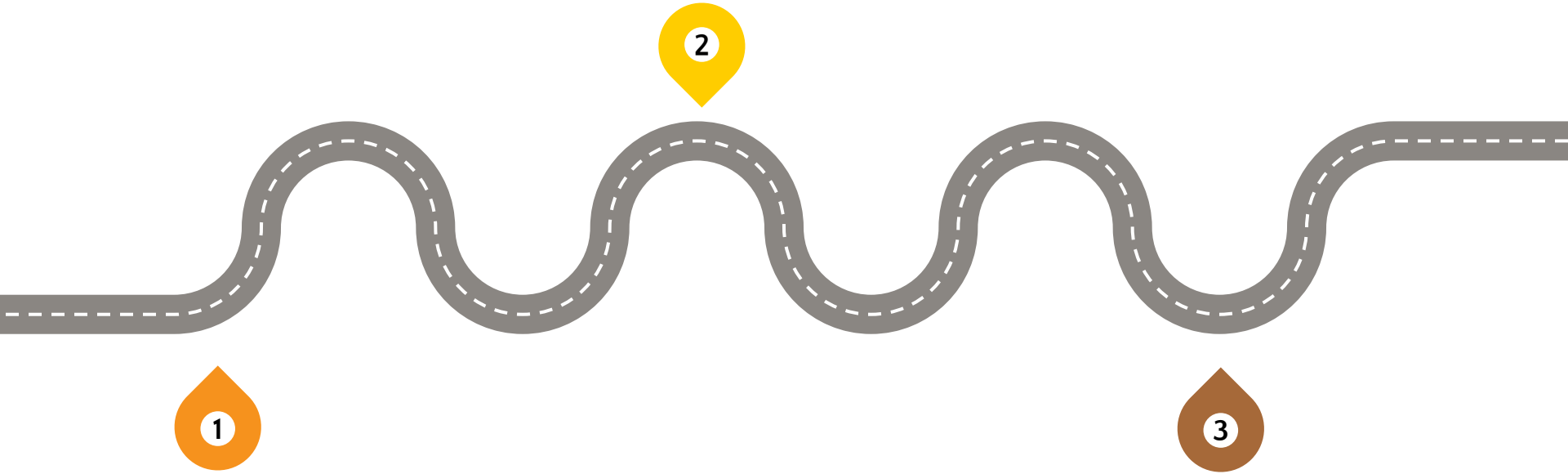
Big Data

Prof. Jefferson O. SILVA, Ph.D.

Course roadmap

An Introduction to PySpark



A **Brief** Introduction to Big Data

End-to-End programs

Learning Outcomes

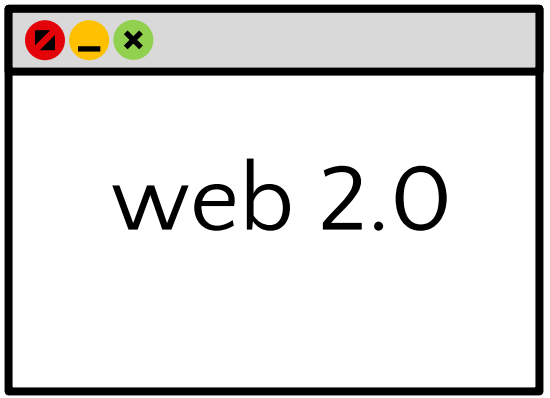
By the end, you should be able to:

- LO#1: Explain what Big Data is
- LO#2: Explain why we need Big Data
- LO#3: Identify typical problems when scaling traditional databases

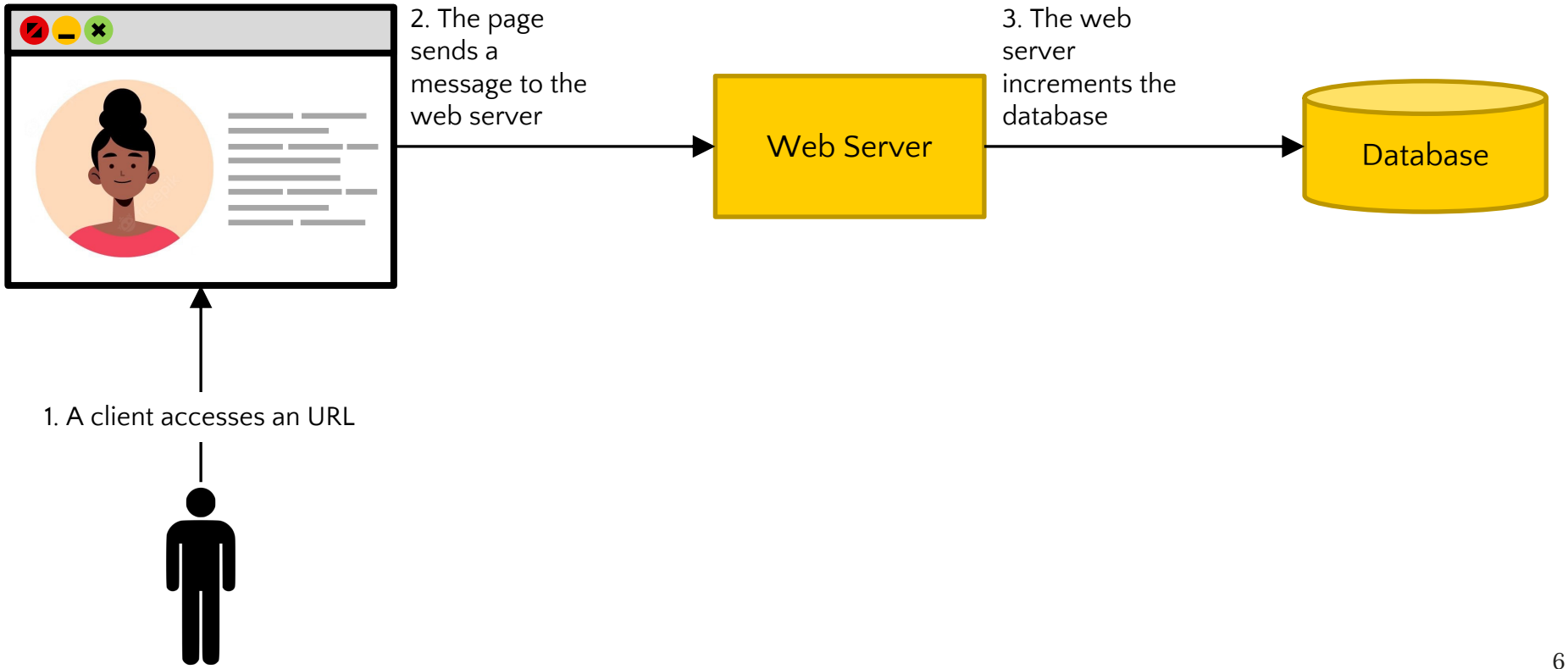
An Introduction to Big Data

The origins of the Web

vs



Building a web analytics application

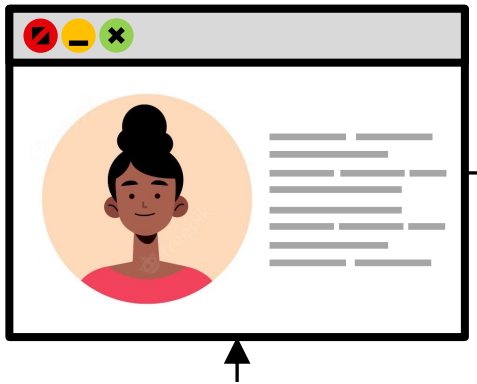


Building a web analytics application: DB schema

Column name	Type
`id`	`integer`
`user_id`	`integer`
`url`	`varchar(255)`
`pageviews`	`bigint`

Relational schema for a simple analytics application

A victim of its own success



2. The page sends a message to the web server

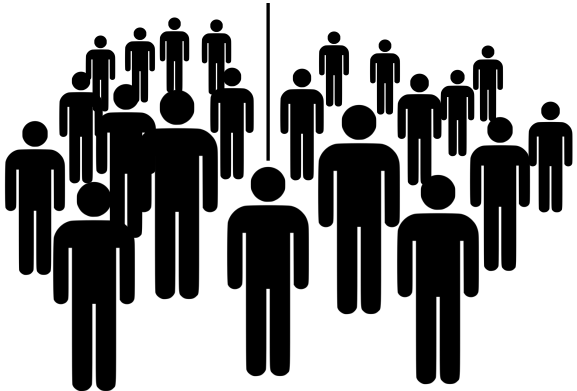
Web Server

3. The web server increments the database

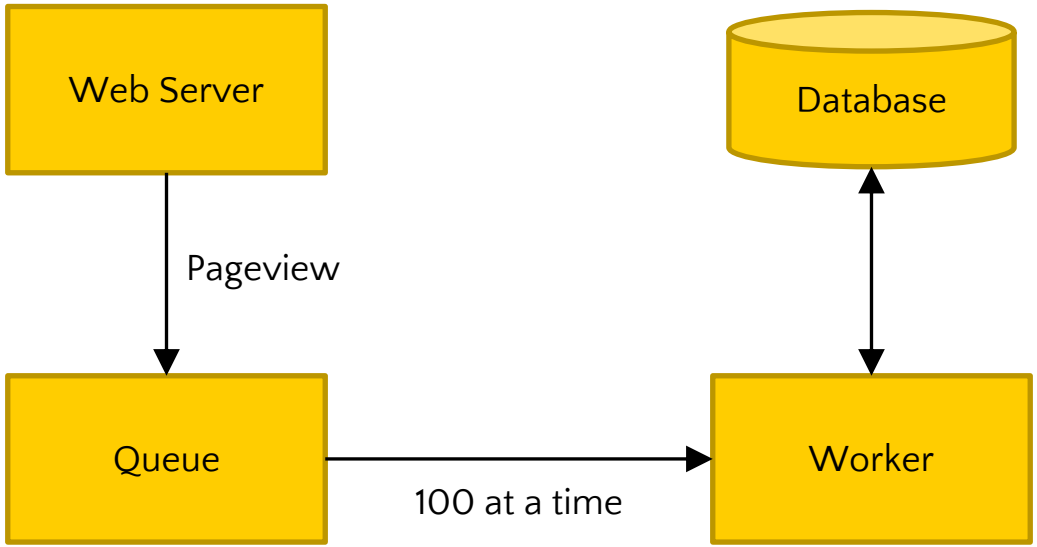
Database

4. The DB increments some requests but fails for others. Timeout errors

1. Lots of clients access the URLs

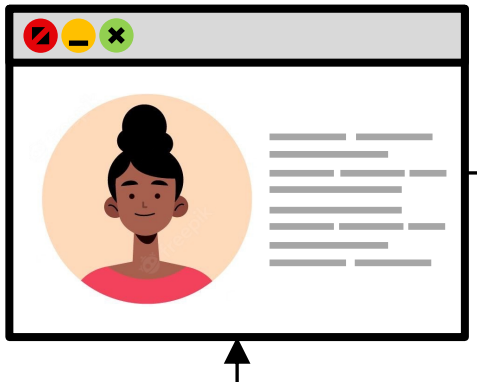


Scaling with a queue

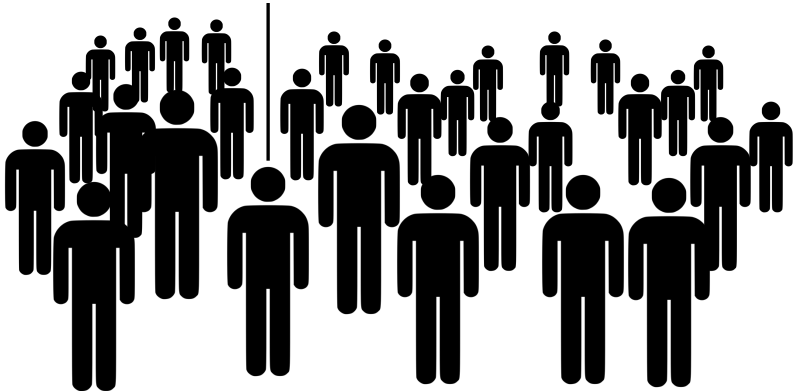


Batching updates with queue and worker

A victim of its own success (part 2)



1. Lots of clients access the URLs



Web Server

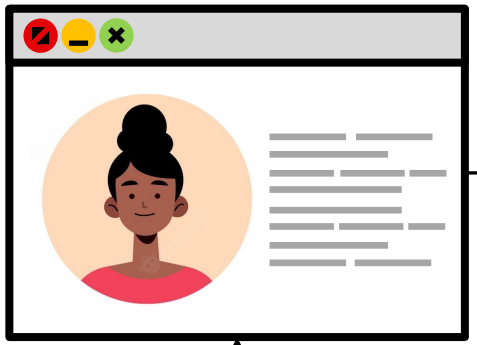
Database

Queue

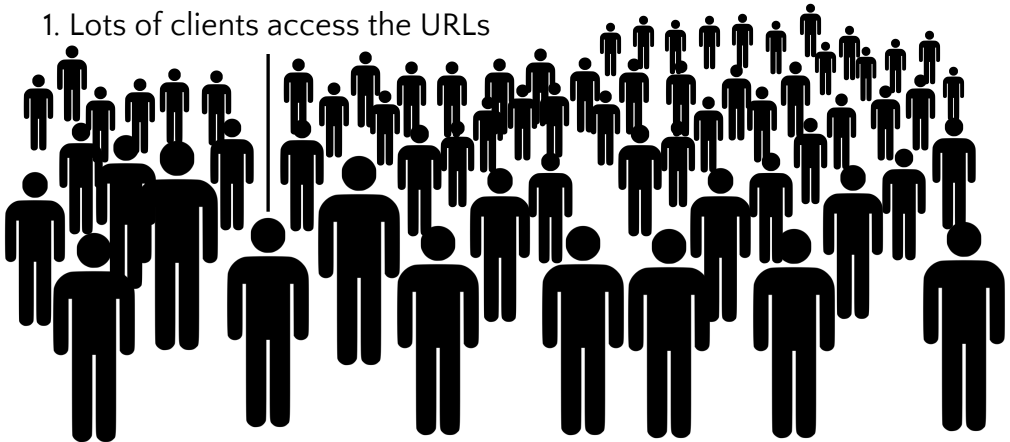
Worker

The worker reads 100 requests off the queue, and batches them in a single update

A victim of its own success (part 3)



1. Lots of clients access the URLs



Web Server

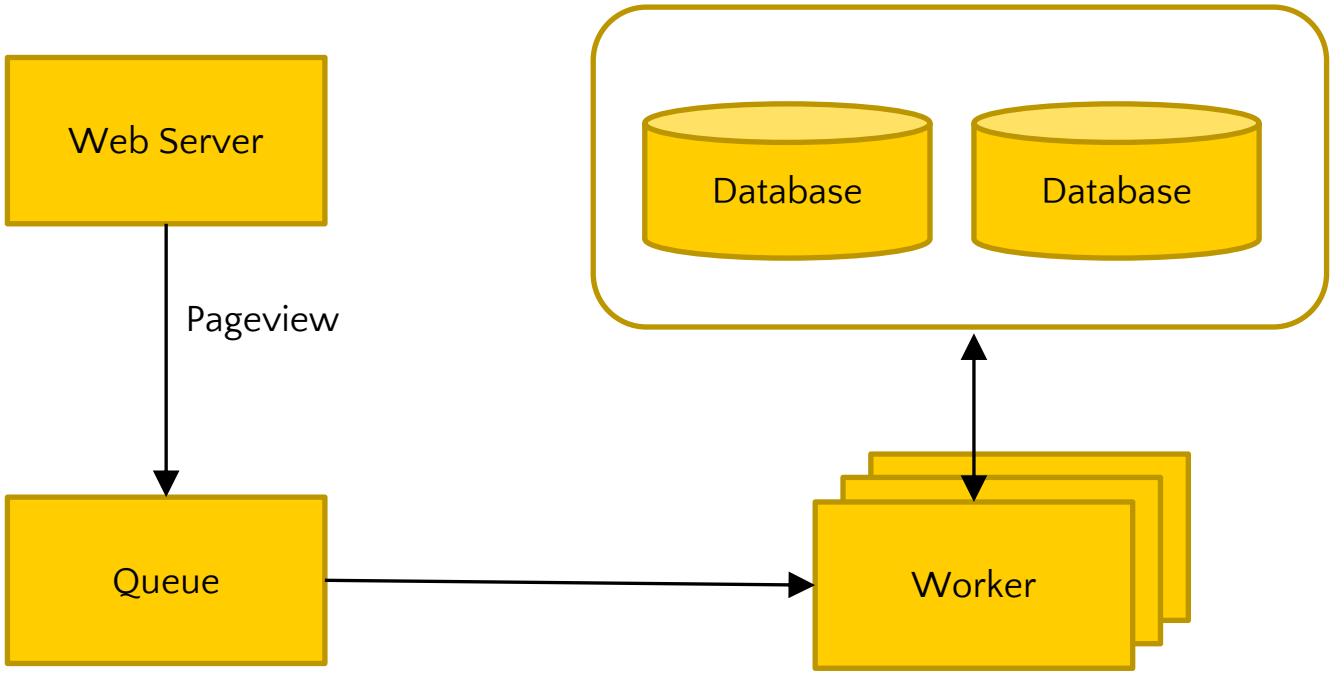
Database

Queue

Worker

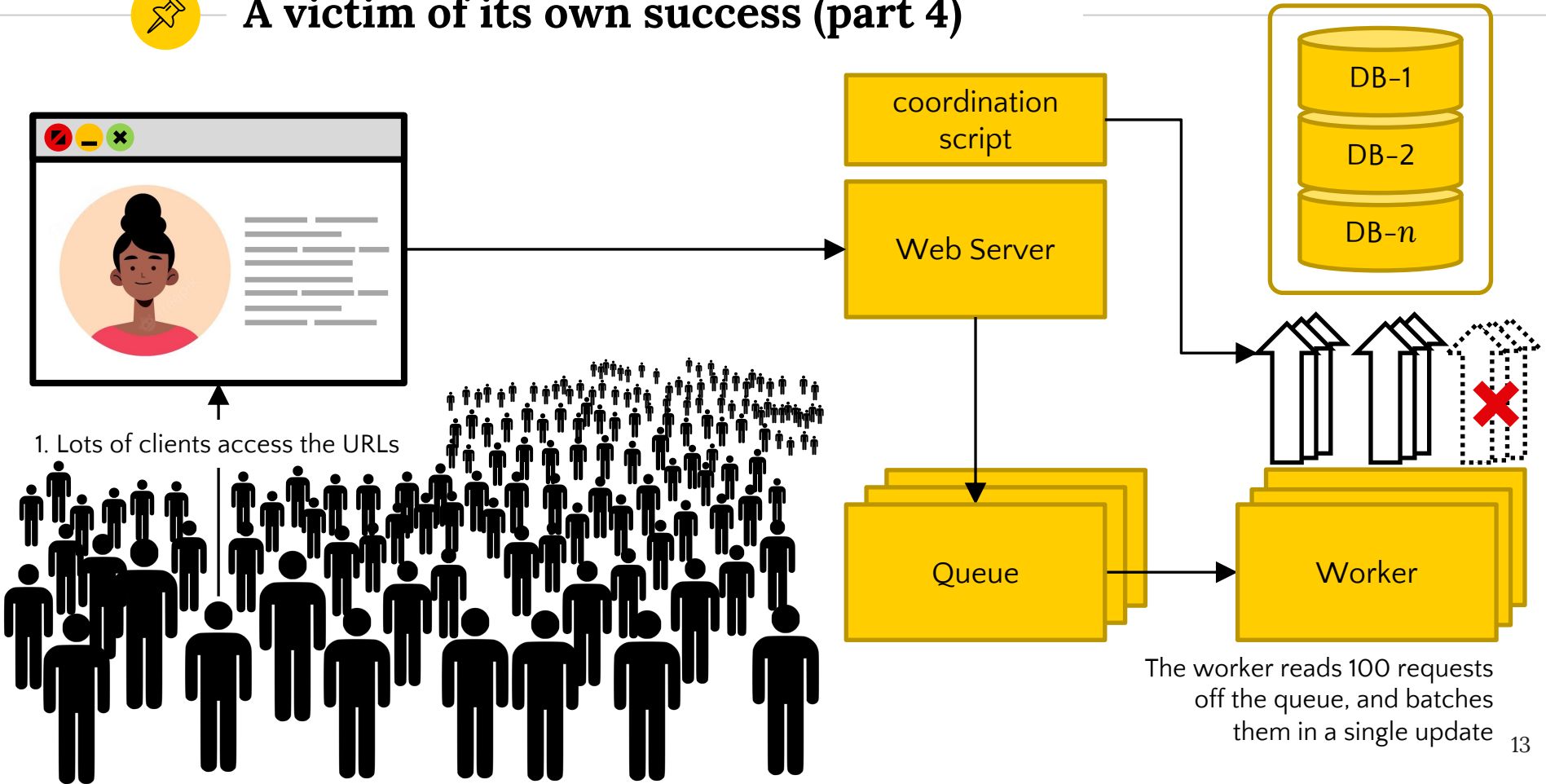
The worker reads 100 requests off the queue, and batches them in a single update

Scaling with horizontal partioning



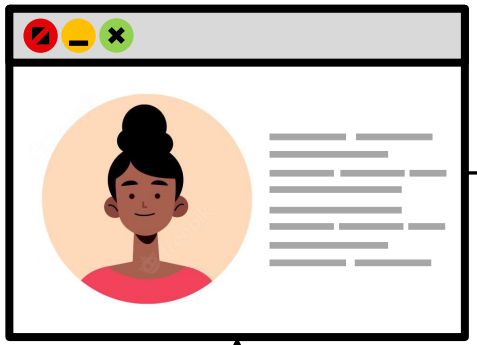
Sharding

A victim of its own success (part 4)

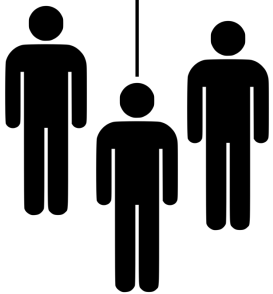


In the early days I spent my time building new features for customers. Now it seems that I am spending all my time just dealing with ***problems of reading and writing****.*

Dealing with fault-tolerance

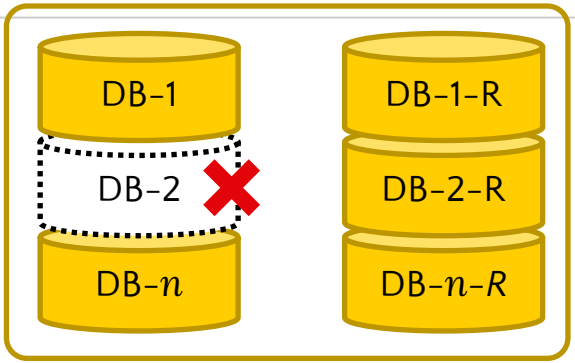


1. Lots of clients access the URLs



coordination script

Web Server



Queue

Worker

The worker reads 100 requests off the queue, and batches them in a single update

Dealing with corruption issues

- You deploy a bug into production that increments the number of pageviews by 2.
- You don’t notice until 24 hours later. Your weekly backups don’t help because there is no way of telling which data got corrupted
- After all this work trying to make your system scalable and fault tolerant, your system has no resilience to a human making a mistake

What went wrong?

- Developing data applications requires more than knowing database schemas. Your code also needs to know how to talk to the right shards
- The database is not self-aware of its distributed nature, so it can't help you deal with shards, replication, and distributed queries
- The system is not engineered for human mistakes
- Backups are not enough; the system must be carefully thought out to limit the damage a human mistake can cause

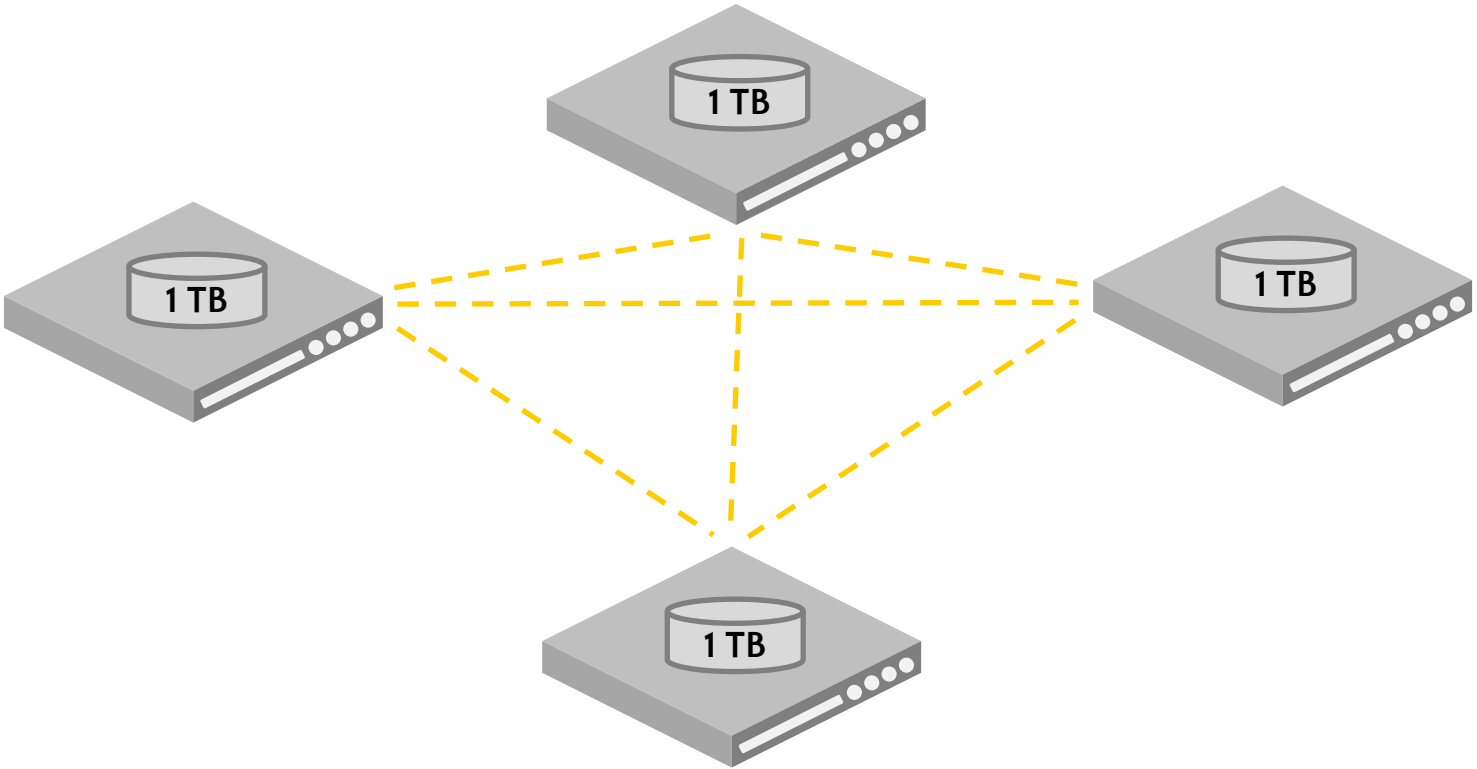
[Practice #01]

- P01. What are the typical problems when we try to scale databases for dealing with applications that need to read and write lots of data?
- P02. Describe fault-tolerance with your words.
- P03. Describe the corruption issues with your words.
- P04. How would you define Big Data?

Traditional relational database-centered architectures are at the heart of the problems Big Data addresses

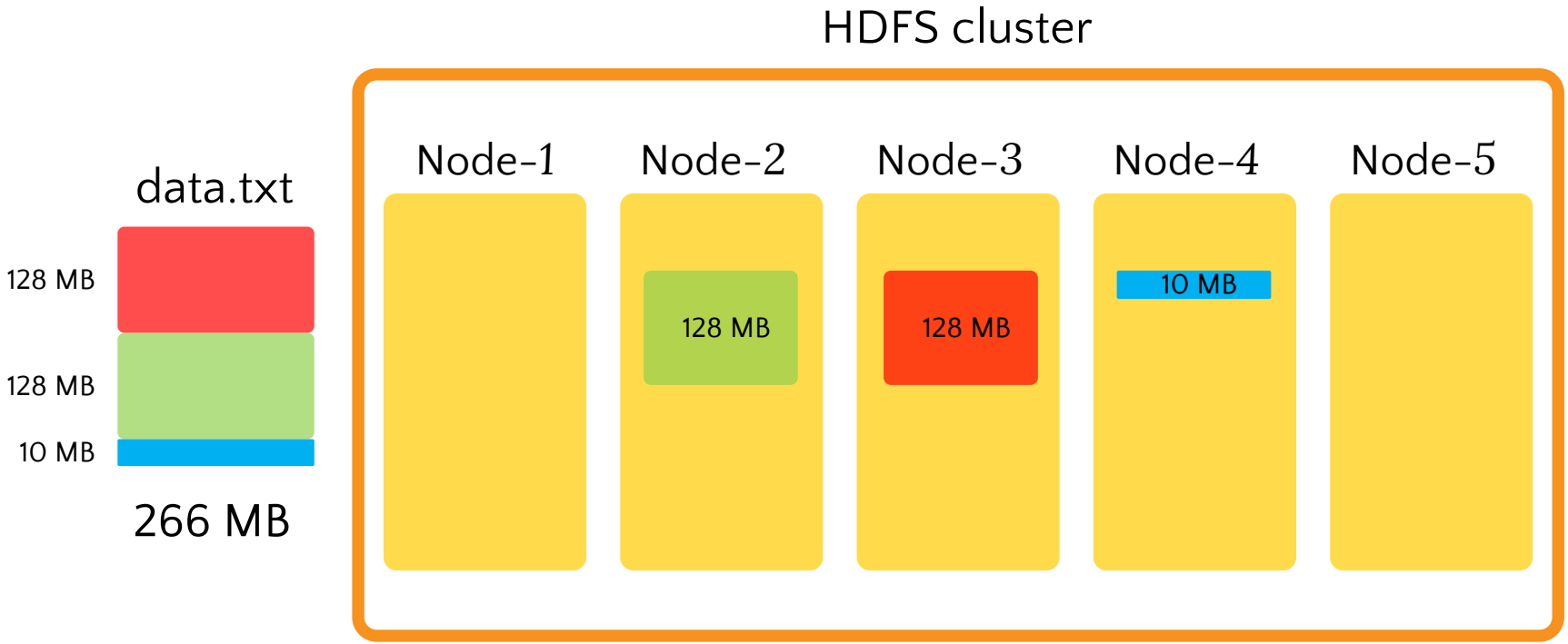
1 Big Data storage solutions

Distributed File Systems

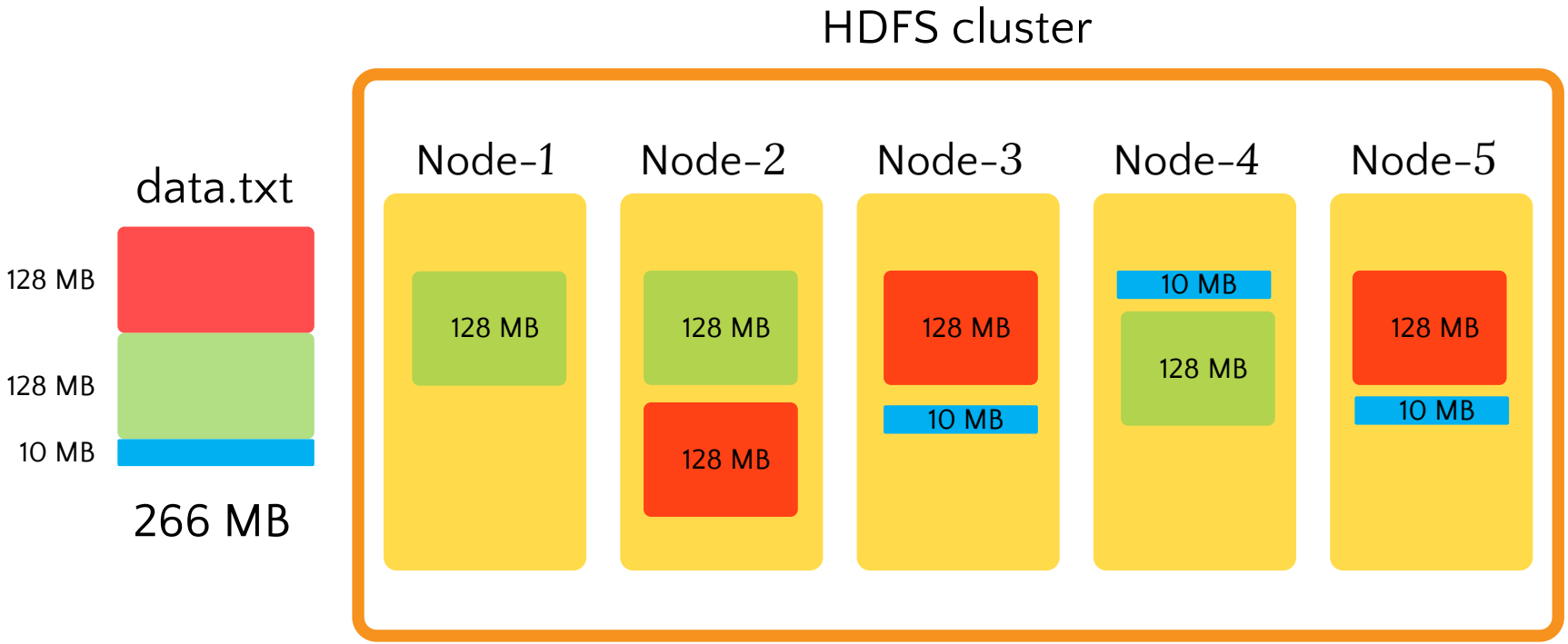


Apache Hadoop is an open source framework that is used to efficiently store and process large datasets ranging in size from gigabytes to petabytes of data. Instead of using one large computer to store and process the data, Hadoop allows clustering multiple computers to analyze massive datasets in parallel more quickly.

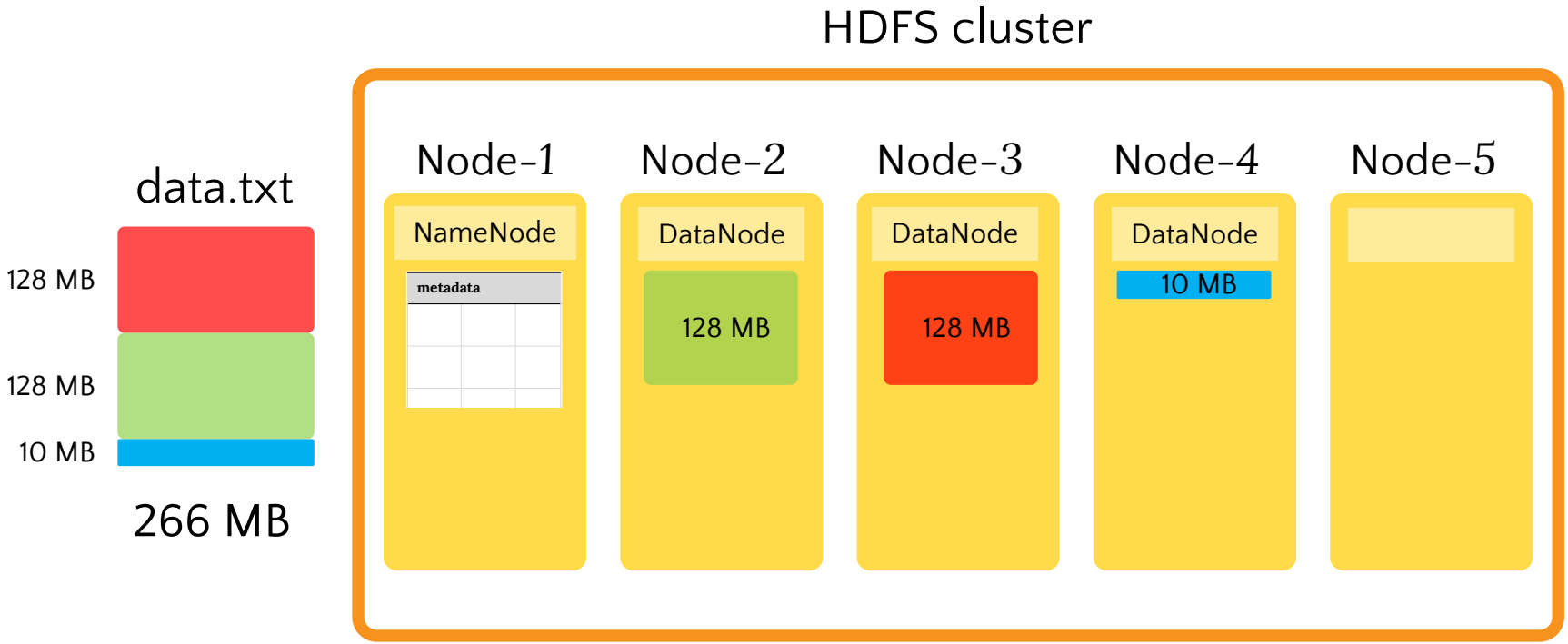
Hadoop Distributed File System (HDFS)



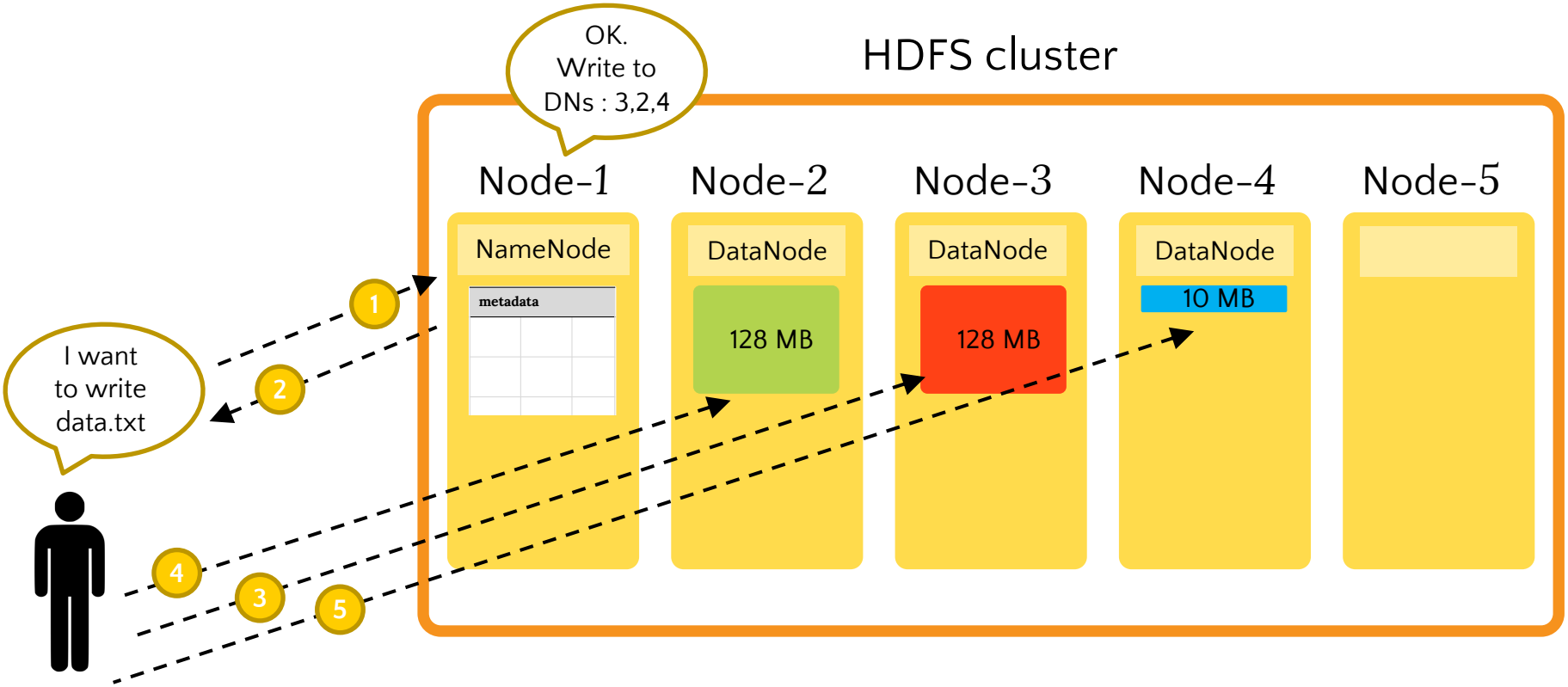
Hadoop Distributed File System: replication factor (2)



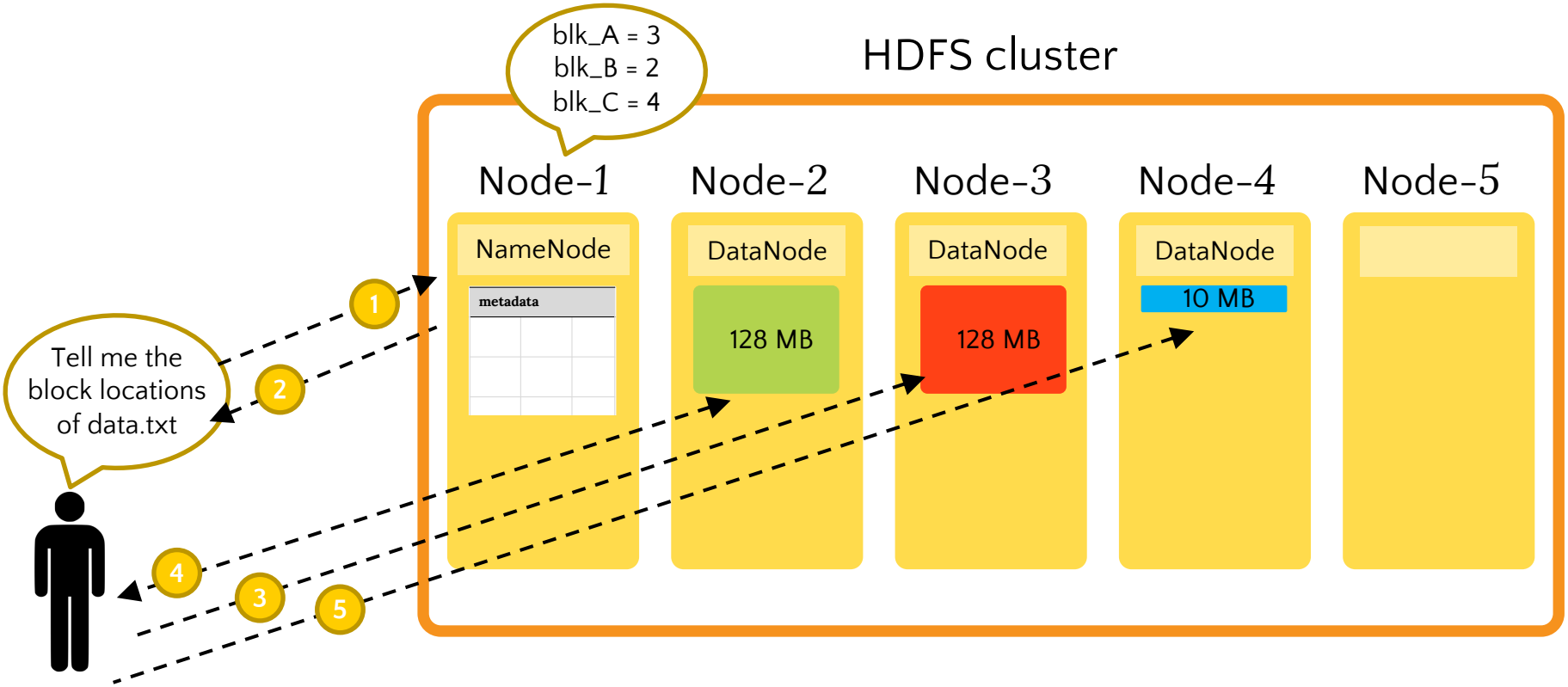
NameNodes and DataNodes



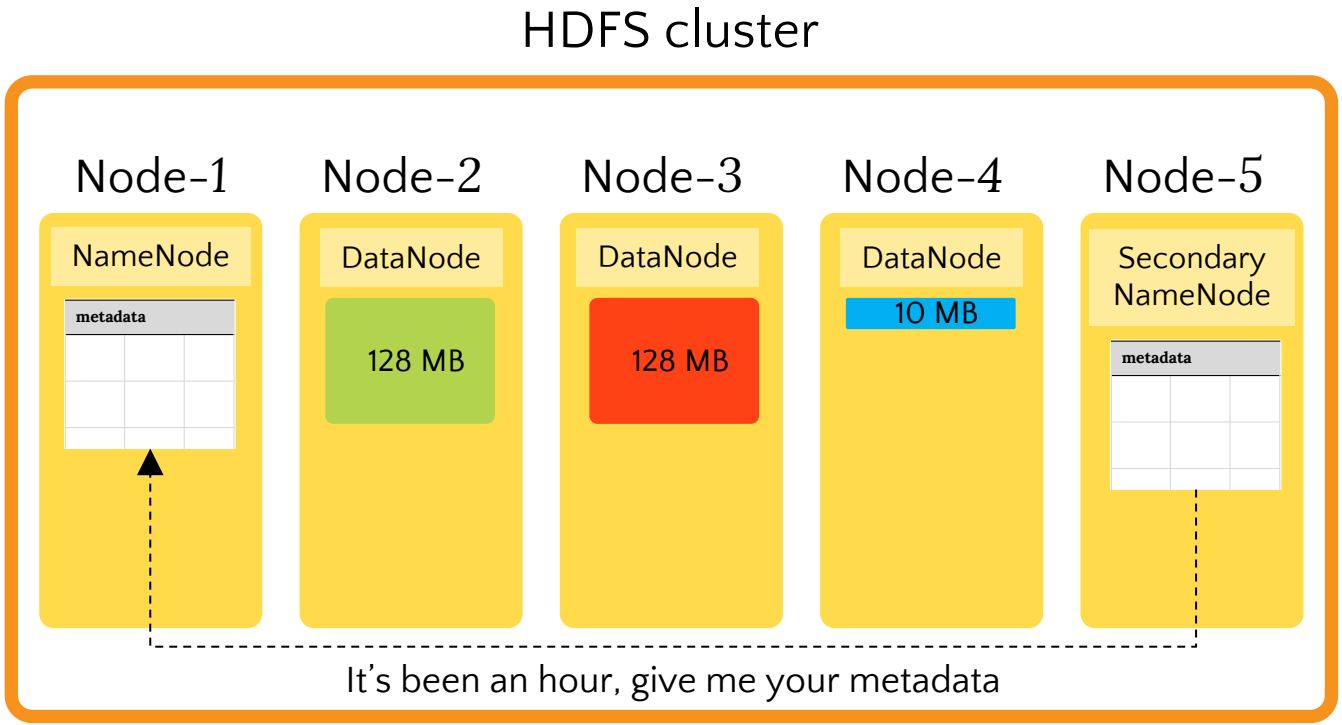
Writing files



Reading files

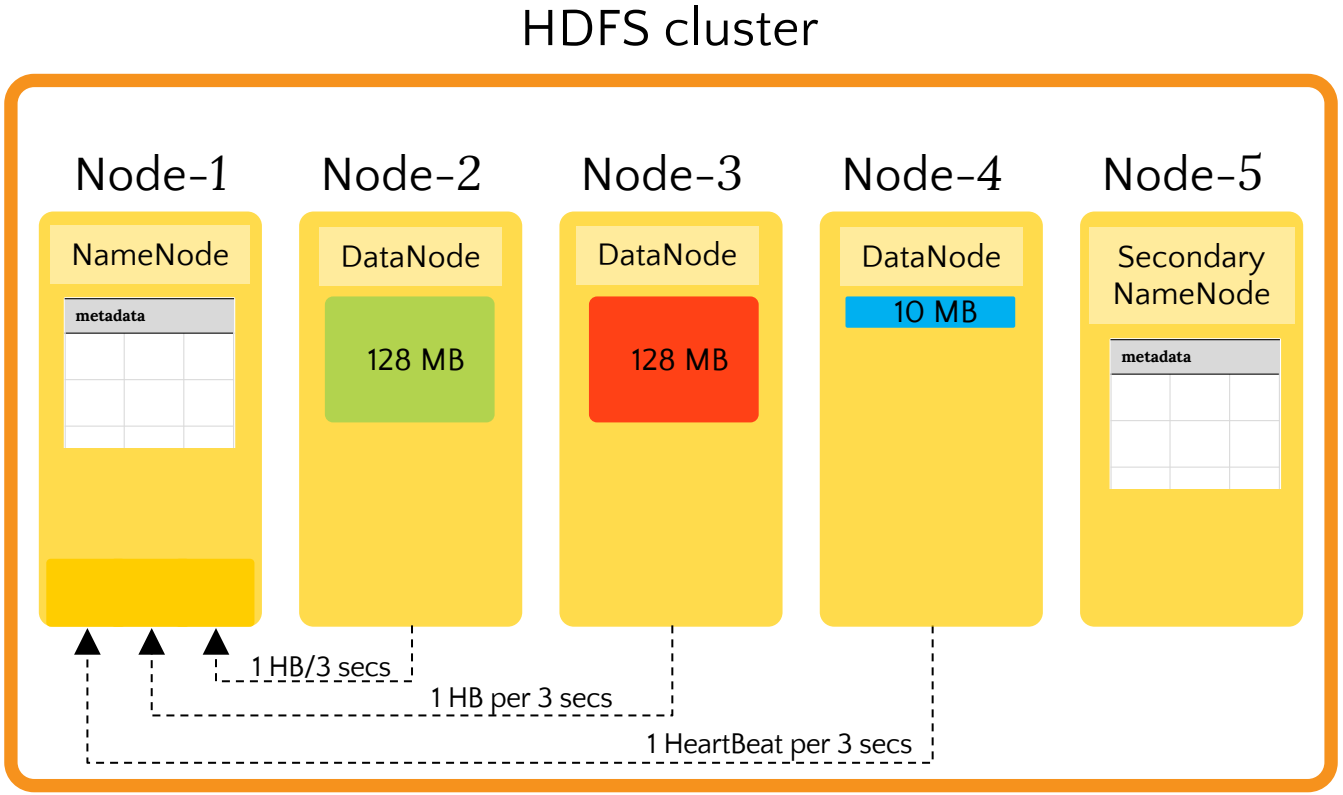


What happens if the NameNode is down?



- The Secondary NameNode is not a backup for the NameNode
- Saved metadata can rebuild a failed NameNode (faster)

Heartbeats


HDFS cluster
Node-1
Node-2
Node-3
Node-4
Node-5
NameNode
metadata
DataNode
128 MB
DataNode
128 MB
DataNode
10 MB
Secondary NameNode
metadata
1 HB/3 secs
1 HB per 3 secs
1 HeartBeat per 3 secs

[Practice #03]

- P01. Describe with your own words the process of **writing** in HDFS.
- P02. Describe with your own words the process of **reading** in HDFS.
- P03. How HDFS achieves fault-tolerance?
- P04. What are the HeartBeats?
- P05. What happens if a NameNode is down?

An Introduction to PySpark

Limitations of Hadoop

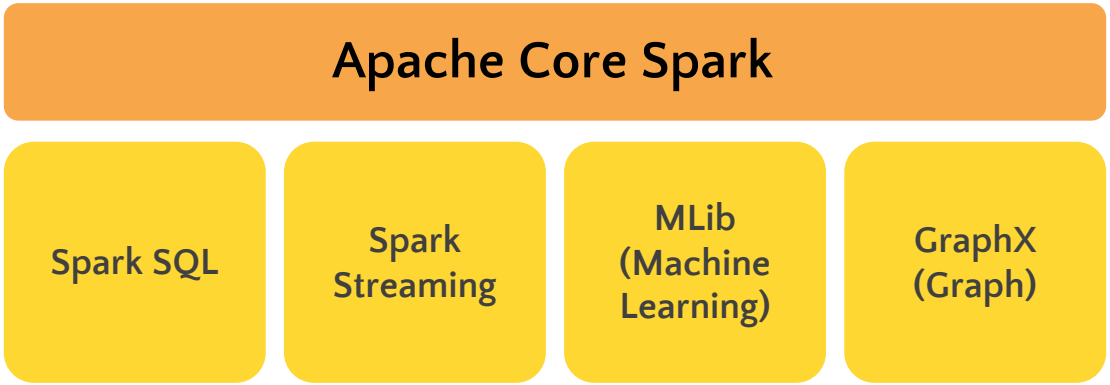
- Hadoop with its HDFS and MapReduce (in their pure form) are too “low level” technologies, especially for non-technical folks.

A Definition of Spark

- Spark is a unified analytics engine for large-scale data processing.
- Spark provides a powerful API that makes it look like you are working with a cohesive, non-distributed source of data, while working hard in the background to optimize your program to use all the power available.

Spark is a good tool because of what you can do with it, but especially because of what you don't have to do with it.

Spark's Components



PySpark

PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment.

The Factory Analogy

Installing PySpark

- Go to Google Colaboratory:

https://colab.research.google.com/

- Open a notebook and run on a cell:

```
!pip install pyspark py4j
```

SparkContext

- `SparkContext` is the entry point to any spark functionality. When we run any Spark application, a driver program starts, which has the main function and your `SparkContext` gets initiated here.

- The driver program then runs the operations inside the executors on worker nodes.

Initializing Spark Session: command

To obtain a `SparkContext` we need to obtain a `SparkSession` first. A `SparkSession` can be used create `DataFrame`, register `DataFrame` as tables, execute SQL over tables, cache tables, and read parquet files.

```
from pyspark.sql import SparkSession
spark = SparkSession \
          .builder \
          .appName("<The name of the app>") \
          .config("<spark.some.config.option>", "some-value") \
          .getOrCreate()
spark
```

Initializing Spark Session: example

An example

```
from pyspark.sql import SparkSession
spark = SparkSession \
          .builder \
          .appName("DAD_BIGDATA") \
          .getOrCreate()
spark
```

Getting the `SparkContext` : command

You can obtain Spark's context in this way.

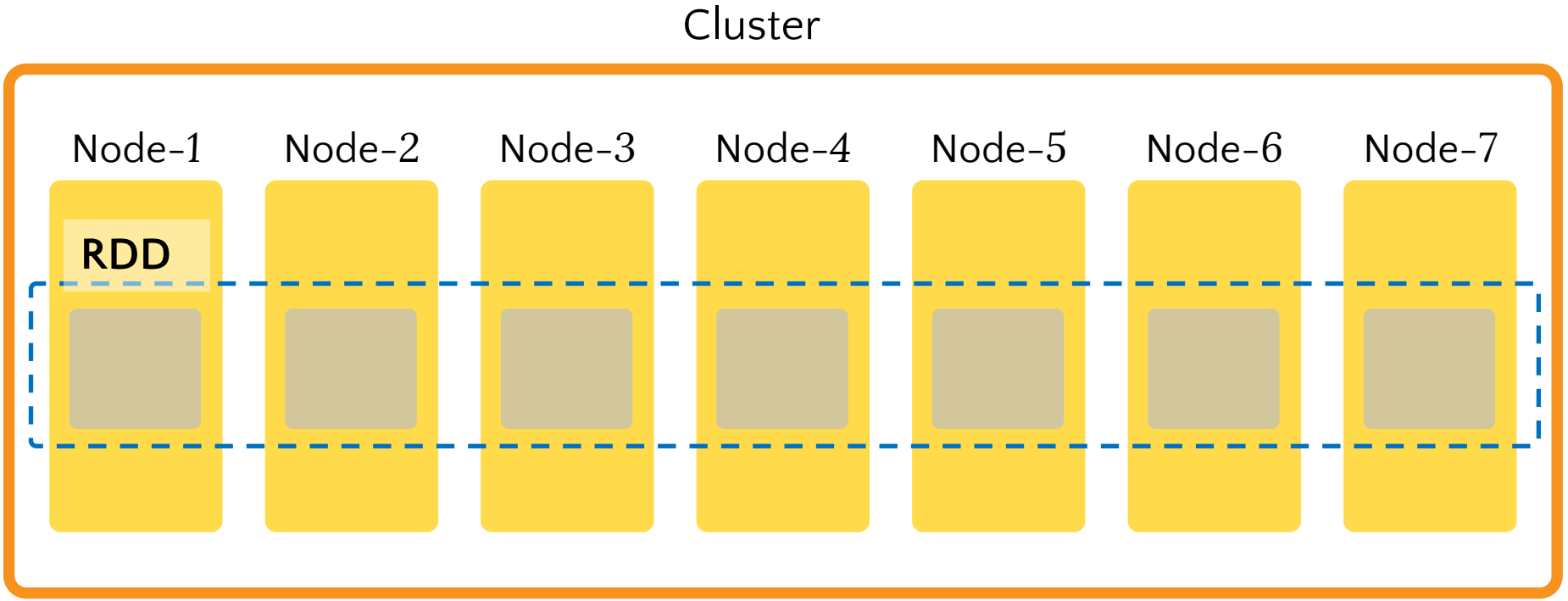
```
sc = spark.sparkContext
sc
```

3 Resilient Distributed Datasets

Resilient Distributed Datasets

- RDD stands for **Resilient Distributed Dataset**, these are the elements that run and operate on multiple nodes to do parallel processing on a cluster.
- RDDs are **immutable** elements, which means once you create an RDD you cannot change it
- RDDs are fault tolerant as well, hence in case of any failure, they recover automatically
- You can apply multiple operations on these RDDs to achieve a certain task.

An illustration of RDDs



RDD Operations

- **Transformation** – These are the operations, which are applied on a RDD to create a new RDD. Filter, groupBy and map are the examples of transformations.

- **Action** – These are the operations that are applied on RDD, which instructs Spark to perform computation and send the result back to the driver.

4 Managing DataFrames

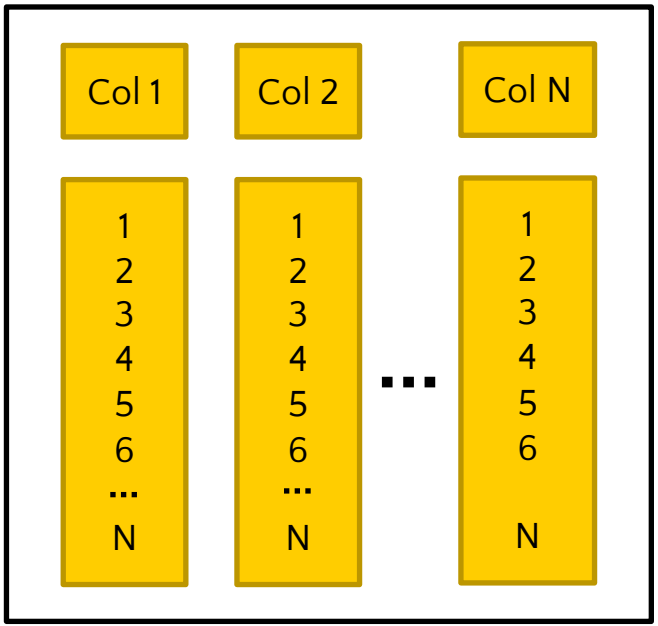
DataFrames

- A DataFrame is equivalent to a relational table in Spark SQL, and can be created using various functions in SparkSession.

RDD vs a data frame

Resilient Distributed Datasets (RDD)

Data frame (DF)



Creating a DataFrame from RDD: example

```
rdd = sc.parallelize([
   ("XS", 2018, 5.65, 2.79, 6.24),
   ("XR", 2018, 5.94, 2.98, 6.84),
   ("X10", 2017, 5.65, 2.79, 6.13),
   ("8Plus", 2017, 6.23, 3.07, 7.12)
])

cols = ['Model', 'Year', 'Height', 'Width', 'Weight’]

df = spark.createDataFrame(rdd, schema=cols)
df
```

Creating DataFrames from files

```
# Reading from CSVs
df_csv = spark.read.csv("<filepath>", \
                        header=True, inferSchema=True)
df_csv.show()

# Reading from JSONs
df_json = spark.read.json("<filepath>")
df_json.show()

# Reading from TXTs
df_txt = spark.read.text("<filepath>")
df_txt.show()
```

Inspecting DataFrames: commands (1/2)

```
# Return df column names and data types
df.dtypes

# Display the content of df
df.show()

# Return first n rows
df.head()

# Return the first row
df.first()

# Return the first n rows
df.take(n)

# Return the schema of df
df.schema
```

Inspecting DataFrames: commands (2/2)

```
# Compute summary statistics
df.describe().show()

# Return the columns of df
df.columns

# Count the number of rows in df
df.count()

# Count the number of distinct rows in df
df.distinct().count()

# Print the schema of df
df.printSchema()

# Print the (logical and physical) plans
df.explain()
```

5 Queries

`select` columns: examples

```
# Show all entries in column name
df.select("name").show()

# Show all entries in column name1, and name2
df.select("name", "age").show()

#  Show all entries in name and add 1 to the ages
df.select(df.name, df.age + 1)

#  Show all entries where age > 30
df.select(df.name, df.age > 30)
```

`alias`: examples

```
# Rename the column to something more meaningful

df.select(
    df.name,
    (df.age + 1).alias("age")
).show()
```

Grouping By: example

```
# Group people by their age

grouped = df.groupBy("age").count()

grouped.show()
```

`filter`: example

Filter people by their age. Only keep those records of which the values are > 30

```
filtered = df.filter(df.age > 30)

filtered.show()
```

`substring`: example

Returns substrings of the column specified

```
df.select(
    df.name.substr(1,3).alias("name")
).show()
```

`like`: example

Shows `name` and a Boolean column if there is a match with the string in `like()`

```
from pyspark.sql.functions import col

ndf = df.select(
          col("name"),
          col("name").like(”%Jona%")
      )
ndf.show()
```

`startswith`: example

Shows a `column` name and a boolean Column if the beginning of the string matches the string in `startswith`.

```
from pyspark.sql.functions import col

ndf = df.select(
          col("name"),
          col("name").startswith("M")
      )
ndf.show()
```

`endswith`: example

Shows a `column` name and a boolean Column if the ending of the string matches the string in `endswith`.

```
from pyspark.sql.functions import col

ndf = df.select(
            col("name"),
            col("name").endswith("M")
        )
ndf.show()
```

between: example

Shows a `column` name and a boolean Column if the column value is between the specified value.

```
from pyspark.sql.functions import col

ndf = df.select(
            col("age"),
            col("age").between(22,24)
       )
ndf.show()
```

6 Exploding columns

Motivating example

Imagine we have some data in which one of the columns is an array

```
data = [('Jaya', '20', ['SQL','Data Science']),
        ('Milan', '21', ['ML','AI']),
        ('Rohit', '19', ['Programming', 'DSA']),
        ('Maria', '20', ['DBMS', 'Networking']),
        ('Jay', '22', ['Data Analytics','ML'])]

columns = ['Name', 'Age', 'Courses_enrolled']

df = spark.createDataFrame(data, columns)
df.printSchema()
df.show()
```

Motivating example: output

```
root
 |-- Name: string (nullable = true)
 |-- Age: string (nullable = true)
 |-- Courses_enrolled: array (nullable = true)
 |    |-- element: string (containsNull = true)

+-----+---+--------------------+
| Name|Age|    Courses_enrolled|
+-----+---+--------------------+
| Jaya| 20| [SQL, Data Science]|
|Milan| 21|            [ML, AI]|
|Rohit| 19|  [Programming, DSA]|
|Maria| 20|  [DBMS, Networking]|
|  Jay| 22|[Data Analytics, ML]|
+-----+---+--------------------+
```

Exploding columns

```
from pyspark.sql.functions import explode

exploded_df = df.select(
    df.Name, explode(df.Courses_enrolled)
)
exploded_df.printSchema()
exploded_df.show()
```

Exploding columns: output

```
root
 |-- Name: string (nullable = true)
 |-- col: string (nullable = true)

+-----+--------------+
| Name|           col|
+-----+--------------+
| Jaya|           SQL|
| Jaya|  Data Science|
|Milan|            ML|
|Milan|            AI|
|Rohit|   Programming|
|Rohit|           DSA|
|Maria|          DBMS|
|Maria|    Networking|
|  Jay|Data Analytics|
|  Jay|            ML|
+-----+--------------+
```

[Practice #04]

For the following exercises, consider the file `movies.csv`

- P01. Obtain the statistics summary for the numeric fields.
- P02. Which movies have an audience score over 70?
- P03. Count how many movies for each year.
- P04. Which movies start with the word “The” in their titles?
- P05. Which movies have a Worldwide Gross between 100 and 200?

End-to-End Programs in PySpark

What are the most popular words in the English language?

Reducing the problem

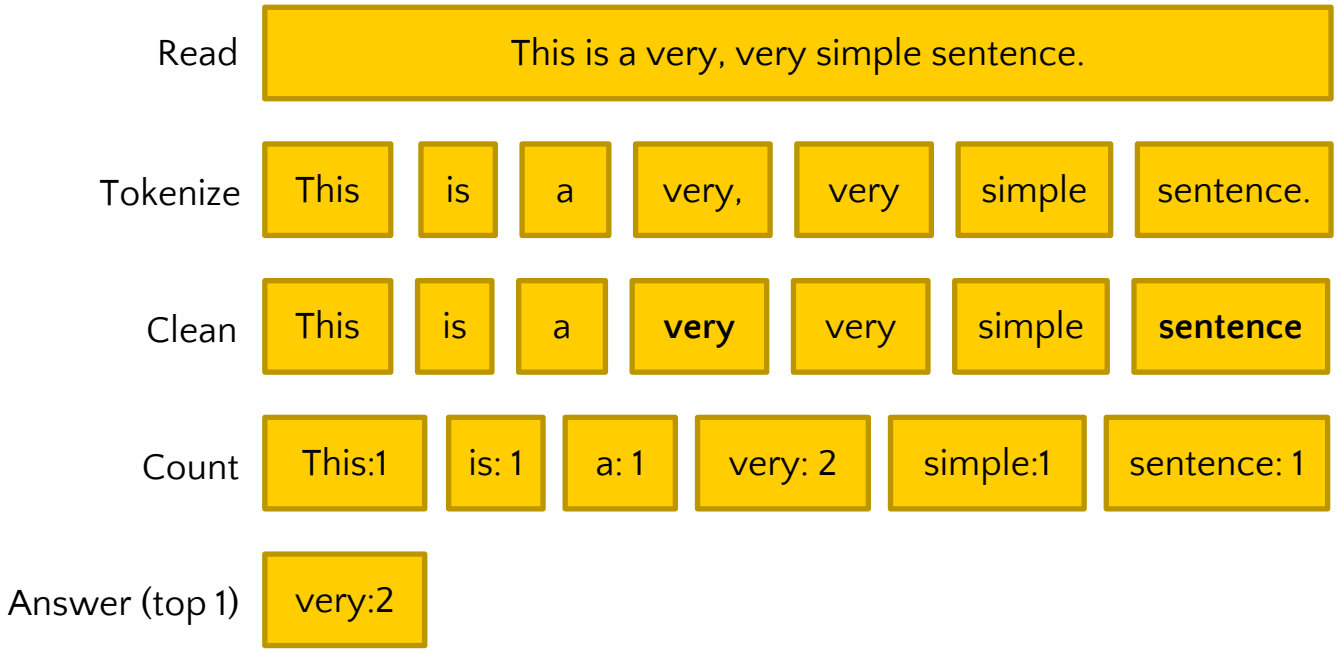
- Counting all the words in the English language is certainly a task worthy of the best Big Data warriors

- For learning purposes, we will start with a relatively small source: “Pride and Prejudice,” a Jane Austen’s book

- After getting our program working, we can scale it to accommodate more and more texts

Major steps

Major steps: example



7 Reading the book

Reading the book

1. Get a file named “`pride-and-prejudice.txt`” in the Course Materials folder on Microsoft Teams

2. Upload the file to Google Colab

3. Execute the following command:

```
book = spark.read.text( \
          "/content/pride-and-prejudice.txt”)
book
```

8 Tokenizing the sentences

Renaming a column via alias

```
from pyspark.sql.functions import split

lines = book.select( \
            split(book.value, " ").alias("line") \
        )
lines.show(5)
```

```
+--------------------+
|                line|
+--------------------+
|[The, Project, Gu...|
|                  []|
|[This, eBook, is,...|
|[almost, no, rest...|
|[re-use, it, unde...|
+--------------------+
only showing top 5 rows
```

Exploding columns

```
# Exploding a column of arrays into rows of elements

from pyspark.sql.functions import explode, col

words = lines.select(explode(col("line")).alias("word"))
words.show(5)

+---------+
|     word|
+---------+
|      The|
|  Project|
|Gutenberg|
|    EBook|
|       of|
+---------+
only showing top 5 rows
```

9 Cleaning

Lowering the case of the words

```
# Lowercase the case of the words in the data frame
from pyspark.sql.functions import lower

lowered_words = words.select( \
                  lower(col("word")).alias(”lowered_word") \
                )
lowered_words.show(5)
```

```
+------------+
|lowered_word|
+------------+
|         the|
|     project|
|   gutenberg|
|       ebook|
|          of|
+------------+
only showing top 5 rows
```

Applying Regular Expressions

```
# Using the regexp_extract to keep what looks like a word
from pyspark.sql.functions import regexp_extract

cleaned_words = lowered_words.select( \
  regexp_extract(col("lowered_word"), "[a-z]*", 0).alias("cleaned_word")
)
cleaned_words.show(5)

+------------+
| cleaned_word |
+------------+
|         the|
|     project|
|   gutenberg|
|       ebook|
|          of|
+------------+
only showing top 5 rows
```

Filtering `null` values

Filtering rows in the dataframe, using where or filter

```
words_nonull = cleaned_words.where(col("cleaned_word") != "")
words_nonull = words_nonull.select(col("cleaned_word").alias("word"))
words_nonull.show(5)
```

```
+---------+
|     word|
+---------+
|      the|
|  project|
|gutenberg|
|    ebook|
|       of|
+---------+
only showing top 5 rows
```

10 Counting

Counting word frequencies

Counting word frequencies using groupby() and count()

```
counts = words_nonull.groupby(col("word")).count()
counts.show(10)
```

```
+---------+-----+
|     word|count|
+---------+-----+
|   online|    4|
|     some|  203|
|    still|   72|
|      few|   72|
|     hope|  122|
|    those|   60|
| cautious|    4|
|imitation|    1|
|      art|    3|
|  solaced|    1|
+---------+-----+
only showing top 10 rows
```

Sorting the results

Sorting and displaying the top 10 word in Jane Austen's Pride and Prejudice

```
counts.orderBy("count", ascending=False).show(10)
```

```
+----+-----+
|word|count|
+----+-----+
| the| 4480|
|  to| 4218|
|  of| 3711|
| and| 3504|
| her| 2199|
|   a| 1982|
|  in| 1909|
| was| 1838|
|   i| 1750|
| she| 1668|
+----+-----+
only showing top 10 rows
```

Our results are totally USELESS

11 Cleaning (part 2)

Grouping and Counting

```
groups = meaningful_nonull.groupby(col("word"))
word_counts = meaningful_nonull.groupby(col("word")).count()
word_counts.show(5)
```

```
+---------+-----+
|     word|count|
+---------+-----+
|   online|    4|
|    still|   72|
|     hope|  122|
| cautious|    4|
|imitation|    1|
+---------+-----+
only showing top 5 rows
```

Sorting

```
groups = meaningful_nonull.groupby(col("word"))
word_counts = meaningful_nonull.groupby(col("word")).count()
word_counts.show(5)
```

```
+---------+-----+
|     word|count|
+---------+-----+
|   online|    4|
|    still|   72|
|     hope|  122|
| cautious|    4|
|imitation|    1|
+---------+-----+
only showing top 5 rows
```

Stop words

- Stop words are words like "the", "to", "of", "and", "her", "a", "in" etc.
- PySpark offers the class `StopWordsRemover` for this task

Each line as a list

`StopWordsRemover` requires a list of words to apply the transformation

```
# Transforming each word into a list
words_lst_df = words_nonull.select(
    split(col("word"), " ").alias("word")
)
words_lst_df.show()
+----------+
|      word|
+----------+
|     [the]|
|  [project]|
|[gutenberg]|
|    [ebook]|
|      [of]|
+----------+
only showing top 5 rows
```

Preparing for removing the stop words

```
from pyspark.ml.feature import StopWordsRemover

swr = StopWordsRemover(
         inputCol="word", outputCol="meaningful”
      )
swr_df = swr.transform(words_lst_df)
swr_df.show(5)
```

```
+-----------+-----------+
|       word| meaningful|
+-----------+-----------+
|      [the]|         []|
|  [project]|  [project]|
|[gutenberg]|[gutenberg]|
|    [ebook]|    [ebook]|
|       [of]|         []|
+-----------+-----------+
only showing top 5 rows
```

Removing the stop words

```
meaningful = swr_df.select(
                    (col("meaningful")[0]).alias("word")
             )
meaningful.show(5)
```

```
+---------+
|     word|
+---------+
|     null|
|  project|
|gutenberg|
|    ebook|
|     null|
+---------+
only showing top 5 rows
```

Filtering null values

```
meaningful_nonull = meaningful.where(col("word") != "null")

meaningful_nonull.show()

+---------+
|     word|
+---------+
|  project|
|gutenberg|
|    ebook|
|    pride|
|prejudice|
+---------+
only showing top 5 rows
```

12 Counting (part 2)

Counting and sorting

```
groups = meaningful_nonull.groupby(col("word"))
word_counts = meaningful_nonull.groupby(col("word")).count()
word_counts.orderBy("count", ascending=False).show(10)
```

```
+---------+-----+
|     word|count|
+---------+-----+
|       mr|  766|
|elizabeth|  632|
|    darcy|  418|
|     said|  401|
|      mrs|  338|
|     much|  327|
|   bennet|  323|
|     must|  314|
|  bingley|  304|
|     jane|  294|
+---------+-----+
only showing top 10 rows
```

13

The answer is actually an analysis

Analyzing our results

- After removing the stop words, we obtained more useful results. Our data shows that Elizabeth is more cited than Mr. Darcy

- Like in real-world scenarios, our work is not complete. We still have words that are useless for analysis such as “mr”, “said”, “mrs”, “must” etc. An analyst would have to find other means to remove these words (including manual removal in some cases)

[Project]

Choose one book from the Course Materials/books folder and do the same analysis that was made in this presentation, counting the most popular words.

Questions?

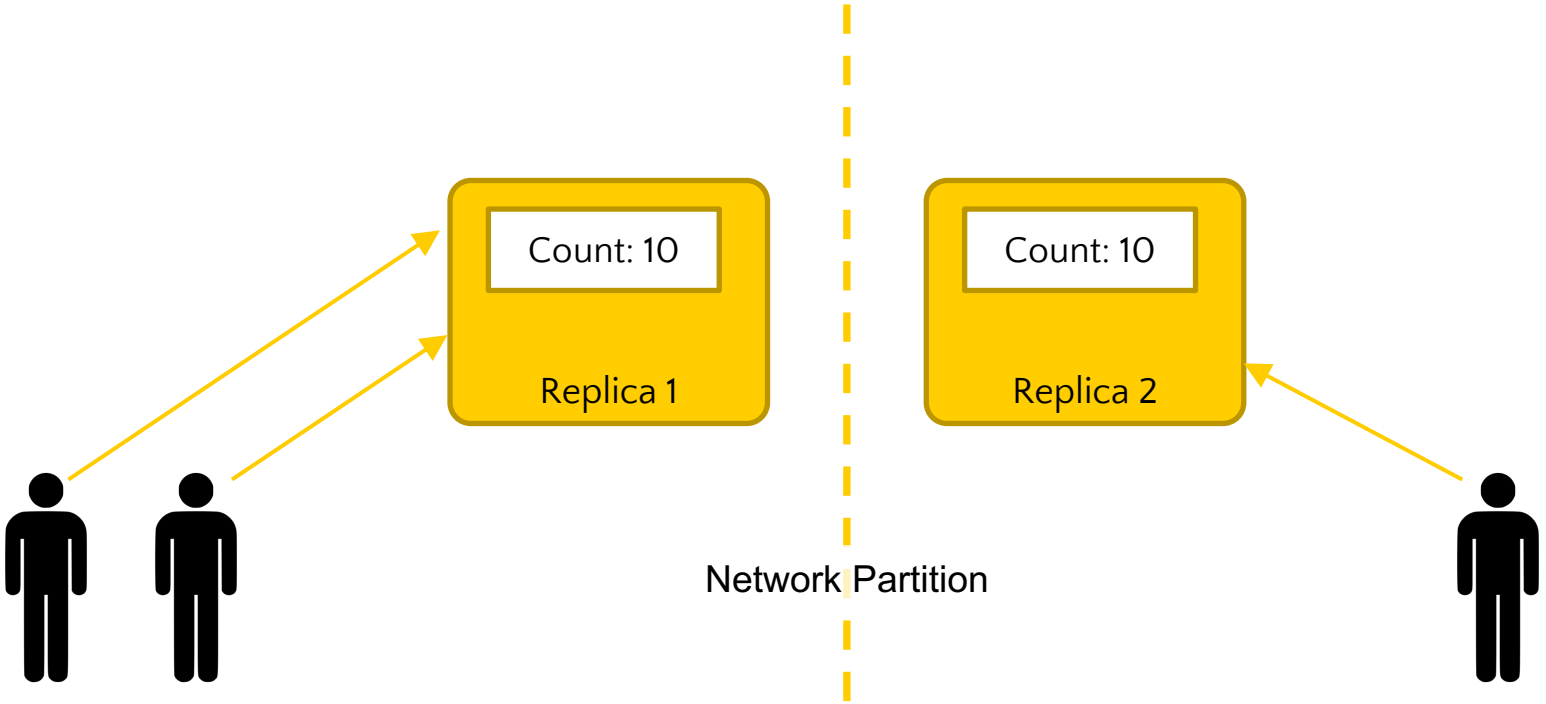
1 Backup

Deciding how chatty you want PySpark to be

```
spark.sparkContext.setLogLevel(KEYWORD)
```

Keyword	Description
`OFF`	No logging at all (not recommended)
`FATAL`	Only fatal errors
`ERROR`	Shows FATAL and other (recoverable) errors
`WARN`	Add warnings
`INFO`	Add runtime information
`DEBUG`	Provide debug information on your jobs
`TRACE`	Trace your jobs (more verbose debug logs)
`ALL`	Everything that PySpark can spit, it will spit

Complex to Achieve Consistency



1

Desired properties of Big Data systems

Learning Outcomes

By the end of this session, you should be able to:

- LO#4: Distinguish the desired properties of a Big Data system

Robustness and fault tolerance

- Systems need to behave correctly despite machines going down, duplicated data, concurrency etc.

- It is imperative for systems to be *human-fault tolerant*. Someone will make a mistake such as deploying incorrect code that corrupts values in a database

Low latency reads and updates

- Most applications require reads to be satisfied with very low latency.
- Update latency varies. In many cases, a few hours is fine. It is important to remember that you design for low update latency when you really need it
- You need to be able to achieve low latency reads and updates without compromising the robustness of the system

Scalability

Scalability is the ability to maintain system performance in the face of increasing data or load by adding resources to the system

Extensibility

- You don’t want to have to reinvent the wheel each time you add a related feature or make a change to how your system works. Extensible features allow functionalities to be added with a minimal development cost

- Extensible systems make it easy to do large-scale migrations (from an old format into a new one)

Ad hoc queries

- Being able to do ad hoc queries on your data is extremely important. Being able to mine a dataset arbitrarily gives opportunities for business optimization and new applications

- You can’t discover interesting things unless you can ask arbitrary questions of it

Minimal maintenance

- Maintenance is the work required to keep a system running smoothly. This includes anticipating when to add machines to scale, keeping processes up and running, and debugging anything that goes wrong in production.

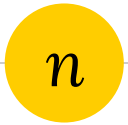
- Choose components that have as little implementation complexity as possible. Rely on components that have simple mechanisms underlying them.

Debuggability

- A Big Data system must provide the information necessary to debug the system when things go wrong.
- The key is to be able to trace, for each value in the system, exactly what caused it to have that value.

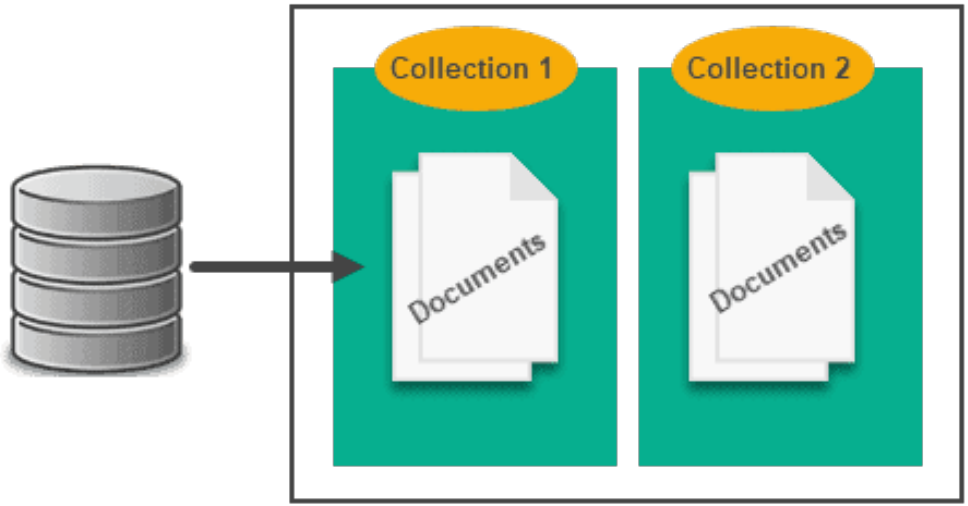
[Practice #02]

- P01. Think of a context in which Big Data could be applied. In your view, how would you rank the desired Big Data properties from the most important to the least? Share your reasons.
- P02. What is the difference between Robustness and Fault-Tolerance?
- P03. With your words, explain why we need Big Data technologies.



NoSQL Databases

Document databases



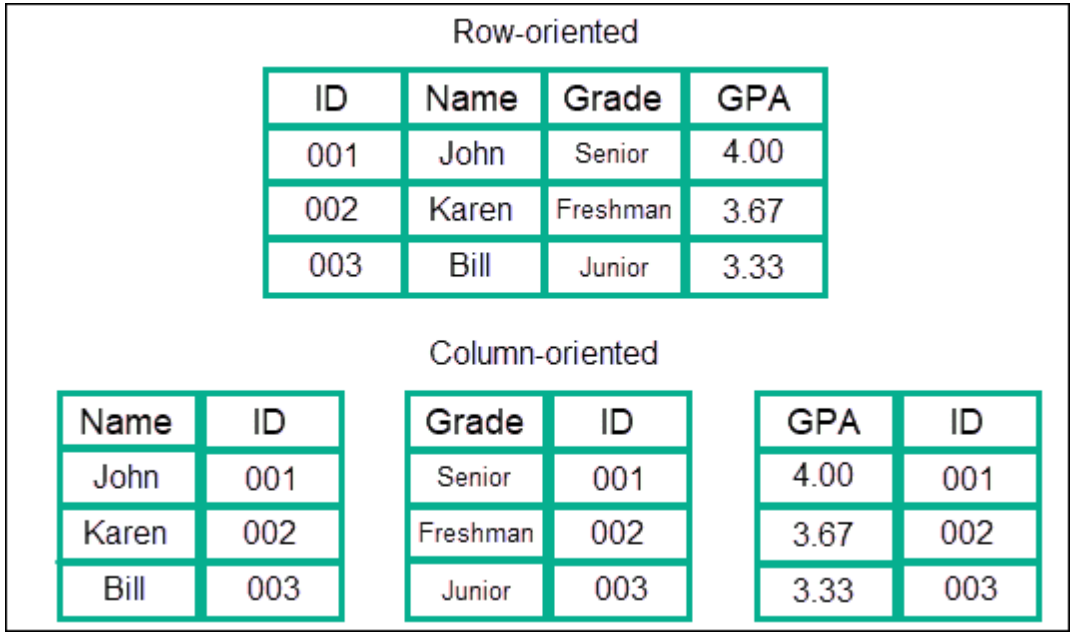
A document database is a type of NoSQL database that consists of sets of key-value pairs stored into a document. These documents are basic units of data which you can also group into collections (databases) based on their functionality.

Document databases: examples

- MongoDB
- CouchDB
- Elasticsearch

Wide-Column databases

In Wide-Column databases data is stored and grouped into separately stored columns instead of rows. Such databases organize information into columns that function similarly to tables in relational databases.



Wide-Column databases: examples

- OrientDB
- RedisGraph
- Neo4J

Graph databases

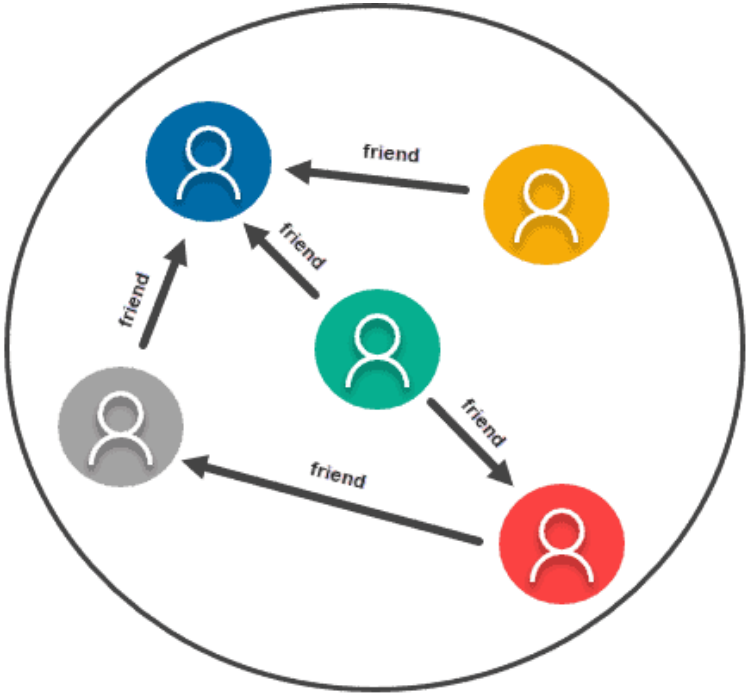
Graph databases use flexible graphical representation to manage data.

These graphs consist of two elements:

1.Nodes (for storing data entities)

2.Edges (for storing the relationship between entities)

These relationships between entities allow data in the store to be linked together directly and, in many cases, retrieved with one operation. Nodes and edges have defined properties, and by using these properties, you can query data easily.



Graph databases: examples

- MongoDB
- CouchDB
- Elasticsearch