Big Data

Hands on Spark

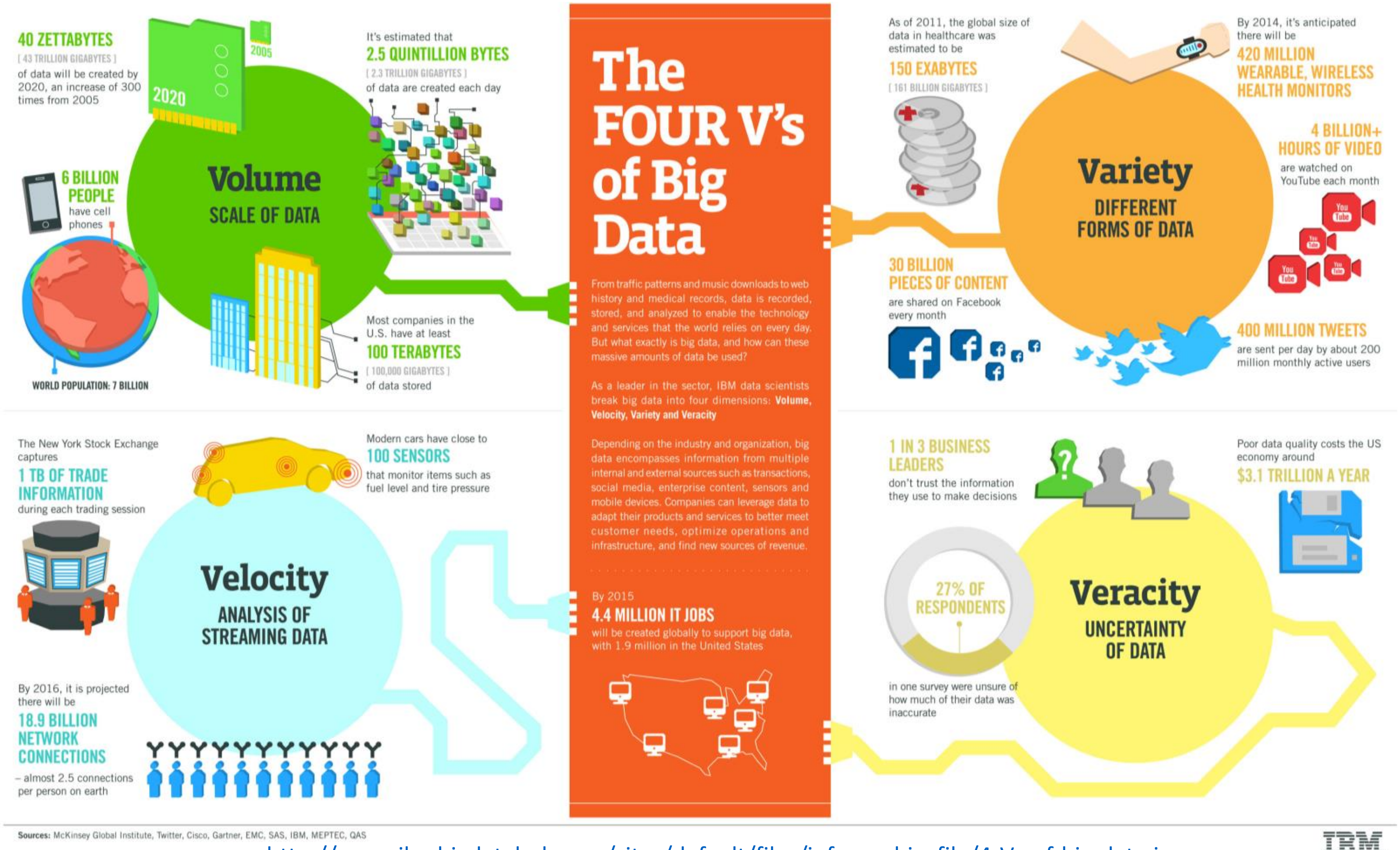
A definition for Big Data

"Big data **exceeds the reach of commonly used hardware** environments **and software** tools to capture, manage, and process it with in a tolerable elapsed time for its user population." - *Teradata Magazine article*, 2011

"Big data refers to data sets whose size is **beyond the ability of typical** database **software tools** to capture, store, manage and analyze." - *The McKinsey Global Institute*, 2012

"Big data is data sets that are so voluminous and complex that **traditional** data processing application **softwares are inadequate** to deal with them." - *Wikipedia*

The four "V's" of Big Data



http://www.ibmbigdatahub.com/sites/default/files/infographic_file/4-Vs-of-big-data.jpg

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

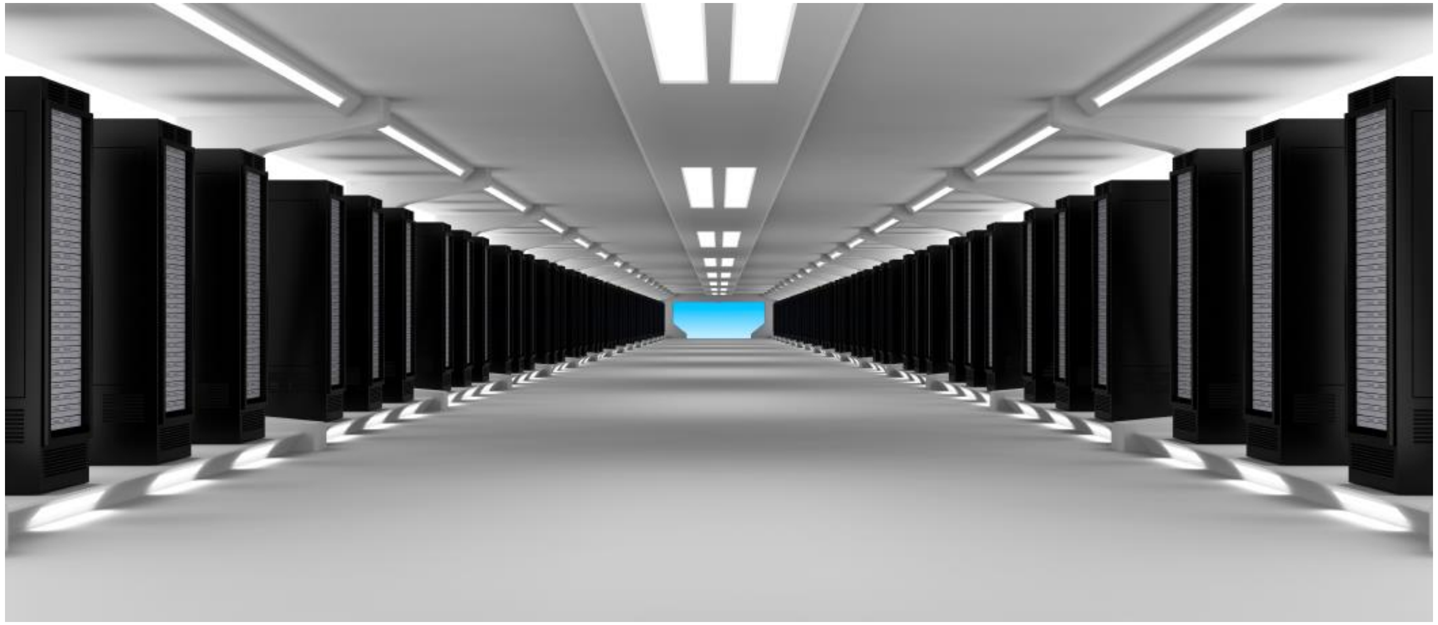
Year:	2014
Size:	1 TB
Speed:	100 MB/s

3 hours

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

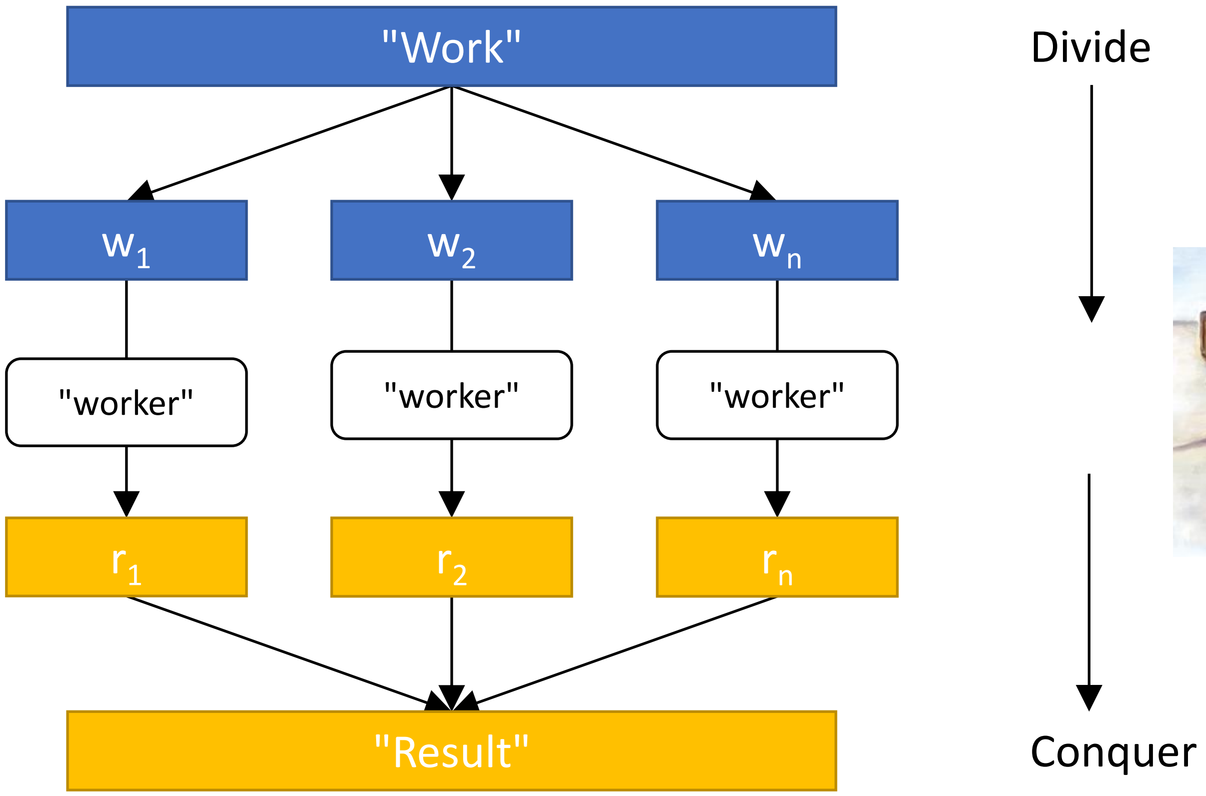
30 minutes

The answer: cluster computing



100 hard disks? 2 mins to read 1TB

Distributed computing: an old idea



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

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.

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

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

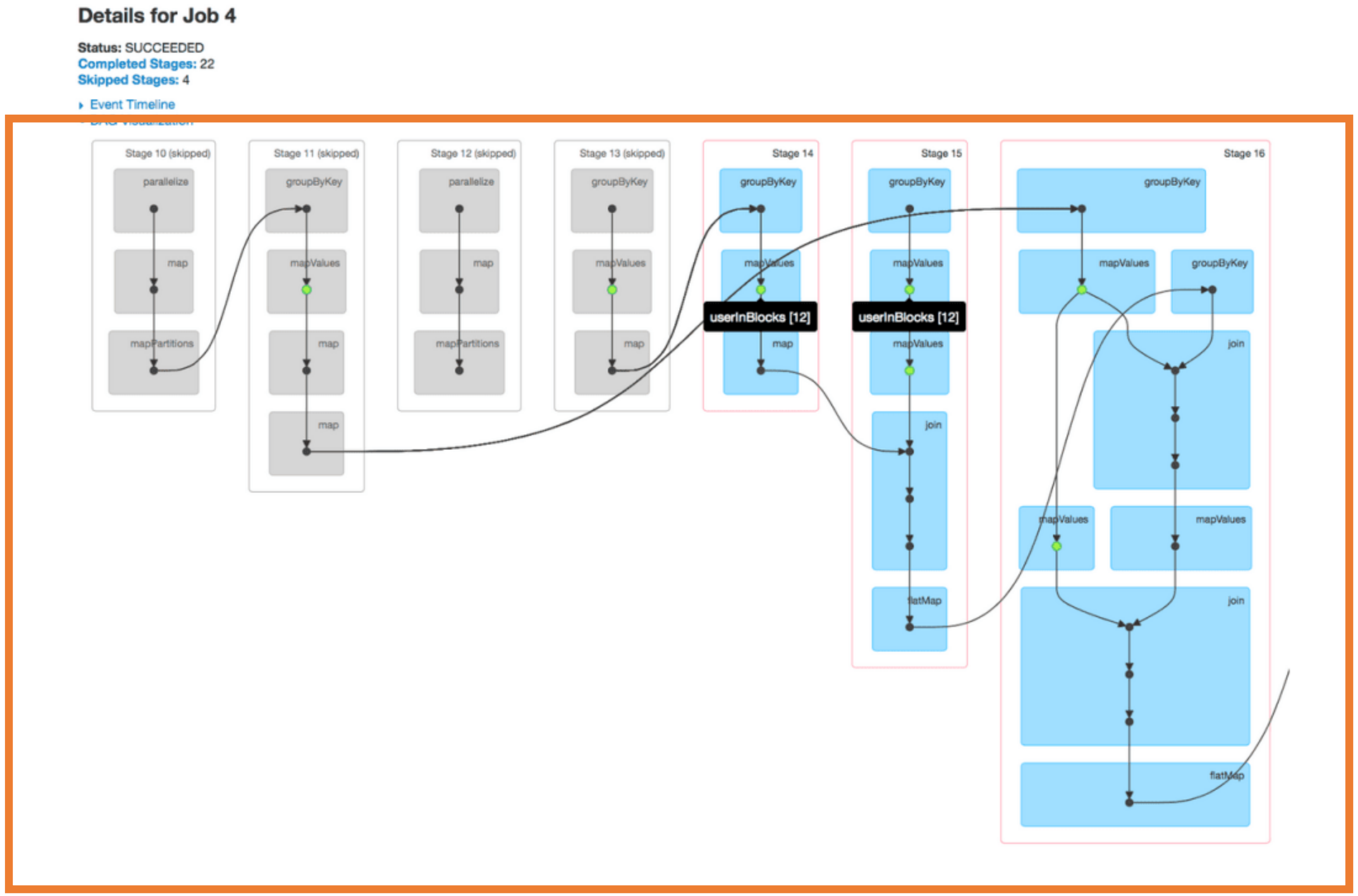
Action

There is no need to compute and store everything

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

Application decomposition

Job



DataFrame and DataSet

RDDs are immutable distributed collection of objects

DataFrames 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, CSV files, other tools (e.g., Hive), RDDs

Type conformity is checked

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

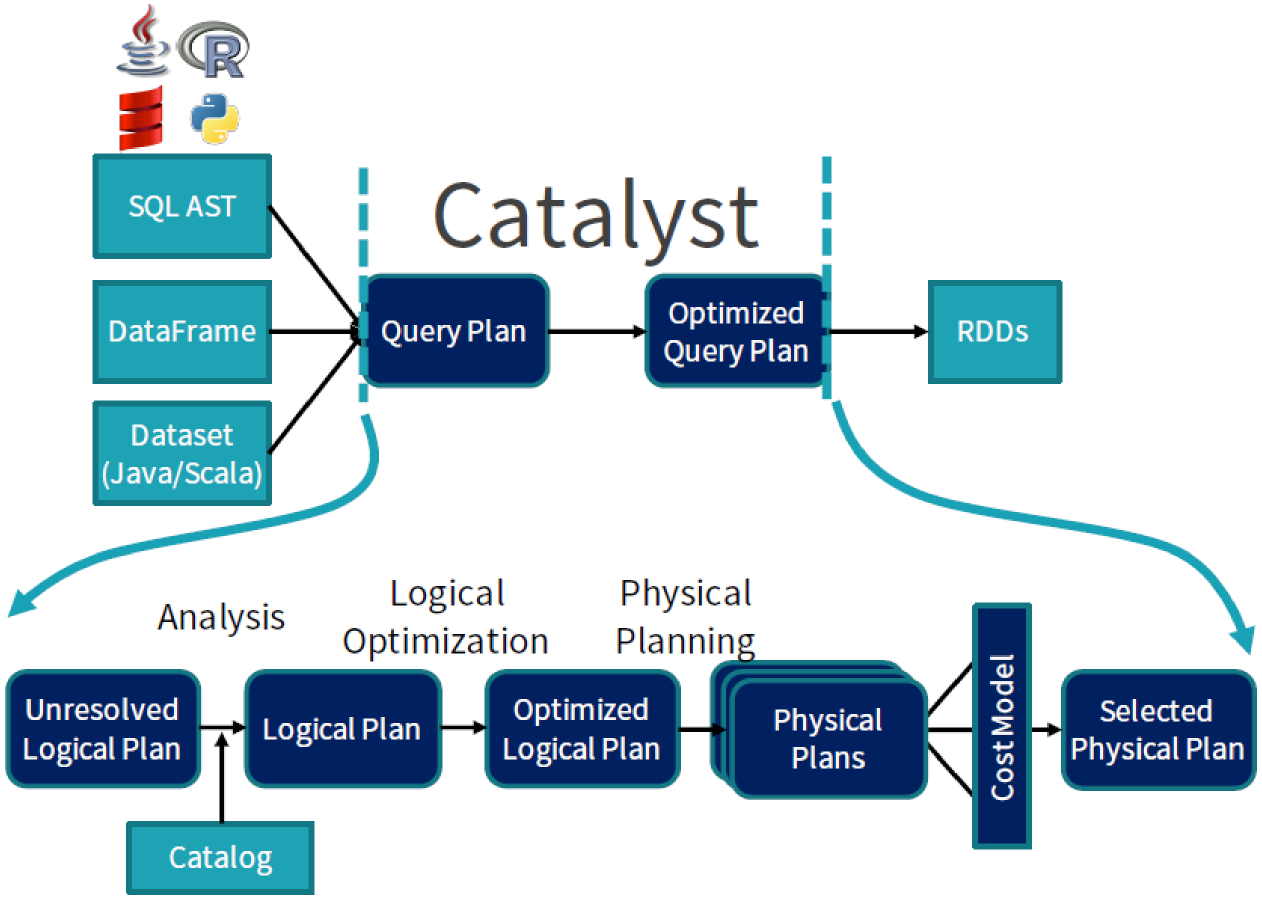
DataFrame and DataSet

Still lazily evaluated…

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

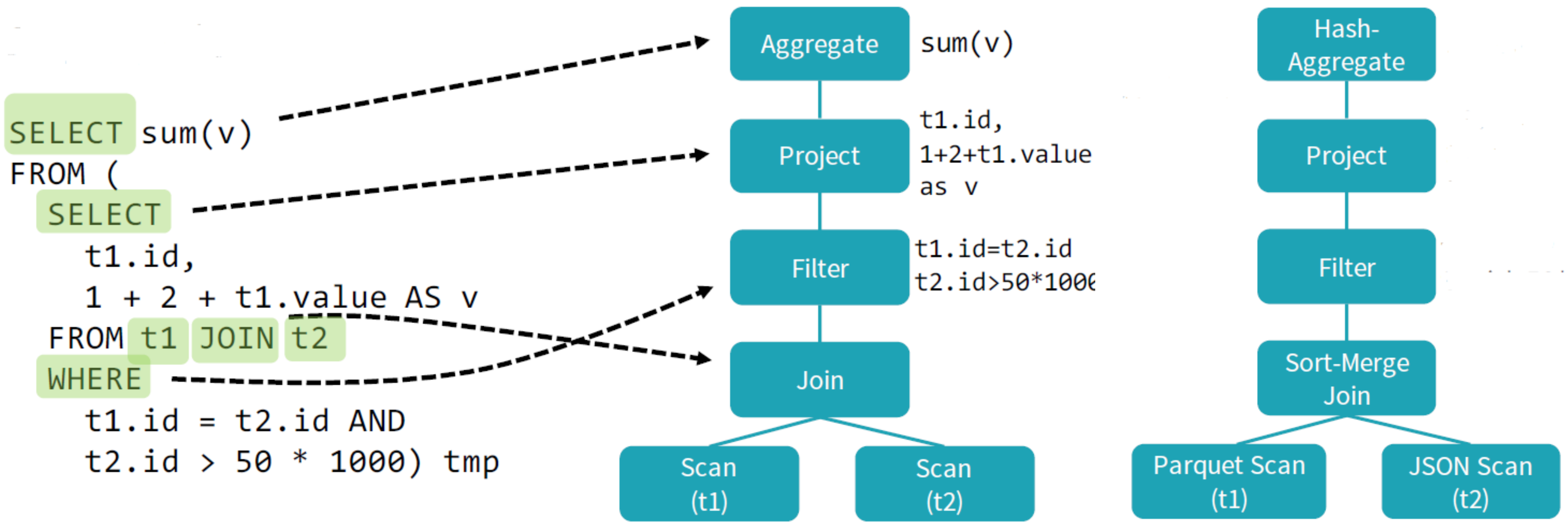
- **Catalyst optimizer creates optimized execution plans**
  - IO optimizations such as skipping blocks in parquet files
  - Logic push-down of selection predicates
- JVM code generation for all supported languages
  - Even non-native JVM languages; e.g., Python

Catalyst



SQL AST
DataFrame
Dataset (Java/Scala)
Catalyst
Query Plan
Optimized Query Plan
RDDs
Analysis
Logical Optimization
Physical Planning
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
Catalog

Logical and Physical Plan



Logical Plan

Describes **what** computation must be done

Physical Plan

Describes **what** computation must be done and **how** to conduct it (i.e., which algorithms are used)

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

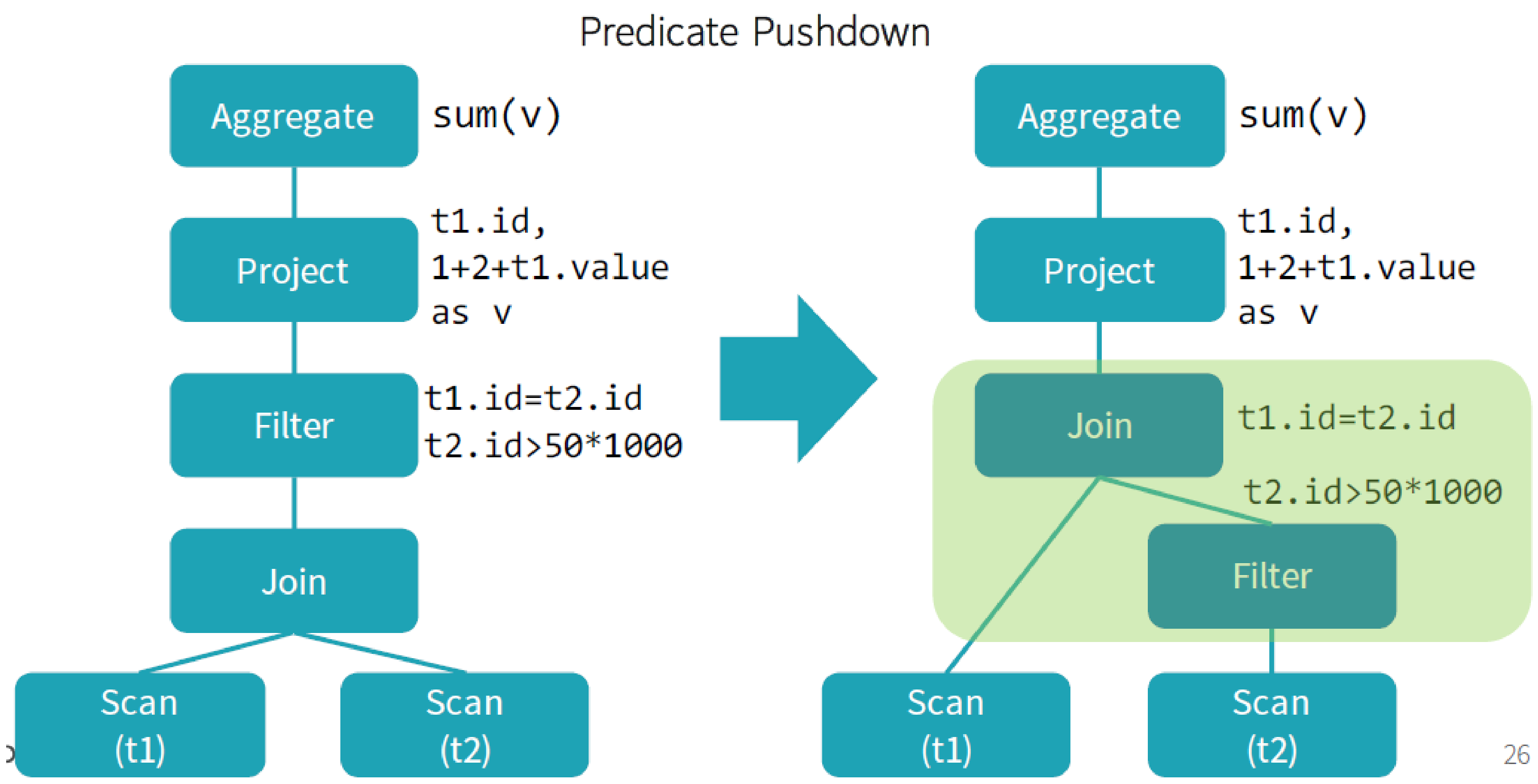
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

Several types of rules

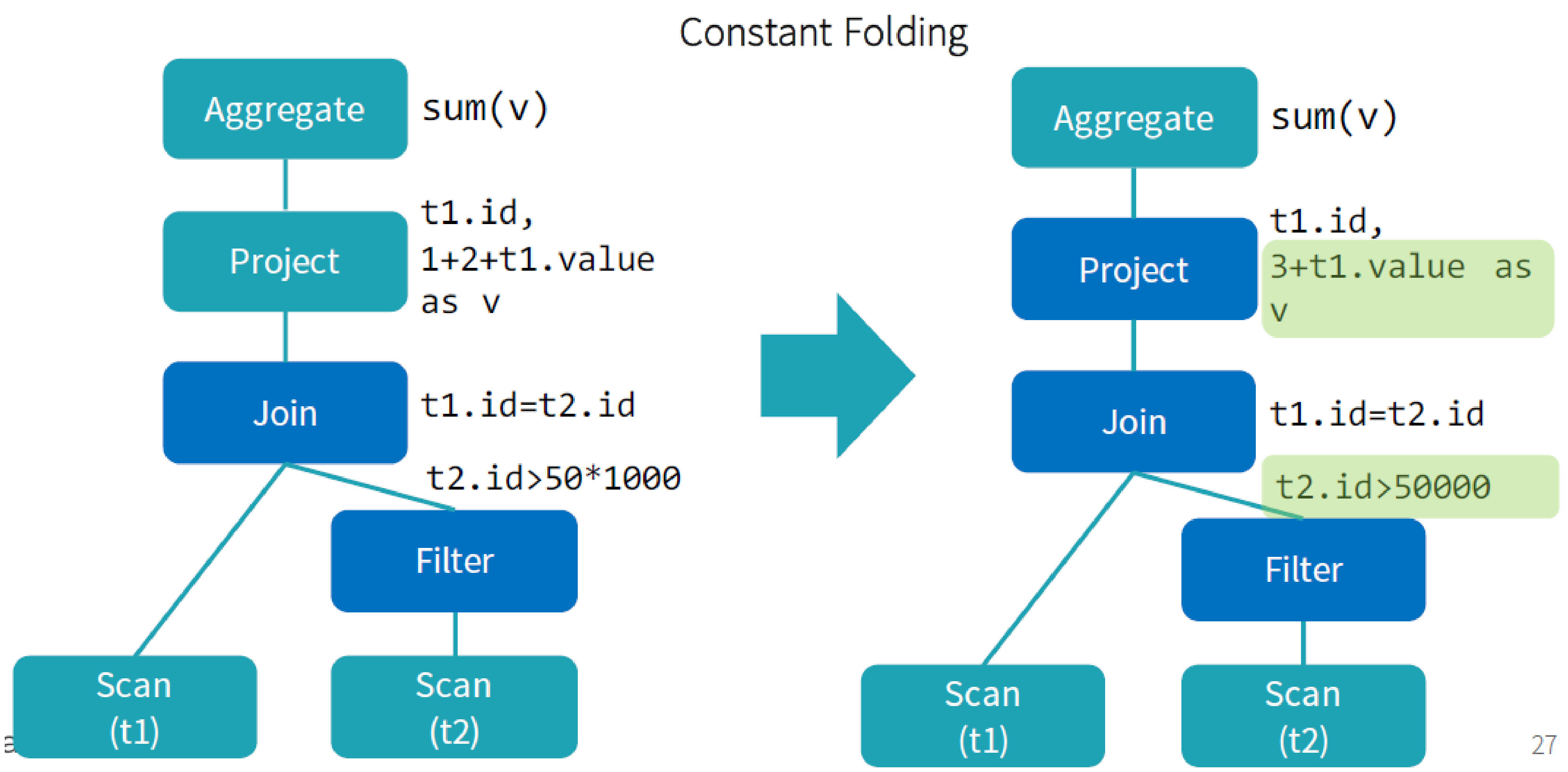
- **Constant folding**: resolve constant expressions at compile time
- **Predicate pushdown**: push selection predicates close to the sources
- **Column pruning**: project only the required column
- **Join reordering**: change the order of join operations

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

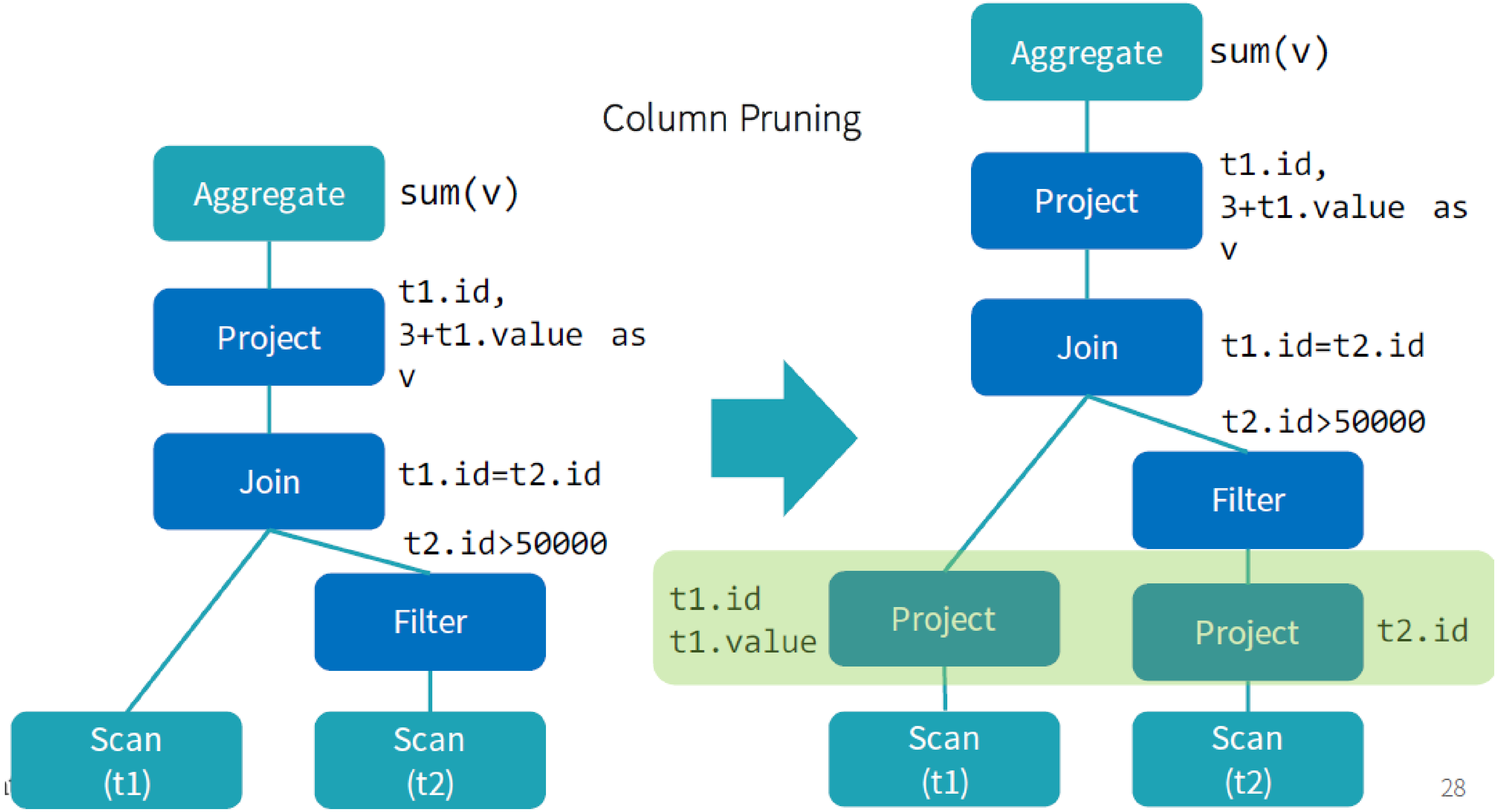
Logical optimization



Logical optimization

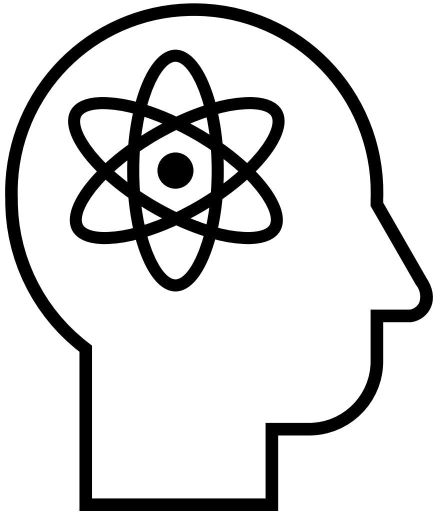


Logical optimization



In action!

Enter the notebook `03-BigData`



https://github.com/w4bo/2022-bbs-dsaa/blob/master/materials/03-BigData.ipynb