

# Infrastructure for Data Analytics and Machine Learning

## Part 2 - Large-scale batch processing

Paweł Wiejacha  
RTB House

# Plan for this lecture

- What is and why we need batch processing?
- How do large-scale data processing engines work?
- How can we use them effectively?

# What is batch processing?

**Batch processing** - processing large volumes of data at once (in batches collected over some time)

**batch** (n.) - a quantity required for or produced as the result of one operation <sup>[1]</sup>

- *mixed a batch of cement; a batch of cookies*

**Stream processing** - processing stream of events as they are produced continuously piece-by-piece in (more or less) real-time

<sup>1</sup> - <https://www.thefreedictionary.com/batch>

# Use cases - batch processing

- *training* machine learning models
  - book and product recommendations
  - image classification, object recognition
- computing user-tailored offers (e.g. credit cards offers)
- processing international money transfers
- processing call records to create monthly billing invoices
- various background processes
  - compressing/re-encoding video
  - performing speech-recognition to add subtitles
- data mining

# Use cases - stream processing

- fraud detection (must be real-time to prevent detected frauds)
- real-time stock trading
- up-to-the-minute retail inventory management
- log and status monitoring, anomaly detection
- edge analytics (IoT, smart cities, manufacturing)
- load controlling - adjusting service level to current number of requests

# When to choose batch processing?

Batch processing works well when:

- you don't need **real-time** analytics results
- it's more important to process **larger volumes** of data
  - e.g. **wider context** is needed
    - full user purchase history instead of last order event
    - joining and aggregating data from different sources
- **more computation** is needed than stream processing pipeline can handle
  - generally (even in real life) batching is faster because you can reduce various overheads
  - since there is no "real-time" requirement, there's more time and flexibility, e.g.
    - we can store intermediate results on a slower medium if they do not fit in the memory
    - we can perform more precise computations (e.g. increase number of epochs in an iterative algorithm)

# When to choose batch processing? (2)

This is not black or white situation, we can:

- process and **prepare data** as much as possible during stream processing to simplify later batch processing
  - compute aggregates
  - partially sort data
  - perform partial joins
  - shard or partition data
- solve some of our (sub-)problems using a dedicated database (e.g. BigQuery, ClickHouse) instead of writing Map/Reduce jobs

# Example - computing recommendations

Computing product recommendations is a good example use case for batch processing.

Imagine you have to create recommendation model for a platform

- with 2 000 000 000 **products**
- using 355 000 000 000 **events**:
  - 2022-11-18 14:00:12.241 user <XYZ> was searching for a product <ABC>
  - 2022-11-18 14:00:12.255 user <ZYX> added product <CBA> to basket

Assuming we only use 8 bytes for product and user identifiers, 4 bytes for an event timestamp, and one byte for event type, that's 6.7 TiB of efficiently written **essential input** data.

- in real world, to extract that essential data you need to process 300-600 TiB of data first
- reading 600 TiB from a single hard drive would take about 36 days. Just to read input data.

To compute recommendation model we need to: sort and group events, compute sparse  $\mathbb{R}^{2*10^9 \times 2*10^9}$  Item-Item matrix, factorize it, fine tune recommendations on GPUs using various machine learning techniques, create ANN index, and store recommendations.



# Data processing engines

# How to approach such task?

There are multiple ways, for example:

1. Creating custom solution from the scratch
2. Using existing Big Data frameworks:
  - Hadoop Map/Reduce, Apache Hive, Apache Spark, Apache Flink, Presto, BigQuery ...
3. Mixing both approaches

# Apache Spark - prerequisites

Before we can use Apache Spark for large scale batch processing you need at least:

- **Computing cluster** - machines with reasonable amount of RAM and CPUs
- **Distributed file system** or object storage - that can store input, output and maybe intermediate data
- **Resource Manager/Scheduler** - that manages our computing cluster

# Apache Spark - prerequisites (2)

The most popular setup is:

- using **HDFS** (Hadoop Distributed File System) as a distributed storage
- HDFS data nodes that store the data blocks are also used as **computing nodes**
  - data nodes don't need a lot of RAM nor CPU cores, but they need network card, motherboard, PSU, ...
  - also performing computations close to data can be very efficient
    - no need to send data over network, OS page cache can be used for frequently accessed files
- using Apache Hadoop **YARN** as a Scheduler and Resource Manager
  - Instead of YARN, Kubernetes can be used as a cluster manager, but support for that setting is not very mature

# Apache Spark - prerequisites (3)

Alternative approach is to use **managed Hadoop in the cloud** (e.g. Google Dataproc, Amazon EMR):

- either using standard Spark + HDFS + YARN combination
- or without using HDFS - AWS S3 or Google Cloud Storage can be used as a distributed storage

# Apache Spark - overview

Spark (Java/Scala) application consists of:

- single **driver** program that orchestrates and executes parallel operations
- those operations are performed by multiple **executors**

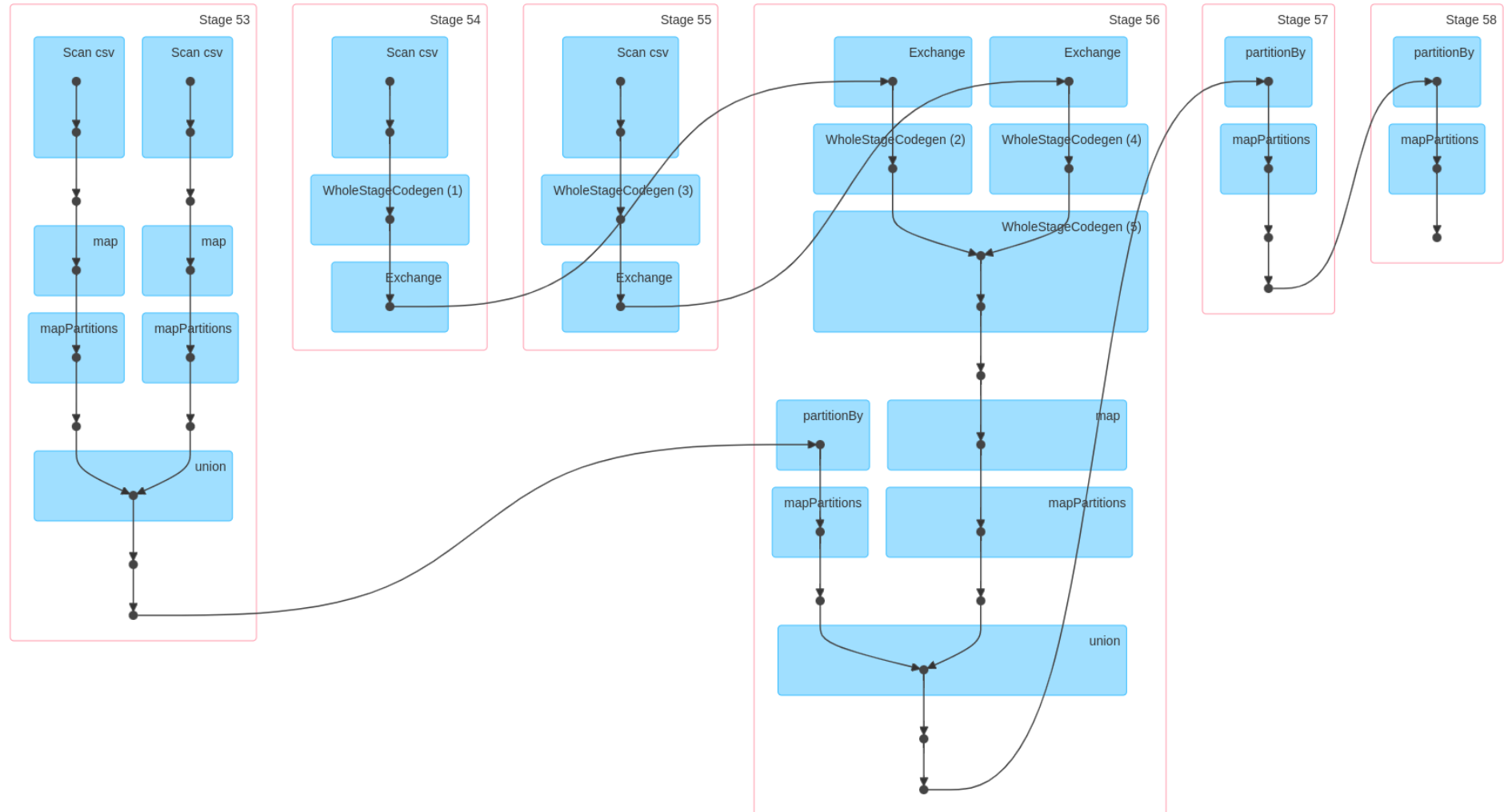
The main abstractions are:

- **Resilient Distributed Dataset (RDD)** - collection of elements partitioned across cluster nodes
  - e.g. having 100 000 files on HDFS (each with multiple records) can result in a RDD with 100 partitions
  - RDDs can be operated in parallel
  - RDDs can be transformed, persisted (to disk or to memory), and recomputed in case of a node failure
- **Broadcast variables** - shared, read-only variables available on all executors

# RDDs

- **transformations** - operations on an RDD that are performed lazily, e.g.
  - `map()`, `filter()`, `groupByKey()`, `reduceByKey()`, `join()`, `union()`, `sortByKey()`
- **actions** - operations on RDD that are performed eagerly and trigger computation of DAGs composed from lazy transformations, e.g.:
  - e.g. `count()`, `collect()`, `reduce()`, `saveAsXYZ()`

# Computations DAGs





# Apache Spark API - example program (1)

Example: E-commerce order logs

## Input

Multiple files on the HDFS inside `/warehouse/purchase_logs/`, e.g.:

```
/warehouse/order_logs/log-1652658664132_40_hXMzbDepoC.csv  
/warehouse/order_logs/log-1652658665521_70_aXZzbDCEPQ.csv  
...  
/warehouse/order_logs/log-1652658278519_13_xiTNJreBBT.csv
```

Each file contains multiple entries with information about products bought by users, e.g.:

```
# <timestamp>           <country>  <customer_id>  <product_id>  <product_price>  
2022-01-23 17:31:00.131, Poland,    User123,      ProductABC,    129.99  
2022-01-23 17:31:03.217, Brazil,   User456,      ProductDEF,    499.99  
2022-01-23 17:39:03.217, Poland,   User123,      ProductGHI,    450.79  
...
```

Information about the same user (or product) can occur in multiple files.

# Apache Spark API - example program (2)

**Goal:** find top 100 customers from Poland that spend the most money in the store

# Apache Spark API - example program (2)

**Goal:** find top 100 customers from Poland that spend the most money in the store

```
type UserId = String; type Price = Float
case class OrderRow(timestamp: String, country: String, userId: UserId, productId: String, price: Price)

val lines: RDD[String] = sparkContext.textFile("/warehouse/order_logs/log-*.csv")
val parsedLines: RDD[Array[String]] = lines.map(line => line.split(", ?"))

val orderRows: RDD[OrderRow] = parsedLines
    .map(words => OrderRow(words[0], word[1], words[2], word[3], word[3].toFloat))

val ordersFromPoland = orderRows.filter(order => order.country == "Poland")

val userPrices: RDD[(UserId, Price)] = ordersFromPoland.map(order => (order.userId, order.price))
val summedUserPrices: RDD[(UserId, Price)] = userPrices.reduceByKey((price1, price2) => price1 + price2)
val top100Users: Array[(UserId, Price)] = summedUserPrices
    .sortBy({ case (userId, priceSum) => -priceSum })
    .take(100)
```

# Execution of the example program (1)

- DAG of computations will be materialized, then
- for each file, one RDD partition will be created

```
task_00: read_lines(file_00) => partition_00 == stream_of[line1, line2, ...]  
task_01: read_lines(file_01) => partition_01 == stream_of[line1, line2, line3, ...]  
...  
task_99: read_lines(file_99) => partition_99 == stream_of[line1]
```

- every executor will process multiple RDD partitions (perform multiple tasks):

```
executor_00: [task_01, task_17, task_87, task_99]  
executor_01: [task_02, task_05, task_43, task_66]  
...  
executor_19: [task_55, task_77, task_88, task_98]
```

## Execution of the example program (2)

- To execute tasks, Spark Driver has to **serialize closure** of a function that has to be executed remotely and send it to every executor
  - e.g. `fun: Array[String] => OrderRow = { words => OrderRow(words[0], word[1], words[2], word[3], word[3].toFloat) }`
- Then executor merges subsequent transformations and produces **stream** of results with approximately following semantic:

```
lines_stream.  
  .map(line => line.split(", ?"))  
  .map(words => OrderRow(words[0], word[1], words[2], word[3], word[3].toFloat))  
  .filter(order => order.country == "Poland")  
  .map(order => (order.userId, order.price))
```

- If our application called `ordersFromPoland.saveAsTextFile("/warehouse/temp/orders_from_poland")` instead of `reduceByKey(...)`, we could read and write multiple files sequentially in parallel using constant memory

# Execution of the example program - shuffle (1)

- Certain Spark operations trigger a slow and complex event called **shuffle**
- Shuffle is a mechanism for re-distributing and re-grouping data across partitions.
- This usually involves **copying data between executors** (all-to-all),
  - and if shuffled data is too large to fit in RAM, writing intermediate results to disk
  - it also requires serializing data and network I/O

## Execution of the example program - shuffle (2)

- Every task operates on a single partition and is executed by one executor, so to perform:

```
userPrices // : RDD[(UserId, Price)]  
  .reduceByKey((price1, price2) => price1 + price2)
```

- which corresponds to:

```
for user in userPrices.keys():  
  group: List[Price] = { find all records where record.key == user }  
  aggregated_value = group.reduce((price1, price2) => price1 + price2)  
  output(user, aggregated_value)
```

- we need to **re-group** all records with given key so that they are located within **the same partition** and can be processed efficiently by a single executor

# Execution of the example program - shuffle (3)

There are at least two ways to perform shuffle:

- **sort based** (used by Hadoop Map/Reduce and Spark)
  - (on-disk) sort records by key using multiple files on a HDFS
    - this complex parallel external sort
  - process sorted results in parallel so that all adjacent records with given key are processed by a single Reduce Task
- **hash-based** (used by Spark)
  - hash keys and write records to resulting buckets (`hash(key) % num_buckets`)
    - spilling intermediate data on disk, when they don't fit into memory
  - process each bucket by one executor

We can reduce work by performing map-side reduces.



# Spark SQL, DataFrames and Datasets

- RDD is a part of the low-level Spark API
- Spark also provides less flexible but (in some cases) more performant APIs:
  - Dataset interface
  - Pandas DataFrame-like interface
  - Spark SQL interface
- **Extra information** about data and computations allows for **additional optimizations**
  - e.g. reading only a subset of columns, choosing the best join method, reordering of operations
- There's also PySpark interface, ML and Graph processing on top of Spark

```
data
  .filter(col("country").equalTo("Mexico"))
  .groupBy(col("age"))
  .count()
```

```
spark.sql("""
SELECT age, count(1)
FROM table_from_rdd
WHERE country = 'Mexico'
GROUP BY age
""")
```

# Useful techniques in batch processing

# Partitioning and clustering

- organizing files to allow common tasks to read only necessary subsets of data
  - by type, by date, by region, vendor or category
- storing the same events multiple times in multiple clusters

```
/warehouse/products/...  
/warehouse/user-reviews/...  
/warehouse/browsing_events/...  
/warehouse/browsing_events/2022-05-01/...  
/warehouse/browsing_events/2022-05-02/...  
/warehouse/browsing_events/2022-05-03/...  
/warehouse/browsing_events/2022-05-04/region=us/...  
/warehouse/browsing_events/2022-05-04/region=eu/log-1652658664132_40_hXMzbDepoC.avro.snappy  
/warehouse/browsing_events/2022-05-04/region=eu/log-1652658665521_70_aXZzbDCEPQ.avro.snappy  
/warehouse/browsing_events/2022-05-04/region=eu/log-1652658278519_13_xiTNJreBBT.avro.snappy  
/warehouse/browsing_events/2022-05-04/region=eu/...  
/warehouse/subscriptions/...  
/warehouse/purchases/...
```

# Sorting and merging

Having sorted data can be useful

- we can use binary search, skip-list-like or sampled indices
- we **merge multiple sorted streams** - very useful operation
  - joining two "tables" using common key
  - processing multiple shards
  - merge sorting
- we can reduce by key easily when data is sorted
- sorted data compresses better

```
# SELECT ... FROM users LEFT JOIN purchases ON (users.id == purchases.user_id)
```

```
# records sorted by User.id:
```

```
users-shard1.avro: U1, U3, ...
```

```
users-shard2.avro: U4, U6, ...
```

```
users-shard3.avro: U2, U4, U7, ...
```

```
# records sorted by Purchase.user_id
```

```
purchases-shard1.avro: P.U1, P.U6, P.U7, ...
```

```
purchases-shard2.avro: P.U3, P.U4, ...
```

# Sharding

**Sharding** - partitioning data or computations horizontally so that each data shard is stored or handled by a different server, process, thread etc.

- similar to partitioning
- but it's rather related to spreading load/computations/responsibility evenly
  - compared to reducing amount of data read

```
def process_single_event(event):
    computed_result = compute(event)
    shard_id = hash(event.user_id) % NUM_SHARDS
    write_to(computed_result, f"/temp/stage4/shard.{worker_id}.{shard_id}.avro")

input_files
    .flatMap(read_events)
    .map(process_single_event) # process multiple files in parallel

get_paths(f"/temp/stage4/shard.*.*.avro")
    .map(sort_file) # also parallel, sorting small files in memory

range(NUM_SHARDS).parallelize() # process every shard in parallel
    .map({ shard_id =>
        get_paths(f"/temp/stage5/sorted-shard.{shard_id}.avro")
            .mergeSortedStreams() # lazy (iterator-based) merging
            .map(process_full_user_history)
    })
```

```
def process_full_user_history(event):
    while event.user_id == previous_user_id:
        ...
```

# Sampling

- For many tasks and queries using sample of data is enough
  - sometimes we are not able store all data
  - sometimes processing all data is too slow and increasing amount of processed data gives diminishing gains
- Sometimes we need samples from different distributions:
  - ```
if random() < 0.1:  
    write_sample(event)
```
  - ```
if hash(event.user_id) % 1000 < 100:  
    write_sample(event)
```
- sometimes can keep all important events (e.g. conversions) and sample of ordinary events (home page visits)
- nothing prevents us from having sampled and whole dataset at the same time:

```
/warehouse/browsing_events/2022-05-04/sample=1/.. # 10 GB  
/warehouse/browsing_events/2022-05-04/sample=10/.. # 90 GB  
/warehouse/browsing_events/2022-05-04/sample=100/.. # 900 GB
```

# Reducing amount of stored and processed information

- sometimes it's worth re-examine stored data and decide if everything we write is useful
  - maybe some fields or columns are no longer needed or redundant?
- during processing **the faster we reduce every record to bare minimum the better**
  - this reduces amount of data that need to be serialized, send over network, shuffled, sorted, spilled to disks, kept in RAM, etc.
- also parsing and casting done early can reduce amount of processed data (e.g. converting timestamp strings to `int64`)

# Pushing projections up

In some cases it may be better to **perform redundant computations** to reduce amount of processed data

```
val userVisits: RDD[(UserId, (VisitStartDate, List[UserAction]))] = ... # 200 TB of data
val lastUserVisits = userVisits
    .reduceByKey(takeMoreRecentVisit) # for every user, find his latest visit

lastUserVisits
    .mapValues(visit => computeFraudScore(visit)) # detect suspicious users using heavy ML algorithm
    .filterValues(fraudScore => fraudScore > 0.8)
    .saveAsAvro("suspiciousUsers.avro")
```

and

```
val userVisits: RDD[(UserId, (VisitStartDate, List[UserAction]))] = ... # 200 TB of data
val suspiciousVisits : RDD[(UserId, (VisitStartDate, Float))] = userVisits
    .mapValues(visit => (visit.date, computeFraudScore(visit)))
    .filterValues((_, fraudScore) => fraudScore > 0.8)

suspiciousVisits # 5 GB of data
    .reduceByKey(takeMoreRecentVisit) # for every user, find his latest visit
    .saveAsAvro("suspiciousUsers.avro")
```



# Denormalization

Sometimes to avoid complex joins it's worth to consider keeping:

- **raw events**
- (redundant) big **complex snapshots** (e.g. snapshot of all records related to user XYZ)
  - those can be stored
    - periodically (e.g. every day at midnight)
    - or even with every request
- optionally, redundant **periodical deltas** containing consolidated information that changed since last snapshot

In our example, to materialize user profile at requested point of time

- instead of reading all records about user XYZ stored in hundreds of files created during last 2 years
- we could use the latest snapshot of user XYZ history + few deltas + few raw events

Storing redundant materialized snapshots will consume **huge amounts of storage space**, but it's worth considering as it might speed up **simplify** batch processing.

# Caching and precomputing

- caching **can introduce subtle bugs**, but it's worth considering caching, especially if
  - performed computation is slow or cached computation result is smaller than data needed to compute that result
- we can cache things in memory or store intermediate results on to disk to read it multiple times
- we can also cache results computed by previous batch processing jobs, for example:
  - if every day we process rolling window consisting of data from last  $n$  days
  - we can change our algorithm to reuse previous results, i.e.
  - remove data from  $n + 1$  day ago and use changes from the newest day

# Tiering, prefetching and buffering

**Tiering** - arranging something in tiers (layers)

We should keep in mind speeds, properties and capacities of various memory types, e.g.

- keep frequently accessed data (or data that need random access) on SSD or in RAM
- using fact that we can read from HDD faster if we read large continuous chunks of data
  - so if we expect data to be read in the near future we can prefetch it the background or keep constant read-ahead buffer
- sometimes manually juggling data paired with sharding can give spectacular effects

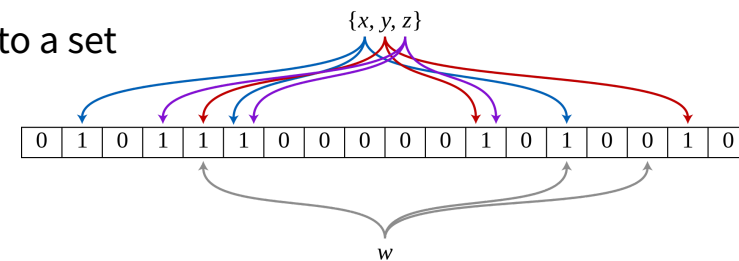
# Bloom filters and sketches (1)

**Bloom filter** is a compact probabilistic data structure that can be used to test if an element is a member of a set.

- no false negatives – if tested element belongs to a set, bloom filter will always return true
- false positives are possible – bloom can return true for element that does not belong to a set

General idea:

- Bloom filter is a bit array of  $m$  bits, all set to 0.
- when we add an element to a set, we hash it using  $k$  different hash functions  $h_i : Universe \rightarrow \{1, \dots, m\}$
- we turn on every bit indicated by  $h_i(element) : i \in \{1..k\}$
- to test whether element belongs to a set we check that every  $bit\_array[h_i(element)] : i \in \{1..k\}$  bits are 1



We can choose  $n, k, m$  to get very compact filter with desired False Positive Rate (FPR). A Bloom filter with a 1% FPR and an optimal  $k$  needs only ~9.6 bits per element (which can be anything identifier, string, blob of bytes, image).

Examples:

- checking bloom filter before trying to retrieve from slower medium or external source
- using bloom filter for broadcast joins

# Bloom filters and sketches (2)

**Count-min sketch** - probabilistic data structure that serves as a frequency table of events.

- hash functions are used to map events to frequencies
- uses only sub-linear space, but can over count events due to hash collisions

We can use  $d$  hash functions  $h_i : Universe \rightarrow \{1, \dots, w\}$ , increase  $d \times w$  counter matrix accordingly and estimate each element frequency as  $\min \{C[1][h_1(elem)], \dots, C[d][h_d(elem)]\}$

```

insert(elem1) -> hash_1(elem1) % w = 2 --> [ ] [ ] [ +1 ] [ ] [ ] [ ] [ ]
                  hash_2(elem1) % w = 5 --> [ ] [ ] [ ] [ ] [ +1 ] [ ] [ ]
                  hash_d(elem1) % w = 3 --> [ ] [ ] [ ] [ ] [ ] [ +1 ] [ ]
                                                    d=3, w=7
    
```

```

insert(elem2) -> hash_1(elem2) % w = 1 --> [ ] [ +1 ] [ 1 ] [ ] [ ] [ ] [ ]
                  hash_2(elem2) % w = 6 --> [ ] [ ] [ ] [ ] [ ] [ ] [ 1 ]
                  hash_d(elem2) % w = 3 --> [ ] [ ] [ ] [ ] [ 1+1 ] [ ] [ +1 ]
    
```

```

insert(elem1) -> hash_1(elem1) % w = 2 --> [ ] [ 1 ] [ 1+1 ] [ ] [ ] [ ] [ ]
                  hash_2(elem1) % w = 5 --> [ ] [ ] [ ] [ ] [ ] [ ] [ 1+1 ]
                  hash_d(elem1) % w = 3 --> [ ] [ ] [ ] [ ] [ 2+1 ] [ ] [ 1 ]
    
```

```
count(elem1) = min(2,3,2) = 2
```

# Bloom filters and sketches (3)

**HyperLogLog** - probabilistic algorithm for approximating the number of distinct elements in a multiset (i.e. count-distinct problem),

- uses sub-linear memory, at the cost of obtaining only an approximation of the cardinality

The basic idea is to hash elements and use the observation that we can estimate cardinality of a multiset of *uniformly distributed random numbers* by calculating **maximum number of leading zeros** in binary representation of each number in the set.

```
elem1 -> hash(elem1) -> . . . 0 0 0 # 3 leading zeros => P(3 zeros) = 1 / 2^3 = 1 / 8
elem2 -> hash(elem2) -> . . . 0 0 1
elem3 -> hash(elem3) -> . . . 0 1 0
elem4 -> hash(elem4) -> . . . 0 1 1
elem5 -> hash(elem5) -> . . . 1 0 0
elem6 -> hash(elem6) -> . . . 1 0 1
elem7 -> hash(elem7) -> . . . 1 1 0
elem8 -> hash(elem8) -> . . . 1 1 1
```

# Effective serialization and file formats (1)

If data elements we process have at least some basic structure we can store the elements either in **columnar format** or **row format**.

Row format:

```
{ name: Jack, age: 23, city: Warsaw }  
{ name: Jill, age: 22, city: NULL }  
{ name: Bill, age: 21, city: Berlin }  
{ name: John, age: 24, city: Berlin }
```

Columnar format:

```
{ name: [Jack, Jill, Bill, John] }  
{ age:  [23, 22, 21, 24] }  
{ city: [Warsaw, NULL, Berlin, Berlin] }
```

# Effective serialization and file formats (2)

**Columnar format** is useful

- when we process only subset of columns
  - we don't have to read whole file, nor deserialize entire object
  - especially when we have very selective predicate
    - e.g. `dataset.filter(user => user.firstName == 'Gościław').map(...)`
  - this is common case for query-like workloads
- because it makes compression easier (similar values are close to each other)
- and allows for vectorization and efficient processing (e.g. processing numbers using AVX instructions)



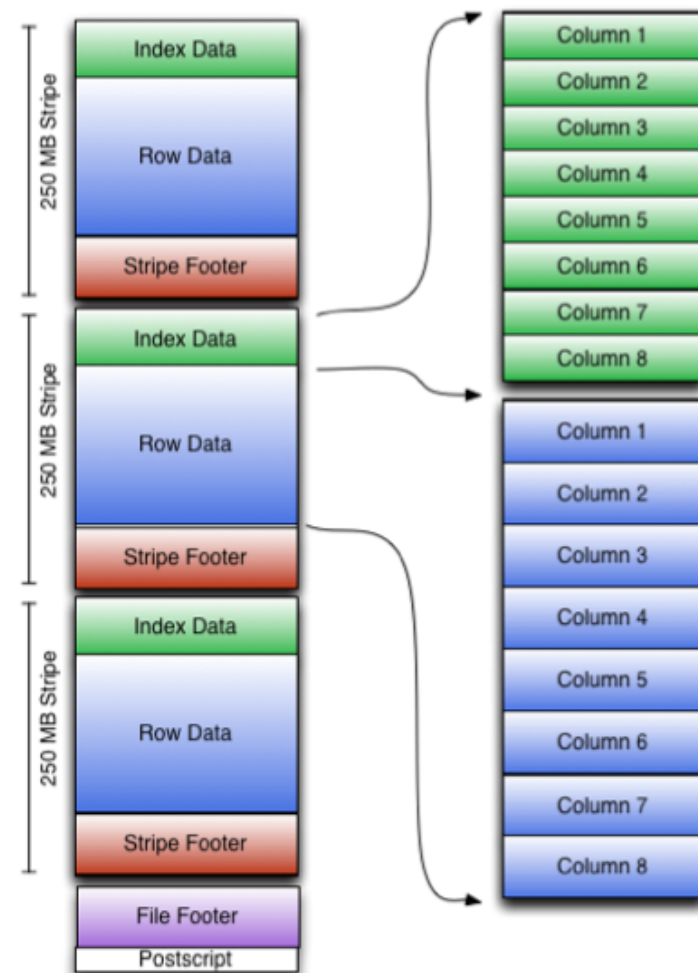
# Effective serialization and file formats (3)

**Row format** is useful

- when we process all fields/columns sequentially
  - in columnar format, accessing  $n$  columns would require  $n$  disk accesses
- easier to serialize (no need to buffer multiple rows)

**Row-Columnar** format:

- we can group rows, and for each group use columnar format
  - e.g. Apache ORC (Optimized Row Columnar)



# Effective serialization and file formats (4)

Space-efficient and/or fast serialization is very important.

- for data serialized **on disk** we might prefer space-efficient serialization, e.g.
  - compression
  - variable length zig-zag integer encoding
  - encoding enums as bytes, enum sets as bit sets
- for data serialized to be transferred **over network** we might prefer fast serialization, e.g.
  - serializing integers in native byte order
  - keeping padding bytes so structures can be memory mapped

# Map-side/Broadcast joins

```
users // large collection
  .join(purchases) // smaller collection
  .keyBy(user => user.id)
  .map({ case (user, order) => (user.email, order.orderedProducts) })
```

The standard way to perform this join is to use slow sort- of hash-based shuffle join.

But we can also broadcast smaller collection to all workers and perform simple map task.

```
val purchaseMap : Broadcast[HashMap[UserId, List[Product]]] = spark.broadcast(purchases.collect().asMap)
users.map(user => (user.email, purchaseMap.get(user.id, List.empty)))
```

# Vertical scaling

If we are able reduce amount of processed data or transform our problem into set of independent tasks we can consider **vertical scaling**.

For example, performing computation using one or few machines with 4TB RAM and 256 CPU threads and fast NVMe SDD RAID-0 disk array can reduce total computation time drastically.

When we perform computations in a single address space on a single machine:

- we don't need to **serialize** data, send it over network nor write to disk
- we can **vectorize** our operations and use native implementations
- we can use efficient data structures and algorithms
- we can use CPU and OS **caches** effectively
- we can reduce various **overheads** related to distributed computing (e.g. no need to use distributed task scheduler)

# Summary

We have discussed following topics:

- What is and when we need batch processing
- Batch processing using Apache Spark
- Useful techniques in batch processing