Apache Spark

Under the Hood

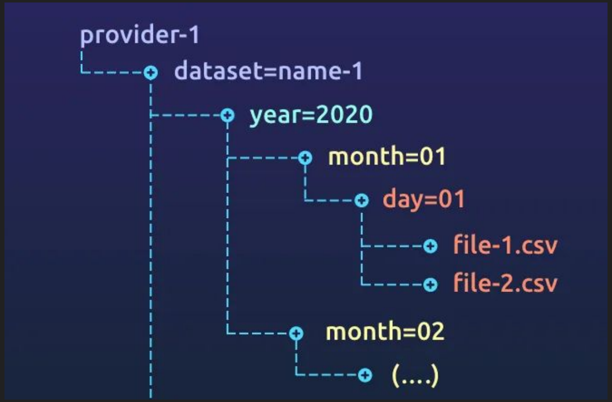
Intro to Data Engineering

- Building systems to enable the collection and usage of data
- Enable subsequent analysis and data science
    - Including training and deployment of ML models
- Data comes from various sources and in various formats
    - Clean up incoming data
    - Enrich with other data sources
    - Aggregate to generate reports and statistics

- 💡 Can’t we do all this in Postgres?
    - How big a DB would we need to process 50TB?
    - How much time would it take? Should a user have to wait that long?
    - Do we need this DB running 24x7?

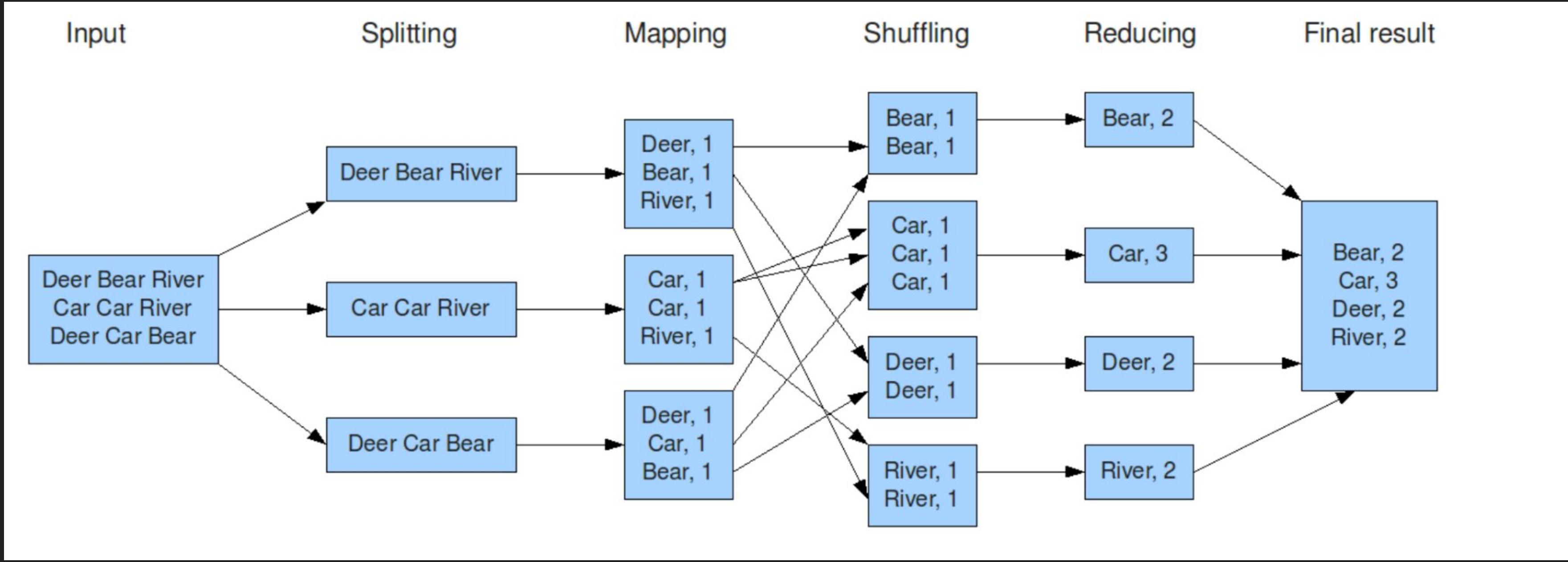
Good DE life choices

- Break large pipelines into smaller jobs
  - Reusable in other places
  - Quickly build from previous work, instead of starting from scratch
  - Easier recovery from failures
- Keep datasets immutable
  - Maintain history
  - Reproducible jobs
  - Easier to debug
- Pre-compute expensive datasets
  - Batch process data, asynchronously
  - Serve to users in real-time



Source: https://sahaj.ai/data-storage-patterns-versioning-and-partitions/

Quick intro to MapReduce



Source: https://hci.stanford.edu/courses/cs448g/a2/files/map_reduce_tutorial.pdf

MapReduce Code example

```java
public void map(Object key, Text value, Context context) {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
}

public void reduce(Text key, Iterable<IntWritable> values, Context context) {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

Introducing Playflix™ 🎬

- Own datasets
  - List of movies
  - User watch history
  - User like history
  - User’s past recommendations
- Other datasets
  - User’s web searches
  - User’s like history on FB
  - User’s Watch history on YT

- Challenge: Generate new movie recommendations for all users

Limitations in MapReduce

- Developer Experience Limitations
  - Only one map and one reduce operation per job
  - Forces a linear flow
  - Have to write each M/R operation result to disk

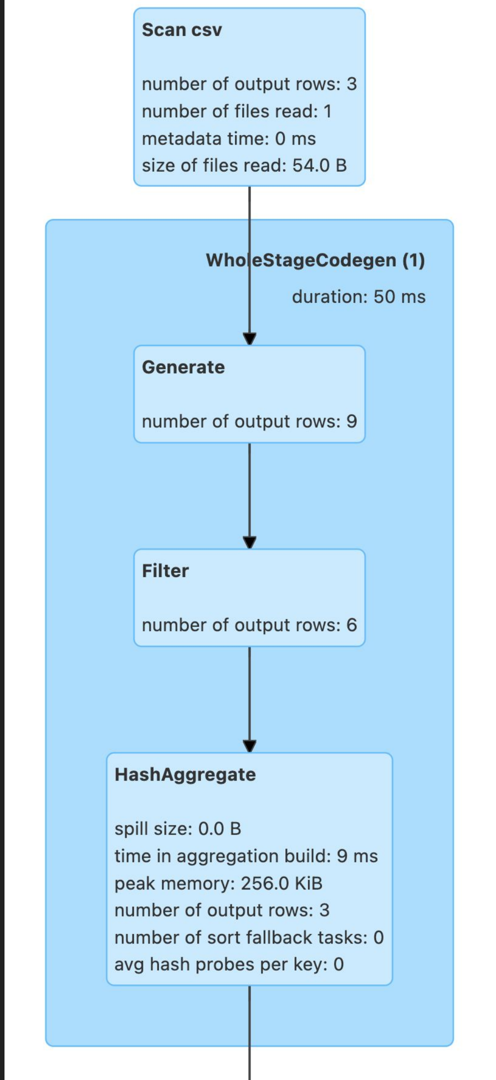
- Performance Limitations
  - Only one map and one reduce operation per job
  - Eager evaluation, leading to waste of compute and storage
  - Need to store intermediate datasets on disk
  - Repeatedly loading data to/from disk leads to a bottleneck

Guiding principles in Spark Jobs

- Lazy evaluation
- All operations happen in memory
- Cache and reuse intermediate data
- Fully SQL compliant
    - Optimize all the things - SQL query, disk read, calculations, data transfers, disk writes

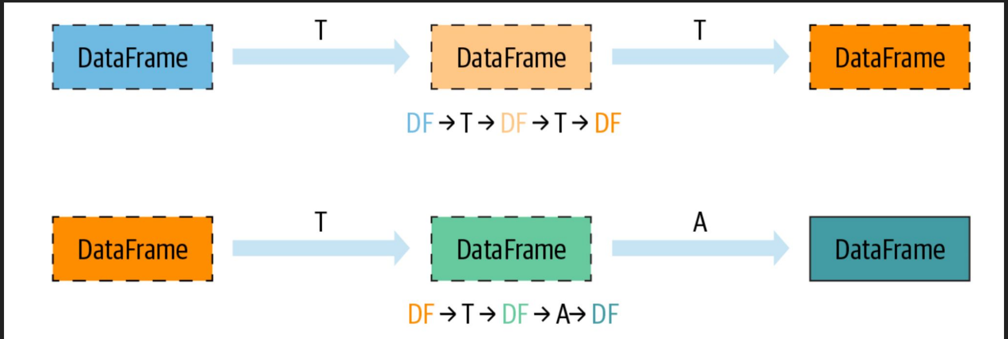
Spark DAGs

- Key to achieving lazy evaluation
- Everything within a block is evaluated in one shot
- 1 Read, 1 Calculation, 1 Write
- No matter how the code is written
- How is this achieved?
  - Lazy evaluation: Actions and transformations
  - Regardless of Code: SQL compilers 🤍



Actions & Transformations

- Transform a Spark DataFrame into a new DataFrame without altering the original data
    - `select()`, `filter()`, `join()`, `withColumn()`
- All Transformations are evaluated lazily
- An Action triggers the evaluation of all pending transformations
    - `write()`, `count()`, `collect()`, `show()`

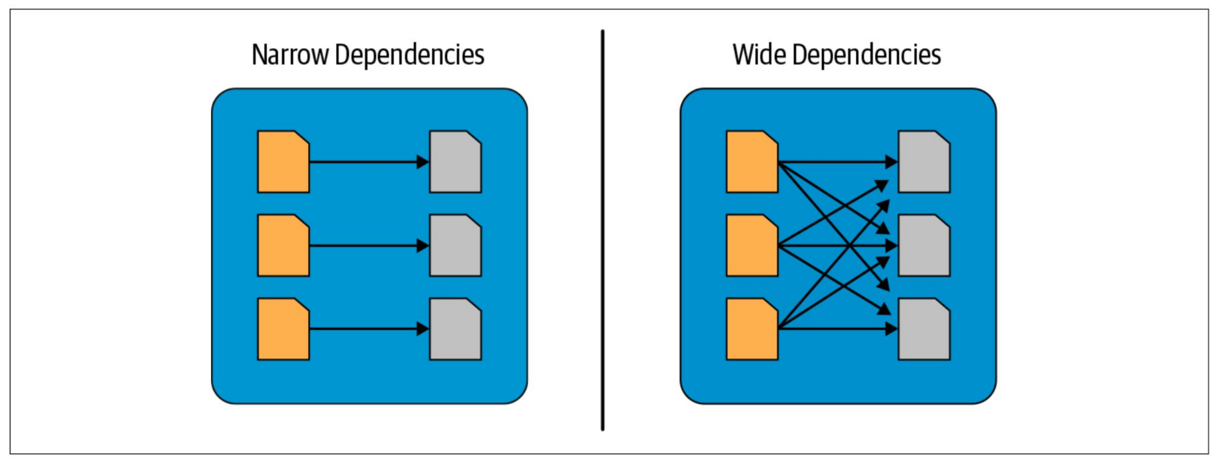


Source: https://pages.databricks.com/rs/094-YMS-629/images/LearningSpark2.0.pdf

Perks of Using Transformations

- Lazy evaluation allows Spark to optimize queries
  - Push down predicate filtering
  - Reorder transformations within a stage
- Dataframe Immutability + Lineage
  - Fault tolerance
  - Reproduce lost data by replaying history
  - Only needs to recalculate lost data

Narrow vs. Wide Transformations



- Which operations can be clubbed together?
  - Anything which can be done in a single pass through a partition
- All wide transformations result in a data shuffle
  - AVOID DATA SHUFFLES!

Narrow vs. Wide Transformations - An Example

`spark.read.csv("data.csv")`	ACTION
`.filter(!contains($"sentence", lit("Cat")))`	NARROW
`.withColumn("split_sentence", split(col("sentence"), " "))`	NARROW
`.withColumn("exploded_word", explode($"split_sentence"))`	NARROW
`.groupBy("exploded_word")`	WIDE
`.count()`	WIDE
`.orderBy($"count".desc)`	WIDE
`.show()`	ACTION

How does it actually run?

- How does spark actually read and write data?
- Data partitions
- No. of input partitions and output partitions?
  - `dataframe.withColumn("upper", upper($"text")).write("...")`
    - 10 inputs, 10 outputs
  - `dataframe.groupBy("word").count().write("...")`
  - `dataframe.join(dictionary, $"word").write(...)`
    - How many outputs?
- What if the inputs are extremely large?

```
→ data ls -l
Size User     Date Modified Name
 11G dakshin 24 Oct 14:53  0.csv
 42M dakshin 24 Oct 14:25  1.csv
 42M dakshin 24 Oct 14:25  2.csv
 42M dakshin 24 Oct 14:25  3.csv
 42M dakshin 24 Oct 14:25  4.csv
 42M dakshin 24 Oct 14:25  5.csv
 42M dakshin 24 Oct 14:25  6.csv
 42M dakshin 24 Oct 14:25  7.csv
 42M dakshin 24 Oct 14:25  8.csv
 42M dakshin 24 Oct 14:25  9.csv
```

Data skew

- Executors evaluate one output partition at a time
- What if a partition has many more records than the others?
- Ex. Viewers of the most popular movie

Summary Metrics for 663 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.6 min	3.6 min	4.1 min	4.4 min	5.2 min
GC Time	1 s	5 s	7 s	9 s	12 s
Input Size / Records	21.8 MiB / 1513940	22.3 MiB / 1553016	22.5 MiB / 1573170	22.7 MiB / 1594599	29.1 MiB / 2012942
Output Size / Records	49.5 MiB / 18510303	52.9 MiB / 20192836	54.1 MiB / 20763829	55.5 MiB / 21354583	73.7 MiB / 27738137

- Rebalancing partitions can improve the situations
- Forces a data shuffle

Thanks!

Feedback

Email: inbox@dakshin.xyz

More Performance Considerations

Bonus Content

Cluster Setup

- Who orchestrates all this work?
  - Driver nodes and worker nodes
- What if we lose a node?
  - Losing worker nodes
  - Losing driver nodes
- Performance considerations with driver nodes
  - Driver memory
    - Required for joins, grouping, count, collect and more
  - Avoid `collect()` and `show()`!

File Formats

- Splittable File formats
  - .txt, .csv, .parquet, .ndjson
- Not Splittable
  - .json, .csv.gz
- Performance considerations with non-splittable files
  - Each input file becomes its own partition
  - Large files can cause OOM errors
  - Must explicitly repartition (and shuffle) the data

- Row (.csv) formats vs. columnar (.parquet) formats

Thanks!

Feedback

Email: inbox@dakshin.xyz