CS422
Database systems

Today: Execution models for distributed computing – 2nd generation

Data-Intensive Applications and Systems (DIAS) Laboratory
École Polytechnique Fédérale de Lausanne

Last week

- Hadoop/MapReduce and architectural choices
- The MapReduce opponents

This week

- The Spark ecosystem – architectural choices
- The Spark SQL interface
- Problems with skew
  - Both Spark & MapReduce

Problems with MapReduce

- Performance: Extensive I/O
  - Everything is a file stored in hard disk
- Programming model: Limited expressiveness
  - E.g., iterations, cyclic processes
  - Procedural code, difficult to optimize

MapReduce is a major step backwards

DeWitt and Stonebreaker, 2008

The data flow model

- Construction goals
  - Improve expressiveness and extensibility of model
  - Make coding easier: strive for high-level code
  - Enable additional optimizations
  - Increase performance by better utilizing the hardware

- Representative examples

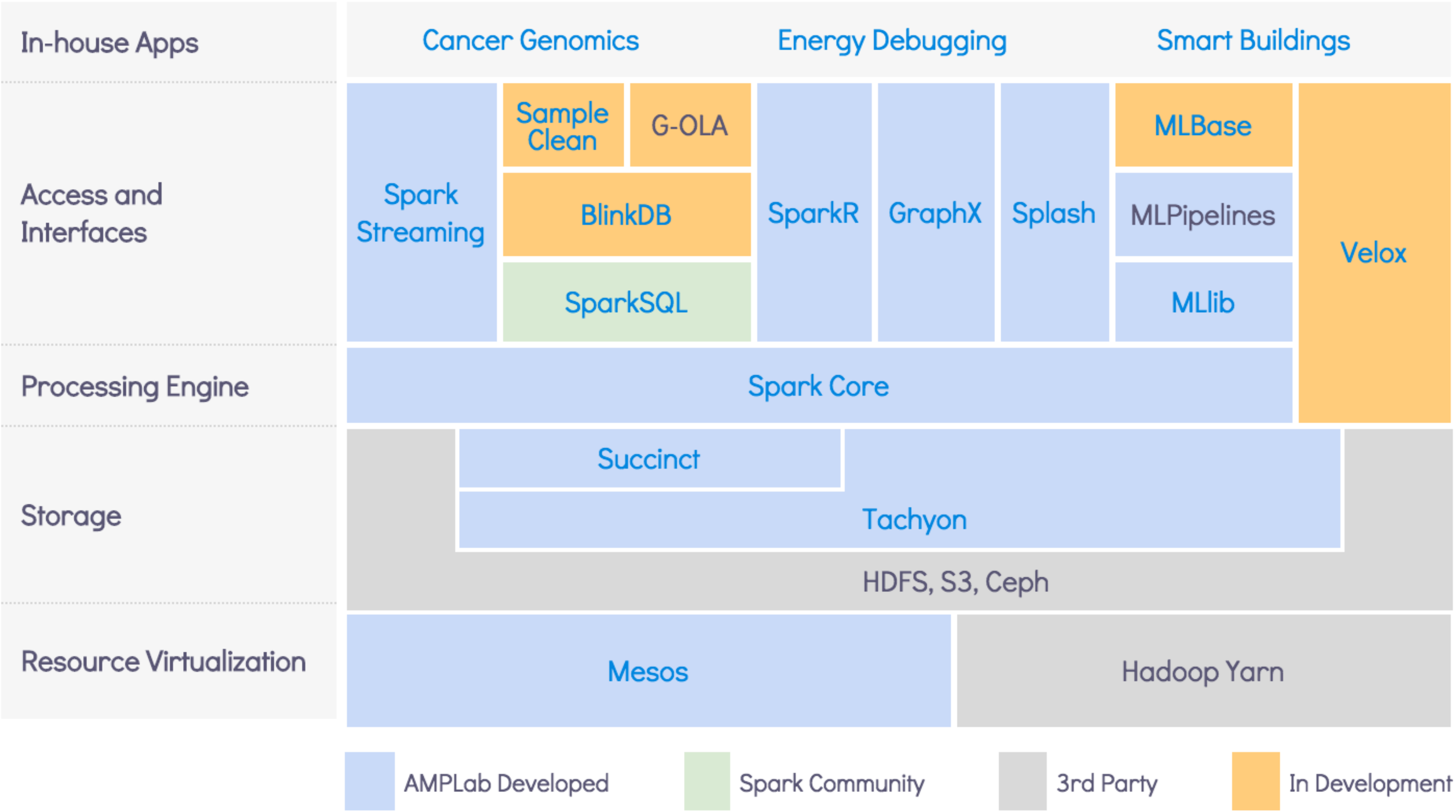
APACHE Spark

Apache Pig

Microsoft Dryad

Google Pregel

The Spark Unified Stack



Architectural choices of Spark

- Storage layer
  - Resilient distributed datasets
- Programming model & exec. engine
- Resource management
- Fault tolerance

Storage layer

- Abstraction: Developers are **NOT** reading from/writing to files explicitly. A distributed storage layer handles all data accesses and utilizes distributed RAM

- MapReduce: Everything is written to the DFS (HDD) to handle fault-tolerance and distribution

- Spark key concept: Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets

- Distributed, fault-tolerant collections of elements that can be operated in parallel
- Created by
  - Loading of data from stable storage, e.g., from HDFS
  - Manipulation of existing RDDs
- Core properties
  - Immutable
  - Distributed
  - Lazily evaluated
  - Cacheable → by default, stored in memory!
  - Replicated ***on request***

Resilient Distributed Datasets (2)

- RDDs contain
  - Details about the data
    - Data location, or data itself
  - Lineage (history) information to enable recreating a lost split of an RDD
    - Dependencies from other RDDs
    - Functions/transformations
    - E.g.,
      RDD1=sc.textFile("hdfs://...");
      RDD2=RDD1.filter(...);
      RDD3=RDD2.map(...); ...

RDDs Vs Hadoop HDFS

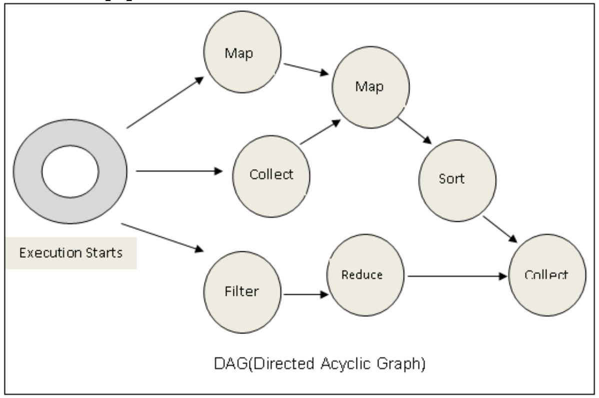
	RDDs	**Hadoop HDFS**
Performance	Exploits RAM	Everything written on disk
Fault tolerance	Lineage (& replication)	Replication
Accessing & manipulation	Functional	Procedural

Architectural choices of Spark

- Storage layer
- Programming model & exec. engine
- Resource management
- Fault tolerance

Programming models – an overview

- MapReduce simple but weak for some reqs.
  - Cannot define complex processes
  - Batch mode, acyclic, not iterative
  - Everything is file-based, no distributed memory
  - Procedural → difficult to optimize
- More powerful models
  - Spark, Dryad: processing expressed as a DAG
    - Vertices: processing tasks, edges: src:target connection
  - Dremel: Declarative (SQL-like)
  - Data models
    - Hierarchical, tabular: Dremel
    - Directed graph with cycles: Pregel



Spark DAG example

Programming models – examples

- Describing the processing tasks
  - Declarative languages, e.g., Dremel

```
SELECT DocId AS Id, COUNT(Name.Language.Code)
WITHIN Name AS Cnt FROM t
WHERE REGEXP(Name.Url, '^http');
```

  - Functional programming, e.g., Spark

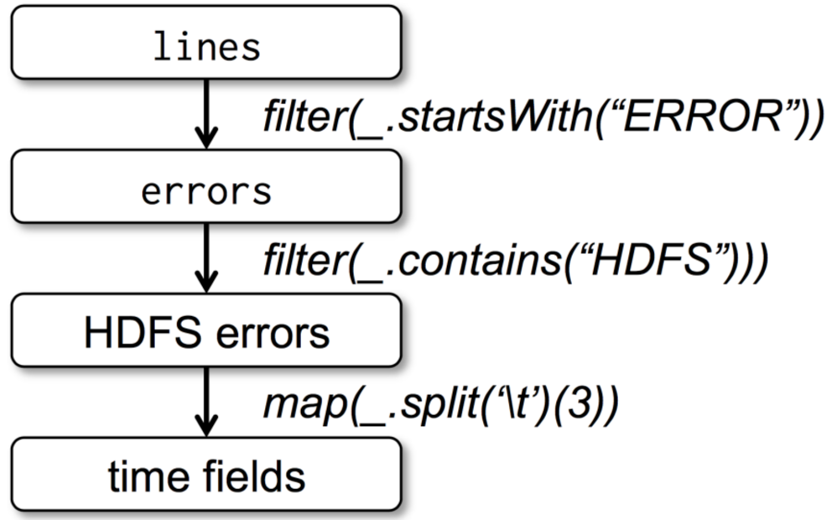
```
val wordCounts = textFile.flatMap(line => line.split(" ")).
        map(word => (word, 1)).
        reduceByKey((a, b) => a + b)
wordCounts.collect()
```

  - Domain-specific languages, e.g., Pregel

```
class PageRankVertex
  : public Vertex<double, void, double> {
     public: virtual void Compute(MessageIterator* msgs) {
        const int64 n = GetOutEdgeIterator().size();
        SendMessageToAllNeighbors(GetValue() / n);
     }
  };
```

Spark: Dataflow and RDDs

- Spark development circles around RDDs
- RDDs enable
  - Actions: count, collect, …
  - Transformations: map, filter, joins, …
- Example



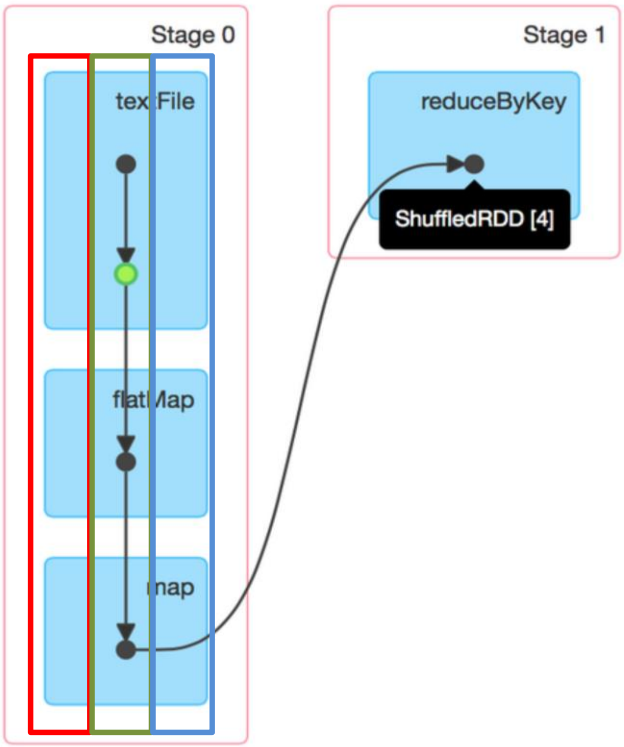
Distributed

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
hdfserrors = errors.filter(_.contains("HDFS"))
               .map(_.split('\t')(3))
hdfserrors.collect()
```

Spark: Dataflow and RDDs (2)

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line=>line.split(" "))
                     .map(word=>(word,1))
                     .reduceByKey(_+_)
```

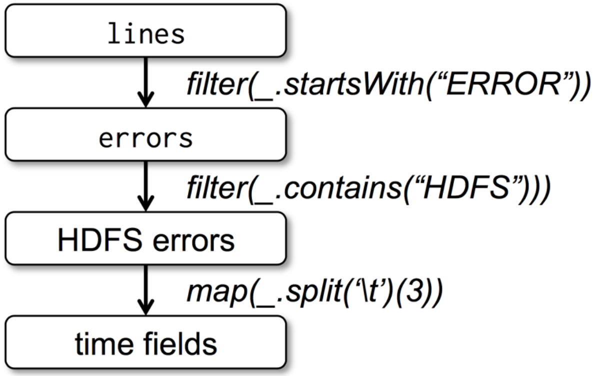
- Word-count example
  - flatMap: split line to words
  - map: emit <word,1> pairs
  - reduceByKey: sum counts for each word



3 nodes - different colors

Lazy evaluation in Spark

- Spark: Functional language → static rule-based optimizations
- Fully exploits lazy evaluation of transformations
- Example




```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
hdfserrors = errors.filter(_.contains("HDFS"))
timestamps = hdfserrors.map(_.split('\t')(3))
timestamps.collect()
```

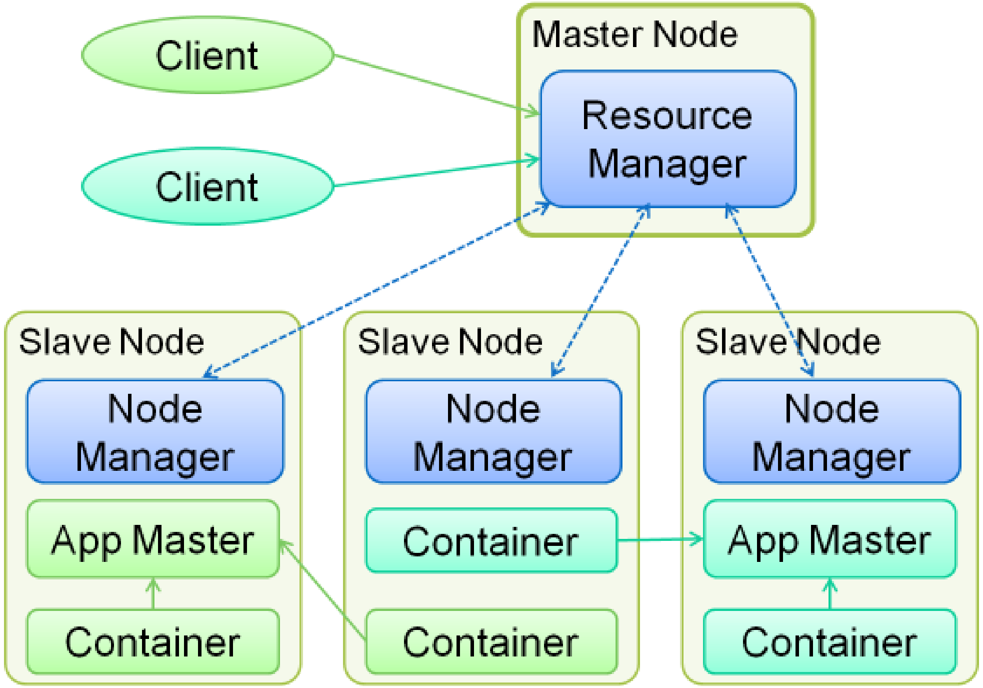
Which types of optimizations are enabled by lazy evaluation?

Design choices of Big Data systems

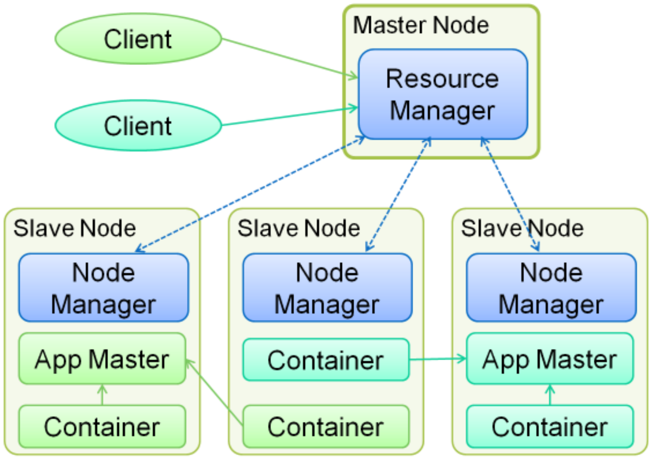
- Storage layer
- Programming model & exec. engine
- Resource management
- Fault tolerance

Resource Management

- How many map/reduce slots to assign per job
  - Early MapReduce versions – simple, within scheduler
  - Yarn/Mesos enable multiple frameworks (Spark, MapReduce, …) to co-exist in harmony over the same nodes
- Yarn: decisions of varying granularities
  - Resource manager
  - ApplicationMaster
  - Node manager
  - Containers



Resource Management (2)



- Yarn: decisions of varying granularities
  - Resource manager
  - ApplicationMaster
  - Node manager
  - Containers
- Yarn design enables
  - Co-existence of multiple frameworks in a shared & secure (isolated) manner
  - Scalability
  - Application-specific optimizations possible

Architectural choices of Spark

- Storage layer
- Programming model & exec. engine
- Resource management
- Fault tolerance

Fault tolerance

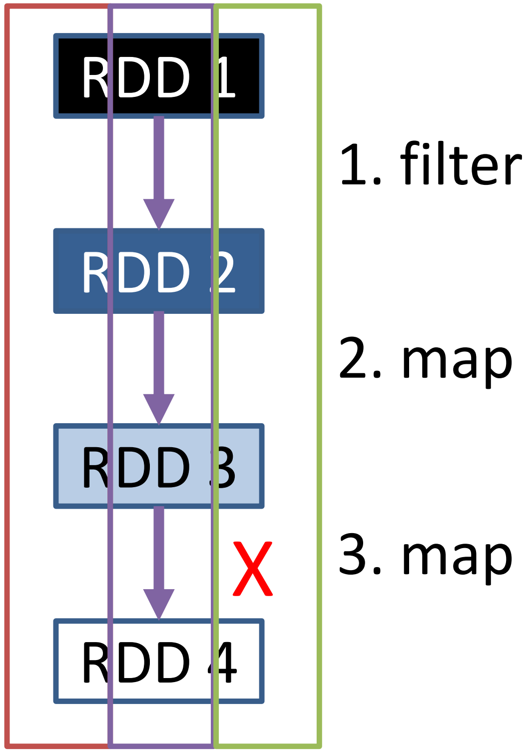
- Hardware/software failures are the rule
  - Heterogeneous hardware
  - Can be low-end, cheap, unstable
  - Network size can be in the order of thousands
  - Data skew
  - Bugs!
- Requirements
  - Data safety
  - Job recovery
    - Minimize effort – recompute as less as possible
    - Mask failures – do not delay the user!

Data safety

(Up to) three levels of data safety

- Lineage information
  - Compact and sufficient to recreate missing RDDs
  - Requires re-computation
- HDFS-inherited safety
  - Replication
- Full copies of RDDs **on request**
  - Not necessary for safety
  - Improves recovery performance

Job recovery in Spark

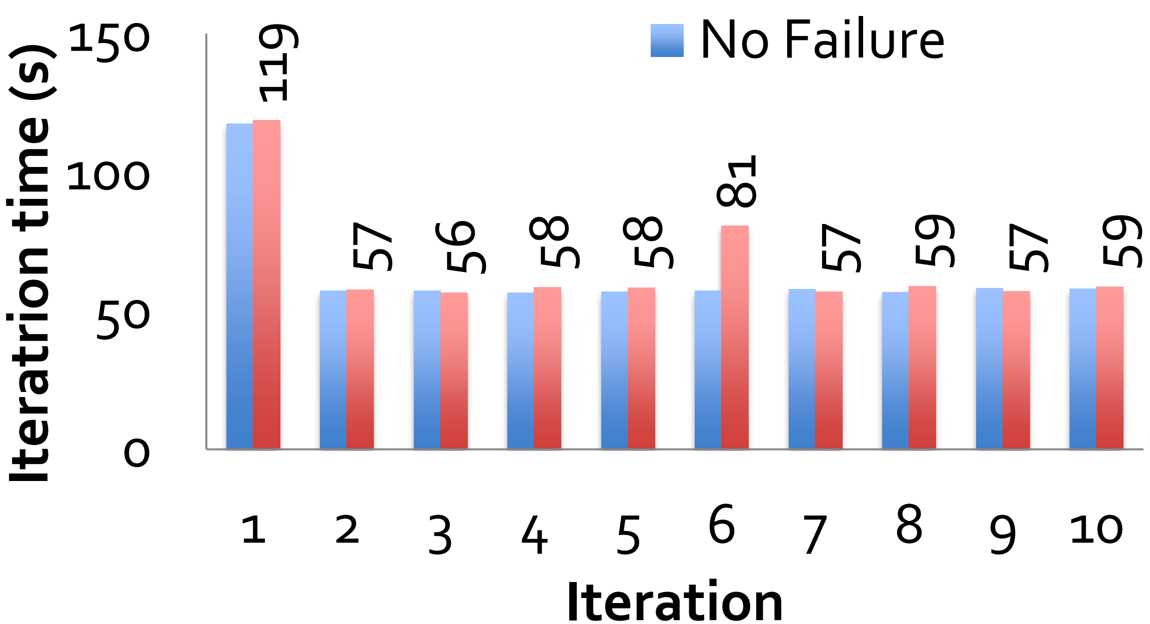


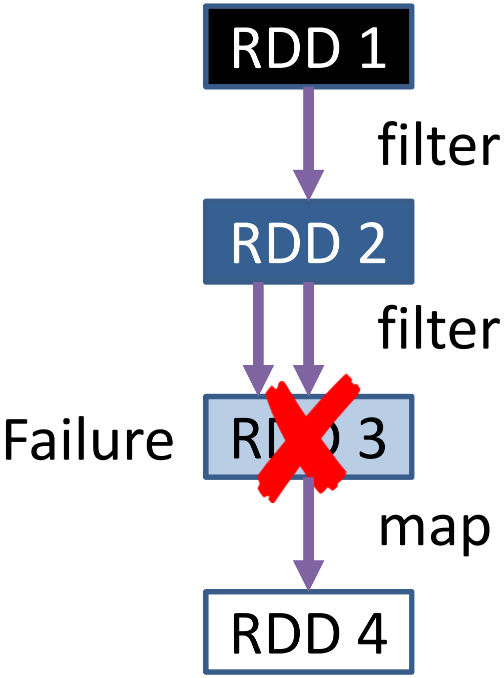
- RDDs are resilient and distributed
  - What happens if green node fails **during** step 3?
- If there is no replica of RDD3?
  - Need lineage information – history to be able to recreate RDD splits from the source

Job recovery in Spark (2)

- RDDs contain
  - data: stored in main memory whenever possible
  - lineage information: for failure resistance
    - RDD2=RDD1.filter(...), RDD3=RDD2.transform(...), ...

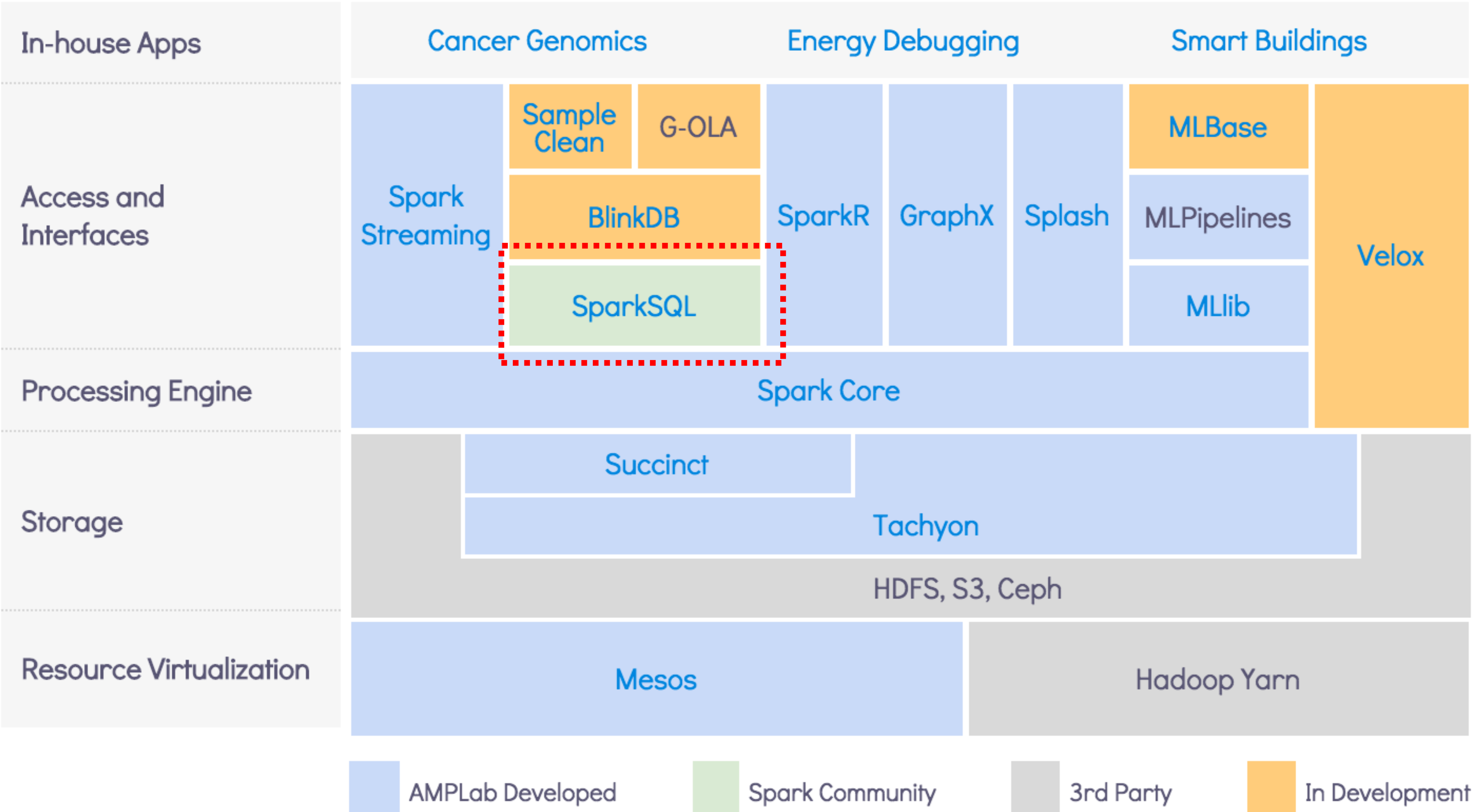
- Performance impact of failures





SPARK SQL

The Spark Unified Stack



Limitations of vanilla Spark

- RDDs are schema-less
  - Inefficient: think of accessing raw text files
  - Expensive: high space overhead

- Important optimizations not supported!
  - Access a single column → parsing to find position

- Queries cannot refer to attributes
  - Think of implementing SQL capabilities

The RDD coding nightmare!

- Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using Pig

```
P = load '/people' as (name, age);
G = group P by name;
R = foreach G generate AVG(G.age);
```

Using DataFrames

```
sqlCtx.table("people") \
   .groupBy("name") \
   .agg("name", avg("age")) \
   .collect()
```

Plan of attack

- **Add schema to RDDs**
  - Space-efficient data representation
  - Computationally-efficient access to individual attributes
- **Offer an extensible SQL-like language**
  - Easier to express queries
  - SQL-like optimizations on data accesses

And its name is…

Spark SQL

Spark SQL programming interface

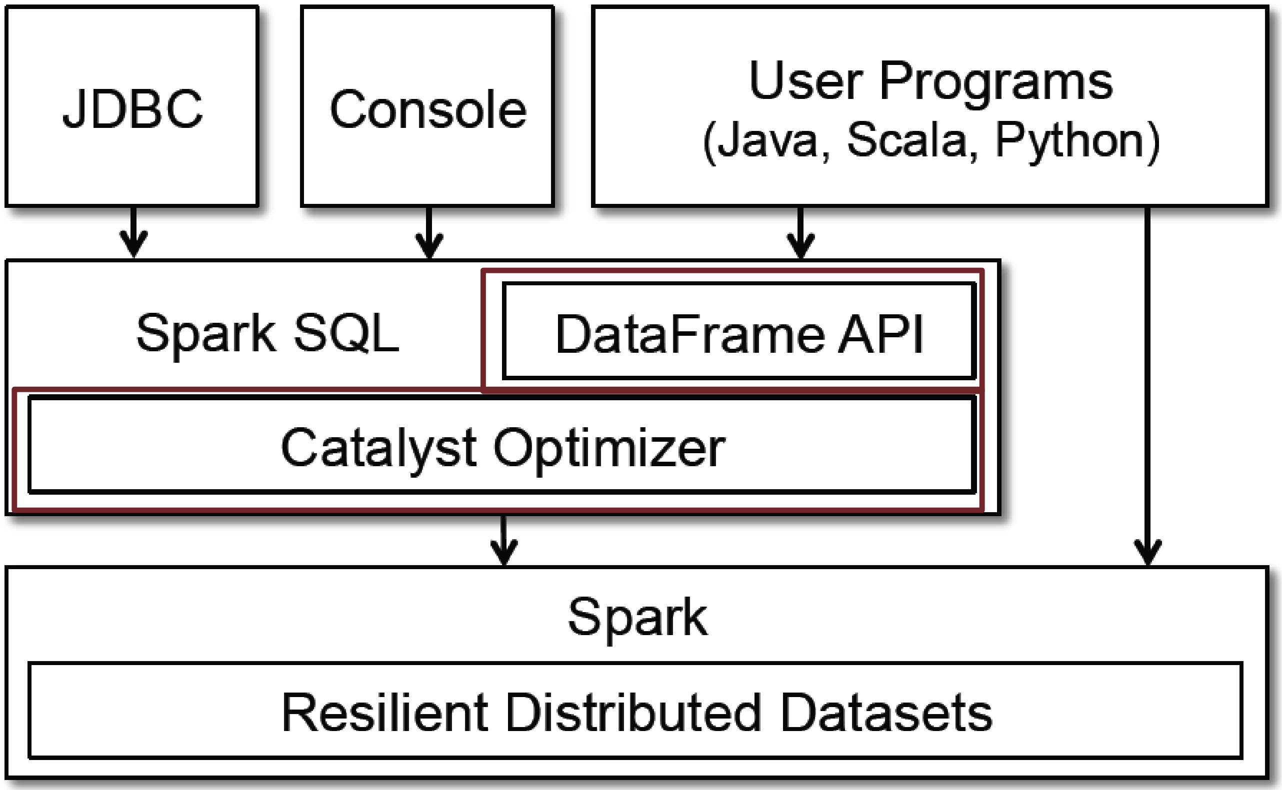


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

Data model

- Nested data model
- Supports all primitive SQL types (boolean, integer, string, …)
- Supports complex types (structs, arrays, maps, unions)
- User-defined types
- Complex data types are first-class citizens, amenable to optimizations

Key notion: Data frames

Unstructured RDD

John, 20, Developer
Marissa, 63, Manager
Bill,61, Owner
Jack,21, Developer
…

createDataFrame(…)

Data frame

name	age	role
John	20	Developer
Maria	63	Manager
Bill	61	Owner
Jack	21	Developer
…	…	…

Columnar representation

Unstructured

RDD

John, 20, Developer
Marissa, 63, Manager
Bill,61, Owner
Jack,21, Developer
…

createDataFrame(…)

Data frame

name	age	role
John	20	Developer
Maria	63	Manager
Bill	61	Owner
Jack	21	Developer
…	…	…

Columnar representation

Data frames created from
- RDDs
- Manipulation of other data frames
- Other data sources
  - relational dbs, csv files, …
  - can utilize capabilities of the data source, e.g., filtering

DataFrame operators

- Relational operations via a Domain Specific Language
  - Input: expression
  - Output: an abstract syntax tree (AST)
- Chaining of operators

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Alternatively, use traditional SQL

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

User Defined Functions (UDFs)

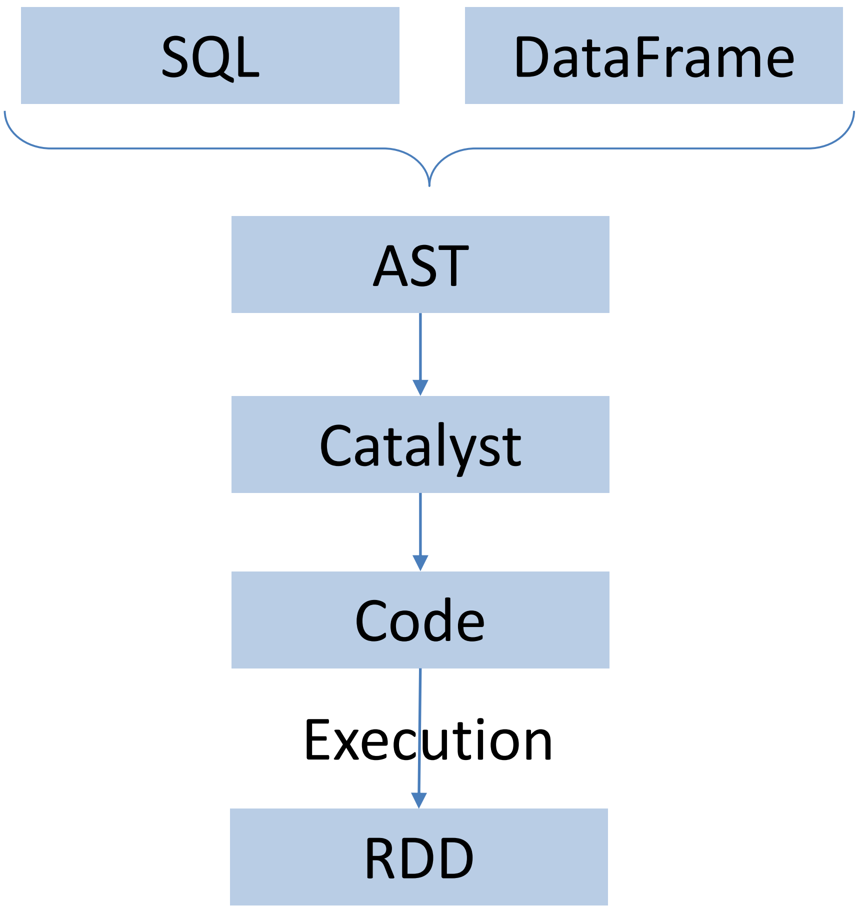
- Easy extension of supported operations
- Allows inline registration of UDFs
- Can be defined on simple data types or on entire tables
- UDFs also available to other interfaces after registration

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

Optimization principles

- Declarative language to express user's information needs
  - Write less code → let the optimizer do the hard work
- Lazy evaluation → holistic optimization
  - Get **ALL** operations and optimize them as a whole
- Code generation to avoid interpretation overhead

Catalyst optimizer



Input

- Abstract Syntax Tree

Performs

- Analysis to resolve references
- Logical optimization
- Physical planning

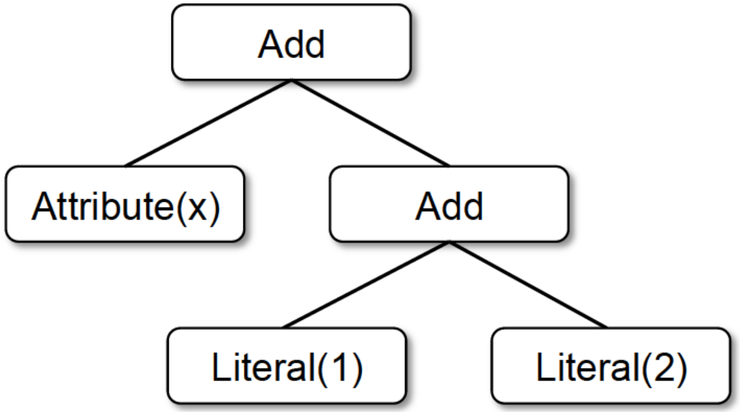
Output

- Optimized AST → Code generation

Abstract syntax trees

- ASTs represent user query
- Nodes of different types

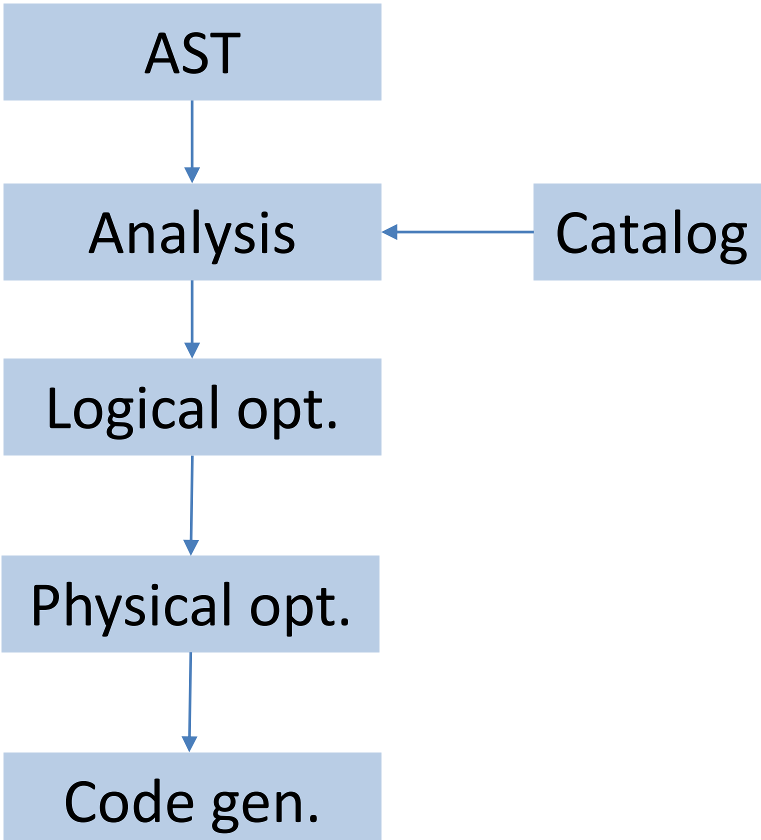
  In example:
  - Attribute(name:String)
  - Literal(value: Int)
  - Add(left: TreeNode,
    right:TreeNode)



x + (1 + 2)

- Similar to a naïve query plan

Optimization process



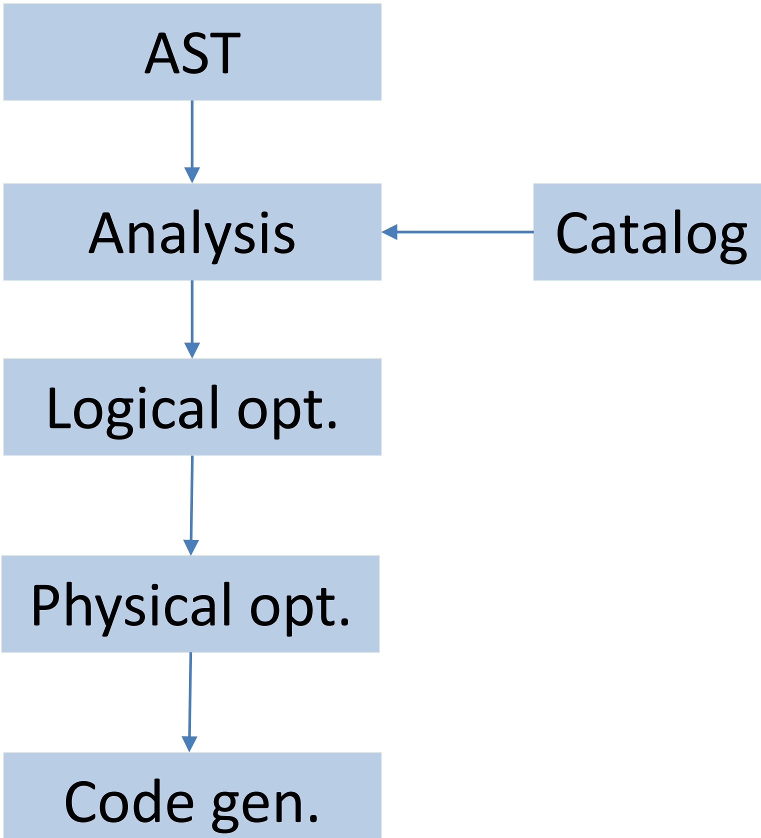
Steps

1. Analysis to resolve references
   - Find unresolved references from the catalog
   - Verify & determine types
   - Map named attributes to the input
   - E.g.,
     `SELECT name from tbl where age>10`
     (using SQL for convenience, in practice AST)

Result: Resolved logical plan, known types, known data location

Optimization process

Steps

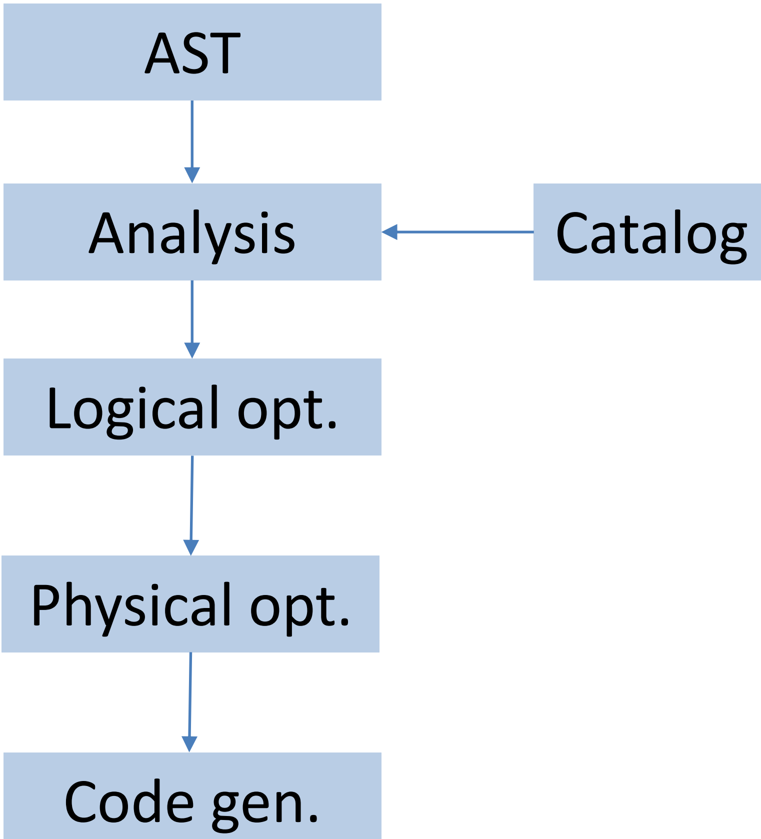


2. Logical optimization

- Standard rule-based optimization, e.g.:
  - Push-down predicates
  - Projection pruning
  - Simplify logical expressions
  - …

Result: Optimized logical plan

Optimization process



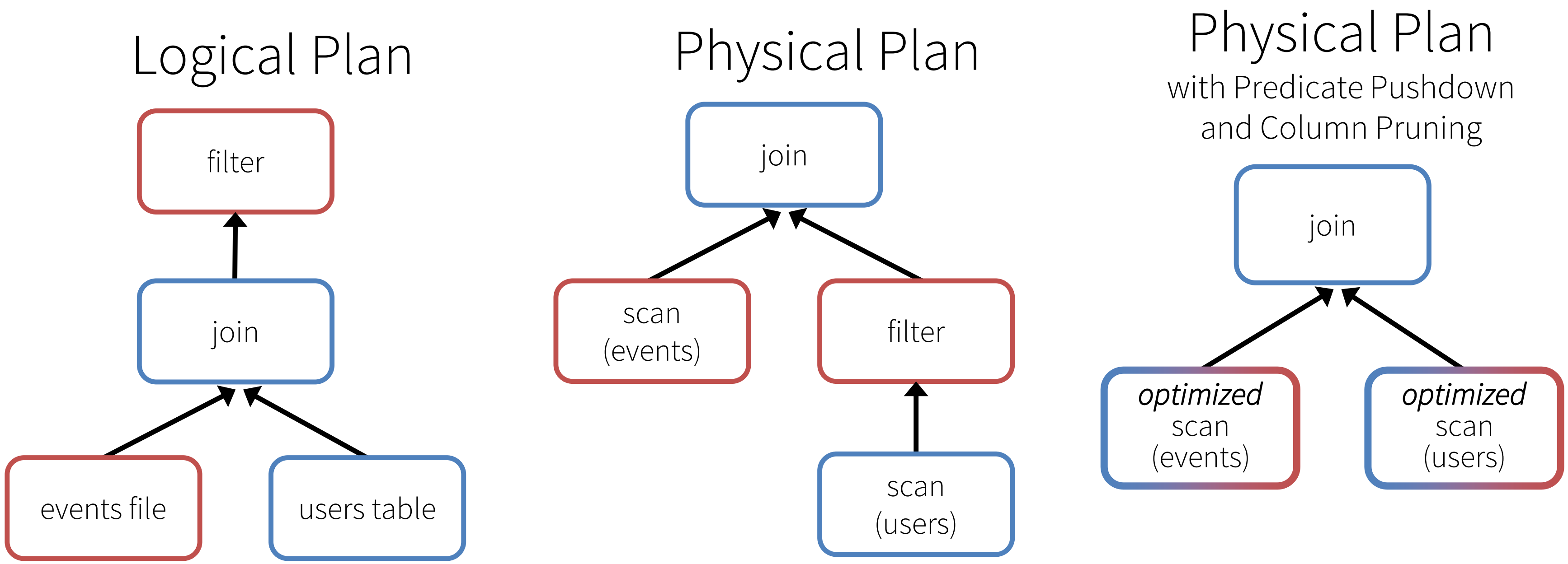
Steps

3. Physical optimization

- Generate candidate physical plans
  - Define the actual implementation of each operator, e.g., joins
  - Combine operators, e.g., filters and projections into a single map operation
  - Push operations from the logical plan into data sources, e.g., to utilize indexes

Result: Optimized physical plan

```
def add_demographics(events):
   u = sqlCtx.table("users")                     # Load partitioned Hive table
   events \
     .join(u, events.user_id == u.user_id) \     # Join on user_id
     .withColumn("city", zipToCity(df.zip))      # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```



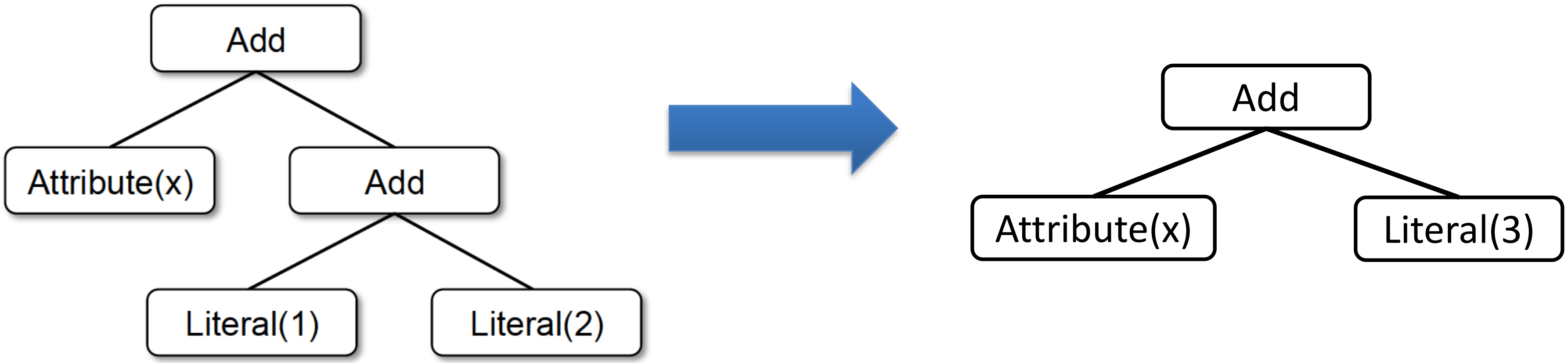
Catalyst Rules

- *Catalyst intelligence* expressed with rules
- *Pattern matching* functions that transform subtrees into specific structures.
- Multiple patterns in the same *transform* call.
  - May take multiple *calls* to reach a *fixed point*.

Transformations on ASTs

Rule-based transformations

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

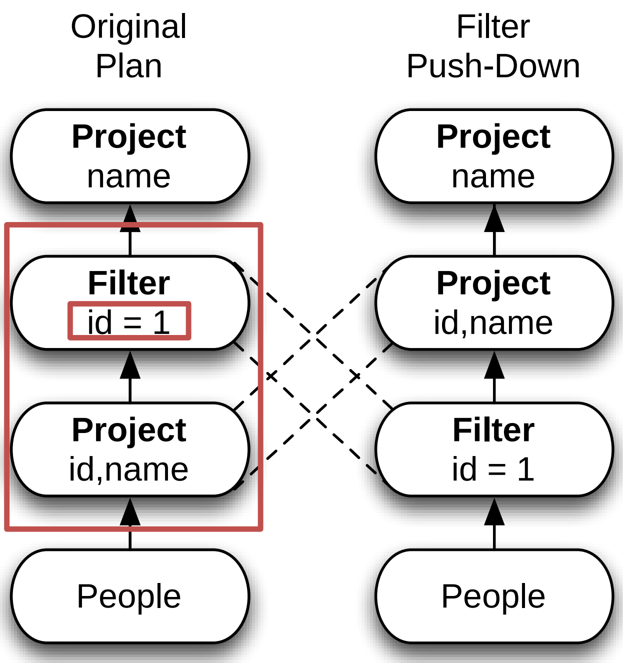


x + (1 + 2)

x + 3

Transformations on ASTs (2)

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.

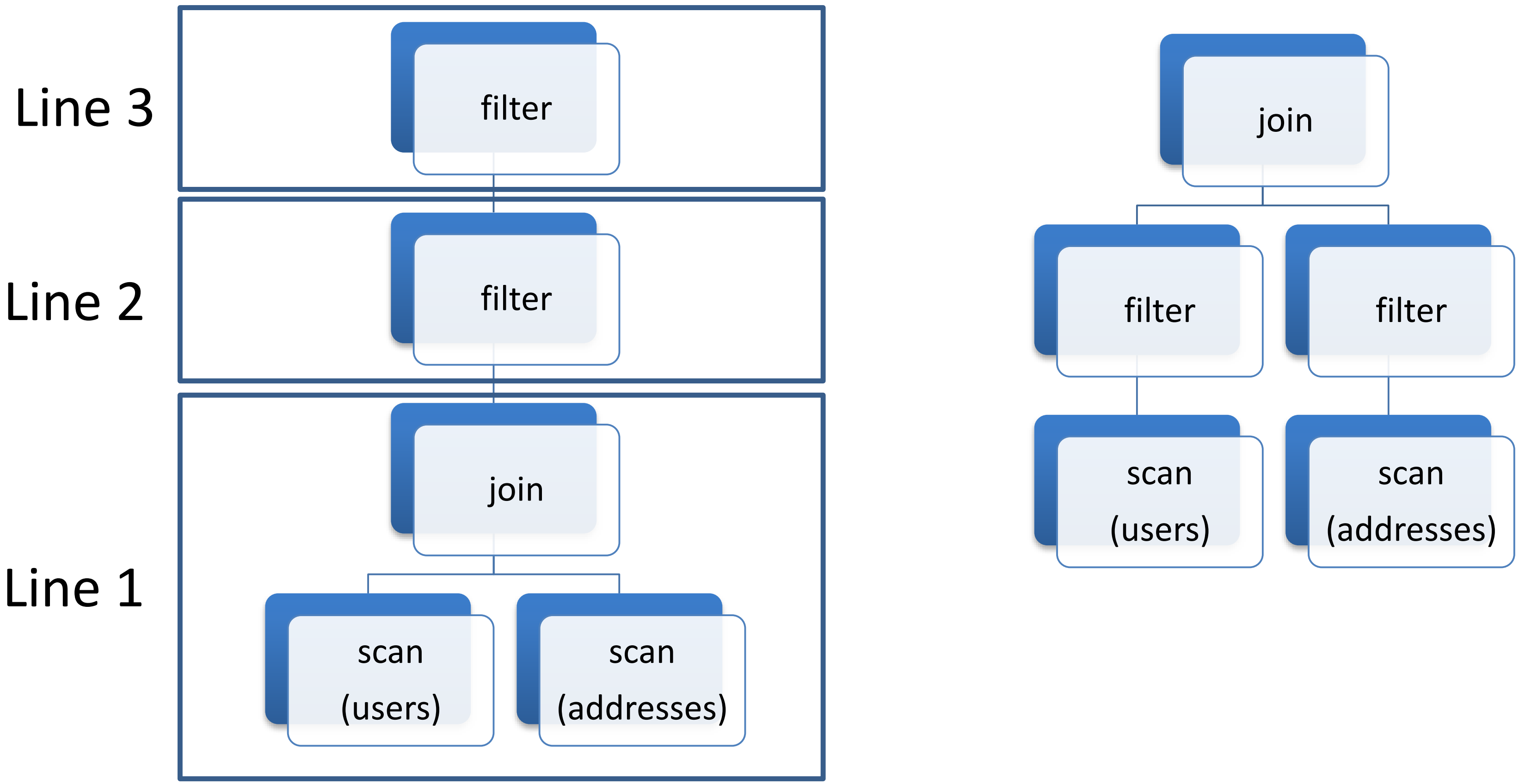


Holistic optimization

- Lazy evaluation → consider all queries together
- Optimize the batch as a whole

- Example

```
usersWithAddress = users.join(users.id===addresses.userid)
usersInLausanne = usersWithAddress.filter(city="Lausanne")
babiesInLausanne = usersInLausanne.filter(age<2).collect()
```

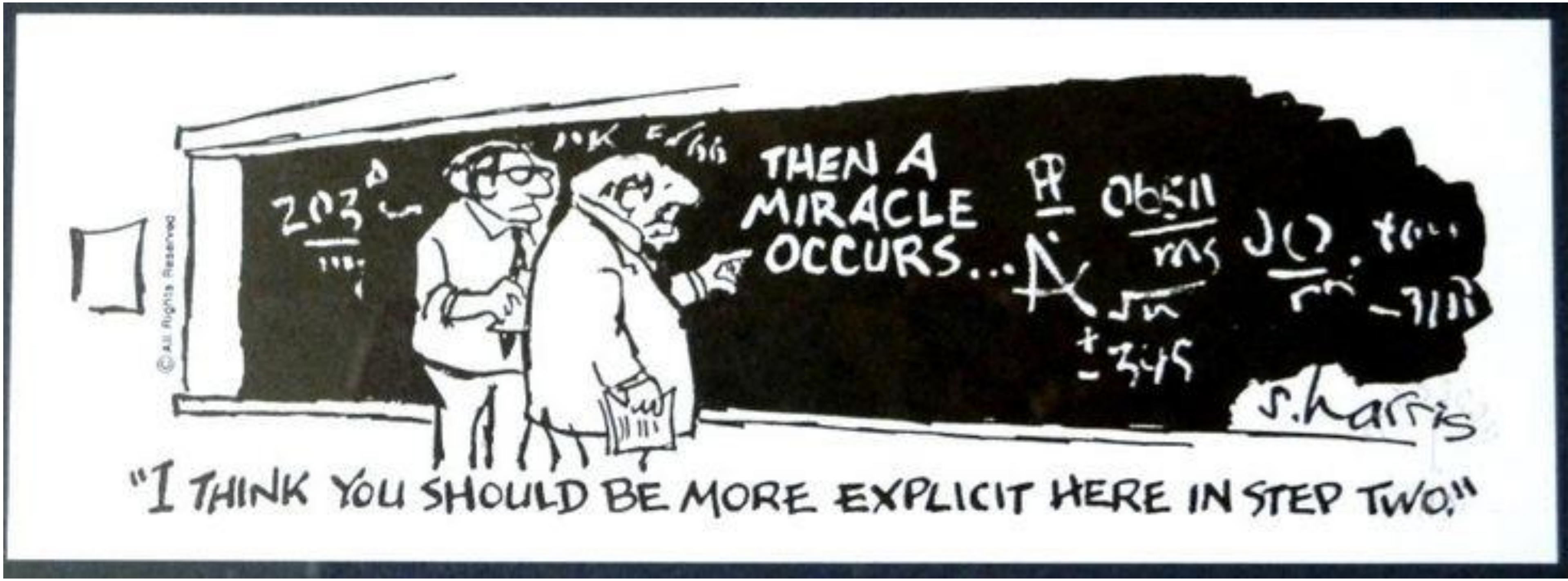


- Example

```
usersWithAddress = users.join(users.id===addresses.userid)
usersInLausanne = usersWithAddress.filter(city="Lausanne")
babiesInLausanne = usersInLausanne.filter(age<2).collect()
```

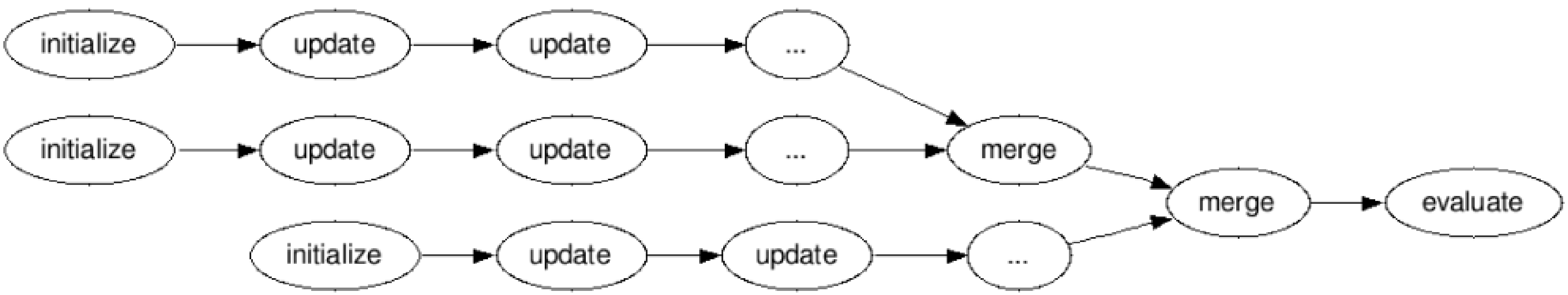
Optimizing UDFs

- UDFs are black boxes
  - For Catalyst, black boxes are **bad** boxes!



Optimizing UDFs

- Making UDAFs optimizable
  - User Defined Aggregate Functions
- Define Initialize, Update, Merge, Evaluate
- Can distribute computation & reduce network



Advantages of model over SQL

- Easier, more flexible language
  - Easy to add control structures (e.g., if, for, …)
  - More intuitive to perform complex operations

- Holistic optimization across functions composed in different languages


OTHERSIDE
RED HOT CHILI PEPPERS

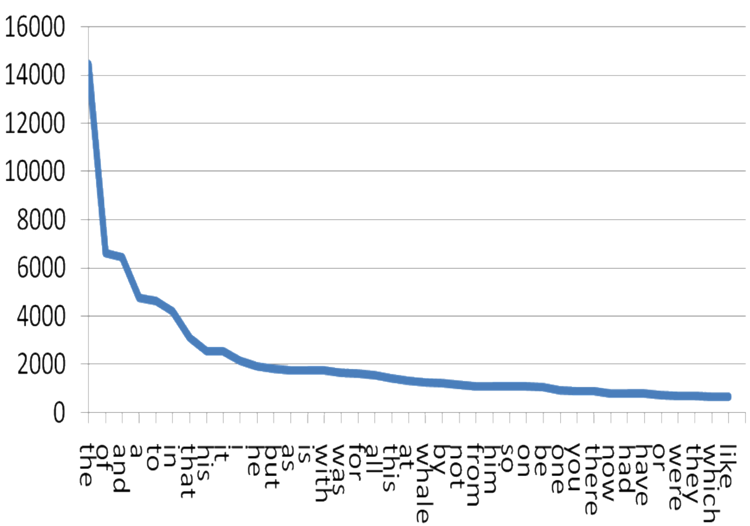
Handling skew

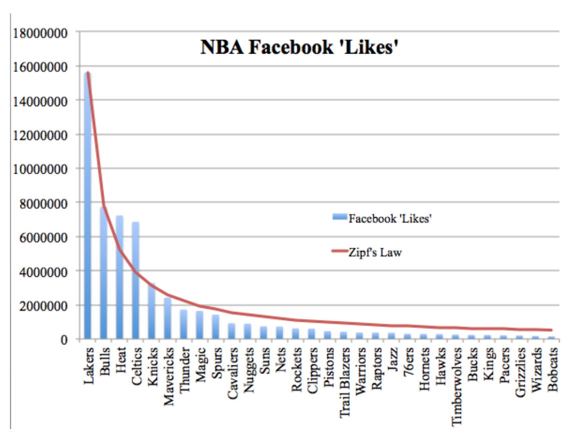
Goal: minimize job completion time

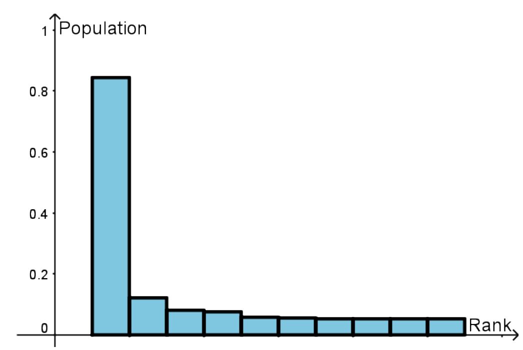
- Available hardware
- Algorithm's degree of parallelism (DoP)
  - Need to be able to parallelize the execution
  - MapReduce & Dataflow models promote high DoP (with assumptions)
- Robustness to skewed data
  - Skewness on **reduction key** affects performance
  - Highly skewed data → many keys end up at the same node
  - Overloaded nodes become the bottleneck

Problems with skew

- Typical partitioning: hash-based
  - Uniform distribution of reduction keys → uniform load at the reducers
- Real data typically follows **skewed** distribution
  - Word "the" occurs far more frequently compared to word "arachnophobia" in a book





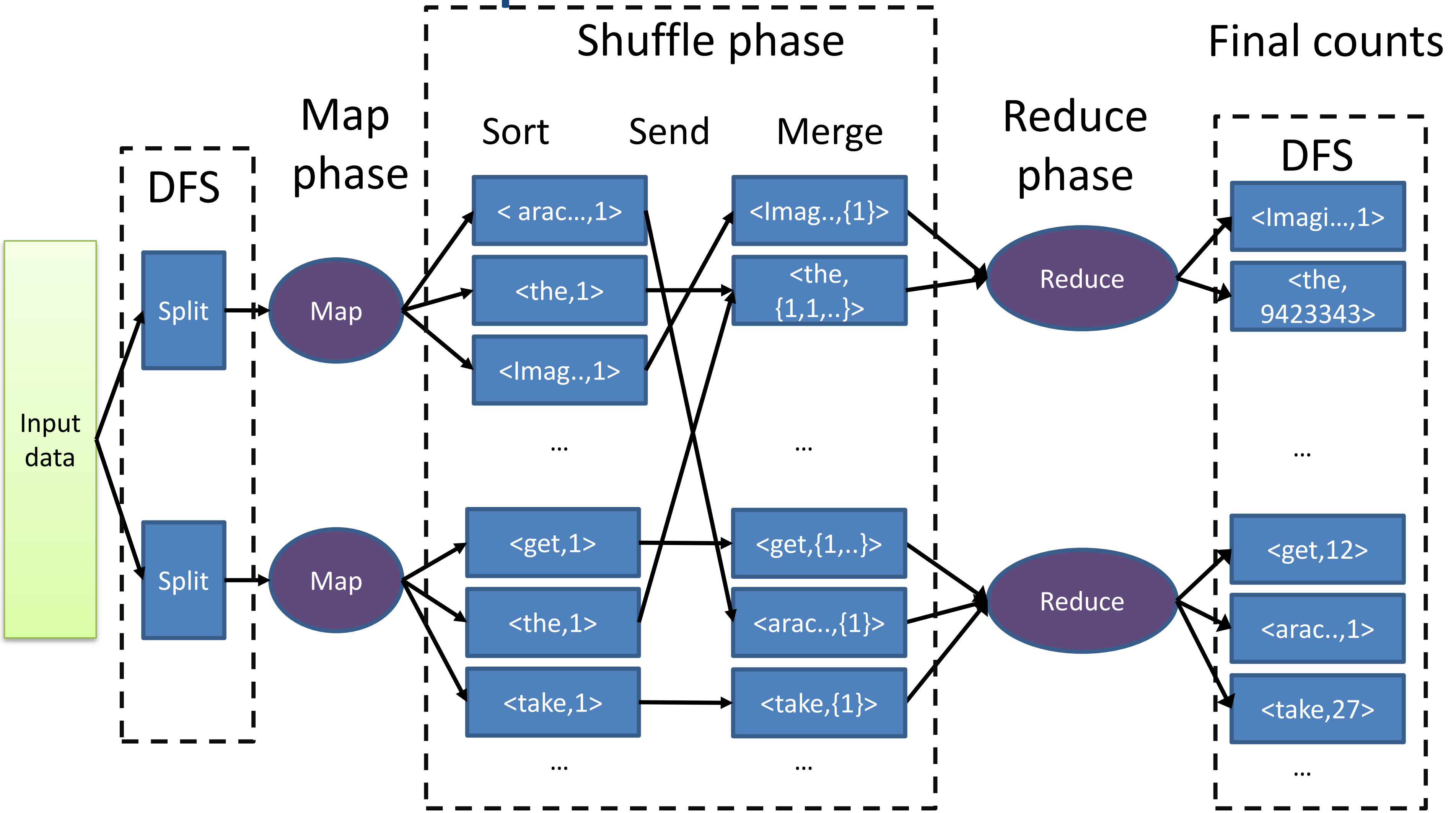


Problems with skew

- Typical partitioning: hash-based
  - Uniform distribution of reduction keys → uniform load at the reducers
- Real data typically follows **skewed** distribution
  - Word "the" occurs far more frequently compared to word "arachnophobia" in a book
  - Number of customers of each company, size of cities, in-links of websites, Facebook likes, …
- Implications for efficient processing

Skew problem underlying both Spark and MapReduce

Warm-up: word-count in MR

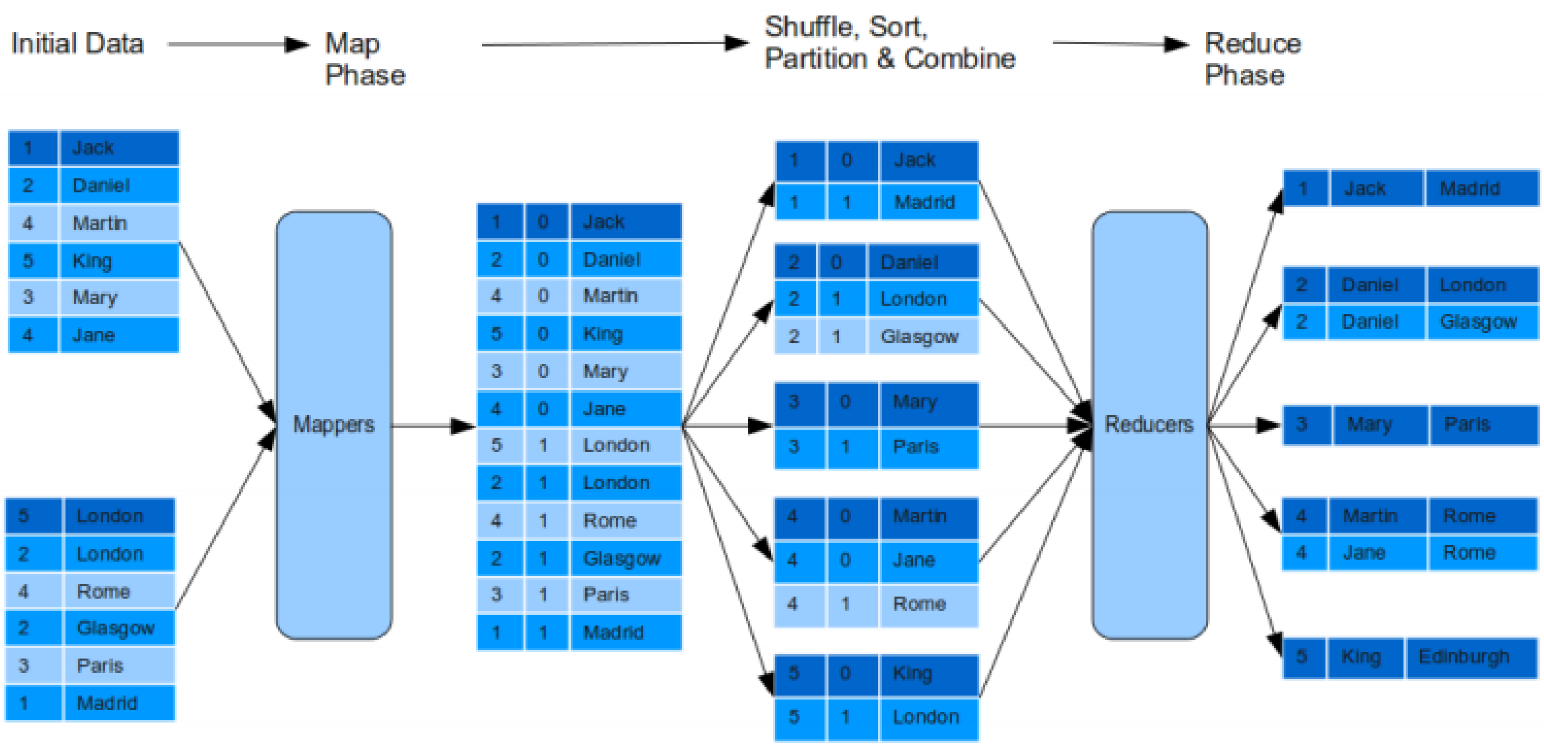



```
for each word in value
    emit pair <word,+1>
```

```
for each value in list(values)
    count+=value
emit pair<word,count>
```

How about joins?

- Join condition: S.A = T.A
- Standard implementation
  - Map(s) = (s.A, s); Map(t) = (t.A, t)
  - Reduce combines S-tuples and T-tuples with the same key



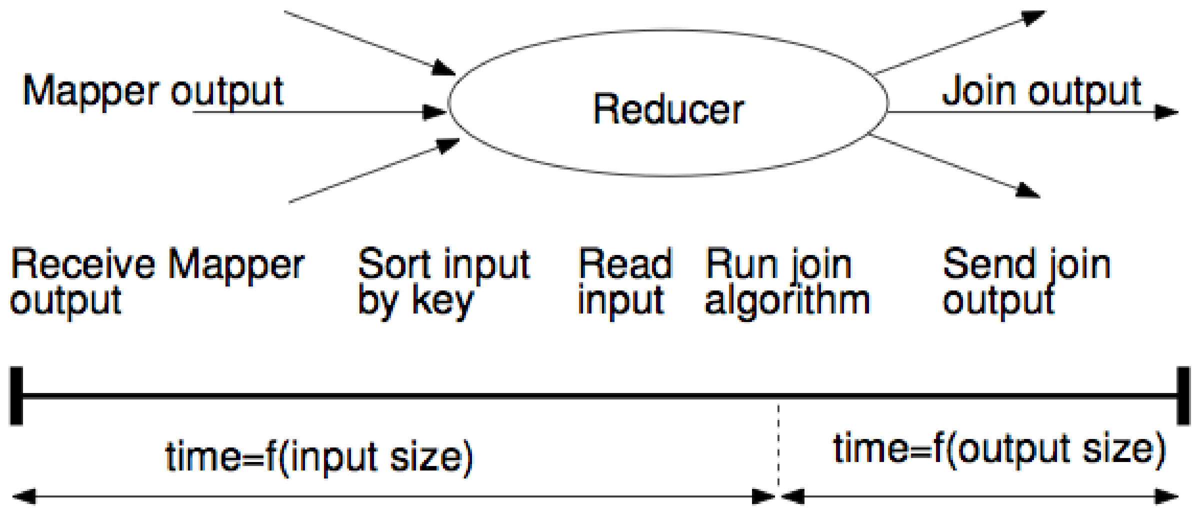
Problems With Standard Approach

- Degree of parallelism limited by the number of distinct A-values
- Data skew
  - If one A-value dominates, the reducer processing that key will become bottleneck
- Does not generalize to other joins
  - Theta-joins: S.A < T.A, |S.A-T.A|<2, …

Alper Okcan et.al.: **Processing theta-joins using MapReduce**. SIGMOD Conference 2011.

Reducer-Centric Cost Model

- Difference between join implementations starts with Map output
- Processing time at reducer is approximately monotonic in input and output size

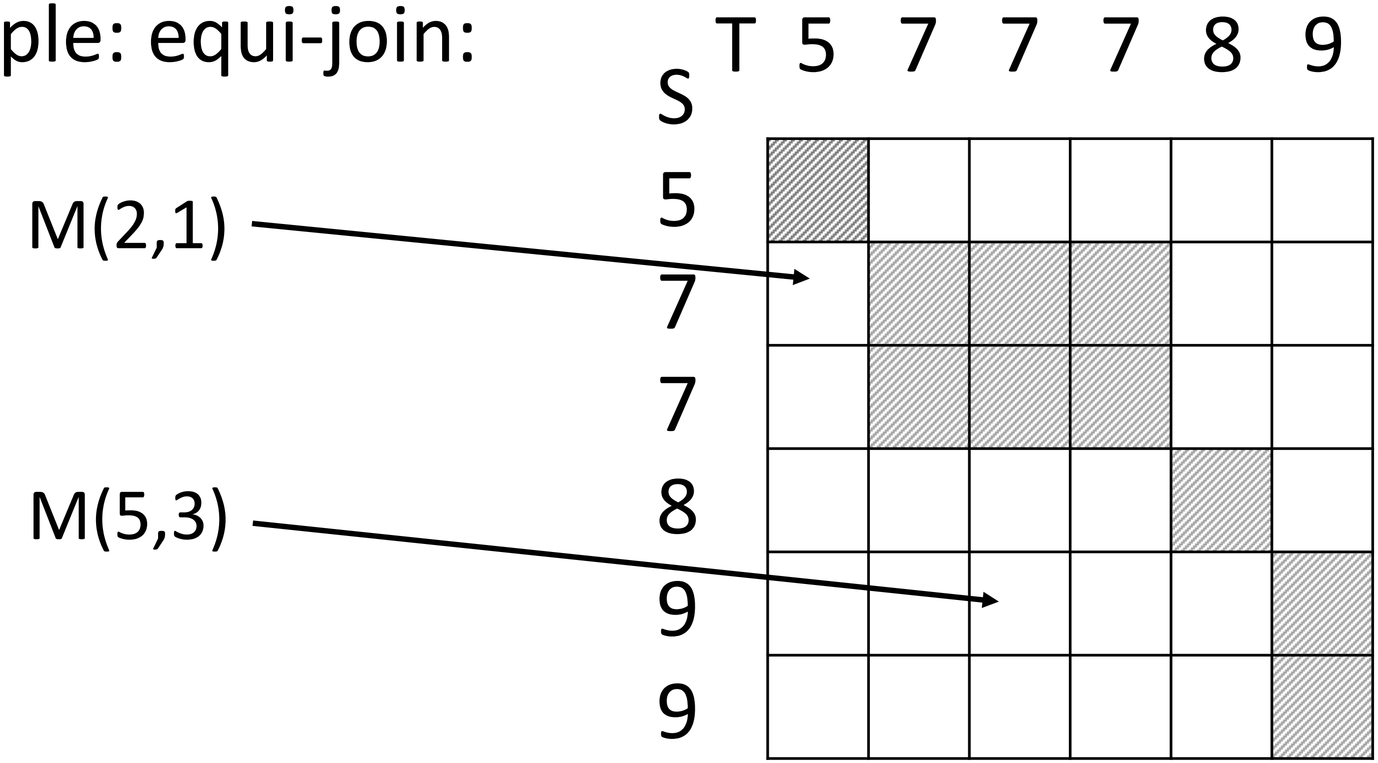


Optimization Goal Revisited

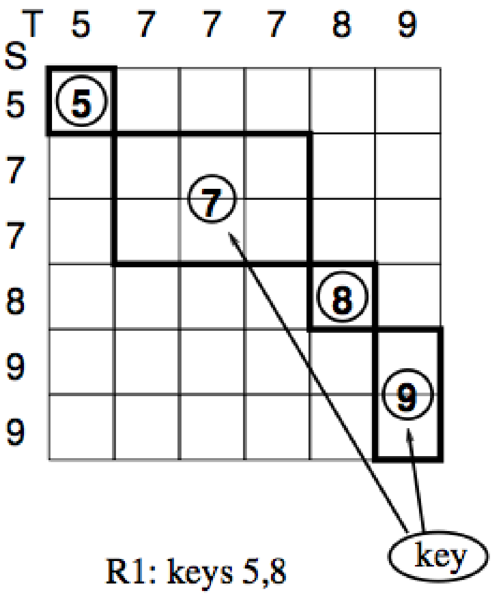
- Assume all reducers are similarly capable
- Need to minimize:
  - Max-reducer-input and/or
  - Max-reducer-output
- Join problem classification
  - Input-size dominated: minimize max-reducer-input
  - Output-size dominated: minimize max-reducer-output
  - Input-output balanced: minimize combination of both

Partitioning the load

- Join-matrix M: M(i,j) = true, iff (s_i, t_j) is in join result
- Cover each true-valued cell by one reducer
- Example: equi-join:



Standard Equi-Join Alg.



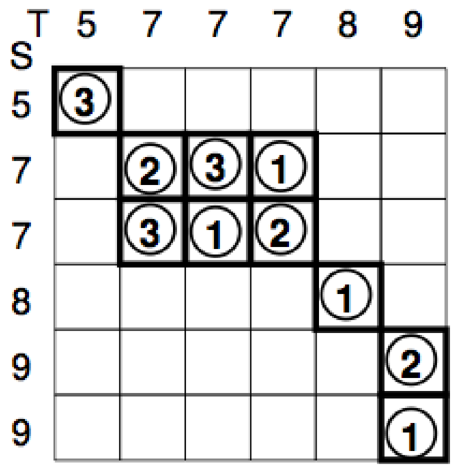
R1: keys 5,8
Input: S1,S4
T1,T5
Output: 2 tuples

R2: key 7
Input: S2,S3
T2,T3,T4
Output: 6 tuples

R3: key 9
Input: S5,S6
T6
Output: 2 tuples

max–reducer–input = 5
max–reducer–output = 6

Random Assignment



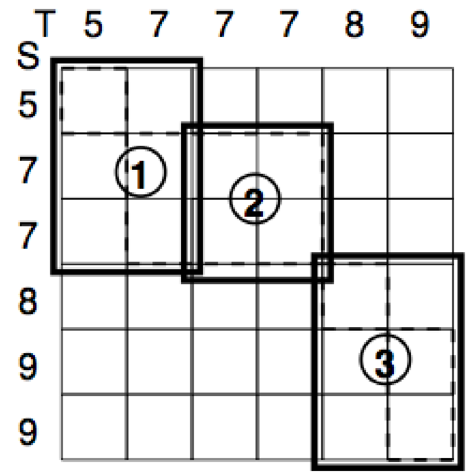
R1: key 1
Input: S2,S3,S4,S6
T3,T4,T5,T6
Output: 4 tuples

R2: key 2
Input: S2,S3,S5
T2,T4,T6
Output: 3 tuples

R3: key 3
Input: S1,S2,S3
T1,T2,T3
Output: 3 tuples

max–reducer–input = 8
max–reducer–output = 4

Balanced Algorithm



R1: key 1
Input: S1,S2,S3
T1,T2
Output: 3 tuples

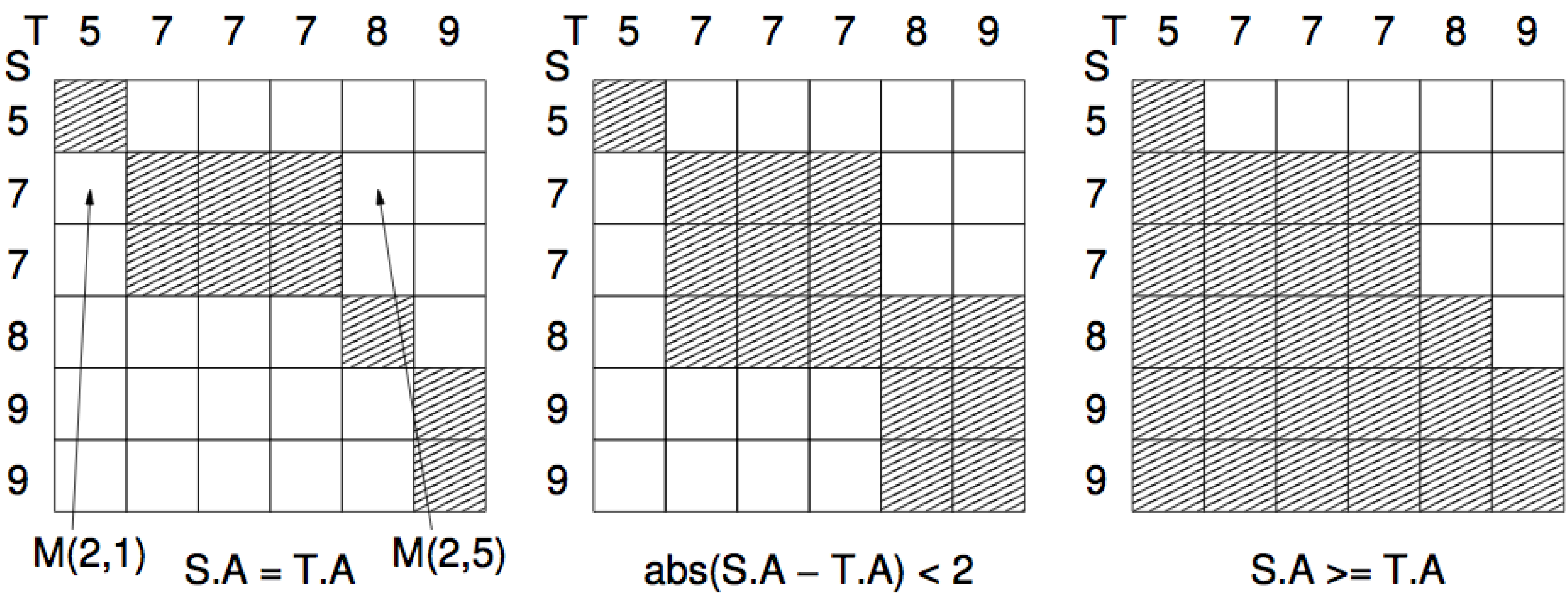
R2: key 2
Input: S2,S3
T3,T4
Output: 4 tuples

R3: key 3
Input: S4,S5,S6
T5,T6
Output: 3 tuples

max–reducer–input = 5
max–reducer–output = 4

Theta-Join Model

- Join-matrix M: M(i,j) = true, iff (s_i, t_j) is in join result



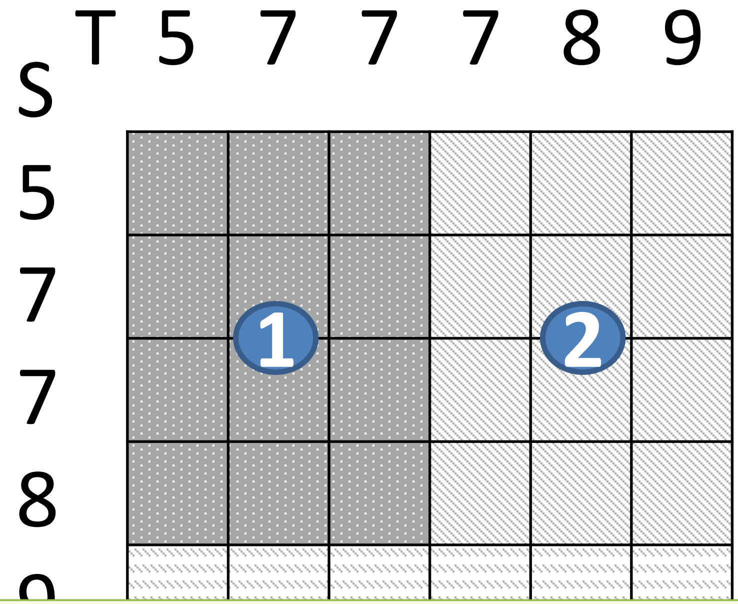
A first attempt: Cartesian product

- Cartesian product: S.A $\times$ T.A
- Assume r=3 (3 reducers)

reducers

S	T
5 $\rightarrow$ (1),(2)	5 $\rightarrow$ (1),(3)
7 $\rightarrow$ (1),(2)	7 $\rightarrow$ (1),(3)
7 $\rightarrow$ (1),(2)	7 $\rightarrow$ (1),(3)
8 $\rightarrow$ (1),(2)	7 $\rightarrow$ (2),(3)
9 $\rightarrow$ (3)	8 $\rightarrow$ (2),(3)

Matrix-to-reducer mapping

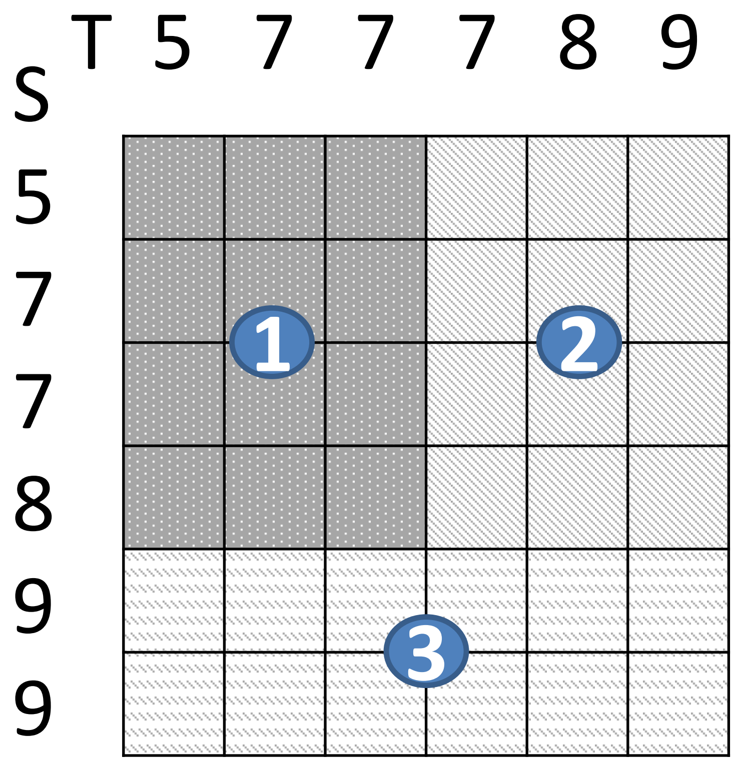


Can apply any join algorithm at the reducers

Discussion on first attempt

- Correct & complete results (why?)
- Each result is produced exactly once
- Uniform load

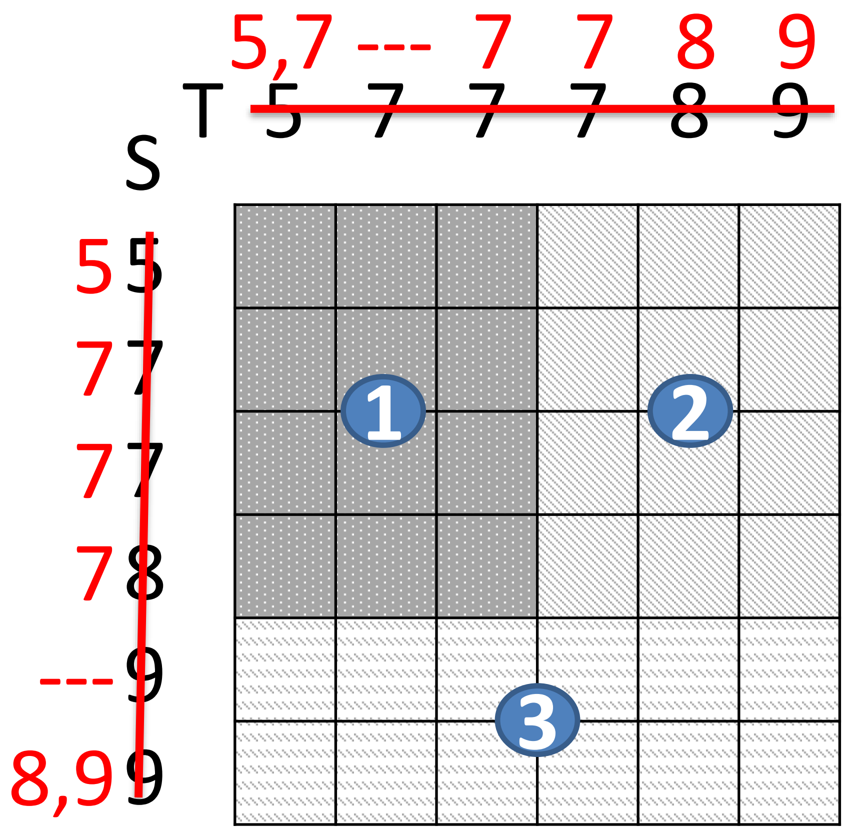
- Need to map each S tuple to a row & each T tuple to a column → requires pre-processing



1-Bucket-Theta algorithm

Observation 1

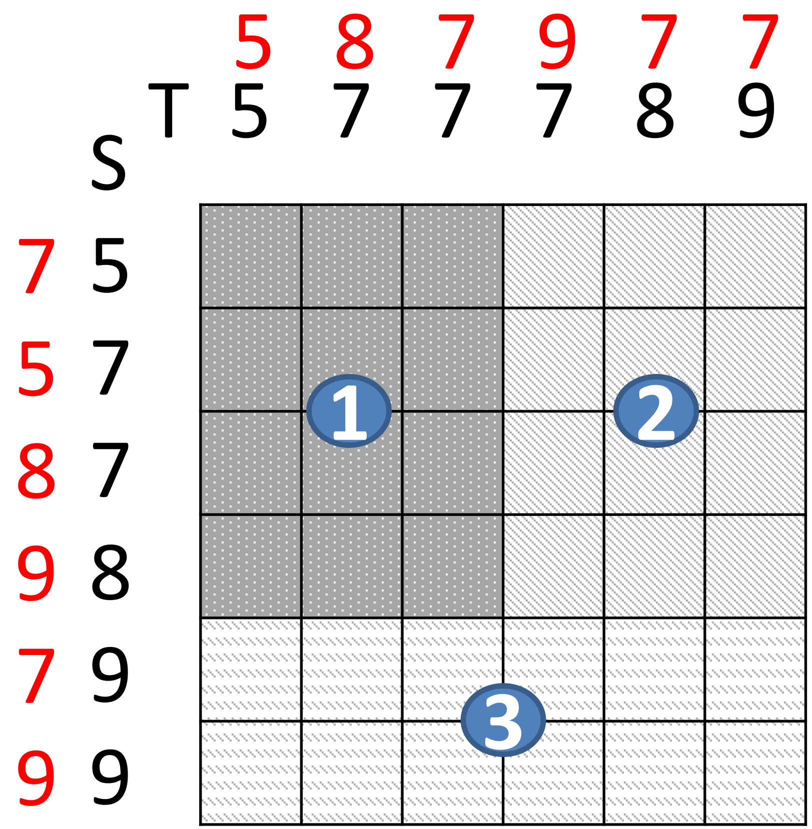
- Correctness/completeness of results unaffected if a row/column contains <>1 tuples



1-Bucket-Theta algorithm

Observation 2

- Order of the columns is not important



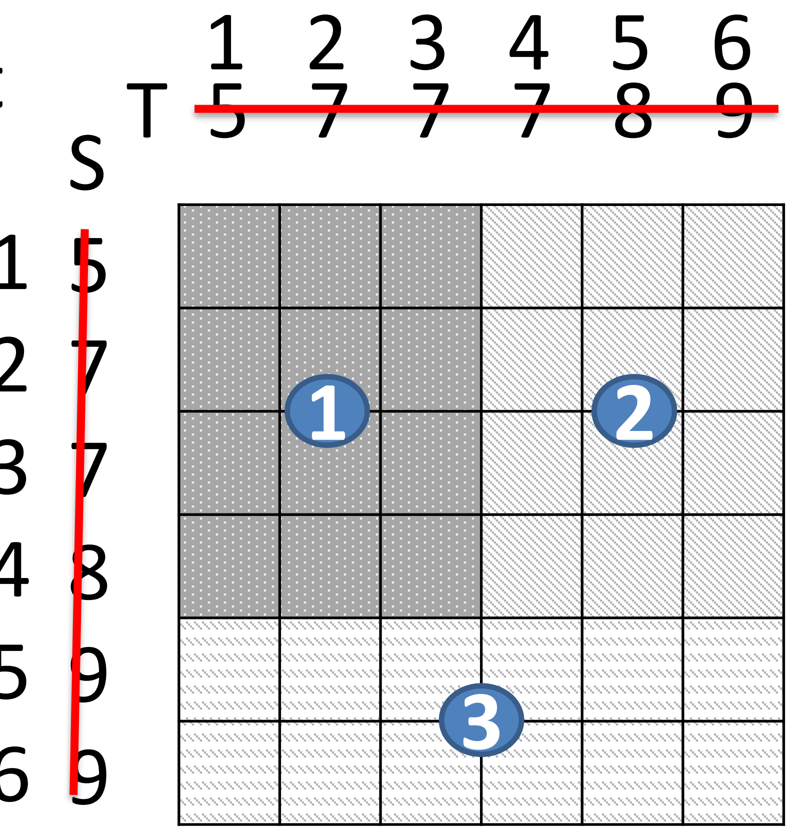
1-Bucket-Theta algorithm

Observations

- Correctness/completeness of results unaffected if a row/column contains <>1 tuples
- Order is not important

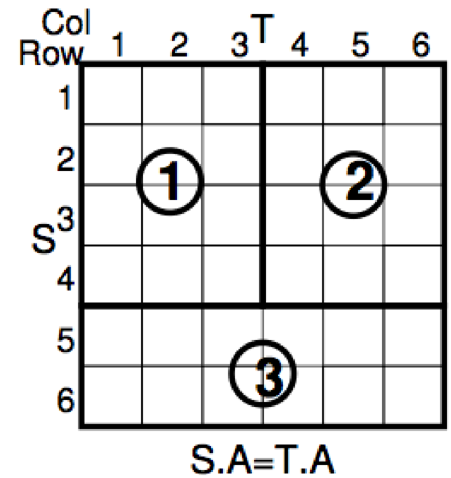
Trick

- Randomly map each tuple to one row/column
- No preprocessing



1-Bucket-Theta: Map

- Input:
  - tuple $x \in T \cup S$
  - matrix-to-reducer mapping lookup table



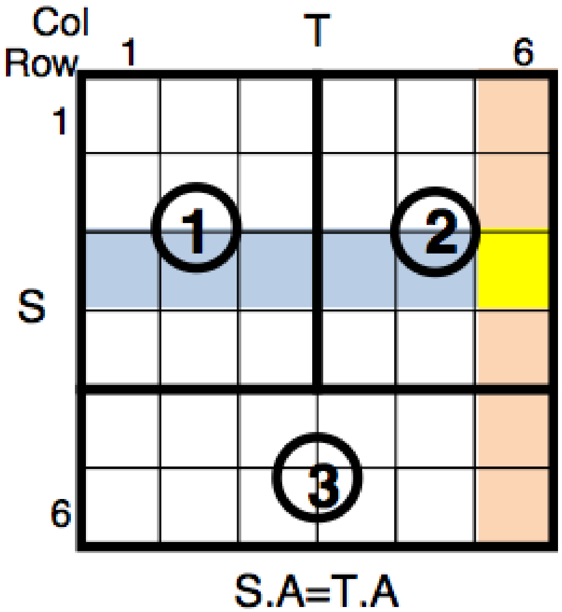

```
if x ∈ S then
    matrixRow = random(1,|S|)
    for all regionID in lookup.getRegions(matrixRow) do
        Output (regionID, (x, "S") ) /* key: regionID */
else
    matrixCol = random(1,|T|)
    for all regionID in lookup.getRegions(matrixCol) do
        Output (regionID, (x, "T") )
```

1-Bucket-Theta: Reduce

- Input:
  - (ID, [(x_1, origin$_1$), … , (x_k, origin$_k$)])

Stuples = $\emptyset$; Ttuples = $\emptyset$
for all (x_i, origin$_i$) in input list **do**
    if origin$_i$ = “S” **then**
        Stuples = Stuples $\cup \{x_i\}$
    else
        Ttuples = Ttuples $\cup \{x_i\}$
joinResult = MyFavoriteJoinAlg(Stuples, Ttuples)
Output(joinResult)

1-Bucket-Theta Example



Map:

Input tuple	Random row/col	Output
S1.A=5	3	(1,S1),(2,S1)
S2.A=7	5	(3,S2)
S3.A=7	1	(1,S3),(2,S3)
S4.A=8	5	(3,S4)
S5.A=9	1	(1,S5),(2,S5)
S6.A=9	2	(1,S6),(2,S6)
T1.A=5	6	(2,T1),(3,T1)
T2.A=7	2	(1,T2),(3,T2)
T3.A=7	2	(1,T3),(3,T3)
T4.A=7	3	(1,T4),(3,T4)
T5.A=8	6	(2,T5),(3,T5)
T6.A=9	4	(2,T6),(3,T6)

Reduce:

Reducer X: key 1
Input: S1, S3, S5 ,S6 T2, T3, T4
Output: (S3,T2),(S3,T3),(S3,T4)

Reducer Y: key 2
Input: S1, S3, S5, S6 T1, T5, T6
Output: (S1,T1),(S5,T6),(S6,T6)

Reducer Z: key 3
Input: S2, S4 T1, T2, T3, T4, T5, T6
Output: (S2,T2),(S2,T3), (S2,T4),(S4,T5)

Why Randomization?

- Avoids pre-processing step to assign row/column IDs to records
- Effectively removes output skew
- Input sizes very close to target
  - Chernoff bound: due to large number of records per reducer, probability of receiving 10% or more over target is virtually zero

Remaining Challenges

- What is the best way to cover all true-valued cells?
  - Assume r reducers
  - Partition the space to r squares, each of size $\sqrt{\frac{|S|\ |T|}{r}}$
  - Constant factor from optimal partitioning
- How do we know which cells are not empty?
  - Maintain histogram statistics
  - Do not send S tuples when T histogram is empty (and vice versa)

Summary

- Spark richer model than MapReduce
  - Addresses some of the major problems of MapReduce
- Adding more machines is not always a solution
  - Skew can drastically reduce degree of parallelism
  - Need to have suitable algorithms

Reading material

- M. Zaharia, et.al.: "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", NSDI 2012.
- Armbrust et al.: Spark SQL: Relational Data Processing in Spark. SIGMOD Conference 2015: 1383-1394
- Shivnath Babu, Herodotos Herodotou: Massively Parallel Databases and MapReduce Systems. Foundations and Trends in Databases 5(1): 1-104 (2013). **Section 5**. Available online at http://www.nowpublishers.com/article/Details/DBS-036
- Alper Okcan et.al.: Processing theta-joins using MapReduce. SIGMOD Conference 2011. http://www.ccs.neu.edu/home/mirek/papers/2011-SIGMOD-ParallelJoins.pdf

Reading material

Technical readings

- Spark DATABRICKS tutorial: https://www.youtube.com/watch?v=VWeWViFCzzg
- Spark references:
  - Learning Spark: Lightning-Fast Big Data Analysis. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia.
  - Official documentation at http://spark.apache.org, http://spark.apache.org/docs/latest/quick-start.html, http://spark.apache.org/docs/latest/programming-guide.html