

# Big Data

---

## Hands on PySpark

# How do we process Big Data?

## Main issues

- Where do we store the data?
- How do we process it?

## Big Data greatly exceeds the size of the typical drives

- Even if a big drive existed, it would be too slow (at least for now)



Year: 1990  
Size: 1.3 GB  
Speed: 4,4 MB/s

5 minutes



Year: 2014  
Size: 1 TB  
Speed: 100 MB/s

3 hours



Year: 2015  
Size: 1 TB  
Speed: 600 MB/s

30 minutes

# The answer: cluster computing



100 hard disks? 18 seconds to read 1TB

# Commodity hardware

You are not tied to expensive, proprietary offerings from a single vendor

You can choose standardized, commonly available hardware from a large range of vendors to build your cluster

Commodity  $\neq$  Low-end!

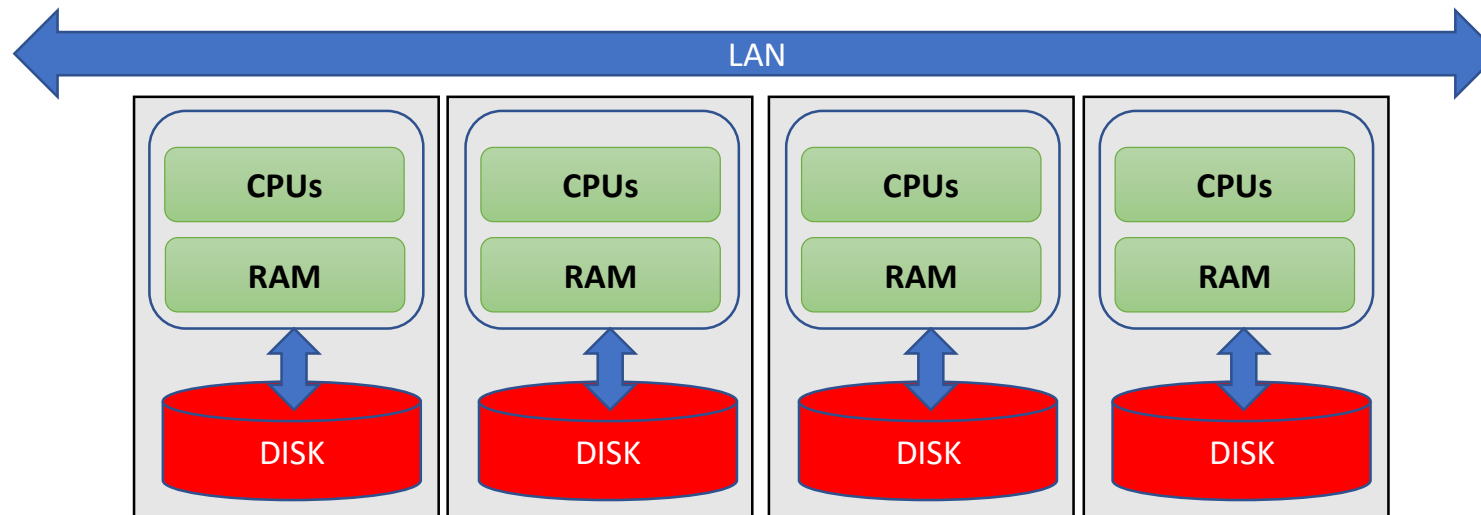
- Cheap components with high failure rate can be a false economy



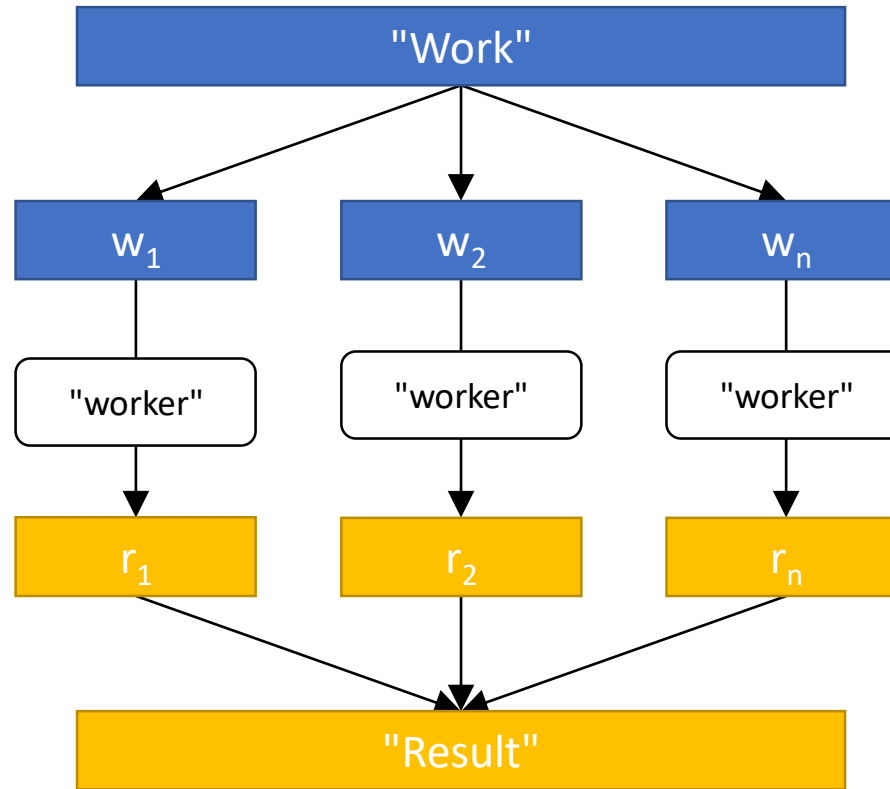
# Cluster Computing Architecture

A computer cluster is a group of linked computers (nodes), working together closely so that in many respects they form a single computer

- Typically connected to each other through fast LAN
- **Every node is a system on its own**, capable of independent operations
  - Unlimited scalability, no vendor lock-in
- Number of nodes in the cluster  $\gg$  Number of CPUs in a node



# Distributed computing: an old idea



Divide



Conquer

# MapReduce

*"MapReduce is a programming model and an associated implementation for processing and generating large data sets.*

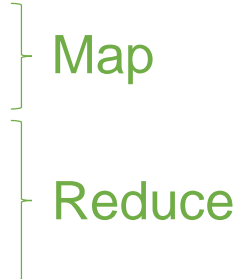
*Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key."*

-- Dean J., Ghemawat S. (Google)

Hadoop MapReduce is an open-source implementation of the MapReduce programming model

# How it works

Take a typical large-data analytical problem

- Iterate over a large number of records
  - Extract something of interest from each
  - Shuffle and sort intermediate results
  - Aggregate intermediate results
  - Generate final output
- 
- Map
- Reduce

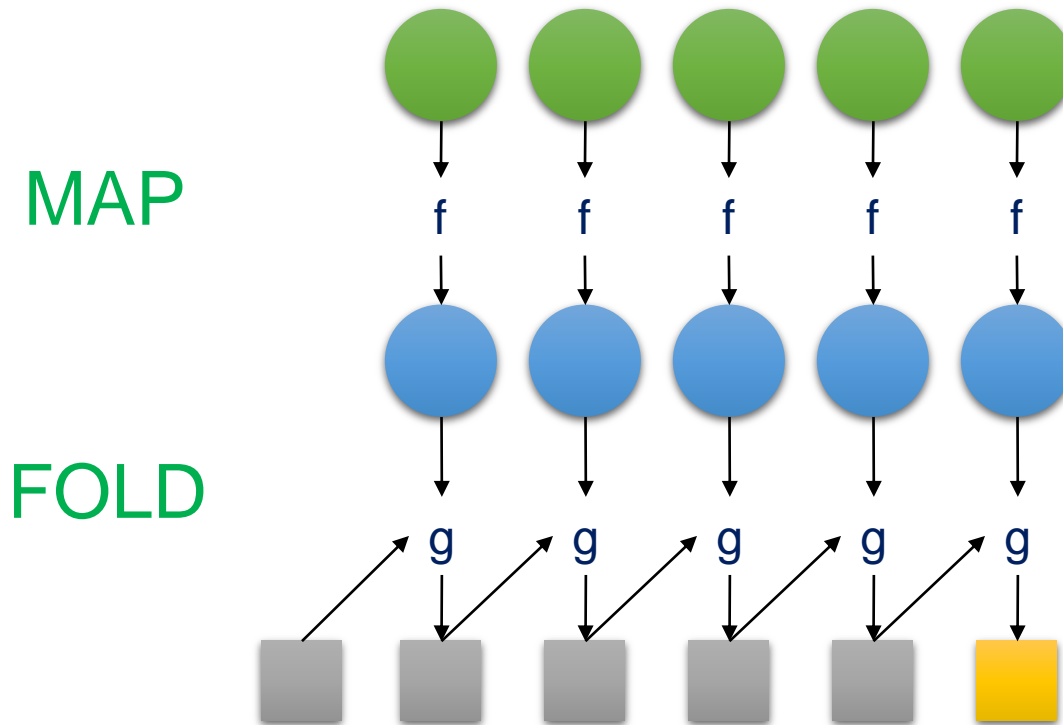
The idea is to provide a functional abstraction for these two operations



# Roots in Functional Programming

MAP takes a function  $f$  and applies it to every element in a list,

FOLD iteratively applies a function  $g$  to aggregate results



# Example of Functional Programming

## Imperative programming

```
a = 0
b = a + 1
```

### Map example

```
names = ['Mary', 'Isla', 'Sam']
name_lengths = []
for i in range(len(names)):
    name_lengths[i] = len(names[i])
```

### Reduce example

```
sentences = [
    'Mary read a story to Sam and Isla.',
    'Isla cuddled Sam.', 'Sam chortled.' ]
sam_count = 0
for sentence in sentences:
    sam_count += sentence.count('Sam')
```

## Functional programming

```
a = 0
b = increment(a)
def increment(a):
    return a + 1;
```

### Map example

```
names = ['Mary', 'Isla', 'Sam']
name_lengths = map(len, names)
```

### Reduce example

```
sentences = [
    'Mary read a story to Sam and Isla.',
    'Isla cuddled Sam.', 'Sam chortled.' ]
sam_count = reduce(
    lambda a, x: a + x.count('Sam'),
    sentences, 0
)
```

# Parallelization of Map and Reduce

The **map operation** (i.e., the application of  $f$  to each item in a list) can be **parallelized in a straightforward manner**, since each functional application happens in isolation

- In a cluster, these operations can be distributed across many different machines

The **reduce operation** has more **restrictions on data locality**

- Elements in the list must be "brought together" before the function  $g$  can be applied

However, many real-world applications do not require  $g$  to be applied to all elements of the list

- If elements in the list can be divided into groups, the fold aggregations can proceed in parallel

# MapReduce program

Basic data structure: **key-value pairs**

- The type of key-value pair can be chosen by the programmer

Programmers specify two functions:

- **map**  $(k1, v1) \rightarrow \text{list}(k2, v2)$
- **reduce**  $(k2, \text{list}(v2)) \rightarrow \text{list}(k3, v3)$ 
  - $(k, v)$  denotes a (key, value) pair
  - Keys do not have to be unique: different pairs can have the same key
  - In text files, each line is treated as a new record;  
the key is the offset of the line within the file (usually irrelevant), the value is the line itself

The execution framework handles everything else!

# MapReduce program

A MapReduce program, referred to as a **job**, consists of:

- Code for Map and Reduce
- Configuration parameters (input/output directories on the underlying distributed file system)

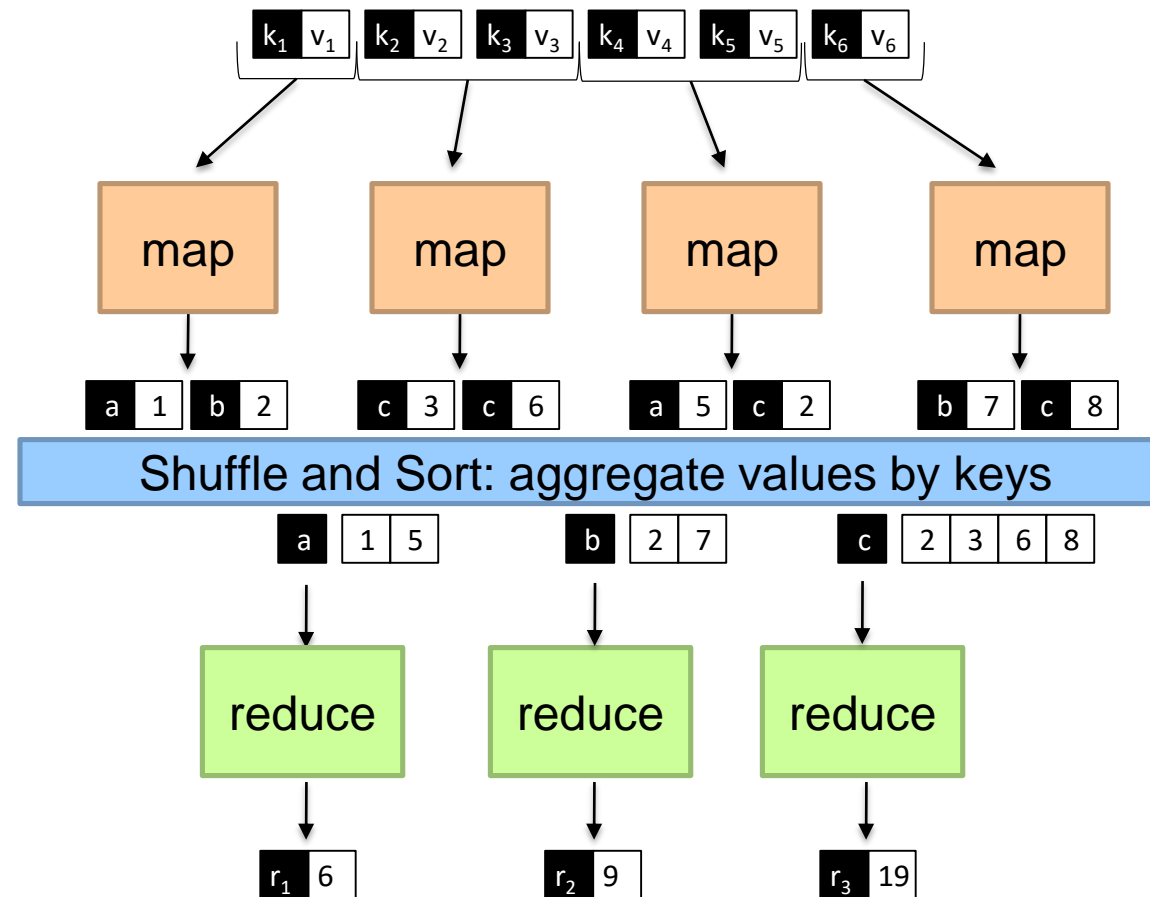
Each MapReduce job is divided by the system into smaller units called **tasks**

- Map tasks
- Reduce tasks

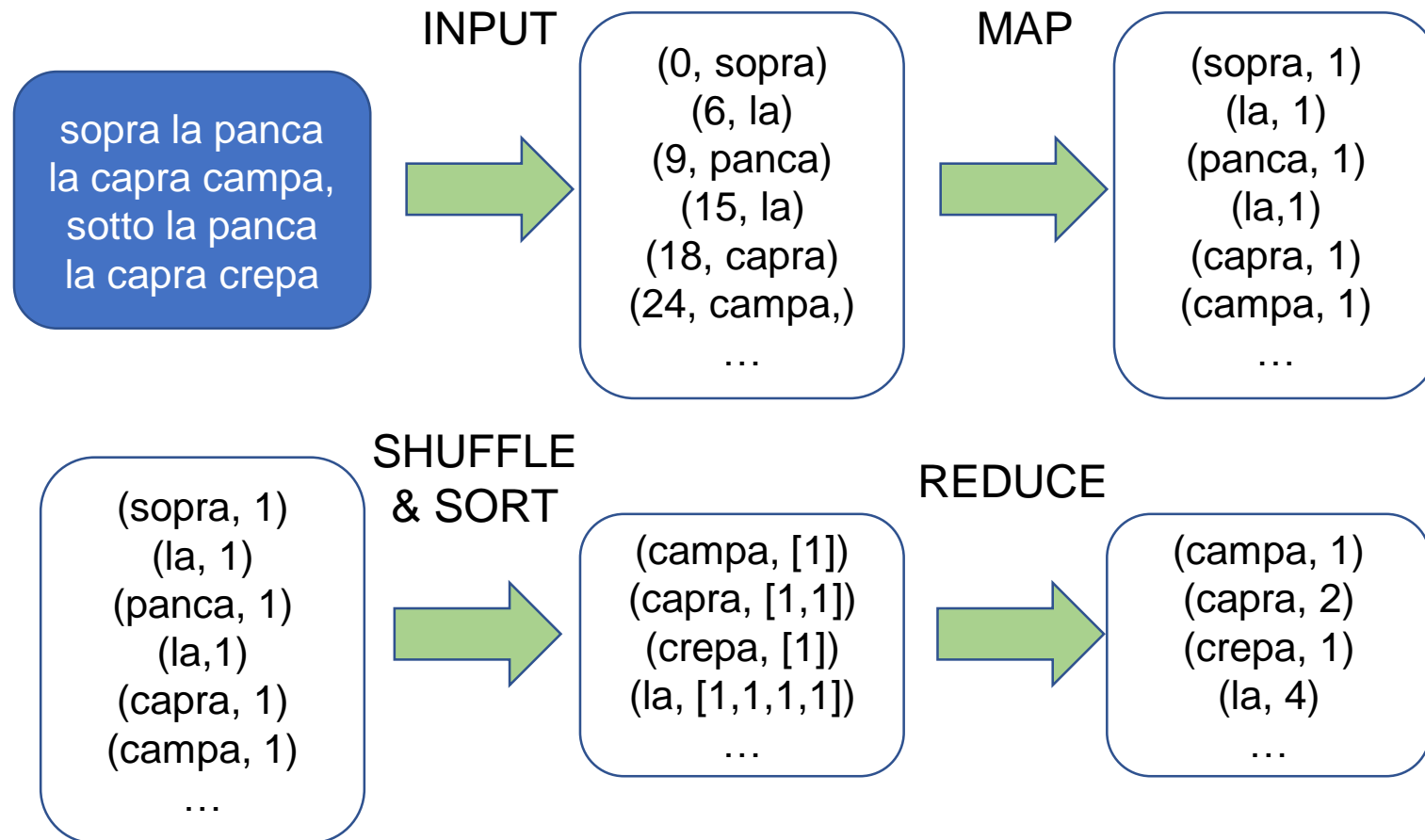
The tasks are scheduled using YARN and run on nodes in the cluster

- If a task fails, it will be automatically rescheduled to run on a different node

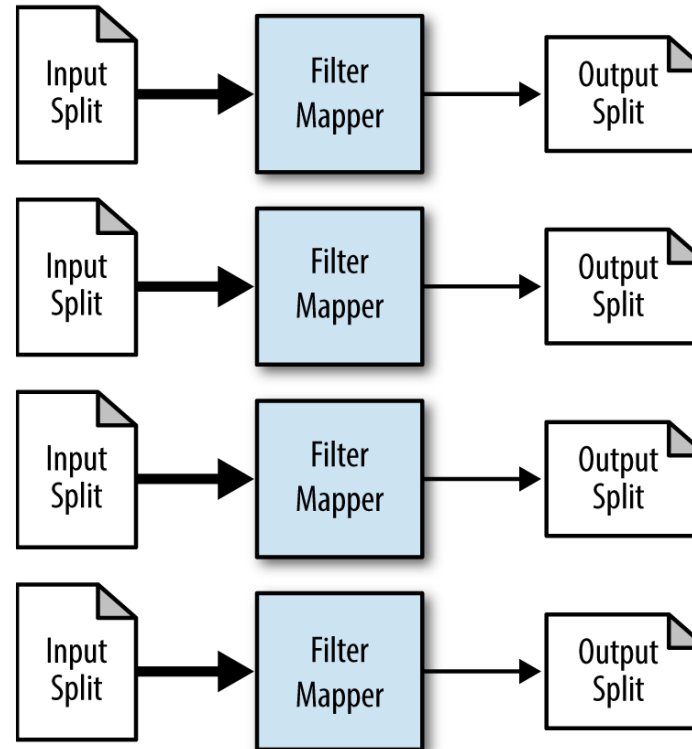
# MapReduce process: an example



# Word count

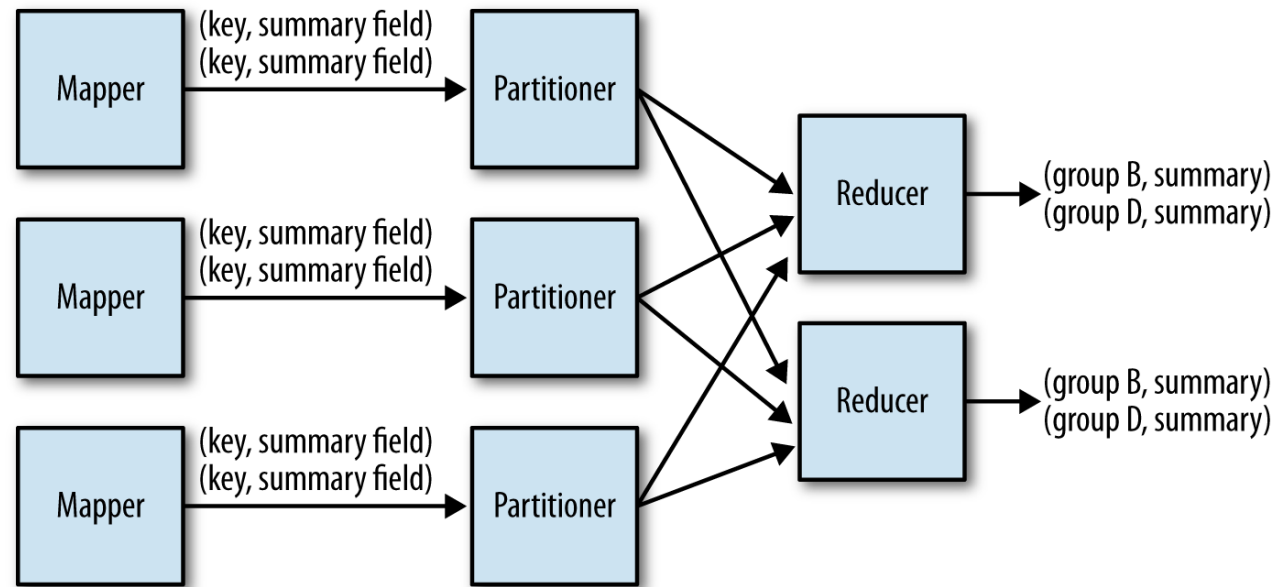


# Filtering pattern

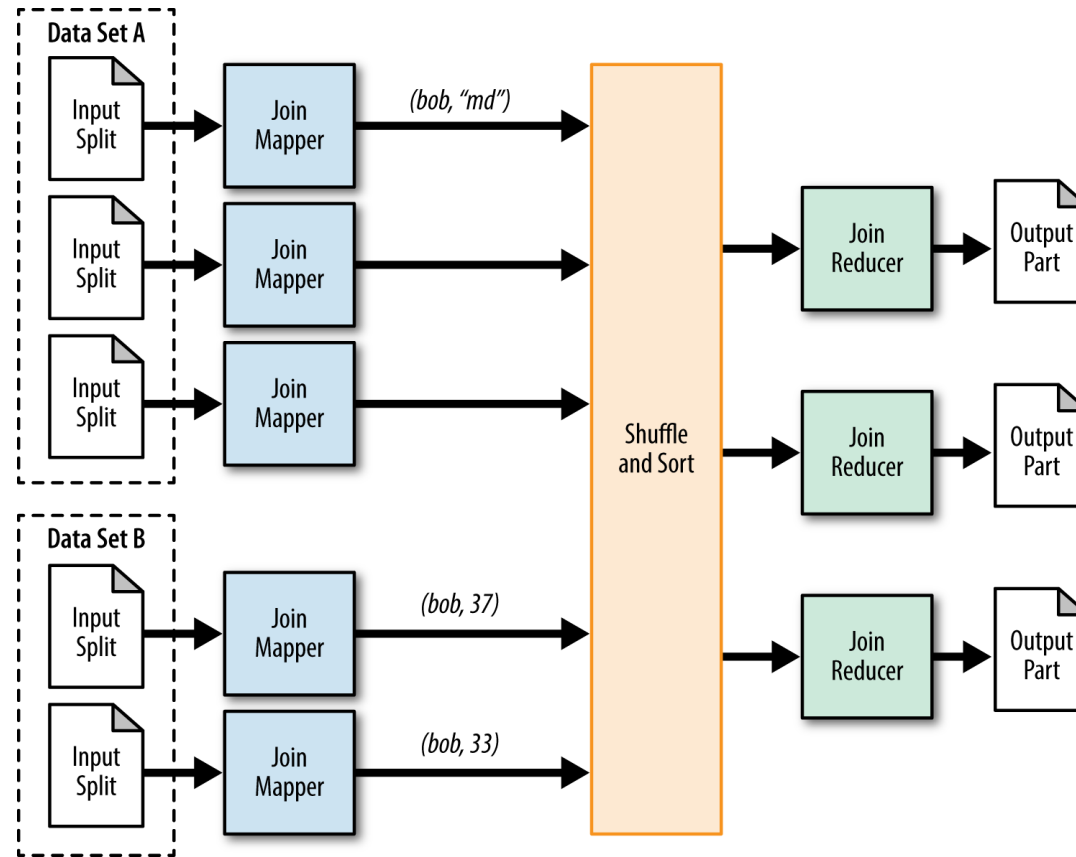




# Summarization pattern



# Join pattern



# Two stage MapReduce

As map-reduce calculations get more complex, it's useful to break them down into stages

- The output of the first stage serves as input to the next one
- The **same output** may be useful for **different subsequent stages**
- The output can be stored in the DFS, forming a **materialized view**

Early stages of map-reduce operations often represent the heaviest amount of data access, so building and saving them once as a basis for many downstream uses saves a lot of work!

# Limitations of Map Reduce

## Designed for batch processing

- Not suitable for iterative algorithms or interactive data mining

## Strict paradigm

- Everything has to fit into Map and Reduce
- Complex algorithms will take multiple jobs and passes on hard disk

## New hardware capabilities are not exploited

- Too much pressure on disk; RAM and multicore not adequately exploited

## Too much complex

# Spark

It is a **fast and general-purpose execution engine**

- **In-memory** data storage for very fast iterative queries
- Easy **interactive** data analysis
- Combines **different processing models** (machine learning, SQL, streaming, graph computation)
- Provides (not only) a MapReduce-like engine...
- ... but it's **up to 100x faster** than Hadoop MapReduce

Compatible with Hadoop's storage APIs

- Can run on top of a Hadoop cluster
- Can read/write to any database and any Hadoop-supported system, including HDFS, HBase, Parquet, etc.

# What does Spark offer?

## In-memory data caching

- HDD is scanned once, then data is written to/read from RAM

## Lazy computations

- The job is optimized before its execution

## Efficient pipelining

- Writing to HDD is avoided as much as possible

# Spark pillars

Two main abstractions of Spark

## **RDD – Resilient Distributed Dataset**

- An RDD is a collection of data items
- It is split into partitions
- It is stored in memory on the worker nodes of the cluster

## **DAG – Direct Acyclic Graph**

- A DAG is a sequence of computations performed on data
- Each node is an RDD
- Each edge is a transformation of one RDD into another

# RDD

RDDs are immutable distributed collection of objects

- **Resilient**: automatically rebuild on failure
- **Distributed**: the objects belonging to a given collection are split into *partitions* and spread across the nodes
  - RDDs can contain any type of Python, Java, or Scala objects
  - Distribution allows for scalability and locality-aware scheduling
  - Partitioning allows to control parallel processing

Fundamental characteristics (mostly from *pure functional programming*)

- **Immutable**: once created, it can't be modified
- **Lazily evaluated**: optimization before execution
- **Cacheable**: can persist in memory, spill to disk if necessary
- **Type inference**: data types are not declared but inferred (≠ dynamic typing)



# RDD operations

RDDs offer two types of operations: *transformations* and *actions*

**Transformations** construct a new RDD from a previous one

- E.g.: map, flatMap, reduceByKey, filtering, etc.
- <https://spark.apache.org/docs/latest/programming-guide.html#transformations>

**Actions** compute a result that is either returned to the driver program or saved to an external storage system (e.g., HDFS)

- E.g.: saveAsTextFile, count, collect, etc.
- <https://spark.apache.org/docs/latest/programming-guide.html#actions>

# RDD operations

RDDs are **lazily evaluated**, i.e., they are computed when they are used in an action

- Until no action is fired, the data to be processed is not even accessed

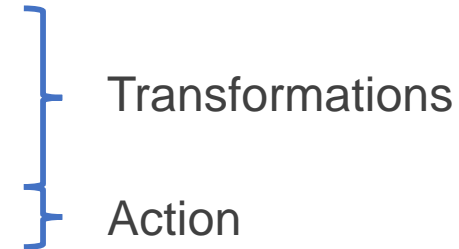
## Example (in Python)

```
sc = new SparkContext
```

```
rddLines = sc.textFile("myFile.txt")
```

```
rddLines2 = rddLines.filter (lambda line: "some text" in line)
```

```
rddLines2.first()
```



Transformations

Action

There is no need to compute and store everything

- In the example, Spark simply scans the file until it finds the first matching line

# DAG

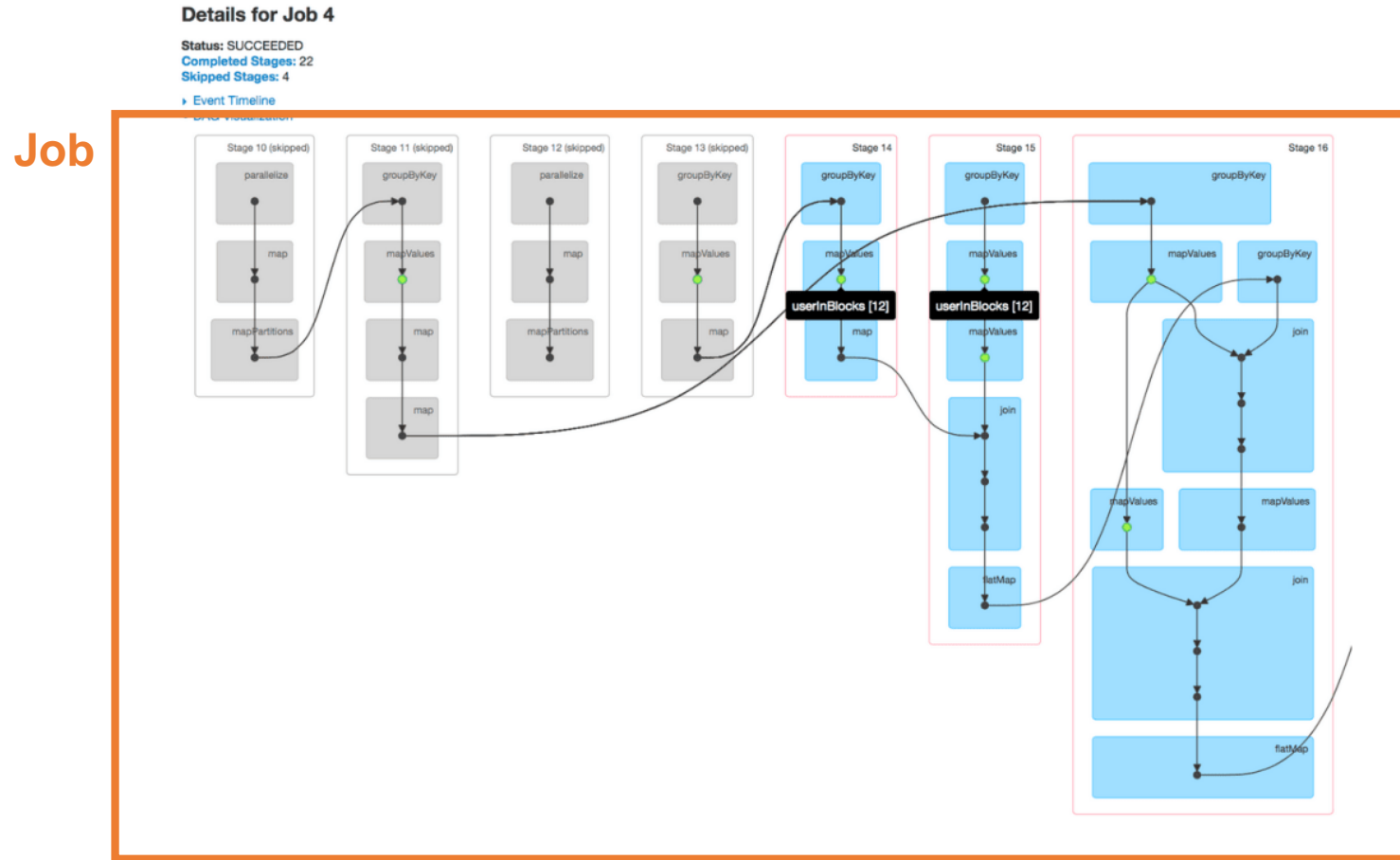
Based on the user application and on the lineage graphs, Spark computes a **logical execution plan** in the form of a DAG

- Which is later transformed into a physical execution plan

The DAG (Directed Acyclic Graph) is **a sequence of computations performed on data**

- Nodes are **RDDs**
- Edges are operations on RDDs
- The graph is Directed: transformations from a partition A to a partition B
- The graph is Acyclic: transformations cannot return an old partition

# Application decomposition



# DataFrame and DataSet

RDDs are immutable distributed collection of objects

DataFrames and DataSets are immutable distributed collection of records organized into named columns (i.e., a table)

- **Simply put, RDDs with a schema attached**
- Support both relational and procedural processing (e.g., SQL, Scala)
- Support complex data types (struct, array, etc.) and user defined types
- Cached using columnar storage

Can be built from many different sources

- DBMSs, files, other tools (e.g., Hive), RDDs

Type conformity is checked

- At *compile time* for DataSets; at *runtime* for DataFrames

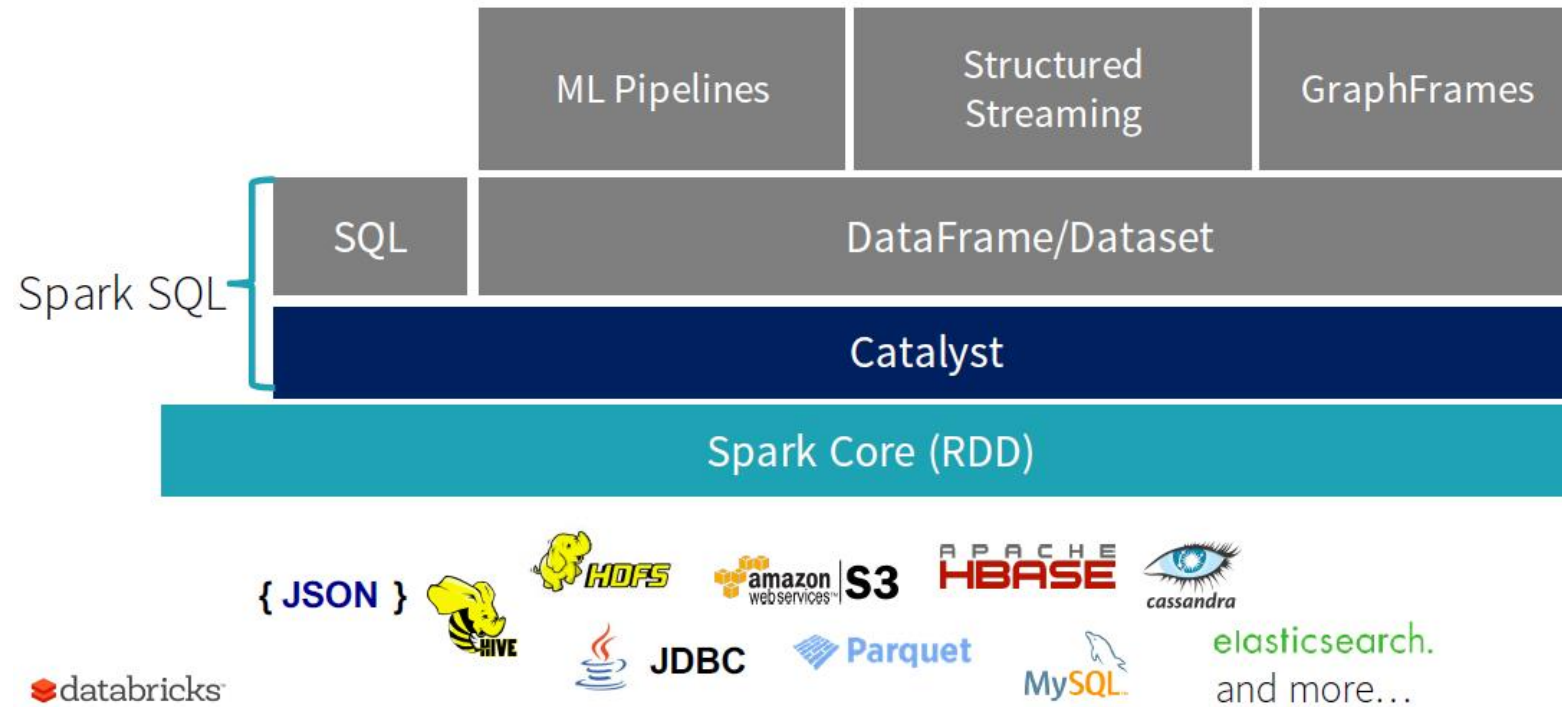
# DataFrame and DataSet

Still lazily evaluated...

...but supports under-the-hood optimizations and code generation

- **Catalyst optimizer creates optimized execution plans**
  - IO optimizations such as skipping blocks in parquet files
  - Logic push-down of selection predicates
- JVM code generation for all supported languages
  - Even non-native JVM languages; e.g., Python

# Spark structured



# Why structure?

## Cons

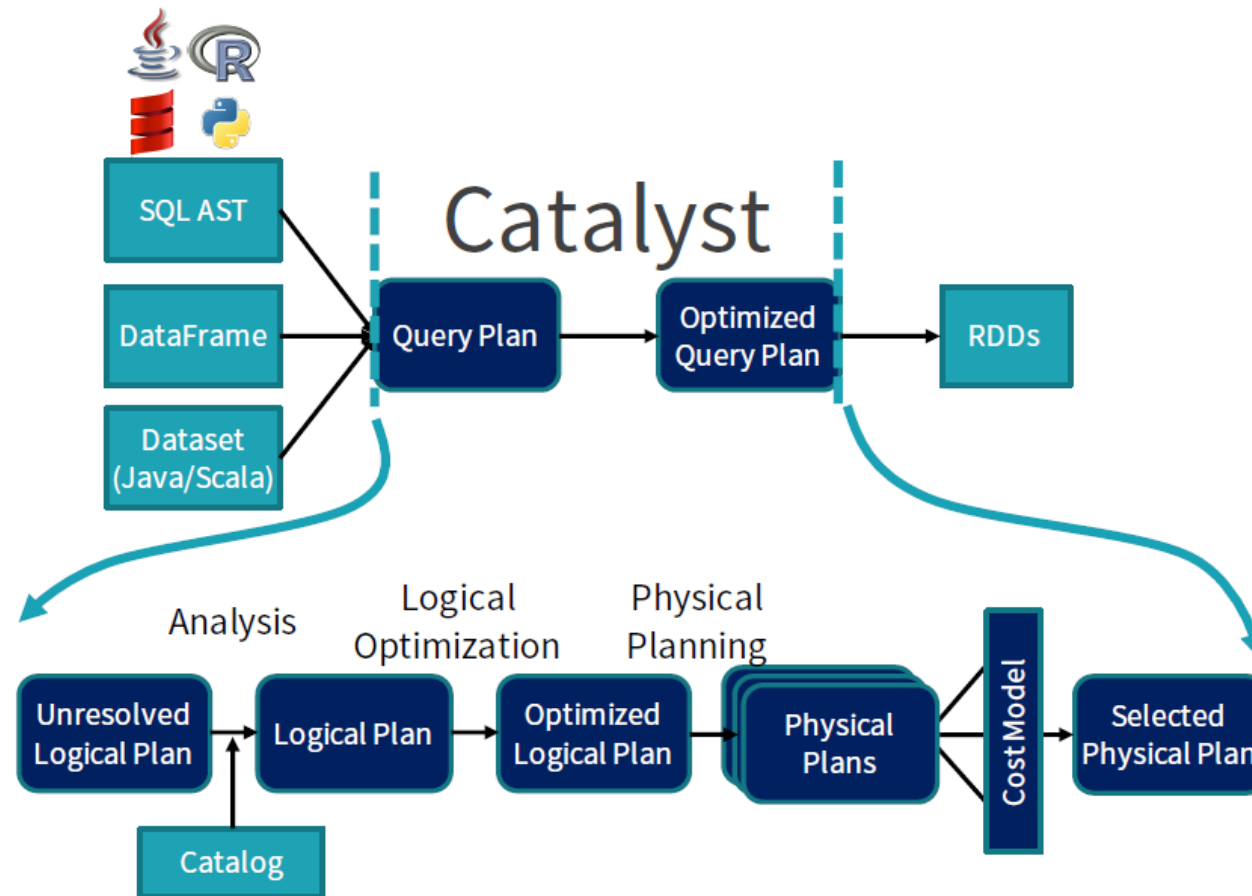
- **Structure imposes some limits**
  - RDDs enable any computation through user defined functions

## Pros

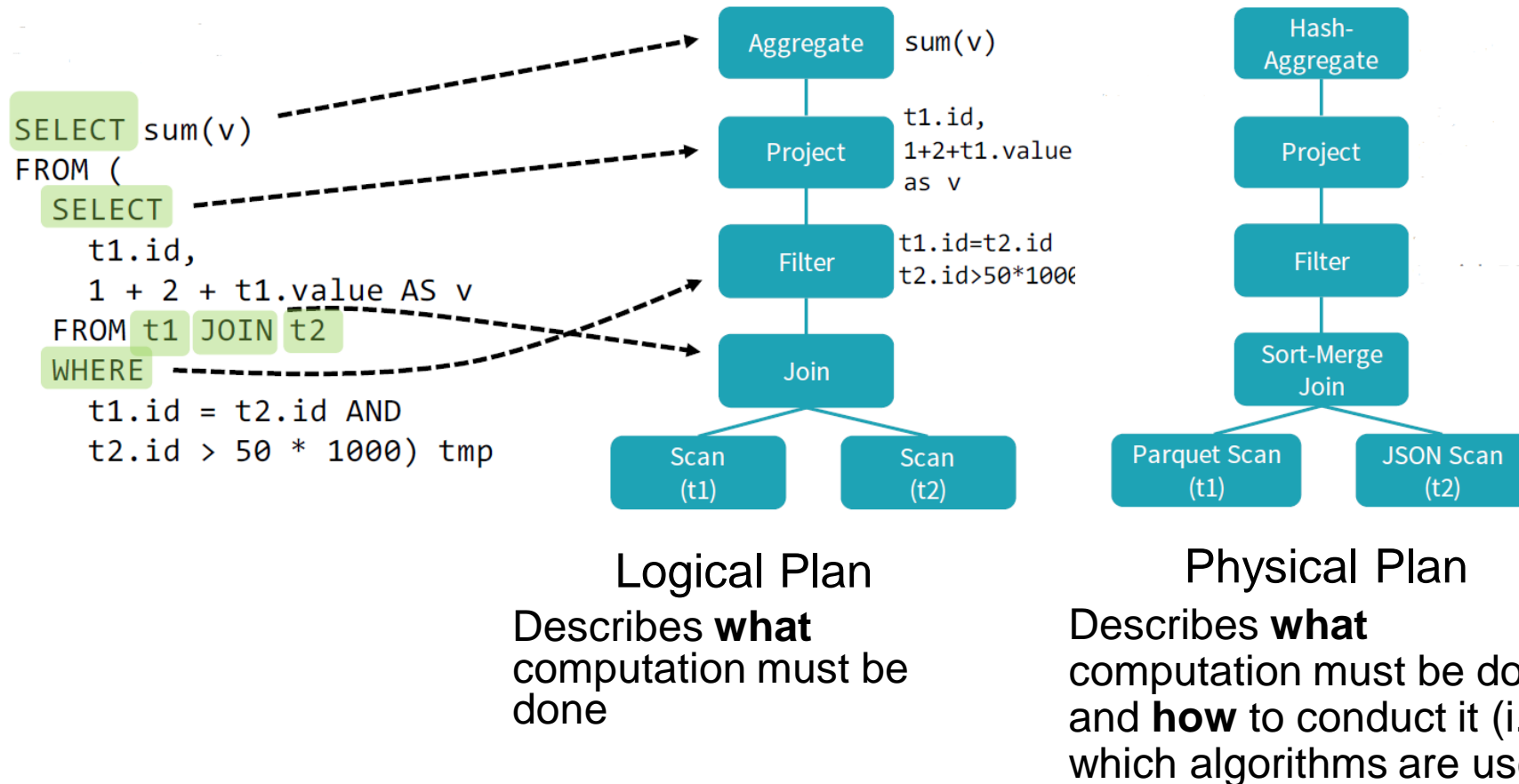
- The most common computations are supported
- Language simplicity
- **Opens the room to optimizations**
  - Hard to optimize a user defined function



# Catalyst



# Logical and Physical Plan



# Logical optimization

## Based on rules

- **A rule is a function** that can be applied on a portion of the logical plan

## Implemented as Scala functions

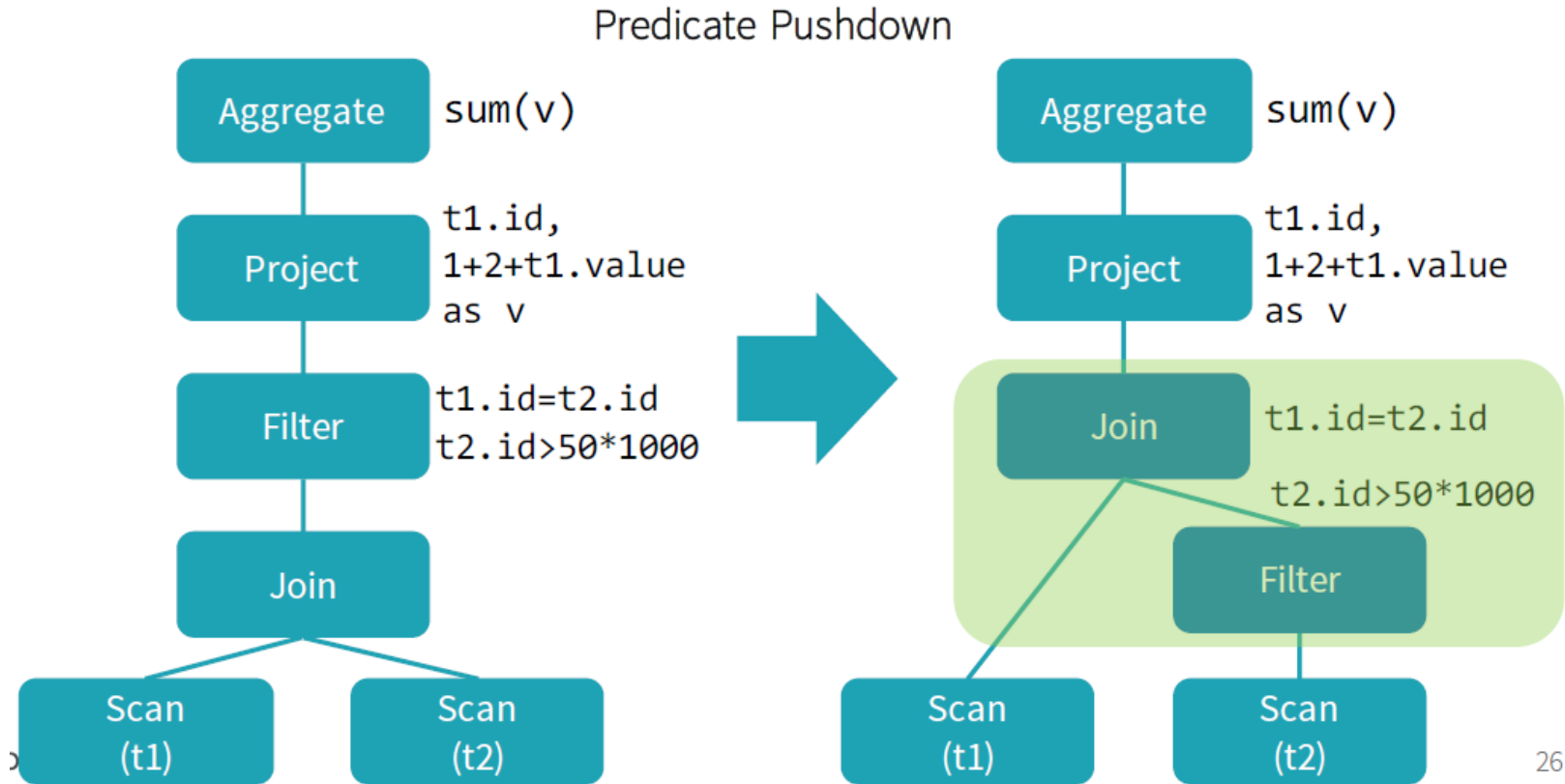
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

## Several types of rules

- **Constant folding**: resolve constant expressions at compile time
- **Predicate pushdown**: push selection predicates close to the sources
- **Column pruning**: project only the required column
- **Join reordering**: change the order of join operations

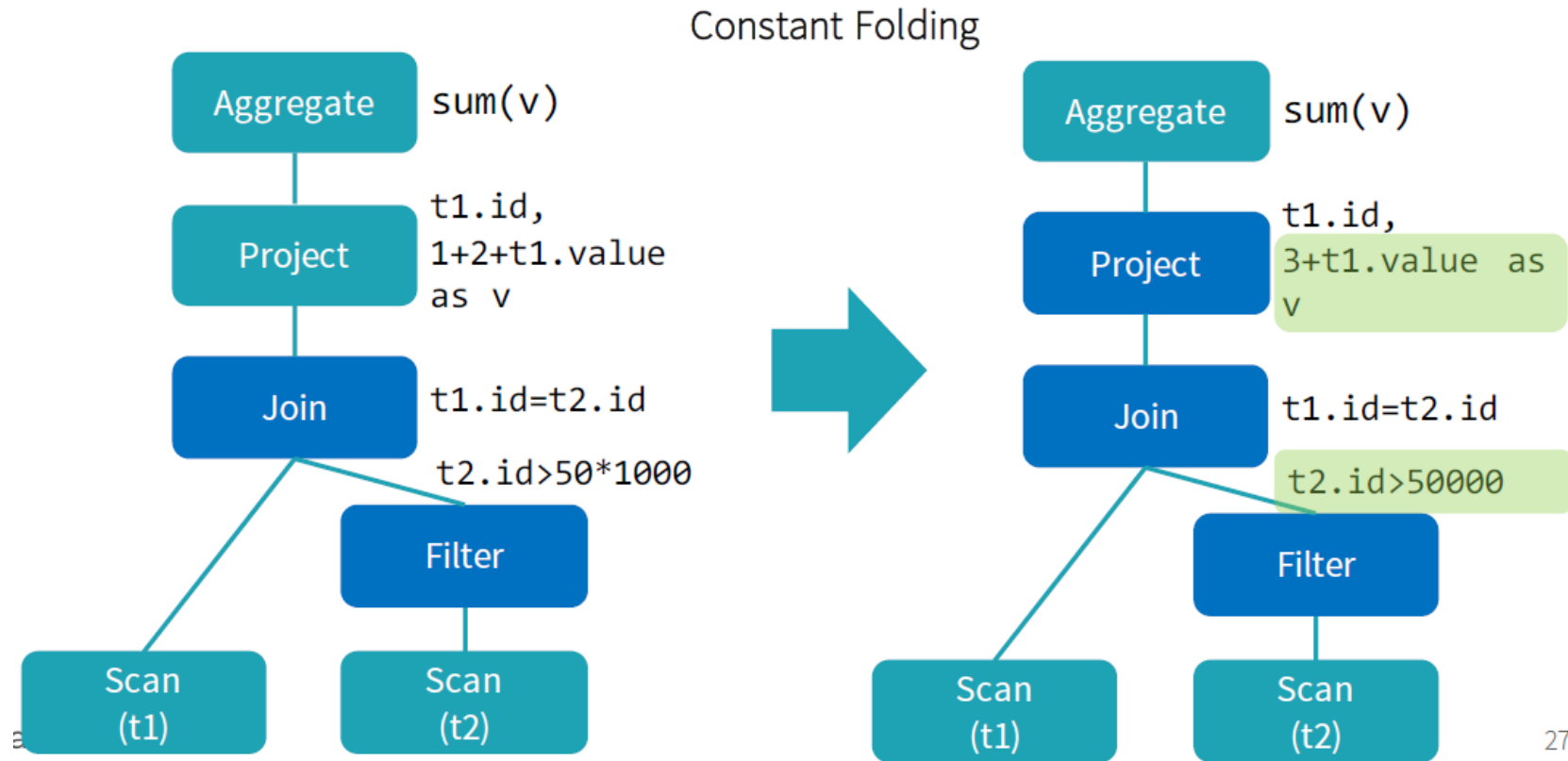
Applied recursively and iteratively until the plan reaches a *fixed point*

# Logical optimization



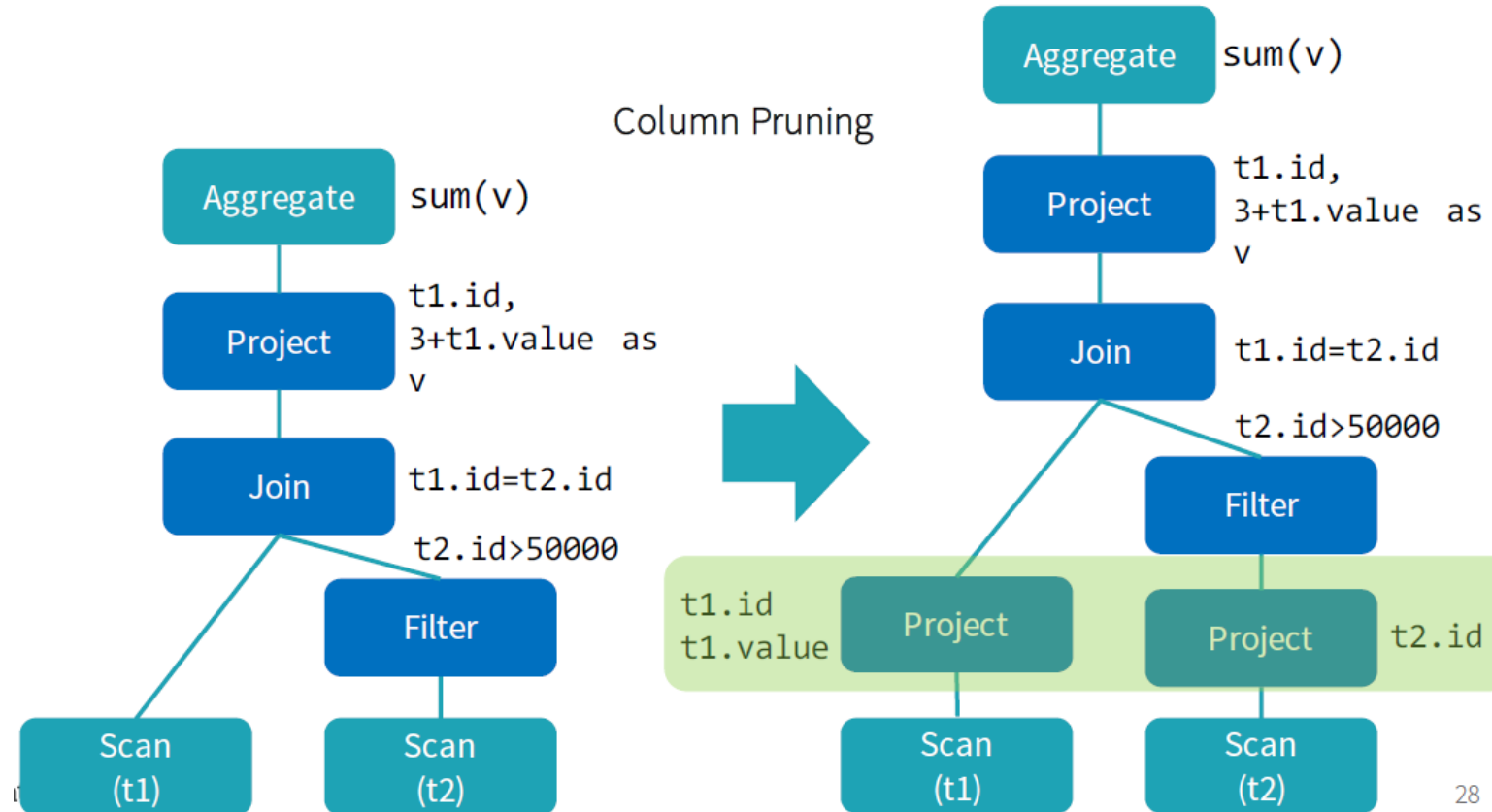
26

# Logical optimization



27

# Logical optimization



28

# Adaptive Query Execution (AQE)

Introduced with version 3.0

## Main idea

- The execution plan is not final
- Reviews are made at each stage boundary
- Additional optimizations are possibly applied, given the information available on the intermediate data

AQE can be defined as a layer on top of the Spark Catalyst which will modify the Spark plan on the fly

## Drawbacks

- The execution stops at each stage boundary for Spark to review its plan
  - But the gain in performance is usually worth
- The Spark UI is more difficult to read
  - Each stage becomes a different job

# AQE - Adaptive Number of Shuffle Partitions

Spark SQL used to set a default number of 200 partitions at each stage.  
AQE automatically adjusts it at runtime.

Tasks: Succeeded/Total
200/200
200/200
2/2
200/200
200/200
2/2

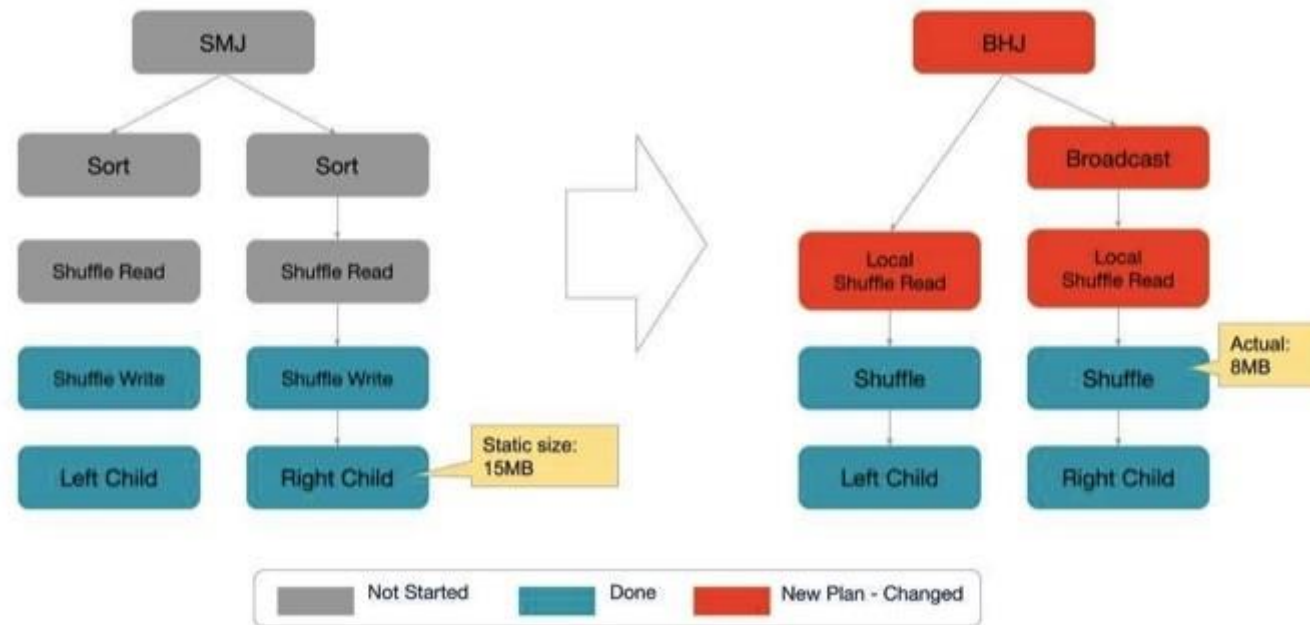


Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1/1 (1 skipped)	1/1 (2 skipped)
1/1 (2 skipped)	1/1 (3 skipped)
1/1 (1 skipped)	1/1 (2 skipped)
1/1	2/2
1/1	2/2
1/1	2/2
1/1	2/2
1/1	2/2

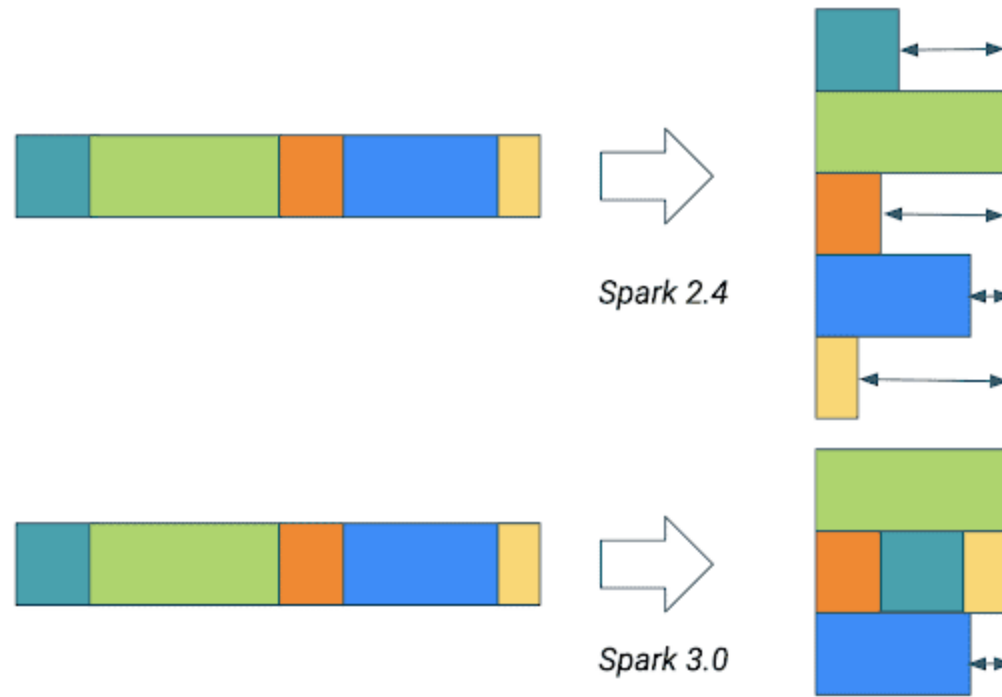


# AQE - Dynamically Converting Sort Merge Joins to Broadcast Joins

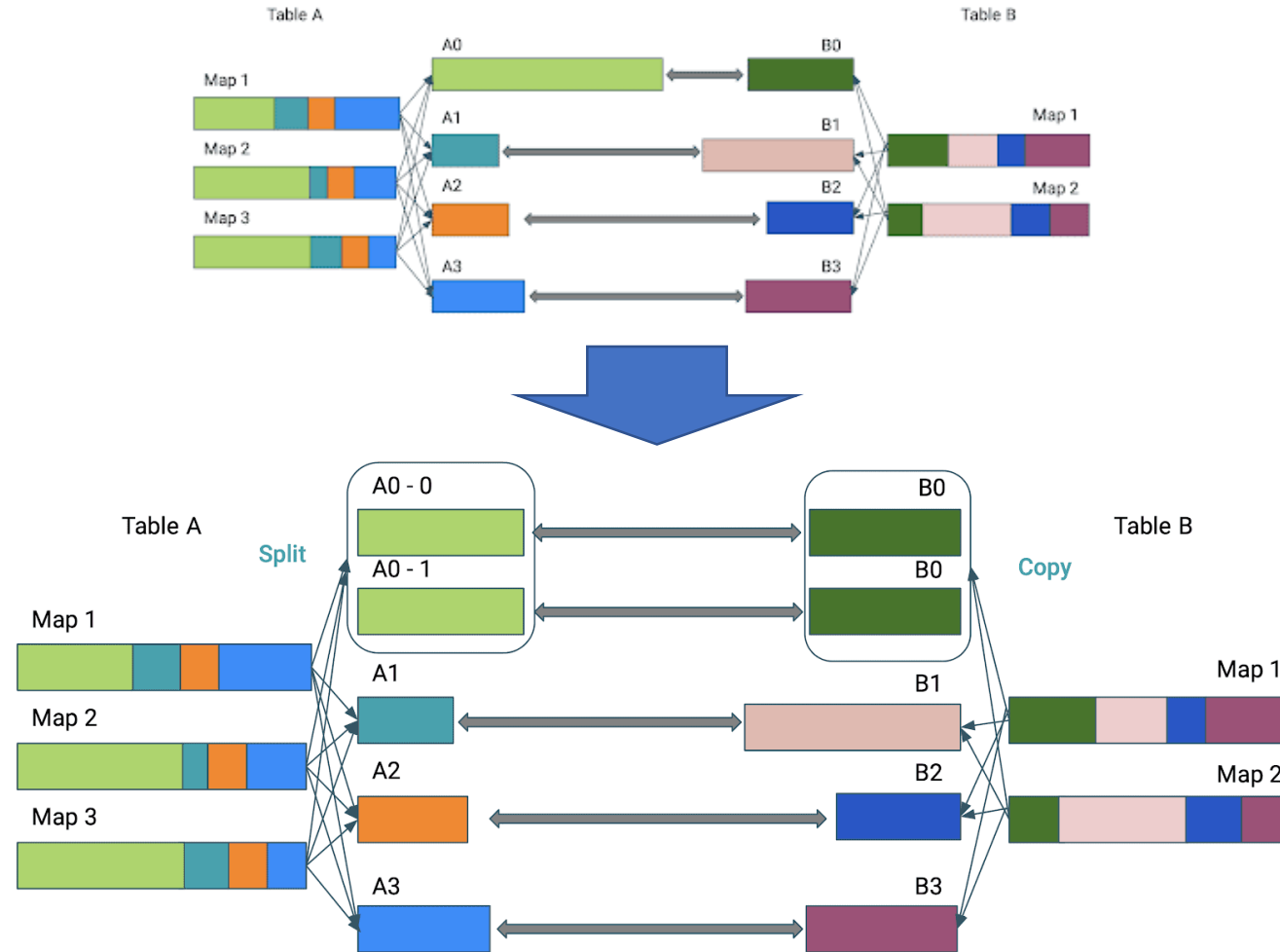
Dynamic switch of join strategies based on actual table sizes.



# AQE - Dynamically Coalesce Shuffle Partitions

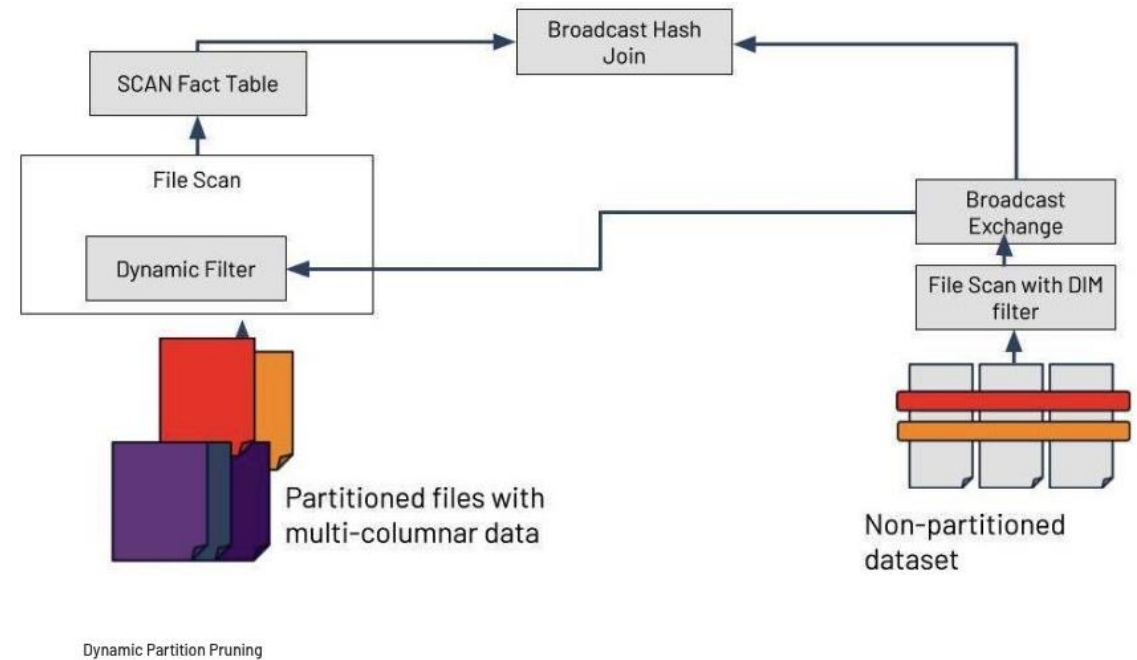
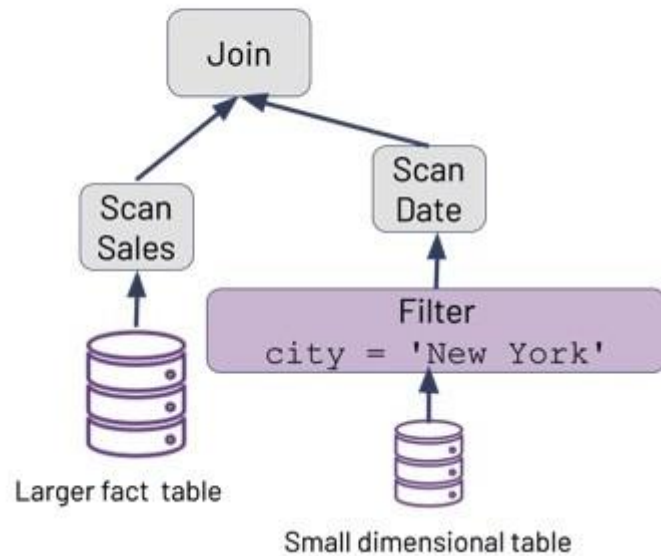


# AQE - Dynamically Optimize Skewed Joins



# AQE - Dynamic Partition Pruning

```
SELECT * FROM Sales JOIN Stores WHERE  
Stores.city = 'New York'
```



# Spark

Suggested reading and resources

