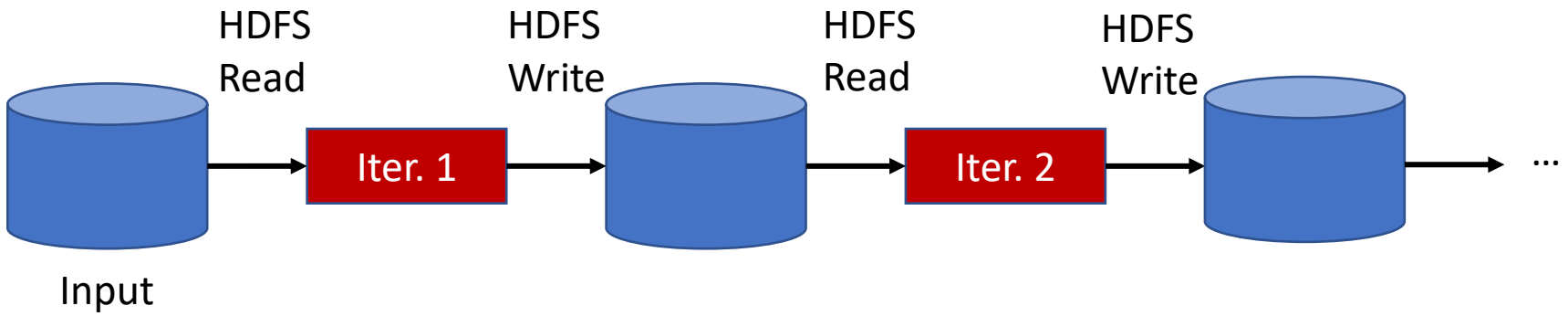
Spark

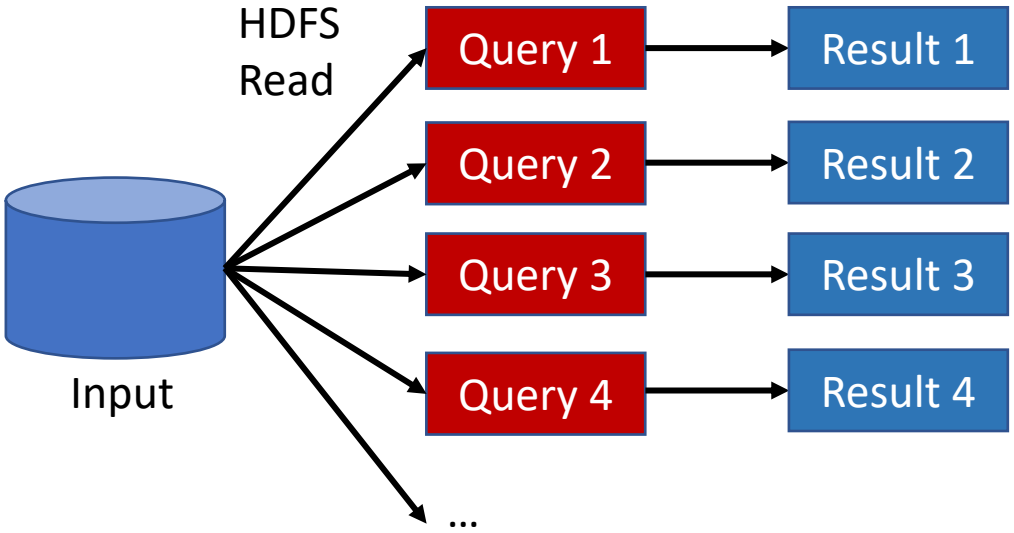
CSCE 678

Problems of MapReduce

- “MapReduce largely simplified big-data analysis on large, unrealistic clusters” – by Matei Zahari
- Users want more complex, multi-stage applications (e.g., iterative ML, graph processing)
- More interactive, real-time jobs than batched jobs

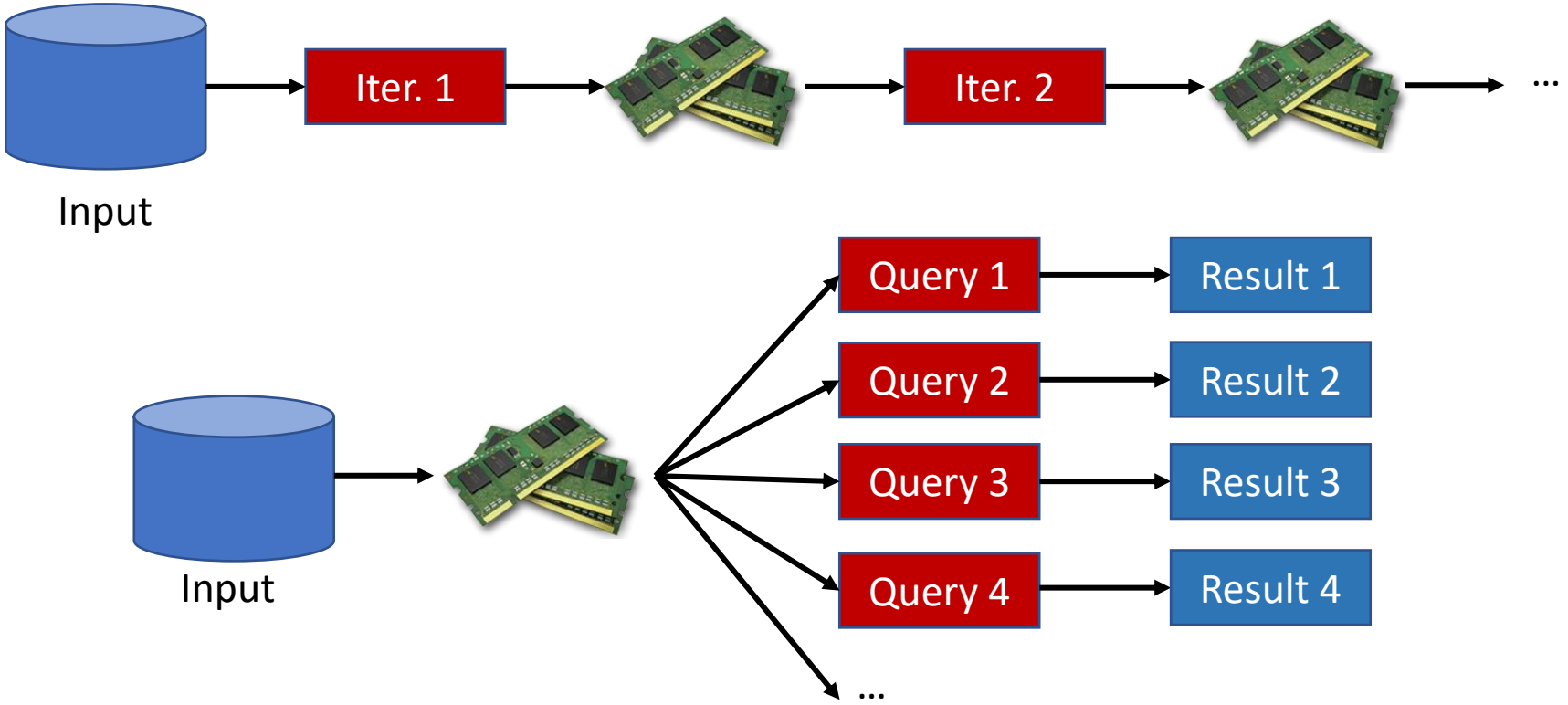
Filesystem-Based Framework





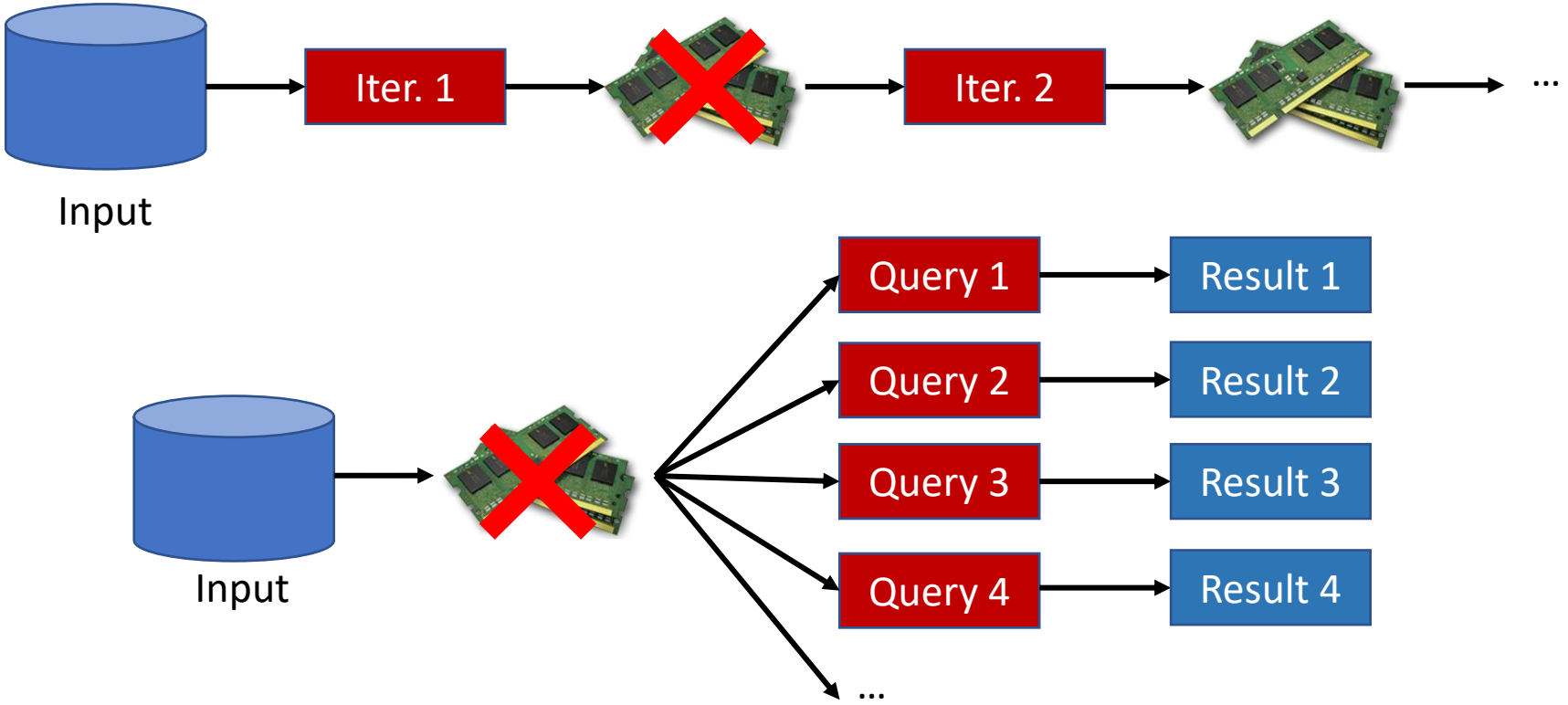
HDFS:
simplified fault tolerance, but too slow due to replication and disk I/O.

Solution: In-Memory Data Sharing



10-100x faster than network/disk

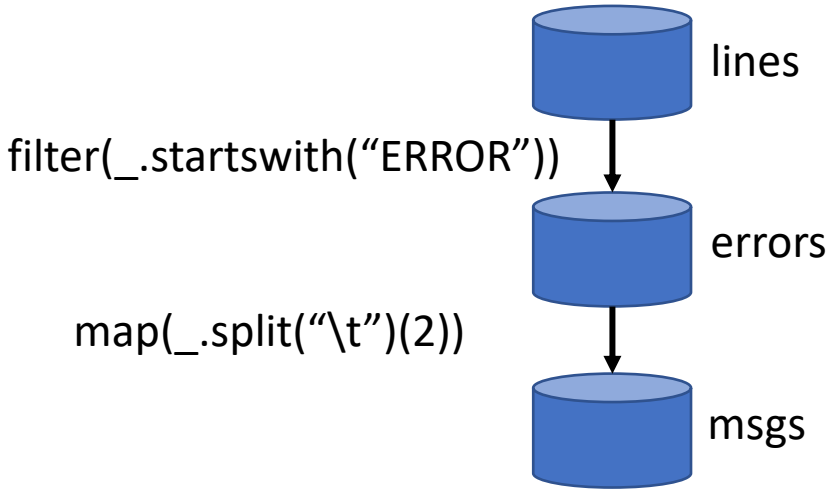
Solution: In-Memory Data Sharing



How to achieve Fault tolerance: What if DRAM crashes?

Resilient Distributed Dataset (RDD)

- Immutable collection of objects
- Statically typed
- Distributed across clusters



(This is Scala code)

```
linesRDD  = spark.textFile("hdfs://log.txt")
errorsRDD = linesRDD.filter(_.startsWith("ERROR"))
msgsRDD   = errorsRDD.map(_.split('\t')(2))

msgsRDD.saveAsTextFile("hdfs://msgs.txt")
```

➡ **Kick off computation (lazy evaluation)**

Spark Programming Interface

- Scala language: lambda-like language on Java
- Run interactively on Scala interpreter
- Interface for specifying dataflow between RDDs:
  - Transformations (filter, map, reduce, etc)
  - Actions (count, persist, save, etc)
  - User-custom controls for partitioning

PySpark

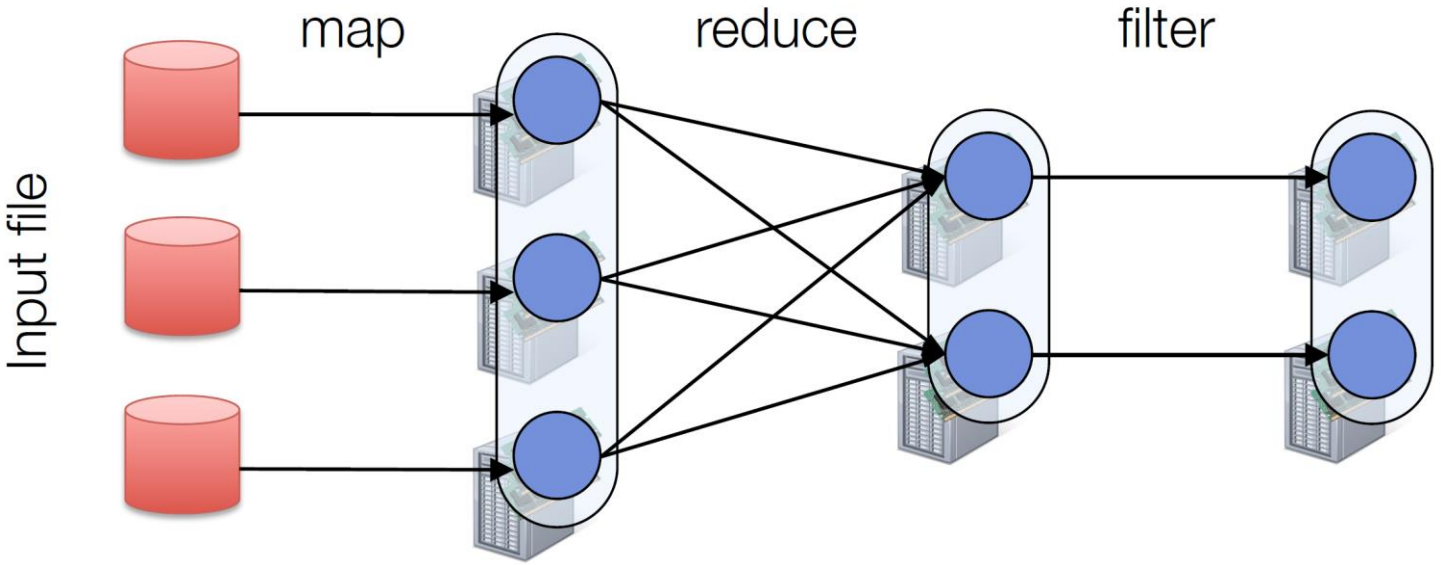
```
linesRDD  = sc.textFile(“hdfs://log.txt")
errorsRDD = linesRDD.filter(lambda x: x.startsWith("ERROR"))
msgsRDD   = errorsRDD.map(lambda x: x.split(‘\t’)(2))

msgsRDD.saveAsTextFile(“hdfs://msgs.txt")
```

Fault Tolerance

- RDDs track **lineage** to rebuild lost data

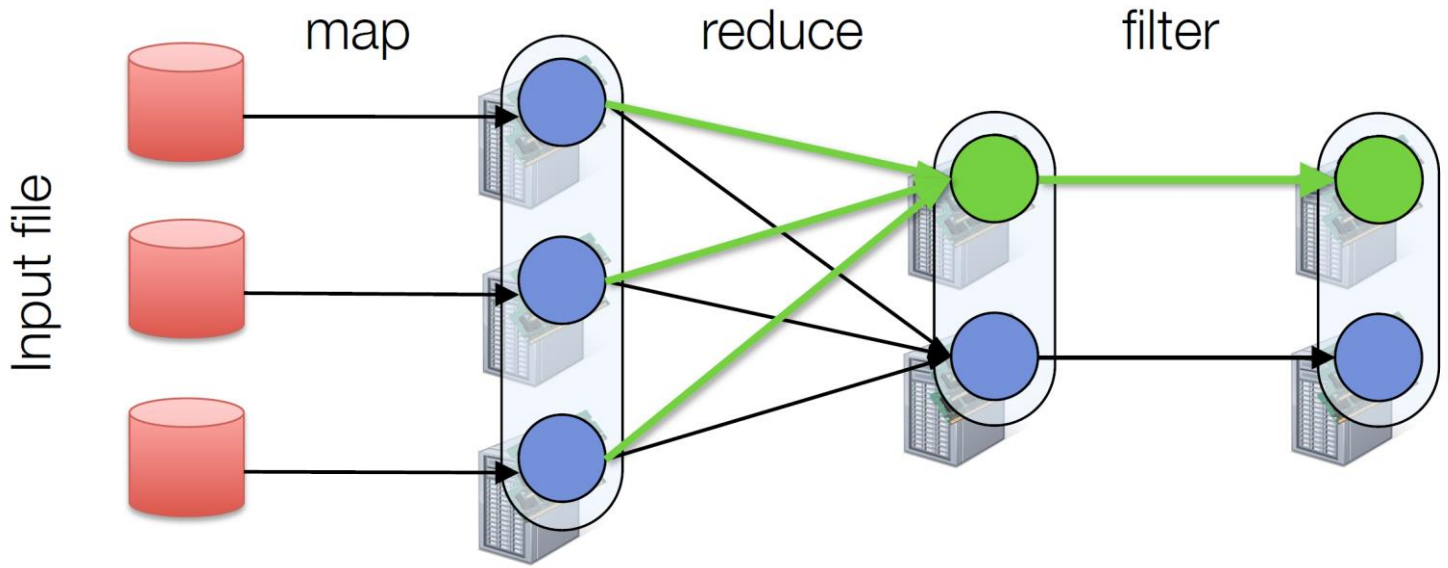
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Fault Tolerance

- RDDs track **lineage** to rebuild lost data

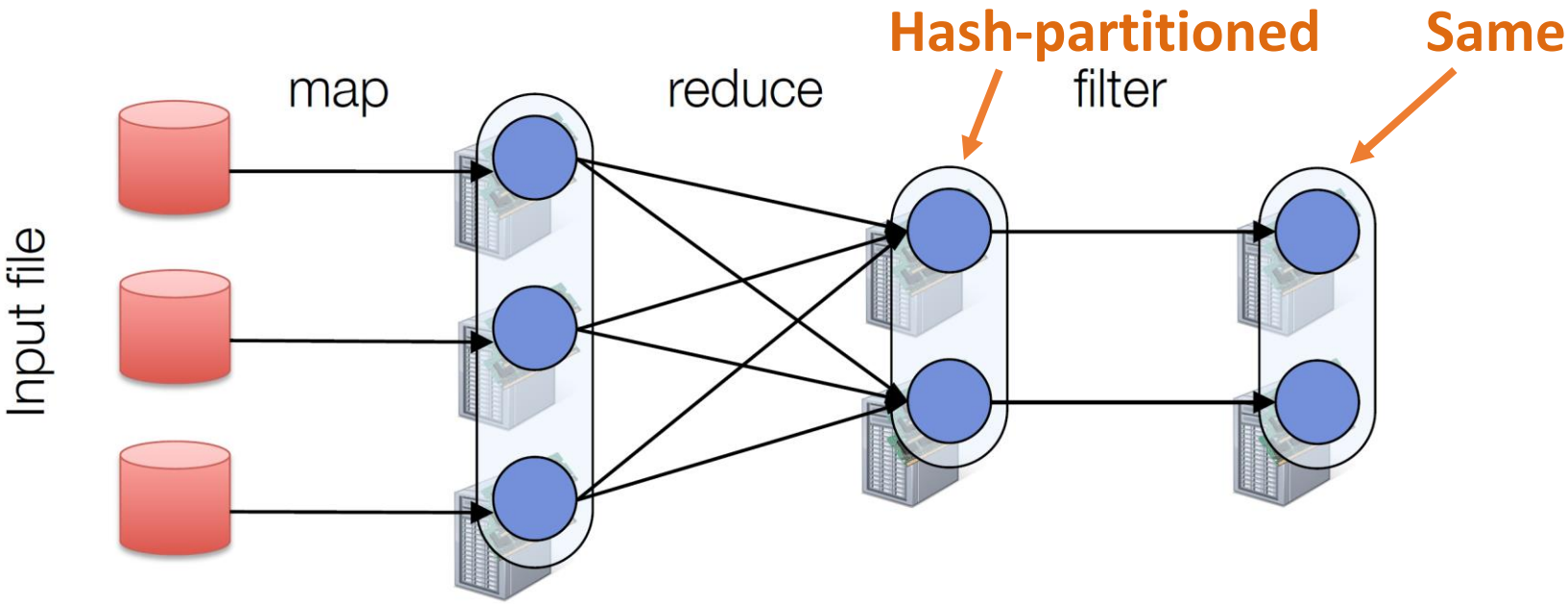
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Partitioning

- RDDs know their partitioning functions

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Example: PageRank

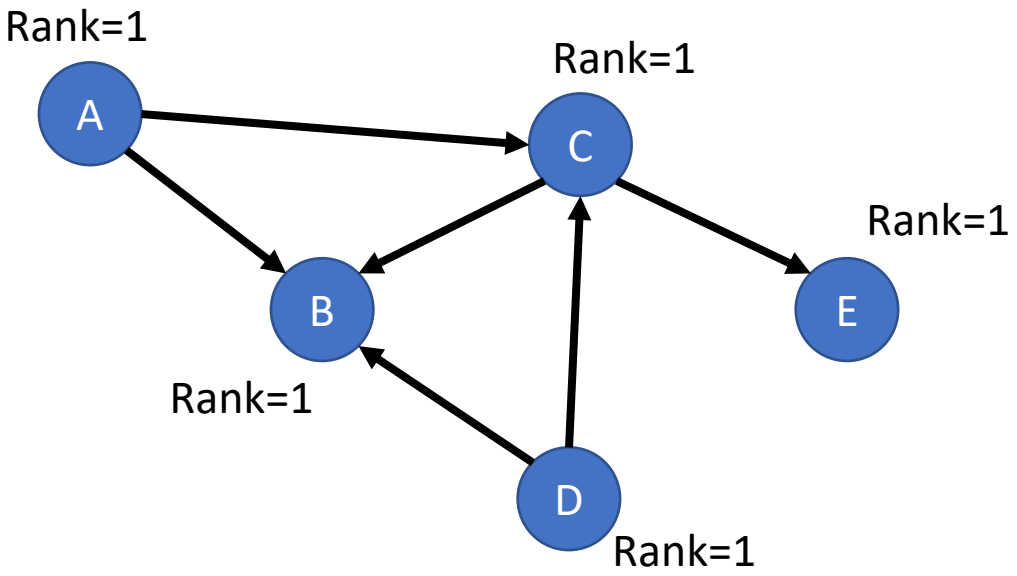
1. Start each page with rank = 1
2. For iteration = 1, …, N, update rank(page) =

$$\sum_{i \in \text{Neighbors}} \text{rank}_i / \, |\text{Neighbors}_i|$$

Example: PageRank

1. Start each page with rank = 1
2. For iteration = 1, …, N, update rank(page) =

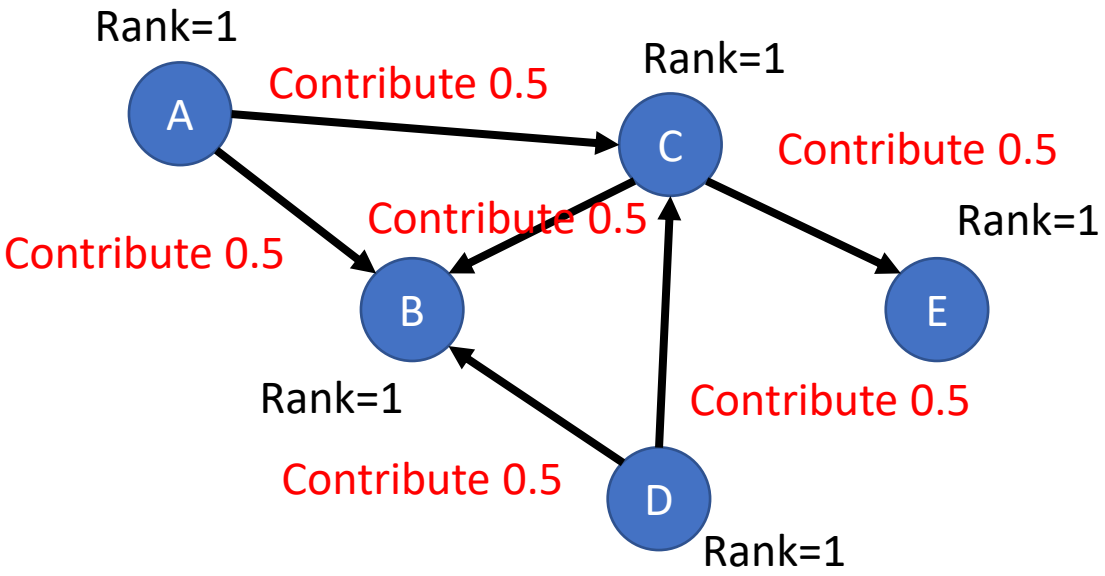
$$\sum_{i \in \text{Neighbors}} \text{rank}_i / \, |\text{Neighbors}_i|$$



Example: PageRank

1. Start each page with rank = 1
2. For iteration = 1, …, N, update rank(page) =

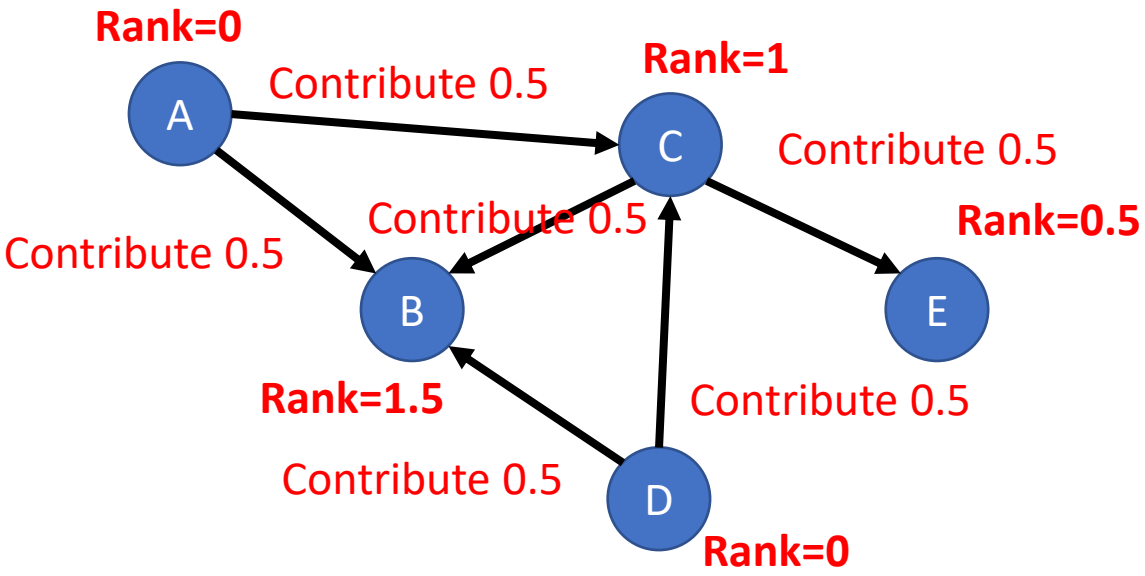
$$\sum_{i \in \text{Neighbors}} \text{rank}_i / \, |\text{Neighbors}_i|$$



Example: PageRank

1. Start each page with rank = 1
2. For iteration = 1, …, N, update rank(page) =

$$\sum_{i \in \text{Neighbors}} \text{rank}_i / \, |\text{Neighbors}_i|$$



Example: PageRank

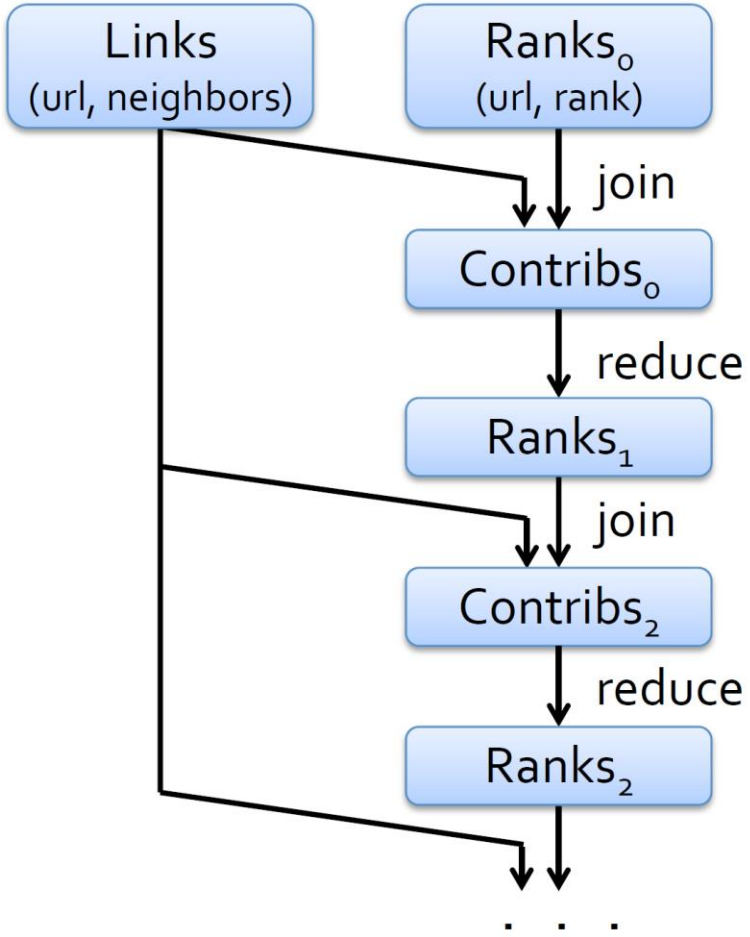
1. Start each page with rank = 1
2. For iteration = 1, …, N, update rank(page) =

$$\sum_{i \in \text{Neighbors}} \text{rank}_i / \, |\text{Neighbors}_i|$$

(This is Scala code)

```
linksRDD = // RDD of (url, neighbors) pairs
ranksRDD = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
    ranksRDD = linksRDD.join(ranksRDD).flatMap {
        (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
}
```

Example: PageRank



- Repeatedly join links & ranks
- Co-partition them (e.g., hash on url) to avoid shuffling
- Specify partitioner:

```
links = links.partitionBy(
        new URLPartitioner())
```

Spark Operations

- Transformations (define a new RDD):

map	flatMap	cartesian
filter	union	pipe
sample	join	coalesce
groupByKey	intersection	mapPartition
reduceByKey	cogroup	mapPartitionWithIndex
sortByKey	cross	repartition
aggregateByByte	distinct	repartitionAndSortWithinPartitions

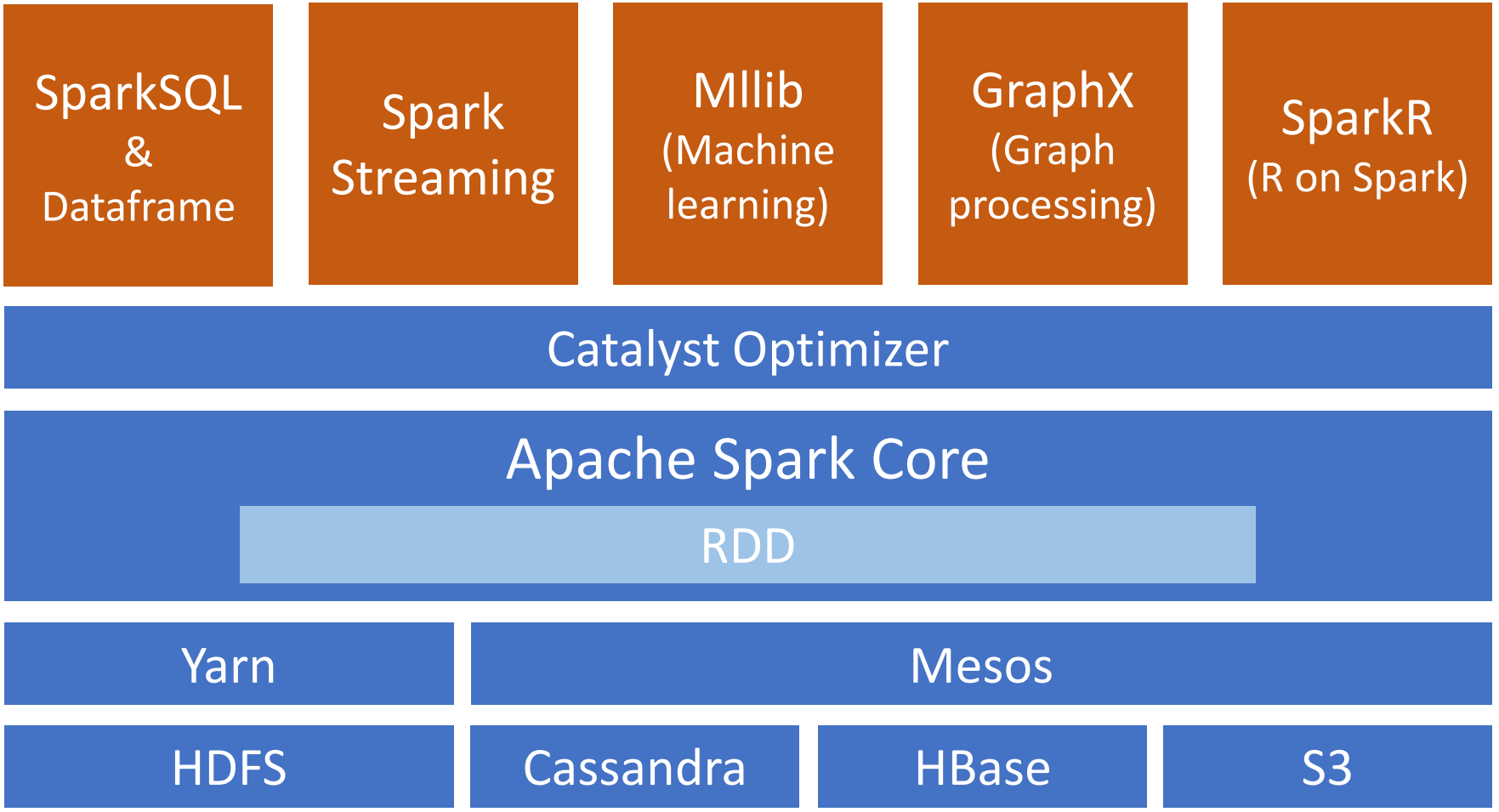
- Actions (return a result to the driver program)

reduce	takeSample	saveAsTextFile
collect	takeOrdered	saveAsSequenceFile
count	countByKey	saveAsObjectFile
first	foreach	

How General is Spark?

- RDDs can express many parallel algorithms
- Dataflow models: Hadoop, SQL, Dryad, …
- Specialized models: graph processing, machine learning, iterative MapReduce (HaLoop), …

Spark Stack

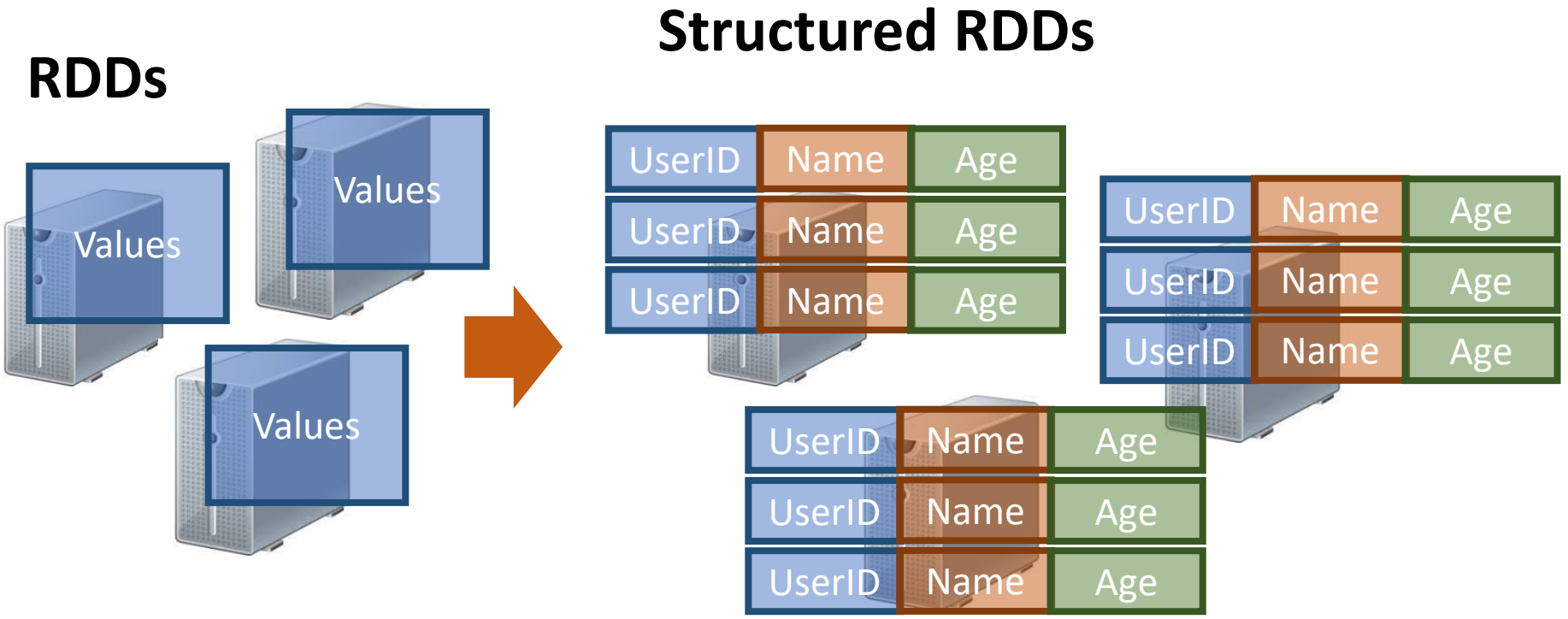

SparkSQL & Dataframe
Spark Streaming
Mllib (Machine learning)
GraphX (Graph processing)
SparkR (R on Spark)
Catalyst Optimizer
Apache Spark Core
RDD
Yarn
Mesos
HDFS
Cassandra
HBase
S3

Spark SQL

- Relational DBMS on Spark
- Using RDDs to define and partition database states
- Optimized with DBMS techniques
  - Caching
  - Query optimizers

DataFrame API

- Spark SQL uses **DataFrame** to express structured dataset with a fixed schema



DataFrame API

```
cities = spark.read.csv('cities', header=True, inferschema=True)

cities.show()
+---------+----------+-------+
|    city |population|   area|
+---------+----------+-------+
|Vancouver|   2463431|2878.52|
+---------+----------+-------+
```

DataFrame API

(This is PySpark)

```
cities = spark.read.csv('cities', header=True, inferschema=True)

cities.show()
+---------+----------+-------+
|    city |population|   area|
+---------+----------+-------+
|Vancouver|   2463431|2878.52|
+---------+----------+-------+

cities.printSchema()
root
 |-- city: string (nullable = true)
 |-- population: integer (nullable = true)
 |-- area: double (nullable = true)
```

DataFrame API

(This is PySpark)

```
cities = spark.read.csv('cities', header=True, inferschema=True)

small_cities = cities.where(cities['area'] < 5000)
density = small_cities(cities['population'] / cities['area'])

density.show()
+-------------------+
+(population / area)|
+-------------------+
|         855.797710|
+-------------------+
```

RDD

RDD

Lazy evaluation:

operations are processed when outputting results

Limitations

- DataFrame API provides many built-in functions
  - Math: abs(), cos(), avg(), ceil(), …
  - Data processing: asc(), base64(), …
  - Aggregation: array(), array_contains(), …
  - String: concat(), substring(), …
- You can't directly use Scala/Python functions because they are not DataFrame API
- What to do? Write **User Define Functions (UDFs)**

User-Defined Function (UDF)

```
@functions.udf(returnType=types.DoubleType())
def my_logic(a, b):
    return a + 2 * b * math.log2(a)


res = df.select(my_logic(df['a'], df['b']),alias('res'))
```

UDF

Including UDF in the expression

User-Defined Function (UDF)

```
@functions.udf(returnType=types.DoubleType())
def my_logic(a, b):
    return a + 2 * b * math.log2(a)
```

UDF

```
res = df.select(my_logic(df['a'], df['b']),alias('res'))
```

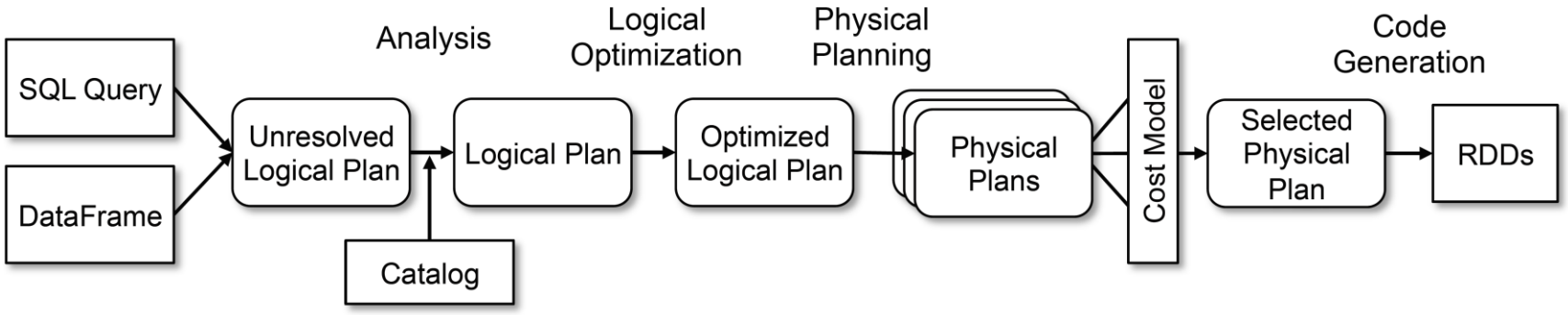
Including UDF in the expression

UDFs are much slower than DataFrame API

	Time
Spark DataFrame API	10 s
Python UDF	437 s

Query Optimization

- RDBMS usually has a query optimizer to find a more ideal version of a query
  - Example: (X + 1) + (Y + 2) ➔ X + Y + 3
- Spark SQL: **Catalyst optimizer**
  - Transforming a query into equivalent logics
  - Selecting the ideal physical plan based on cost model



References

- Spark class hosted by Stanford ICME: http://stanford.edu/~rezab/sparkclass/
- RDD Talk in NSDI (by Zaharia): https://www.usenix.org/sites/default/files/conference/protected-files/nsdi_zaharia.pdf