Structured data management on top of massively parallel platforms

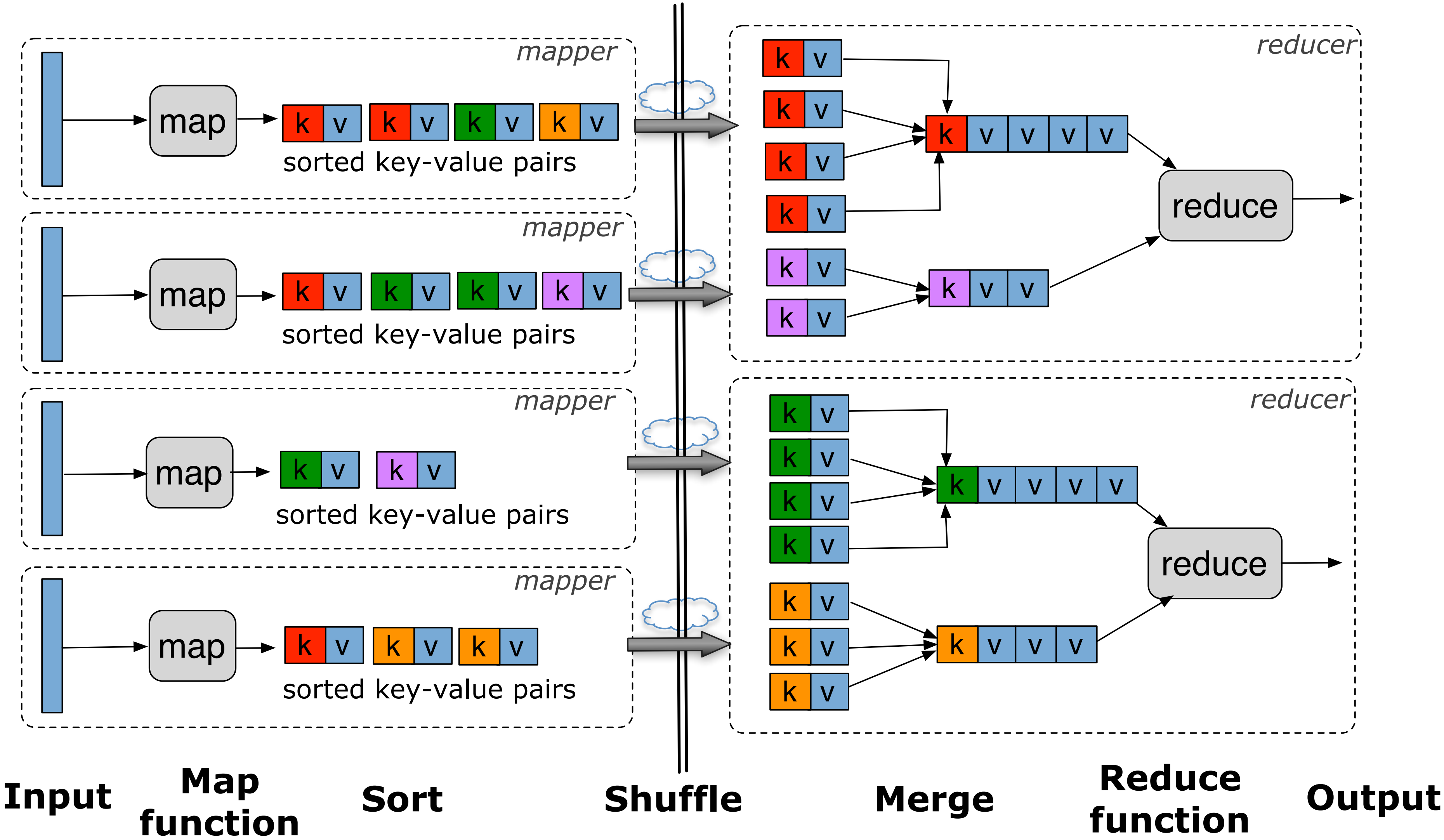
Ioana Manolescu
INRIA Saclay & Ecole Polytechnique
ioana.manolescu@inria.fr
http://pages.saclay.inria.fr/ioana.manolescu/

M2 Data and Knowledge
Université de Paris Saclay

Outline

- MapReduce and other massively parallel platforms are becoming the norm for large-scale computing
- How to build Big Data management architectures based on such architectures ?
- We will see:
  - Improving data access performance
  - Implementing algebraic operations on MapReduce
  - A few visible Big Data platforms implemented on top of MapReduce clusters
  - Query optimization revisited for MapReduce (also multi-query optimization)
  - Some open problems in this area

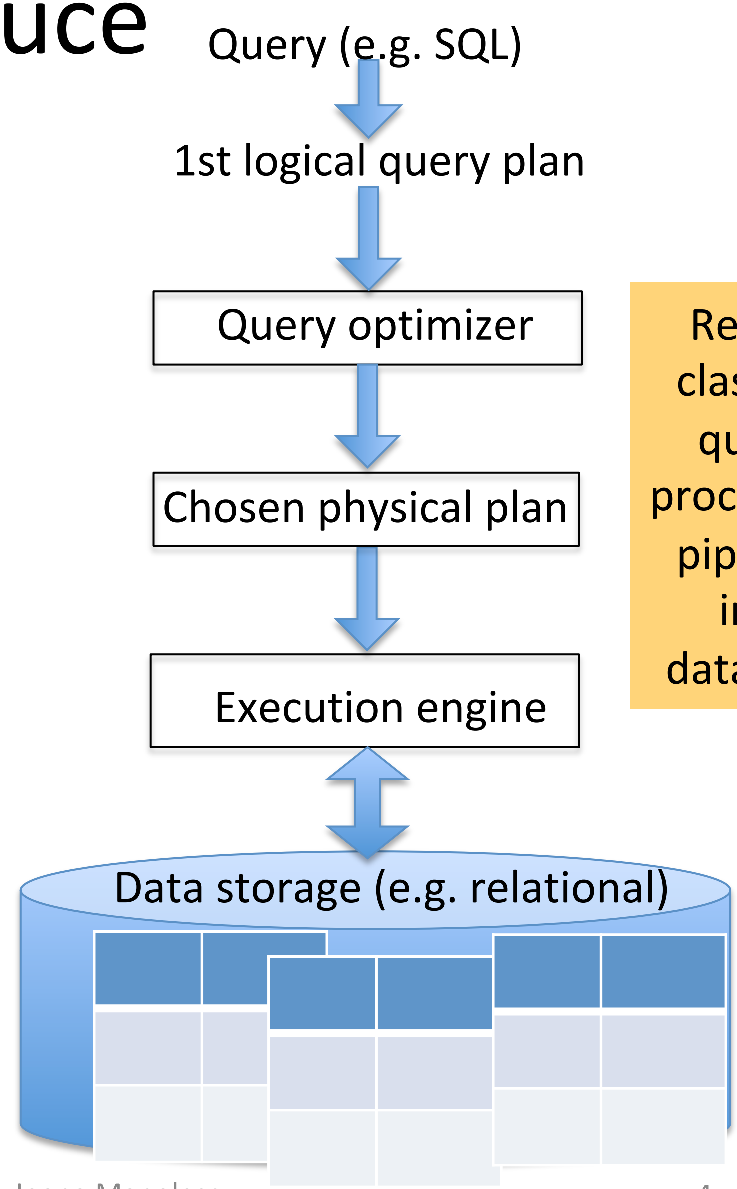
Recall: Map/Reduce outline



Data management based on MapReduce

How can a DBMS architecture be established on top of a distributed computing platform?

- **Store (distribute) the data** in a distributed file system
  - How to split it?
  - How to store it?
- **Process queries** in a parallel fashion based on MapReduce
  - How to evaluate operators?
  - How to optimize queries

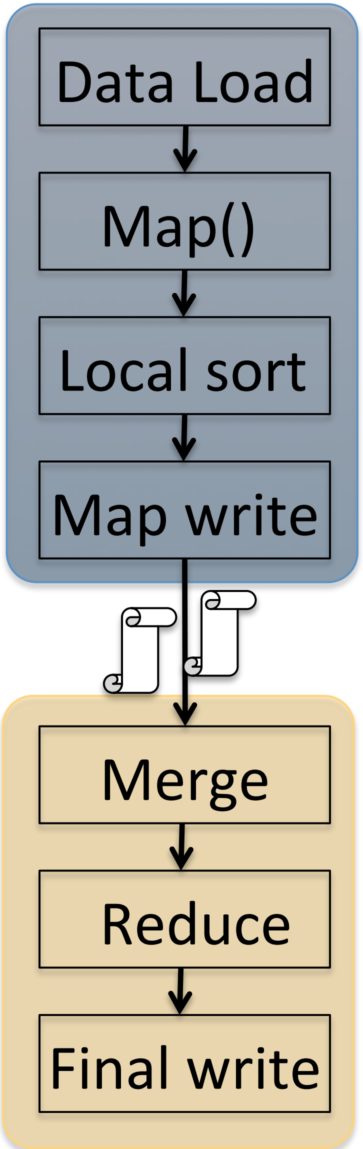


Recall: classical query processing pipeline in a database

IMPROVING DATA ACCESS PERFORMANCE IN A DISTRIBUTED FILE SYSTEM

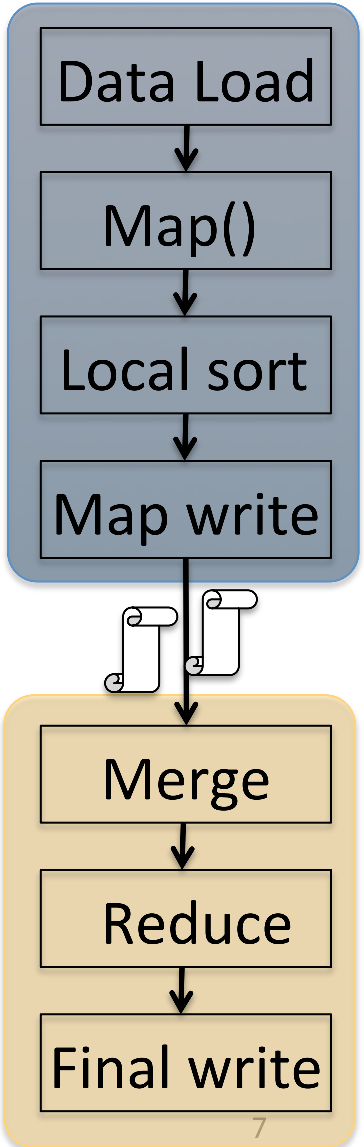
Data access in Hadoop

- Basic model: *read all the data*
  - If the tasks are selective, we don't really need to!
- Database indexes? But:
  - Map/Reduce works on top of a file system (e.g. Hadoop file system, HDFS)
  - Data is stored only once
  - Hard to foresee all future processing
    - "Exploratory nature" of Hadoop



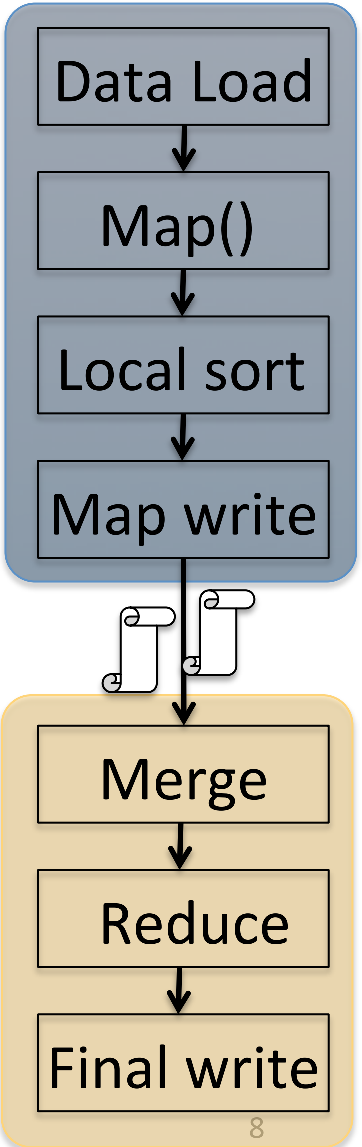
Accelerating data access in Hadoop

- Idea 1: Hadop++ [JQD2011]
  - Add header information to each data split, summarizing split attribute values
  - Modify the RecordReader of HDFS, used by the Map().
    Make it prune irrelevant splits

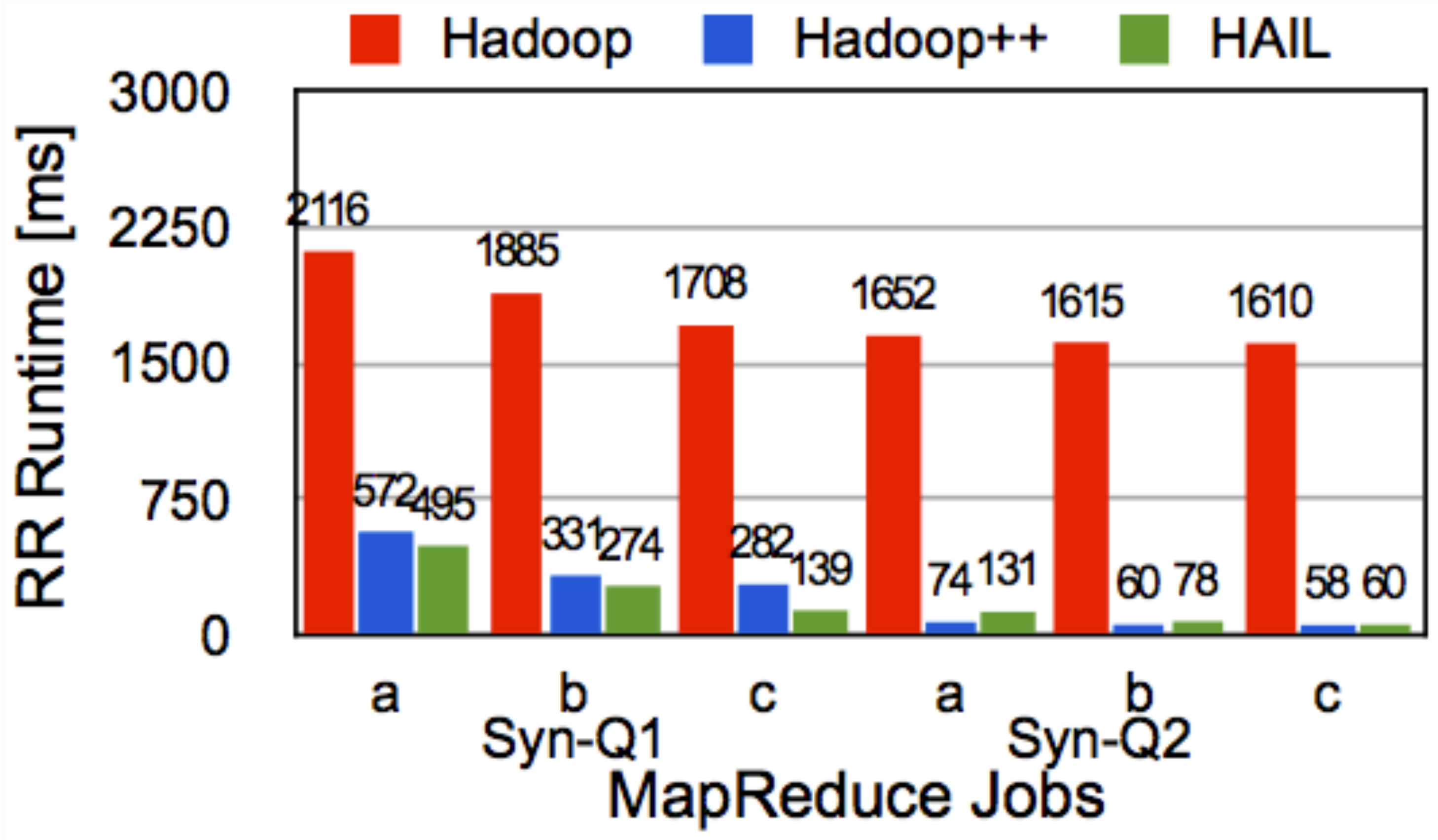


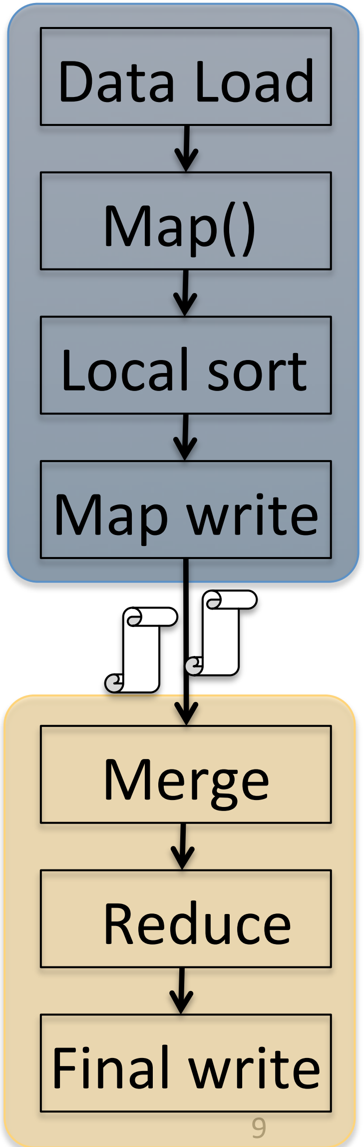
Accelerating data access in Hadoop

- Idea 2: HAIL [DQRSJS12]
  - Each storage node builds an in-memory, clustered index of the data in its split
  - There are three copies of each split for reliability → Build three different indexes!
  - Customize RecordReader



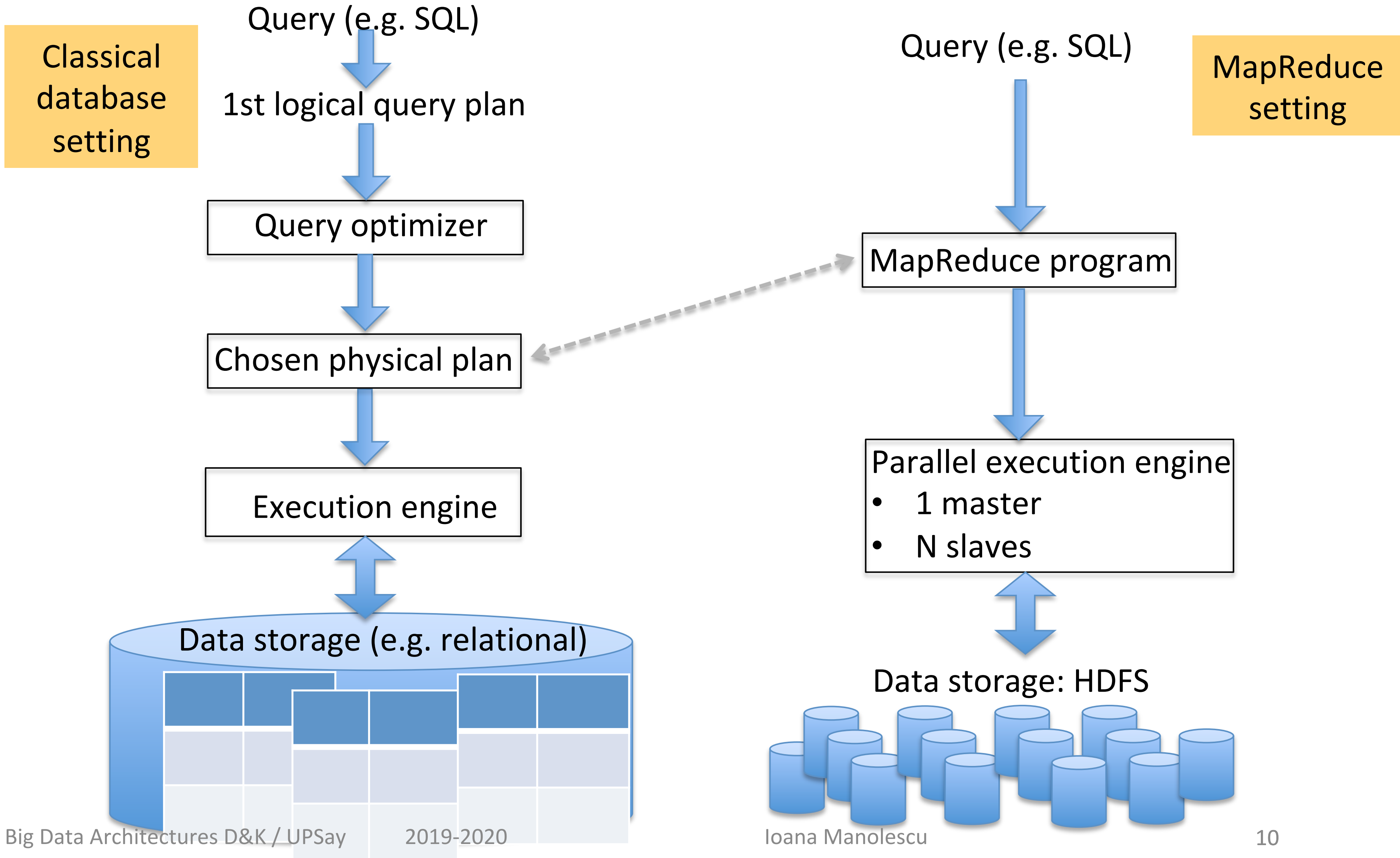
Hadoop, Hadoop++ and HAIL





Data management based on MapReduce

First idea: translate each query into a program



PROCESSING STRUCTURED QUERIES THROUGH MAPREDUCE

First idea: write a MapReduce program for every query

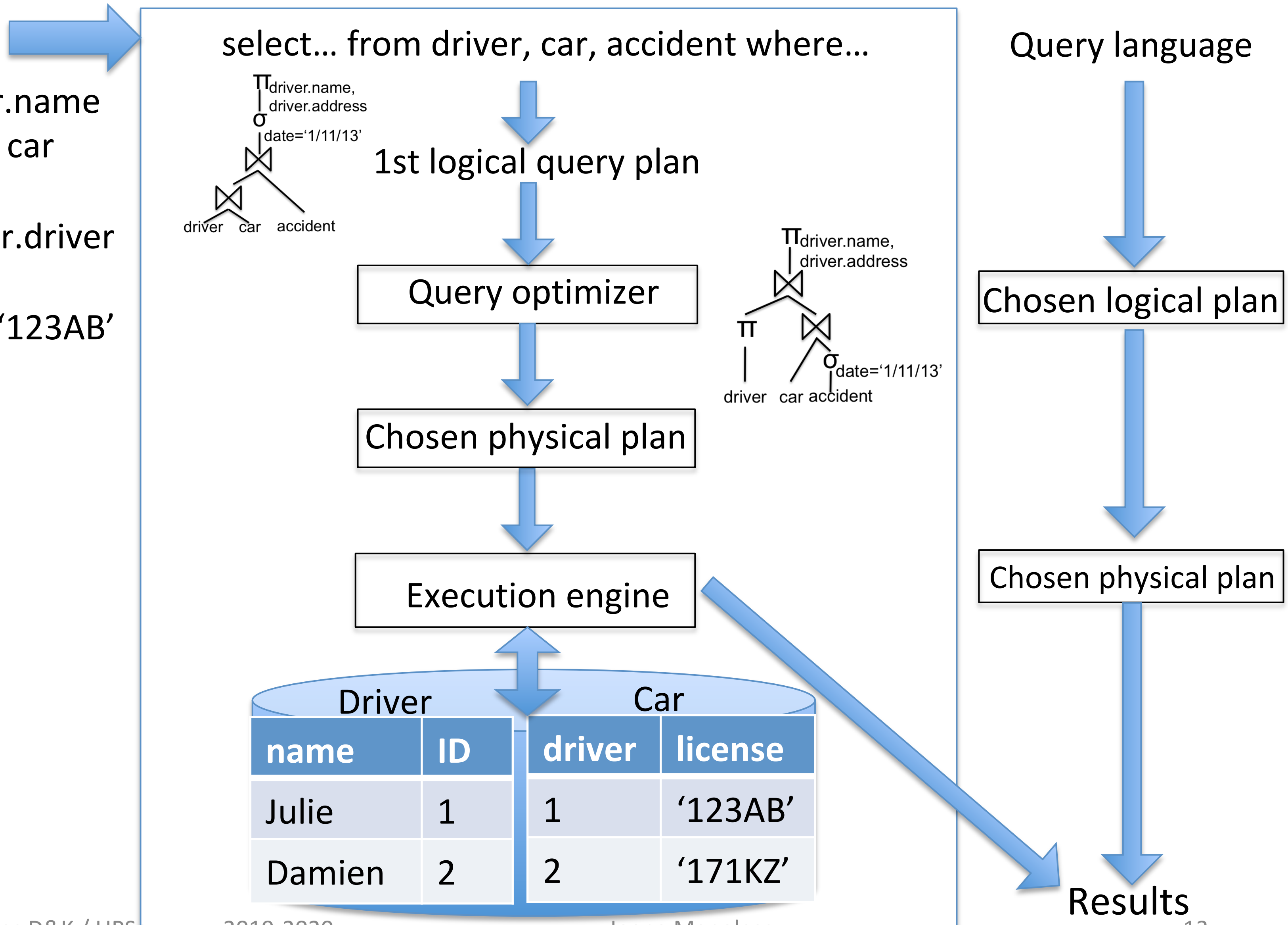
What are the MapReduce

- SELECT MONTH(c.start_date), COUNT(*)
  FROM customer c
  GROUP BY MONTH(c.start_date)
- SELECT c.name, o.total
  FROM customer c, order o
  WHERE c.id=o.cid
- SELECT c.name, SUM(o.total)
  FROM customer c, order o
  WHERE c.id=o.cid
  GROUP BY c.name

Recall: query processing stages in a DBMS

SQL

select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license='123AB'



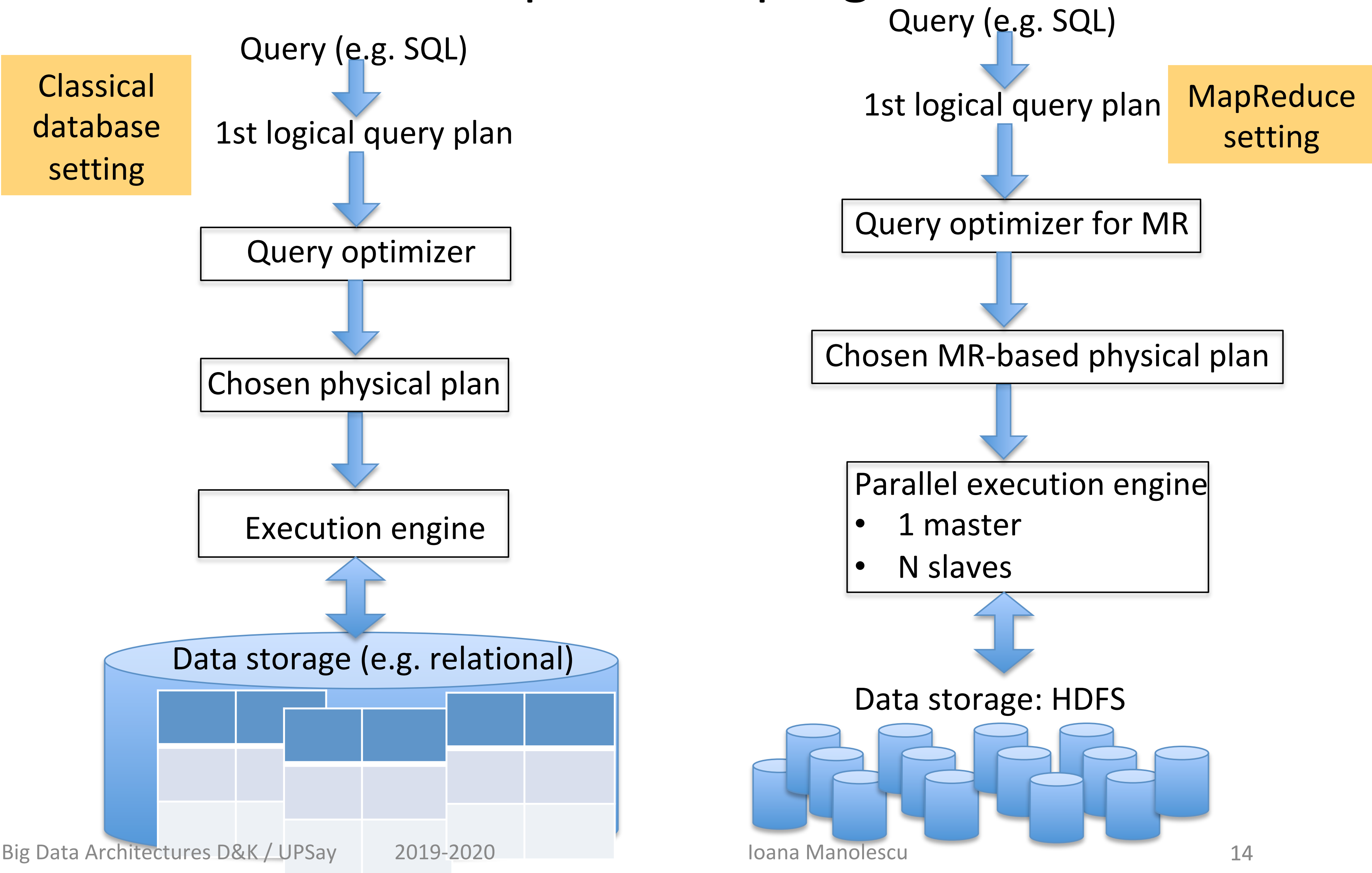
Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	'123AB'
2	'171KZ'

Second idea: translate every physical operator into a MapReduce program



Implementing physical operators on MapReduce

- **To avoid writing code for each query!**

- If each operator is a (small) MapReduce program, we can evaluate queries by composing such small programs
- The optimizer can then chose the best MR physical operators and their orders (just like in the traditional setting)
- Translate:
  - Unary operators (σ and π)
  - Binary operators (mostly: $\bowtie$ on equality, i.e. equijoin)
  - N-ary operators (complex join expressions)

Implementing unary operators on MapReduce

- Selection (σ_{pred} (R)):
  - Split the R input tuples over all the nodes
  - **Map**:
    foreach t which satisfies pred in the input partition
    - Output (hn(t.toString()), t); // hn fonction de hash
  - **Reduce**:
    - Concatenate all the inputs

What values should hn take?

Implementing unary operators on MapReduce

- Projection (π_{cols}(R)):
  - Split R tuples across all nodes
  - **Map**:
    foreach t
      output (hn(t), π_{cols}(t))
  - **Reduce**:
    - Concatenate all the inputs

- Better idea?

Recall: physical operators for binary joins (classical DBMS scenario)

Example: equi-join (R.a=S.b)

Nested loops join:

O(|R|x|S|)

```
foreach t1 in R{
  foreach t2 in S {
    if t1.a = t2.b then output (t1 || t2)
  }
}
```

Merge join: // requires sorted inputs

O(|R|+|S|)

```
repeat{
 while (!aligned) { advance R or S };
 while (aligned) { copy R into topR, S into topS };
 output topR x topS;
} until (endOf(R) or endOf(S));
```

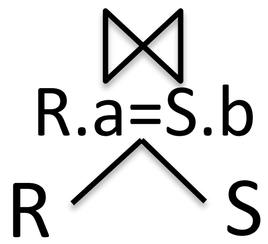
Hash join: // builds a hash table in memory

```
While (!endOf(R)) { t ← R.next; put(hash(t.a), t); }
While (!endOf(S)) { t ← S.next;
                    matchingR = get(hash(S.b));
                    output(matchingR x t);
                  }
```

O(|R|+|S|)

Also:
Block nested loops join
Index nested loops join
Hybrid hash join
Hash groups / teams
…

Implementing equi-joins on MapReduce (1)



Repartition join [Blanas 2010] (~symetric hash)

Mapper:

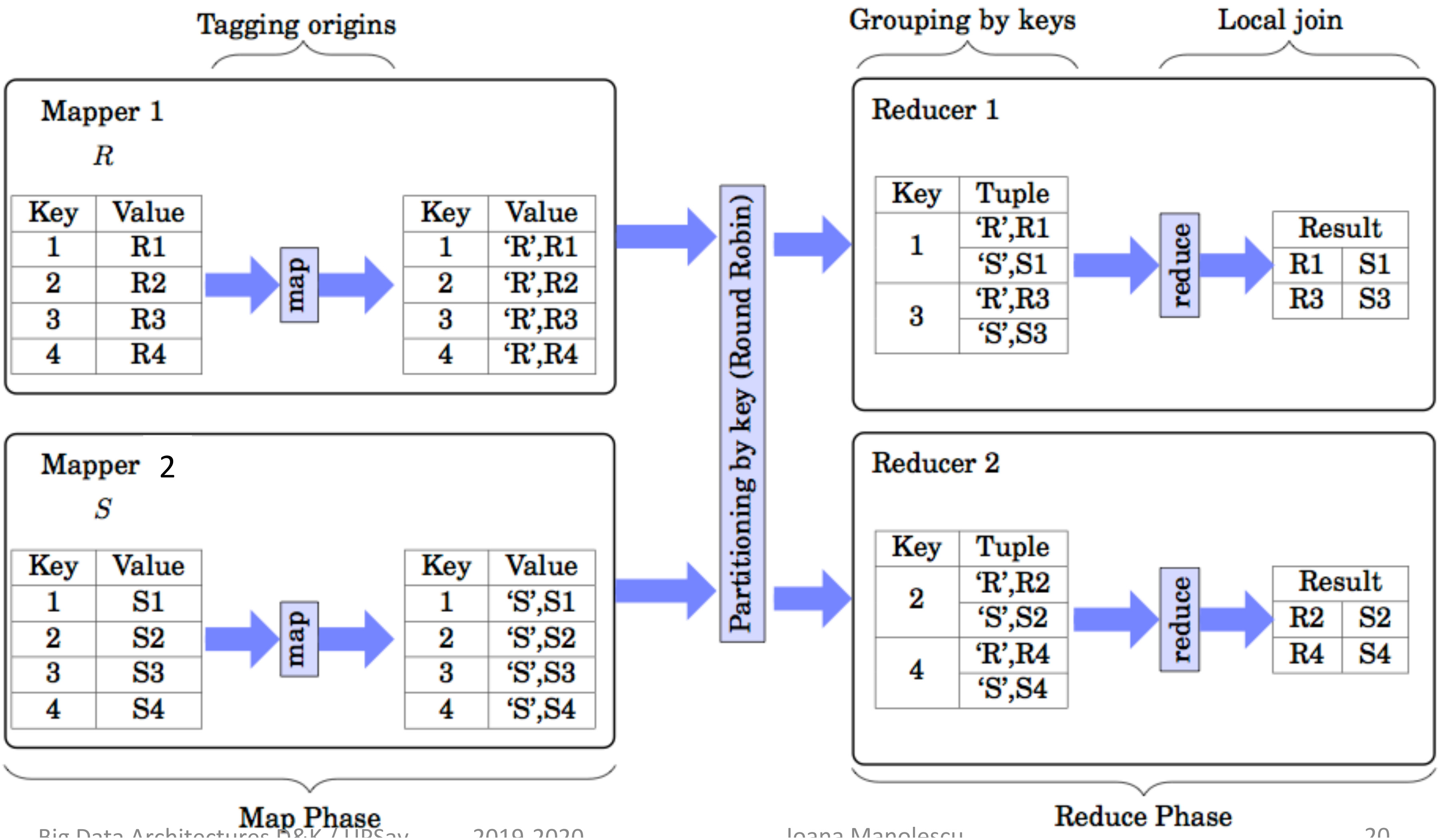
- Output (t.a, («R», t)) for each t in R
- Output (t.b, («S», t)) for each t in S

Reducer:

- Foreach input key k
  - Res_k = set of all R tuples on k × set of all S tuples on k
- Output Res_k

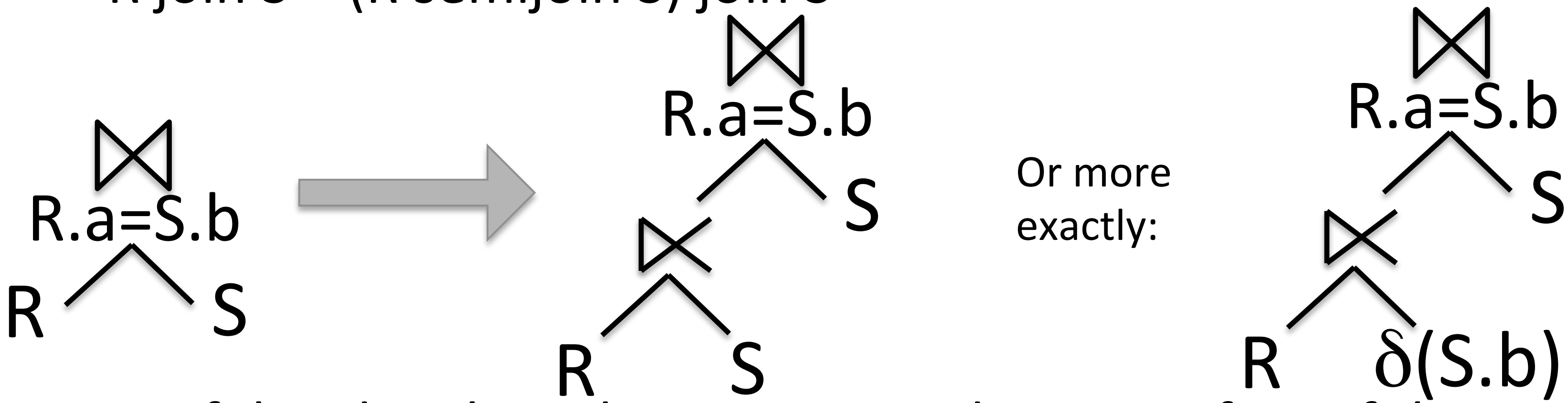
Implementing equi-joins on MapReduce (1) Repartition join

- R(rID, rVal) join(rID = SID) S(sID, sVal)



Implementing equi-joins on MapReduce (2)

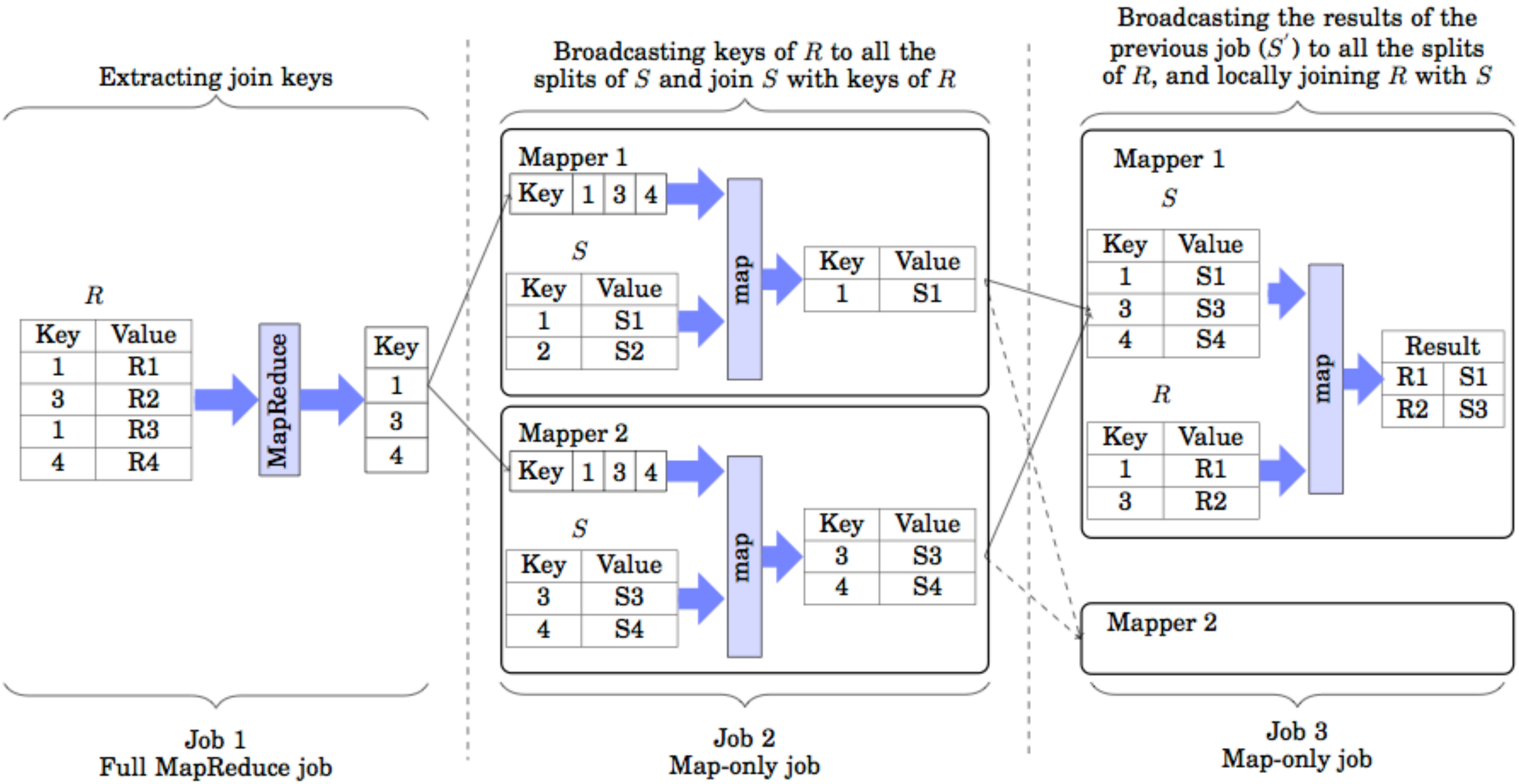
- **Semijoin-based MapReduce join**
- Recall: semijoin optimization technique:
  - R join S = (R semijoin S) join S



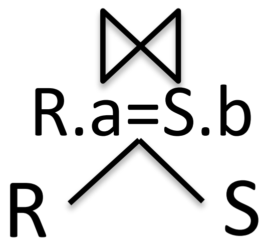
  - Useful in distributed settings to reduce transfers: *if the distinct S.b values* are smaller than *the non-matching R tuples*
  - Symetrical alternative: R join S = R join (S semijoin R)

Implementing equi-joins on MapReduce (2)

- Semijoin-based MapReduce join



Implementing equi-joins on MapReduce (3)



Broadcast (map-only) MapReduce join [Blanas2010]

If |R| << |S|, broadcast R to all nodes!

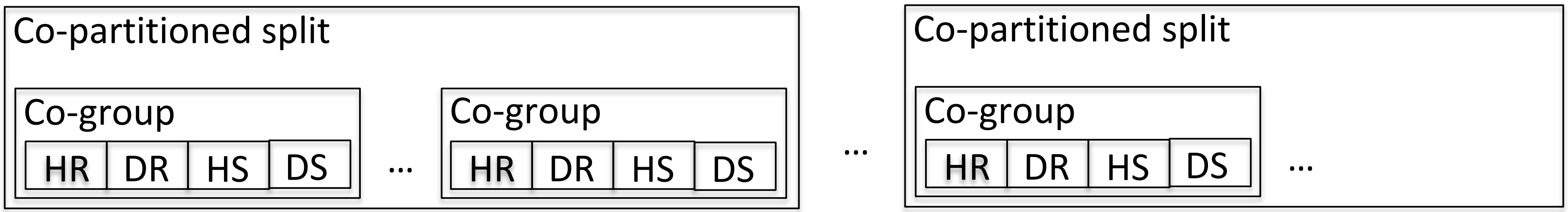
- Example: S is a *log* data collection (e.g. log table)
- R is a *reference* table e.g. with user names, countries, age, …
- Facebook: 6 TB of new log data/day

Map: Join a partition of S with R.

Reduce: nothing (« map-only join »)

Implementing equi-joins on MapReduce (4)

- Trojan Join [Dittrich 2010]
- A Map task is sufficient for the join if relations are already co-partitioned by the join key
  - The slice of R with a given join key is already next to the slice of S with the same join key
  - This can be achieved by a MapReduce job similar to repartition join but which builds co-partitions at the end



  - Useful when the joins can be known in advance (e.g. keys – foreign keys)

Implementing binary equi-joins in MapReduce

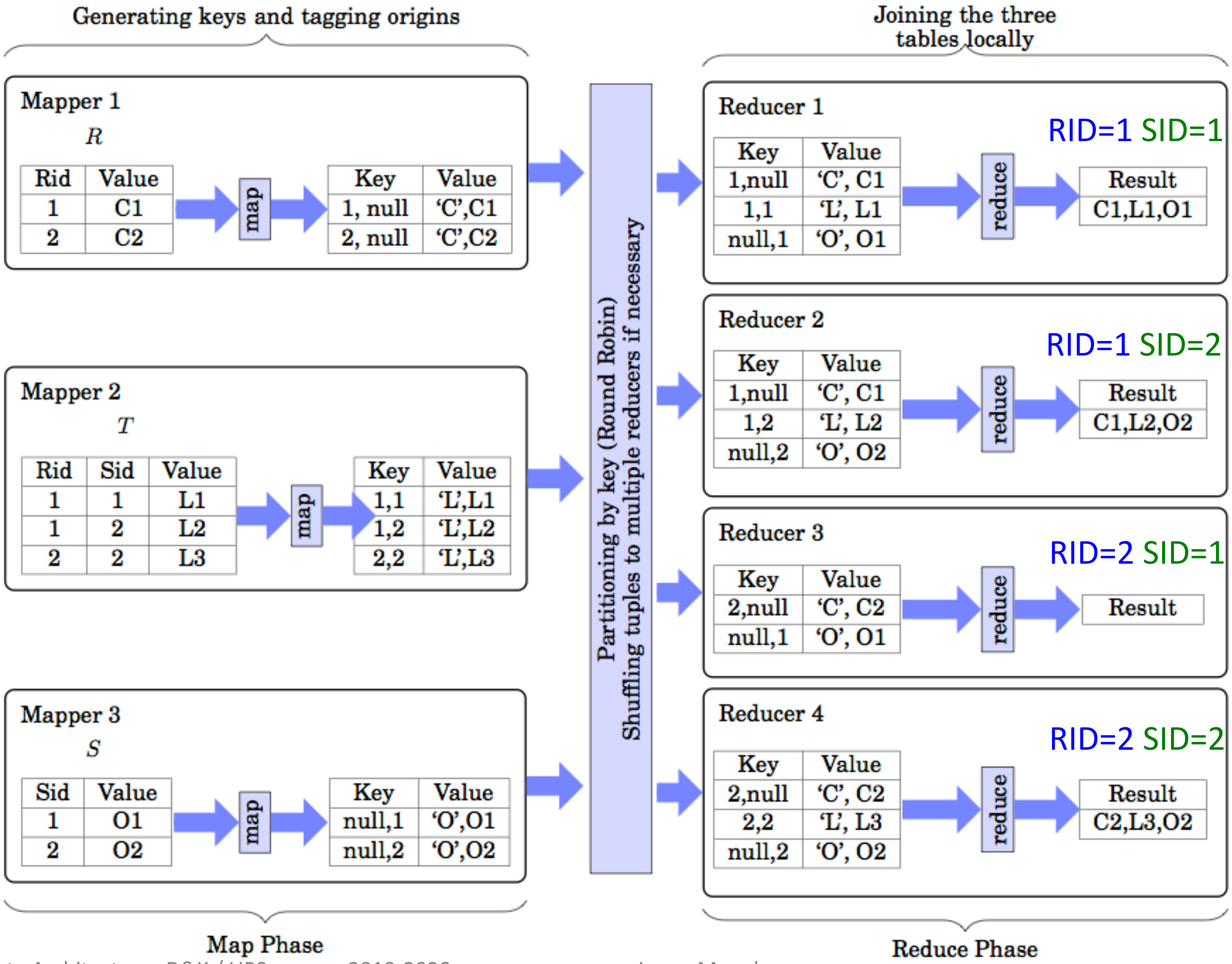
Algorithm	+	-
Repartition Join	Most general	Not always the most efficient
Semijoin-based Join	Efficient when semijoin is selective (has small results)	Requires several jobs, one must first do the semi-join
Broadcast Join	Map-only	One table must be very small
Trojan Join	Map-only	The relations should be co-partitioned

Implementing n-ary (« multiway ») join expressions in MapReduce

- R(RID, C) join T(RID, SID, O) join S(SID, L)
- « Mega » operator for the whole join expression?...

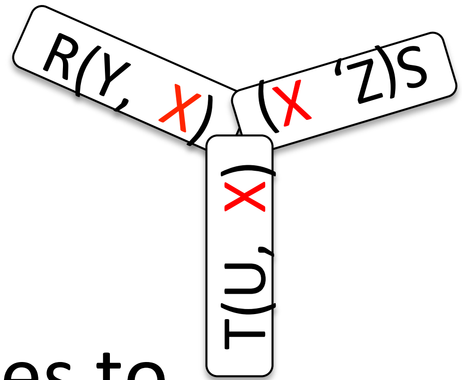
- Three relations, two join attributes (RID and SID)
- Split the SIDs into Ns groups and the RIDs in Nr groups. Assume Nr x Ns reducers available.
- Hash **T** tuples according to a composite key made of the two attributes. Each **T** tuple goes to one reducer.
- Hash **R** and **S** tuples on *partial keys* (RID, null) and (null, SID)
- Distribute **R** and **S** tuples to each reducer where the non-null component matches (potentially multiple times!)

Implementing multi-way joins in MR: replicated joins



Particular case of multi-way joins: star joins on MapReduce

- Same join attribute in all relations:
  R(x, y) join S(x, z) join T(x, u)



- If N reducers are available, it suffices to partition the space of x values in N
- Then co-partition R, S, T → map-only join

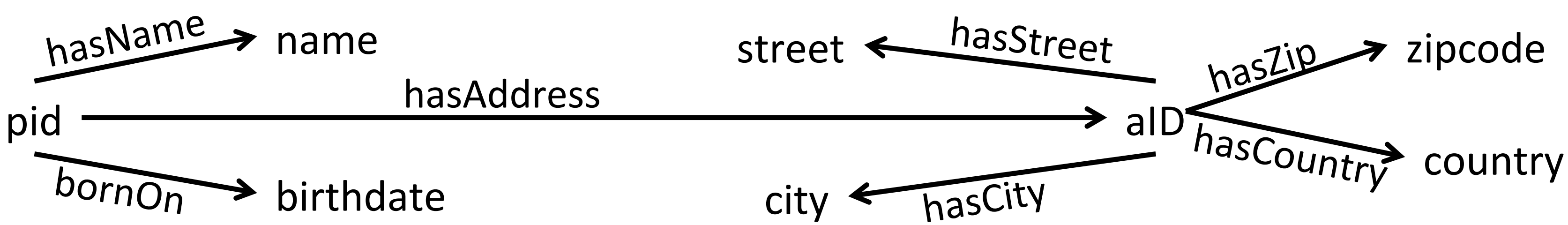
QUERY OPTIMIZATION FOR MAPREDUCE

Query optimization for MapReduce

- Given a query over relations R1, R2, …, Rn, how to translate it into a MapReduce program?
  – Use **one replicated join**. Pbm: the space of composite join keys (Att1|Att2|…|Attk) is limited by the number of reducers →
    may shuffle some tuples to many reducers.
  – Use **n-1 binary joins**
  – Use **n-ary (multiway) joins only**

A yardistick for MapReduce query optimization: SPARQL

- The standard language for RDF
- Conjunctive query = join of triples
- Relational vs. RDF data modeling:
  - **Relational**: **2** atoms
    Person(id, name, birthdate), **Address**(pID, street, city, zipcode, country)
  - **RDF**: **7** atoms
    triple(pID, hasName, name), **triple**(pID, bornOn, birthDate), **triple**(pID, hasAddress, aID), **triple**(aID, hasStreet, street), **triple**(aID, hasCity, city), **triple**(aID, hasZip, zipCode), **triple**(aID, hasCountry, country)



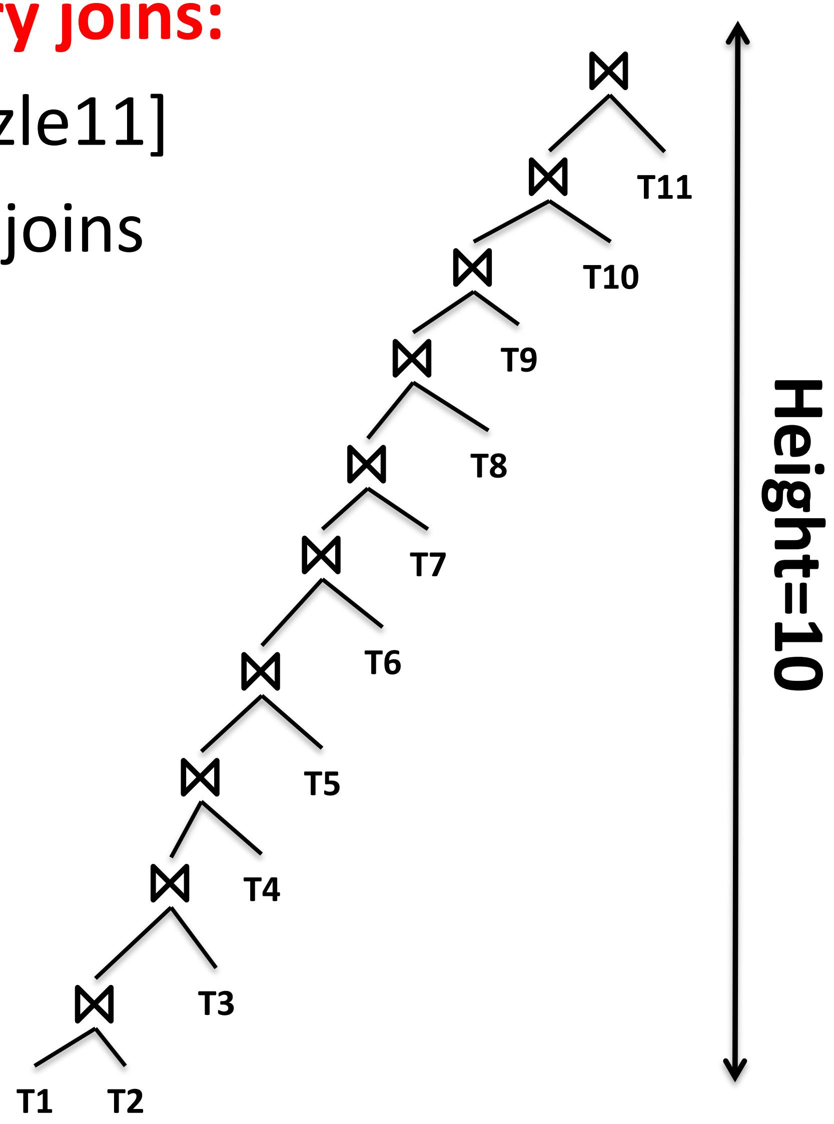
  - SPARQL query optimization is a stress test for MapReduce platforms

Query plans on MapReduce

- **Left deep plans with binary joins:**

[Olston08][Rohloff10][Schatzle11]

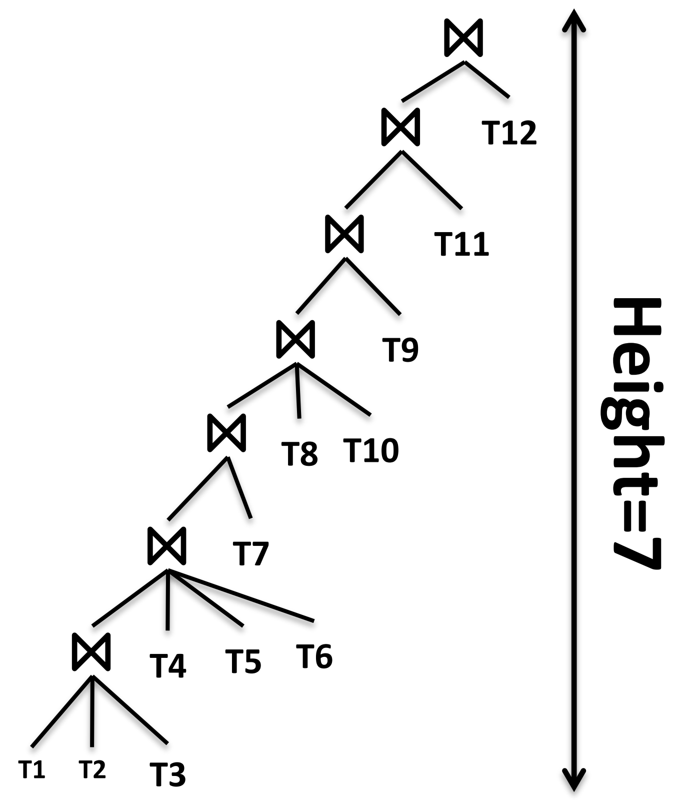
- Left deep plans with n-ary joins



Query optimization overview

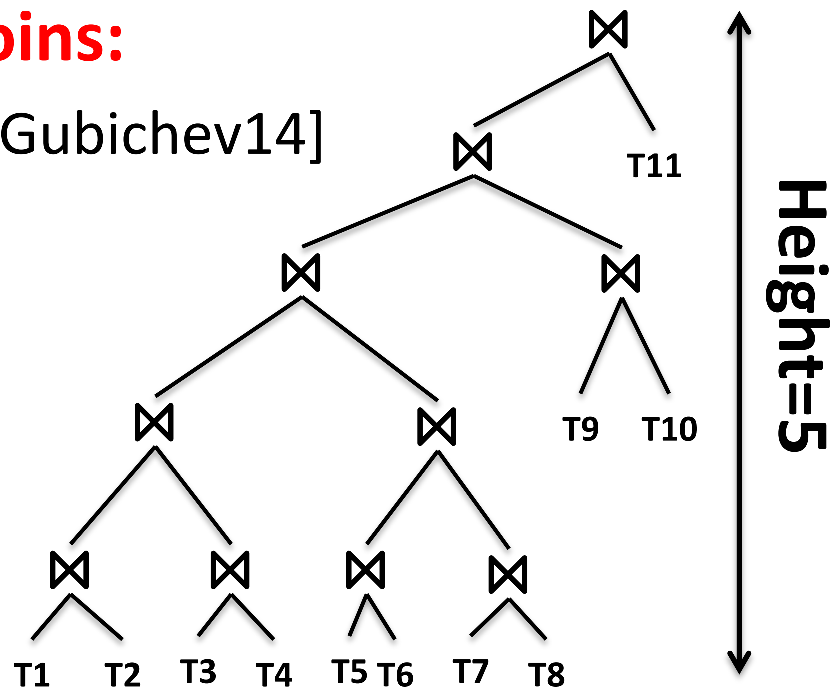
– Left deep plans with binary joins [Olston08][Rohloff10][Schatzle11]

– **Left deep plans with n-ary joins: [Papailiou13]**



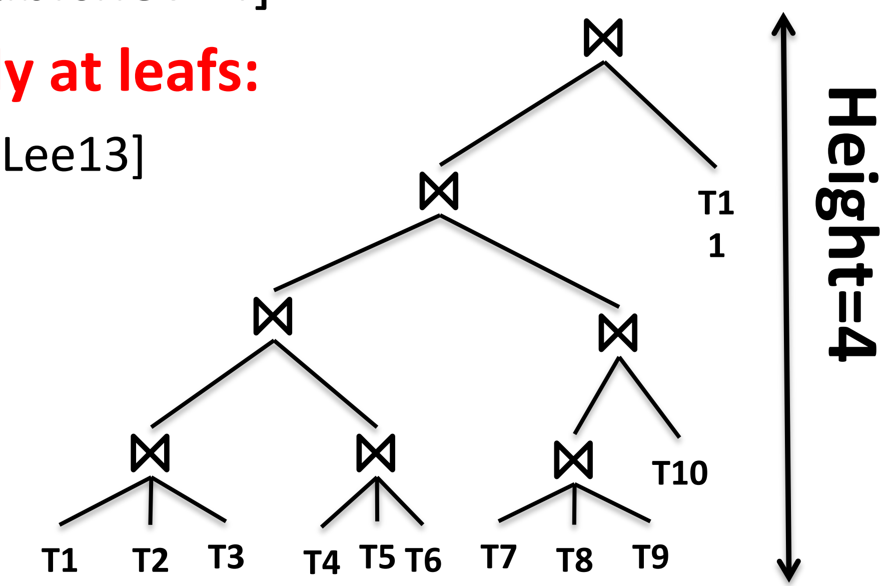
Query optimization overview

- Left deep plans with binary joins
  [Olston08][Rohloff10][Schatzle11]
- Left deep plans with n-ary joins
  [Papailiou13]
- **Bushy plans with binary joins:**
  [Neumann10][Tsialiamanis12][Gubichev14]



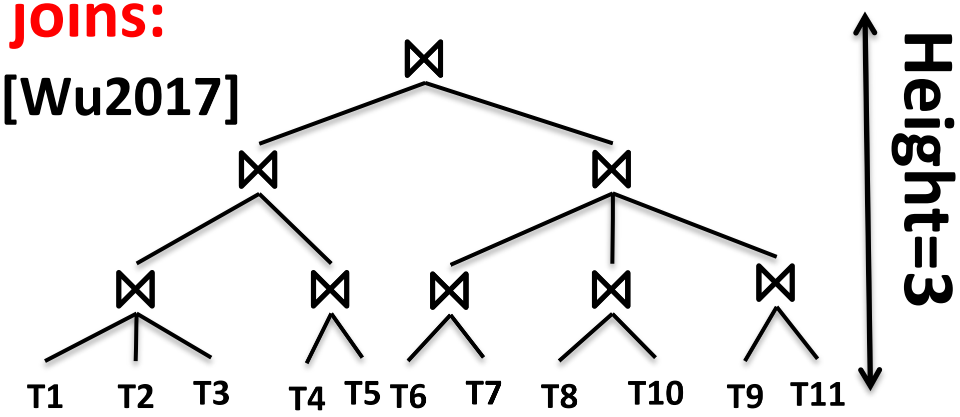
Query optimization overview

- Left deep plans with binary joins [Olston08][Rohloff10][Schatzle11]
- Left deep plans with n-ary joins [Papailiou13]
- Bushy plans with binary joins [Neumann10][Tsialiamanis12][Gubichev14]
- **Bushy plans with n-ary joins only at leafs:** [Wu11][Kim11][Huang11][Ravindra11][Lee13]

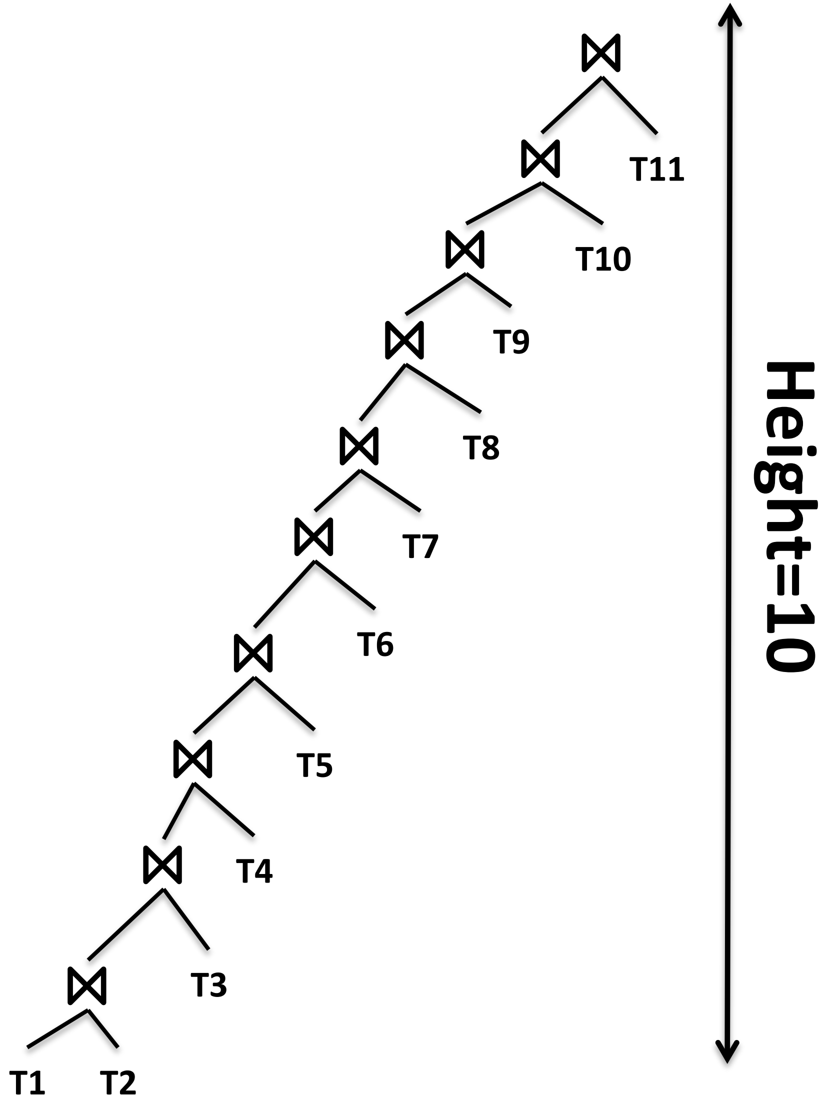


Query optimization overview

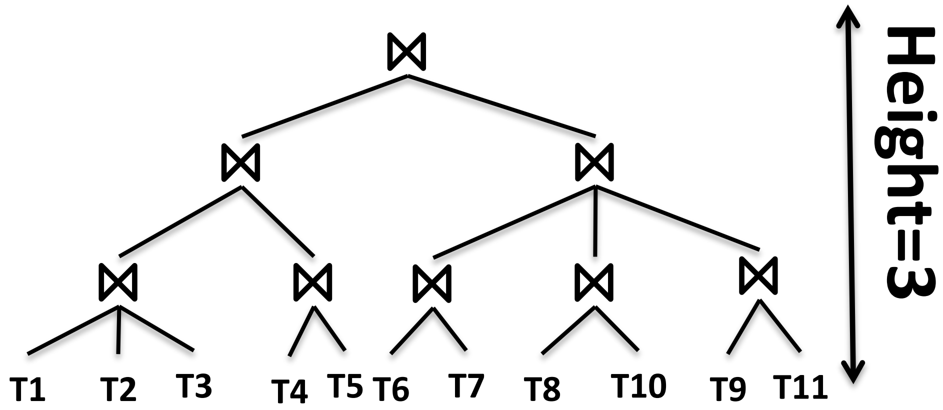
- Left deep plans with binary joins [Olston08][Rohloff10][Schatzle11]
- Left deep plans with n-ary joins [Papailiou13]
- Bushy plans with binary joins [Neumann10][Tsialiamanis12][Gubichev14]
- Bushy plans with n-ary joins only at leafs [Wu11][Kim11][Huang11][Ravindra11][Lee13]
- **Bushy plans with n-ary joins:** **[Husain11][Goasdoué2015][Wu2017]**



Query optimization in MapReduce



- Usually, each join layer is translated into a set of parallel MR jobs
- The plan height = the number of successive jobs
- Impacts execution time!



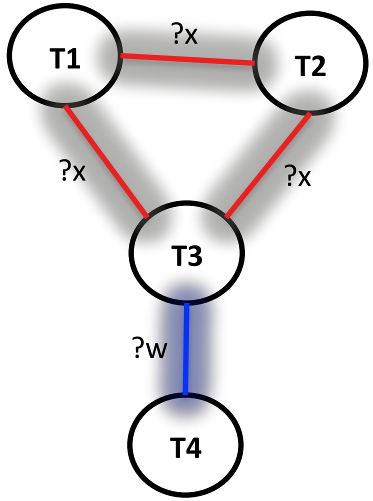
CliqueSquare: flat plans for massively parallel RDF queries

- **Focus:** Build massively parallel **flat** plans for RDF queries by exploiting **n-ary (star)** equality joins.
- Publication, code at: https://team.inria.fr/oak/projects/cliquesquare/
- Main idea:
  - identify subsets of **>=2 triples** that can be joined through an **n-ary join** on a common variable at a given moment
  - reiterate on the intermediary results thus obtained until there is only one set of tuples left (nothing to join: the query result has been obtained)

CliqueSquare algorithm: Variable Graphs

- Represent incoming queries and intermediary relations

SELECT ?x ?y
WHERE {
T1: **?x** takesCourse ?y .
T2: **?x** member ?z .
T3: **?w** advisor **?x** .
T4: **?w** name ?u .}

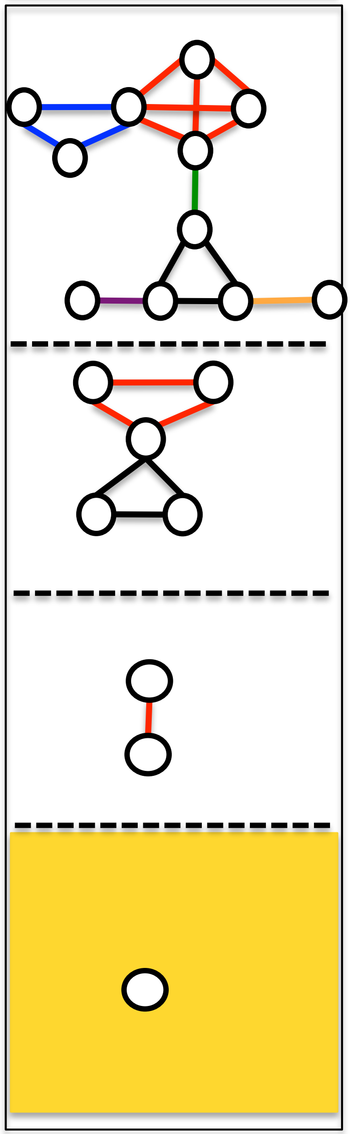


Query

Variable graph

Nodes are connected with an **edge** if they share a **variable**

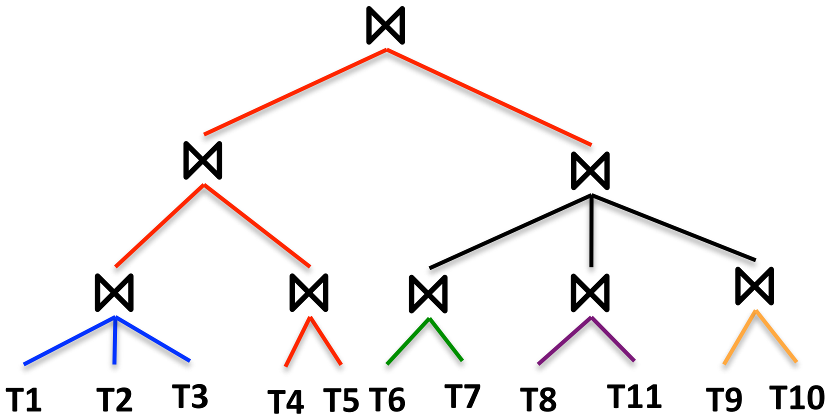
States



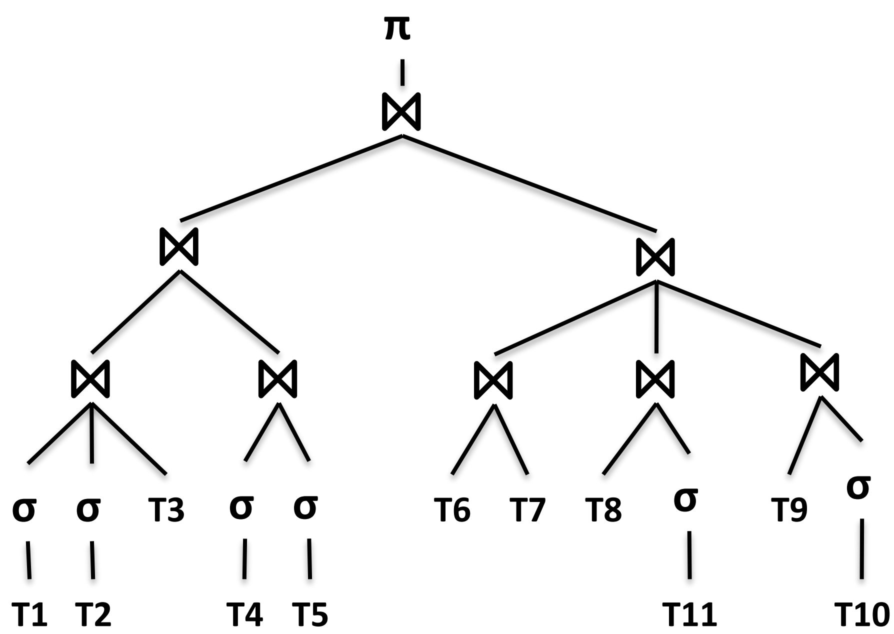
CliqueSquare: optimization with n-ary joins

Each **node** of a graph corresponds to a **clique** of nodes of the previous graph.

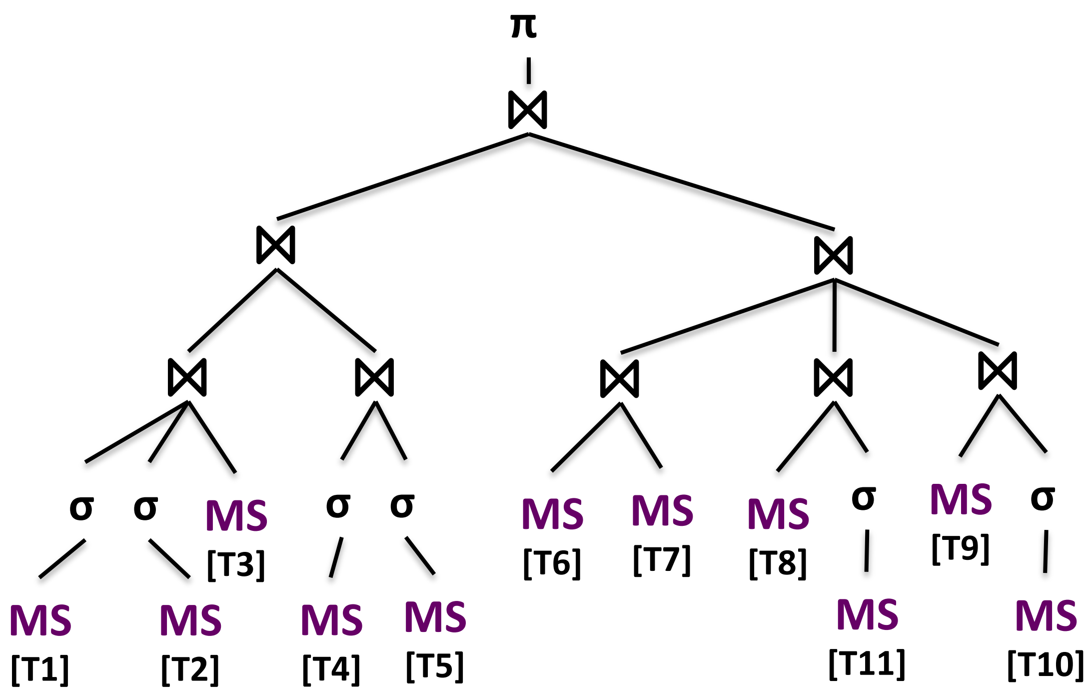
A join operator corresponds to the "collapsing" of one clique (triples that all join on the same variables) into a single node



From logical plan to physical plans

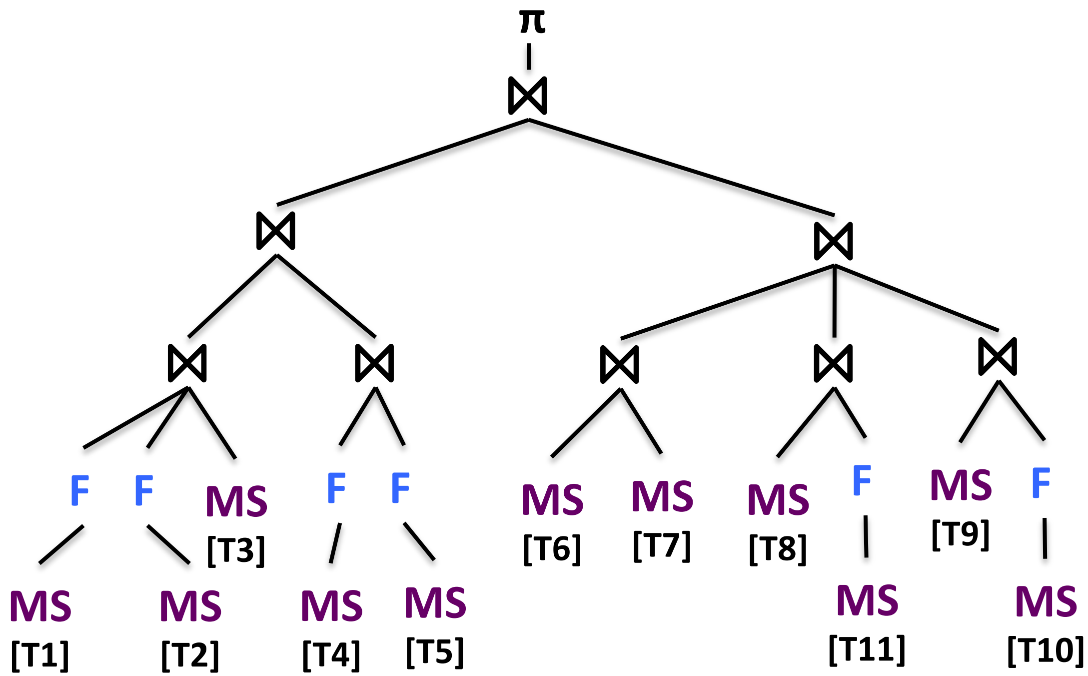


Logical plan → physical plan



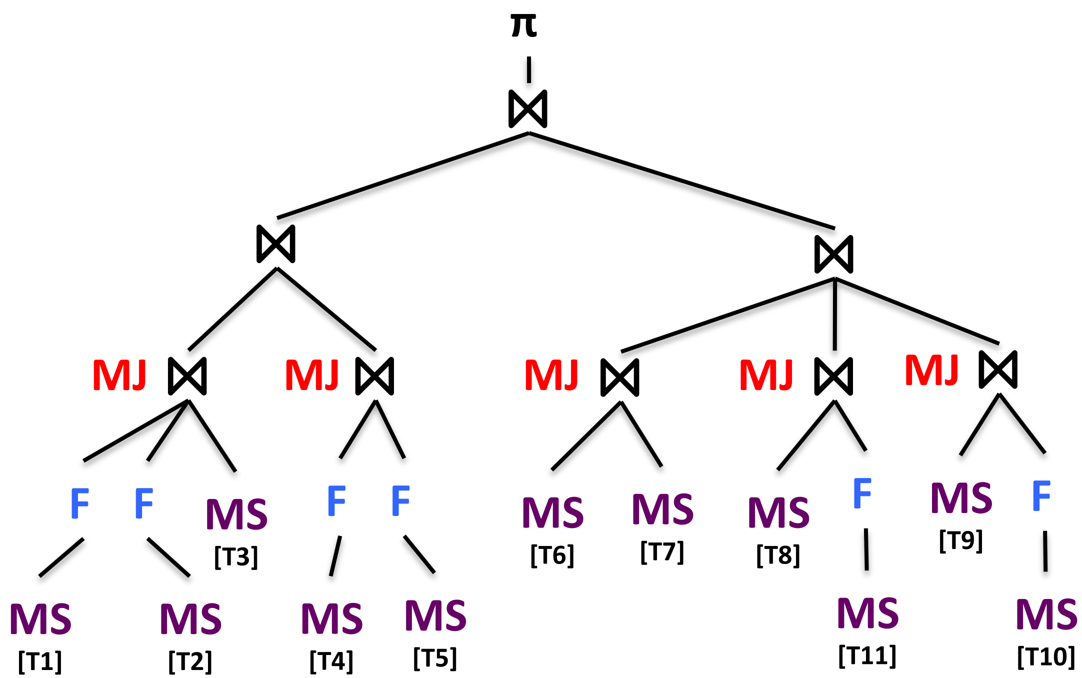
- Reading the triples from HDFS requires a Map Scan (**MS**) operator

Logical plan → Physical plan



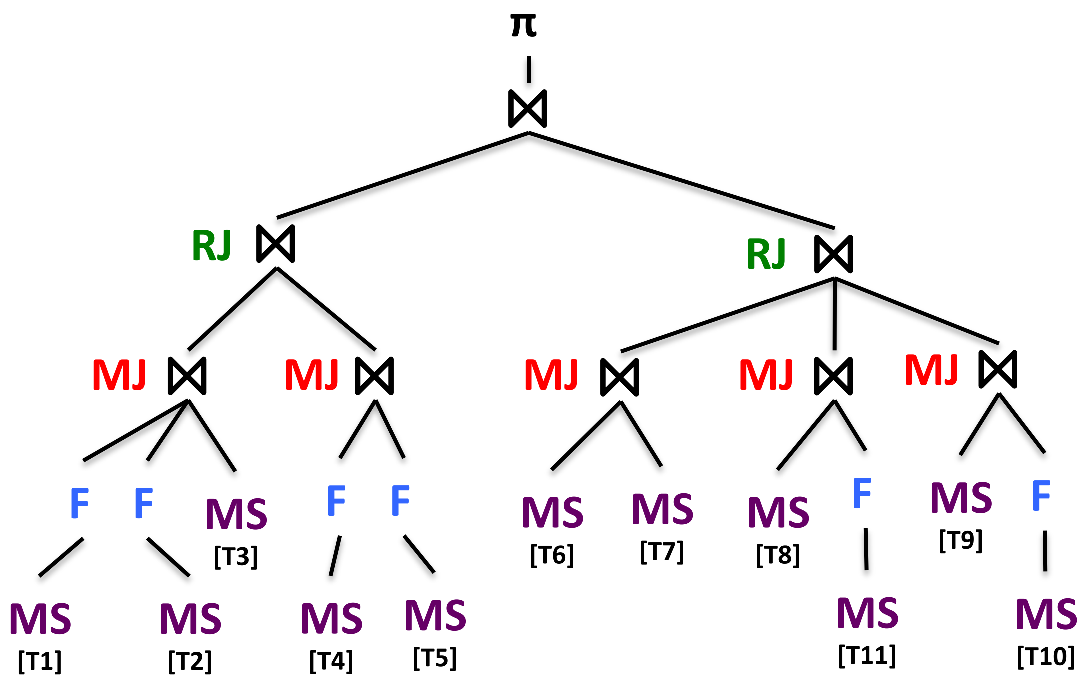
- Logical selections (**σ**) are translated to physical selections (**F**)

Logical plan → Physical plan



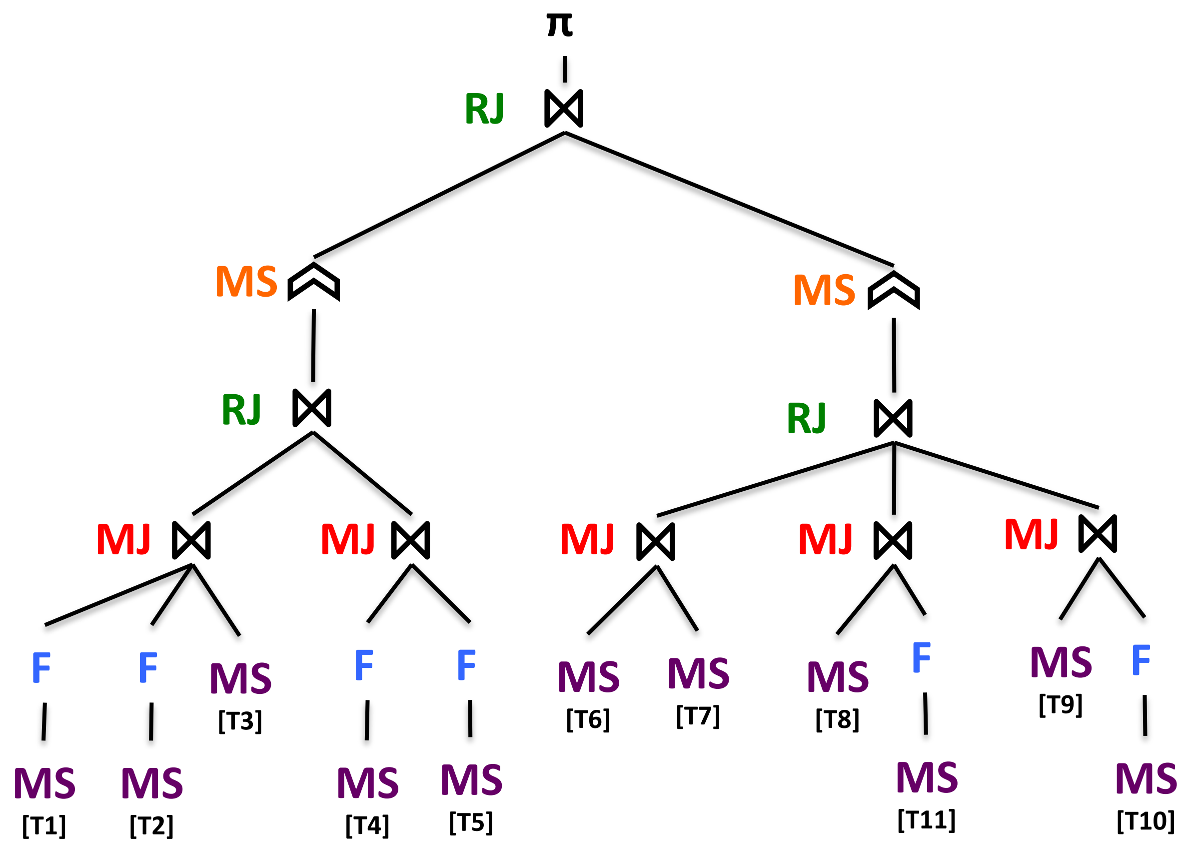
- First level joins are translated to Map side joins (MJ) taking advantage of the **data partitioning**

Logical plan → Physical plan



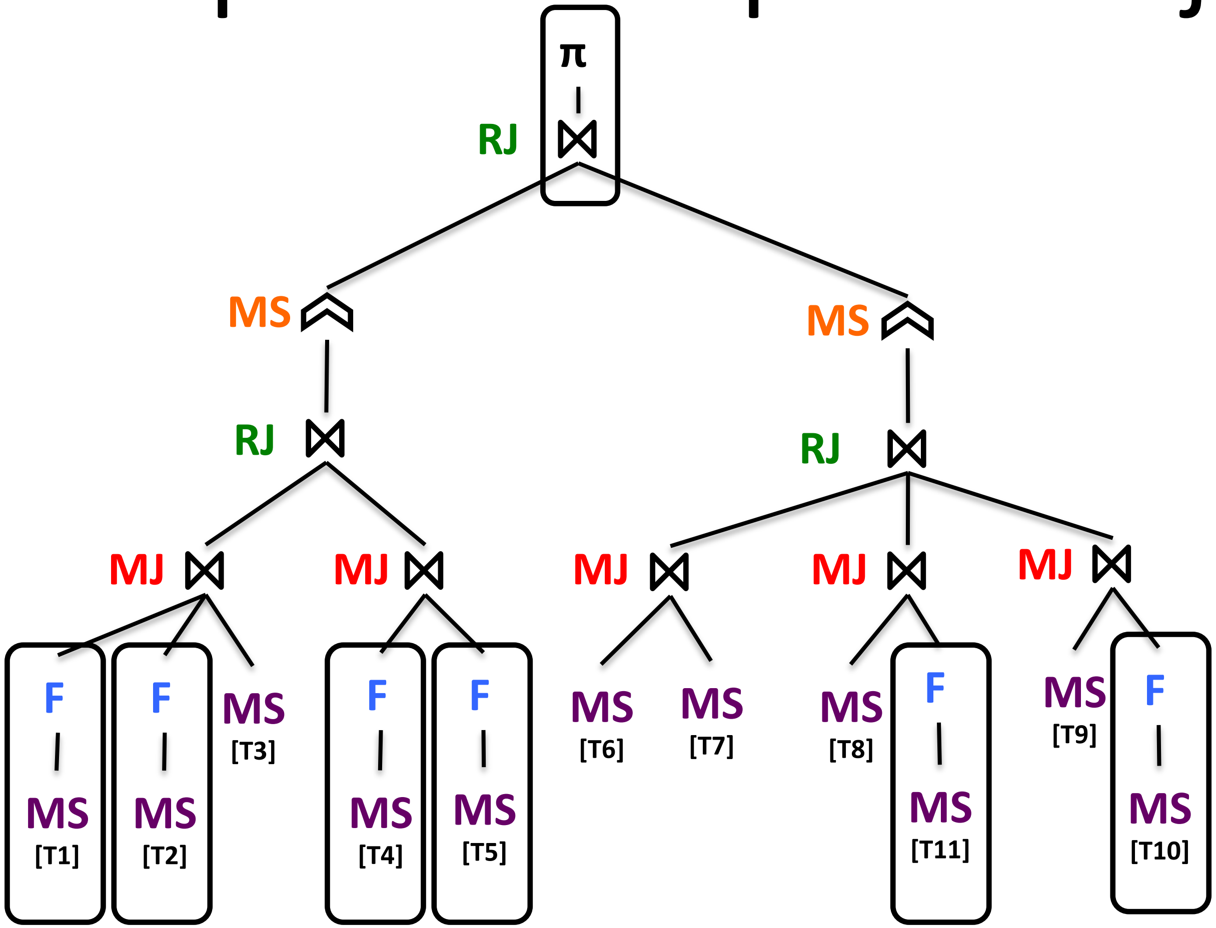
- All subsequent joins are translated to Reduce side joins (**RJ**)

Physical plan → MapReduce jobs



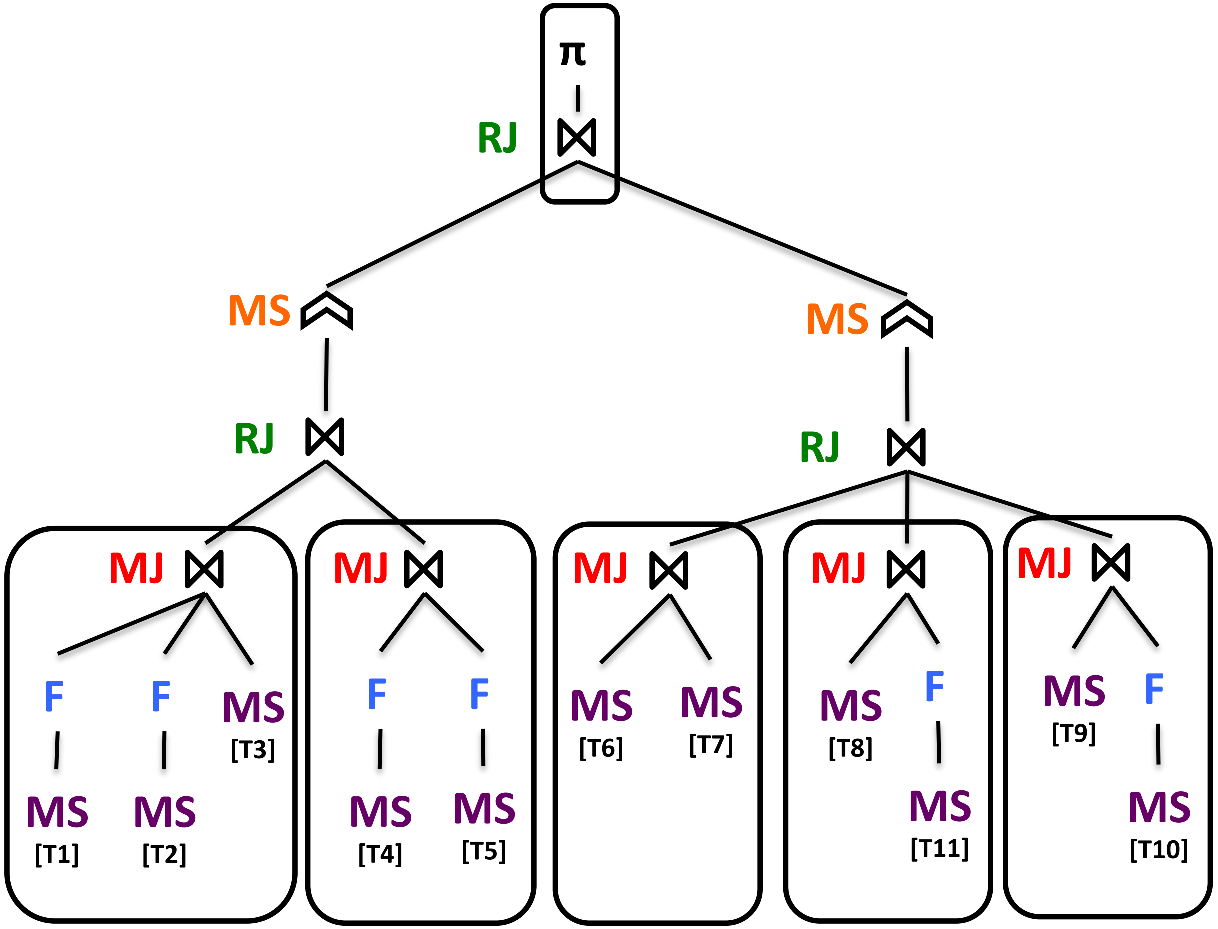
- Group the physical operators into Map/Reduce **tasks** and **jobs**

Physical plan → MapReduce jobs



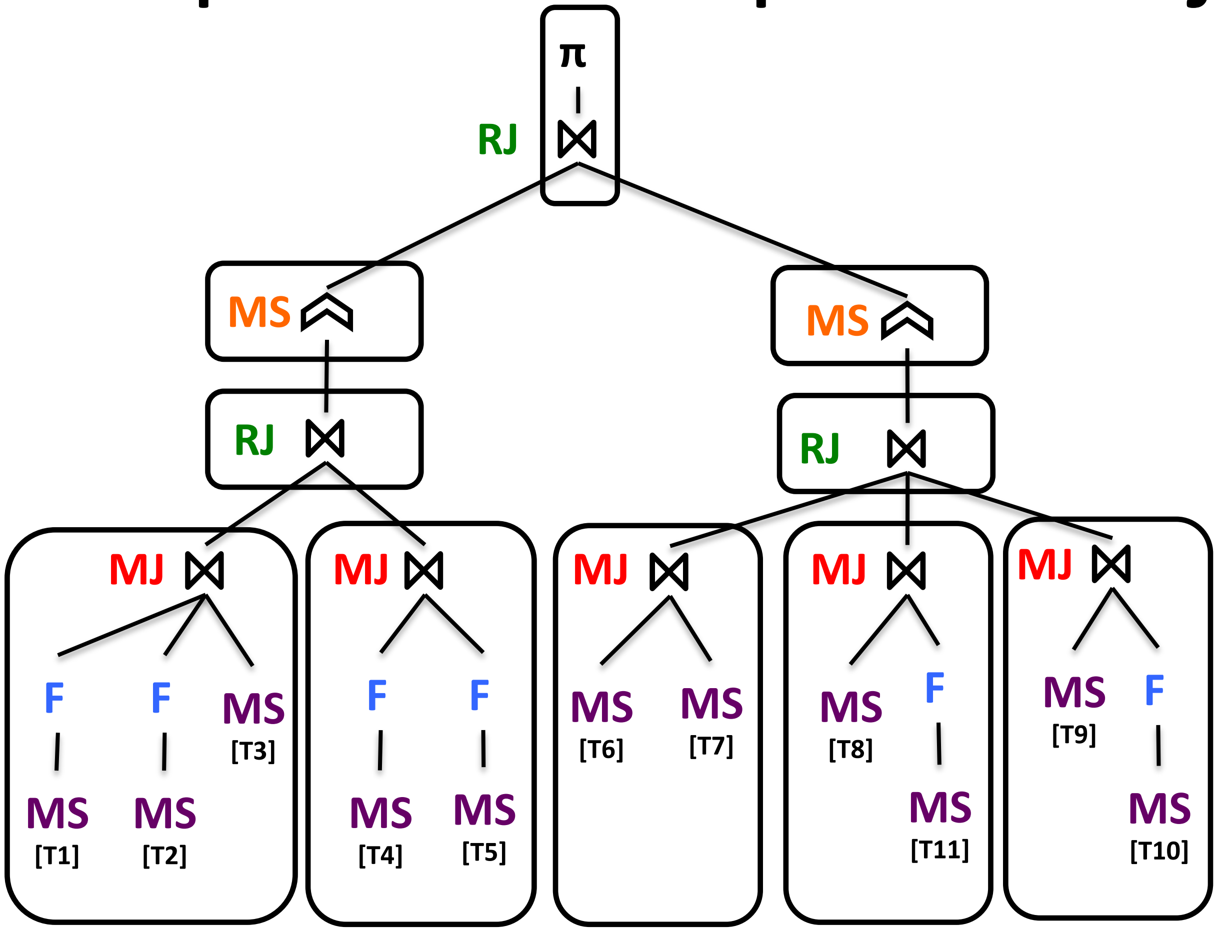
- Selections (F) and projections (π) belong to the same task as their child operator

Physical plan → MapReduce jobs



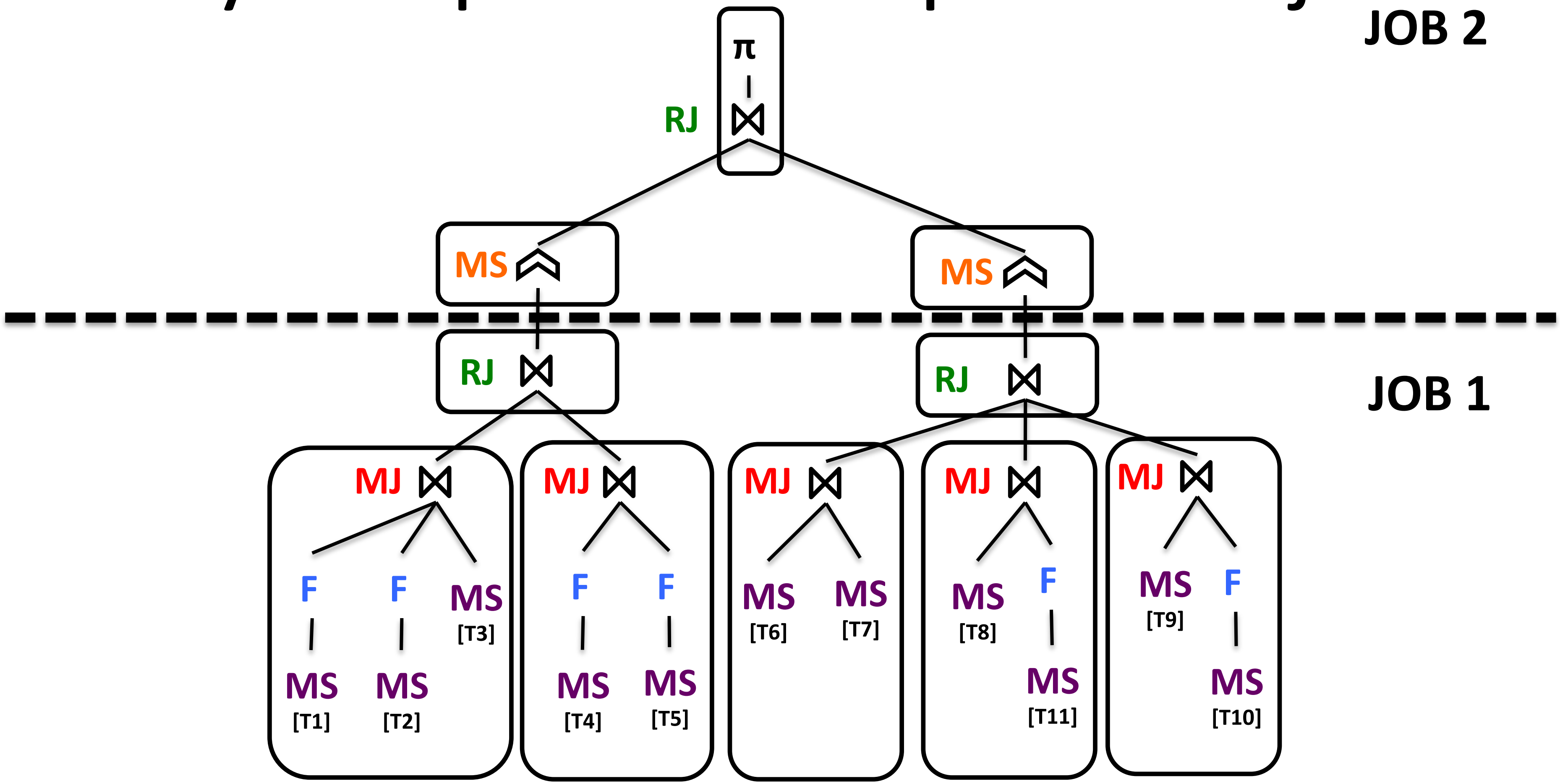
- Map joins (**MJ**) along with all their descendants are executed in the same task

Physical plan → MapReduce jobs



- Any other operator (**RJ** or **MS**) is executed in a separate task

Physical plan → MapReduce jobs

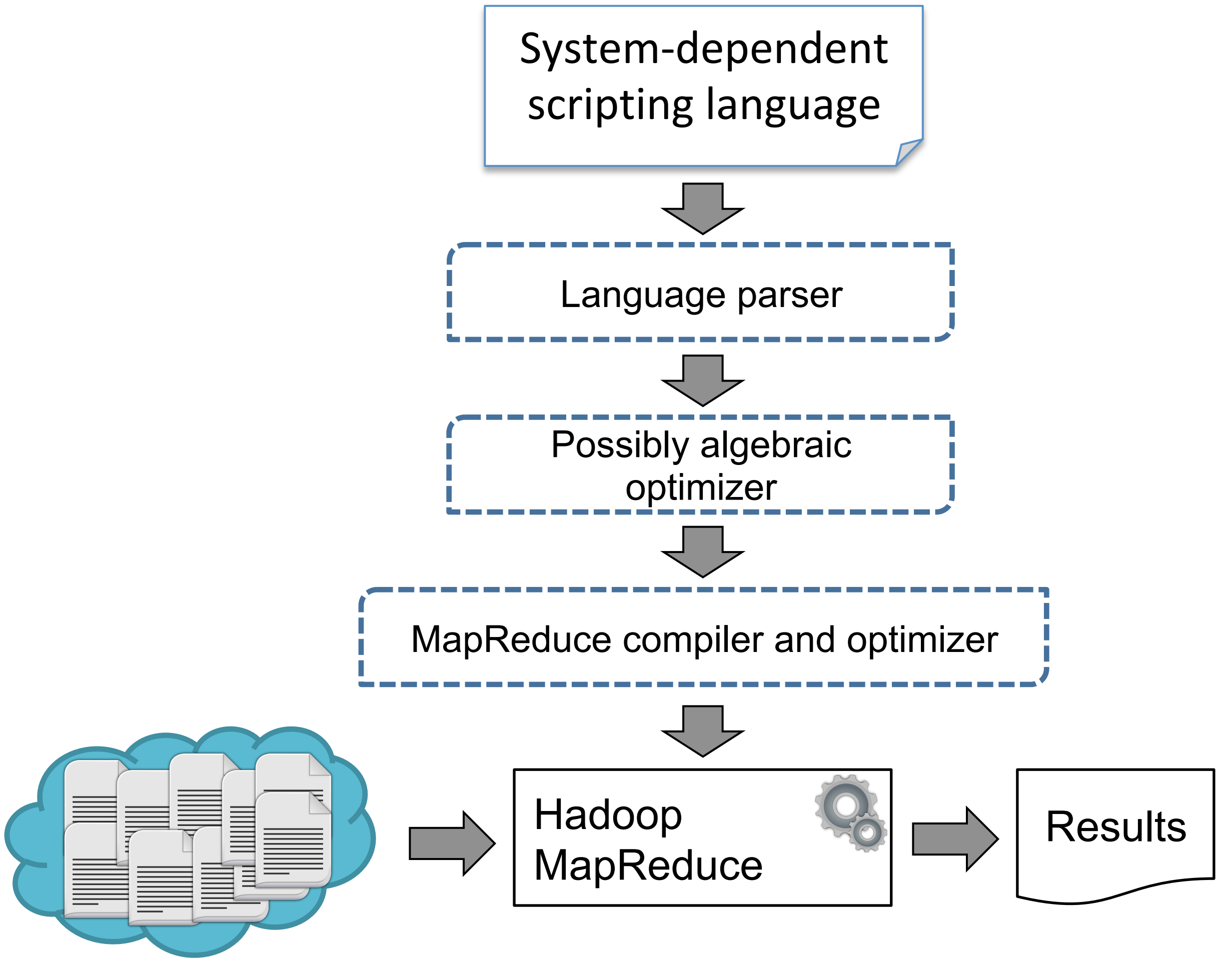


- **Tasks** are grouped into **jobs** in a bottom-up traversal

Structured DM on top of MapReduce

- We have seen:
  - Techniques for improving **data access selectivity** in a distributed file system (headers; multiple indexes)
  - Algorithms for **implementing operators**: select, project, join
  - **Query optimization** for massively parallel, n-ary joins
- Next:
  - A few highly visible **systems**
  - Some of their mechanisms for **consistency** in a distributed setting

Structured DM on top of MapReduce



Google Bigtable [CDG+06]

- One of the earliest NoSQL systems
- **Goal**: store data of varied form to be used by Google applications:
  – Web indexing, Google Analytics, Finance etc.
- **Approach**:
  – very large, heterogeneous-structure table
- Data model:

Row key → column key → timestamp → value

Different rows can have different columns, each with their own timestamps etc.

Google Bigtable

r1

c0	c1	c4	c7
…	…	…	…

r2

c1	c2	c3	c4	c5	c6
ts11:v1	ts21:v22 ts22:v22	ts31:v31 ts32:v32 ts33:v33	ts41:v41 ts42:v42	ts22:v51	ts61:v61 ts22:v62

Google Bigtable

- **Row key → column key → timestamp → value**
- Rows stored **sorted** in lexicographic order by the key
- Row range dynamically partitioned into **tablets**
  - Tablet = distribution / partitioning unit
- Writes to a row key are atomic
  - row = concurrency control unit
- Access control unit = **column families**
  - Family = typically same-type, co-occurring columns
  - « At most hundreds for each table »
  - E.g. **anchor** column family in Webtable

Apache projects around Hadoop

Hive: relational-like interface on top of Hadoop

- HiveQL language:

CREATE table pokes (foo INT, bar STRING);

SELECT a.foo FROM invites a WHERE a.ds='2008-08-15’;

FROM pokes t1 JOIN invites t2 ON (t1.bar = t2.bar) INSERT OVERWRITE TABLE events SELECT t1.bar, t1.foo, t2.foo;

+ possibility to plug own Map or Reduce function when needed…

Apache projects around Hadoop

APACHE HBASE

- **HBASE:** very large tables on top of HDFS («*goal: billions of rows x millions of columns* »), based on « *sharding* »
- Apache version of Google's BigTable [CDG+06] (used for Google Earth, Web indexing etc.)

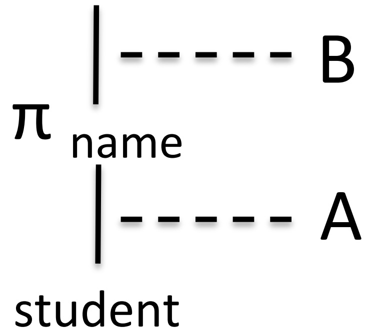
- Main strong points:
  - Fast access to individual rows
  - read/write consistency
  - Selection push-down (~ Hadoop++)
- Does not have: column types, query language, …

Apache projects around Hadoop

PIG: rich dataflow (« SQL + PL/SQL » style) language on top of Hadoop

Suited for many-step data transformations (« extract-transform-load »)

```
A = LOAD 'student' USING PigStorage()
     AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name;
DUMP B;
```

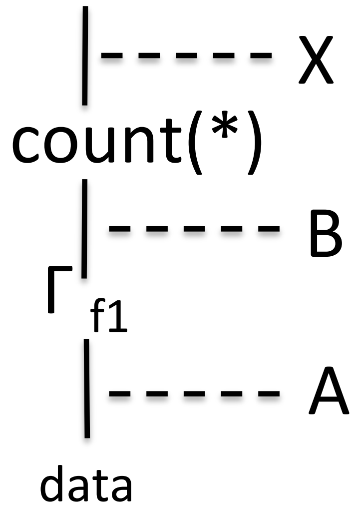


- Flexible data model (~ nested relations)
- Some nesting in the language (< 2 FOREACH ☺)

Apache projects around Hadoop

PIG: rich dataflow (« SQL + PL/SQL » style) language on top of Hadoop

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
DUMP A;
(1,2,3) (4,2,1) (8,3,4) (4,3,3) (7,2,5) (8,4,3)
B = GROUP A BY f1;
DUMP B;
(1,{(1,2,3)}) (4,{(4,2,1),(4,3,3)}) (7,{(7,2,5)})
(8,{(8,3,4),(8,4,3)})
X = FOREACH B GENERATE COUNT(A);
DUMP X;
(1L) (2L) (1L) (2L)
```



PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

r

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)} : B);
STORE F INTO 'S2out';
STORE F INTO 'S3out';
```

45% of the original s_1 + s_2 execution time

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

r

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)} : B);
STORE F INTO 'S2out';
STORE F INTO 'S3out';
```

Join

45% of the original s_1 + s_2 execution time

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

r

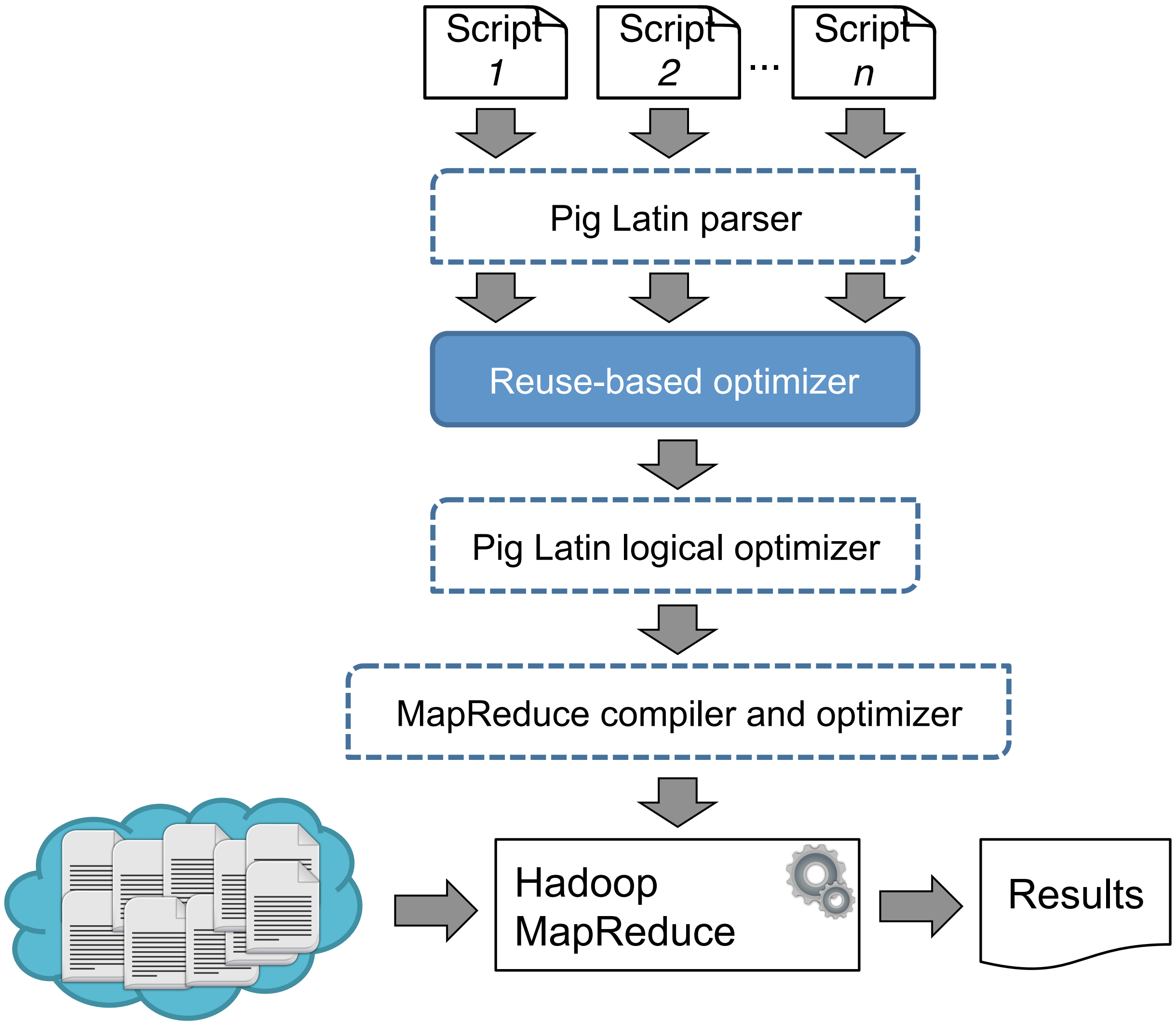
```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)} : B);
STORE F INTO 'S2out';
STORE F INTO 'S3out';
```

Join

Left outer join

45% of the original s_1 + s_2 execution time

Reuse-based optimizer within Pig [CCH+16]



Optimizer:

- **Translates** PigLatin programs into nested relational algebra for bags
- Applies equivalence laws to **identify repeated subexpressions**
- **Replaces** all but one of the subexpressions, **reuses** the result of the last
- Reduced execution time by x4

Spanner: A More Recent Google Distributed Database [CD+12]

- A few **Universes** (e.g. one for production, one for testing)
- Universe = set of zones
  - **Zone** = unit of administrative deployment
  - One or several zones in a datacenter
  - 1 zone = 1 **zone master** + 100s to 1000s of **span servers**
  - The zone master assigns data to span servers
  - Each span servers answers client requests
  - Each span server handles 100 to 1000 tablets
- **Tablet** = { key → timestamp → string }
- **Table** = set of tablets.

More on the Spanner data model

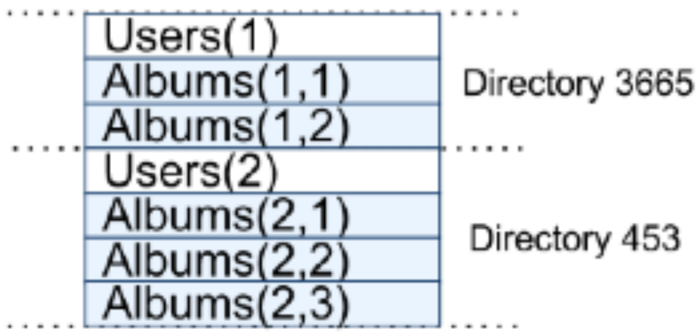
- Basic: **key → timestamp → value**
- **Directory** (or **bucket**): set of contiguous keys that share a common prefix
  - Data moves around by the bucket/directory
- On top of the basic model, applications see a **surface relational model**
  - Rows x columns (tables with a **schema**)
  - **Primary keys**: each table must have one or several primary-key columns

Spanner tables

- Tables can be organized in **hierarchies**
  - Tables whose primary key **extends the key of the parent** can be stored **interleaved** with the parent
  - Example: photo album metadata organized first by the user, then by the album

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



Spanner replication

- Used **for very high-availability** storage
- Store data with a **replication** factor (3 to 5)
- Applications can control:
  – Which datacenters control which data
  – How far data is from users (to control read latency)
  – How far replicas are from each other (to control write latency)
  – How many replicas are maintained
- Concurrency control relies on a **global timestamp mechanism** called « TrueTime » (see next)

Spanner TrueTime service

- TT.now() returns **a Ttinterval [earliest; latest]**
  - Uncertainty interval made explicit
  - The interval is guaranteed to contain the absolute time during which TT.now() was invoked
  - TrueTime clients **wait** to avoid the uncertainty
- Based on GPS and atomic clocks
  - Implemented by a set of **time master machines** per datacenter and a **time slave daemon** per machine
  - Every daemon polls a variety of masters to **reduce vulnerability** to
    - Errors from a single master
    - Attacks

Spanner consistency guarantees

- Linearizability:
  If transaction T1 commits before T2 starts
  Then the commit timestamp of T1 is guaranteed to be smaller than the commit timestamp of T2

→ globally meaningful commit timestamps

→ globally-consistent reads across the database at a timestamp

May not read the *last* version, but one from 5-10 seconds ago! (Last globally committed version.)

Spanner consistency guarantees

- Linearizability:
  If transaction T1 commits before T2 starts
  Then the commit timestamp of T1 is
  gu … it
  tin

→ gl … s

→ gl … abase

at a

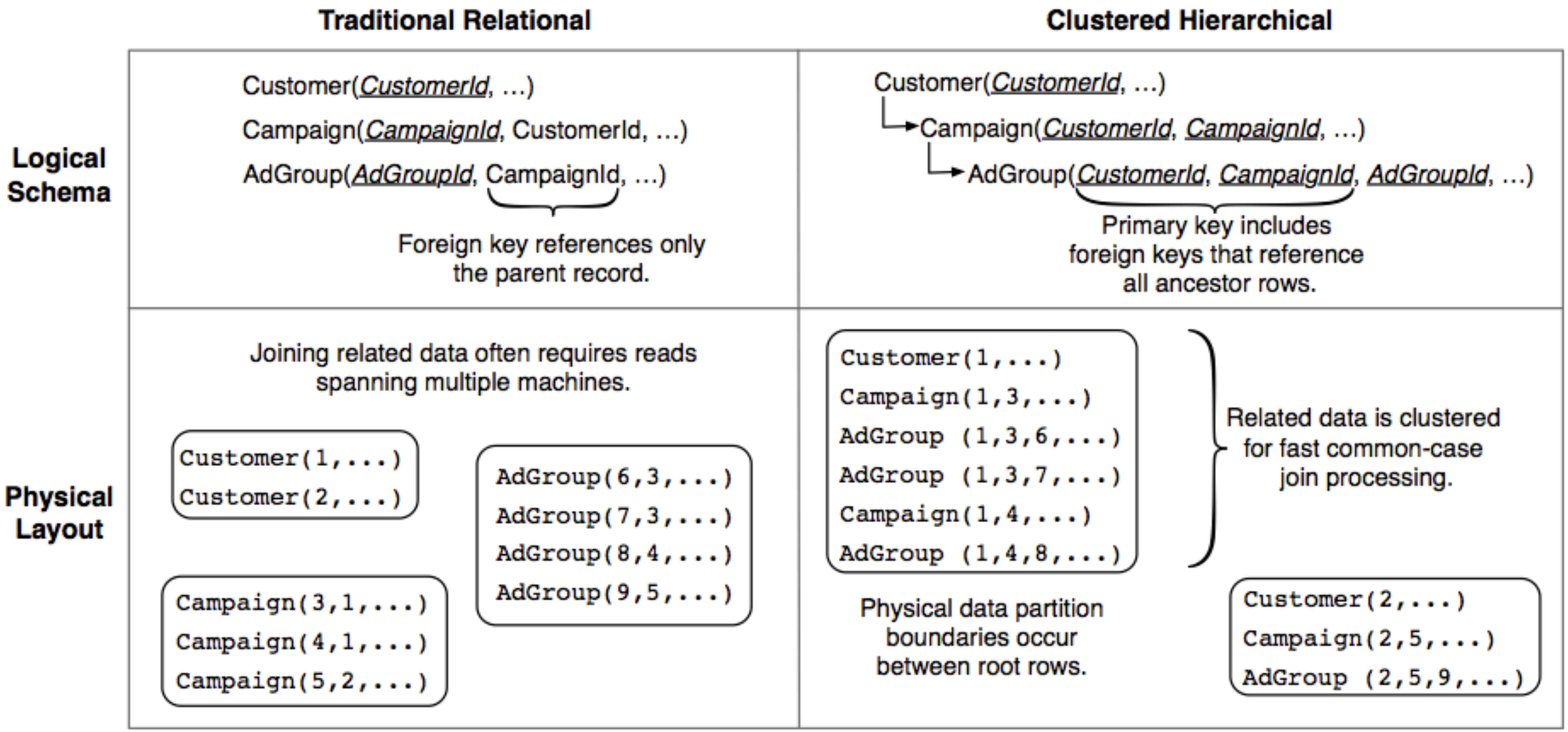
> « Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems it brings. We **believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.** »

F1: Distributed Database from Google [SVS+13]

- Built on top of Spanner
- Goals:
  - Scalability, availability
  - Consistency (almost ACID)
  - Usability (= full SQL + transactional indexes etc.)
- F1 from genetics « Filial 1 Hybrid » (cross mating of very different parental types)
  - F1 is a hybrid between relational DBs and scalable NoSQL systems

F1 data model

- Surface model: relational
- Storage: Clustered, inlined table hierarchies (Spanner)



Transactions in F1

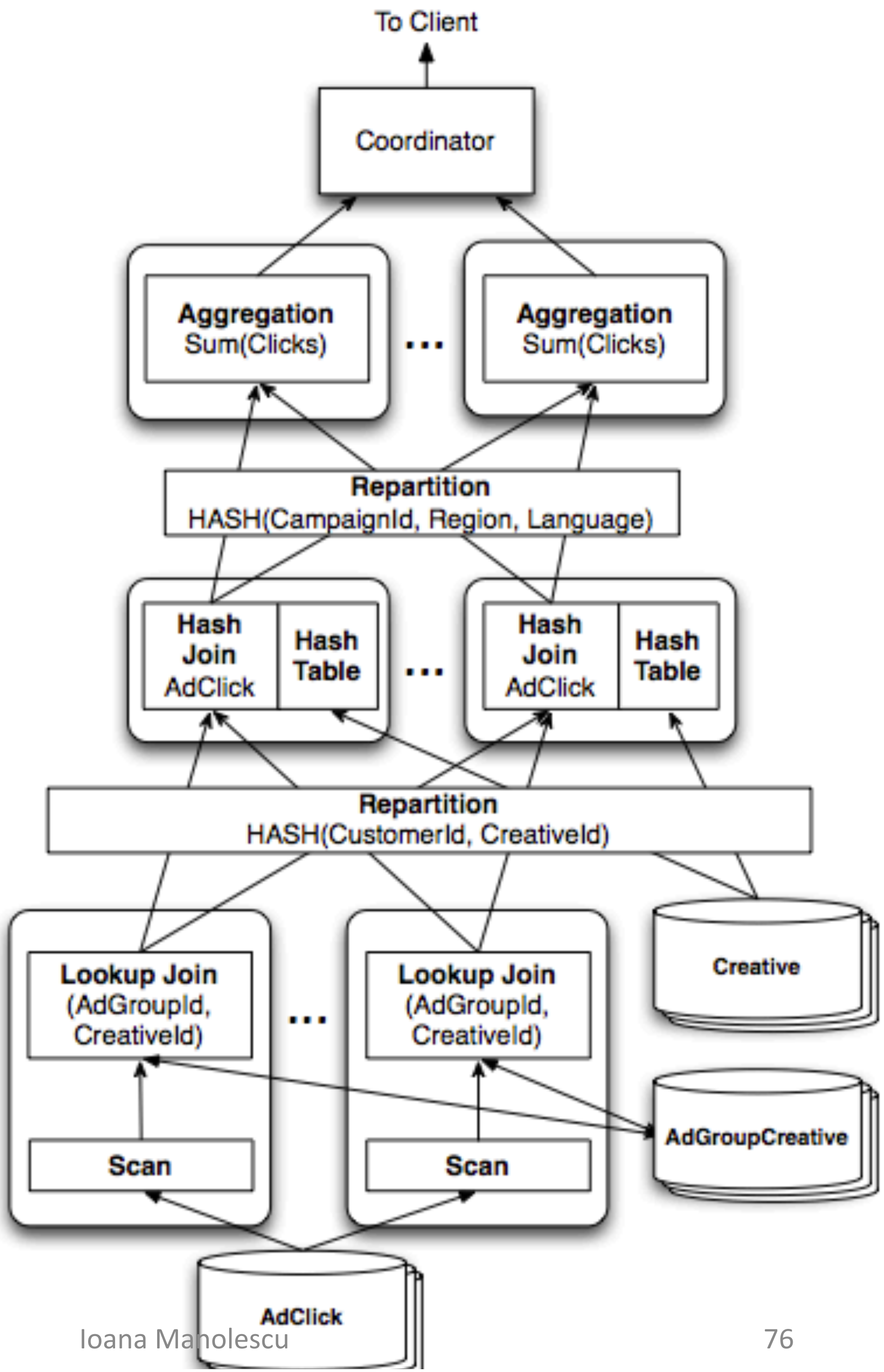
- **Snapshot** (read-only) transactions (no locks)
  - Read at Spanner's global safe timestamp, typically 5-10 seconds old, from a local replica
  - Default for SQL and MapReduce. All clients see the same data at the same timestamp.
- **Pessimistic** transactions (provided by Spanner)
  - Shared or exclusive locks; may abort
- **Optimistic** transactions
  - Read phase (no lock), then short write phase
  - Each row has last modification timestamp
  - To commit optimistic T1, F1 creates a short pessimistic T2 which attempts to read all of T1's rows. If T2 has a different version than T1, then T1 is aborted. Otherwise, T1 commits.

More on transactions in F1

- Benefits of optimistic transactions:
  - Reads never hold locks, never conflict with writes
    - Avoid performance drawback when a read runs for too long or aborts
    - Can run for a long time without hurting performance
- Self-contained: can be retried (after abort) at the F1 server, hiding transient Spanner errors
  - Pessimistic transactions cannot be retried at the server, because they require re-running client operations that took locks
- Drawbacks:
  - Concurrency control through last modif timestamp only works for existing rows → insertion phantoms
    - The same transaction may get different results in two successive reads of the same data
  - Low throughput if high contention as many transactions will abort (pessimistic ones will also abort in this case).

Query optimization in F1

```
SELECT agcr.CampaignId, click.Region,
       cr.Language, SUM(click.Clicks)
FROM AdClick click
  JOIN AdGroupCreative agcr
    USING (AdGroupId, CreativeId)
  JOIN Creative cr
    USING (CustomerId, CreativeId)
WHERE click.Date = '2013-03-23'
GROUP BY agcr.CampaignId, click.Region,
         cr.Language
```



OPEN PROBLEMS IN MASSIVELY PARALLEL DATA MANAGEMENT

Open problems in MapReduce-style processing

- The performance of a MapReduce execution in a Hadoop / Spark cluster depends on a large number of parameters

https://arxiv.org/pdf/1106.0940.pdf

Variable	Hadoop Parameter	Default Value	Effect
pNumNodes	Number of Nodes		System
pTaskMem	mapred.child.java.opts	-Xmx200m	System
pMaxMapsPerNode	mapred.tasktracker.map.tasks.max	2	System
pMaxRedPerNode	mapred.tasktracker.reduce.tasks.max	2	System
pNumMappers	mapred.map.tasks		Job
pSortMB	io.sort.mb	100 MB	Job
pSpillPerc	io.sort.spill.percent	0.8	Job
pSortRecPerc	io.sort.record.percent	0.05	Job
pSortFactor	io.sort.factor	10	Job
pNumSpillsForComb	min.num.spills.for.combine	3	Job
pNumReducers	mapred.reduce.tasks		Job
pInMemMergeThr	mapred.inmem.merge.threshold	1000	Job
pShuffleInBufPerc	mapred.job.shuffle.input.buffer.percent	0.7	Job
pShuffleMergePerc	mapred.job.shuffle.merge.percent	0.66	Job
pReducerInBufPerc	mapred.job.reduce.input.buffer.percent	0	Job
pUseCombine	mapred.combine.class or mapreduce.combine.class	null	Job
pIsIntermCompressed	mapred.compress.map.output	false	Job
pIsOutCompressed	mapred.output.compress	false	Job
pReduceSlowstart	mapred.reduce.slowstart.completed.maps	0.05	Job
pIsInCompressed	Whether the input is compressed or not		Input
pSplitSize	The size of the input split		Input

Table 1: Variables for Hadoop Parameters

Open problems in MapReduce-style processing

- The performance of a MapReduce execution in a Hadoop / Spark cluster depends on a large number of parameters

  https://arxiv.org/pdf/1106.0940.pdf

- Even when hidden to (casual) users, these parameters i**mpact the performance** of a job
- ... while the choices made also impact the **monetary costs**

- How to **automatically set the values** for these parameters, while respecting users' budget constraints and ensuring efficient execution?
  - Cost-based optimization
  - Learning and re-setting parameters during execution
- Jobs combine SQL-style and ML processing
- Iterative; low response time

European Research Council

Yanlei Diao (Ecole Polytechnique)

Open problems in MapReduce-style processing

- For optimization, we need to understand when two computations are equivalent
  - Similar to algebraic equivalence
- When can we push a selection below a classifier?
  - Need to reason about the properties of ML operation
- When is the partial result of a (ML+SQL) job reusable for future computations?
  - Similar to view-based query rewriting
  - Declarative data analytics [MV19]

Declarativeness criteria for data analytics systems

1. **Data abstractions**: matrices, vectors, tables etc. are available as abstractions and independently from their implementation (e.g., sparse/dense etc.)
2. **Data processing operators**: join, group by etc. are available in the platform (do not need to be coded)
3. **Advanced analytics operators**: linear algebra, probability distribution are available in the platform
4. **Plan optimization**: users' programs automatically optimized by the system

Declarativeness criteria for data analytics systems

5. **Lack of control flow**: the user does not have access to control flow constructs, which specify a given order of execution
6. **Automatic computation of the solution**: the parameters of a machine learning model should be computed by the system in a way transparent to the user.
7. **No need for code with unknown semantics**: such code hinders/breaks the optimization process

Example: benchmark task for declarativeness of data analytics tools

- **Predict** the median value of Boston suburban houses based on a number of features about a suburb
  - e.g., crime rate, distance from employment centers etc.
  - Use linear regression with gradient descent to minimize error

$$\hat{y} = \sum_{i=1}^{m} x_i w_i,\ m = number\ of\ features$$

$$\min \sum_{i=1}^{n} (\hat{y}_i - y_i)^2,\ n = number\ of\ training\ observations$$

- Before training, preprocess (**filter**) the data to locations very close to Charles river
- How can this be implemented in different systems? What do users still need to do in a non-declarative fashion?

Example: PigLatin on the declarative analytics benchmark

- Supports:
  - Data abstractions, data processing operators, plan optimization, no control flow, UDF-free operators
- Does not support:
  - ML parameter tuning
  - Iteration (loop-until)!
    - Therefore, this had to be coded outside the main PigLatin script (write to file...)
    - This compromises declarativeness <u>and</u> performance

Example: Spark on the declarative analytics benchmark

- Supports:
  - Data processing operators
  - At least some linear algebra operators
- Does not (fully) support:
  - Linear algebra, to the extent that users have to be aware, e.g., of the details of various matrix implementations, and not all operations (e.g. transpose) are available on all of them...
  - Libraries exist for this (e.g. Breeze) but require data conversion code
  - Operators are essentially 2nd order functions (invisible semantics)
  - User isolation from the control flow

Declarative data analytics benchmark results (1)

System/Language	Independence of Data Abstractions	DP Ops	ML Ops	Plan Optimization	Lack of Control Flow	Automatic Computation of Solution	Lack of Code with Unknown Semantics	Scope
Pig Latin	✓	✓		✓	✓		✓	DP
Jaql	✓	✓		✓	✓		✓	DP
U-SQL	✓	✓		✓	✓		✓	DP
Spark		✓	✓	✓				DP, ML
Flink (Stratosphere)		✓		✓				DP
DryadLINQ		✓		✓				DP
Tupleware		✓		✓				DP
SystemML	✓		✓	✓			✓	ML
Mahout Samsara			✓	✓			✓	ML

Declarative data analytics benchmark results (2)

System/Language	Independence of Data Abstractions	DP Ops	ML Ops	Plan Optimization	Lack of Control Flow	Automatic Computation of Solution	Lack of Code with Unknown Semantics	Scope
BUDS	✓		✓		✓		✓	ML
TensorFlow			✓	✓		✓	✓	ML
SciDB	✓	✓	✓	✓	✓		✓	DP, ML
LogicBlox	✓	✓		✓	✓	✓	✓	DP, LP/QP
MLog			✓		✓	✓	✓	ML
ReLOOP		✓				✓	✓	LP/QP
An extension of SQL with linear algebra [36]	✓	✓	✓	✓	✓		✓	DP, ML

Conclusion

- Data management based on MapReduce: brave new world
  - Large storage and computing capabilities
  - Re-design/re thinking from scratch the multiple layers
  - ML gaining ground

References

- [BPERST10] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita and Y. Tian, “A Comparison of Join Algorithms for Log Processing in MapReduce,” in SIGMOD 2010.
- [LMDMcGS11] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. "*A Platform for Scalable One-Pass Analytics using MapReduce*", ACM SIGMOD 2011
- [DQRSJS] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, Jorg Schad. "*Only Aggressive Elephants are Fast Elephants*", VLDB 2012
- [Goasdoué2015] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz and S. Zampetakis. "*CliqueSquare: Flat plans for massively parallel RDF Queries*", ICDE 2015
- [JQD11] A.Jindal, J.-A.Quiané-Ruiz and J.Dittrich. "*Trojan Data Layouts: Right Shoes for a Running Elephant*" SOCC, 2011
- [MW19] N. Makrynioti and V. Vassalos. "Declarative Data Analytics: A Survey", 2019
- [Wu2017] Buwen Wu ; Yongluan Zhou ; Hai Jin ; Amol Deshpande. "*Parallel SPARQL Query Optimization*", ICDE 2017