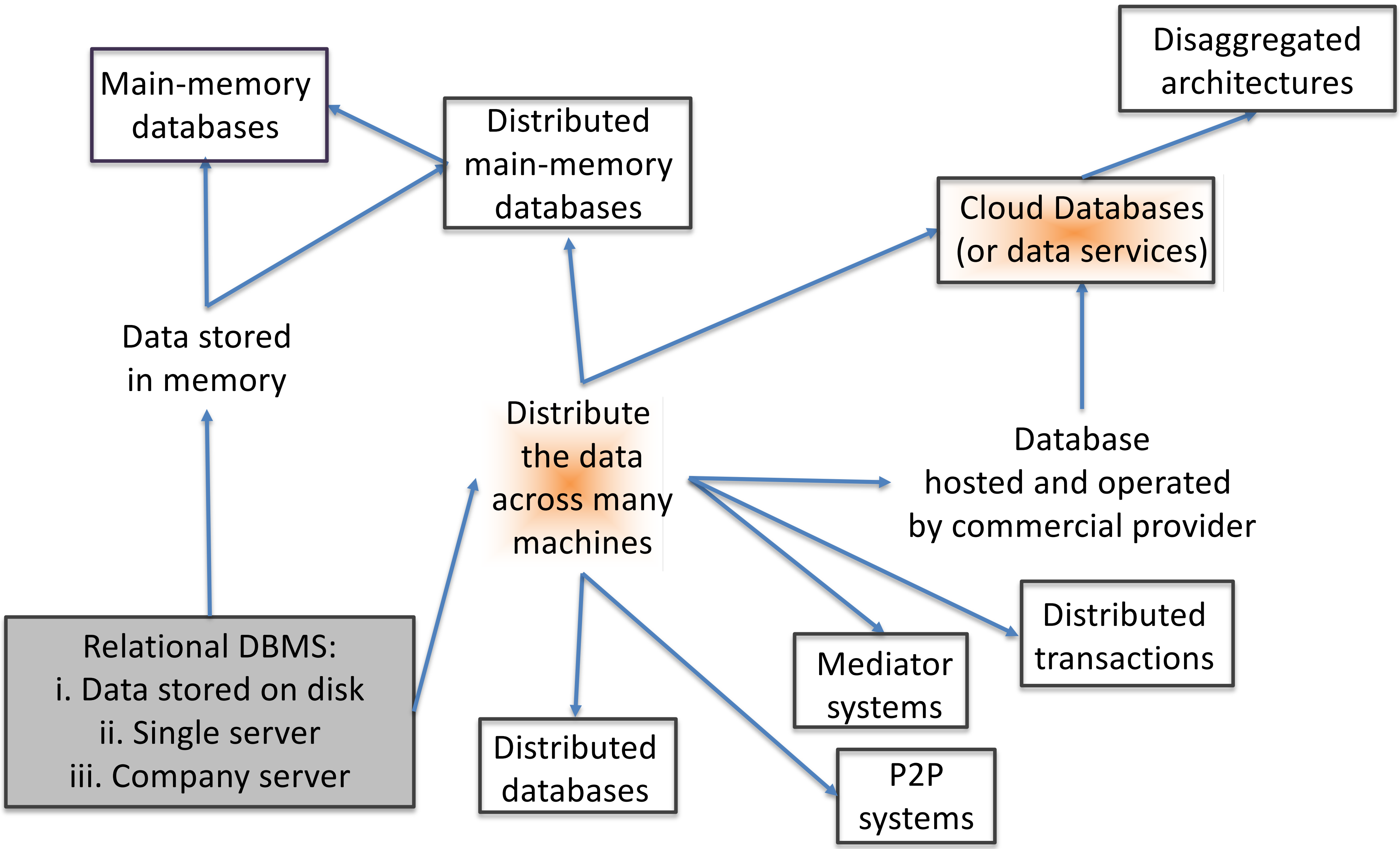
Architectures for Big Data

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/

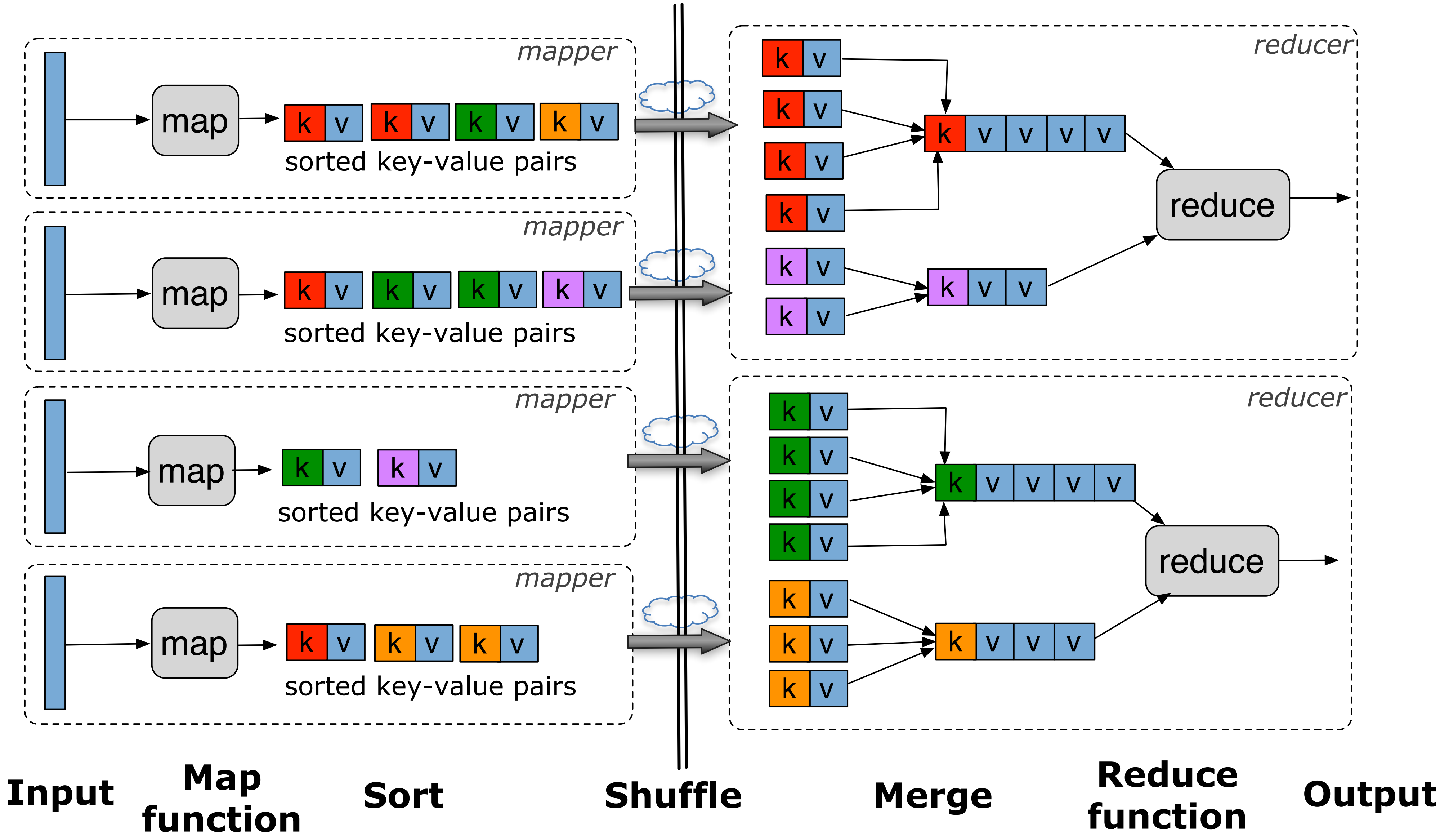
From databases to Big Data



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
  - Query optimization revisited for MapReduce (also multi-query optimization)
  - A few visible Big Data platforms implemented on top of MapReduce clusters
  - Some open problems in this area

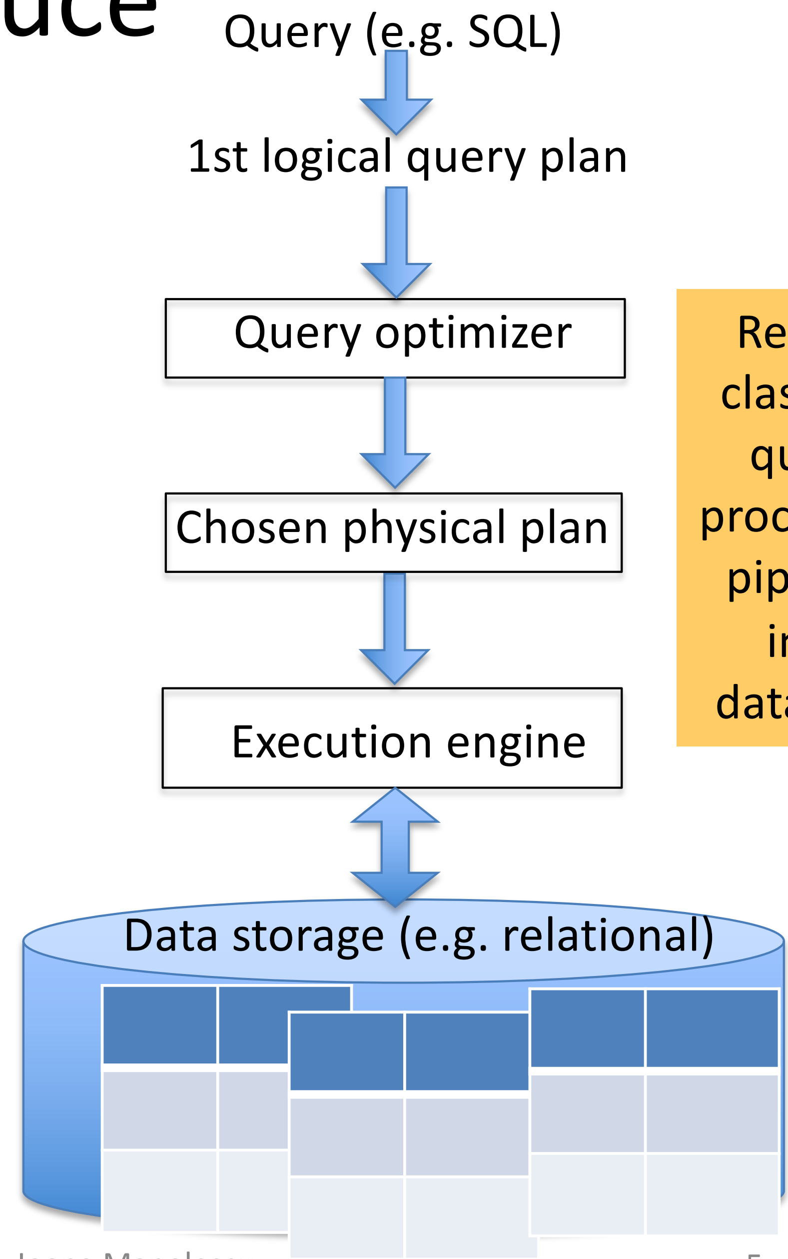
Recall: Map/Reduce outline



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

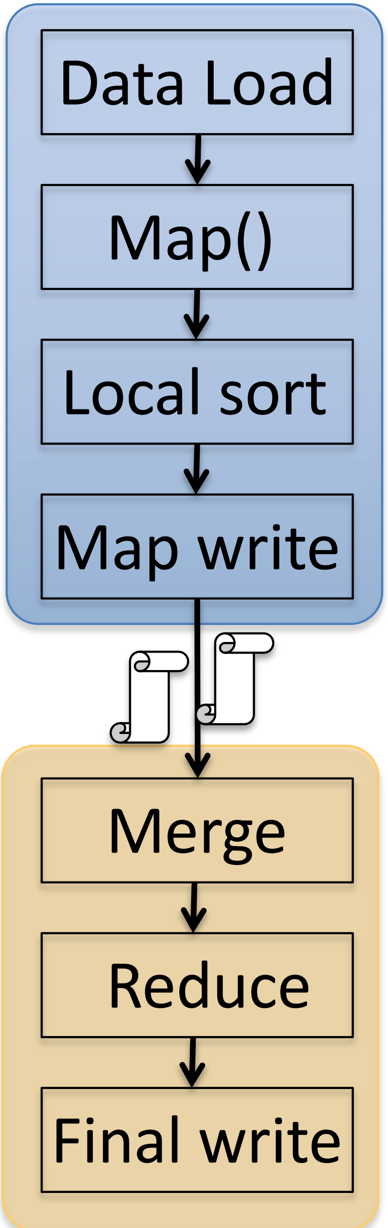


Recall: classical query processing pipeline in a database

IMPROVING DATA ACCESS PERFORMANCE IN A DISTRIBUTED FILE SYSTEM

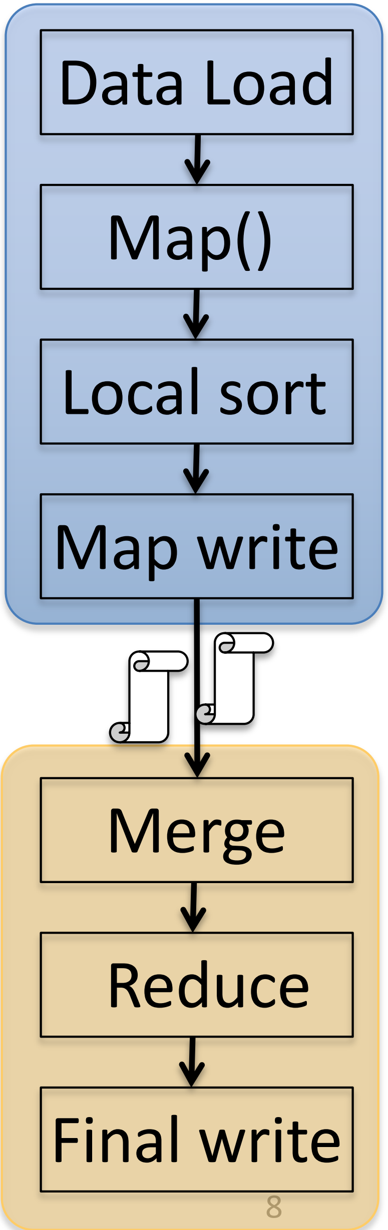
Data access in Hadoop

- 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



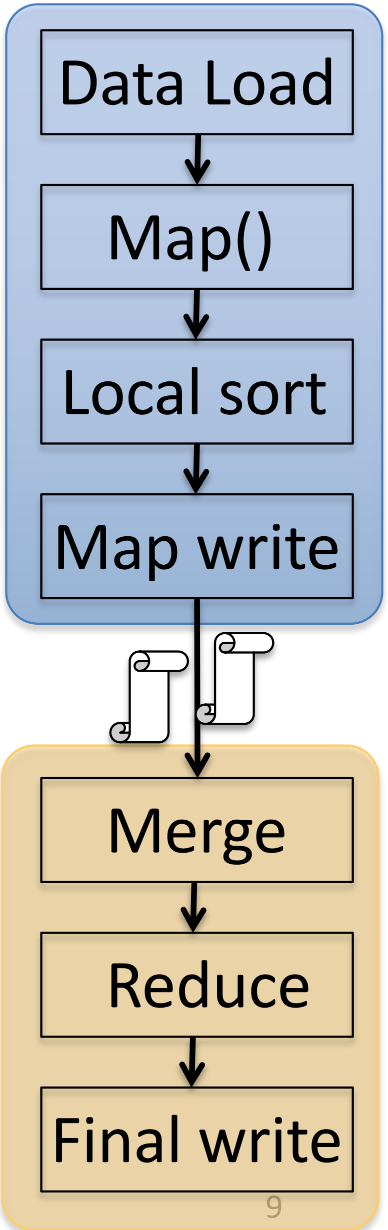
Accelerating data access in Hadoop

- 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

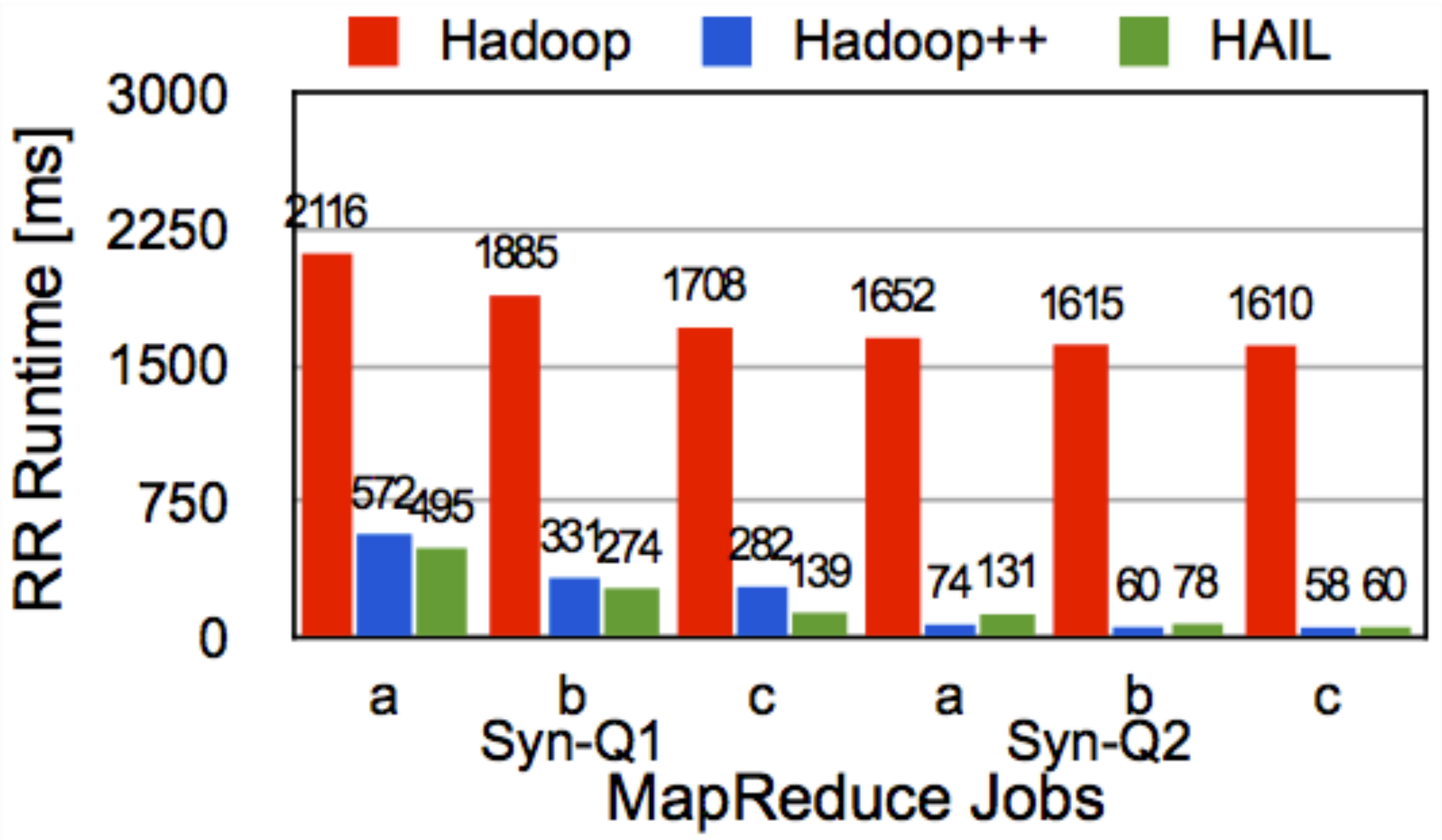


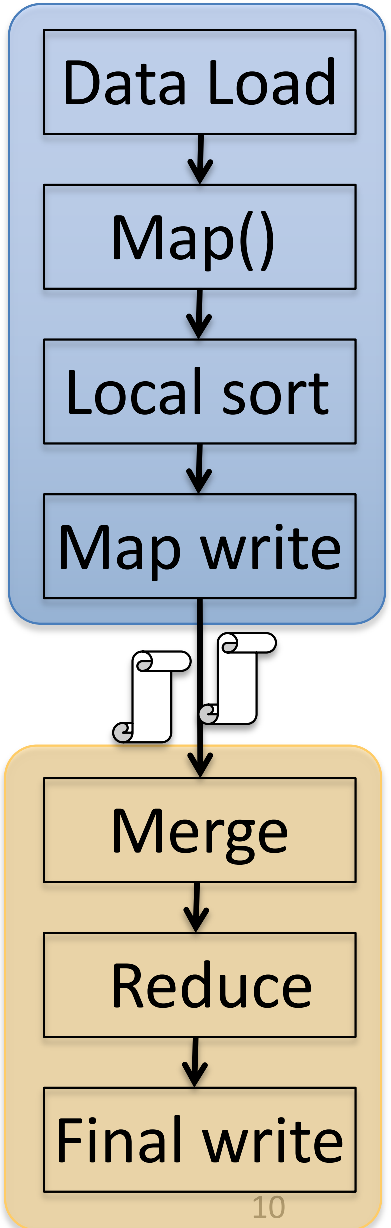
Accelerating data access in Hadoop

- 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



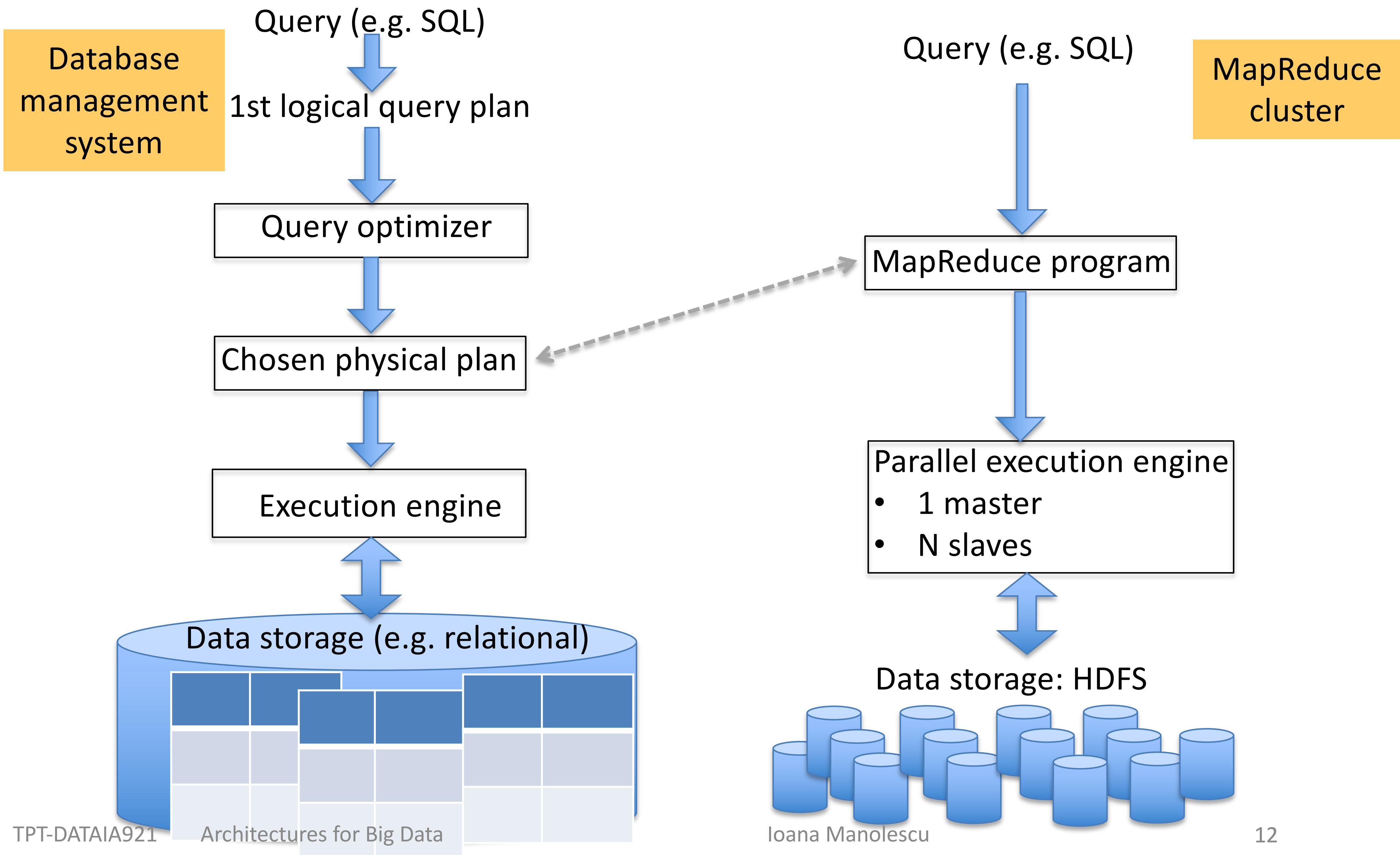
Accelerating data access in a Hadoop-like distributed file system





STRUCTURED BIG DATA MANAGEMENT
THROUGH THE MAPREDUCE FRAMEWORK

First idea: write a MapReduce program for each query



First idea: write a MapReduce program for every query

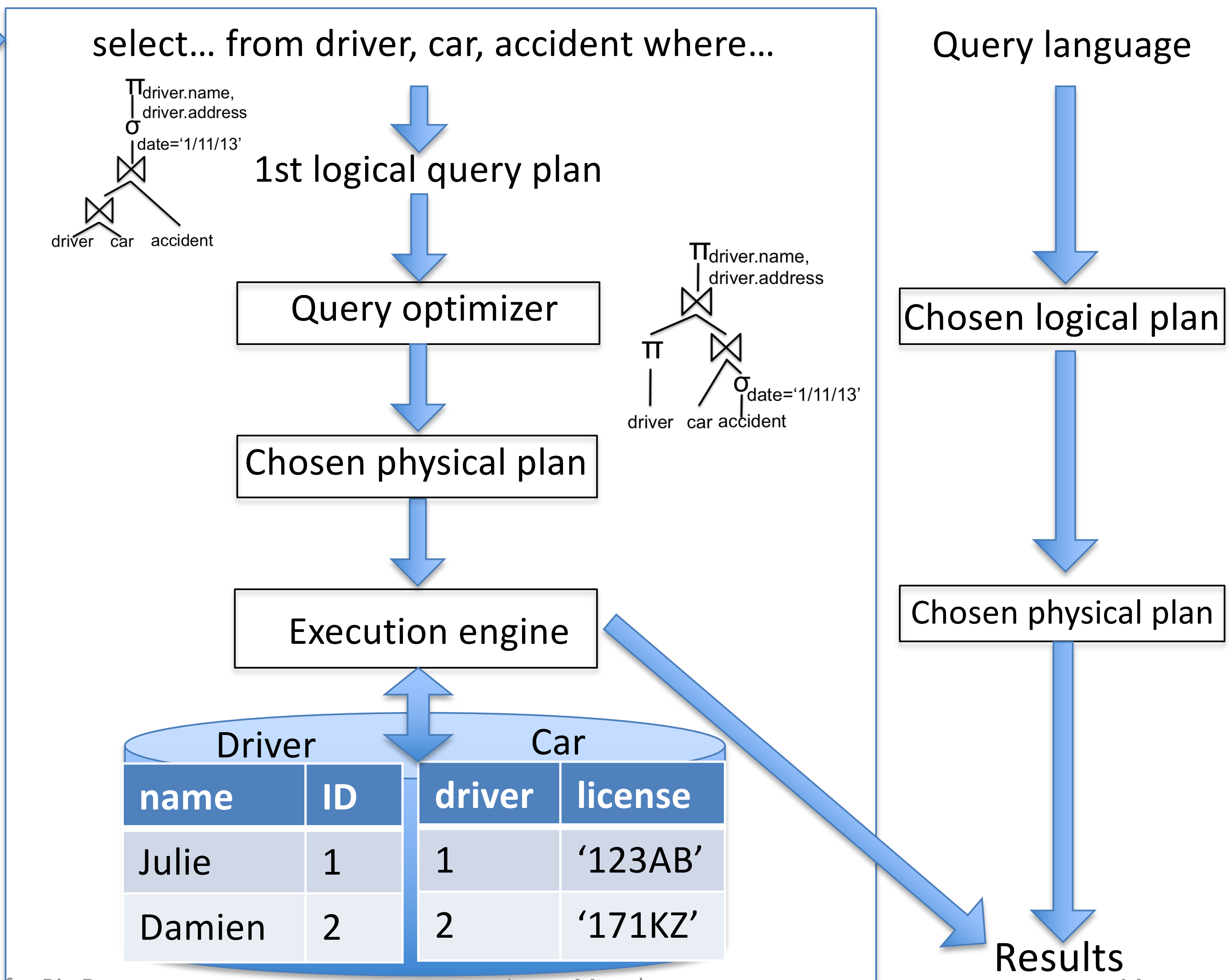
Examples:

- 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

Users did less work when using a DBMS! How to regain this for Big Data?

SQL

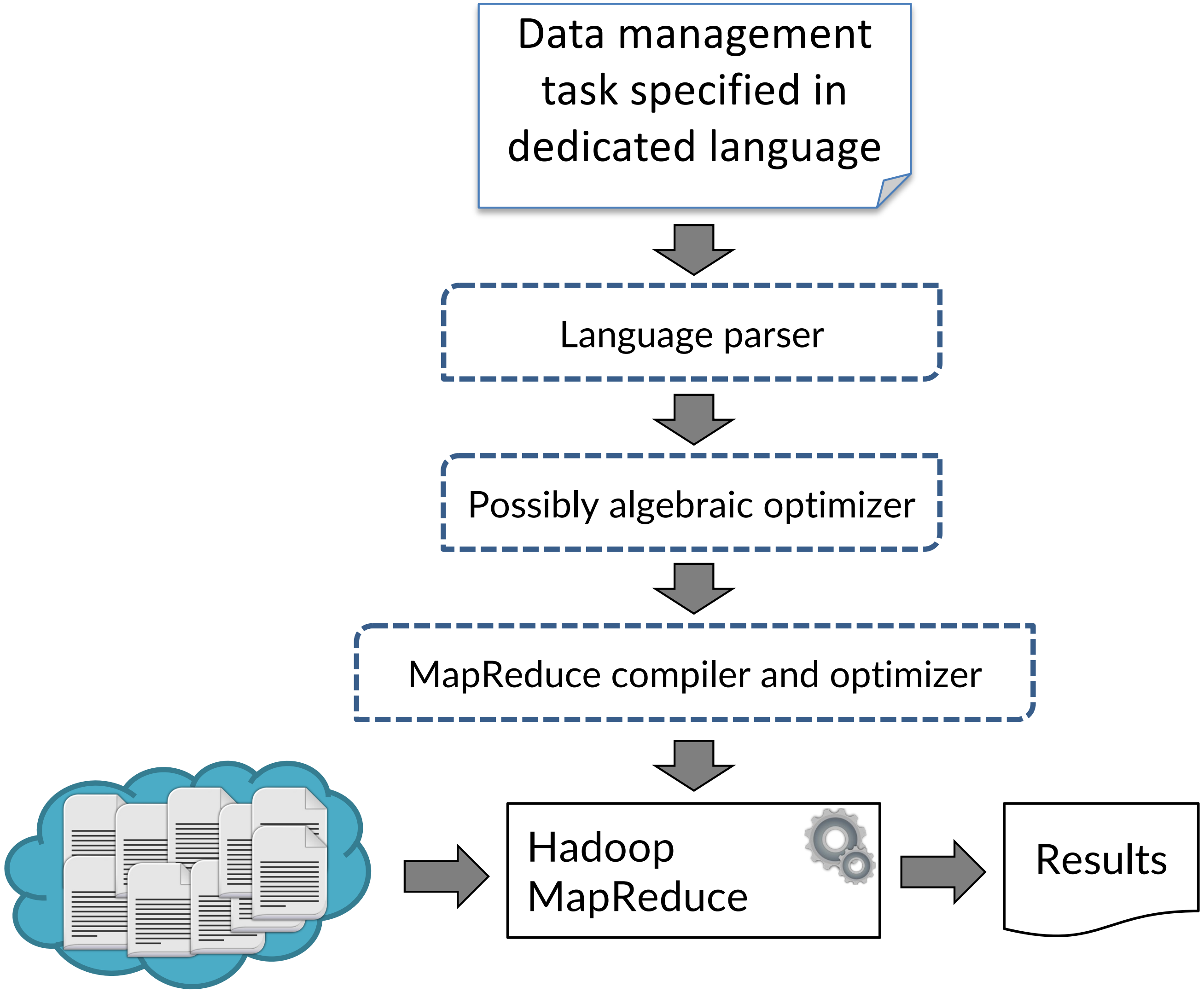
select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license='123AB'



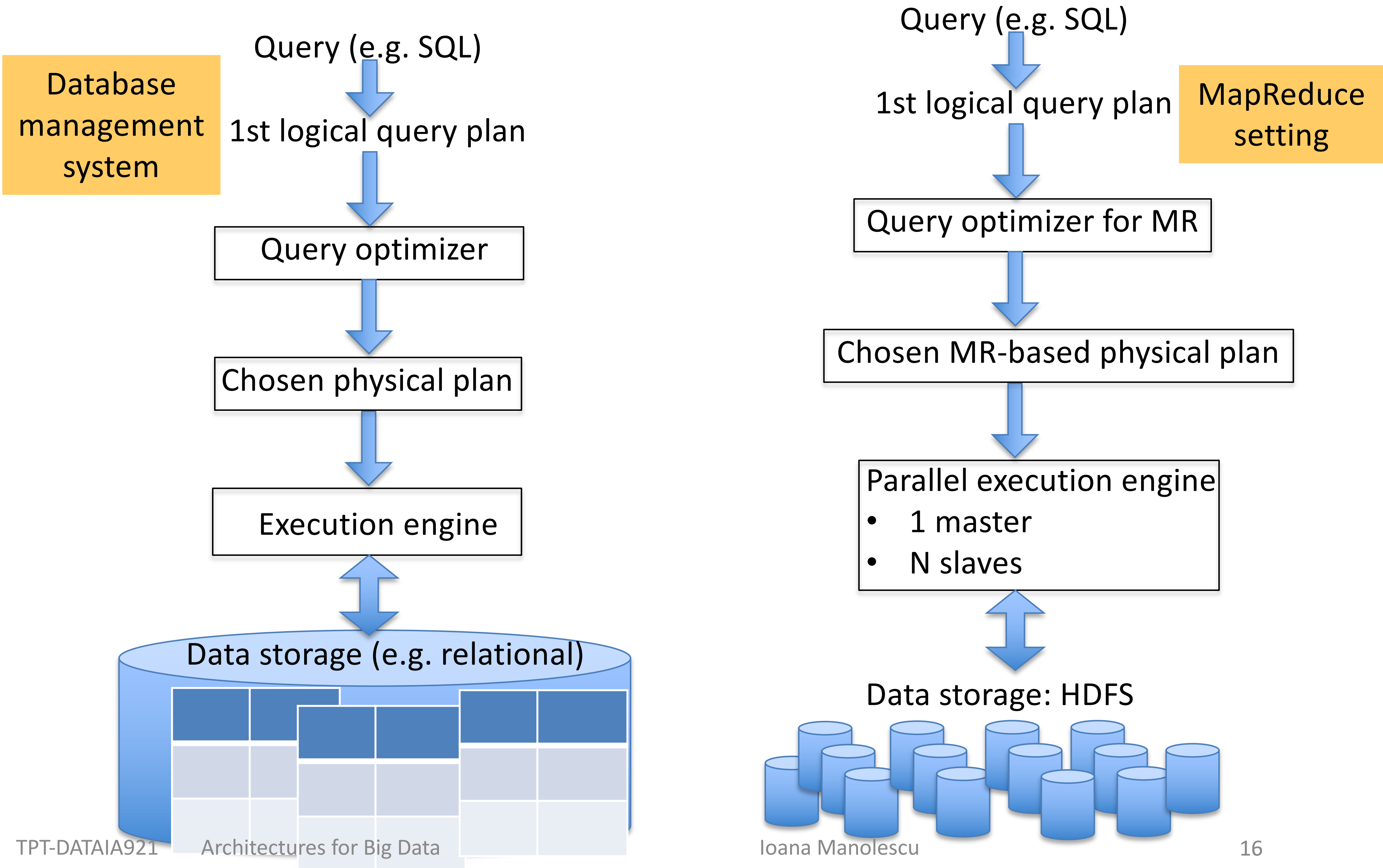
name	ID
Julie	1
Damien	2

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

Second idea: new architecture for structured DM on top of MapReduce



Use a MapReduce program for every physical operator



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));
```

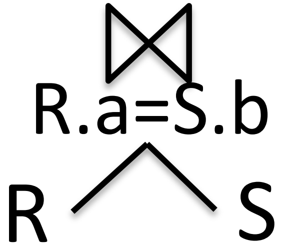
Hash join: // builds a hash table in memory

```
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:

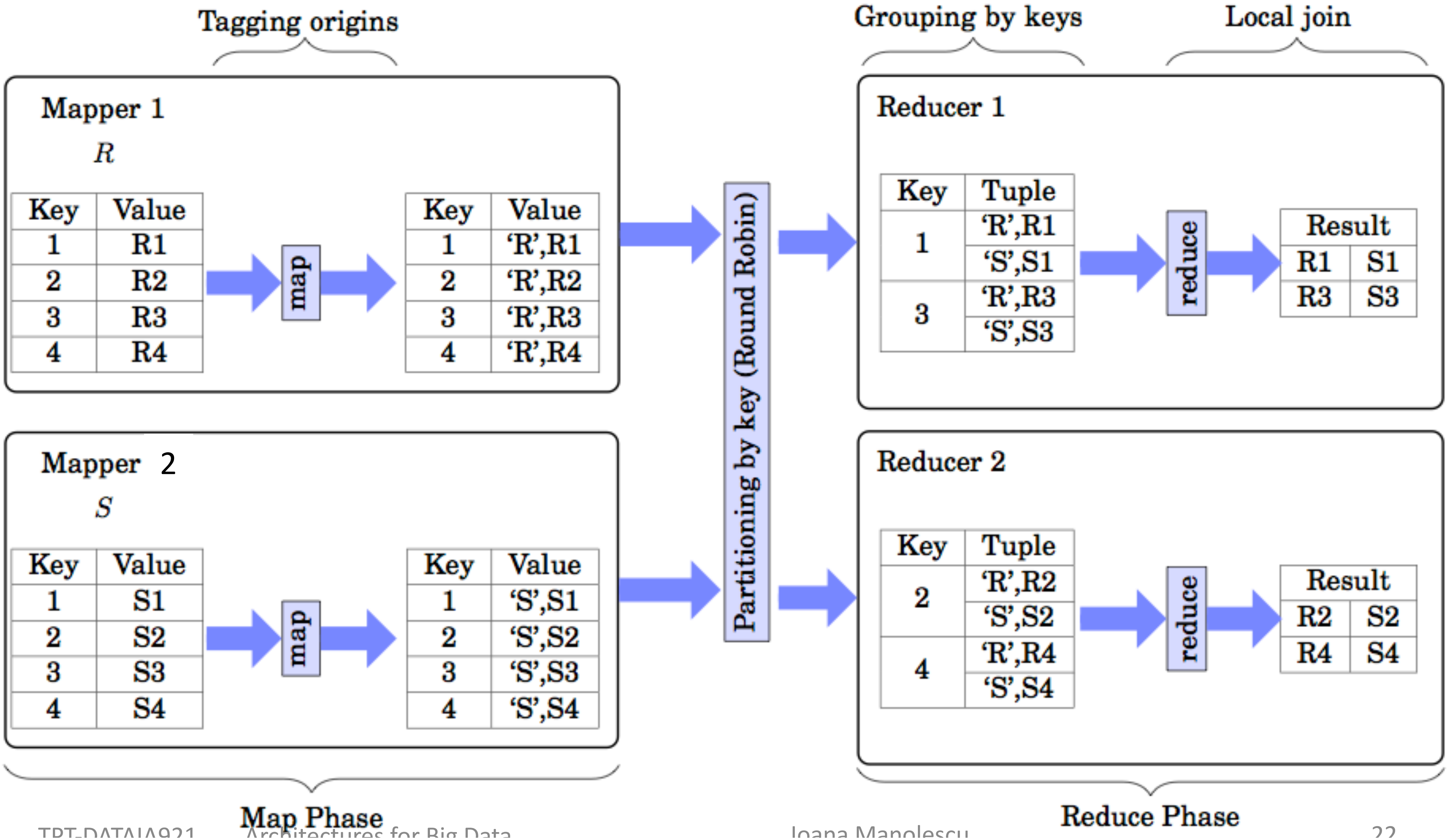
- Output (t.a, («R», t)) for each t in R
- Output (t.b, («S», t)) for each t in S

Reducer:

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

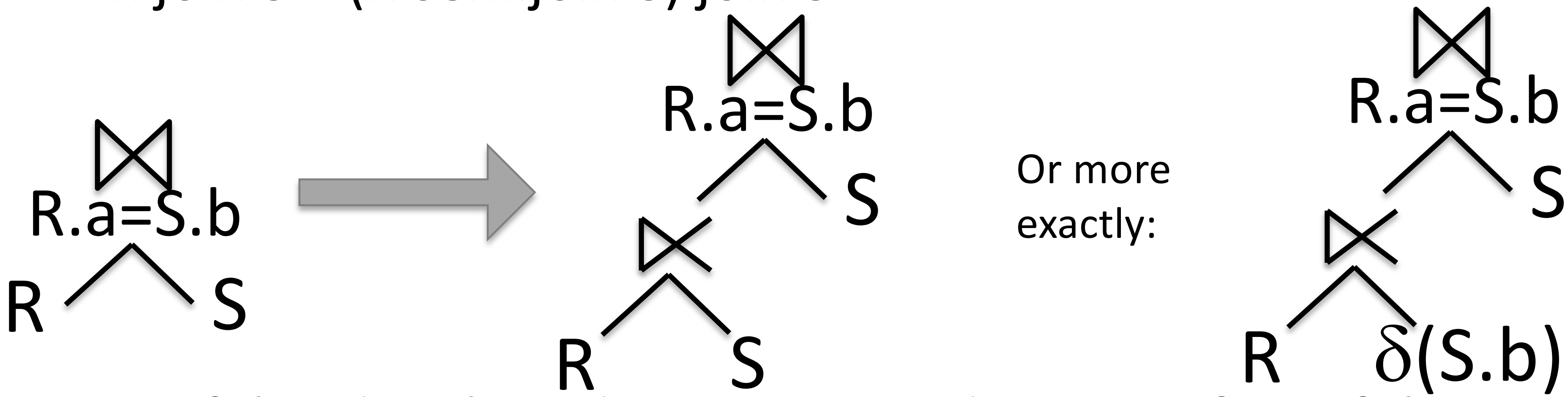
Implementing equi-joins on MapReduce (1)
Repartition join

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



Implementing equi-joins on MapReduce (2)

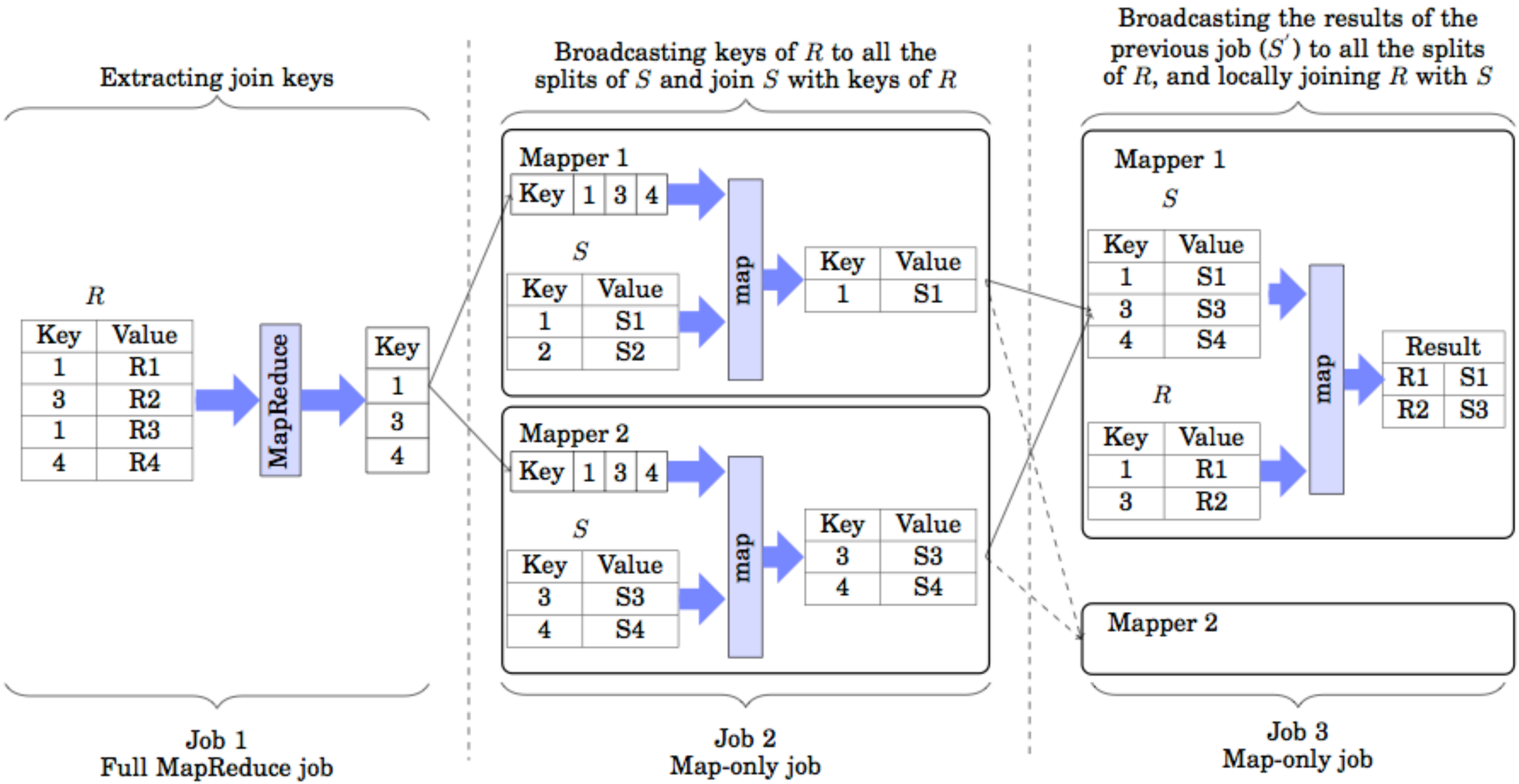
- **Semijoin-based MapReduce join**
- Recall: semijoin optimization technique:
  - R join S = (R semijoin S) join S



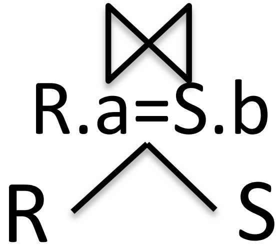
  - Useful in distributed settings to reduce transfers: *if the distinct S.b values* are smaller than *the non-matching R tuples*
  - Symetrical alternative: R join S = R join (S semijoin R)

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!

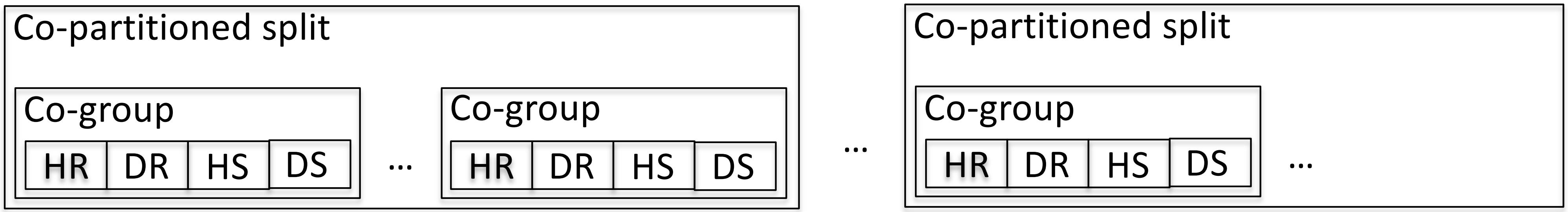
- Example: S is a *log* data collection (e.g. log table)
- R is a *reference* table e.g. with user names, countries, age, …
- Facebook: 6 TB of new log data/day

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

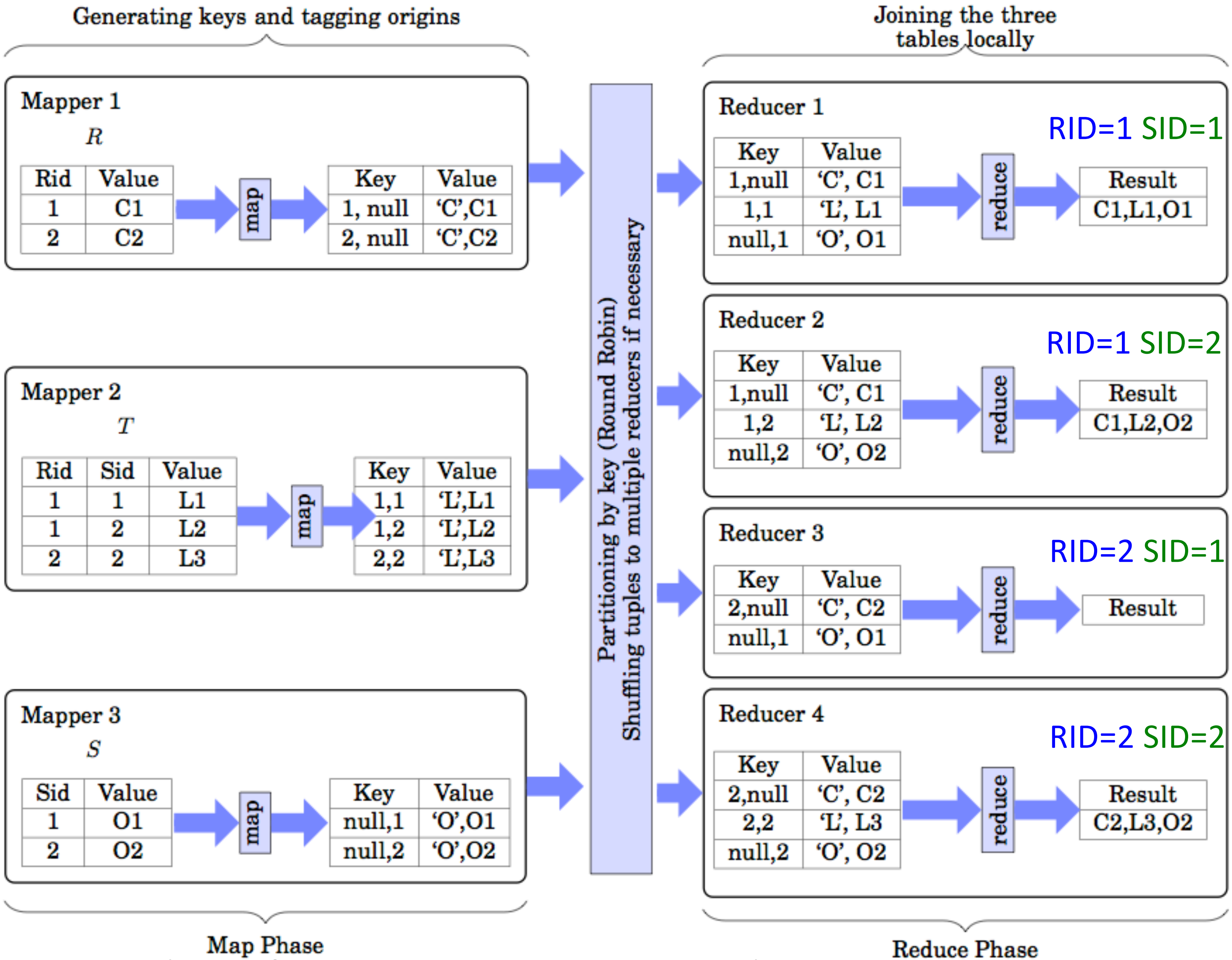
Algorithm	+	-
Repartition Join	Most general	Not always the most efficient
Semijoin-based Join	Efficient when semijoin is selective (has small results)	Requires several jobs, one must first do the semi-join
Broadcast Join	Map-only	One table must be very small
Trojan Join	Map-only	The relations should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

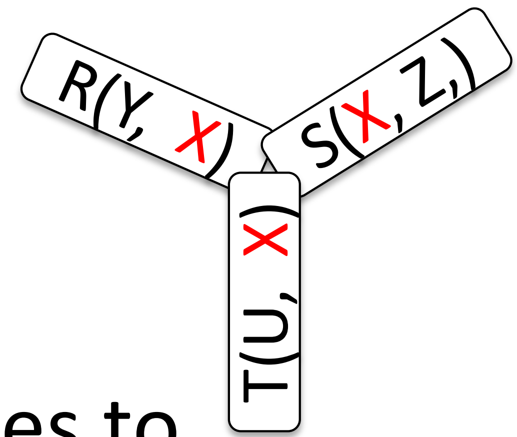
- Three relations, two join attributes (RID and SID)
- Split the SIDs into Ns groups and the RIDs in Nr groups. Assume Nr x Ns reducers available.
- Hash **T** tuples according to a composite key made of the two attributes. Each **T** tuple goes to one reducer.
- Hash **R** and **S** tuples on *partial 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**

What is the full space of alternatives?
How to explore it?

RDF query optimization for MapReduce

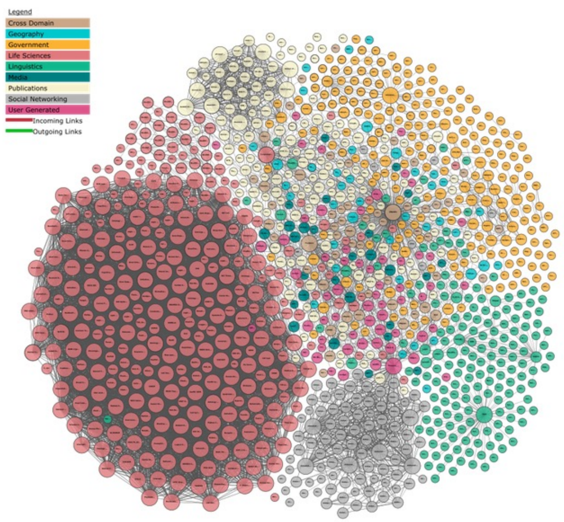
How can we manage large volumes of Linked Open Data (RDF) based on MapReduce?

Data management task specified in dedicated language

Language parser

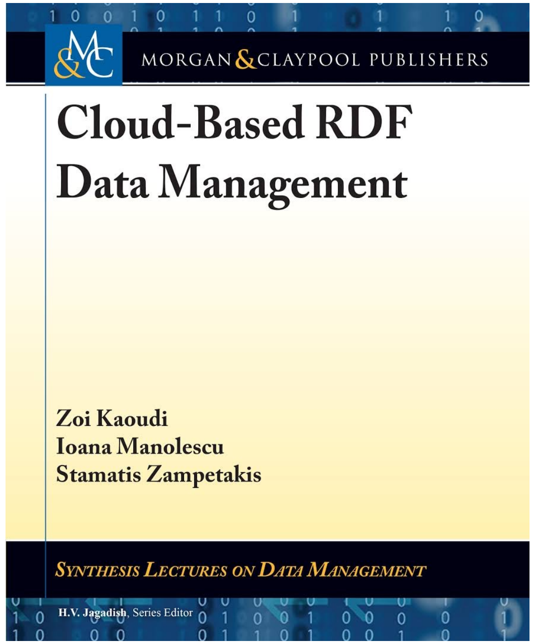
Possibly algebraic optimizer

MapReduce compiler and optimizer



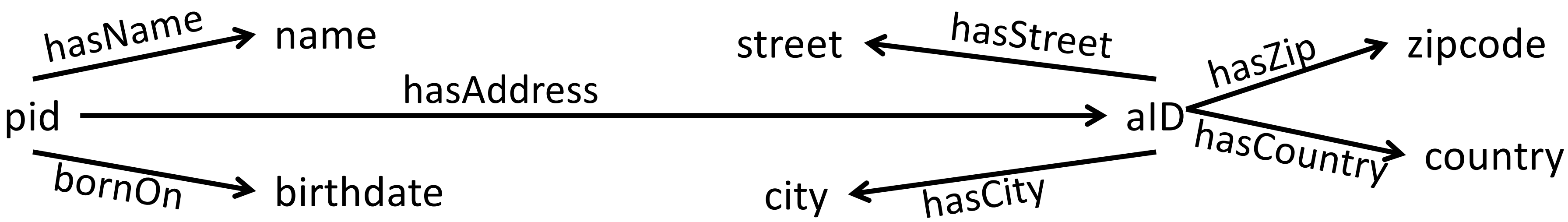
Hadoop MapReduce

Results



RDF query optimization for MapReduce

- Standard query language for RDF: SPARQL
- 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

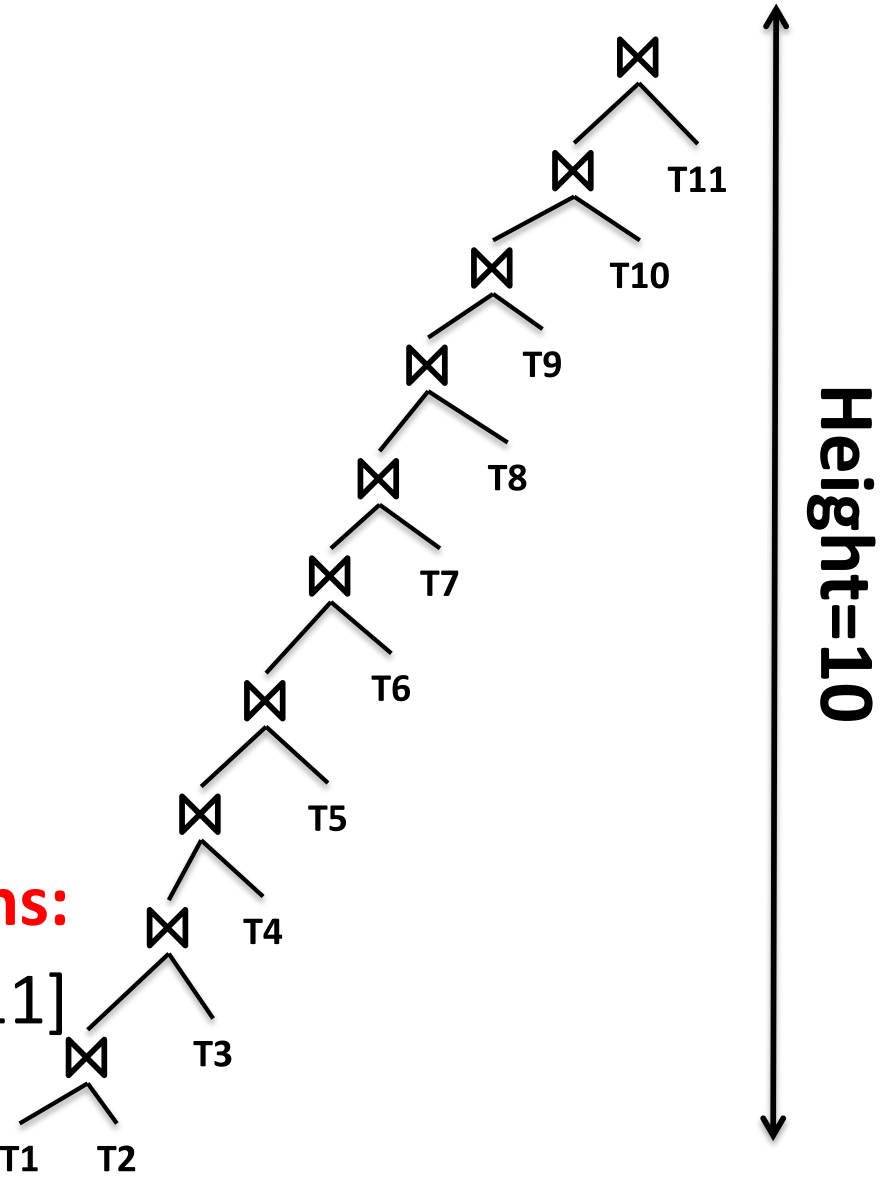
Query plans on MapReduce

Query:

SELECT ?x ?y
WHERE {
T1: ?w :prop1 <C1> .
T2: ?w :prop2 <C2> .
T3: ?w :prop3 ?x .
T4: ?x :prop4 <C3> .
T5: ?x :prop5 <C4> .
T6: ?x :prop6 ?z .
T7: ?z :prop7 ?f .
T8: ?f :prop8 ?y .
T9: ?f :prop9 ?h .
T10: <C5> :prop10 ?h .
T11: ?y :prop11 <C6> .}

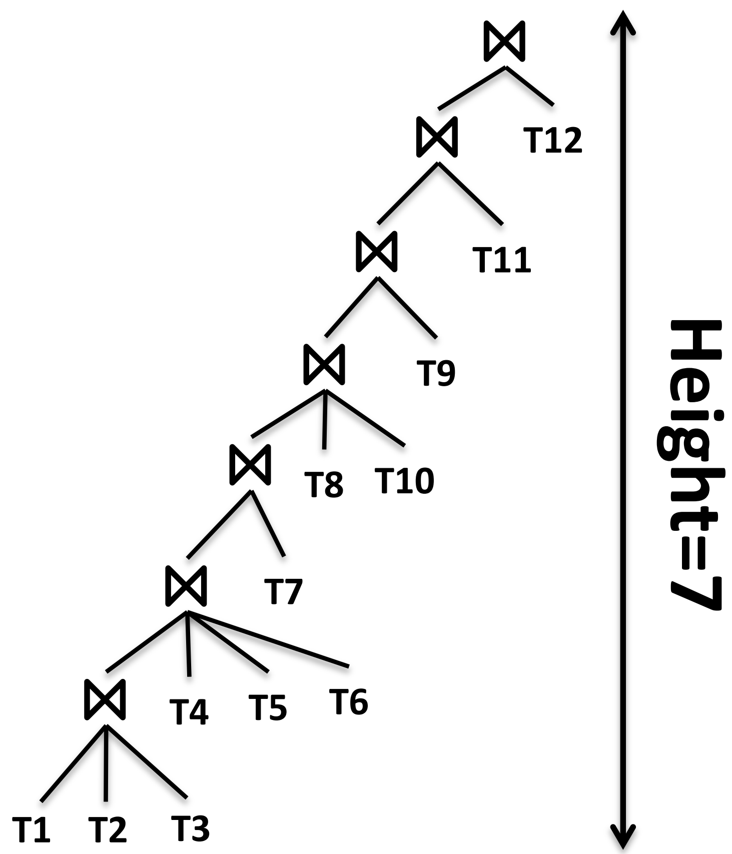
Left deep plans with binary joins:

[Olston08][Rohloff10][Schatzle11]



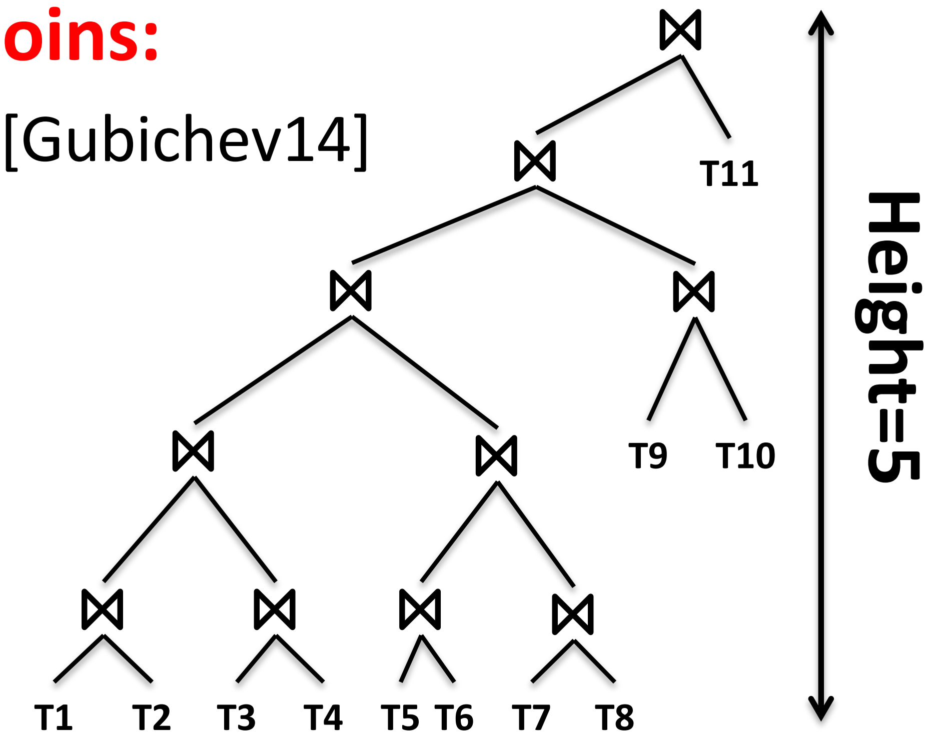
Query plans on MapReduce

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



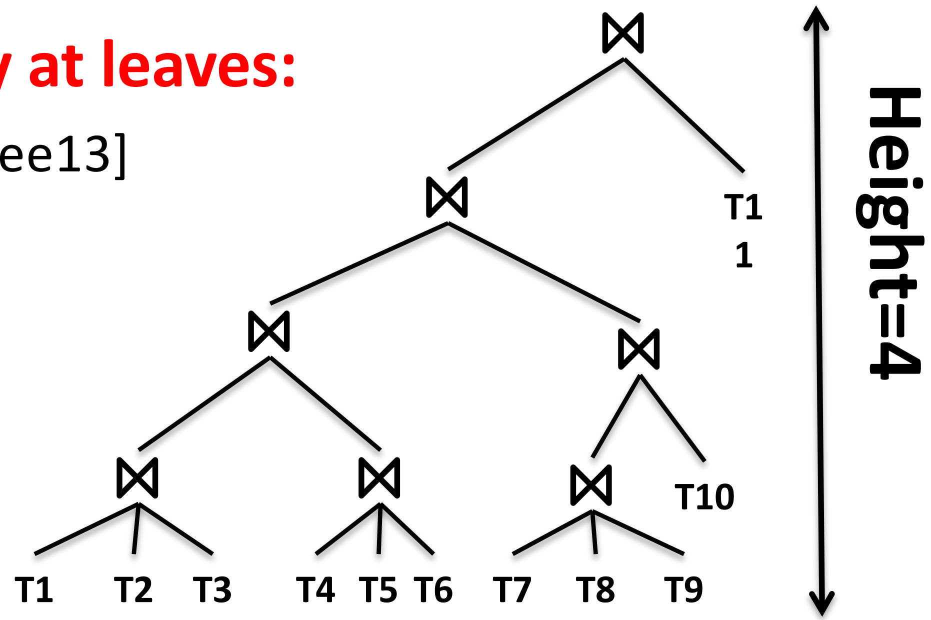
Query plans on MapReduce

- 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]



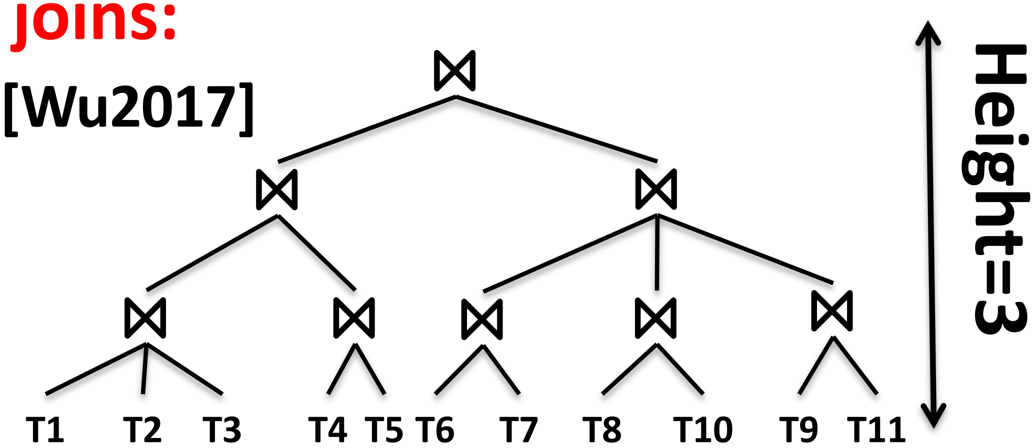
Query plans on MapReduce

- 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 leaves:** [Wu11][Kim11][Huang11][Ravindra11][Lee13]

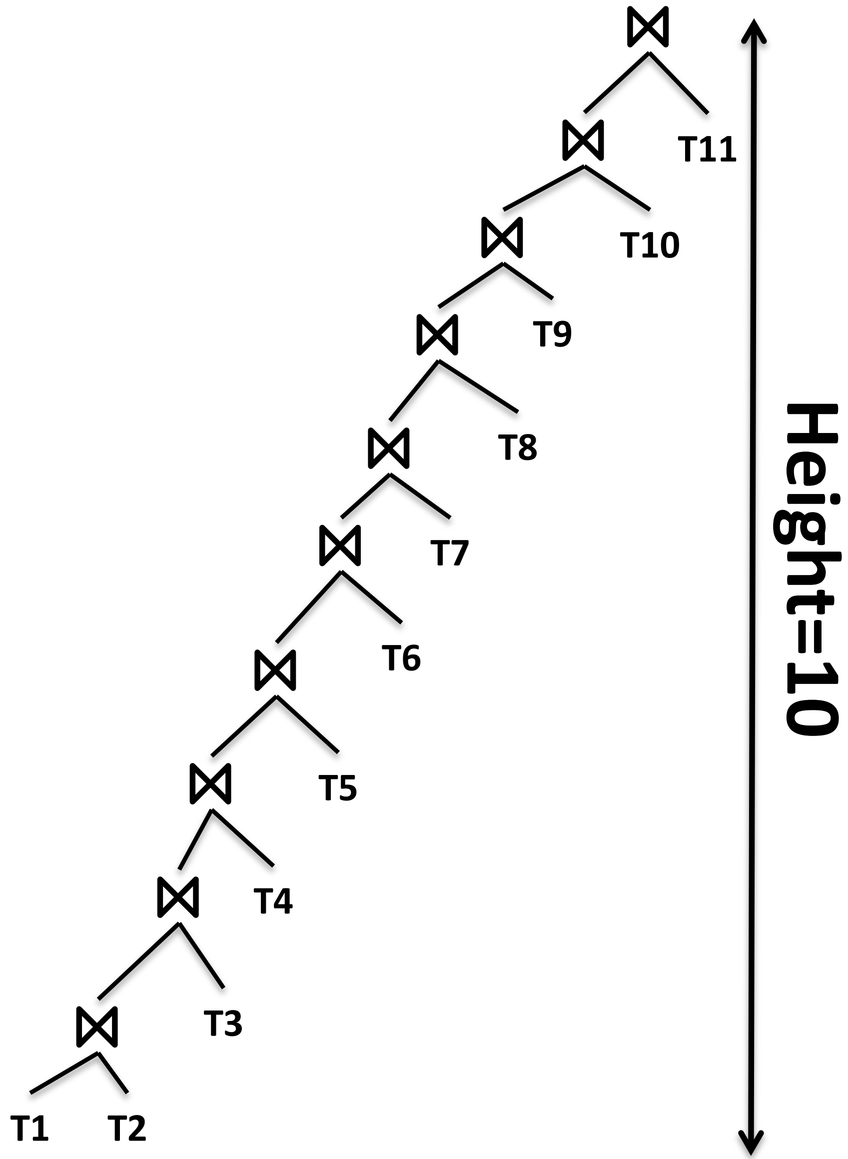


Query plans on MapReduce

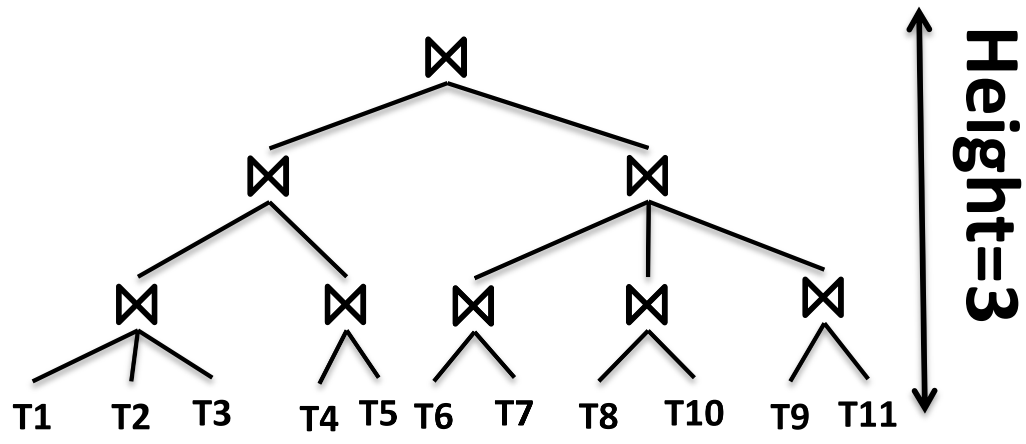
- Left deep plans with binary joins [Olston08][Rohloff10][Schatzle11]
- Left deep plans with n-ary joins [Papailiou13]
- Bushy plans with binary joins [Neumann10][Tsialiamanis12][Gubichev14]
- Bushy plans with n-ary joins only at leaves [Wu11][Kim11][Huang11][Ravindra11][Lee13]
- **Bushy plans with n-ary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



Query plans on 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!



Query plans in CliqueSquare [Goasdoué2015]

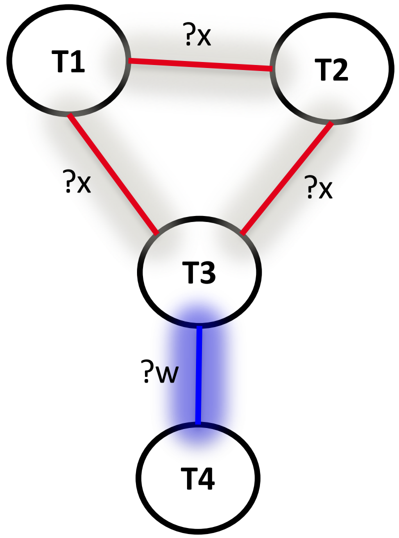
- Goal: build **flat** plans for RDF queries by exploiting **n-ary (star)** equality joins.

- Idea: identify **cliques** = subsets of *n >= 2* triples sharing a common variable, use an *n*-ary join to combine them
  - Then find another clique and similarly join them, etc.
  - …
  - Until all triples have been joined

CliqueSquare algorithm: Variable Graphs

Represent queries and intermediary results

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

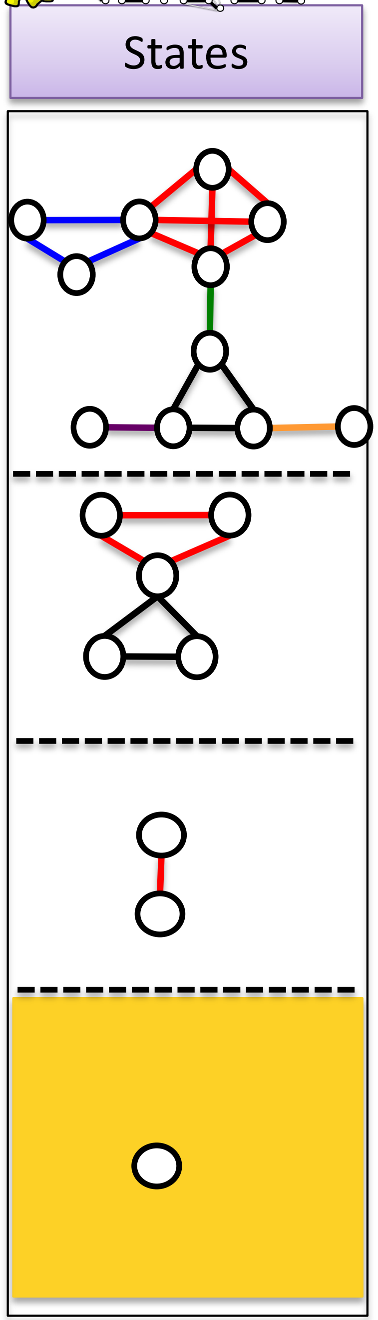


Query

Variable graph

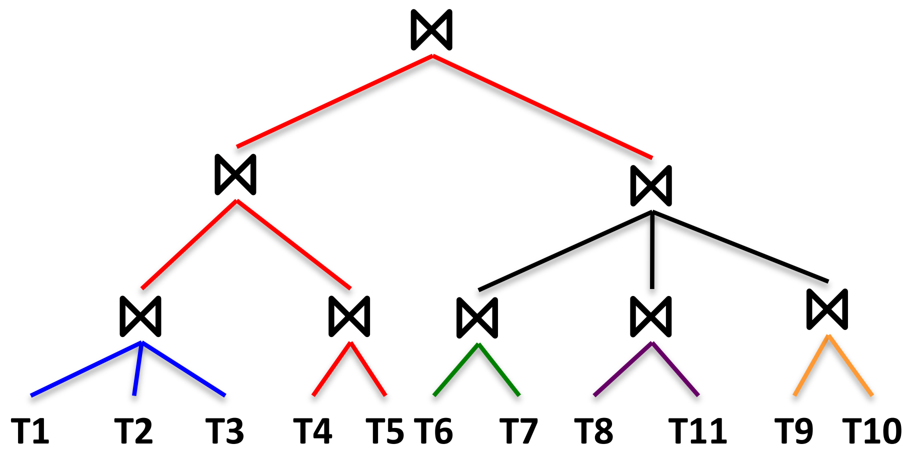
Nodes are connected with an **edge** if they share a **variable**

CliqueSquare: optimization with *n*-ary joins

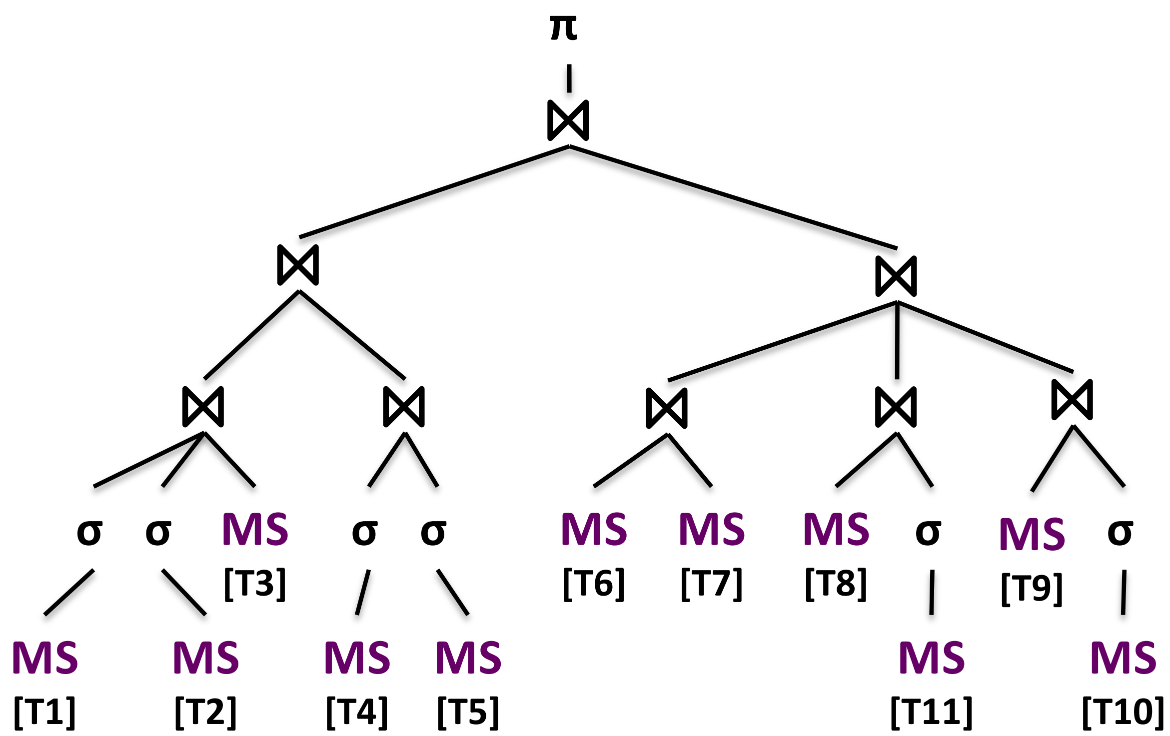


Each **node** of a graph corresponds to a **clique** of nodes of the previous graph.

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

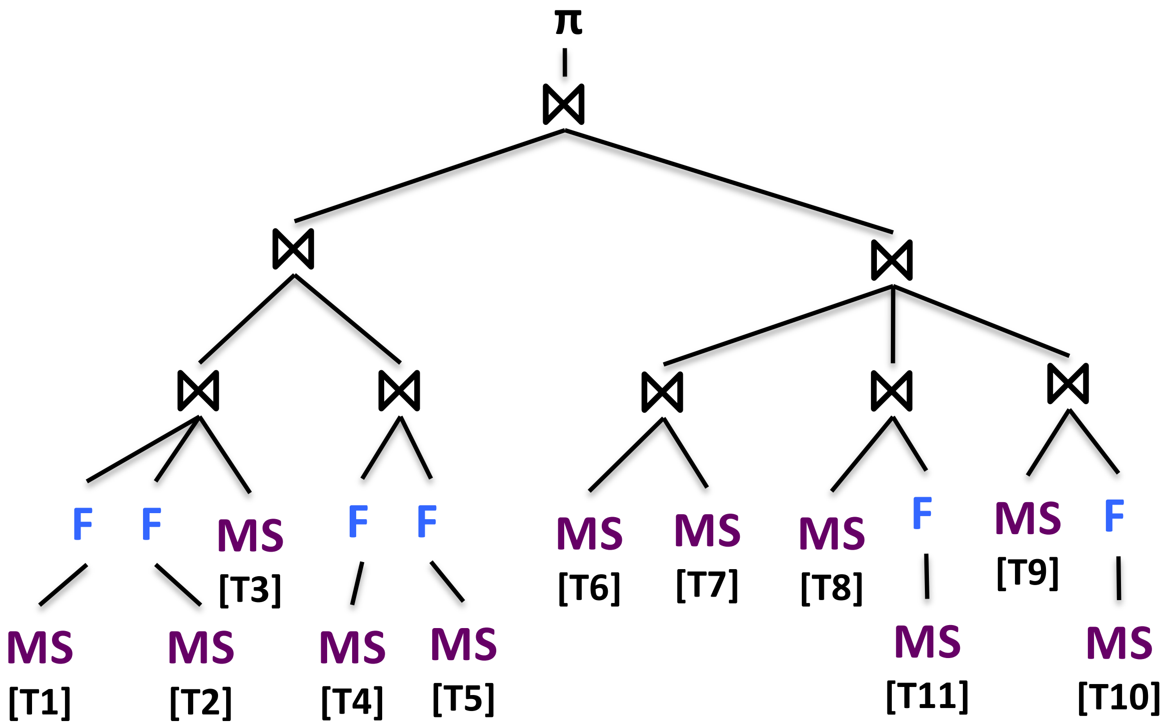


Logical plan → Physical plan



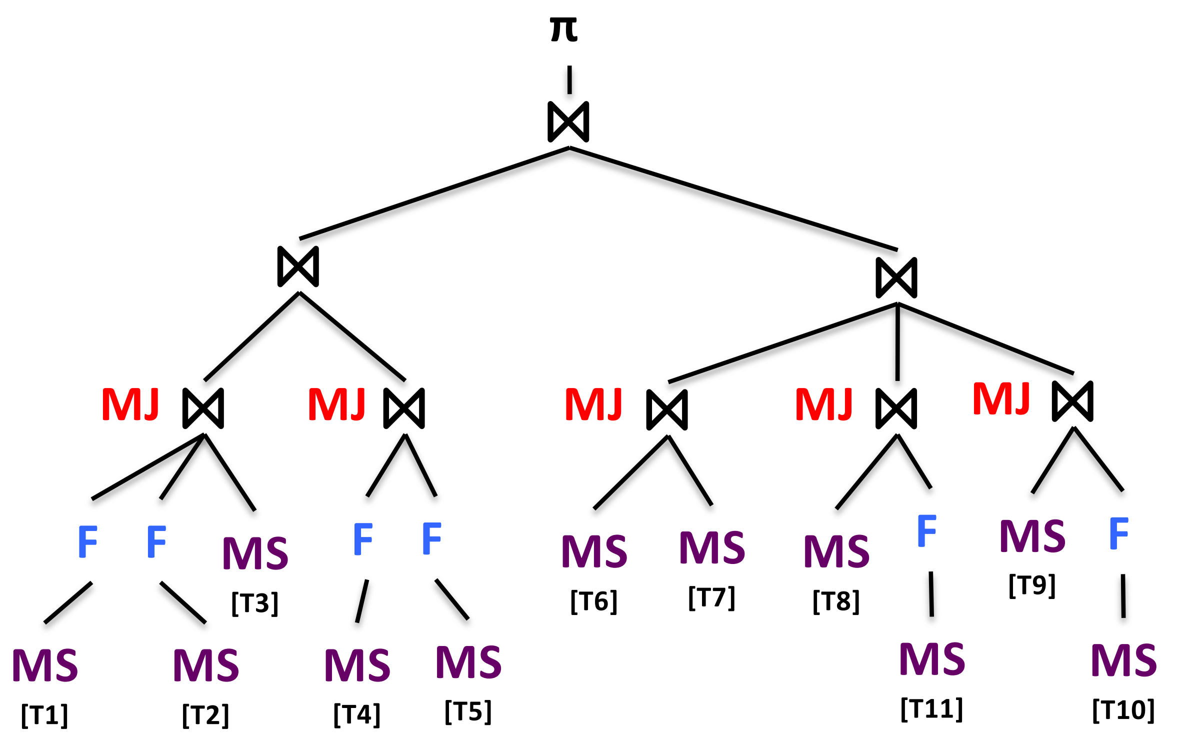
- Reading the triples from HDFS requires a Map Scan (**MS**) operator

Logical plan → Physical plan



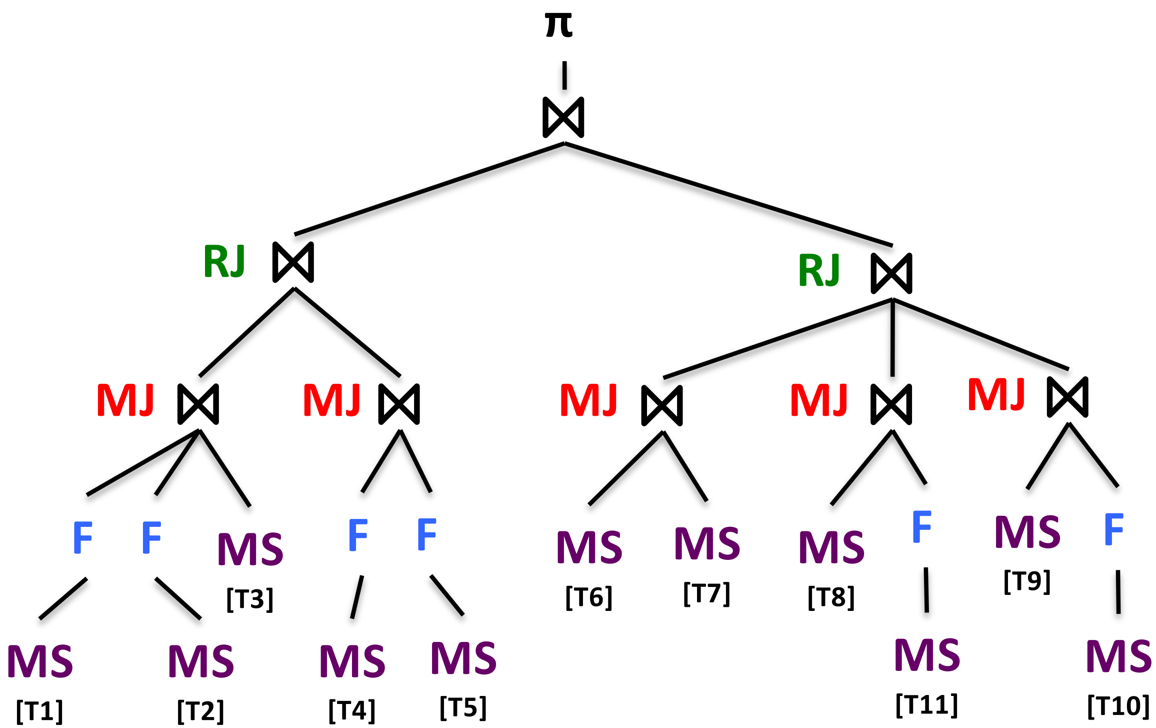
- Logical selections (σ) are translated to physical selections (**F**)

Logical plan → Physical plan



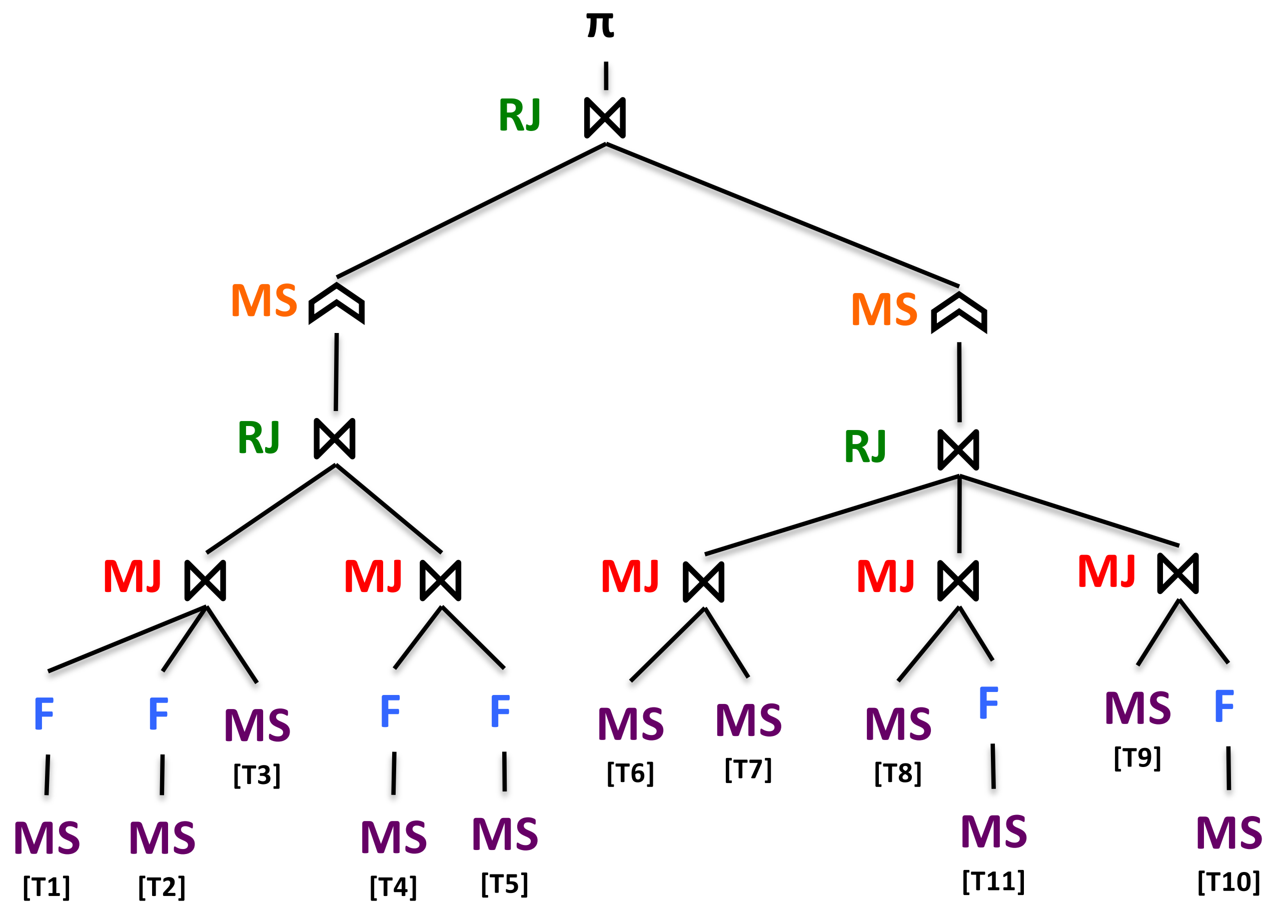
➢ First level joins are translated to Map side joins (MJ) taking advantage of the **data partitioning** (triples stored three times, hashed by subject, property, object)

Logical plan → Physical plan



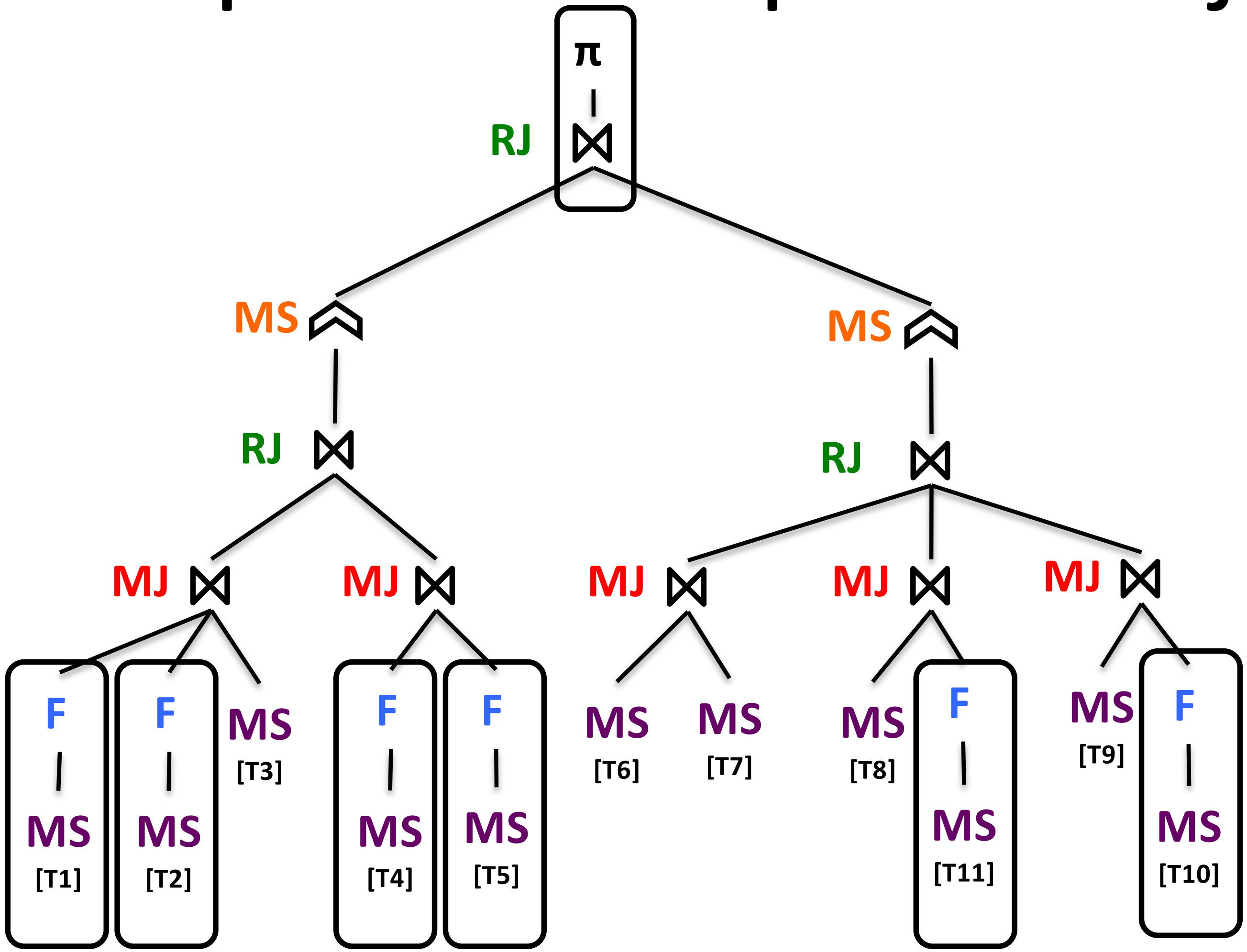
- All subsequent joins are translated to Reduce side joins (**RJ**)

Physical plan → MapReduce jobs



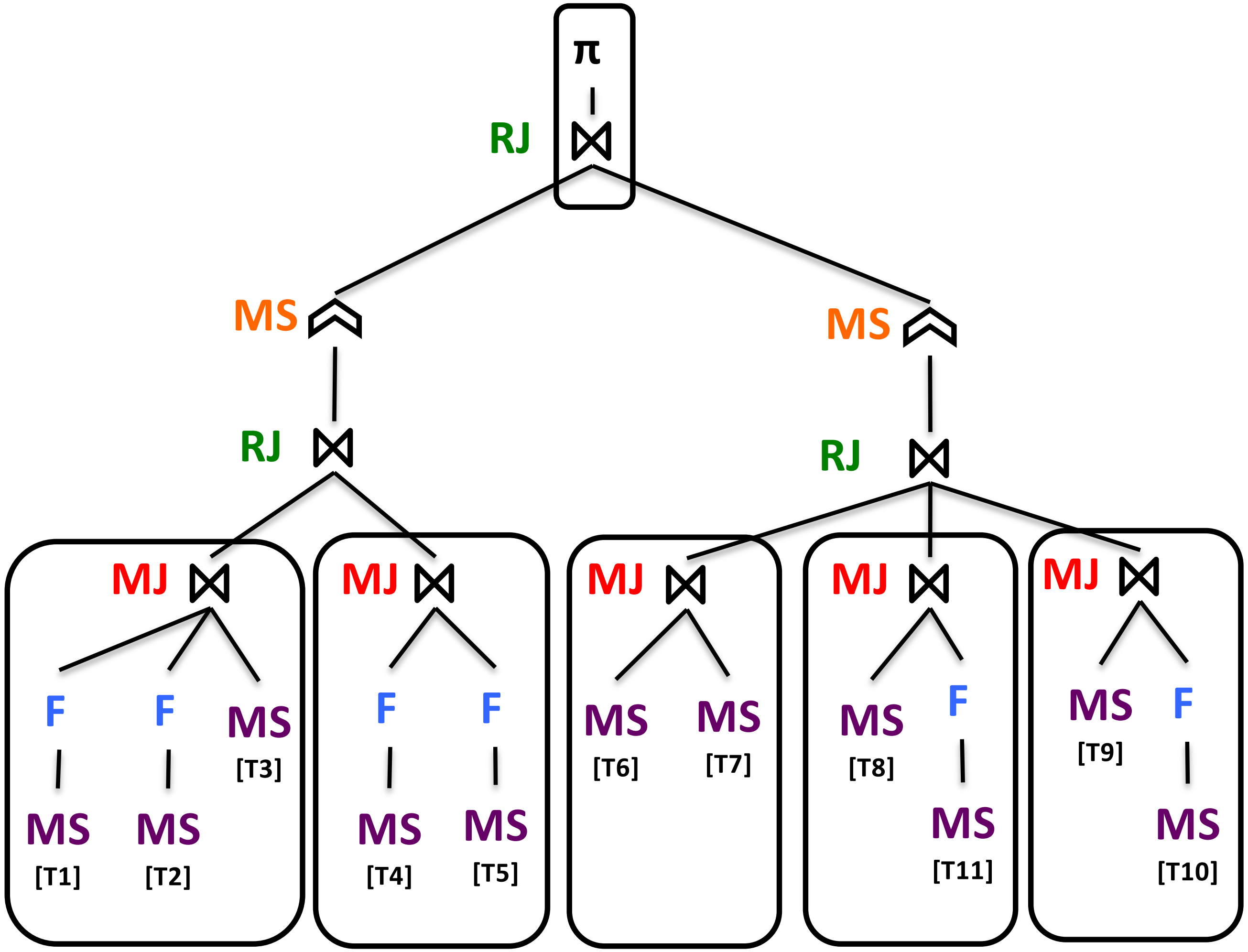
- Group the physical operators into Map/Reduce **tasks** and **jobs**

Physical plan → MapReduce jobs



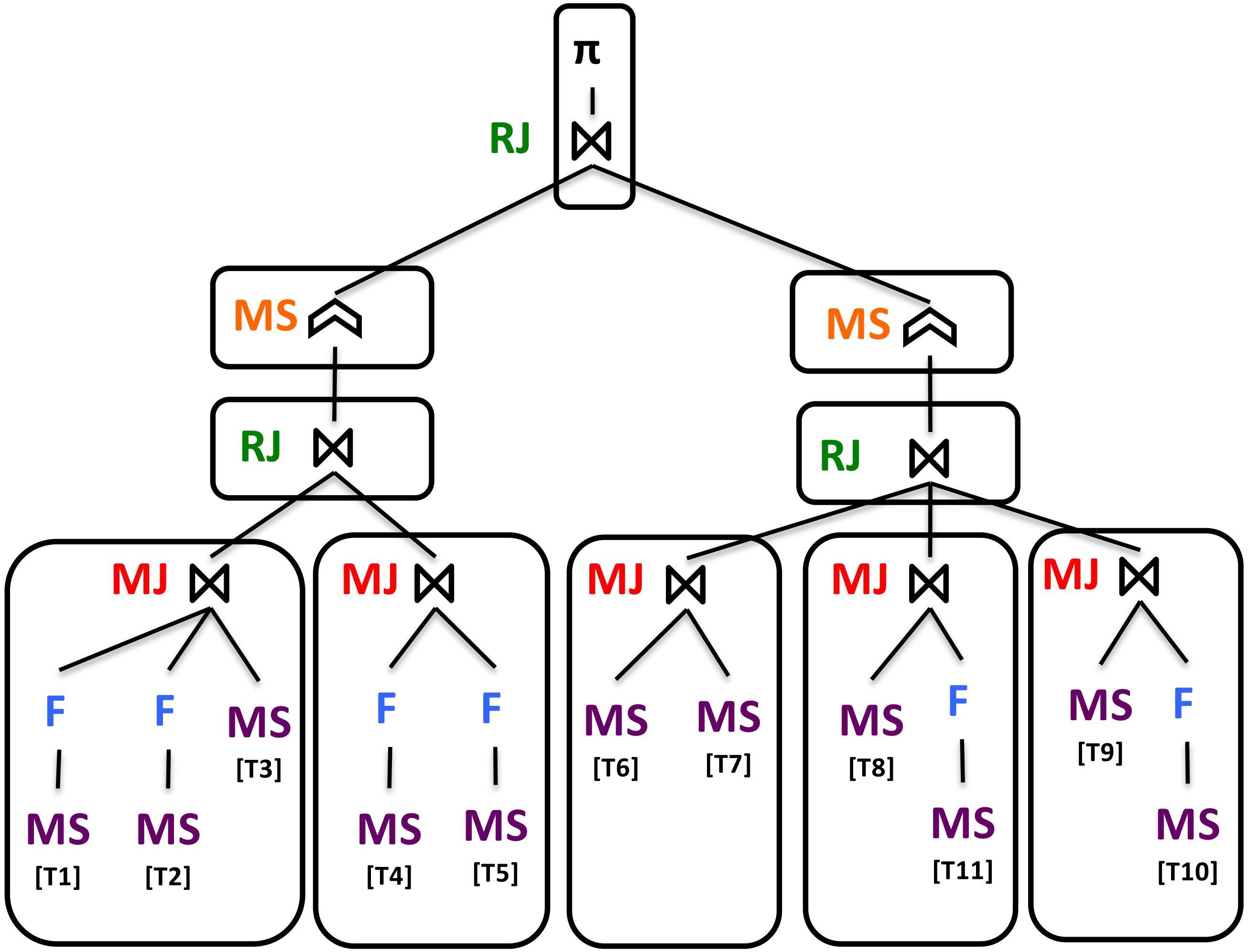
- Selections (F) and projections (π) belong to the same task as their child operator

Physical plan → MapReduce jobs



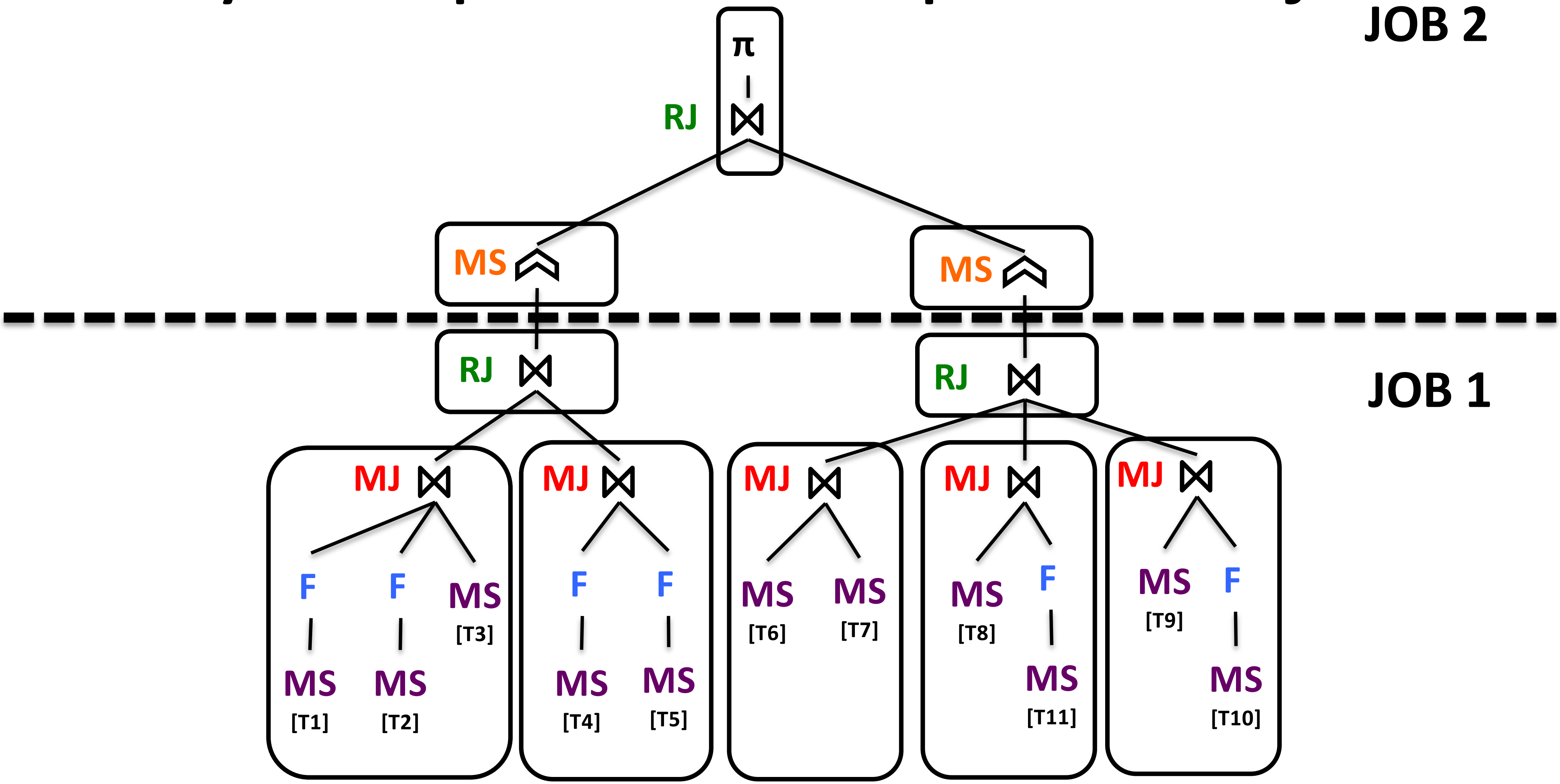
- Map joins (**MJ**) along with all their descendants are executed in the same task

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

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.)

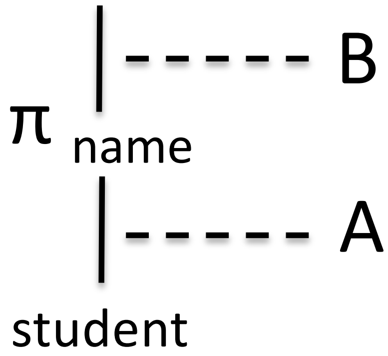
- Main strong points:
  - Fast access to individual rows
  - read/write consistency
  - Selection push-down (~ Hadoop++)
- Does not have: column types, query language, …

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;
```

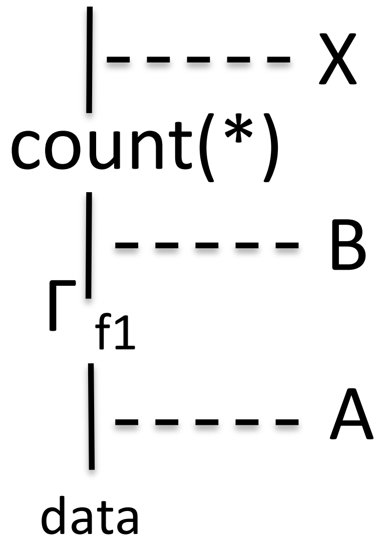


- Flexible data model (~ nested relations)
- Some nesting in the language (< 2 FOREACH ☺)

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

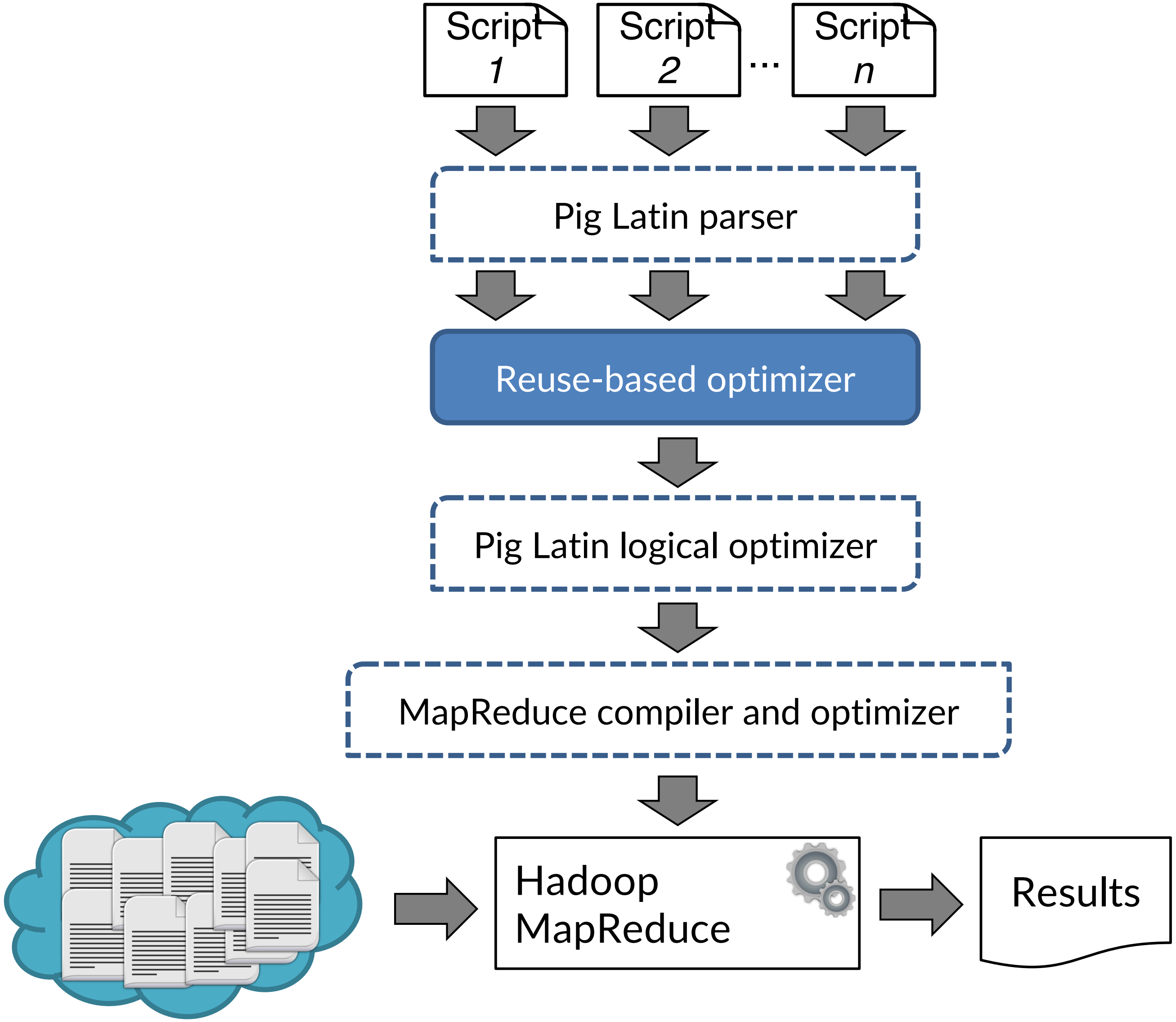
```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)}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

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