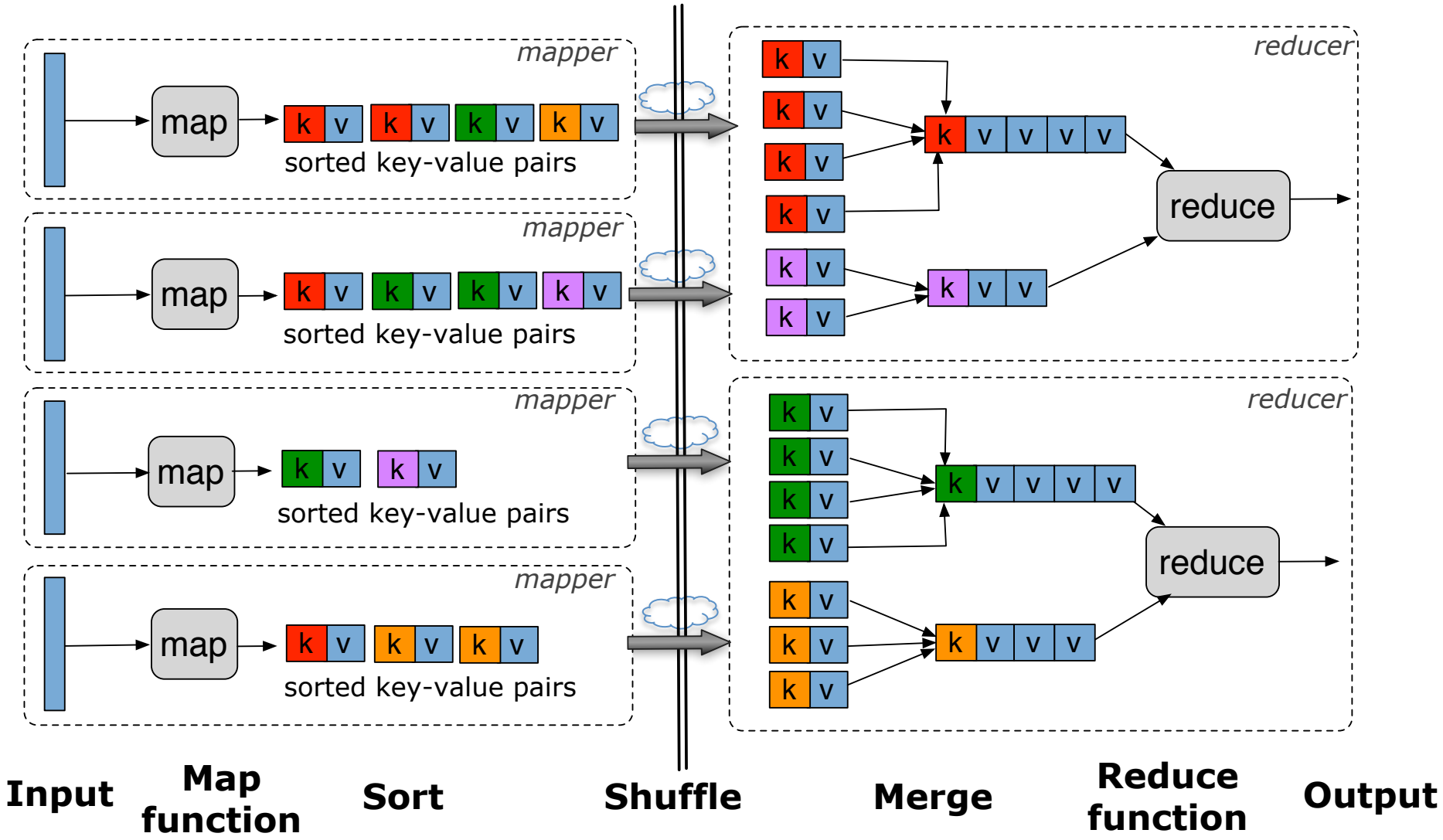
# Distributed Big Data Management Systems (Part 2)

**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

# STRUCTURED DATA MANAGEMENT ON TOP OF MAP-REDUCE

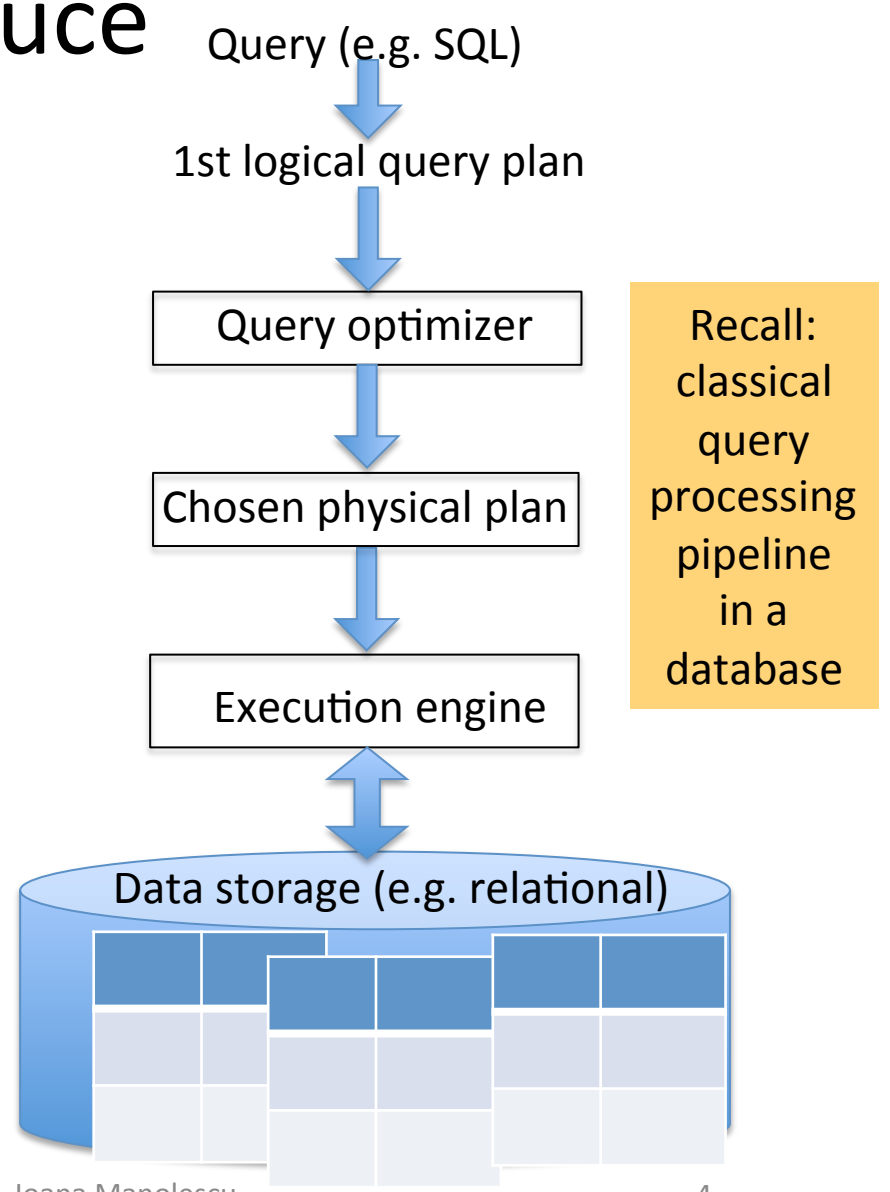
# Recall: Map/Reduce outline



# 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

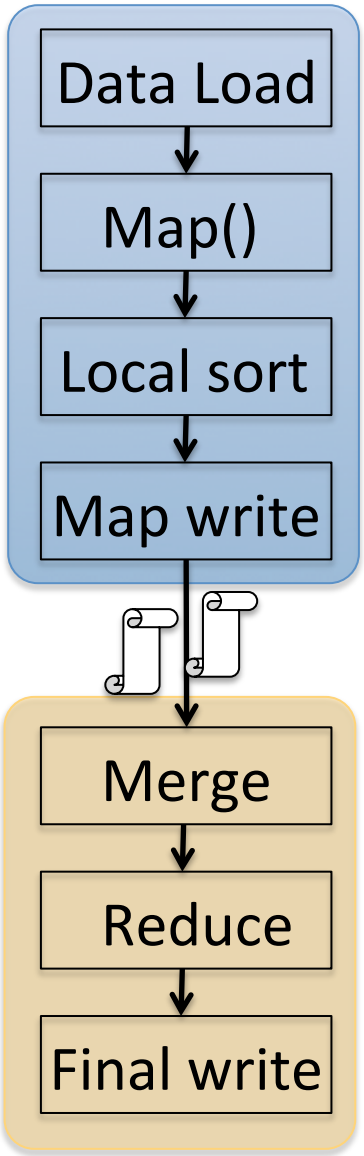



Recall: classical query processing pipeline in a database

# IMPROVING DATA ACCESS PERFORMANCE IN A DISTRIBUTED FILE SYSTEM

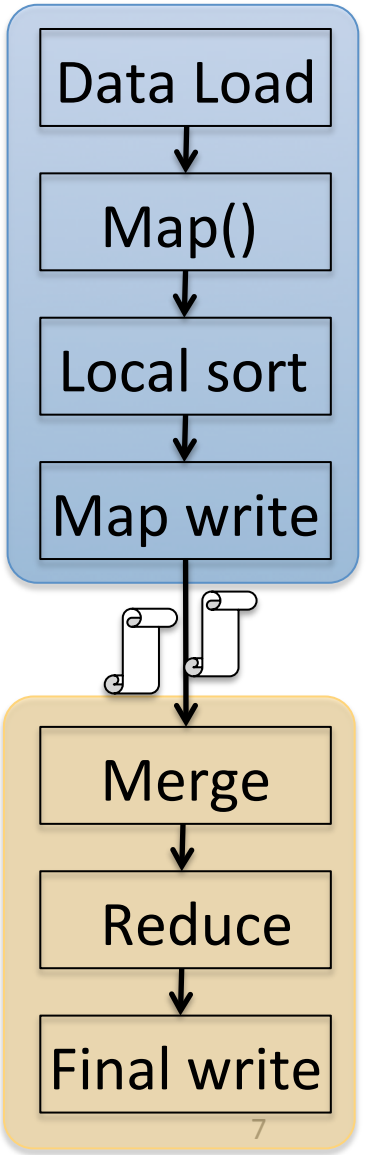
# Data access in Hadoop

- 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



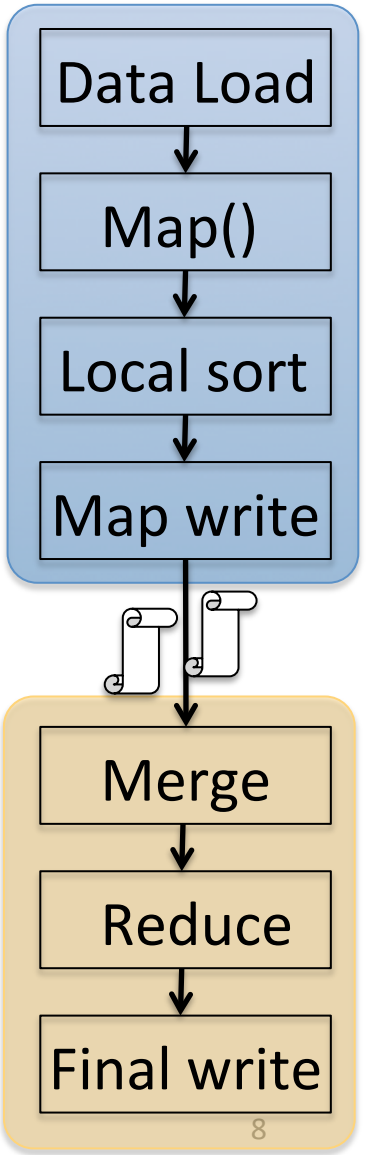
# Accelerating data access in Hadoop

- 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

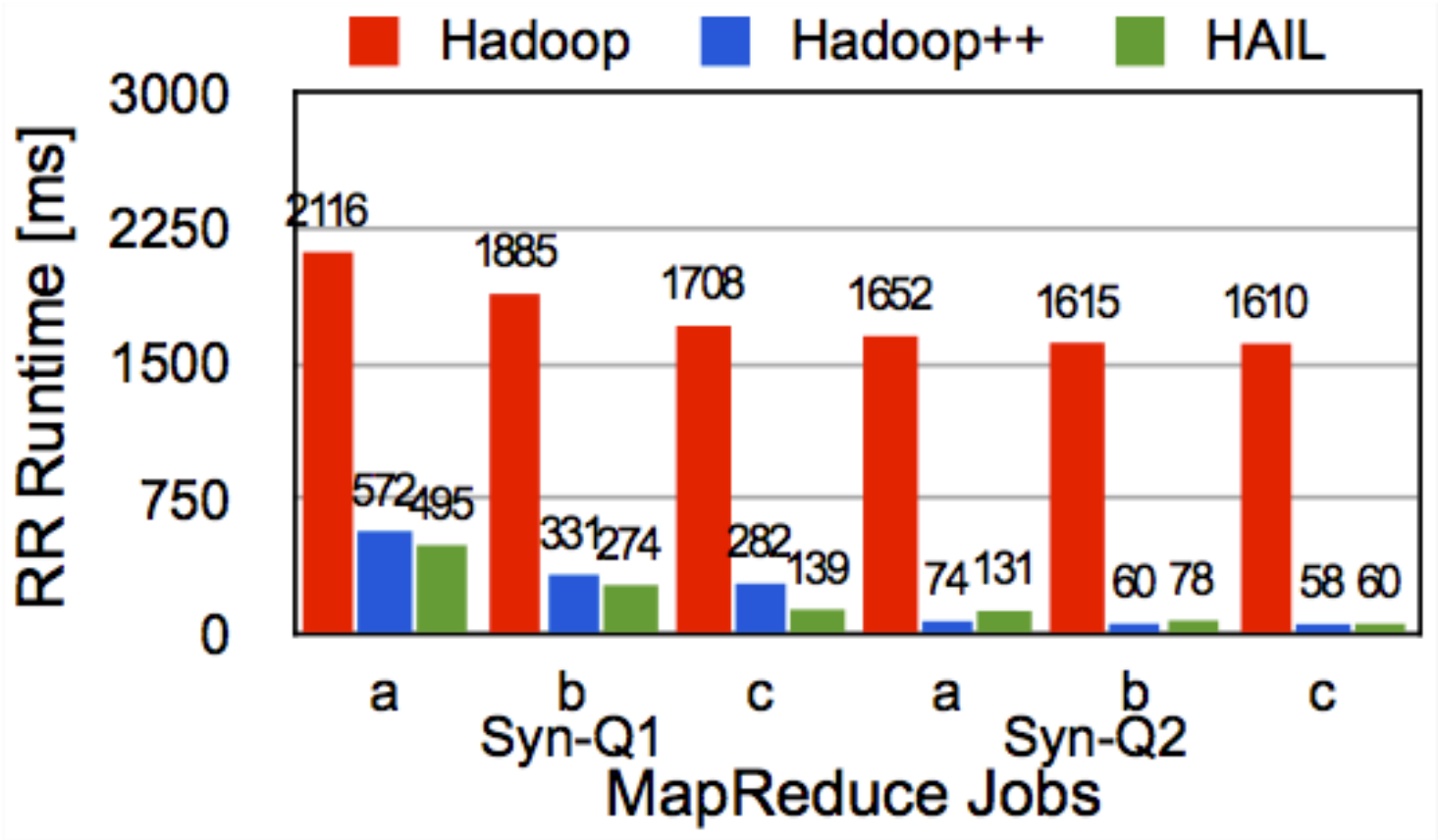


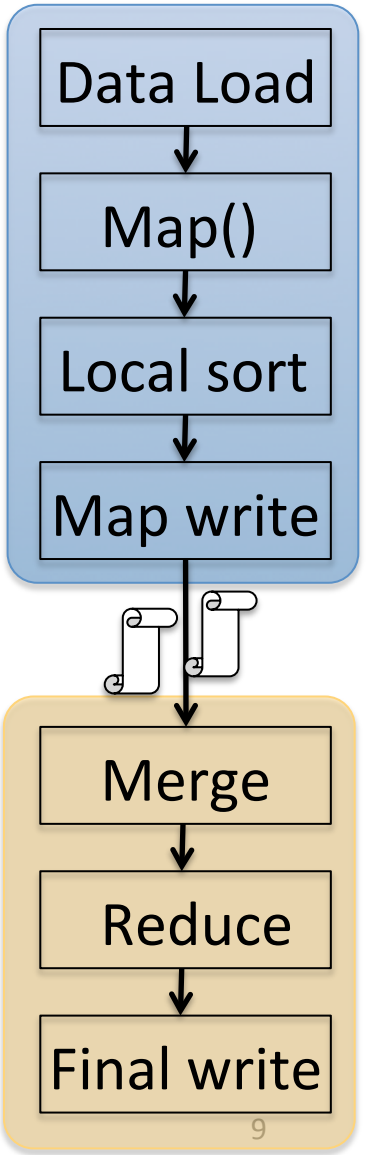
# Accelerating data access in Hadoop

- 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



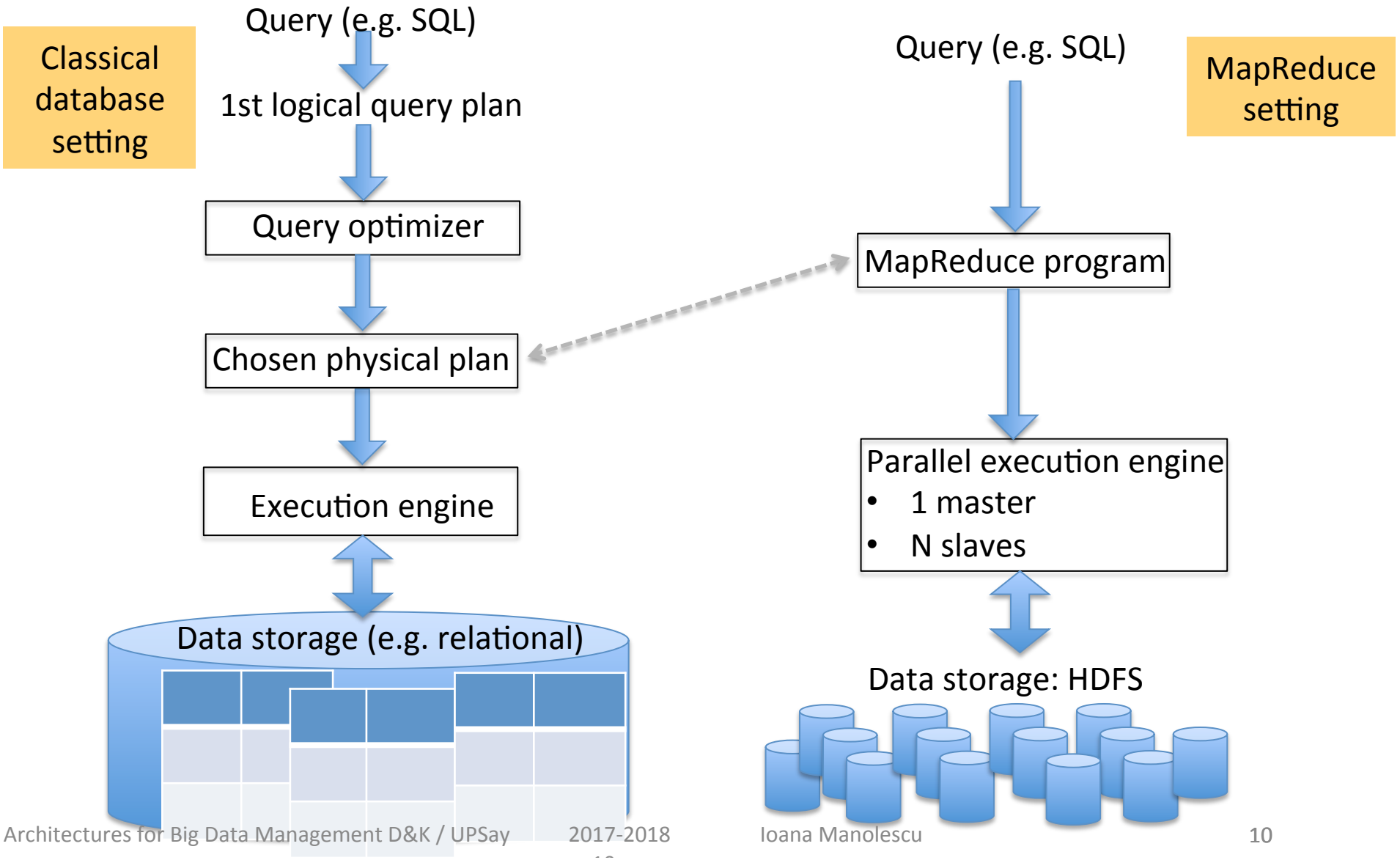
# Hadoop, Hadoop++ and HAIL



# Data management based on MapReduce

**First idea**: translate each query into a program



# Implementing queries through MapReduce

- SELECT MONTH(c.start_date), COUNT(*)
  FROM customer c
  GROUP BY MONTH(c.start_date)

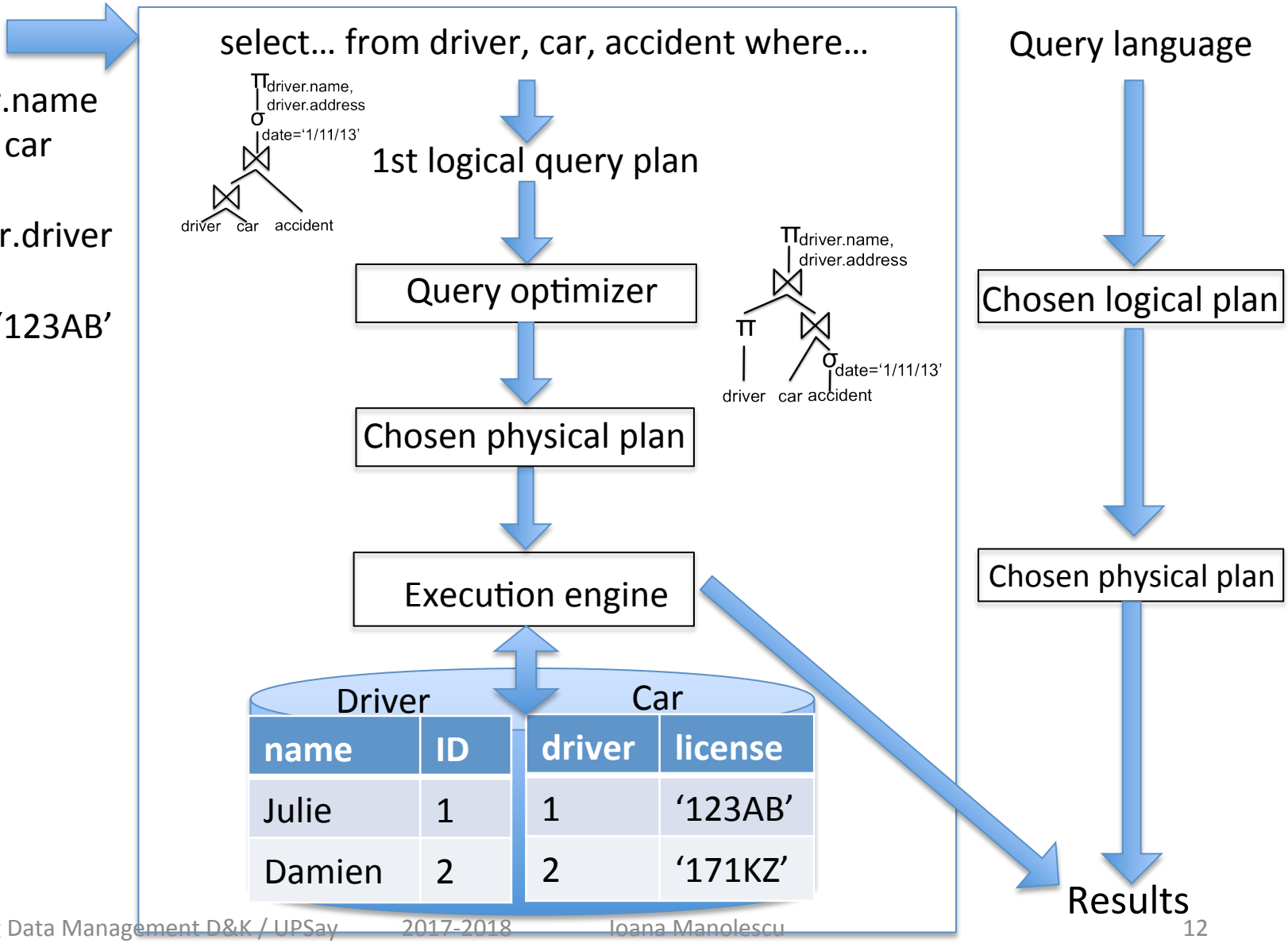
- SELECT c.name, o.total
  FROM customer c, order o
  WHERE c.id=o.cid

- 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

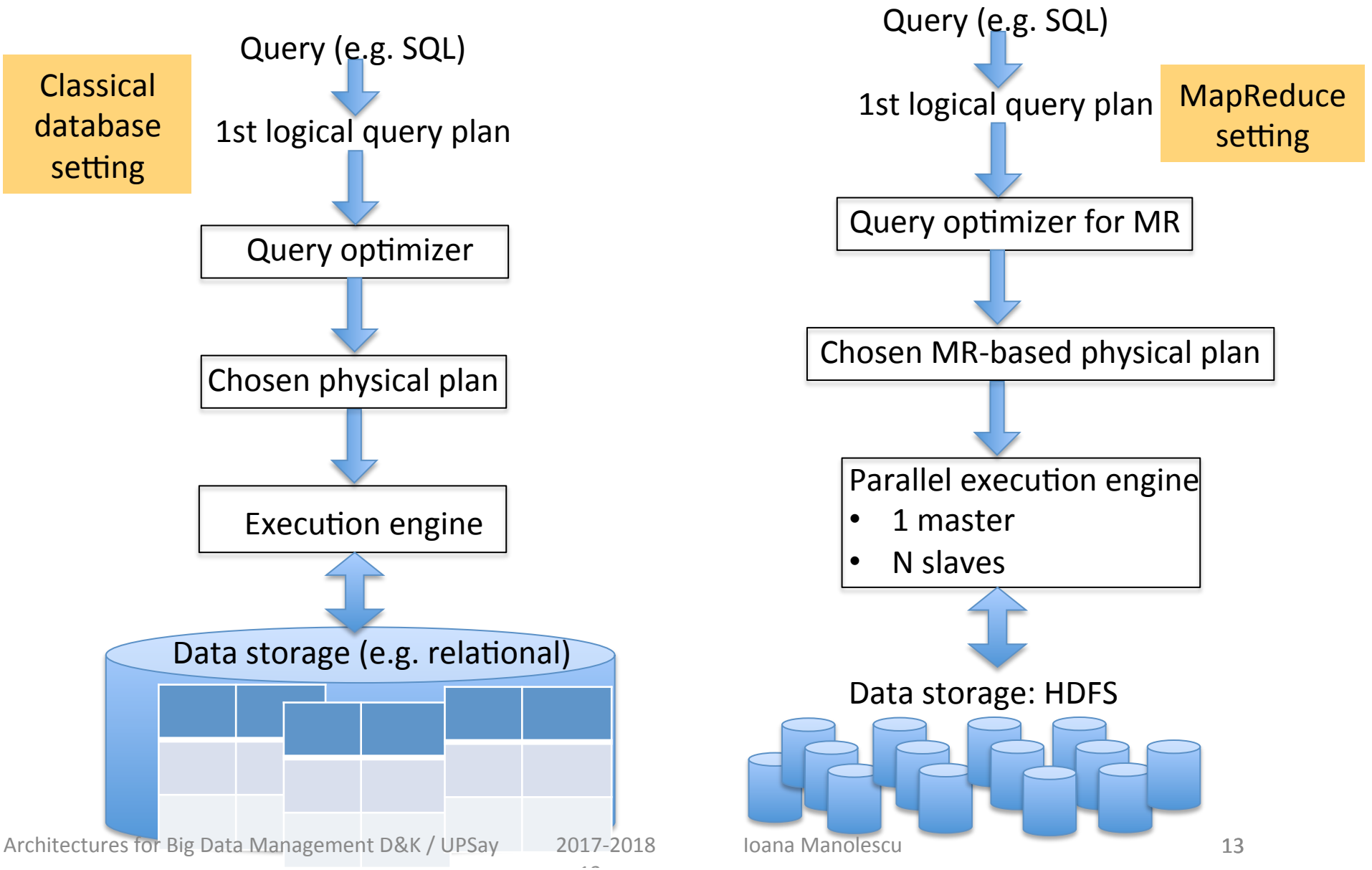
SQL

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



# Data management based on MapReduce

**Second idea**: translate every physical operator into a 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 ( $\sigma$ and $\pi$ )
  - Binary operators (mostly: $\bowtie$ on equality, i.e. equijoin)
  - N-ary operators (complex join expressions)

# Implementing unary operators on MapReduce

- Selection ($\sigma_{pred}$ ( 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 ($\pi_{cols}(R)$):
  - Split R tuples across all nodes
  - **Map**:
    foreach t
      output (hn(t), $\pi_{cols}(t)$)
  - **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|)

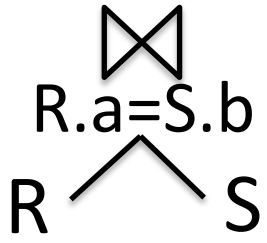
```
repeat{
 while (!aligned) { advance R or S };
 while (aligned) { copy R into topR, S into topS };
 output topR x topS;
} until (endOf(R) or endOf(S));
```

**Hash join**: // builds a hash table in memory O(|R|+|S|)

```
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);
                  }
```

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**:

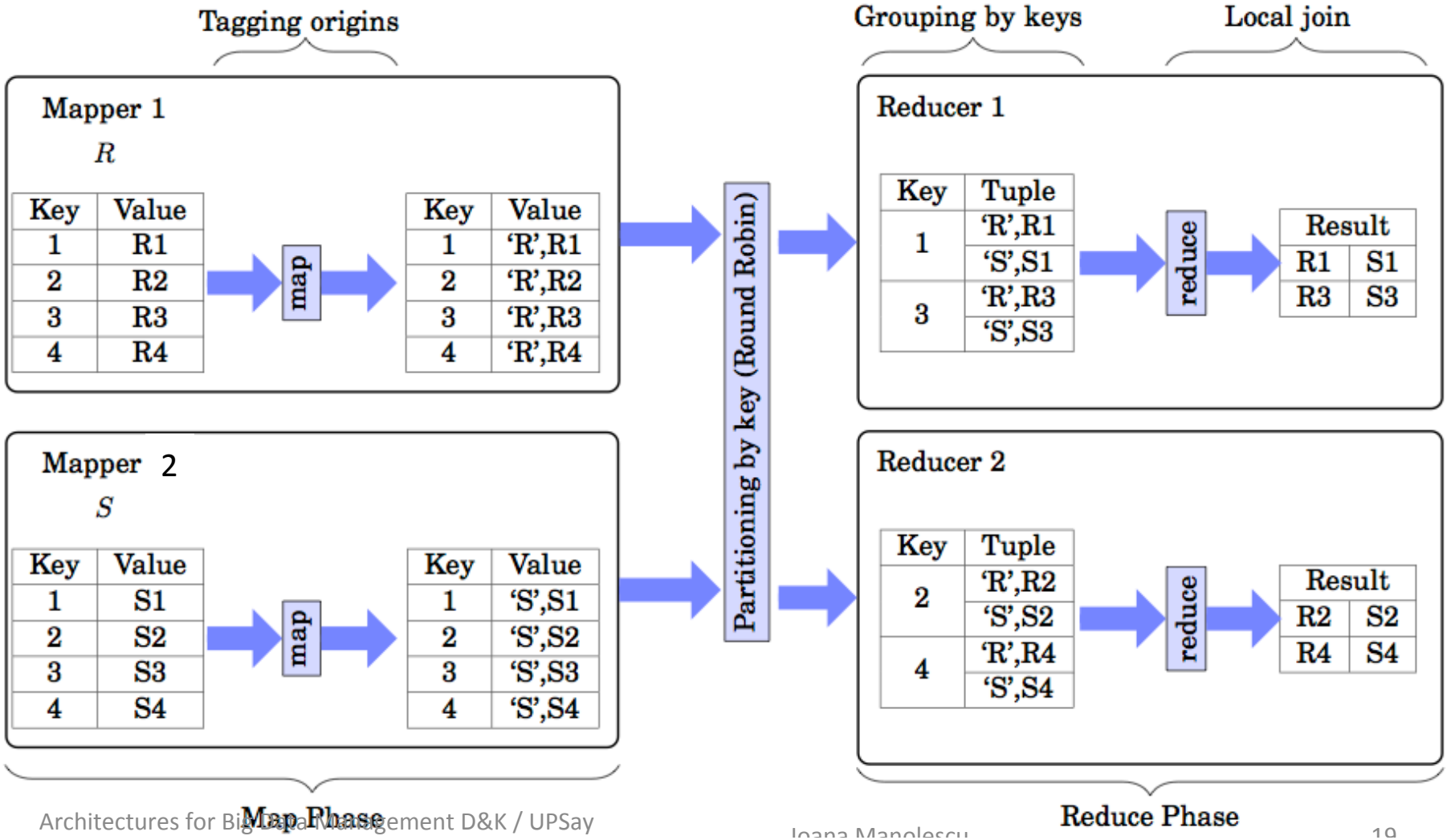
- Output (t.a, («R», t)) for each t in R
- Output (t.b, («S», t)) for each t in S

**Reducer**:

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

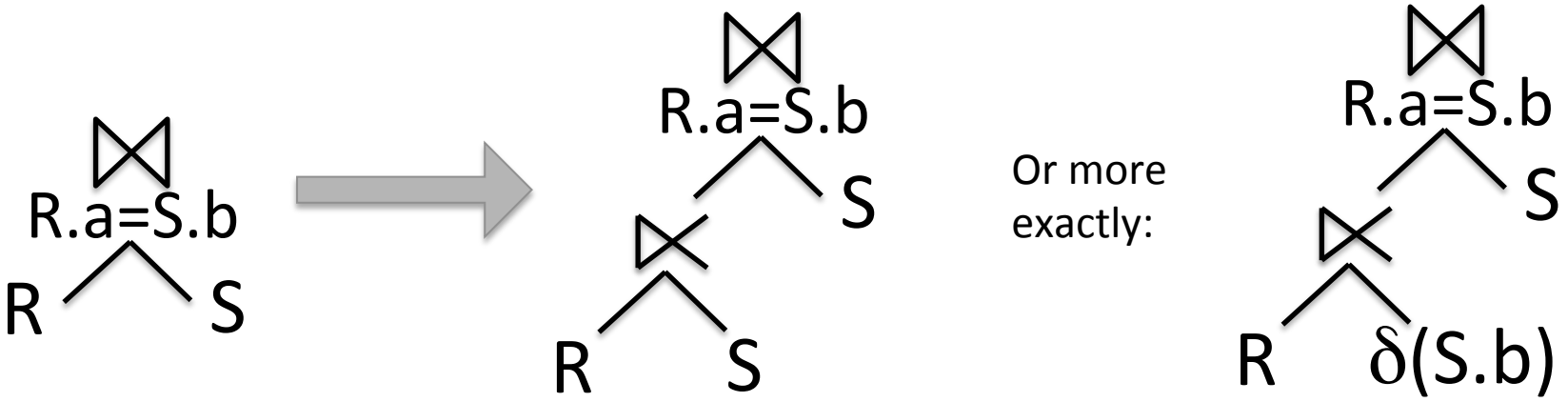
# Implementing equi-joins on MapReduce (1) Repartition join

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



# Implementing equi-joins on MapReduce (2)

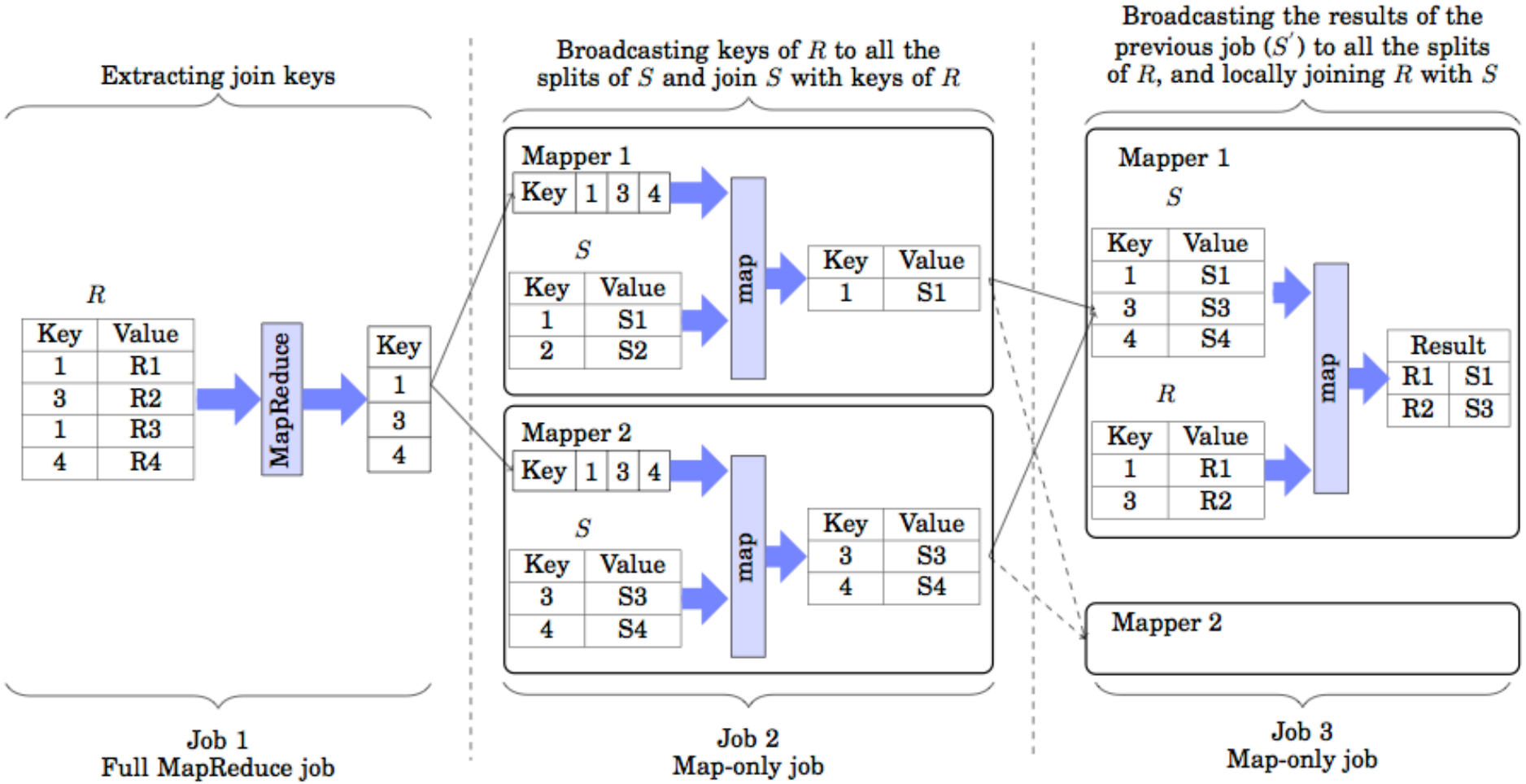
- **Semijoin-based MapReduce join**
- Based on the classical semijoin optimization technique:
  - R join S = (R semijoin S) join S



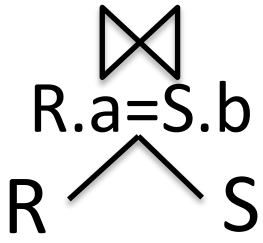

  - Useful in distributed settings to reduce transfers: *if the distinct S.b values* are smaller than *the non-matching R tuples*
  - Symetrical alternative: R join S = R join (S semijoin R)

# 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!

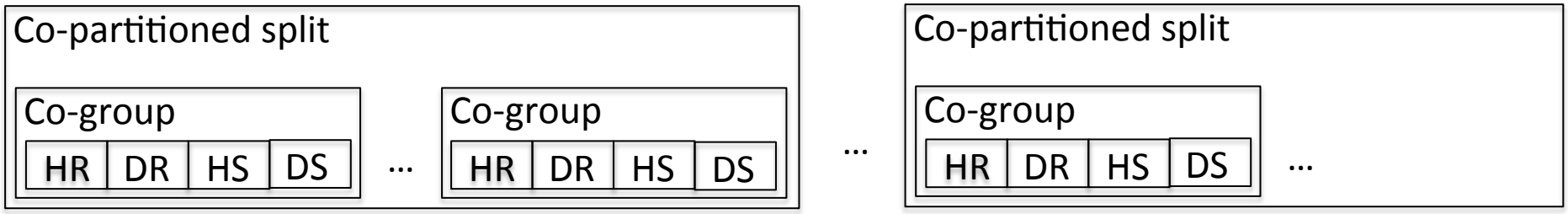
- Example: S is a *log* data collection (e.g. log table)
- R is a *reference* table e.g. with user names, countries, age, …
- Facebook: 6 TB of new log data/day

**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

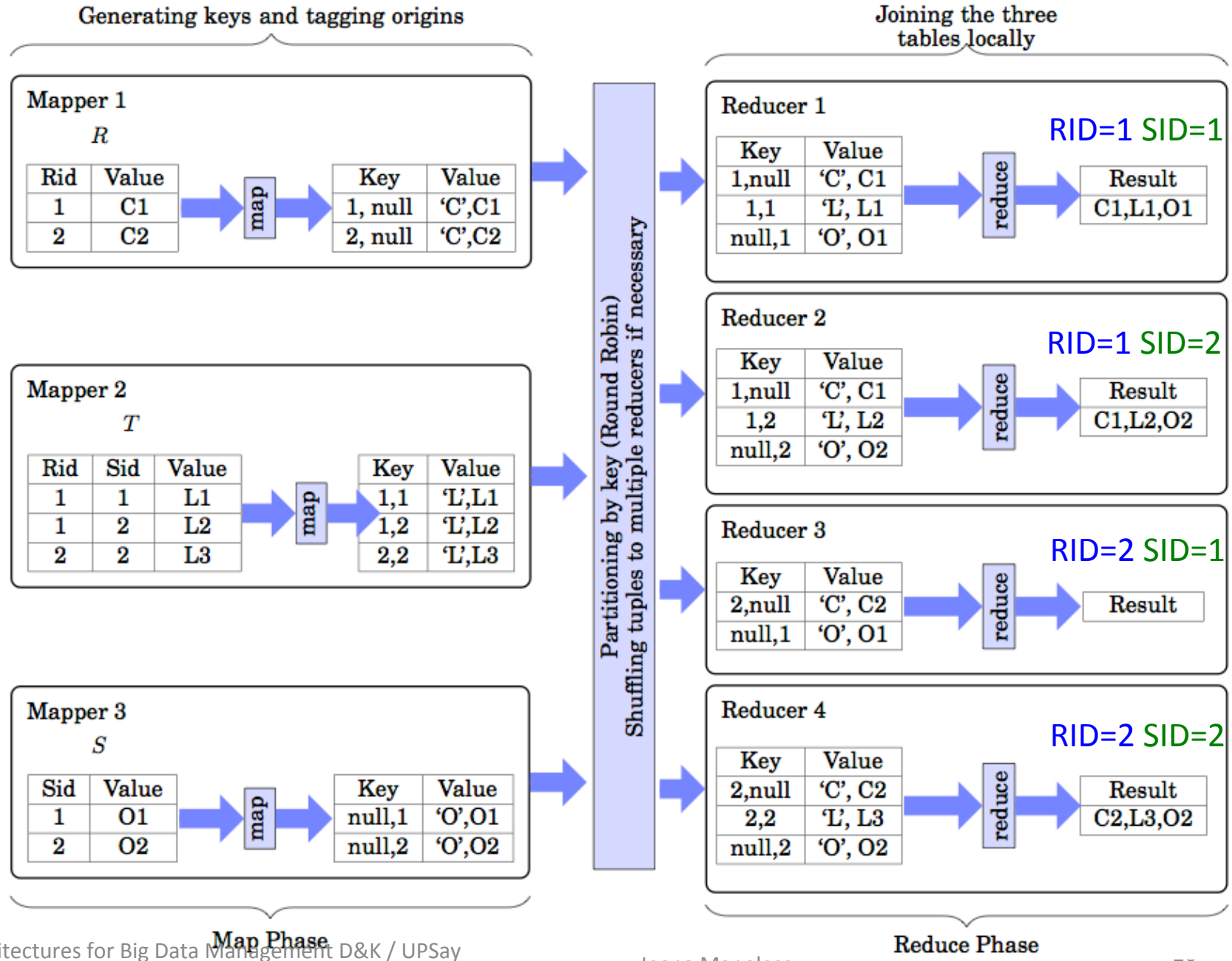
| Algorithm | + | - |
|---|---|---|
| Repartition Join | Most general | Not always the most efficient |
| Semijoin-based Join | Efficient when semijoin is selective (has small results) | Requires several jobs, one must first do the semi-join |
| Broadcast Join | Map-only | One table must be very small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

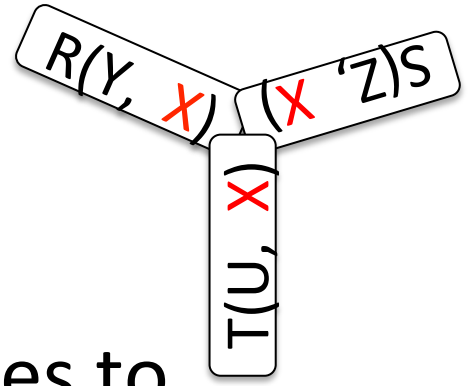
- Three relations, two join attributes (RID and SID)
- Split the SIDs into Ns groups and the RIDs in Nr groups. Assume Nr x Ns reducers available.
- Hash **T** tuples according to a composite key made of the two attributes. Each **T** tuple goes to one reducer.
- Hash **R** and **S** tuples on *partial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

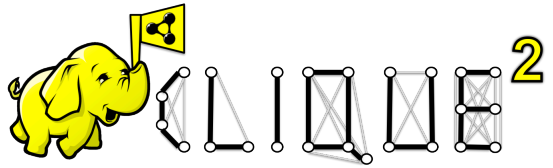
# Query optimization for MapReduce

- 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**: CliqueSquare (next)

# CLIQUESQUARE: OPTIMIZATION WITH N-ARY JOINS FOR A MAPREDUCE ENVIRONMENT

[Goasdoué, Kaoudi, Manolescu, Quiané, Zampetakis, IEEE ICDE 2015]

# CliqueSquare: flat plans for massively parallel RDF queries



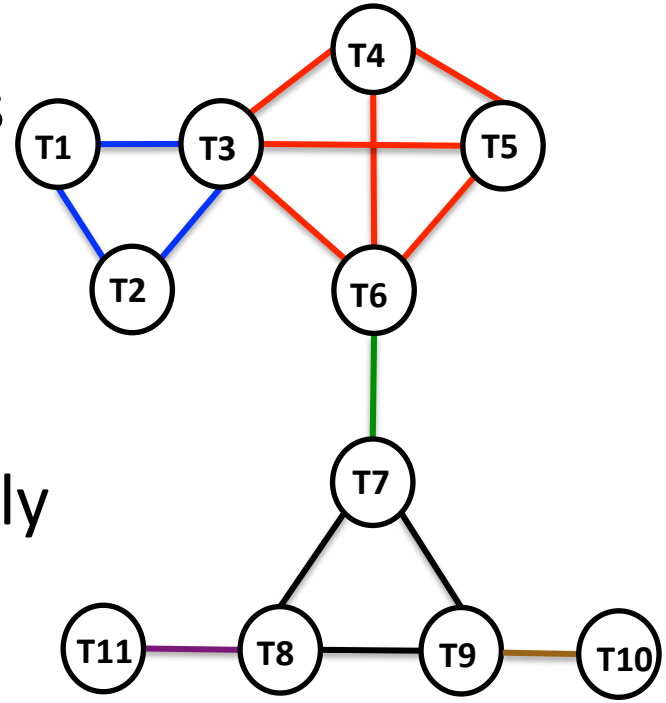

- **Focus:** Build massively parallel **flat** plans for RDF queries by exploiting **n-ary (star)** equality joins.
- **We will see:**
  - Query optimization algorithms relying on **n-ary** equality joins
  - **Formal guarantees** regarding the **flatness** (number of successive jobs) of the plans
  - Data placement to help processing
  - Translation of a plan in a MapReduce program

Publication, code at:

https://team.inria.fr/oak/projects/cliquesquare/

# Query optimization overview

- Left deep plans with binary joins
- Left deep plans with n-ary joins
- Bushy plans with binary joins
- Bushy plans with n-ary joins only at leaves
- Bushy plans with n-ary joins

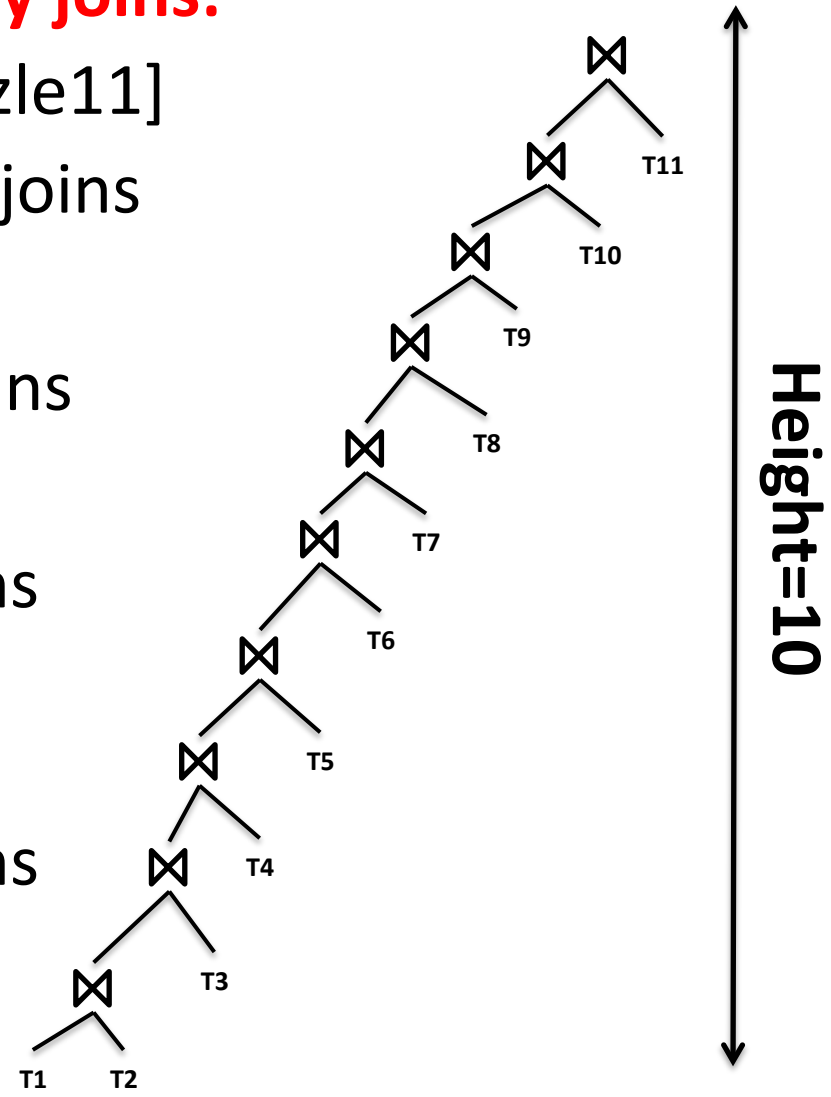


# Query plans on MapReduce

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

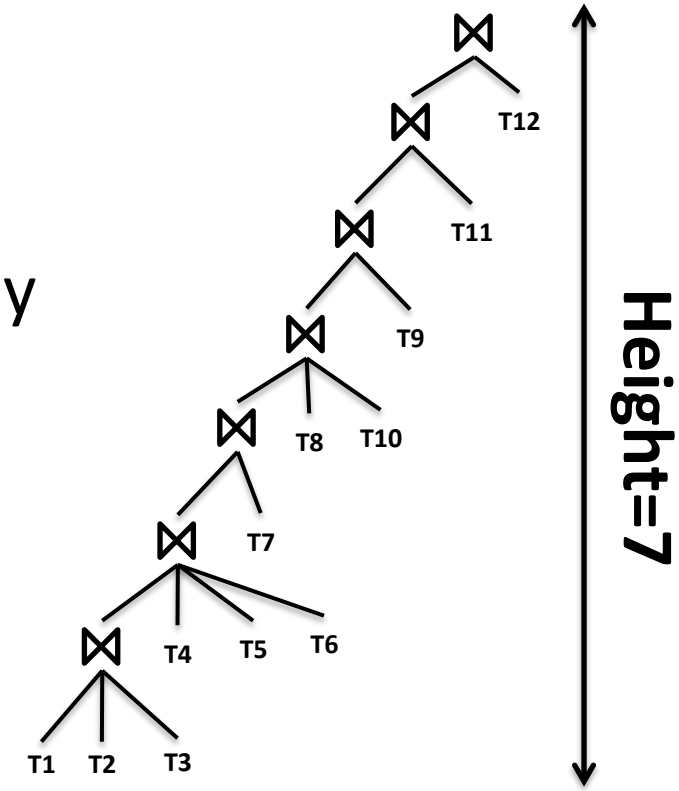
[Olston08][Rohloff10][Schatzle11]

- Left deep plans with n-ary joins
- Bushy plans with binary joins
- Bushy plans with n-ary joins only at leaves
- Bushy plans with n-ary joins



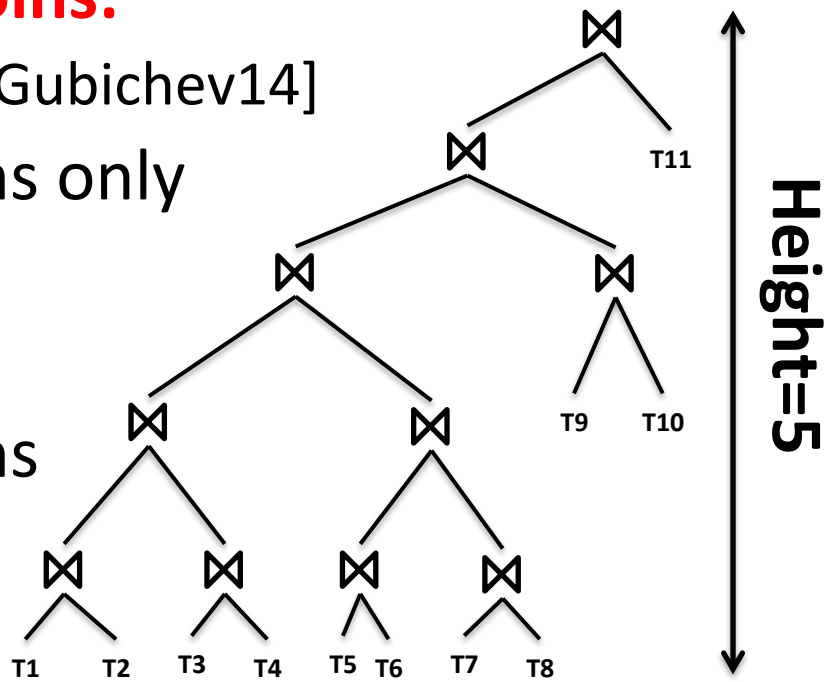
# Query optimization overview

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



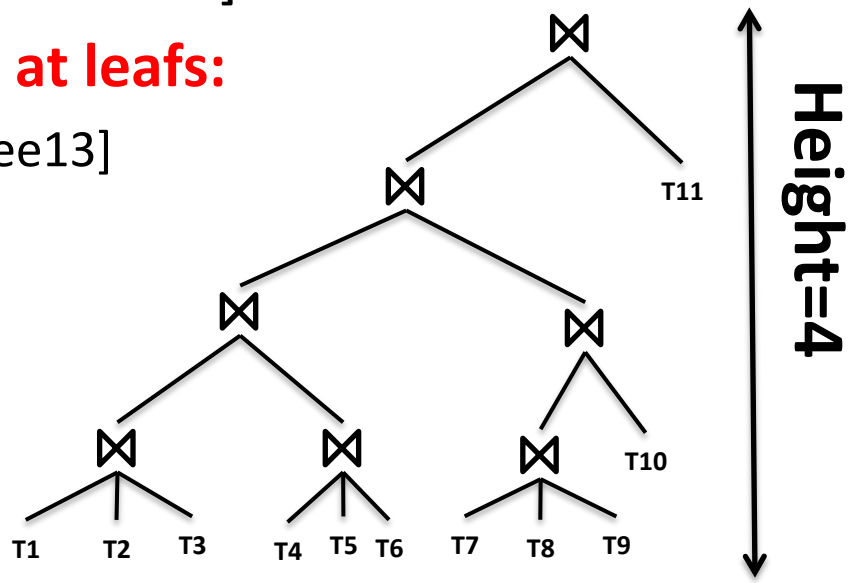
# Query optimization overview

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



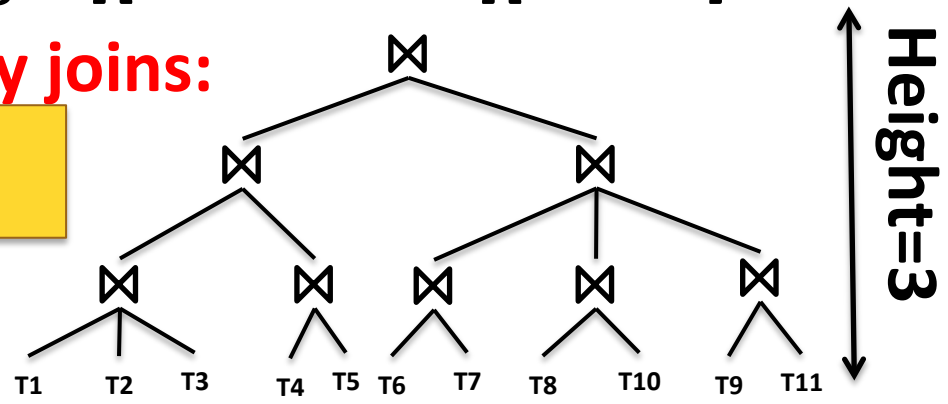
# Query optimization overview

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

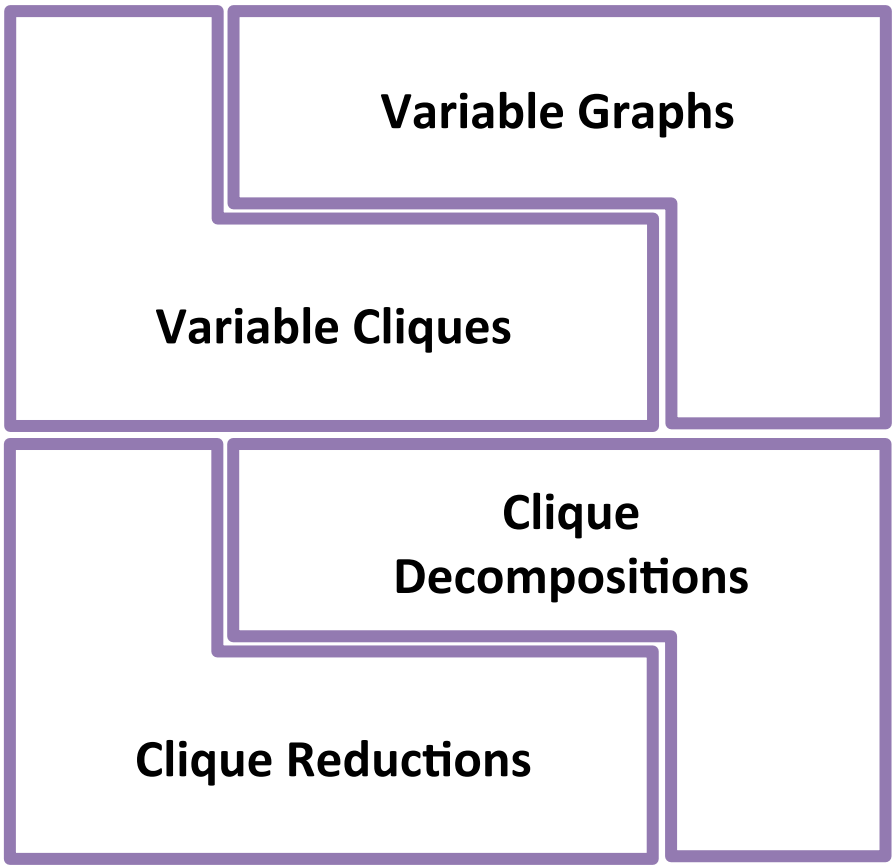


# Query optimization overview

- 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]** **CliqueSquare [Goasdoué15]**



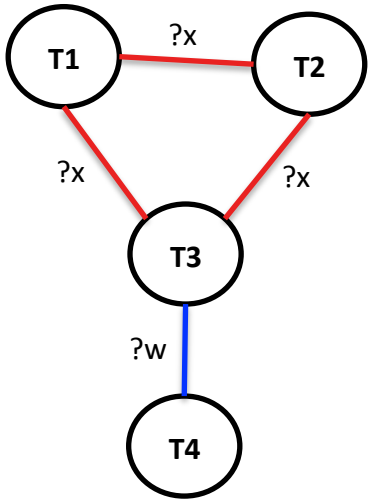
# CliqueSquare optimization algorithm: Components



# 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 .}



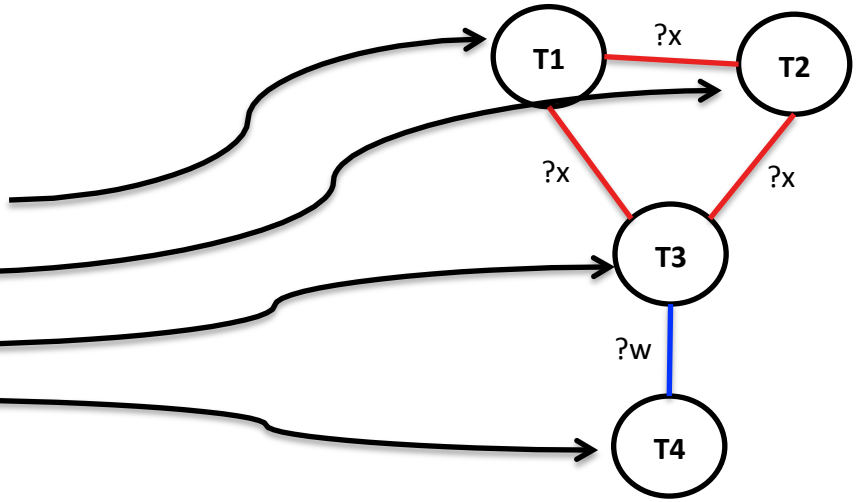

**Query**

**Variable graph**

# 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**

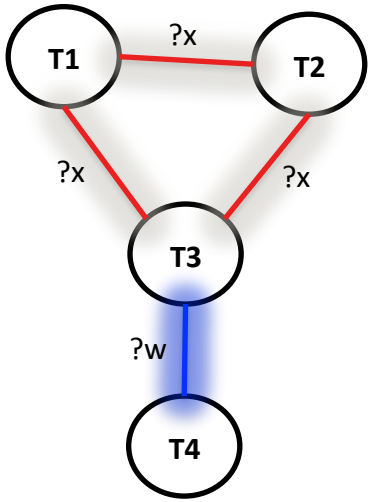
**Variable graph**

Each **triple pattern** corresponds to a **node** in the graph

# 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**

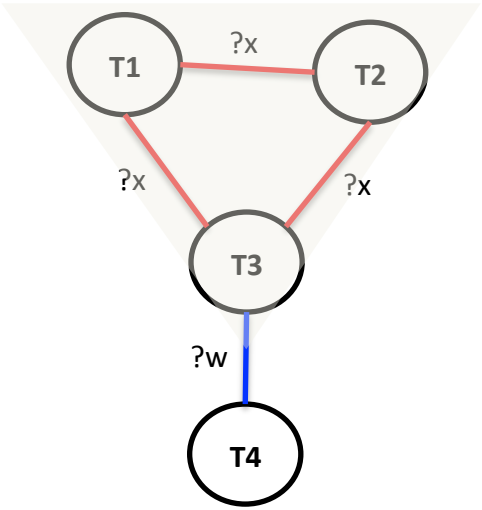
# CliqueSquare algorithm: Variable Cliques

- Model n-ary joins among relations

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

**Query**

**Maximal Clique of '?x'**




**Variable graph**

A **variable clique** is a set of nodes which are connected to each other with the **same** edge

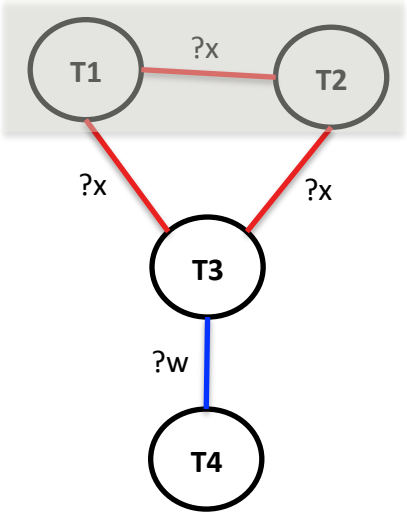
# CliqueSquare algorithm: Variable Cliques

- Model n-ary joins among relations

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

**Query**

**Partial Clique of '?x'**

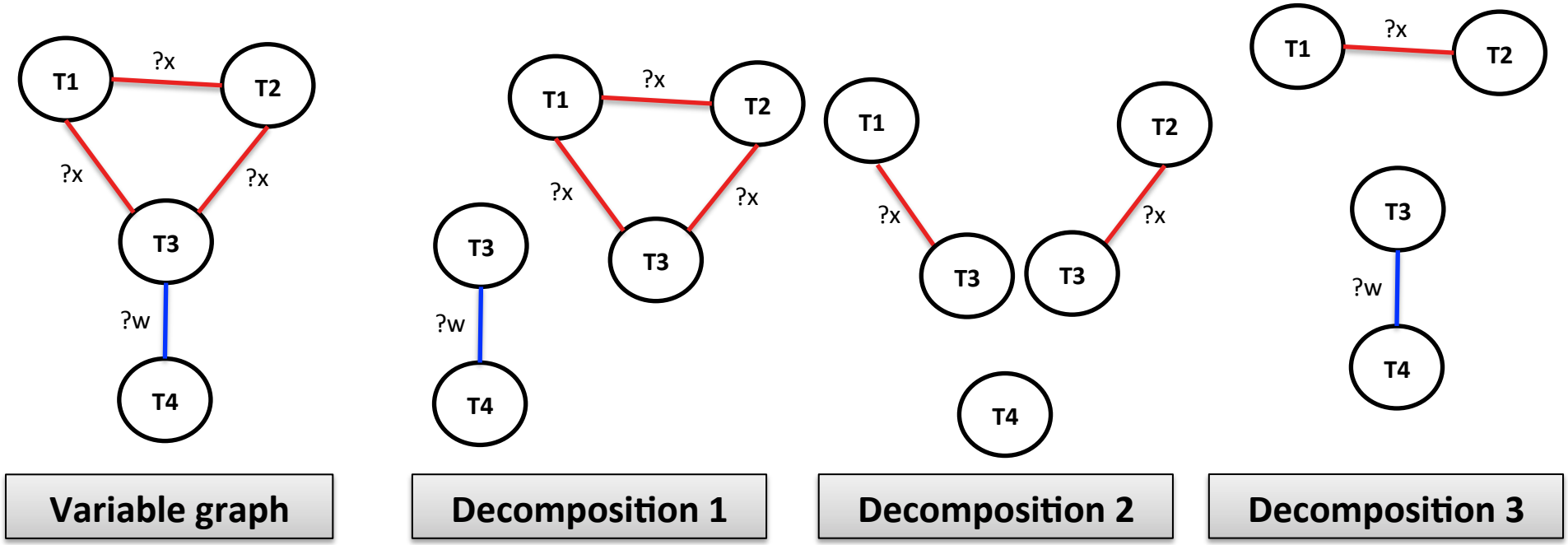



**Variable graph**

A **variable clique** is a set of nodes which are connected to each other with the **same** edge

# CliqueSquare algorithm: Clique Decompositions

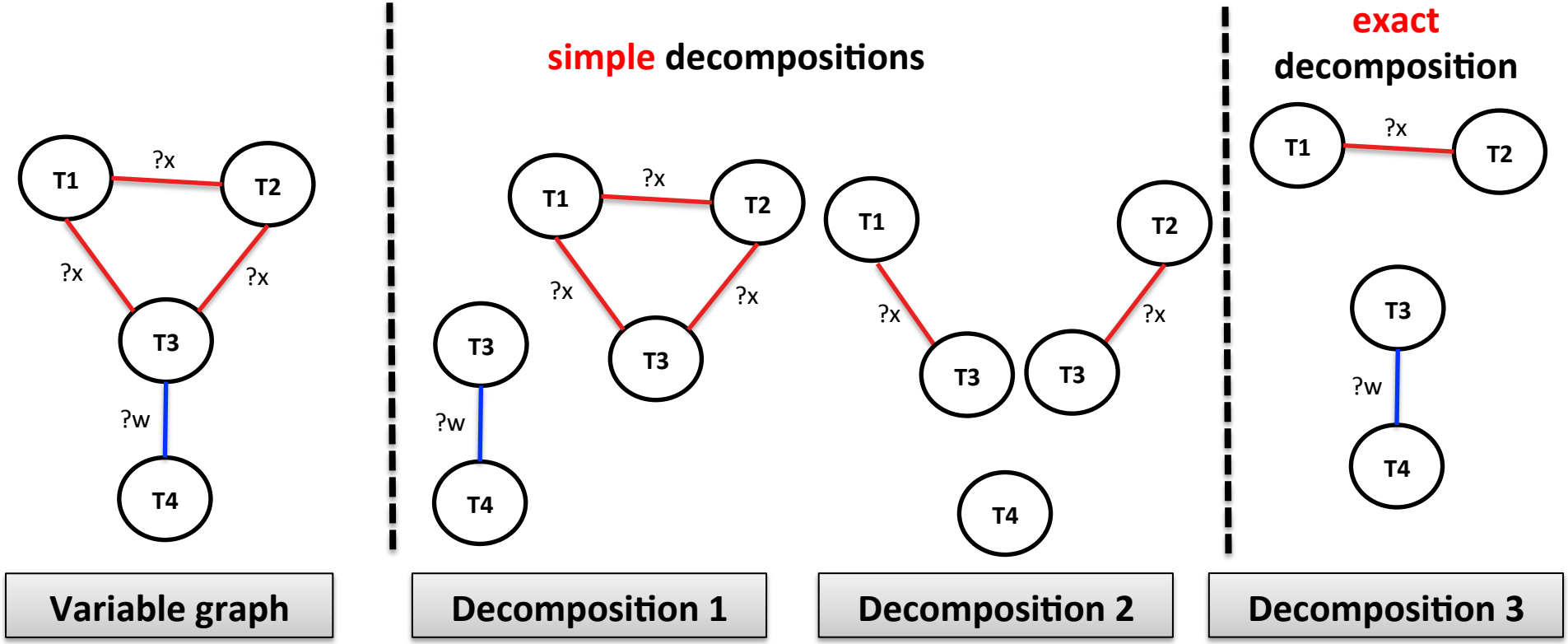
- Correspond to identifying partial results to be joined




A **clique decomposition** is a set of variable cliques which covers all the nodes of the graph

# CliqueSquare algorithm – Clique Decompositions

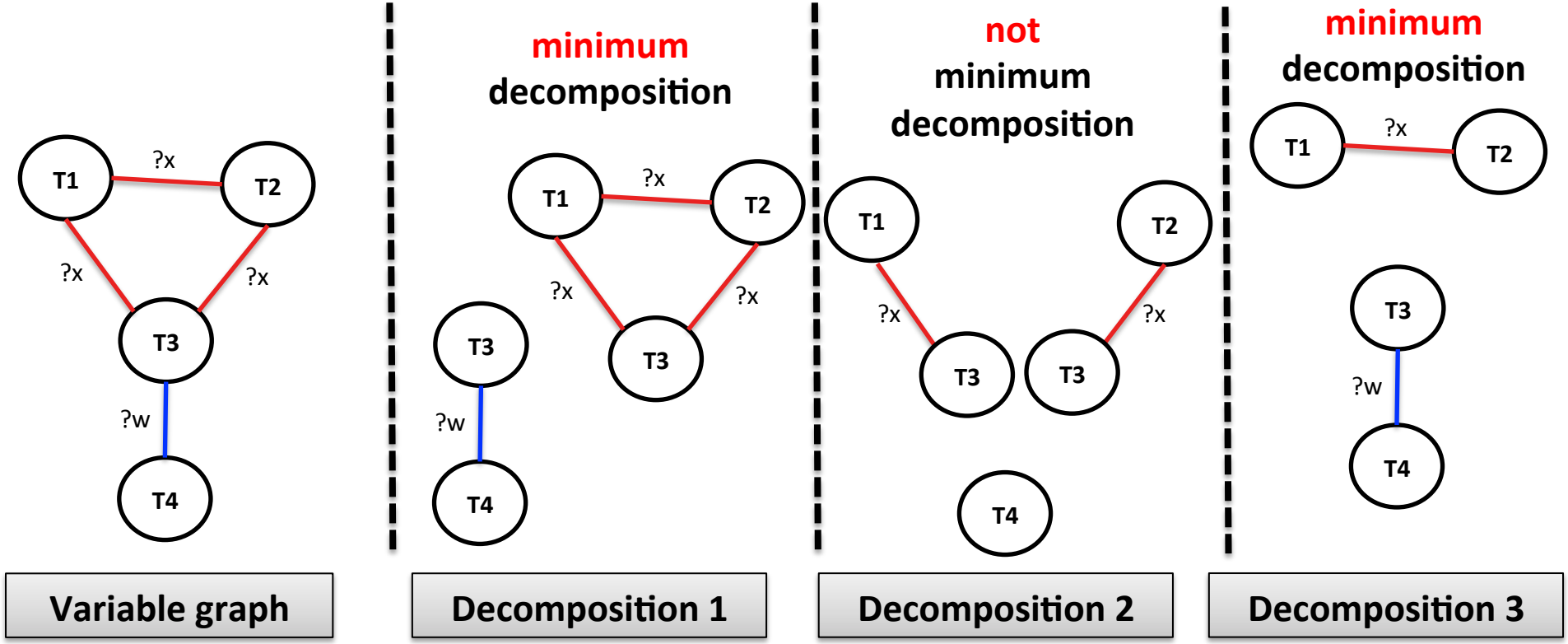
- Correspond to identifying partial results to be joined




A **clique decomposition** is a set of variable cliques which covers all the nodes of the graph

# CliqueSquare algorithm – Clique Decompositions

- Correspond to identifying partial results to be joined



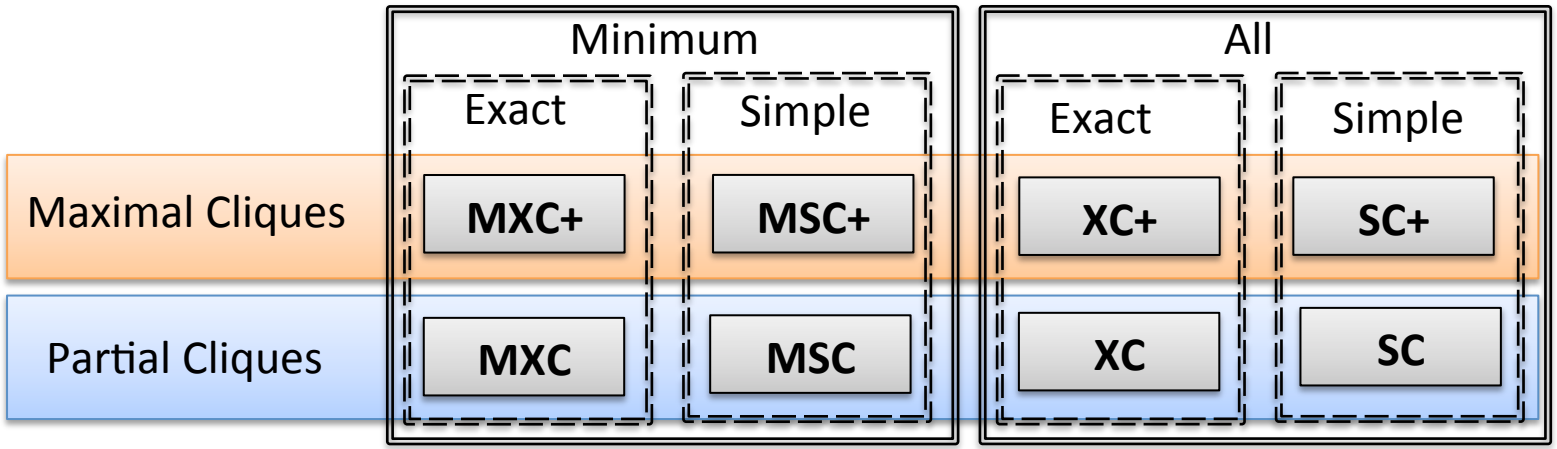

A **clique decomposition** is a set of variable cliques which covers all the nodes of the graph

# CliqueSquare algorithm – Clique Decompositions

- Correspond to identifying partial results to be joined

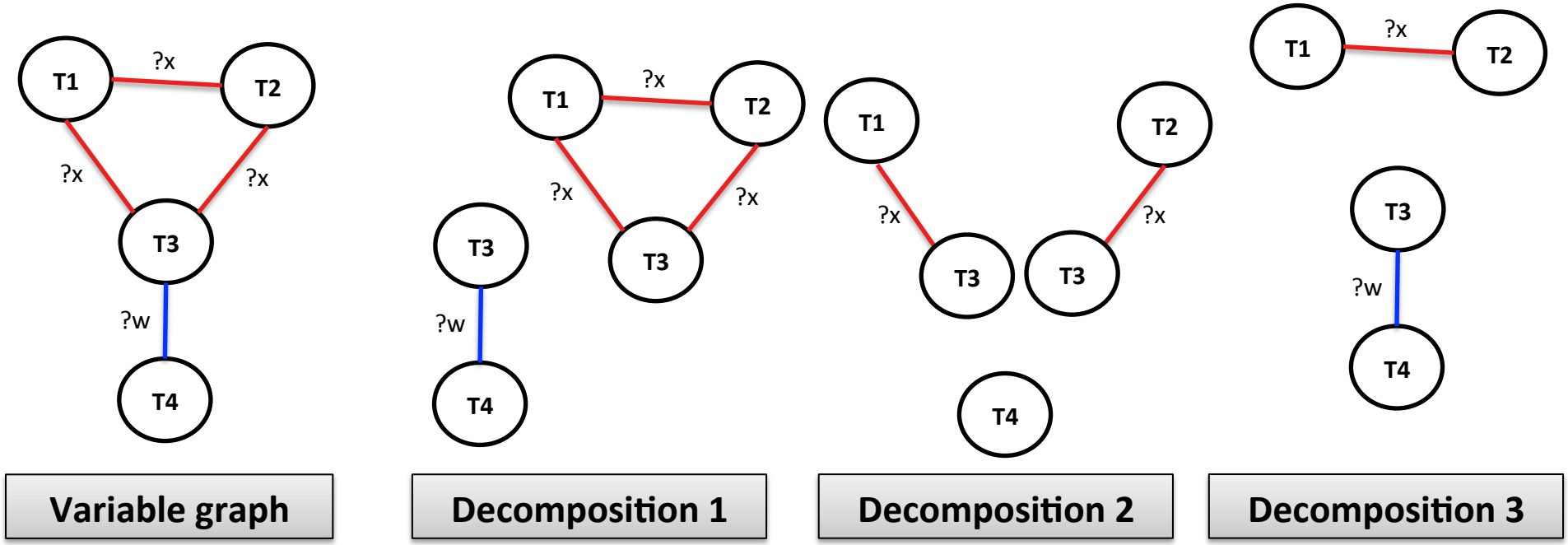
The clique decomposition is **not unique**

- We identify **eight** decomposition alternatives based on:
  - Type of variable cliques: maximal or partial
  - Type of cover: simple or exact
  - Cover size: minimum or not



# CliqueSquare algorithm – Clique Reductions

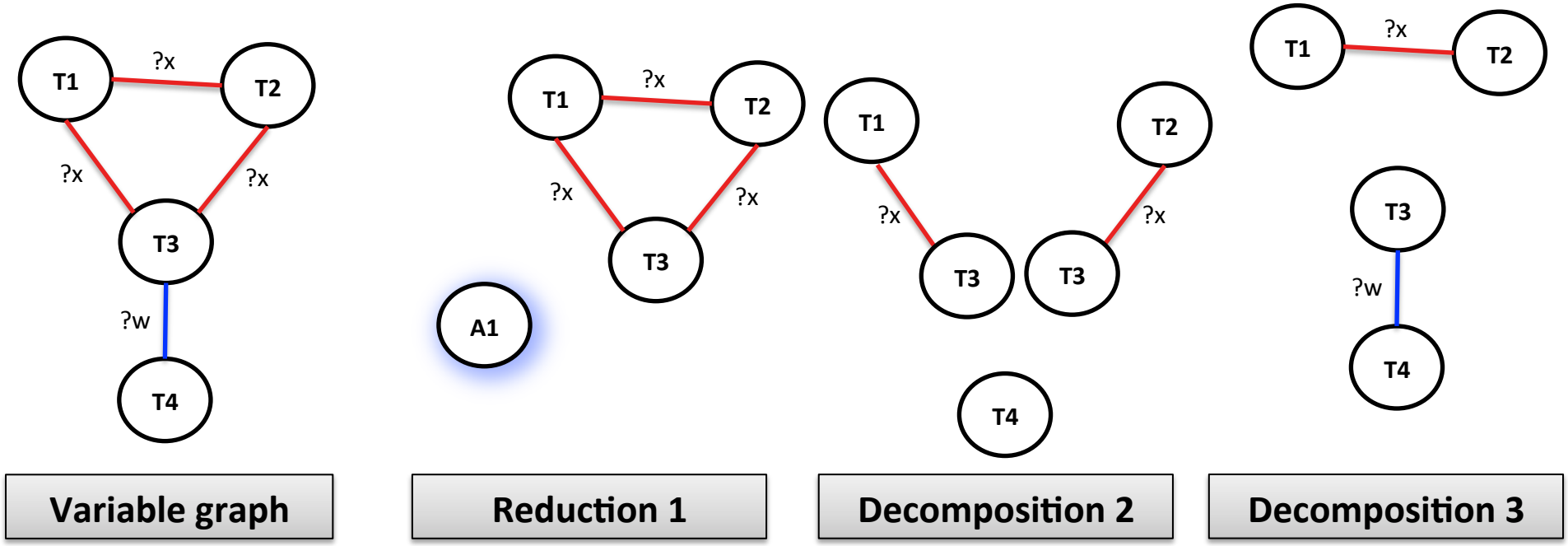
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

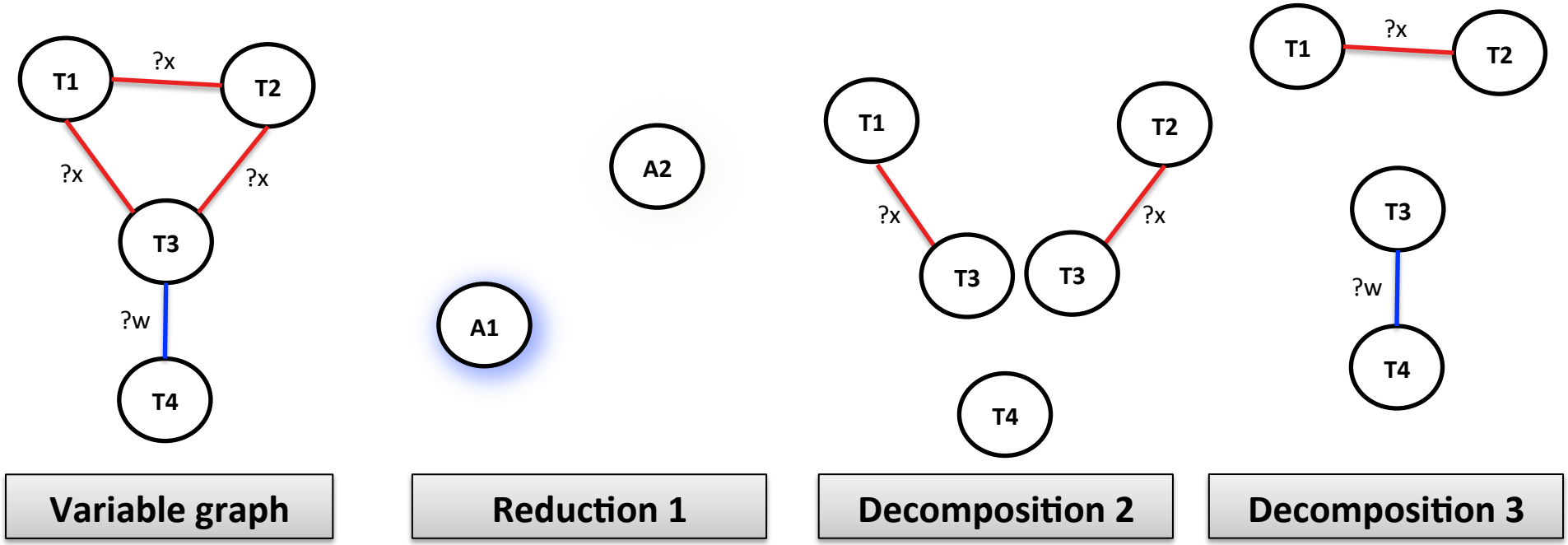
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

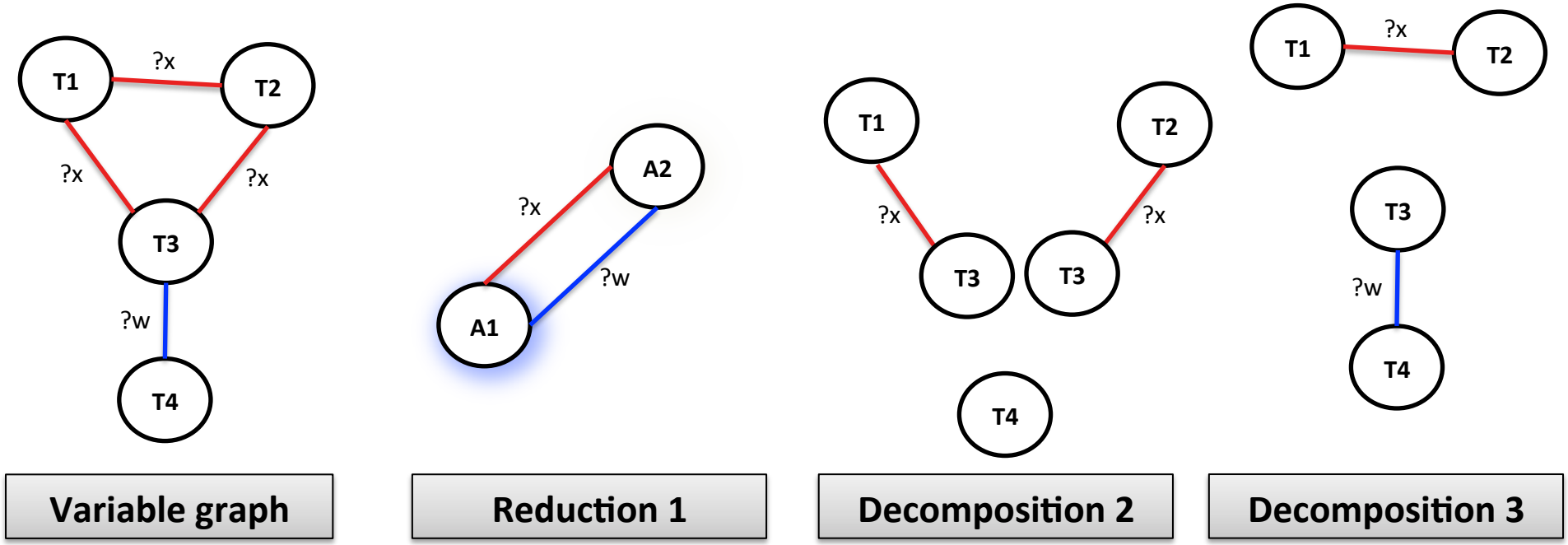
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

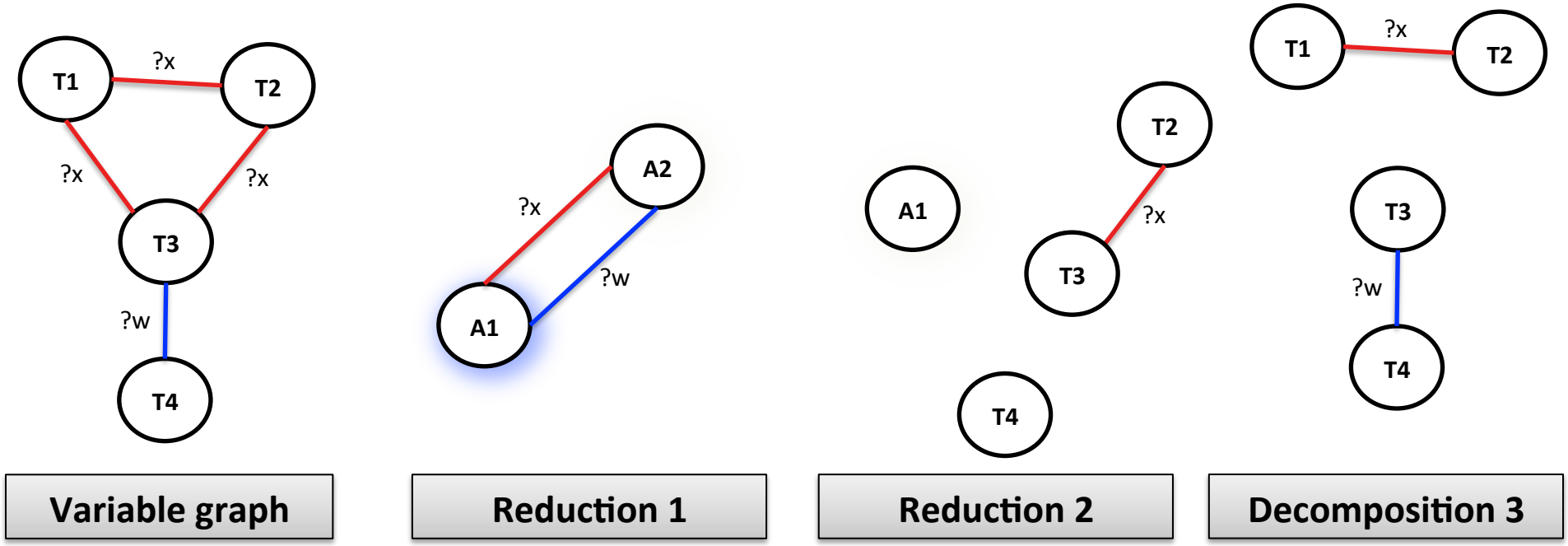
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

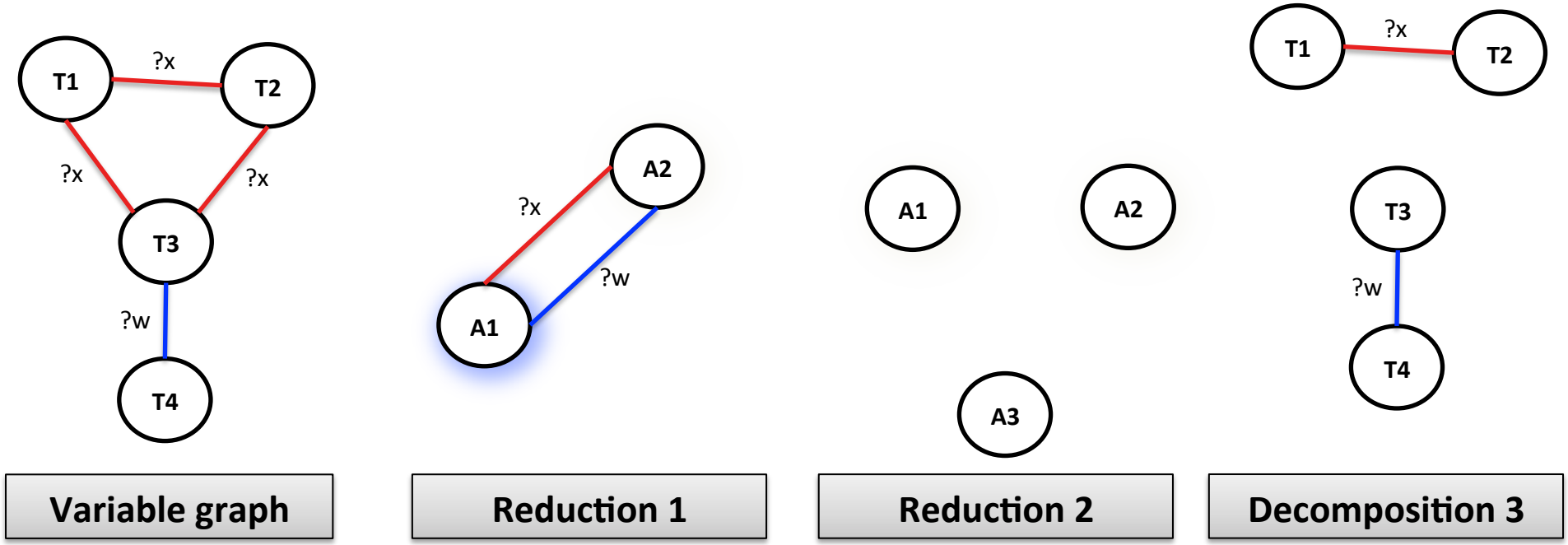
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

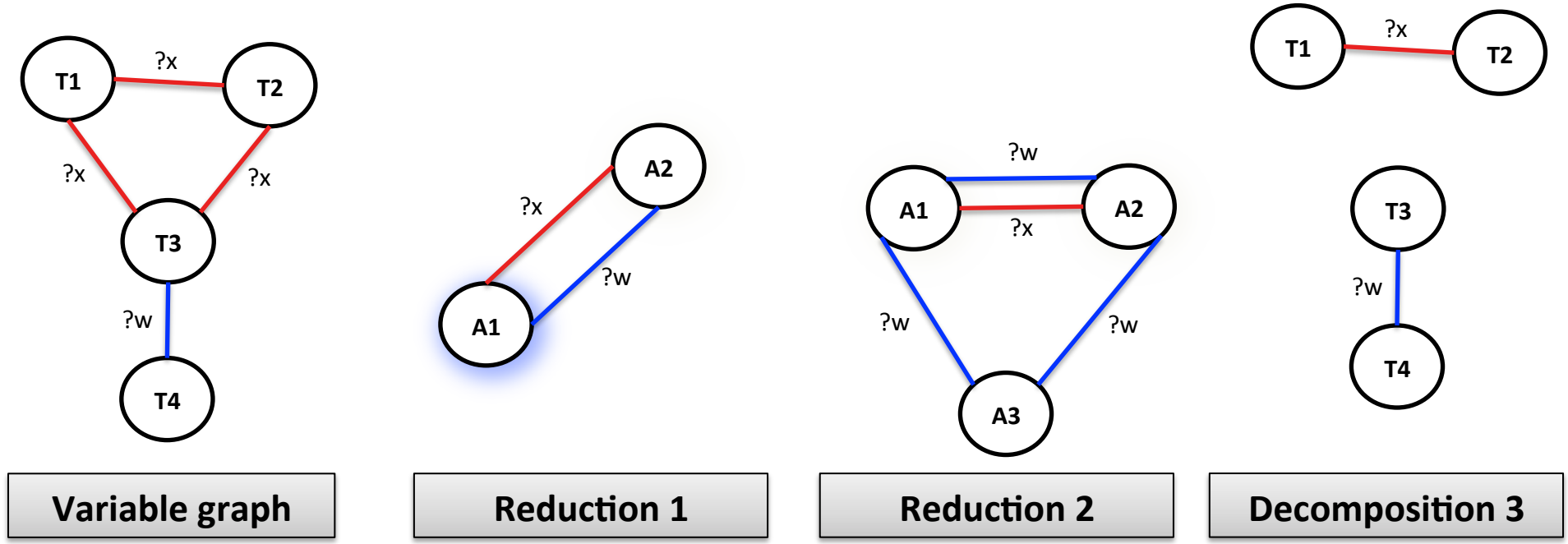
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

- Correspond to applying the joins identified by the decompositions

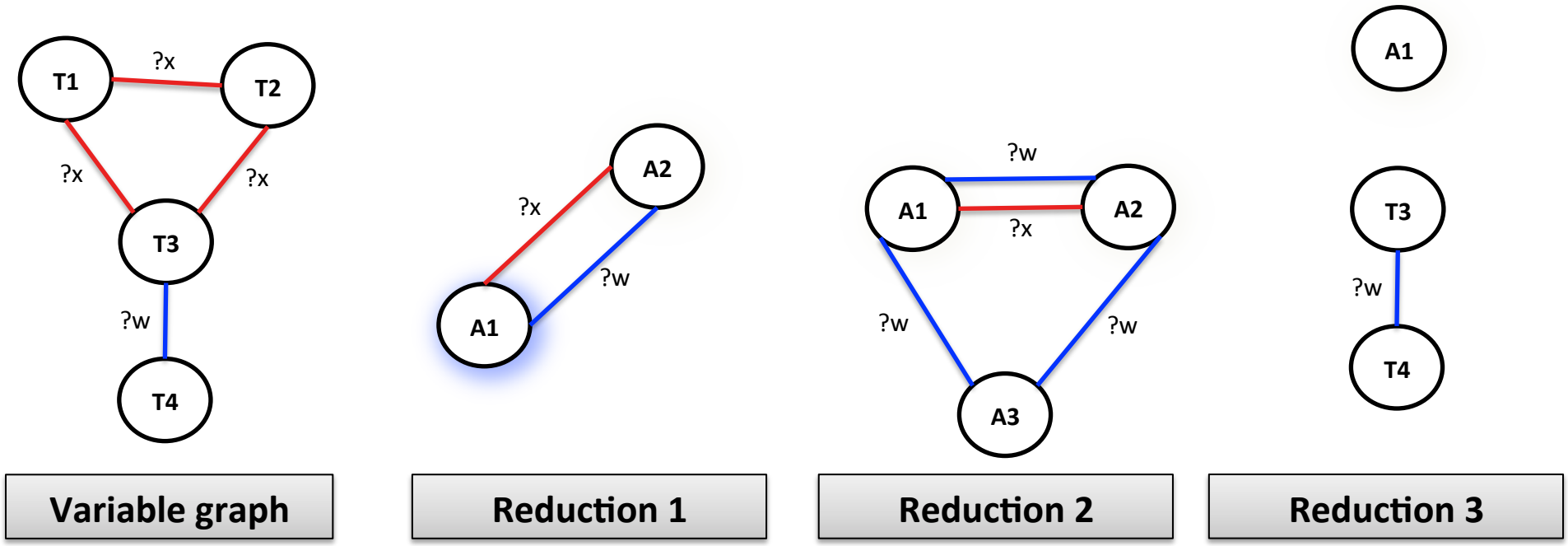



**Variable graph** | **Reduction 1** | **Reduction 2** | **Decomposition 3**

A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

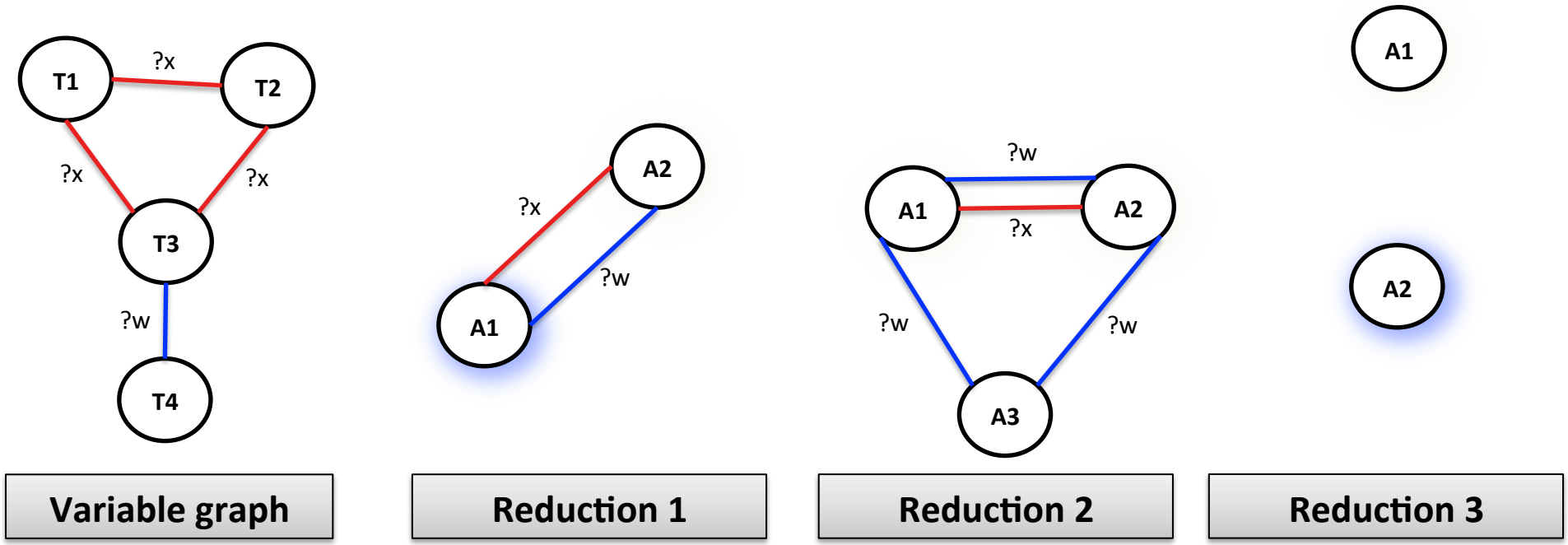
- Correspond to applying the joins identified by the decompositions




A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

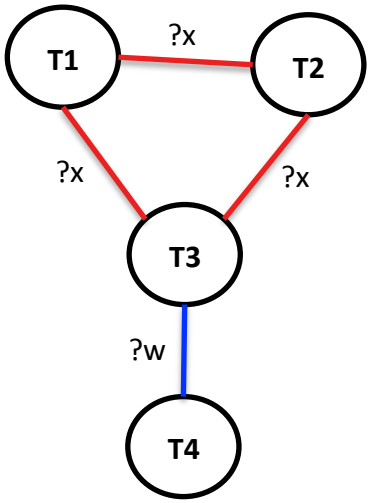
- Correspond to applying the joins identified by the decompositions



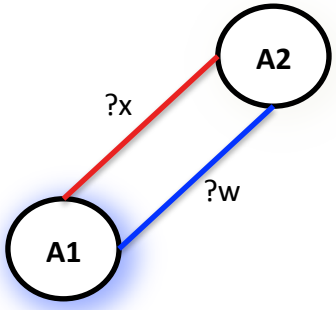

A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm – Clique Reductions

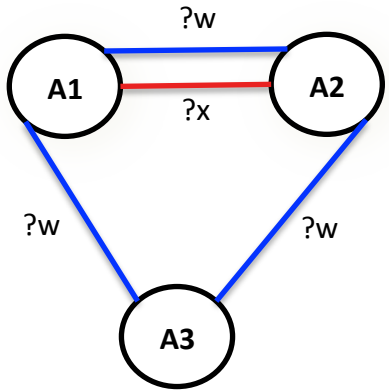
- Correspond to applying the joins identified by the decompositions



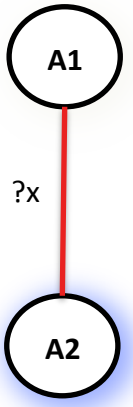

Variable graph




Reduction 1




Reduction 2




Reduction 3

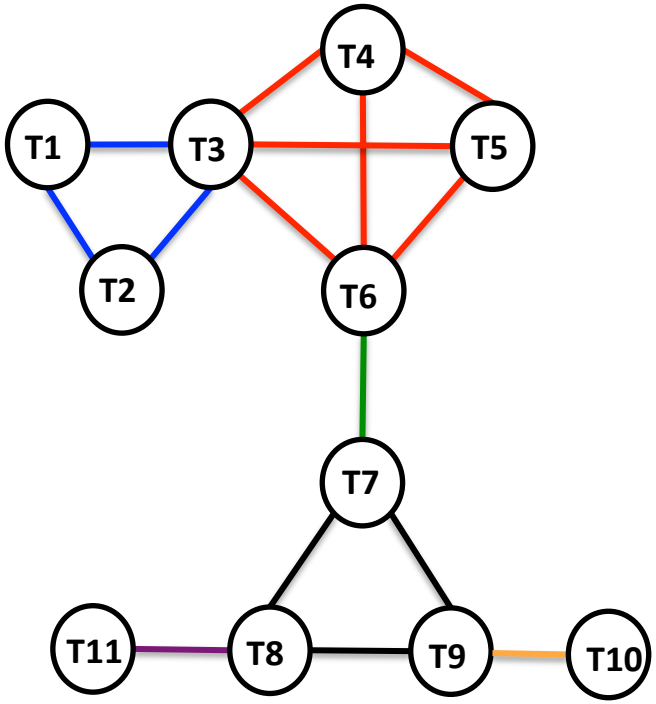
A **clique reduction** is a **new** variable graph: (i) each clique becomes a node in the **new** graph; (ii) cliques sharing a variable are connected

# CliqueSquare algorithm - Example

- Create the variable graph for the given 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> .}

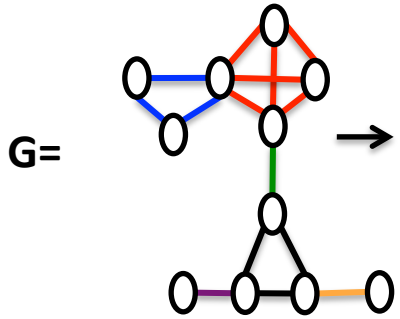
**Query**




**Variable Graph**

# CliqueSquare algorithm example

States



G=

States = States U G

CliqueDecomposition(G)

D=

Loop

[For each d in D]

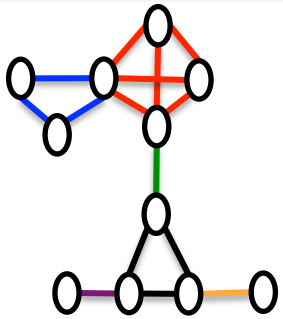
CliqueReduction(d)

G’

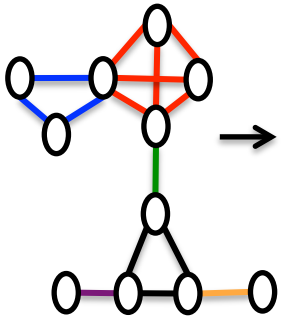
CliqueSquare(G’)

# CliqueSquare algorithm example

States



G=



States = States U G

CliqueDecomposition(G)

D=

Loop

[For each d in D]

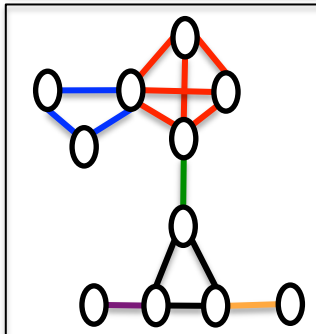
CliqueReduction(d)

G’

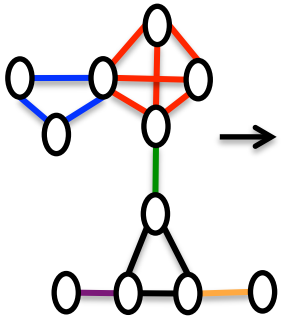
CliqueSquare(G’)

# CliqueSquare algorithm example

States



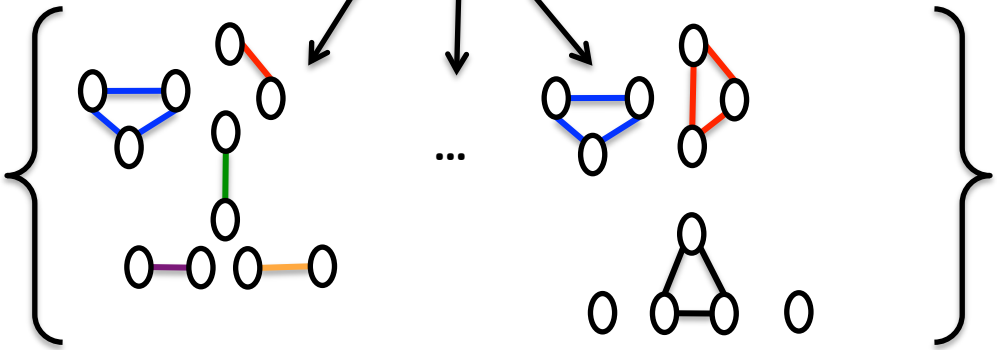
G=



States = States U G

**CliqueDecomposition(G)**

D=



Loop

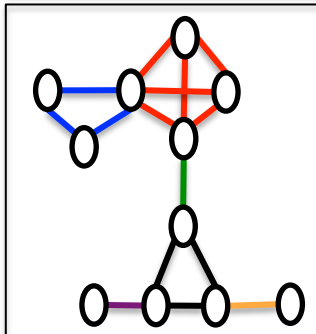
[For each d in D]

← CliqueReduction(d) ←

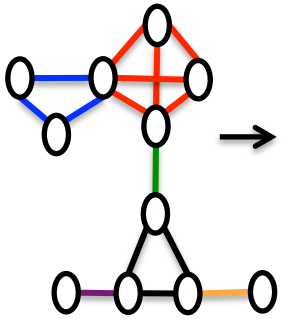
G’ → CliqueSquare(G’)

# CliqueSquare algorithm example

States



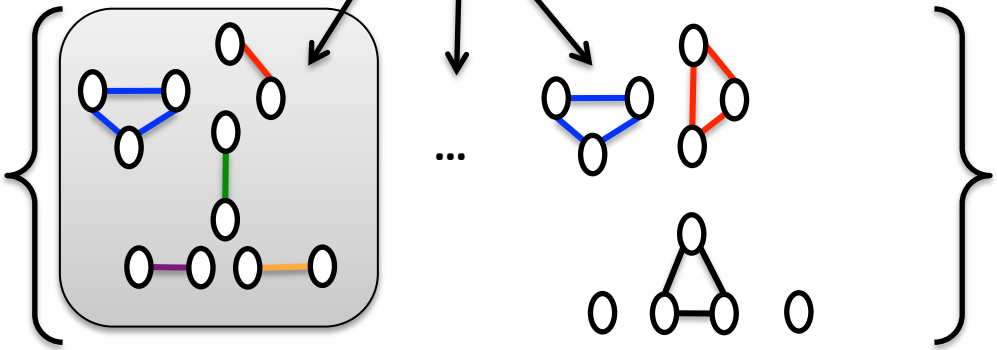
G=



States = States U G

CliqueDecomposition(G)

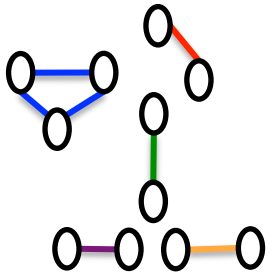
D=



Loop

**[For each d in D]**

CliqueReduction(d)

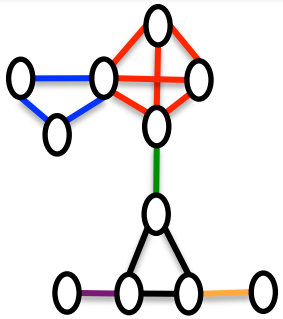


G’

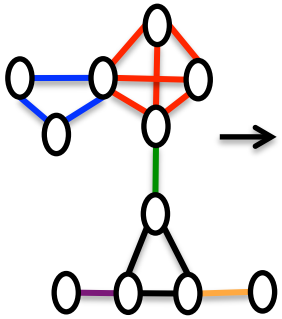
CliqueSquare(G’)

# CliqueSquare algorithm example

States



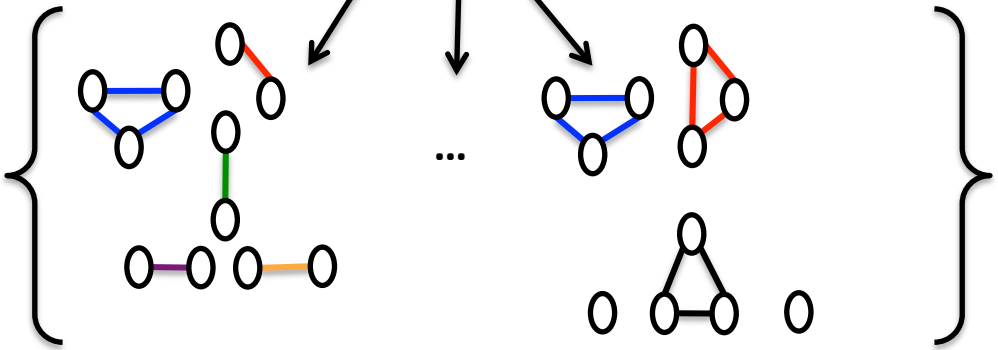
G=



States = States U G

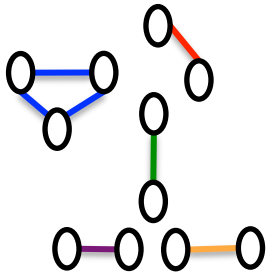
CliqueDecomposition(G)

D=



Loop

[For each d in D]



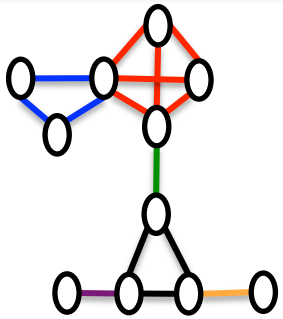
**CliqueReduction(d)**

G’

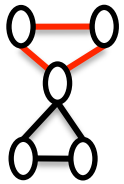
CliqueSquare(G’)

# CliqueSquare algorithm example

States



G=



States = States U G

CliqueDecomposition(G)

D=

Loop

[For each d in D]

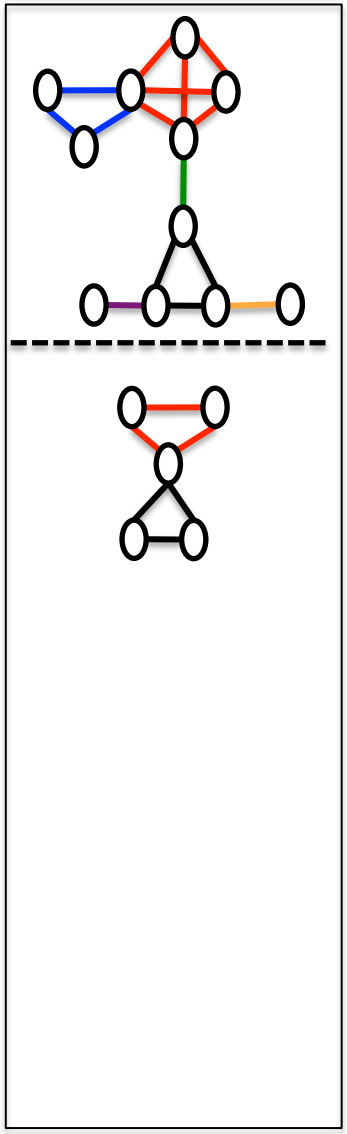
CliqueReduction(d)

G’

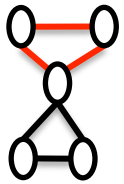
CliqueSquare(G’)

# CliqueSquare algorithm example

States



G=



→ **States = States U G**

↓

CliqueDecomposition(G)

↓

D= { }

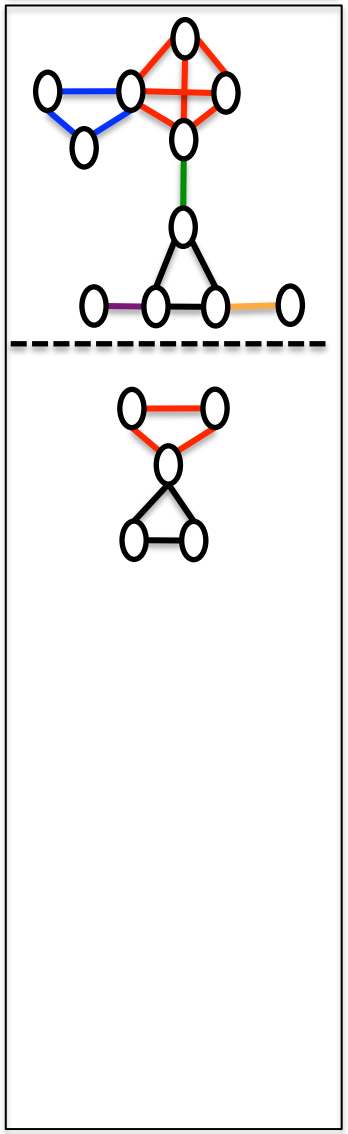
Loop

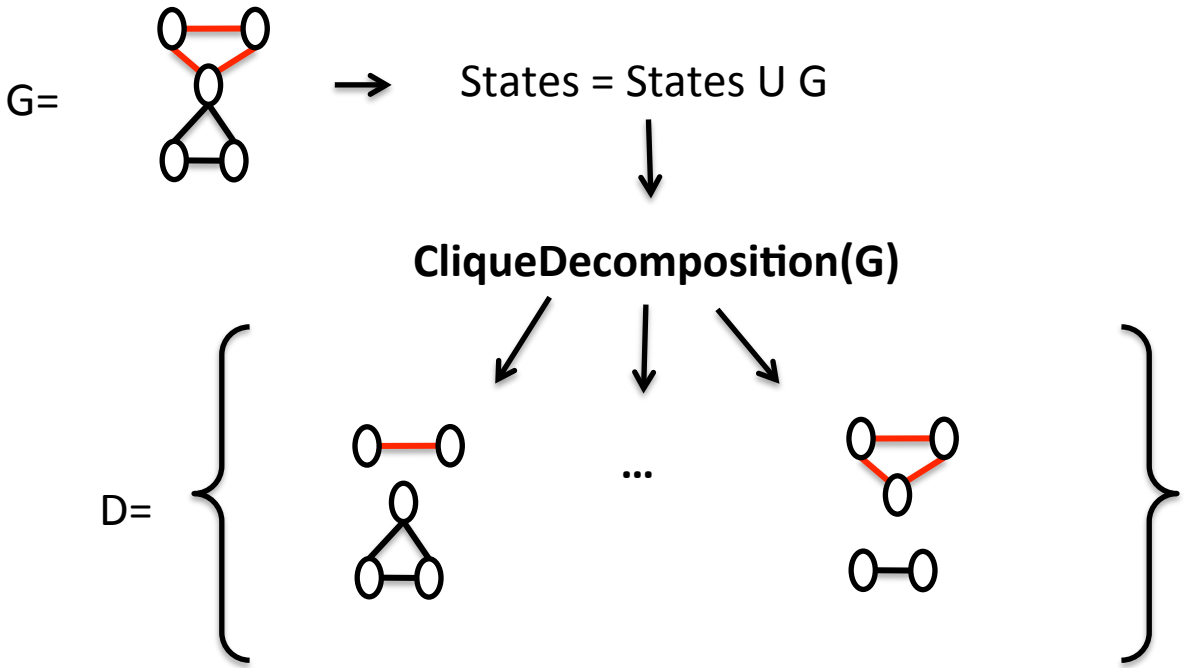
[For each d in D]

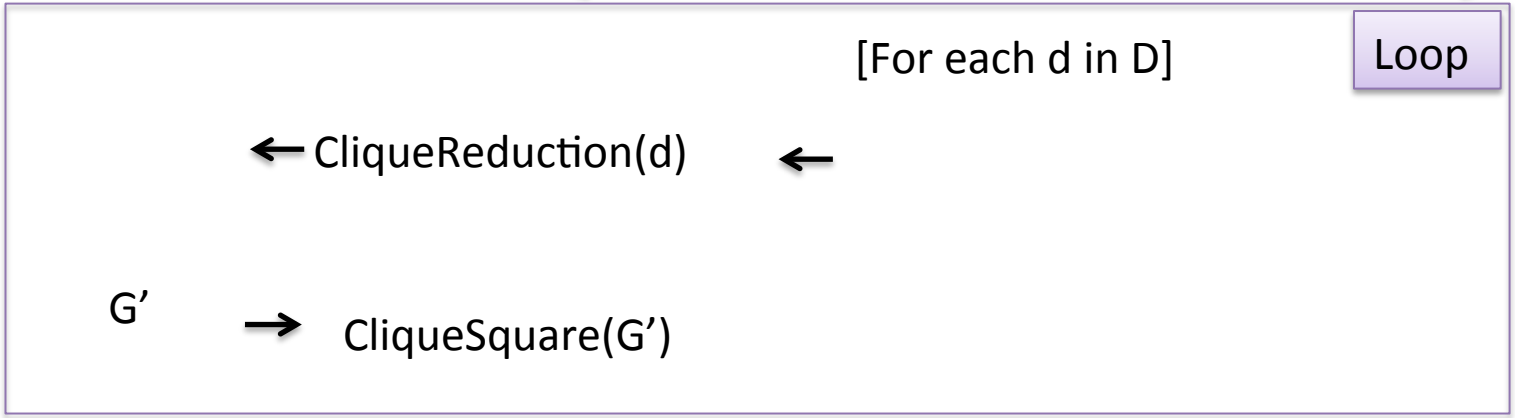
← CliqueReduction(d) ←

G’ → CliqueSquare(G’)

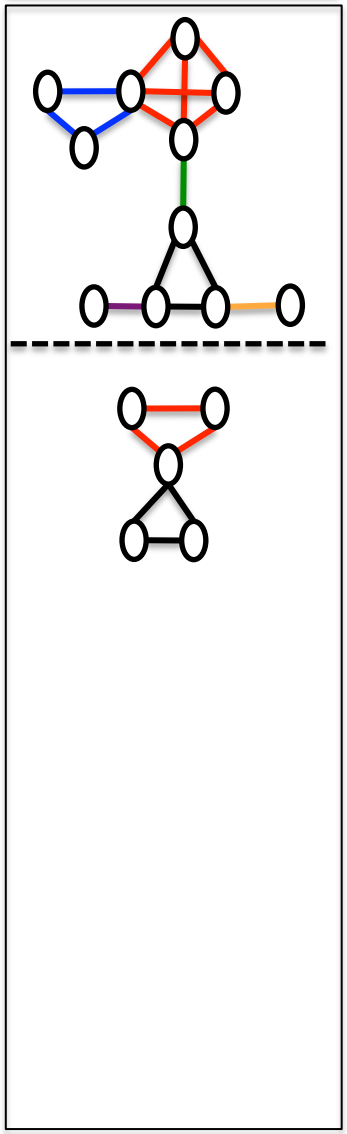
# CliqueSquare algorithm example

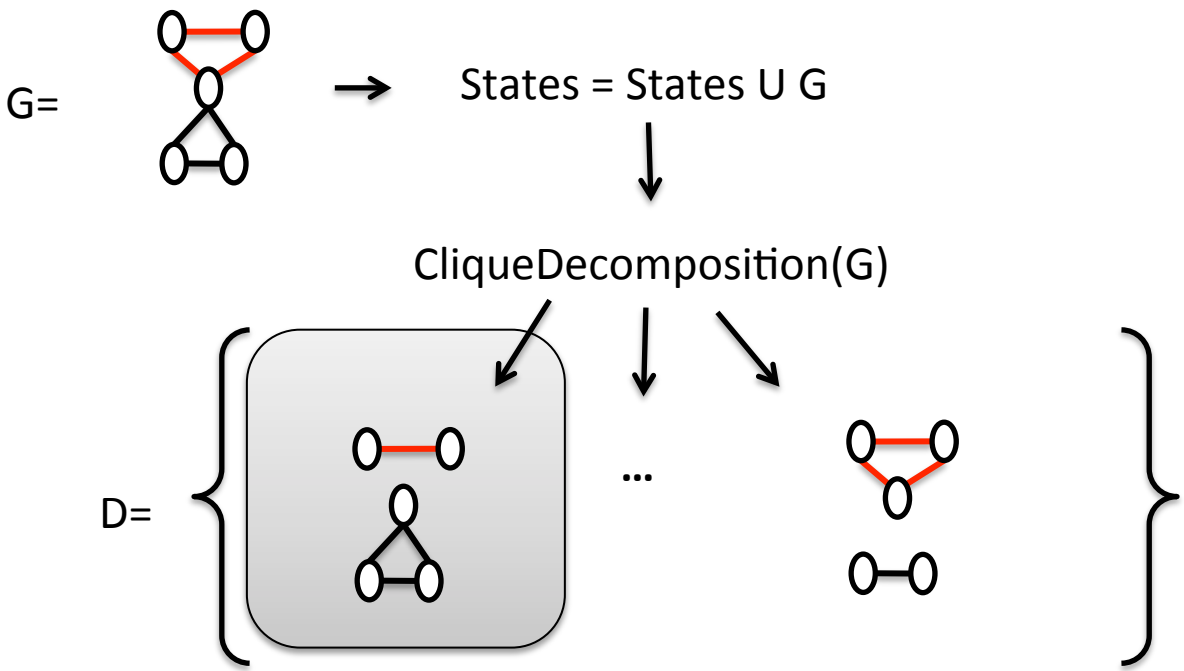


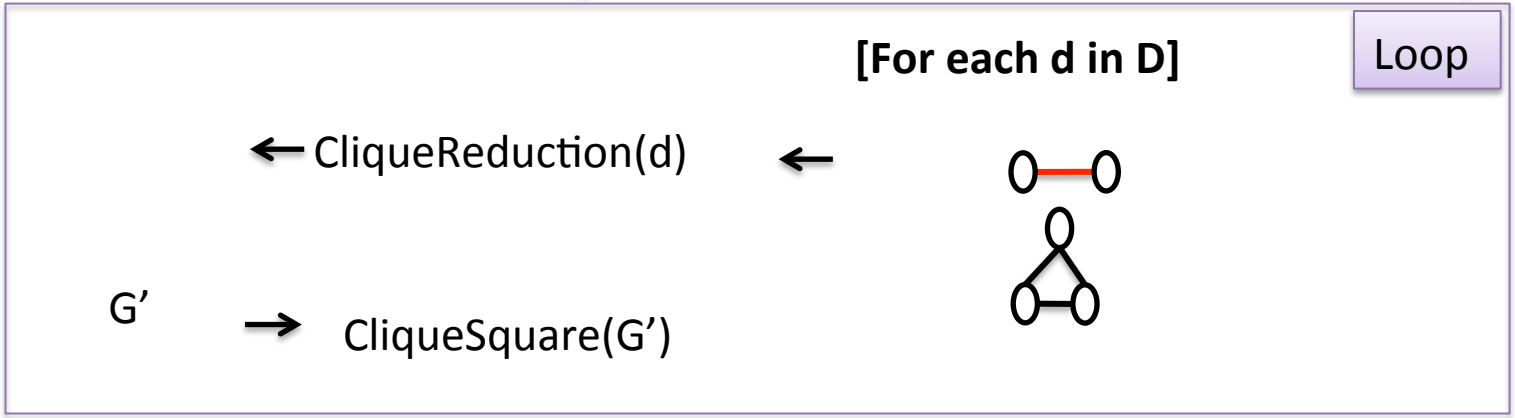



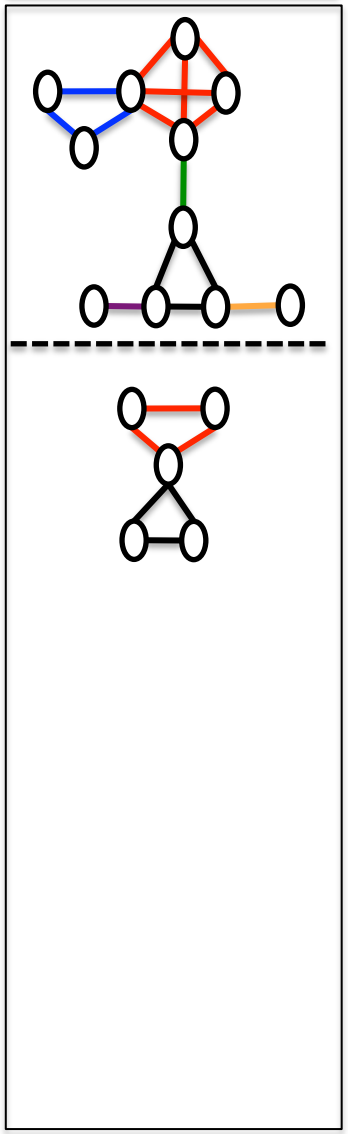
# CliqueSquare algorithm example

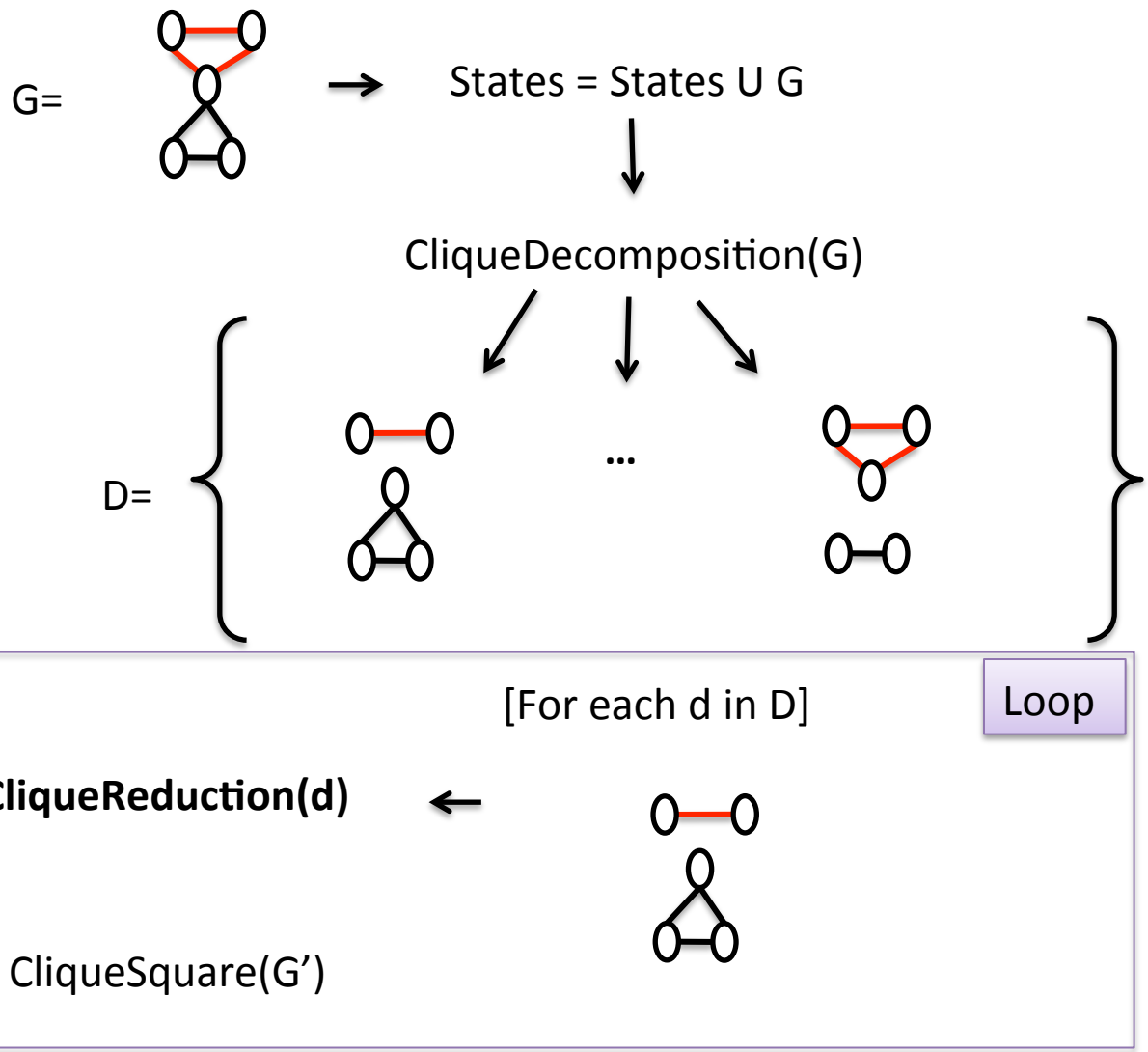


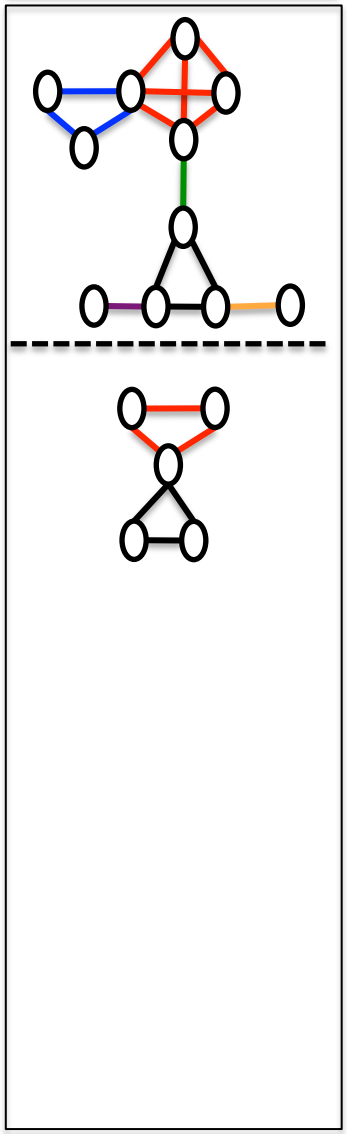



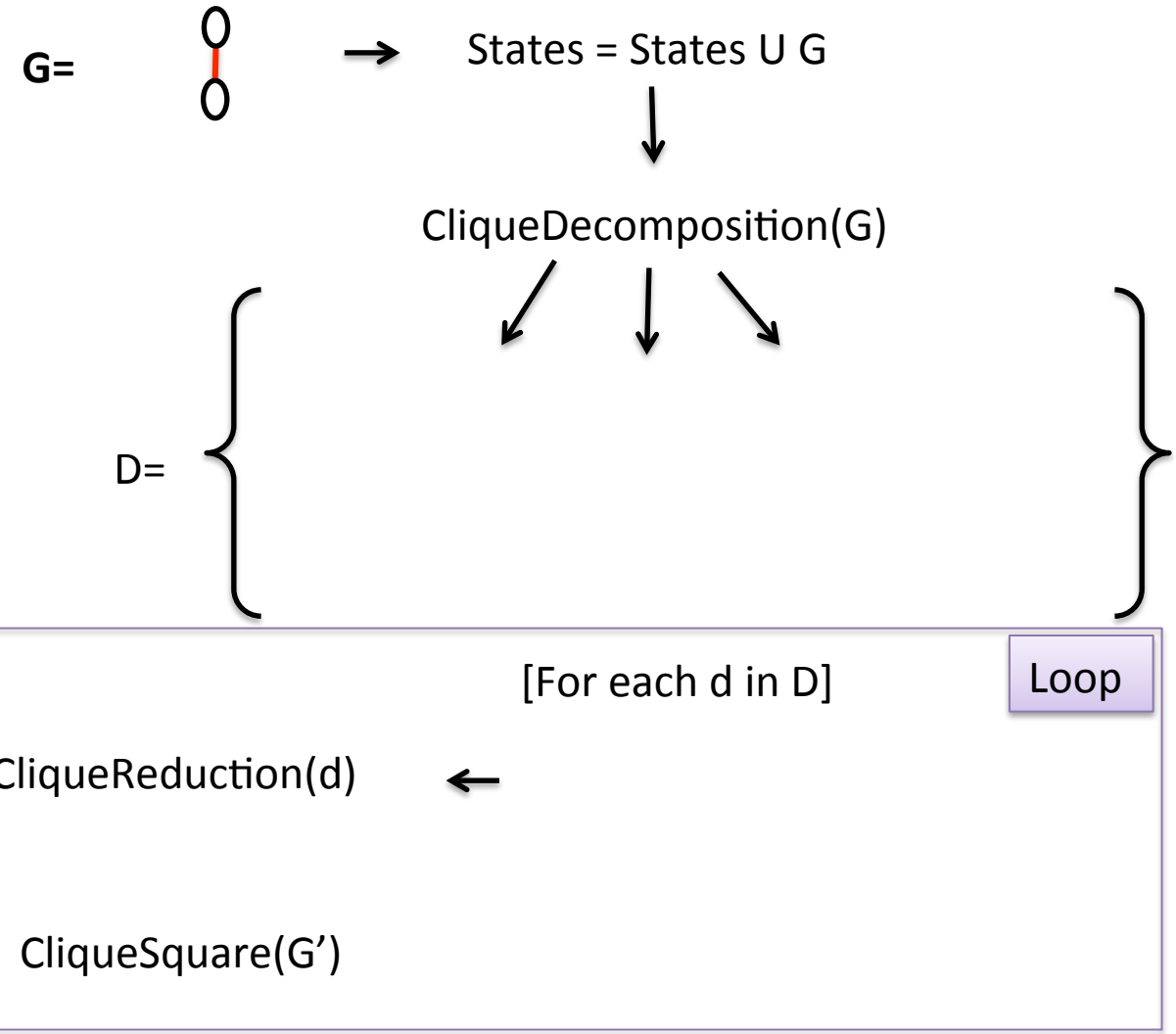
# CliqueSquare algorithm example



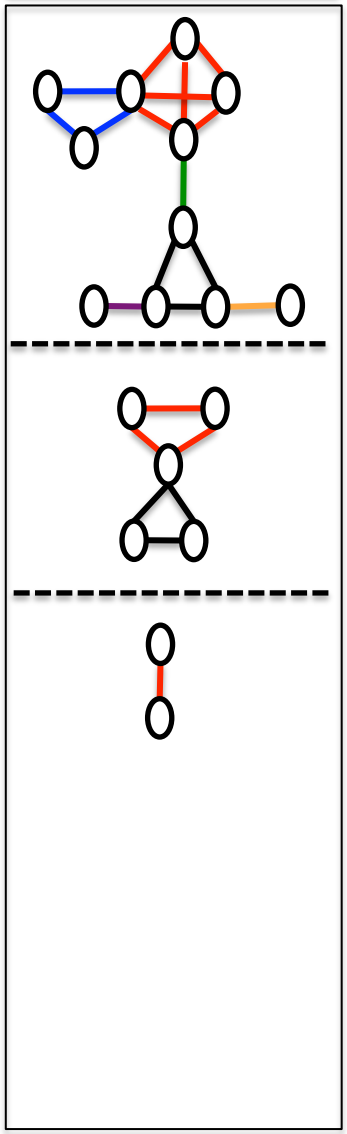



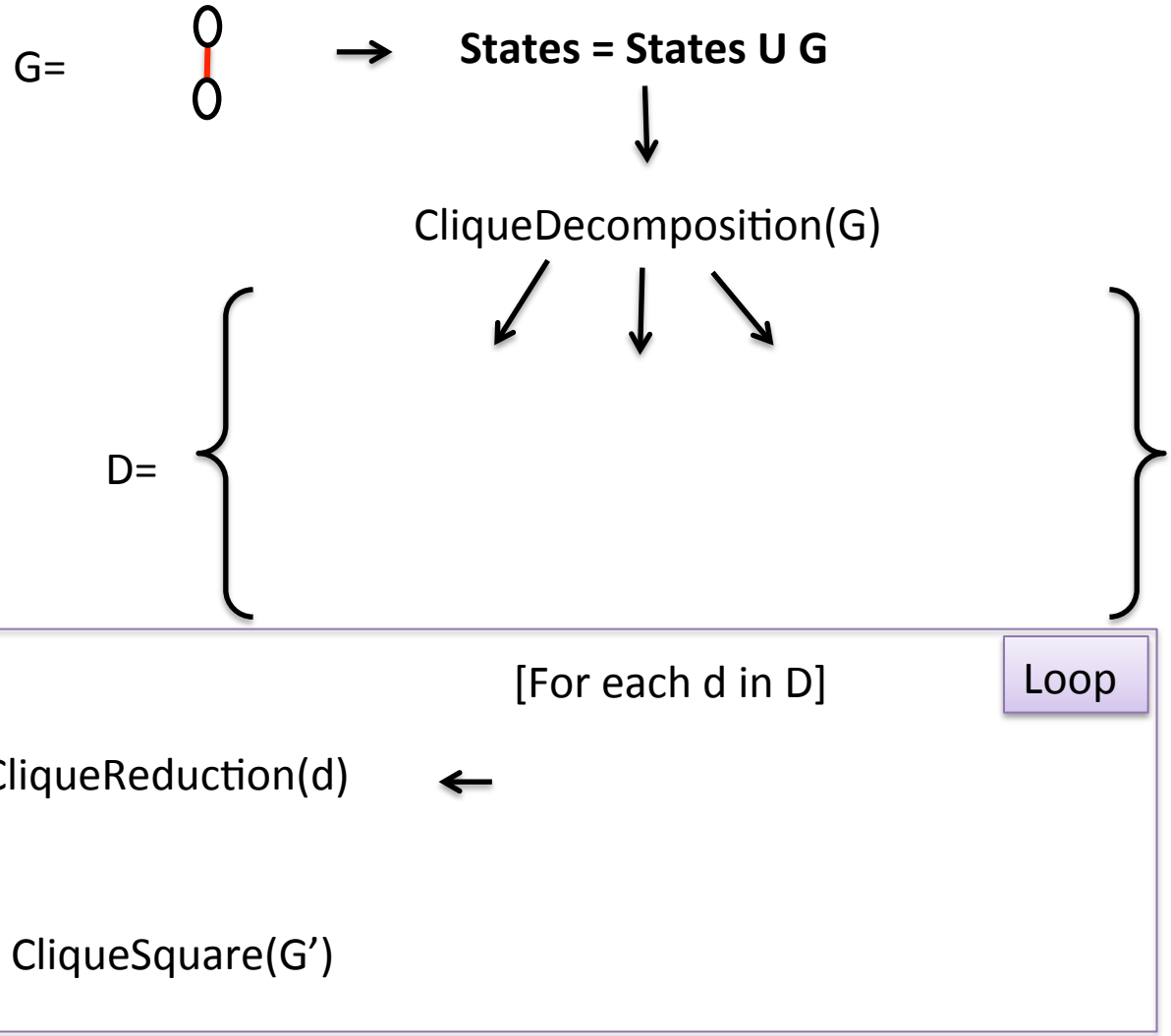
# CliqueSquare algorithm example



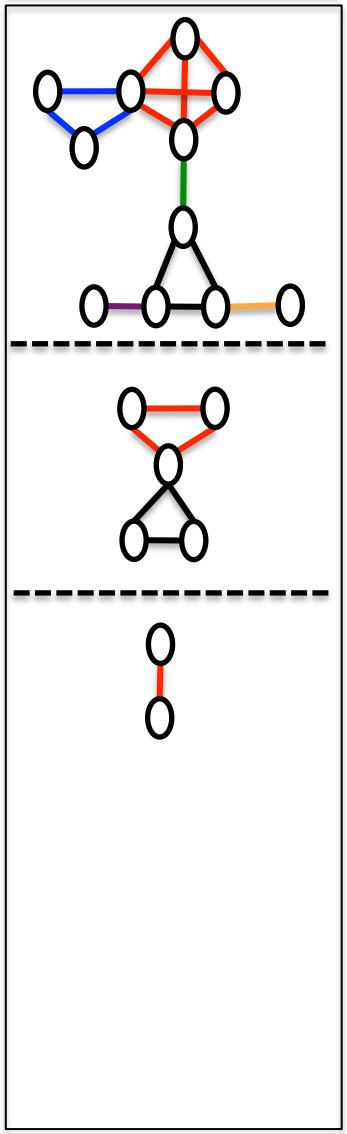



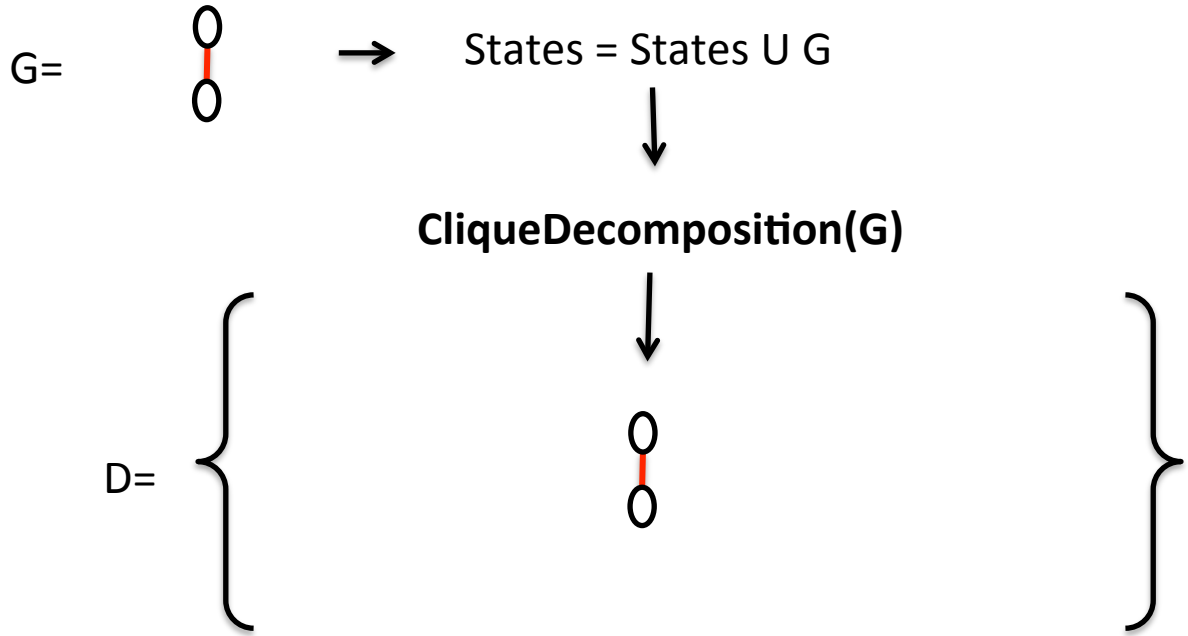
# CliqueSquare algorithm example



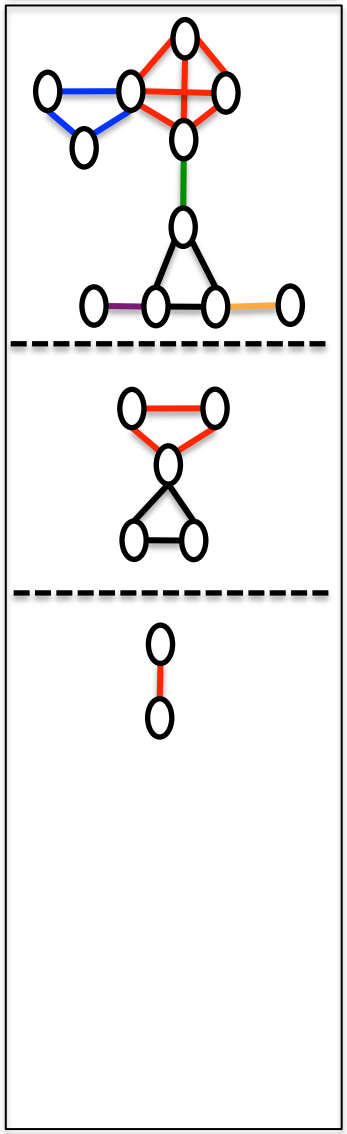



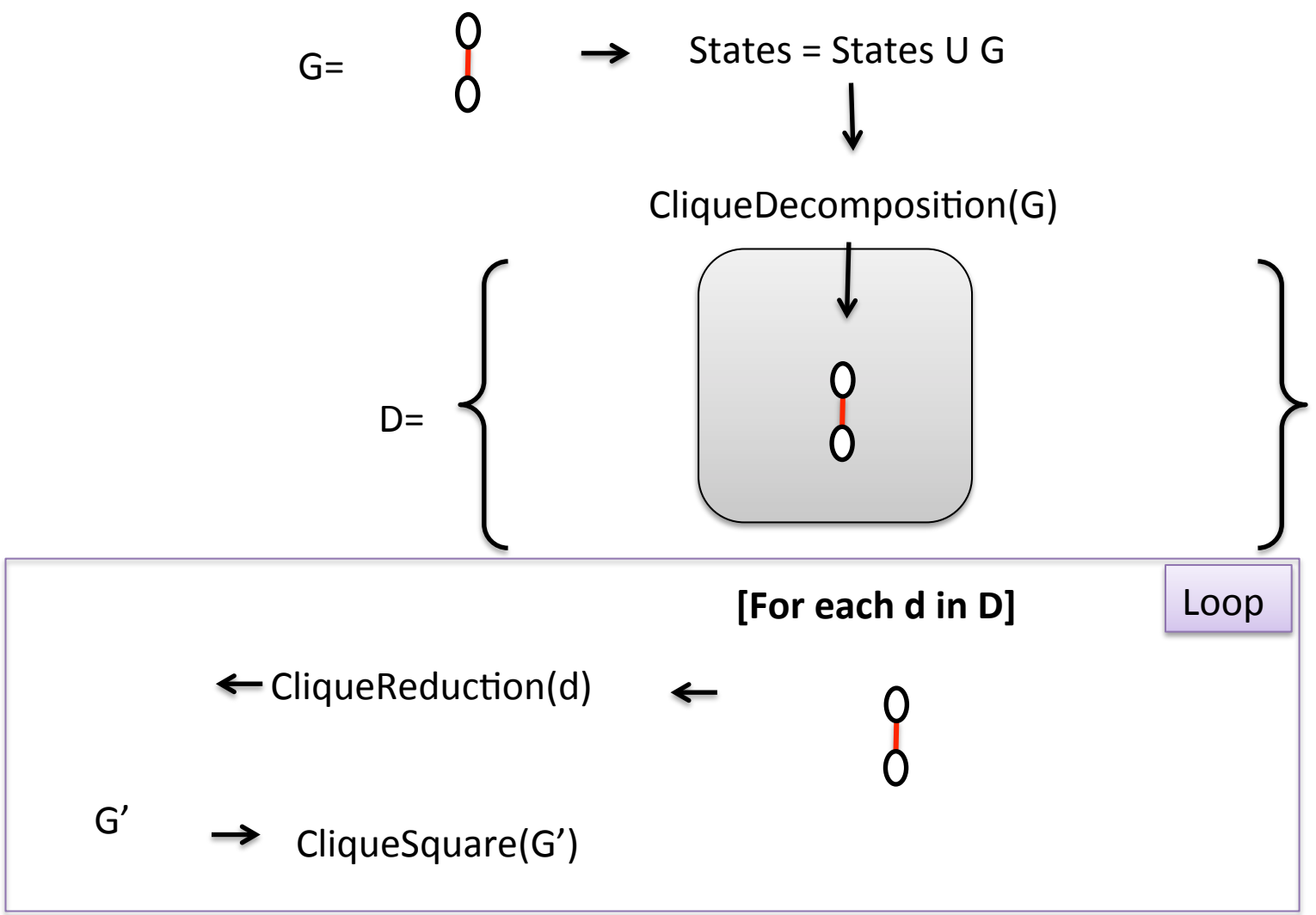
# CliqueSquare algorithm example



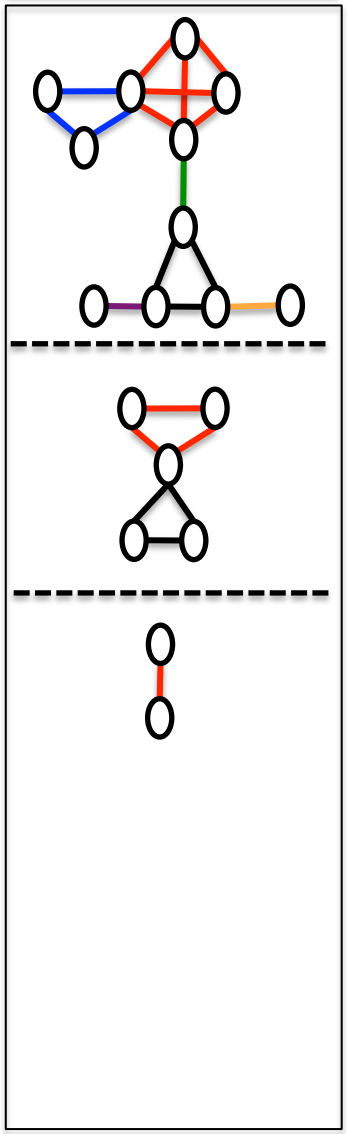



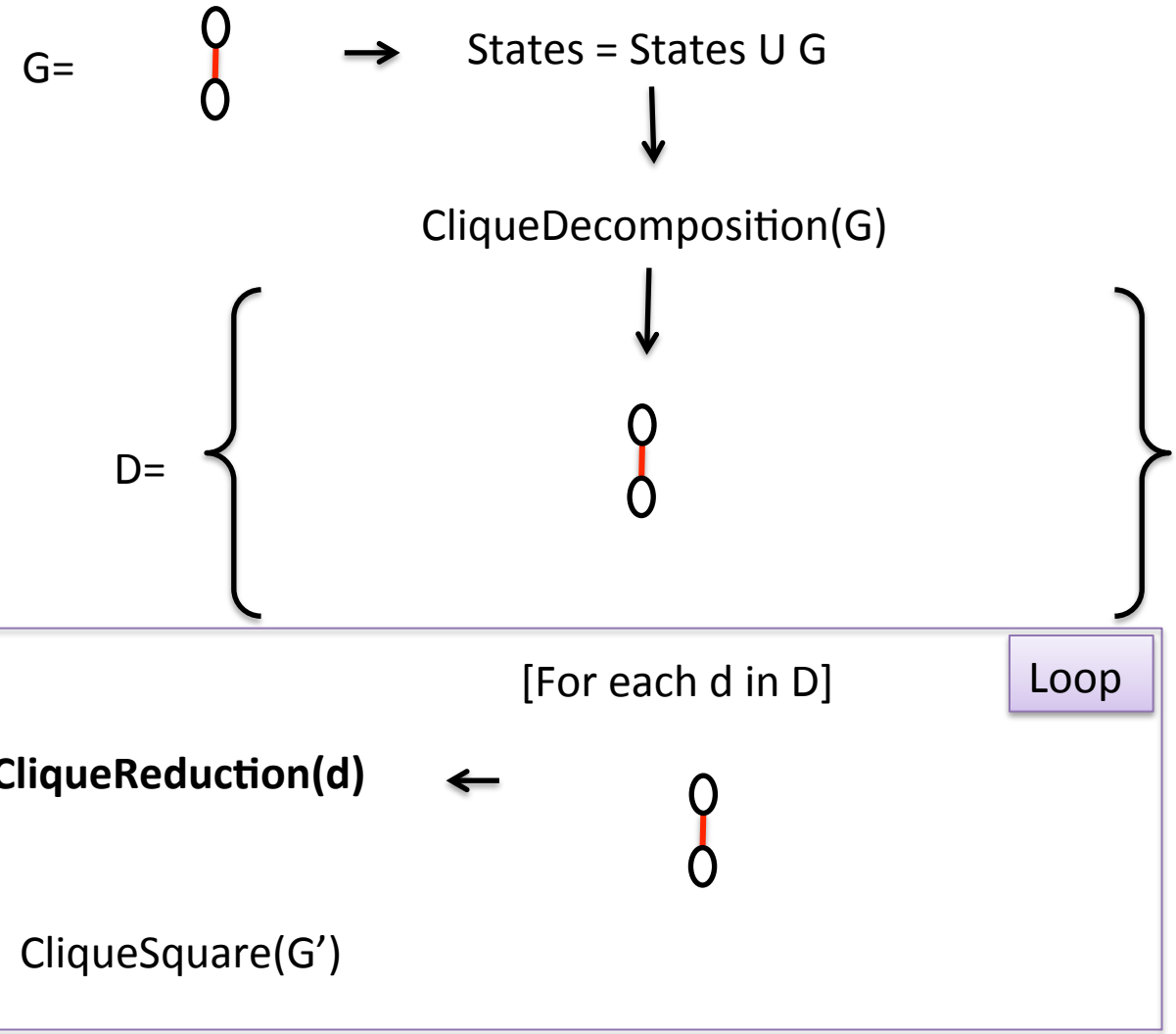

# CliqueSquare algorithm example



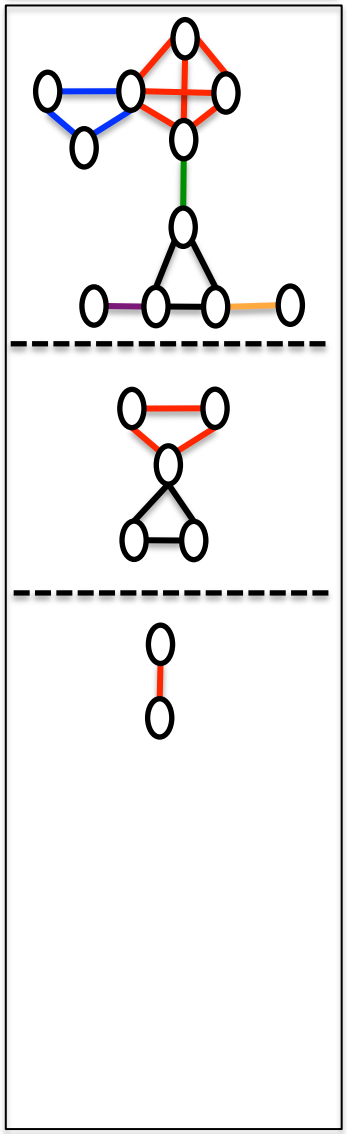



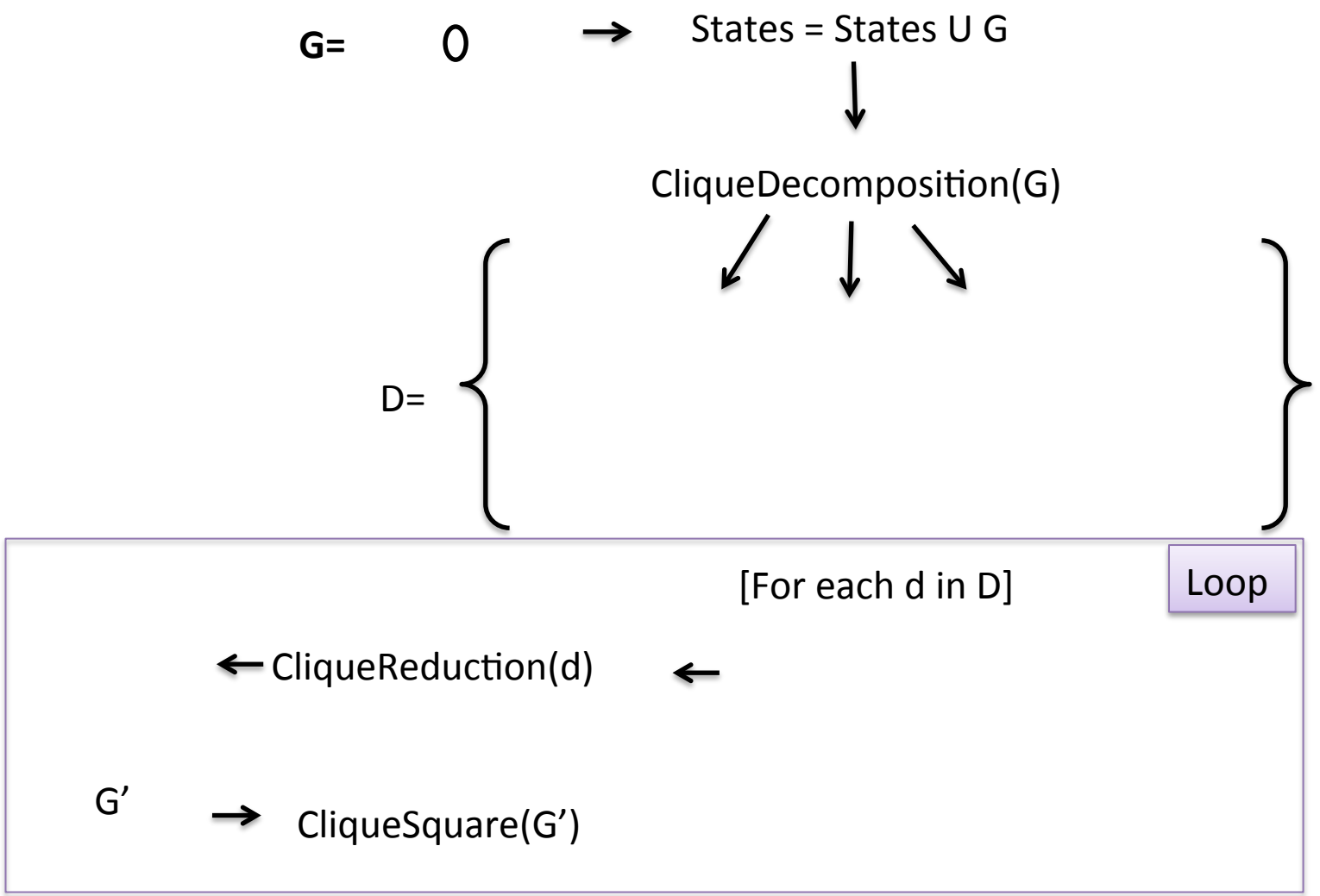
# CliqueSquare algorithm example



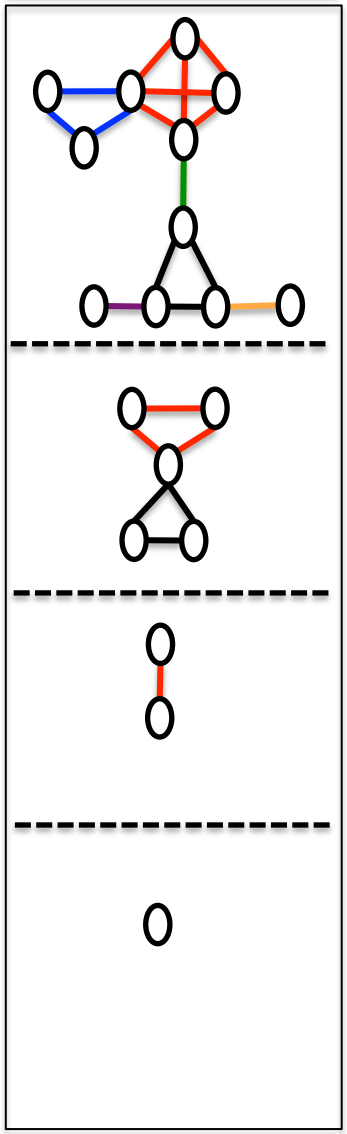



# CliqueSquare algorithm example



# CliqueSquare algorithm example




G= ∅ → **States = States U G**

CliqueDecomposition(G)

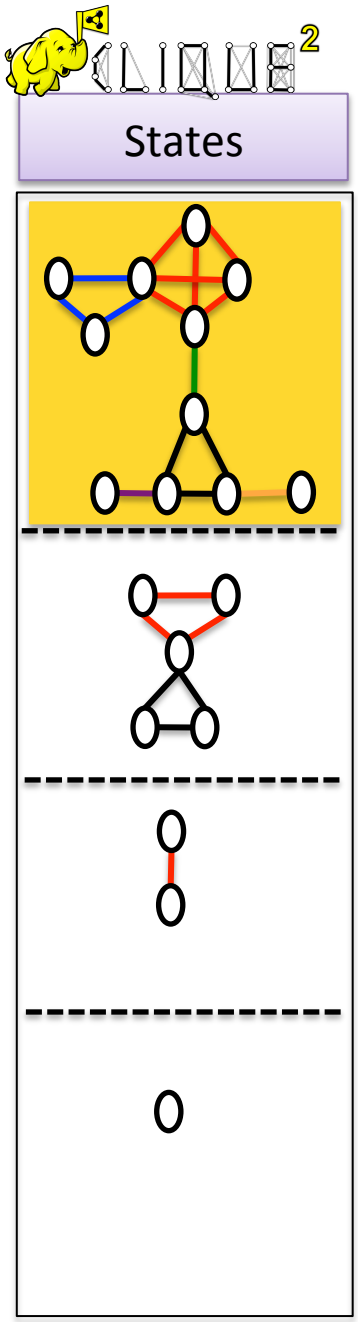
D= { }

Loop

[For each d in D]

← CliqueReduction(d) ←

G' → CliqueSquare(G')

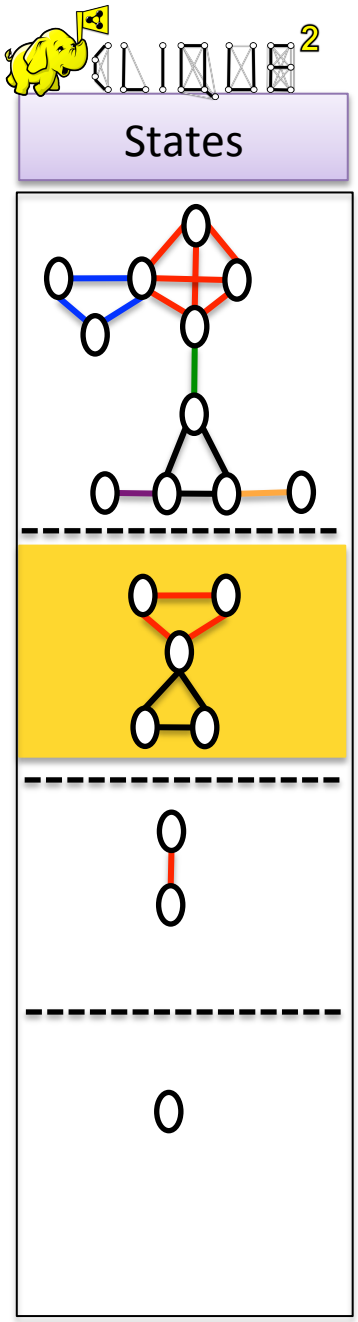



# CliqueSquare algorithm example

**CreateQueryPlan(States)**

Each node of the graph corresponds to a scan operator

**T1** **T2** **T3** **T4** **T5** **T6** **T7** **T8** **T11** **T9** **T10**

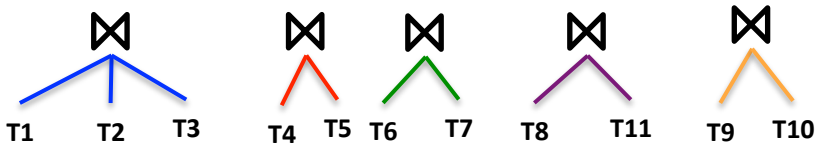


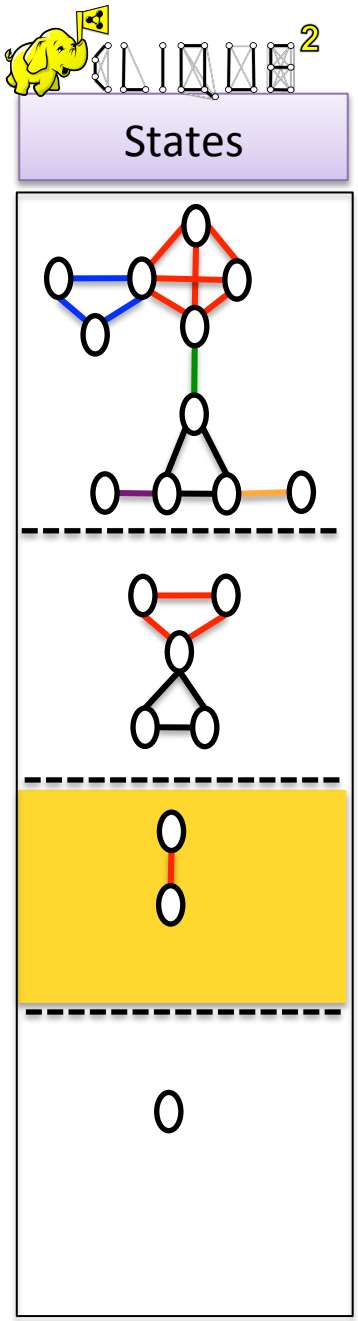

# CliqueSquare algorithm example

**CreateQueryPlan(States)**

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

Introduce the **join** operators.



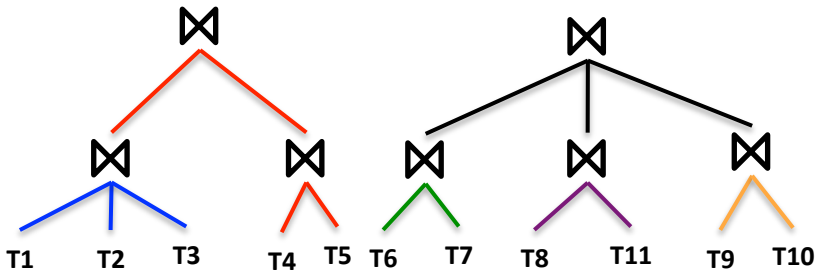


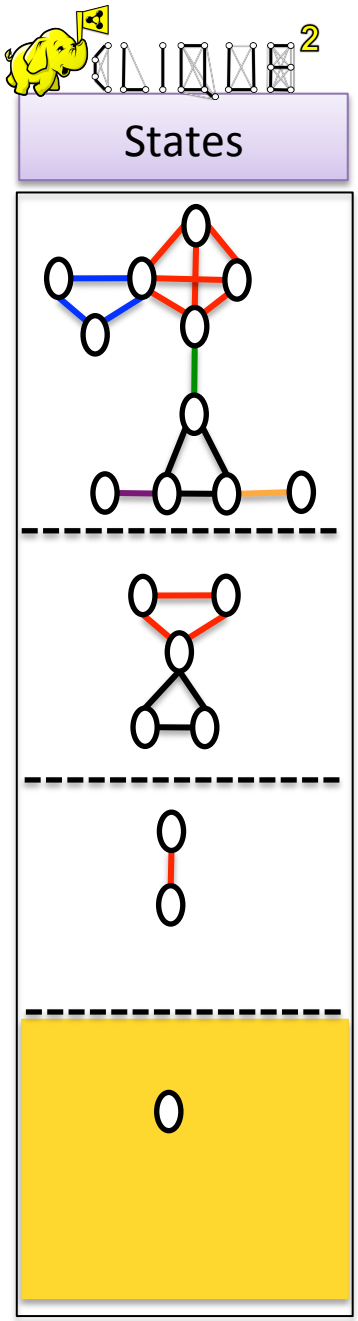

# CliqueSquare algorithm example

**CreateQueryPlan(States)**

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

Introduce the **join** operators.



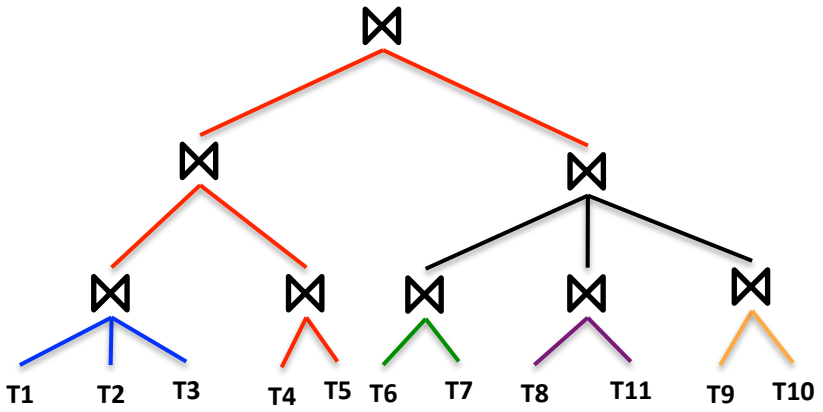



# CliqueSquare algorithm example

**CreateQueryPlan(States)**

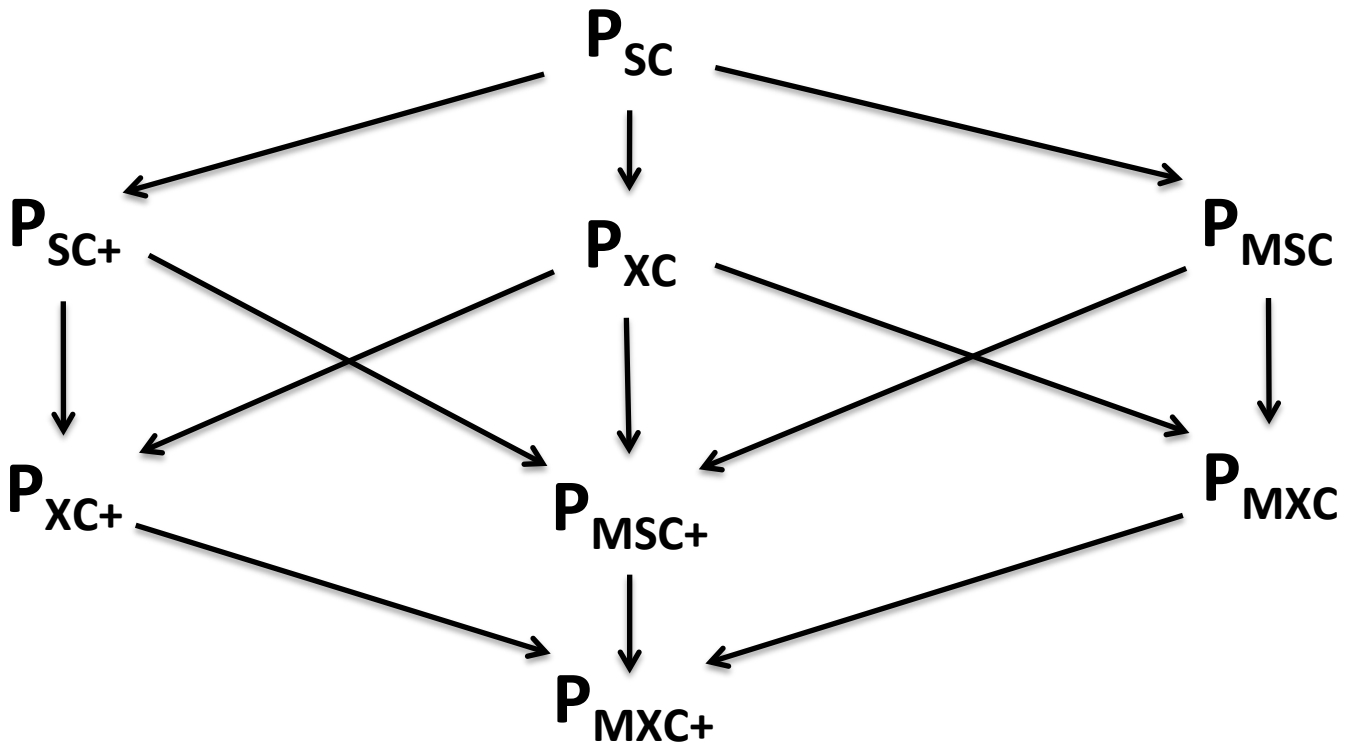
Each **node** of the graph corresponds to a **clique** of nodes of the previous graph.

Introduce the **join** operators.



# CliqueSquare algorithm – Logical plans

- The choice of the decomposition method (*m)* affects the production of plans - **Plan space** (P*m*)
- 8 CliqueSquare variants with 8 plan spaces
- Plan space **inclusion relationships**:



# CliqueSquare algorithm – Logical plans

- **Height-Optimal (Flat)** plans: having the least **height**
- Decomposition variants are categorized based on height optimality into:
  - **HO-Complete:** **all** height optimal plans for a query *q*

    **SC**
  - **HO-Partial:** at **least one** height optimal plan for a query *q*

    **MSC+** **SC+** **MSC**
  - **HO-Lossy:** possibly **no** height optimal plan for a query *q*

    **MXC** **MXC+** **XC+** **XC**

# CliqueSquare algorithm – Logical plans

- **Height-Optimal (Flat)** plans: having the least **height**
- Decomposition variants are categorized based on height optimality into:
  - **HO-Complete: all** height optimal plans for a query *q*

    **SC**
  - **HO-Partial: at least one** height optimal plan for a query *q*

    **MSC+** **SC+** **MSC**
  - **HO-Lossy:** possibly **no** height optimal plan for a query *q*

    **MXC** **MXC+** **XC+** **XC**

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

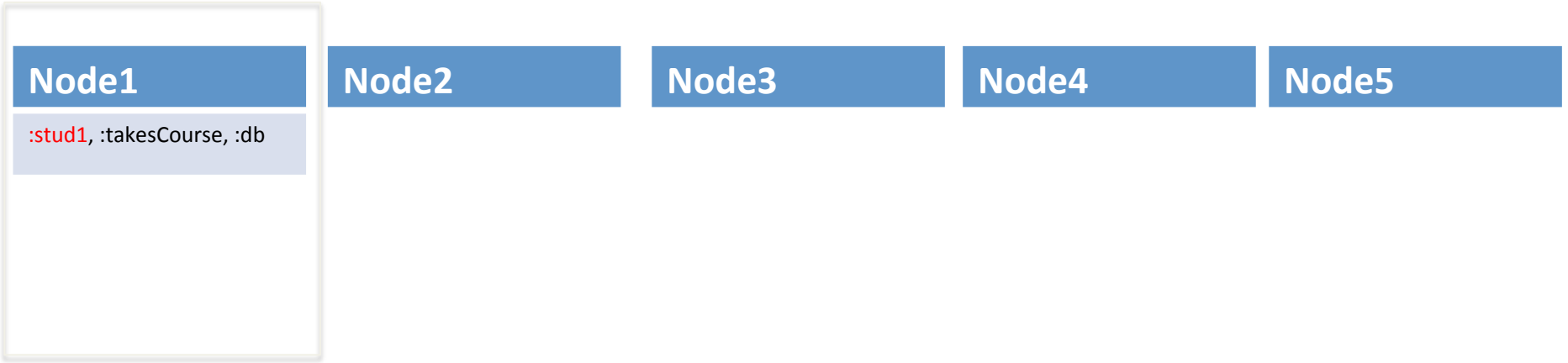
| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |

Hash triple by **subject**

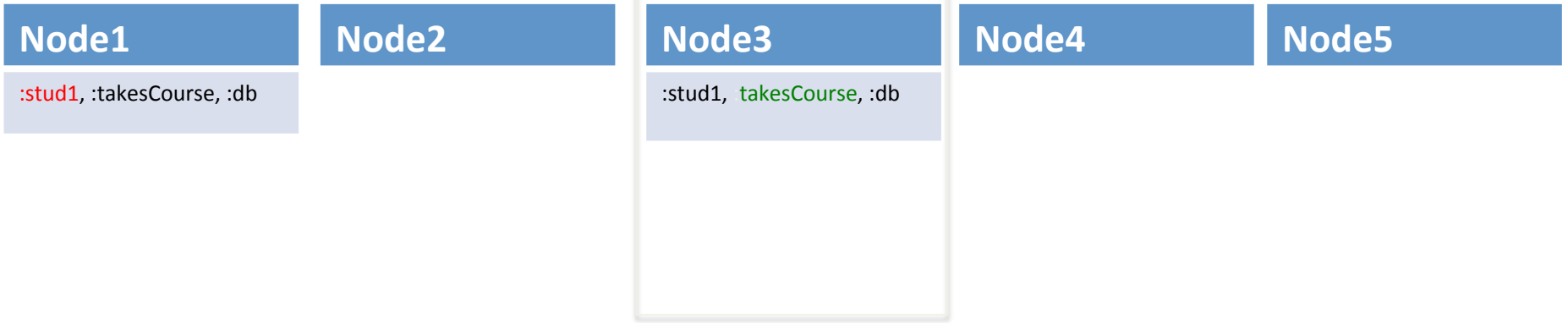


# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |

Hash triple by **property**

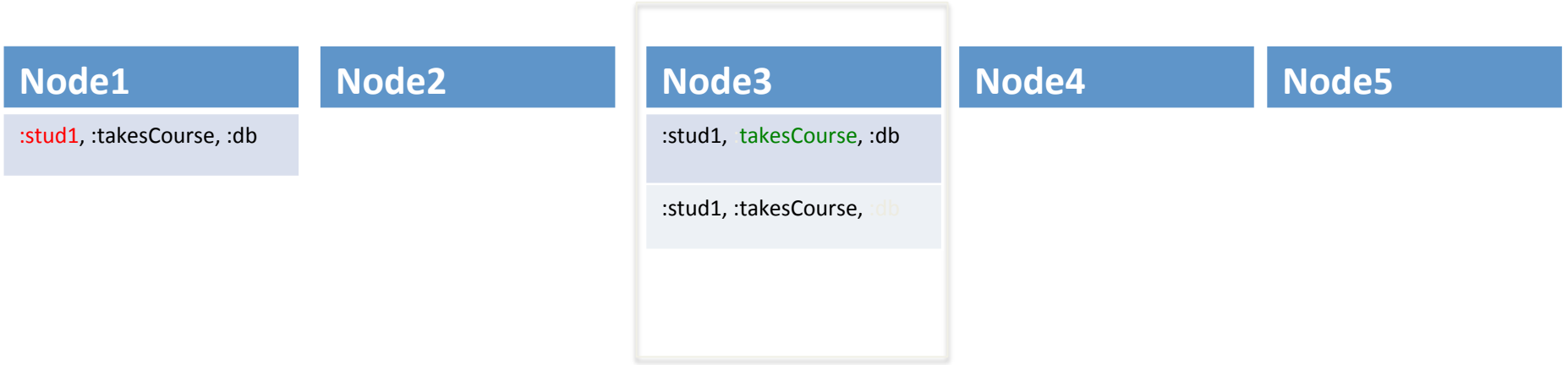


# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |

Hash triple by **object**



# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |

| Node1 | Node2 | Node3 | Node4 | Node5 |
|---|---|---|---|---|
| :stud1, :takesCourse, :db | | :stud1, takesCourse, :db | | |
| | | :stud1, :takesCourse, :db | | |

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |
| t2 | :stud2 | :takesCourse | :os |

Hash triple by **subject**

| Node1 | Node2 | Node3 | Node4 | Node5 |
|---|---|---|---|---|
| :stud1, :takesCourse, :db | :stud2, :takesCourse, :os | :stud1, takesCourse, :db | | |
| | | :stud1, :takesCourse, :db | | |

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |
| t2 | :stud2 | :takesCourse | :os |

Hash triple by **property**

| Node1 | Node2 | Node3 | Node4 | Node5 |
|---|---|---|---|---|
| :stud1, :takesCourse, :db | :stud2, :takesCourse, :os | :stud1, takesCourse, :db<br>:stud1, :takesCourse, :db<br>:stud2, :takesCourse, :os | | |

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |
| t2 | :stud2 | :takesCourse | :os |

Hash triple by **object**

| Node1 | Node2 | Node3 | Node4 | Node5 |
|---|---|---|---|---|
| :stud1, :takesCourse, :db | :stud2, :takesCourse, :os | :stud1, takesCourse, :db | :stud2, :takesCourse, :os | |
| | | :stud1, :takesCourse, :db | | |
| | | :stud2, :takesCourse, :os | | |

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |
| t2 | :stud2 | :takesCourse | :os |
| t3 | :prof1 | :advisor | :stud1 |
| t4 | :prof2 | :advisor | :stud2 |

| Node1 | Node2 | Node3 | Node4 | Node5 |
|---|---|---|---|---|
| :stud1, :takesCourse, :db | :stud2, :takesCourse, :os | :stud1, takesCourse, :db | :stud2, :takesCourse, :os | |
| | | :stud1, :takesCourse, :db | | |
| | | :stud2, :takesCourse, :os | | |

# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**

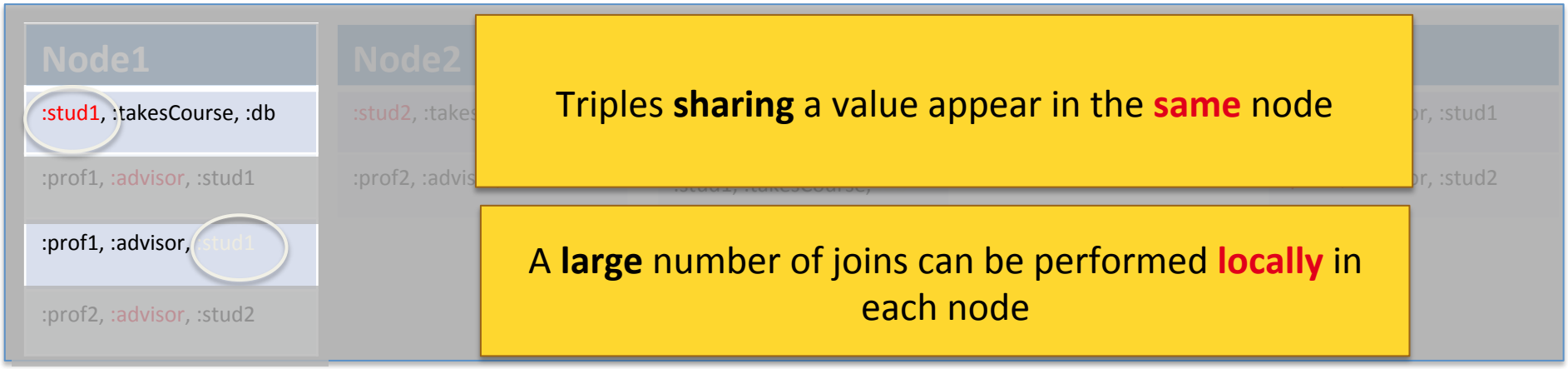
| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |
| t2 | :stud2 | :takesCourse | :os |
| t3 | :prof1 | :advisor | :stud1 |
| t4 | :prof2 | :advisor | :stud2 |

| Node1 | Node2 | Node3 | Node4 | Node5 |
|---|---|---|---|---|
| :stud1, :takesCourse, :db | :stud2, :takesCourse, :os | :stud1, takesCourse, :db | :stud2, :takesCourse, :os | prof1, :advisor, :stud1 |
| :prof1, :advisor, :stud1 | :prof2, :advisor, :stud2 | :stud1, :takesCourse, :db | | :prof2, :advisor, :stud2 |
| :prof1, :advisor, :stud1 | | :stud2, :takesCourse, :os | | |
| :prof2, :advisor, :stud2 | | | | |

# Data organization within CliqueSquare

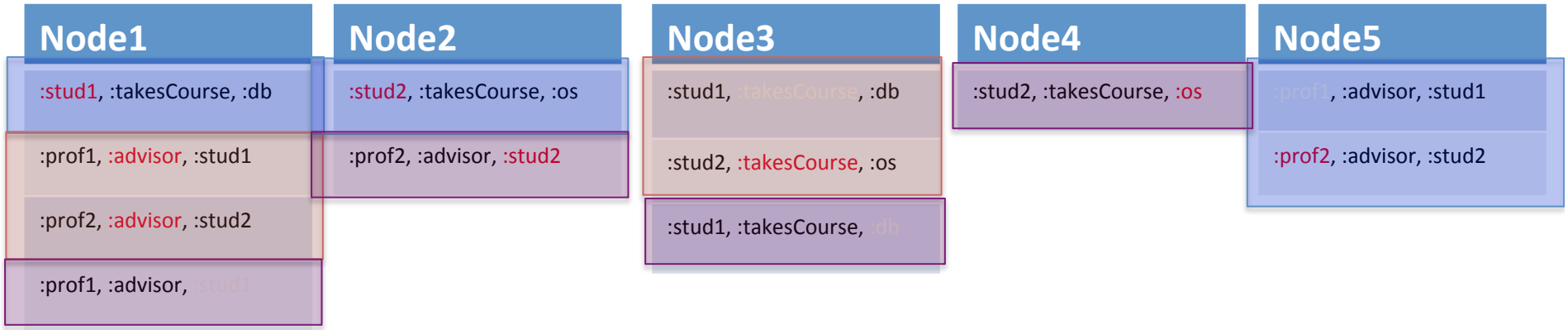
1. Each triple is partitioned based on its **subject**, **property** & **object**

| tid | Subject | Property | Object |
|---|---|---|---|
| t1 | :stud1 | :takesCourse | :db |
| t2 | :stud2 | :takesCourse | :os |
| t3 | :prof1 | :advisor | :stud1 |
| t4 | :prof2 | :advisor | :stud2 |



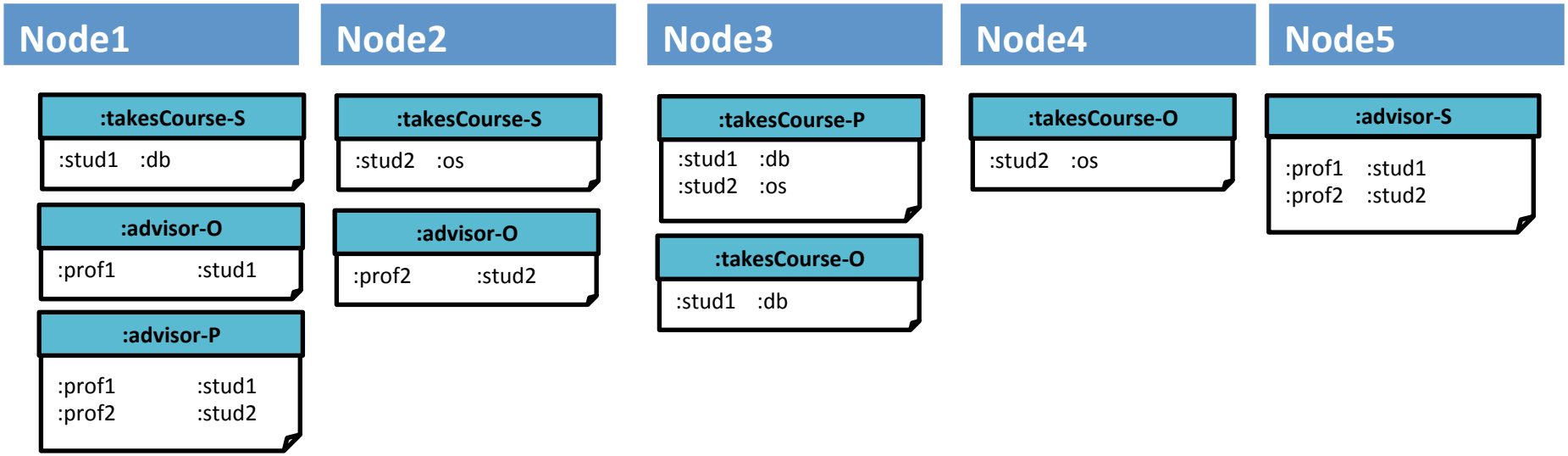
# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**
2. The triples are grouped into **3** partitions (**subject**, **property**, **object**) based on the **partition attribute**



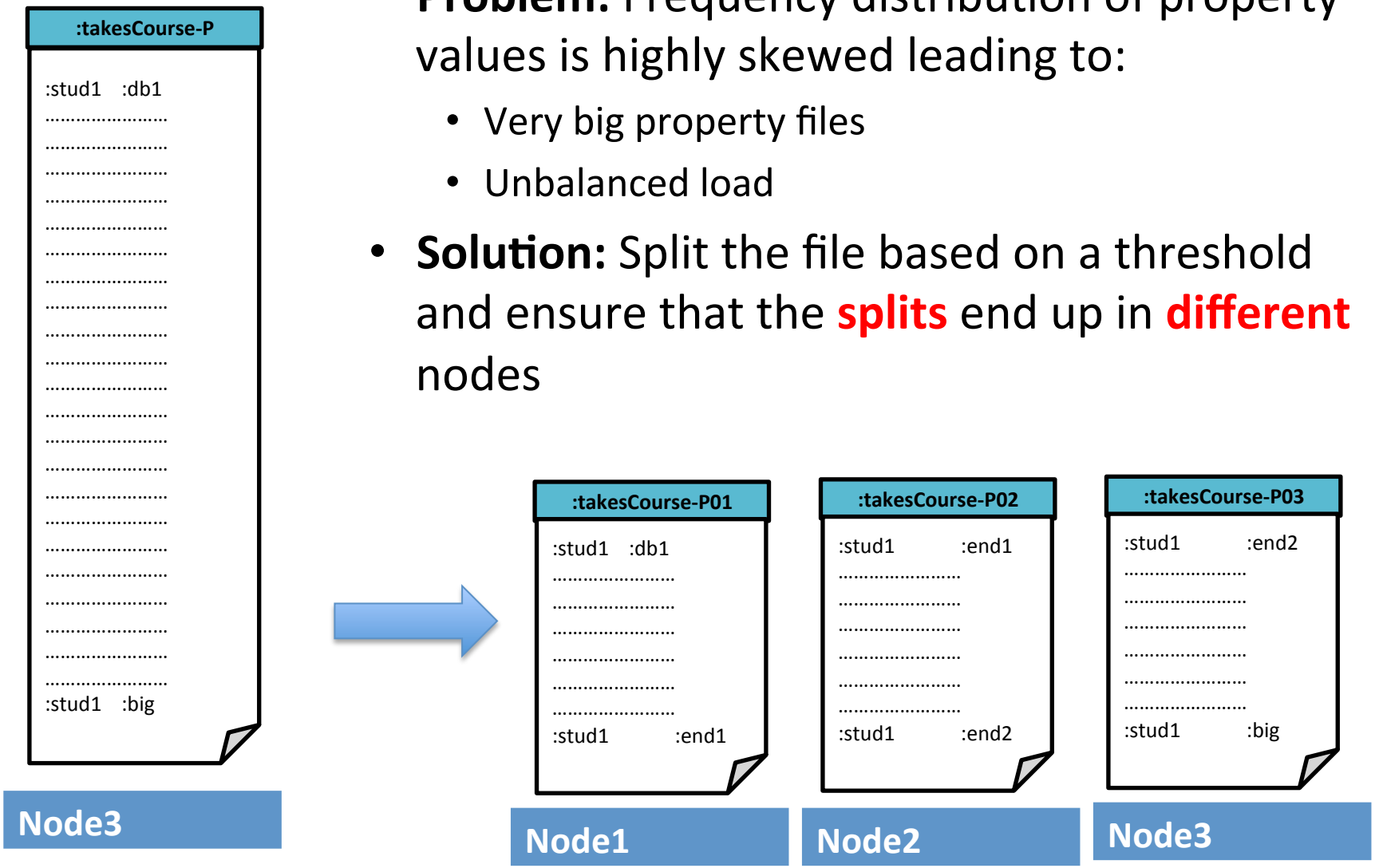
# Data organization within CliqueSquare

1. Each triple is partitioned based on its **subject**, **property** & **object**
2. The triples are grouped into **3** partitions (subject, property, object) based on the **partition attribute**
3. The triples inside **each** partition are grouped by **property** and stored on the filesystem

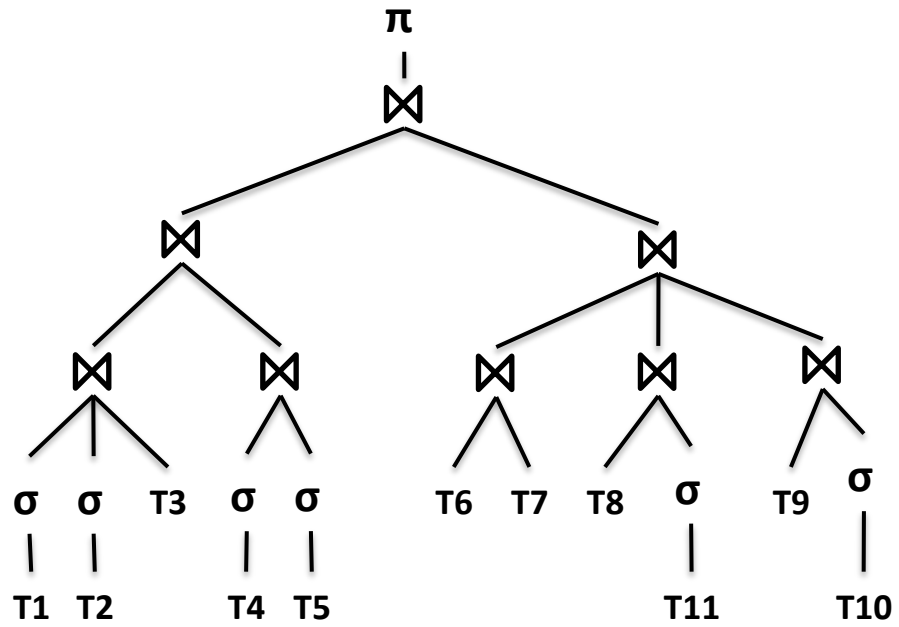


# Handling property skew

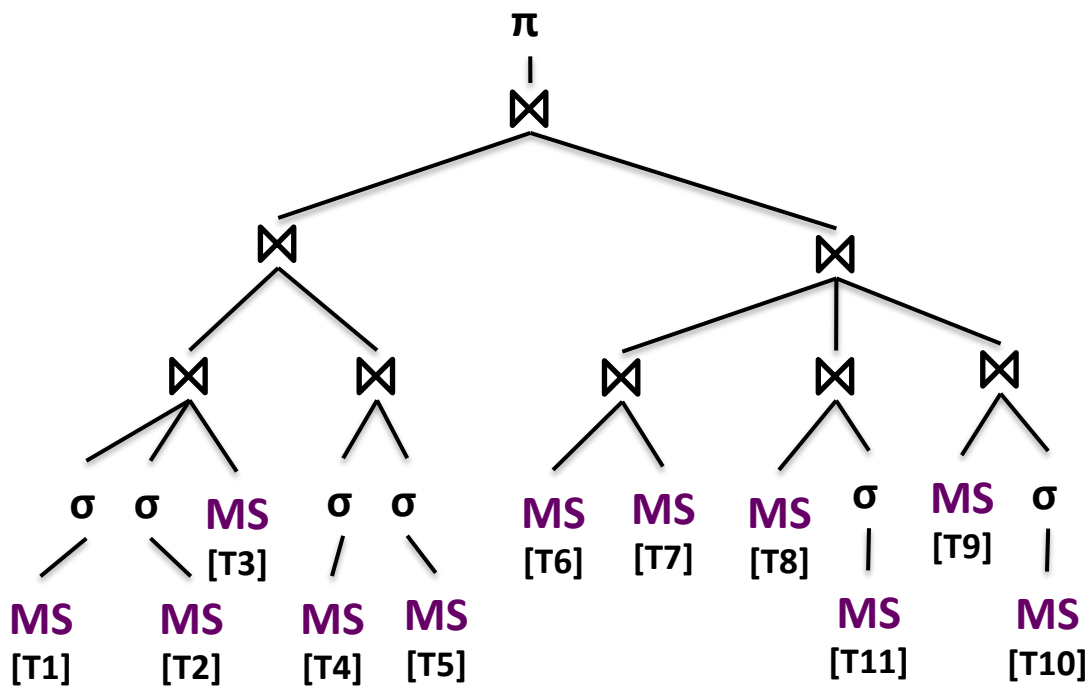
- **Problem:** Frequency distribution of property values is highly skewed leading to:
  - Very big property files
  - Unbalanced load
- **Solution:** Split the file based on a threshold and ensure that the **splits** end up in **different** nodes



# From logical plan to physical plans

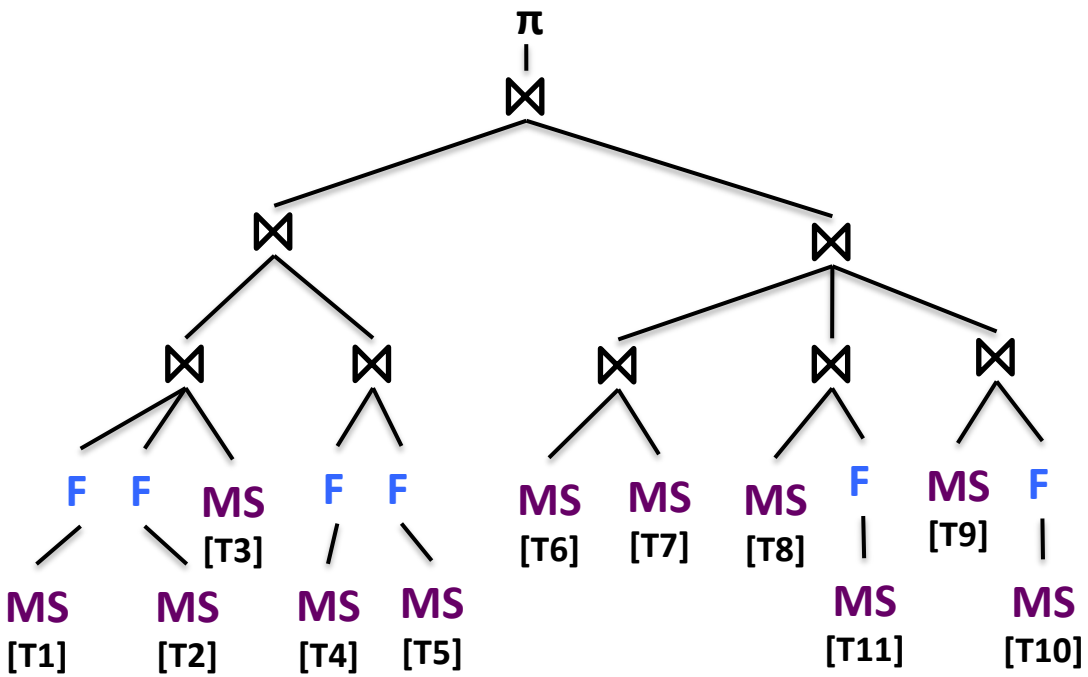


# Logical plan → physical plan



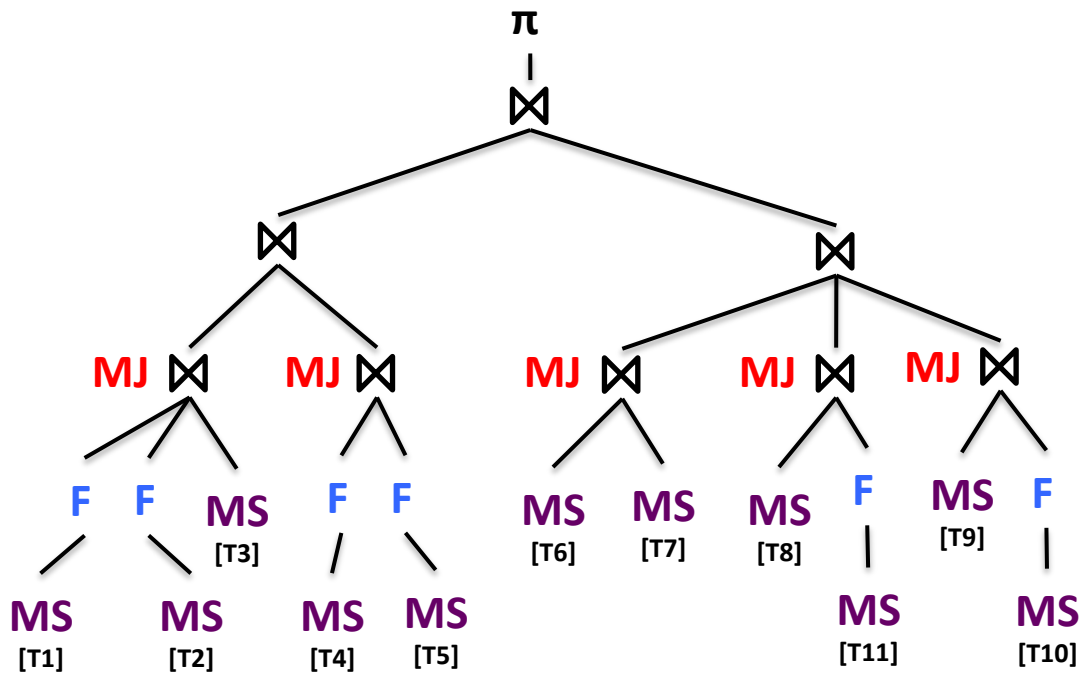

- Reading the triples from HDFS requires a Map Scan (**MS**) operator

# Logical plan → Physical plan



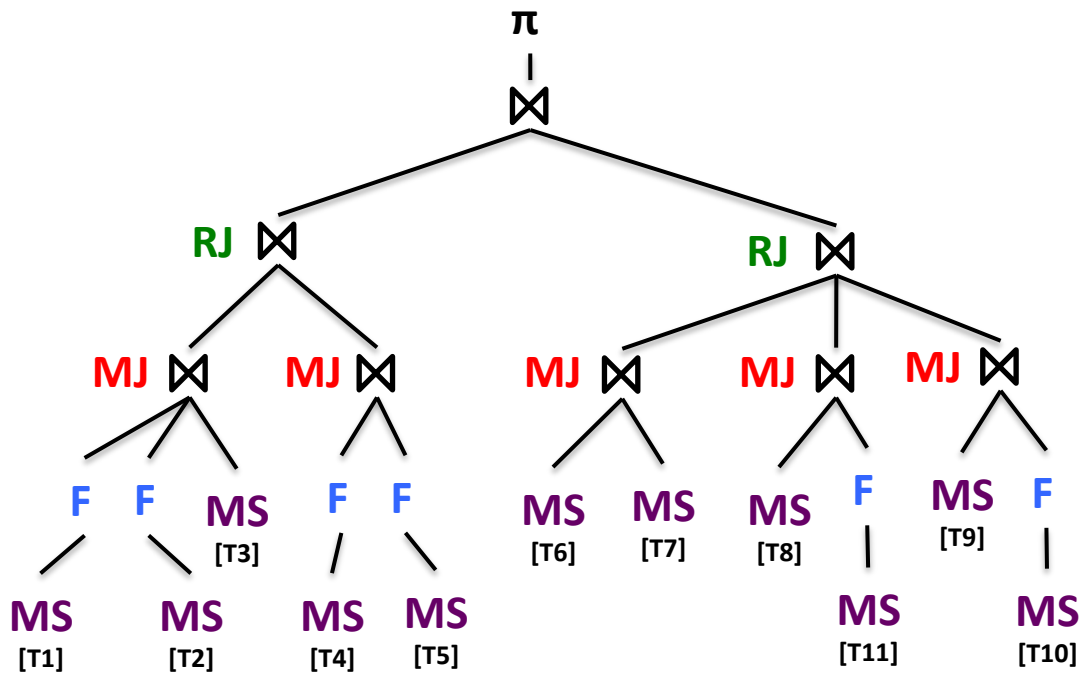

- Logical selections (**σ**) are translated to physical selections (**F**)

# Logical plan → Physical plan



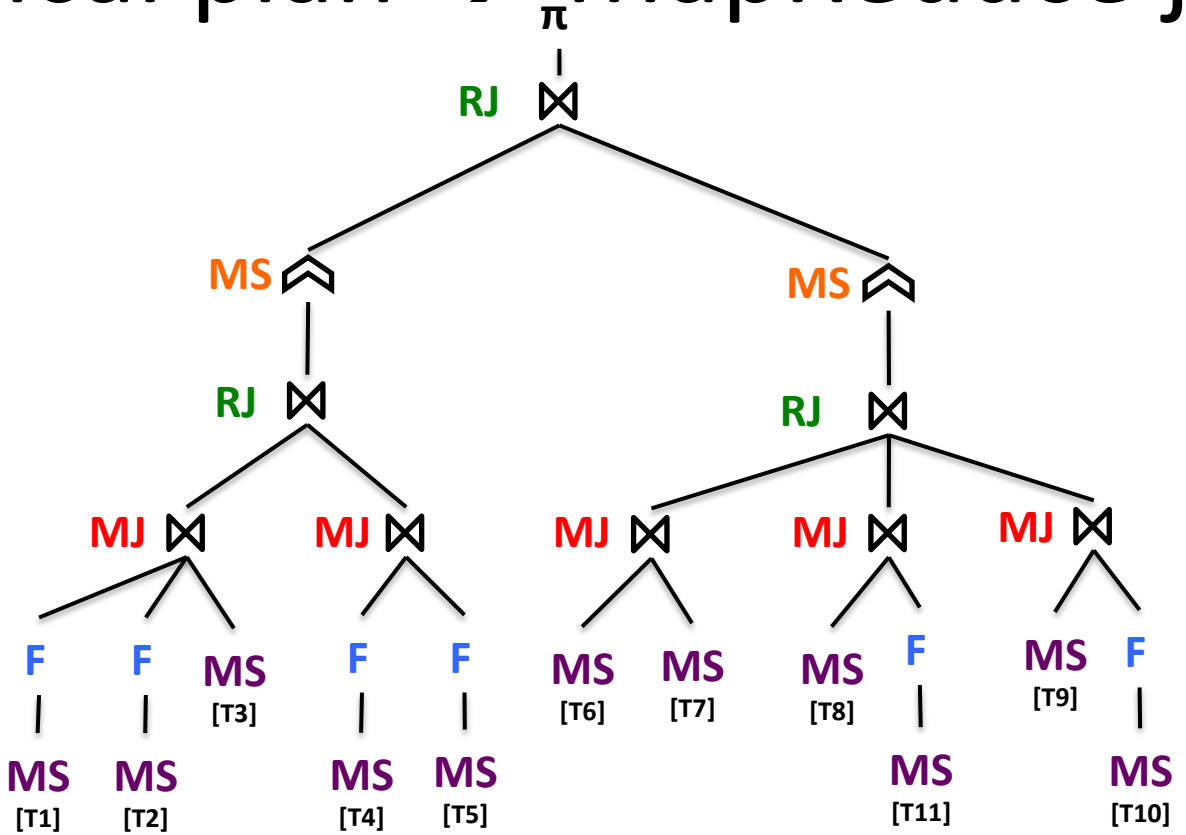

➢ First level joins are translated to Map side joins (**MJ**) taking advantage of the **data partitioning**

# Logical plan → Physical plan



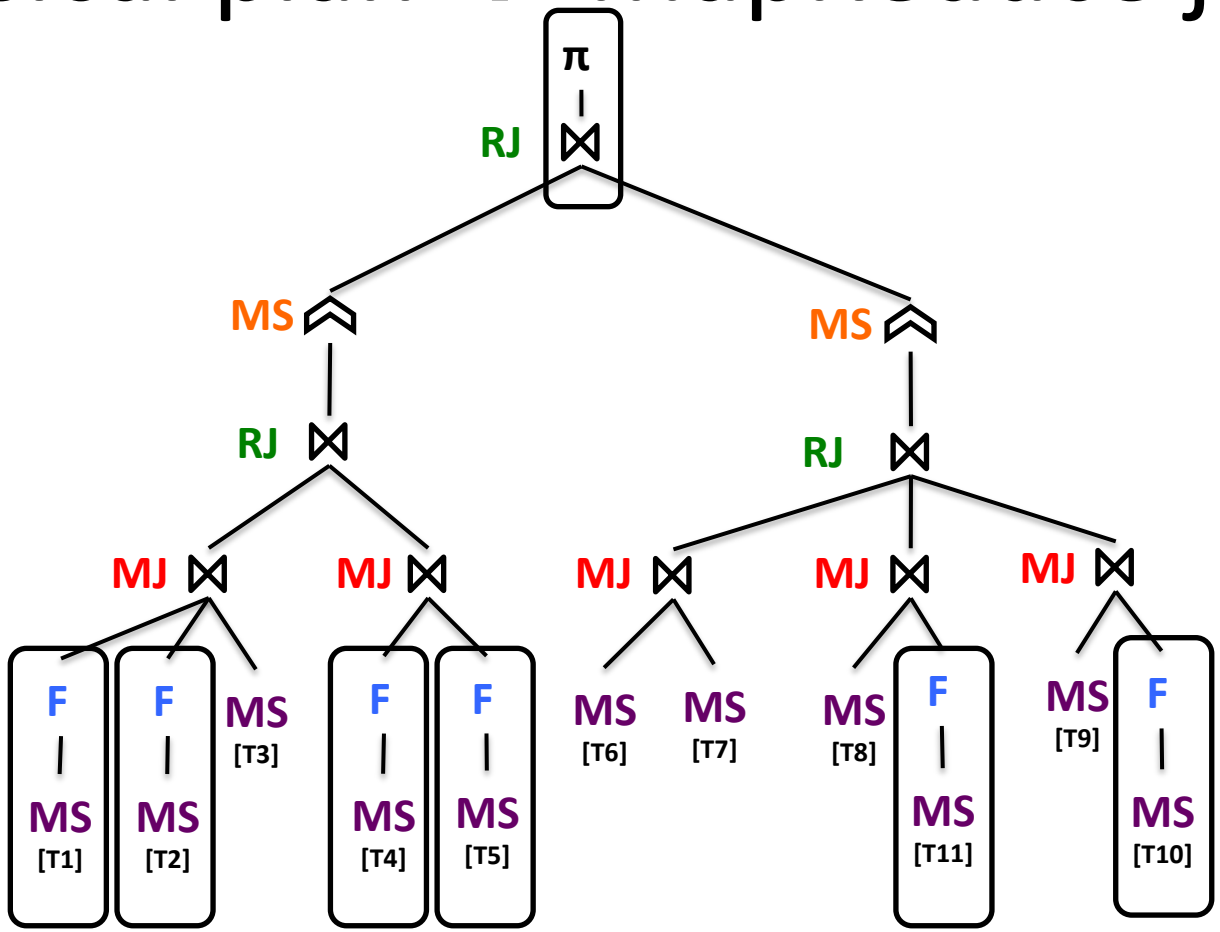

- All subsequent joins are translated to Reduce side joins (**RJ**)

# Physical plan → MapReduce jobs



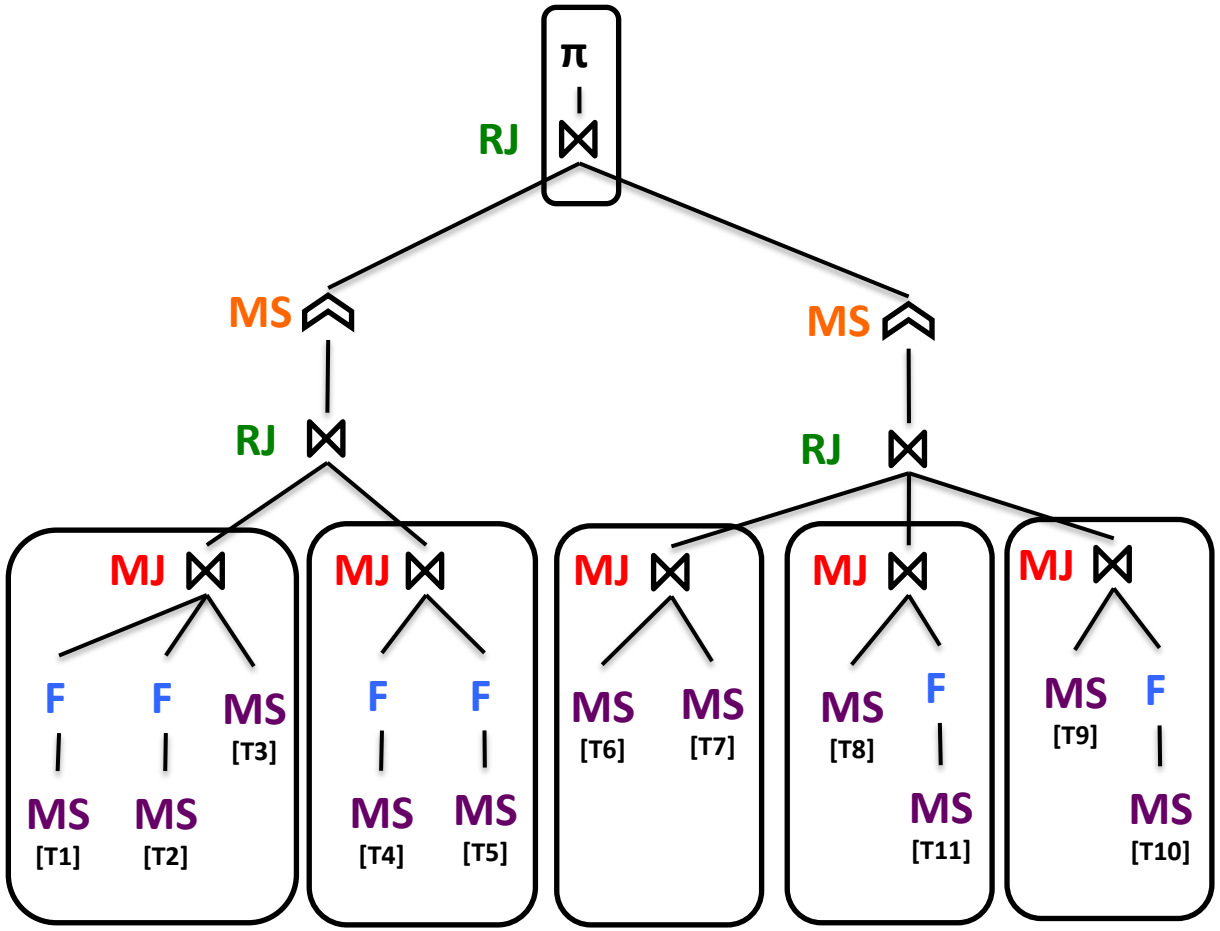

- Group the physical operators into Map/Reduce **tasks** and **jobs**

# Physical plan → MapReduce jobs



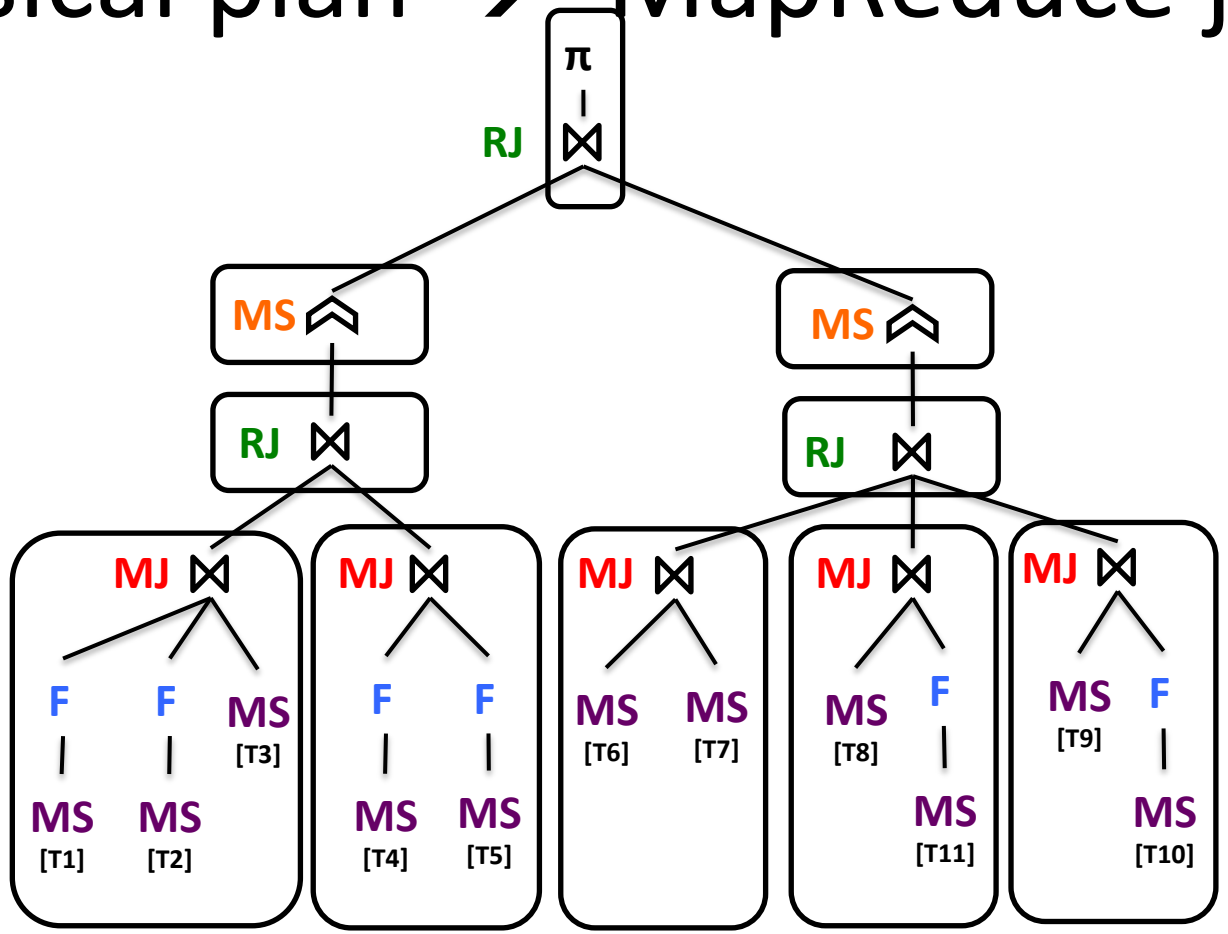

- Selections (F) and projections (π) belong to the same task as their child operator

# Physical plan → MapReduce jobs



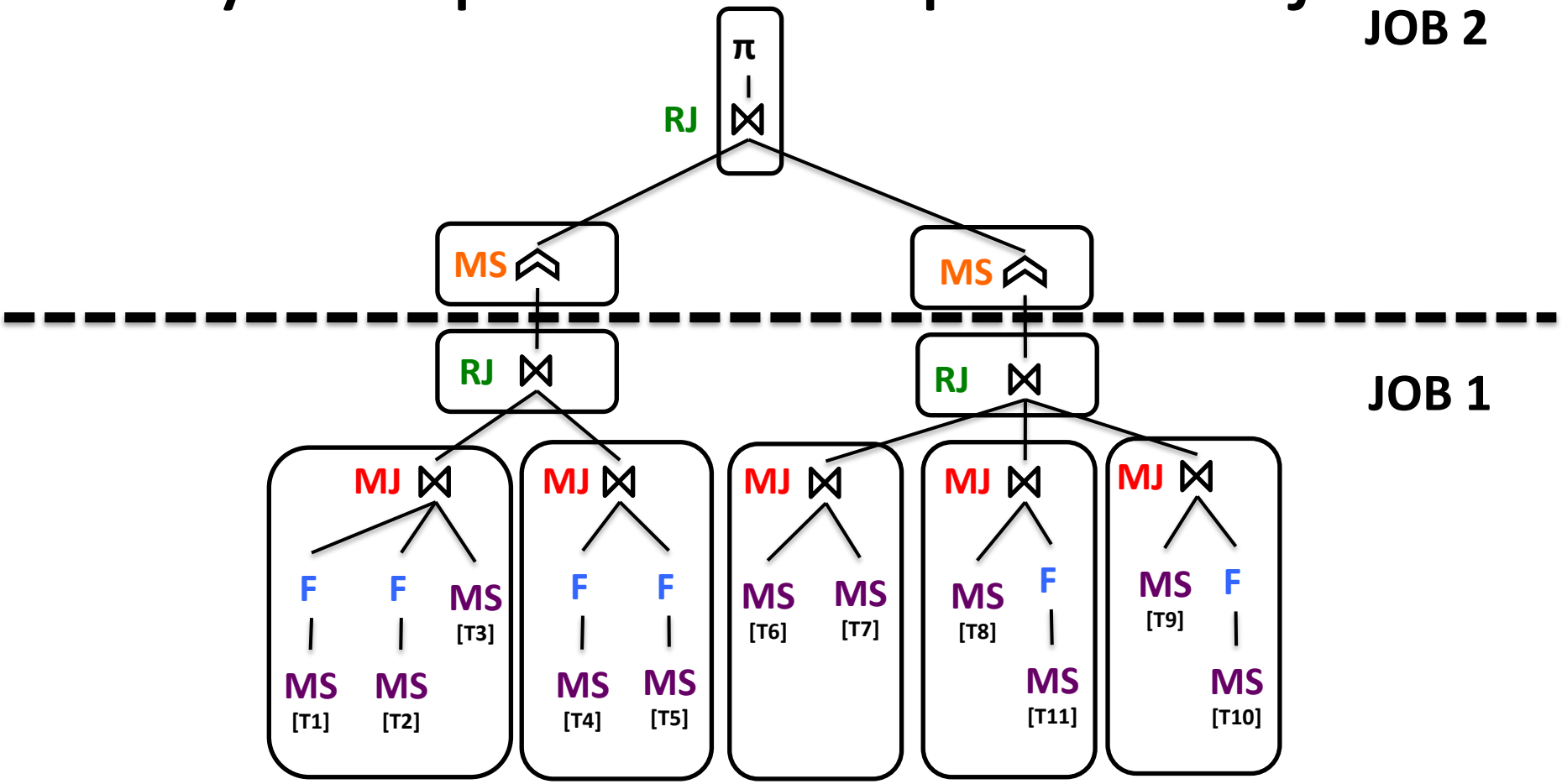

- Map joins (**MJ**) along with all their descendants are executed in the same task

# Physical plan → MapReduce jobs



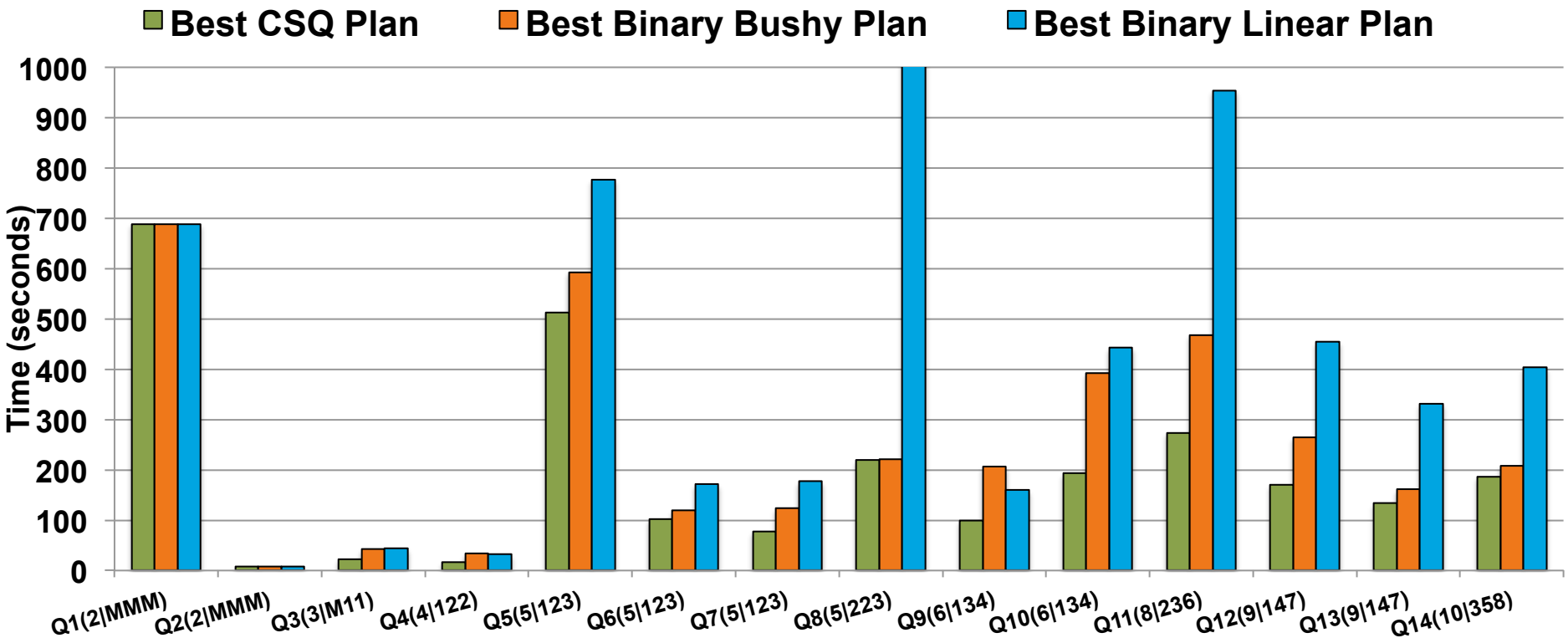

- Any other operator (**RJ** or **MS**) is executed in a separate task

# Physical plan → MapReduce jobs



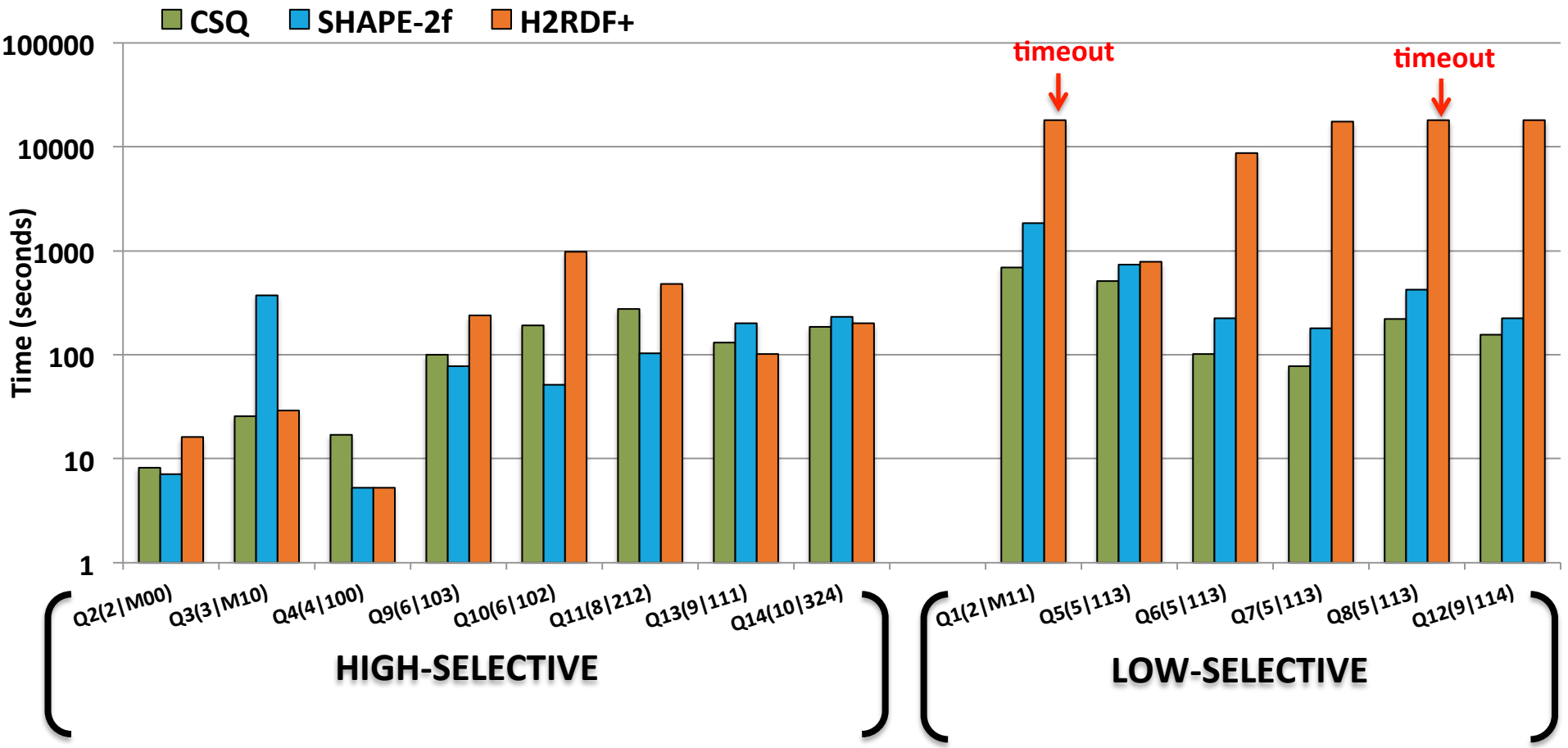

- **Tasks** are grouped into **jobs** in a bottom-up traversal

# Experiments – Plan comparison




- ✓ **CSQ flat** plans **faster** than bushy plans (up to X2)
- ✓ **CSQ flat** plans **faster** than linear plans (up to X16)
- ✓ **CSQ flat** plans require less number of jobs

# Experiments – System comparison




✓**CSQ** outperforms both systems for non-selective queries

✓**CSQ** performs closely for selective queries

# 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, 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
- [JQD11] A.Jindal, J.-A.Quiané-Ruiz,and J.Dittrich. "*Trojan Data Layouts: Right Shoes for a Running Elephant*" SOCC, 2011