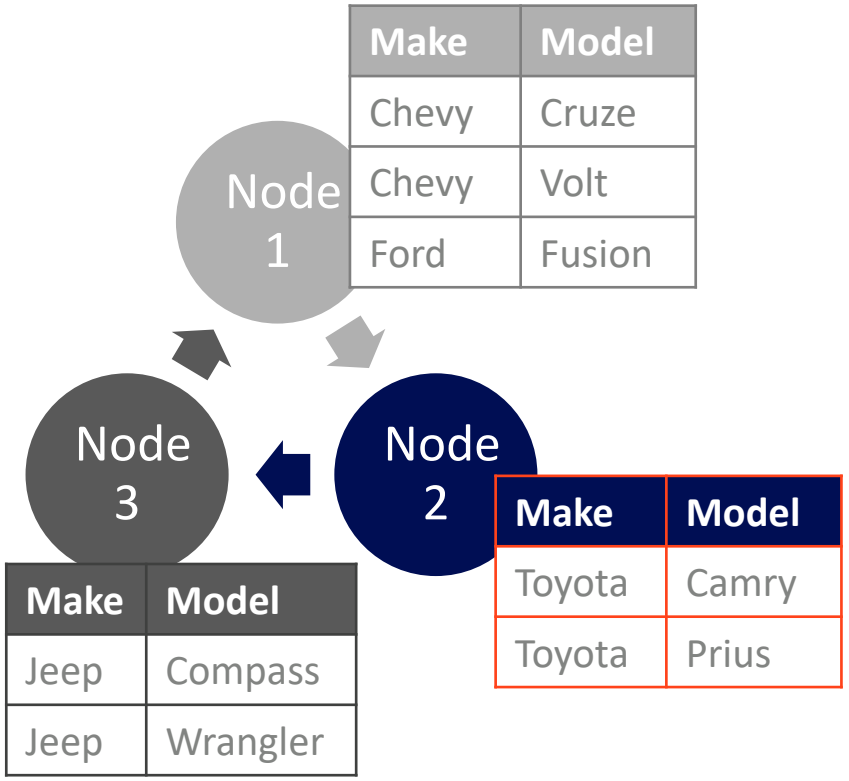
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt
  2. Toyota Prius
  3. Chevy Cruze
  4. Jeep Wrangler
  5. Ford Fusion

Cassandra Cluster Visualized



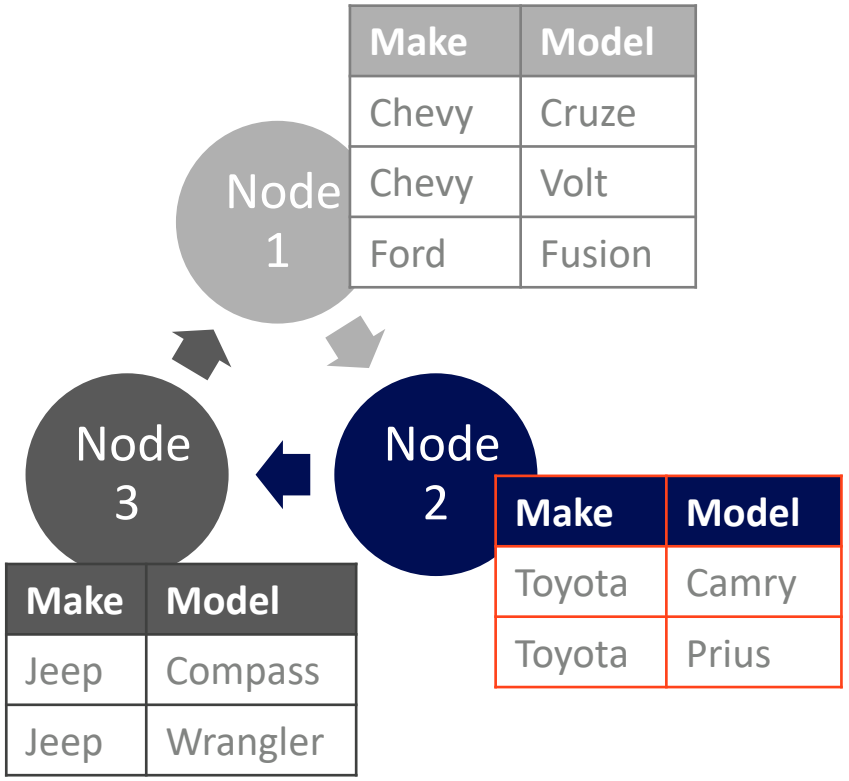
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt
  2. Toyota Prius
  3. Chevy Cruze
  4. Jeep Wrangler
  5. Ford Fusion
  6. Toyota Camry

Cassandra Cluster Visualized



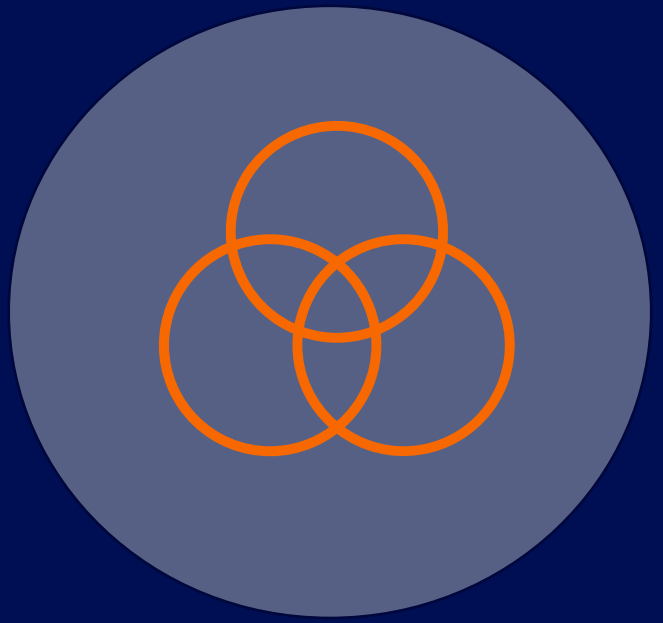
- Let's assume that we have a cars table with:
  - Partition key = make
  - Cluster key = model
- Writing the following data:
  1. Chevy Volt
  2. Toyota Prius
  3. Chevy Cruze
  4. Jeep Wrangler
  5. Ford Fusion
  6. Toyota Camry
  7. Jeep Compass

Cassandra Query: Partition Key Matters



- I can query by make:
  1. SELECT * FROM cars WHERE Make = ‘Toyota’
- I can NOT query by Model
  2. SELECT * FROM cars WHERE Model = ‘Compass’

This is because 1. can determine which node participates without asking other nodes. (‘Toyota’ hashes to node 2)
Every node must be “asked” if they have ‘Compass’



Wide Column Database Design

Data Modeling for Cassandra

Data Modeling

Relational

- Complex data model, normalized
- Flexible, performant Ad hoc filter operations on any column.
- One table per entity set, FK's represent data relationships
- Table joins
- Data redundancy is undesirable

Wide Column

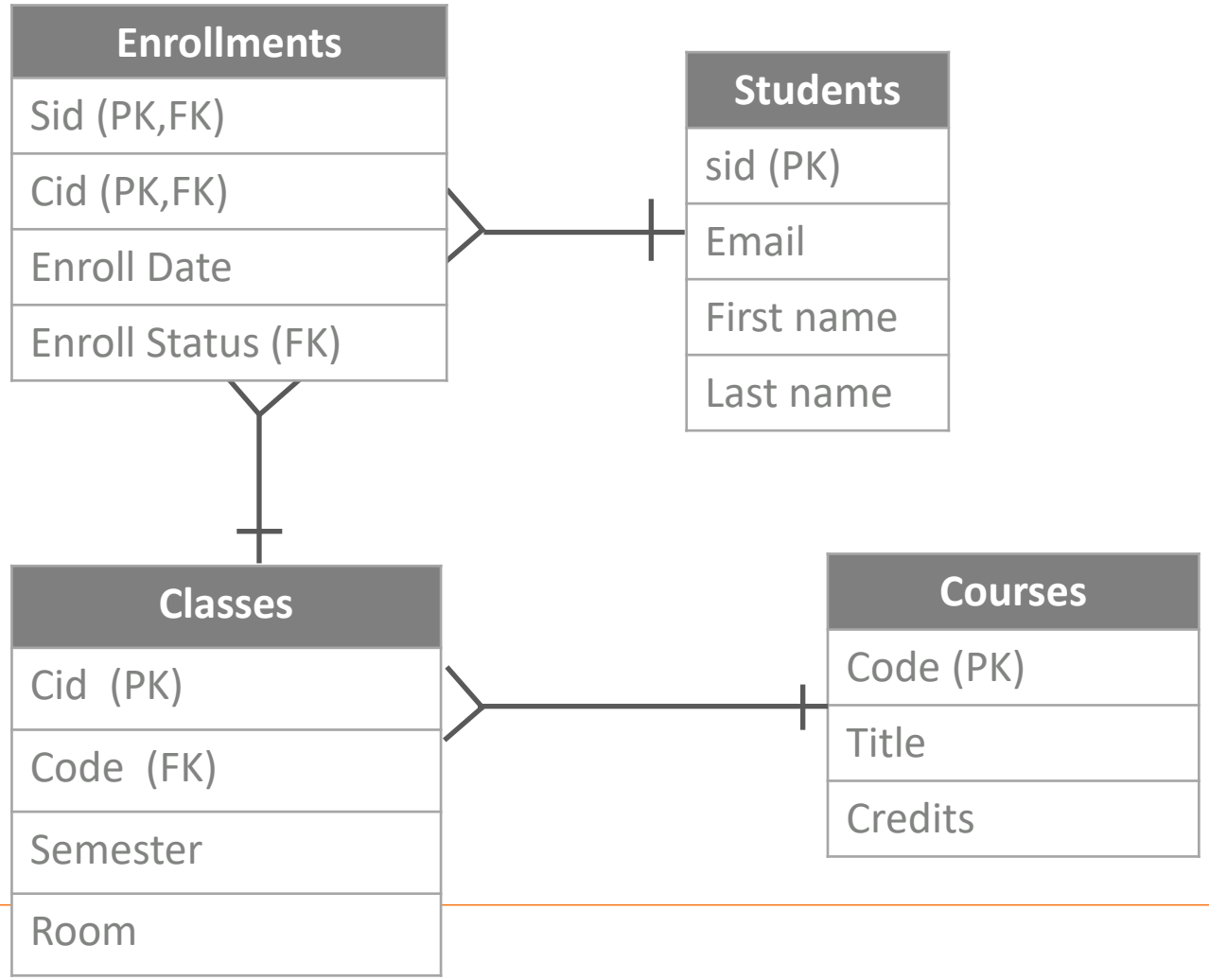
- Flat model, denormalized
- Can only query by row keys no filters on non key data
- One table per query type with related data in table.
- No table joins
- Data redundancy is expected

Designing For Wide-Column

- Design by Query: How will the data be retrieved?
- Process-Oriented Data is the Appropriate use case.
- Keys matter!
  - Partition key should be used limit the number of nodes that need to participate in the query. One is ideal.
  - Cluster key should be sequential (surrogate, timestamp, etc..) to ensure data is appended to the end, and not inserted into the middle.

Enrollment: The Relational Way

- Courses are offered as classes.
- Students enroll in classes.
- Enrollment status
  - Add
  - Drop
  - Withdraw
- Enrollments is a business process, the rest are reference-oriented Data



Enrollment: The Wide Column Way

- No Surrogate keys.
- Design by Query:
  - Q1: Find course sections (search by course number, retrieve sections)
  - Q2: Find Section enrollments (with class id, retrieve enrollments)
- No joins so we must embed columns
- PK = Partition Key, CK = Cluster Key
- Cannot search by semester. Need a different table or materialized view for that!

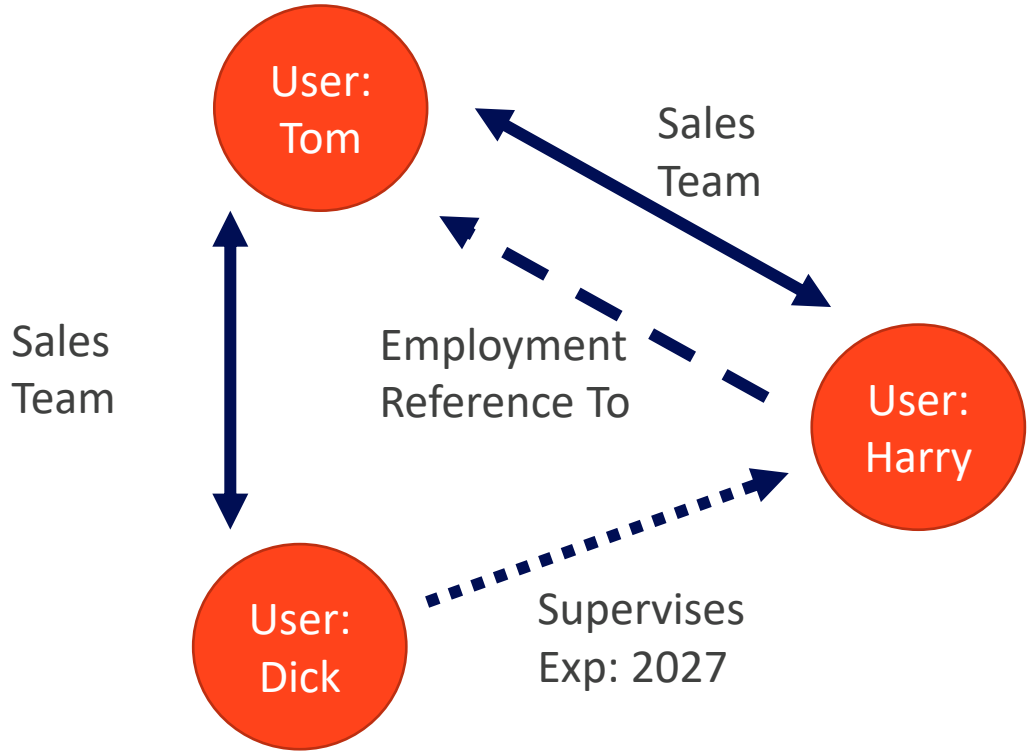
ClassesByCourse
ClassNo (CK)
Code (PK)
Semester
Room
Course {Title, Credits}
Instructor {Name, Phone, Office, Email}

EnrollmentByClass
ClassNo (PK)
Student Id (CK)
Student {name, email}
Enrollment Date
Enrollment Status

Graph Data Model

Graph Databases

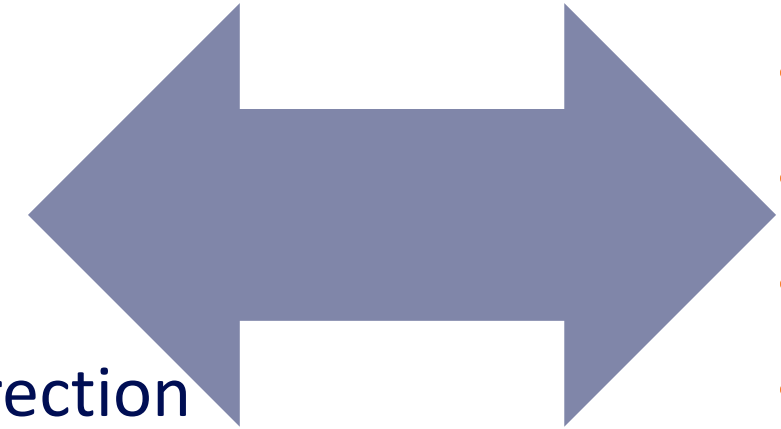
- The Graph Data Model consists of Nodes and Edges
- Nodes are like Entities. They have a type and attributes.
- Edges connect nodes. They can be one-way or two-way.
- Edges can carry attributes as well



Graph and The Relational Equivalent

Graph

- Node Label
- Node
- Node Property
- Relationship
- Relationship Direction
- Relationship Attributes

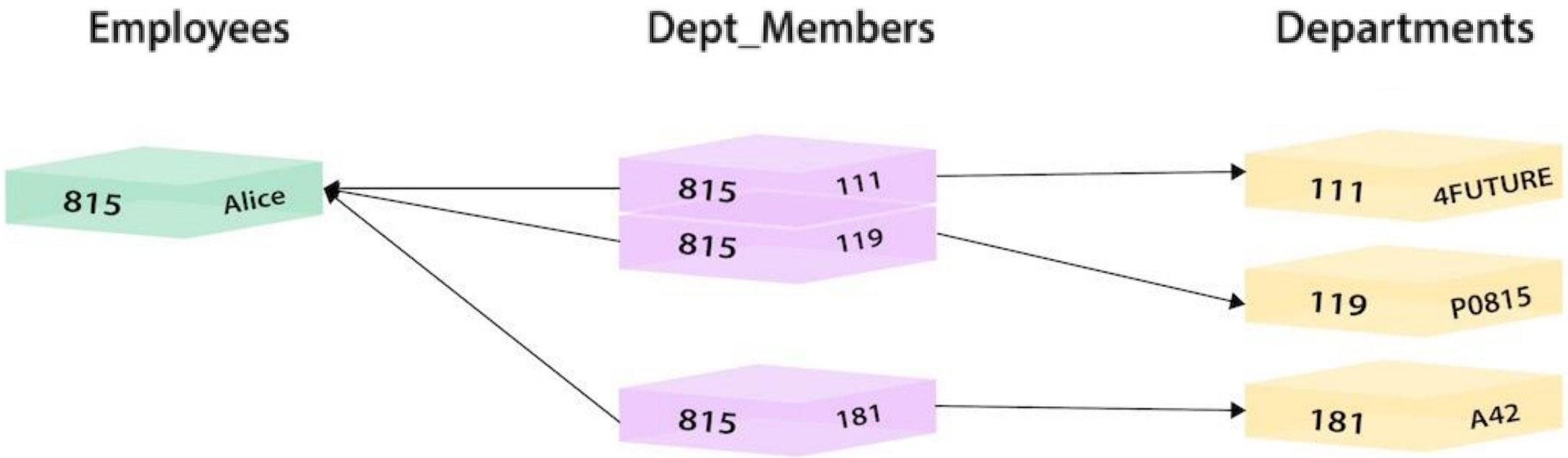


Relational

- Table
- Row in Table
- Column in Table
- Foreign Key
- (No equivalent)
- Associative Entity / Bridge table

Relational: “Employees and Departments”

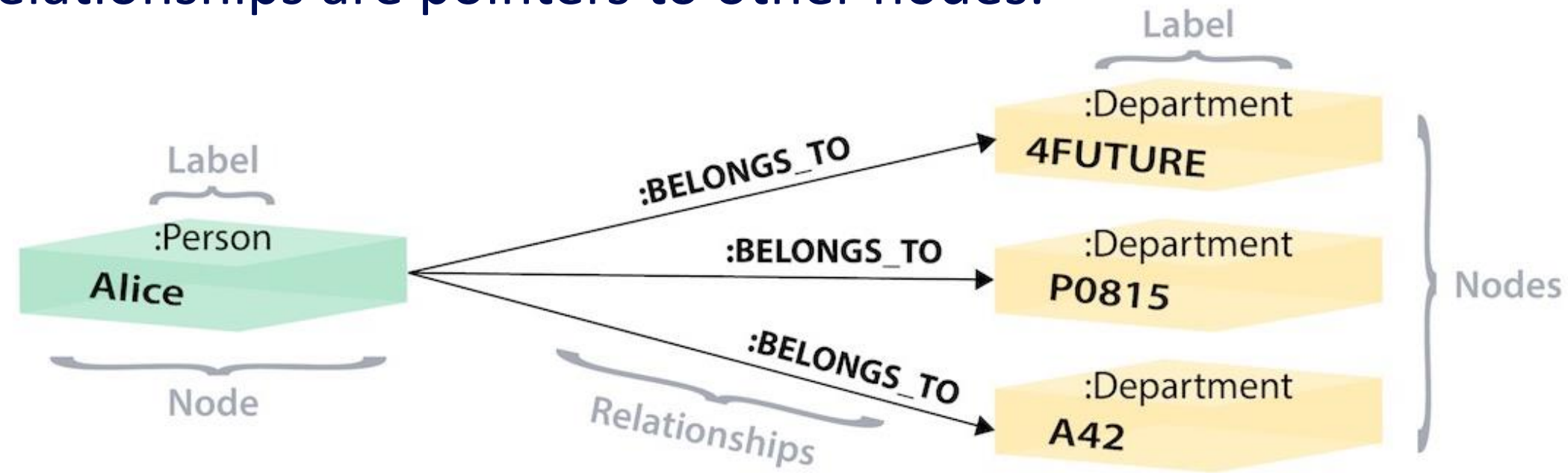
Many-to-Many, Requires an Associative Entity/Bridge Table



Graph: “Employees and Departments”

Entities are Labeled Nodes.

Relationships are pointers to other nodes.



Why Graph instead of Relational?

- Relational is not good for complex relationships
- One-way Relationships (Most Common)
  - Paying an Invoice, Enrolling in a course
- Recursive Relationships (Parent-Child)
  - Organizational Charts
  - Product Categories
  - Security Permissions
  - File Systems
- Complex Relationships (True Graphs)
  - Email Interactions / Social Media – Community detection
  - Path finding
  - Centrality – like Google's PageRank Algorithm

Neo4j

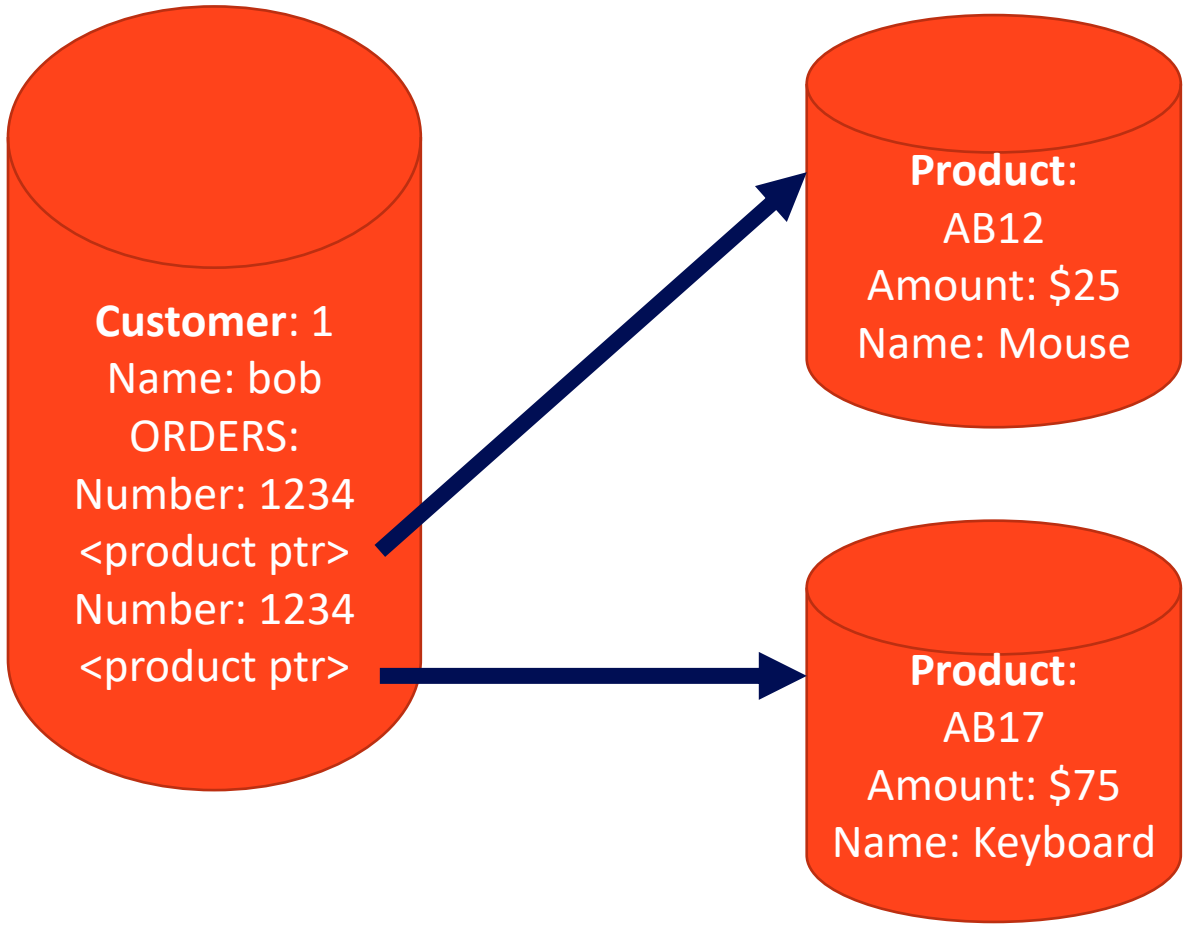
Open Source Graph Database

Neo4j

- Open-Source Graph Model Database
- ACID-Compliant (CP System: Scales Through Single Master)
- Commercial Version supports High-Availability
- Custom Query Language called ***Cypher***
- Written in Java
- Very performant:
  - Can handle billions of nodes on a single instance
  - Can traverse 1000's of relationships in sub-second time

Neo4J Physical Data Model

- Each Node Contains:
  - ID - Globally Unique
  - Labels
  - Attributes
  - Defined Relationships
  - Pointers to other nodes that satisfy the relationship
- Relationship data stored, not calculated
- Faster than relational indexing
- Can't be queried with Standard SQL – Need a different Language
- Diagram shows 3 nodes, but these are all stored together.



Neo4j Logical Data Model

- Community Edition limited to single database - **neo4j**
- **Labels** allow us to group common nodes together. `:Label`
  - These are like a table, but with no required schema!
  - They just categorize common nodes.
- **Relationships** connect nodes together `-[RELATIONSHIP]->`
  - Relationships are one way or two way
- **Attributes** – key value pairs `{key:value}`
  - Can be assigned to nodes or relationships
- Every node and relationship gets its own id, globally unique to the db.

```
(f:Faculty {name:'Mike’})
      -[r:TEACHES {semester:'fall2021'}]->
            (c:Course {code:'IST769'})
```

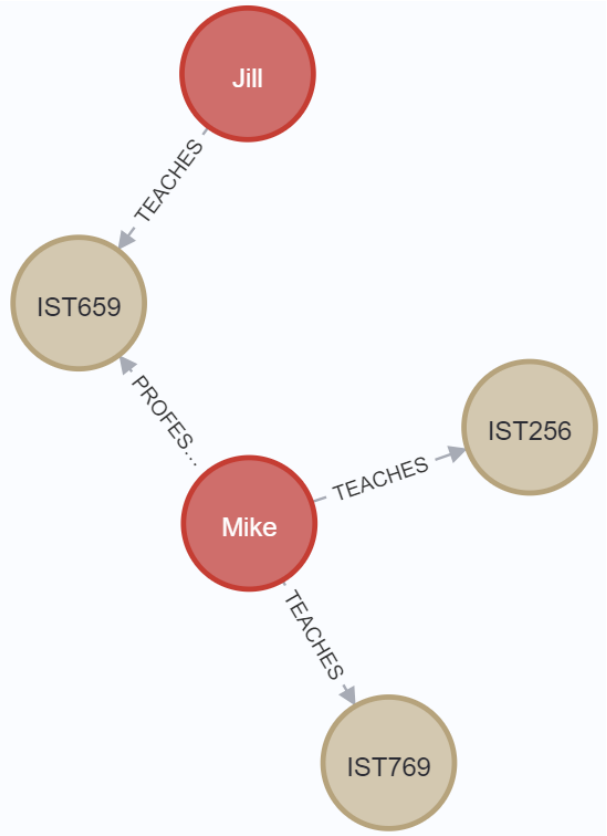
Cypher Basics

Cypher Query Language

Cypher: Build a Graph CREATE / MERGE

- CREATE Adds a node and/or relationship. It will re-add the same data.
- MERGE Will not a add the same data. (UPSERT)

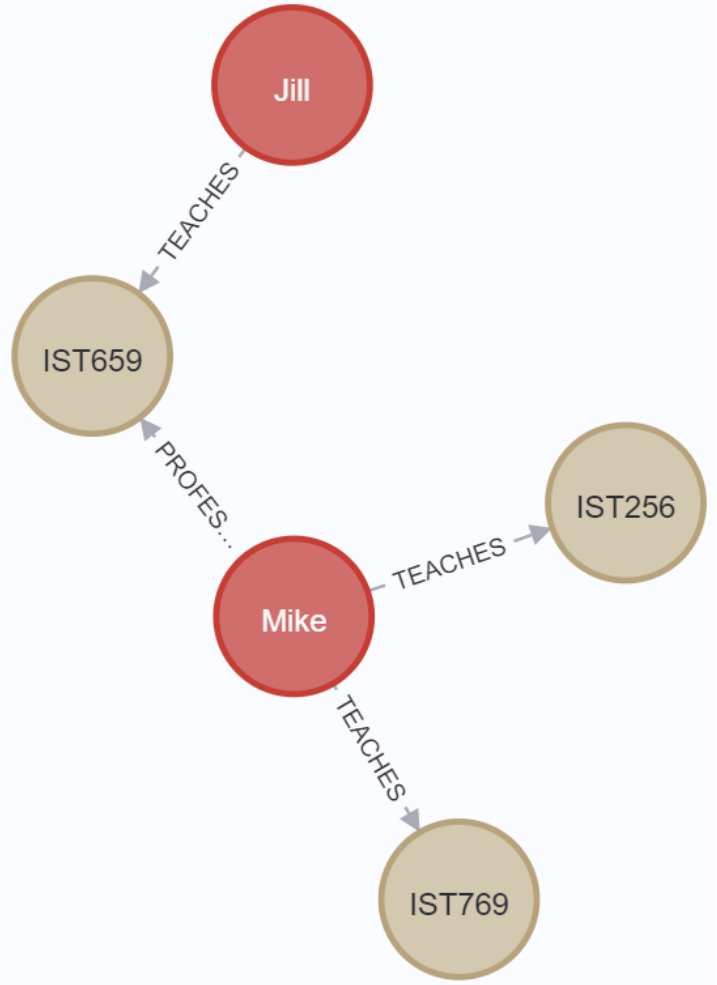
```
merge (f1:Faculty {name:'Mike'})-
      [t1:TEACHES {semester:'fall2021'}]->
      (c1:Course {code:'IST769'})
merge (f1)-[t2:TEACHES {semester:'fall2021'}]->
      (c2:Course {code:'IST256'})
merge (f1)-[p1:PROFESSOR_OF_RECORD]->
      (c3:Course {code:'IST659'})
merge (f2:Faculty {name:'Jill'})-
      [t3:TEACHES {semester:'fall2021'}]->(c3)
```



Cypher : MATCH

- Match is used to query data.
- Here is the SELECT * Equivalent

```
match (f:Faculty)-
      [t:TEACHES]->
      (c:Course)
return f,t,c
```



Cypher: MATCH with WHERE

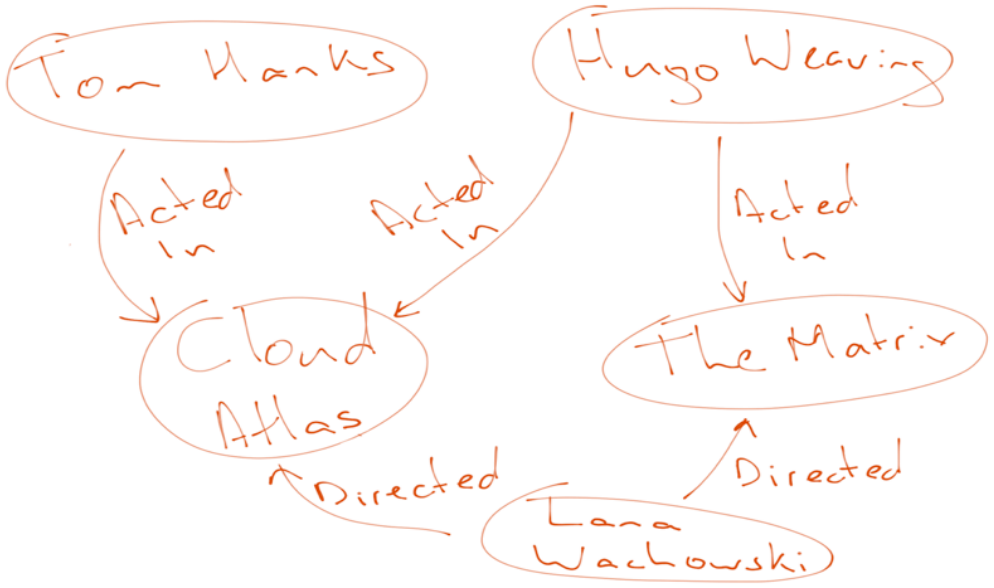
- Who is the professor of record for IST659?

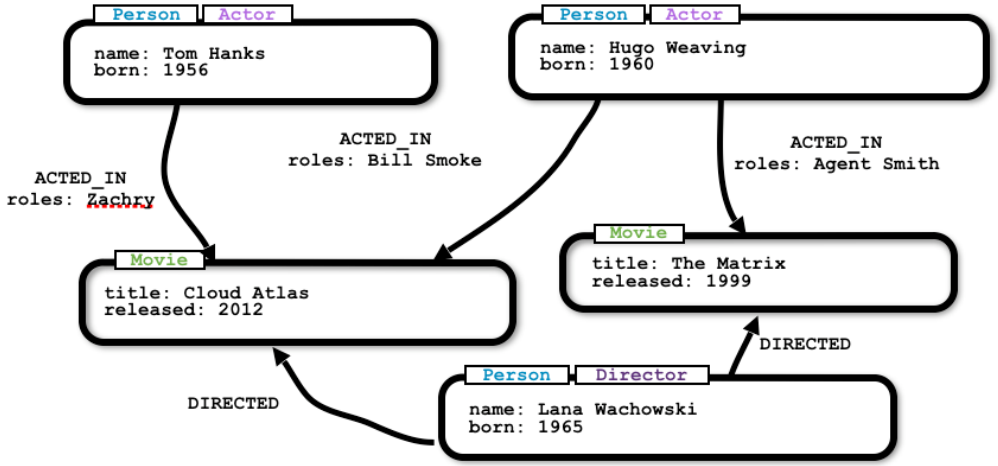
```
match (f:Faculty)-
      [p:PROFESSOR_OF_RECORD]->(c:Course)
where c.code ='IST659'
return f,p,c
```

Graph Data Modeling

https://neo4j.com/developer/data-modeling/

"Whiteboard Friendly"



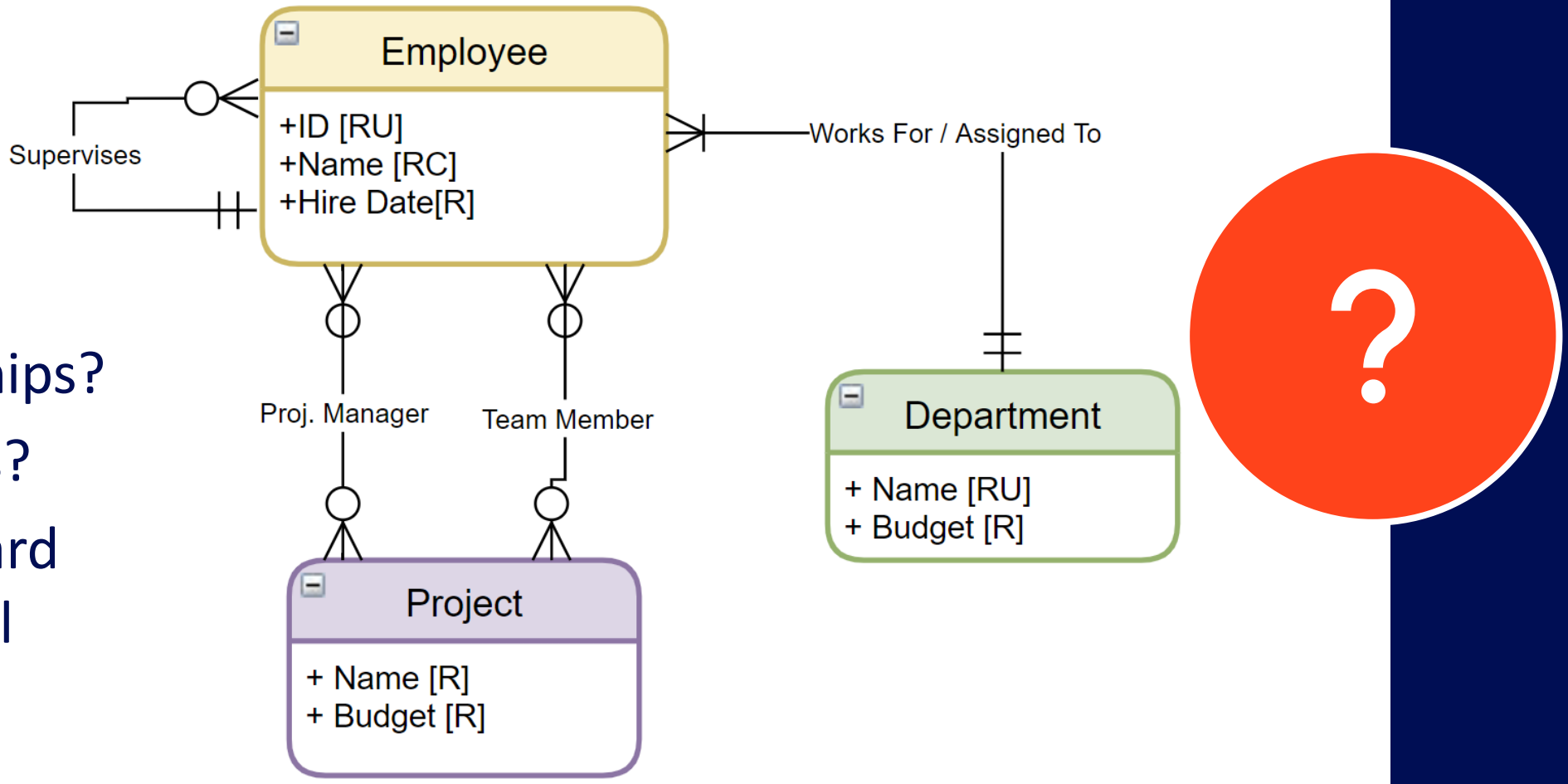


Graph Data Modeling

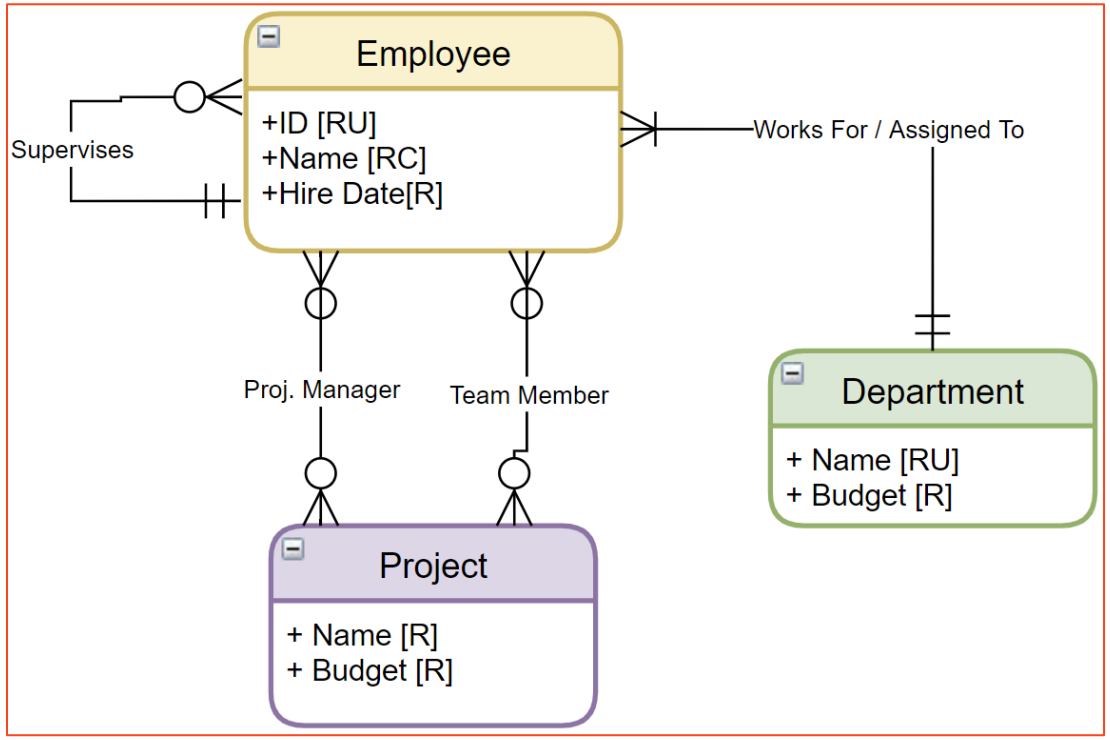
- Conceptual E-R Modeling works
  - Entities Become Nodes Labeled to match the Entity
  - Relationships become Relationships of course
  - Cardinality doesn't matter
  - Resolve Multi-valued attributes to Relationships
  - Lots of relationships are fine. This is what graph is for!
- When in doubt:
  - more nodes and relationships with fewer attributes
  - Instead of fewer nodes / relationships with more attributes
  - **Think: Exact opposite of the Document Model!!!**

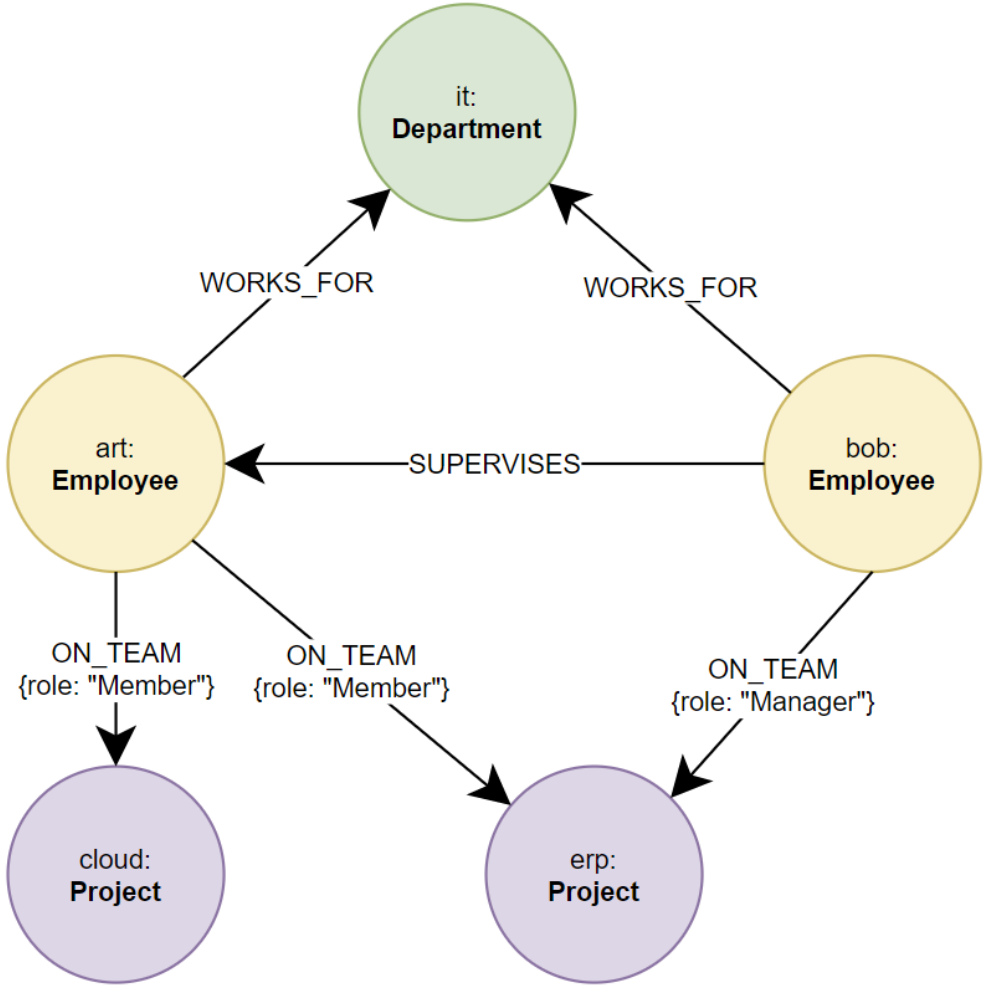
Challenge: Graph model This!

- Nodes?
- Relationships?
- Properties?
- White Board it with real data!



A Graph Solution





Graph Data Science Library

https://neo4j.com/product/graph-data-science-library/

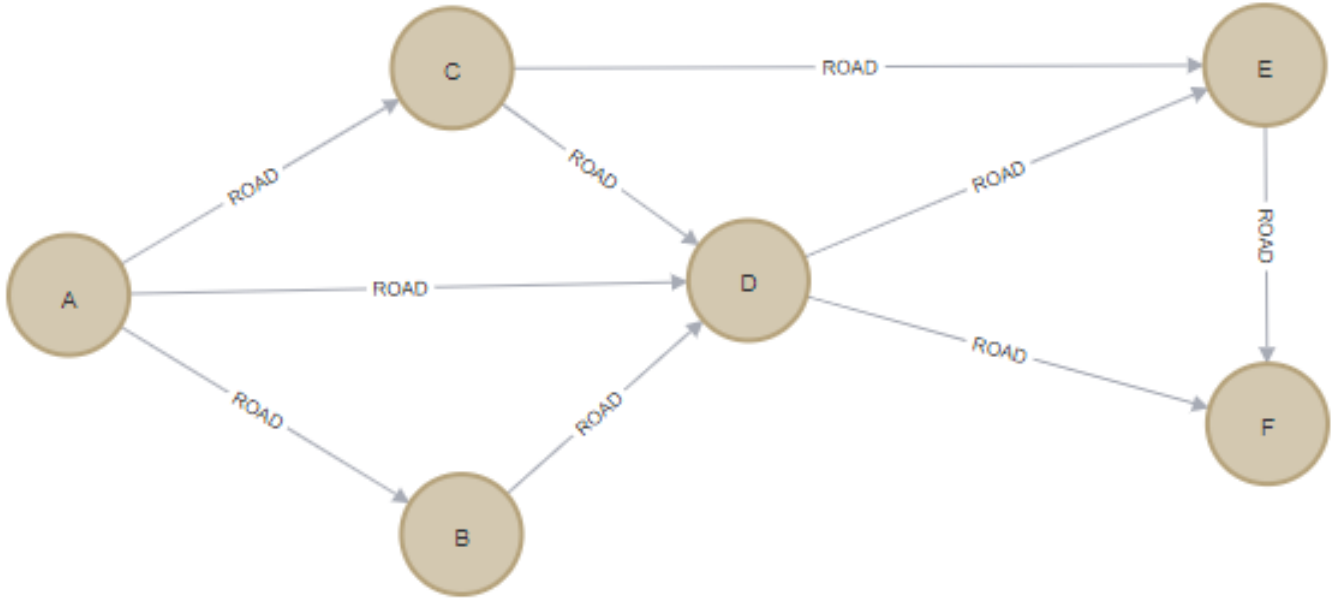
Algorithms

- **Community Detection** – find significant interactions (relationships) among nodes.
- **Centrality (Importance)** – Find influential nodes (PageRank)
- **Similarity** – Discover similar nodes based on relationships / properties
- **Heuristic Link Prediction** – predict the likelihood of a new relationship being formed.
- **Pathfinding** – find shortest, or most efficient paths between nodes.
- **Node Embedding** – Convert node relationships to vectors for ML tasks

Example: Shortest Path

- ```
  CREATE (a:Location {name: 'A'}),
  (b:Location {name: 'B'}),
  (c:Location {name: 'C'}),
  (d:Location {name: 'D'}),
  (e:Location {name: 'E'}),
  (f:Location {name: 'F'}), (a)-
  [:ROAD {cost: 50}]->(b), (a)-[:ROAD
  {cost: 50}]->(c), (a)-[:ROAD {cost:
  100}]->(d), (b)-[:ROAD {cost: 40}]-
  >(d), (c)-[:ROAD {cost: 40}]->(d),
  (c)-[:ROAD {cost: 80}]->(e), (d)-
  [:ROAD {cost: 30}]->(e), (d)-[:ROAD
  {cost: 80}]->(f), (e)-[:ROAD {cost:
  40}]->(f);
  ```

```
CALL gds.graph.create(
  'myGraph',
  'Location',
  'ROAD', {
   relationshipProperties: 'cost'
 })
```

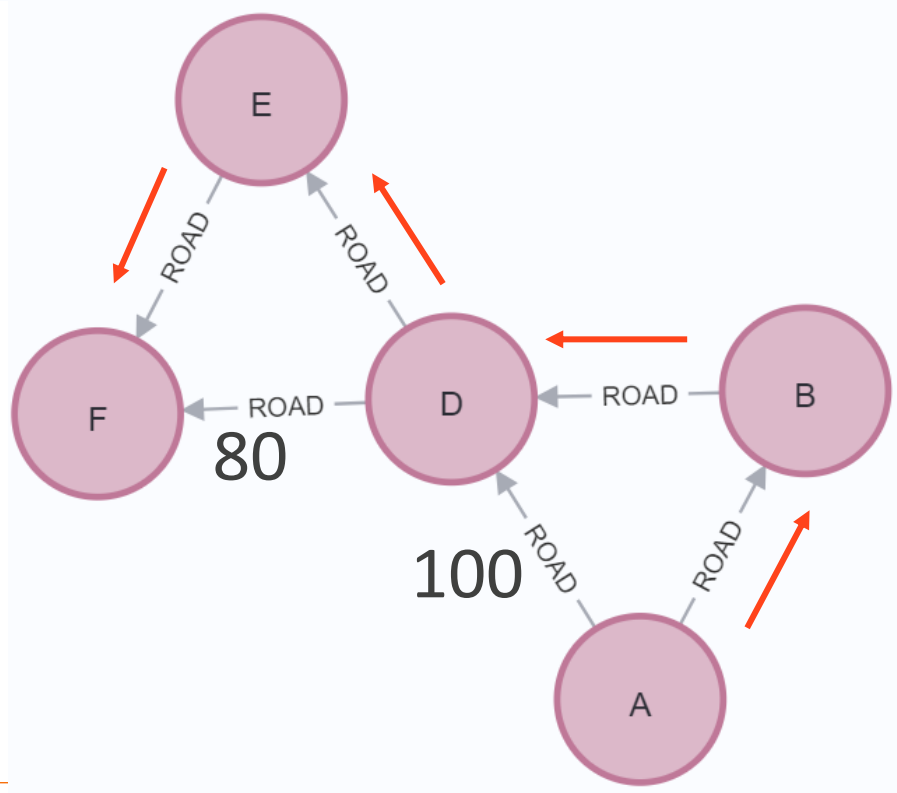

```

Example: The Algorithm Call on 'myGraph'

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
    sourceNode: source, targetNode: target,
relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    index,
    gds.util.asNode(sourceNode).name AS sourceNodeName,
    gds.util.asNode(targetNode).name AS targetNodeName,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
    costs,   nodes(path) as path
ORDER BY index
```

Example: Results

sourceNodeName	targetNodeName	totalCost	nodeNames	costs
"A"	"F"	160.0	["A", "B", "D", "E", "F"]	[0.0, 50.0, 90.0, 120.0, 160.0]



Summary

1. Don’t consider big data unless your data needs to be distributed
2. Hadoop was foundational in Big Data but is difficult to manage
3. Its easy to start with Spark and scale out as your needs grow, and use the cloud.
4. NoSQL is more about special-purpose data models at scale than replacing relational DB’s.
5. We looked at 3 different models: Document, Big Table, and Graph discussing designs and use cases.

Syracuse University
School of Information Studies

Applications of Opensource in Big Data Management with Examples

Michael Fudge

Professor of Practice

Syracuse University

CASL Workshop, December 2022

https://github.com/mafudge/cas-library-bigdata-workshop-2022