ARCHITECTURES FOR BIG DATA MANAGEMENT (INCL. CLOUD)

Ioana Manolescu
Inria and Ecole polytechnique
https://pages.saclay.inria.fr/ioana.manolescu
Ioana.manolescu@inria.fr

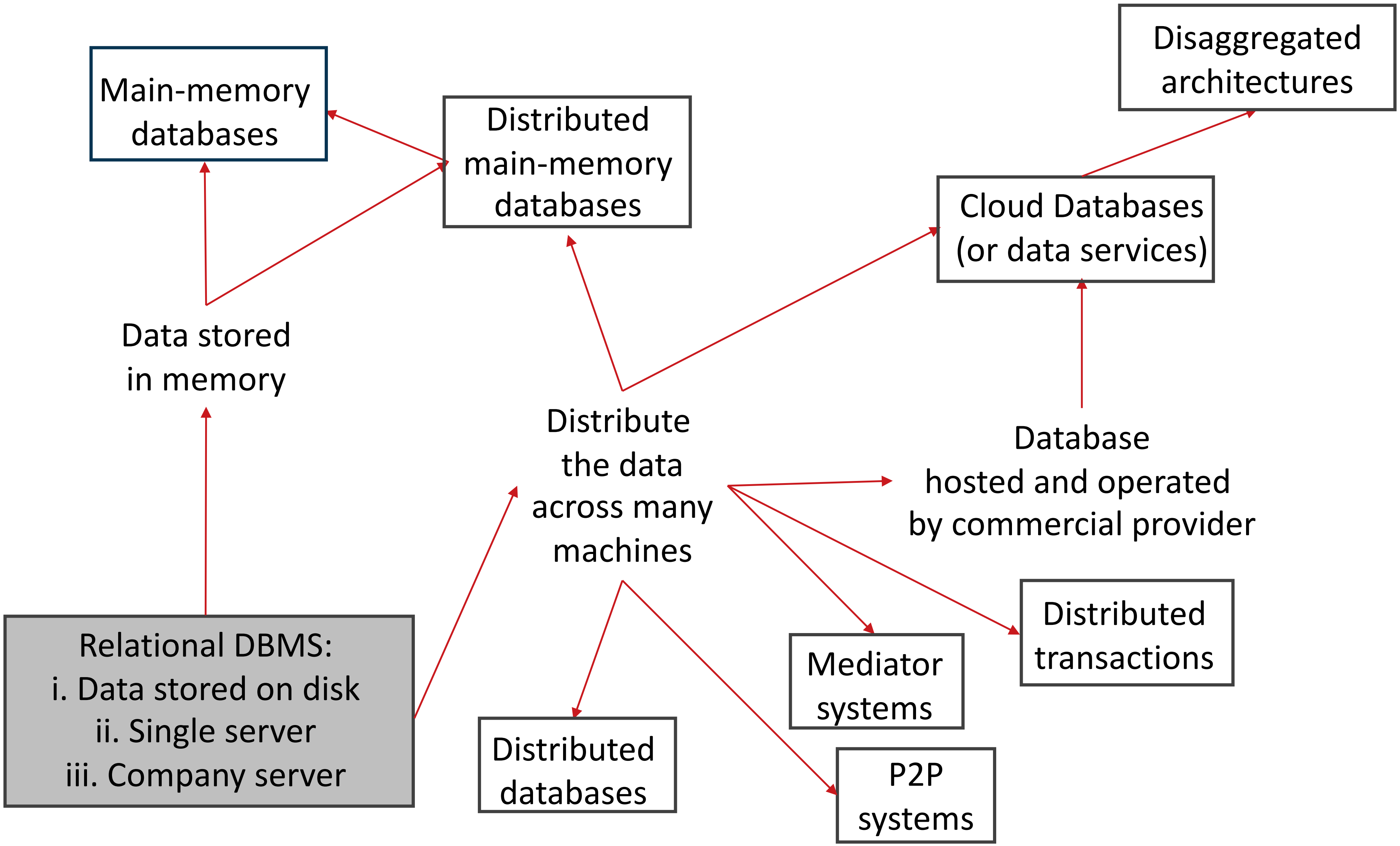
Plan

1. Distributed data management: main architectures

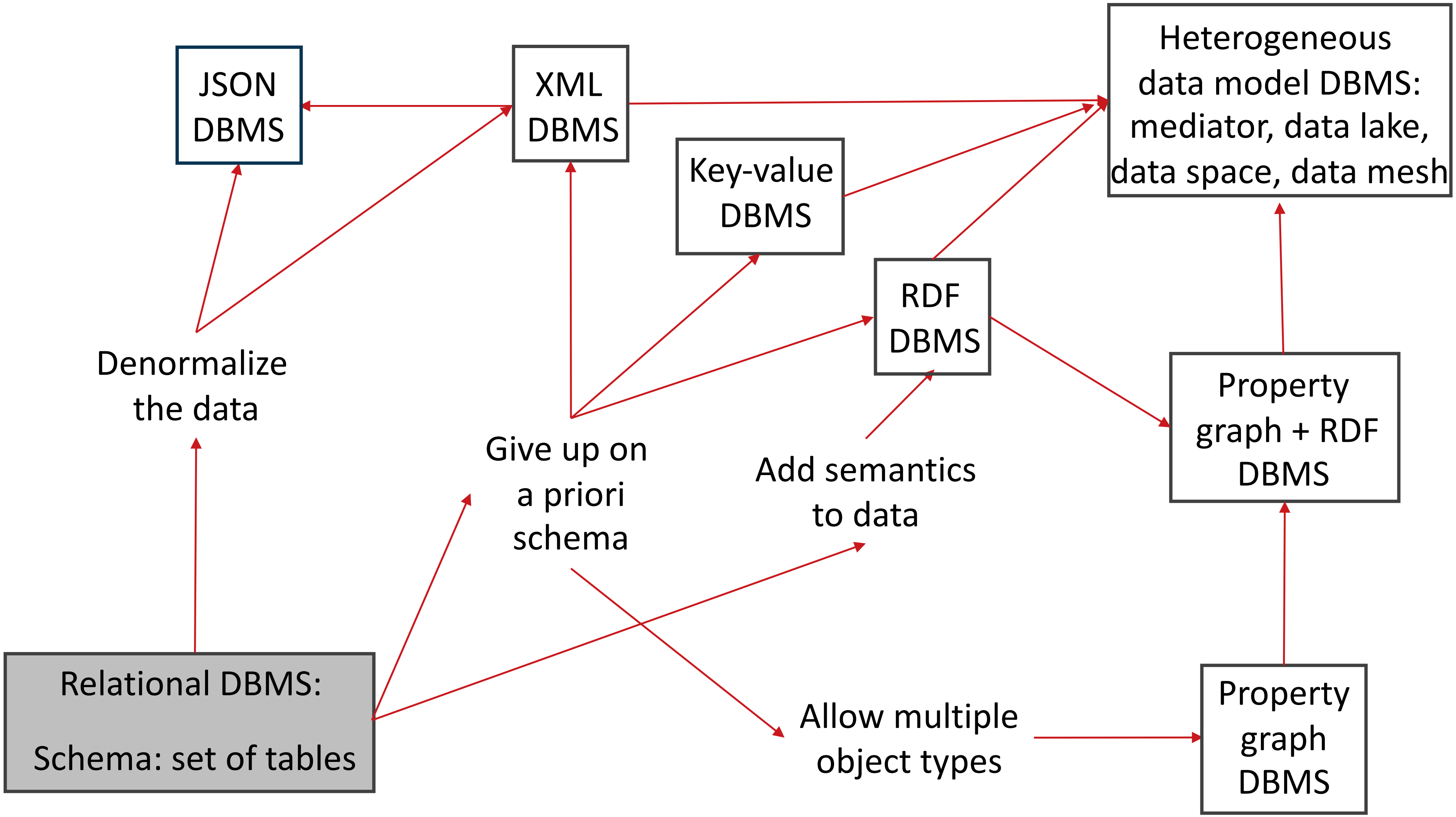
2. Cloud computing

3. Data management in the cloud (including graphs)

From a database to Big Data systems: architectures



From a database to Big Data systems: data models



Dimensions of Big Data architectures

Data model(s)

- Relations, trees (XML, JSON), graphs (RDF, PGs), nested relations

Heterogeneity (Data Model, Query Language):

- None, some, a lot

Hardware:

- Hardware type: from disk to memory
- Scale of distribution: small (~10-20 sites) or large (~10.000 sites)

Data distribution and replication

- What are the logical relations between distributed data collections?

Interoperability and control:

- Who decides: data structure, data publication, data placement
- Who does what when processing queries or updates

Distributed data management architectures

Fundamental operations: Distribution and replication

Distribution: splitting a dataset, e.g., a database, or a relation, among two or more distributed nodes

To *scale up* across more hardware

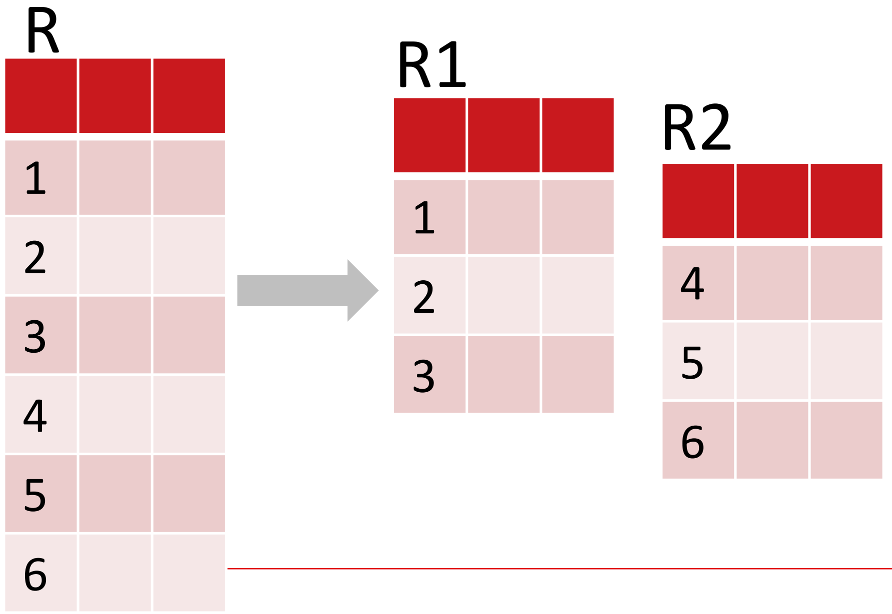
To *parallelize* computations

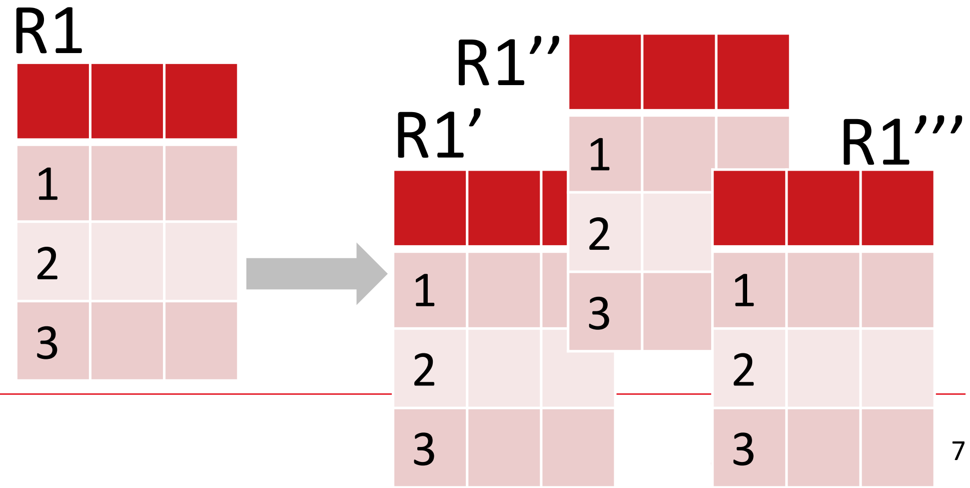
Replication: copying a dataset, e.g., a database, or a relation on one or more sites.

To ensure *durability* even in the face of hardware (storage) destruction

To increase *availability* during a software crash at one site

To increase *speed* for queries that run on a replica which is close to the query





Big Data management architectures

1. Distributed databases (since 1970)
2. **Data warehouses** (since 1970)
3. Data integration systems (since 1990s)
4. Peer-to-peer databases (since 2000)
5. **Data lakes** (since 2010), lakehouses (since 2020s)
6. Data mesh (since 2020s)
7. **Cloud databases** (since 2010s)

Distributed databases

Oldest distributed architecture ('70s): IBM System R*

Illustrate/introduce the main priciples

Data is relational (tables).

Data is distributed among many *nodes* (*sites*, *peers*...)

Data catalog: information on which data is stored where

- Catalog stored at a master/central server.
- E.g., « Paris sales are stored in Paris », « Lyon sales are stored in Paris », « Client data is stored in London », etc.

Queries are distributed (may come from any site)

First analyzed through catalog

Query processing is distributed

Operators may run on different sites → network transfer

Traditional distributed relational databases (since 1970)

Server **DB1@site1**: R1(a,b), S1(a,c)

Server **DB2@site2**: R2(a,b), S2(a,c),

Server **DB3@site3**: R3(a,b), S3(a,c) defined as:

```
select * from DB1.S1 union all
    select * from DB2.S2 union all
    select R1.a as a, R2.b as c
 from DB1.R1 r1, DB2.R2 r2
    where r1.a=r2.a
```

DB3@site3 decides what to import from site1, site2 (« hard links »)

Site1, site2 are independent servers

Query evaluation in distributed databases: query unfolding

DB1: **R1(a,b), S1(a,c)**

DB2: **R2(a,b), S2(a,c)**

DB3: **R3(a,b), S3(a,c)** defined as:

```
select * from S1 union all
select * from S2 union all
select r1.a as a, r2.b as c
from DB1.R1 r1, DB2.R2 r2
where r1.a=r2.a
```

Query on DB3:

```
select a
from S3
where a = 3;
```

The query is formulated on S3, but there is no actual data there!

- The query is reformulated (or unfolded) based on the definition of S3

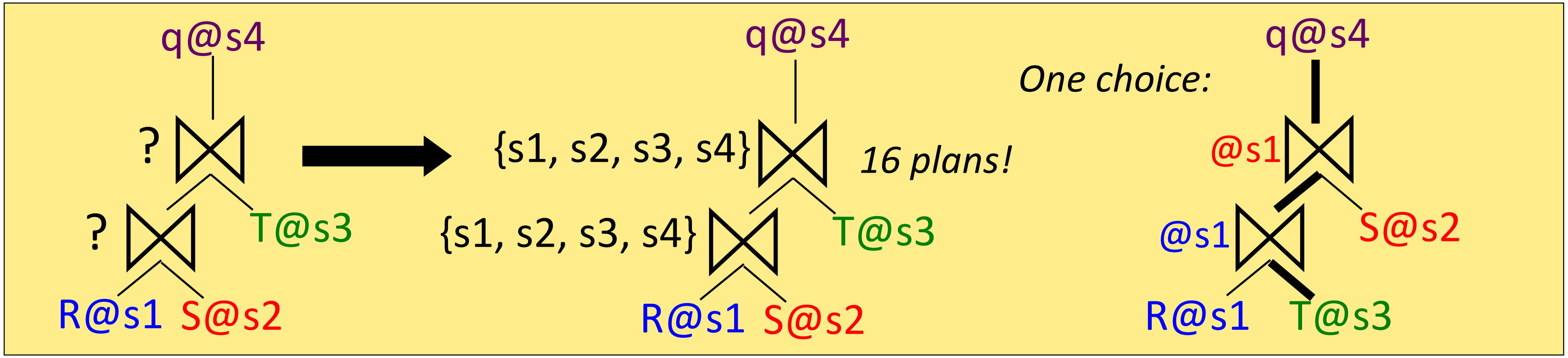
In classical DBMSs, a query over a view is also unfolded (demo)

How is a query unfolded?

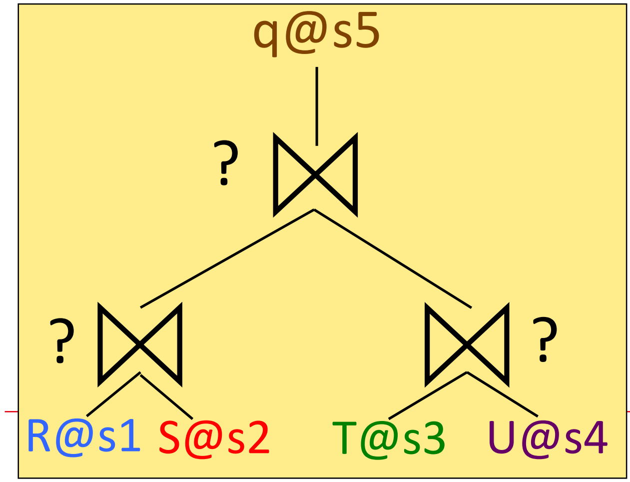
Based on its logical algebra translation

Distributed query optimization

Example 1: R@s1, S@s2, T@s3, q@s4



Example 2: R@s1, S@s2, T@s3, U@s4, q@s5

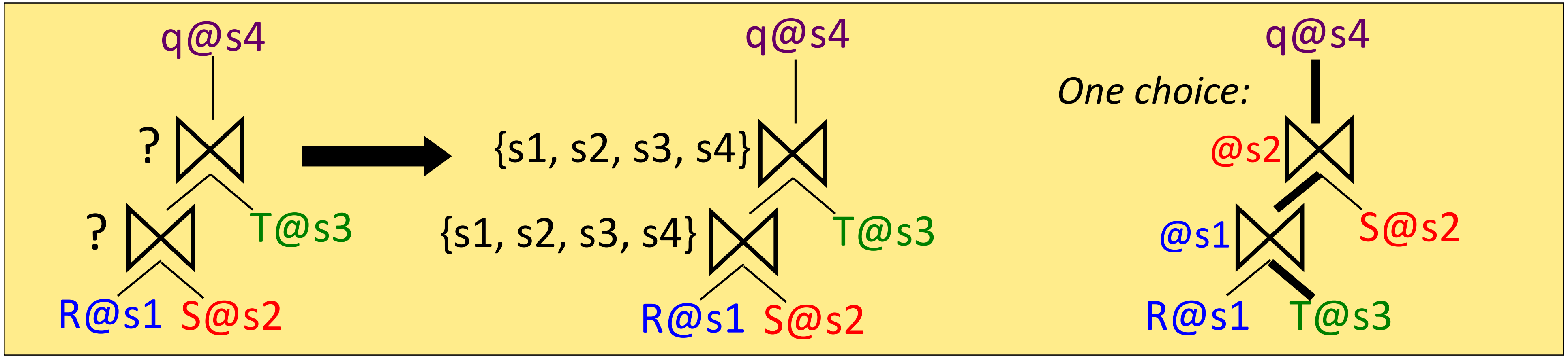


Plan pruning criteria if all the sites and network connections have equal performance:

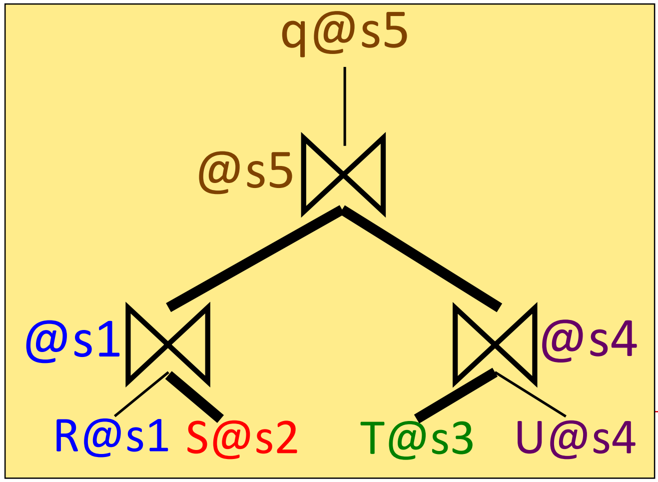
- *Ship the <u>smaller</u> collection*

Distributed query optimization

Example 1: R@s1, S@s2, T@s3, q@s4



Example 2: R@s1, S@s2, T@s3, U@s4, q@s5

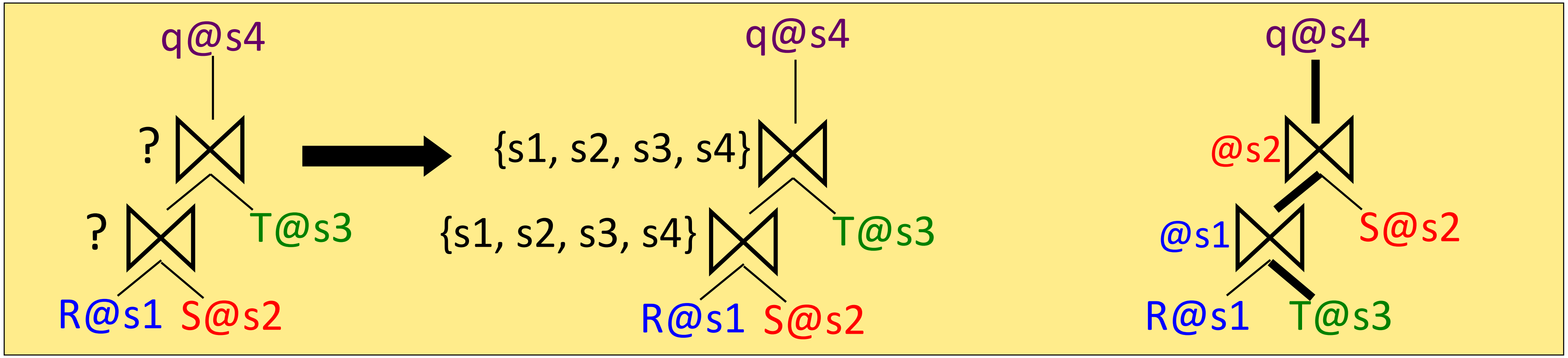


Plan pruning criteria if all the sites and network connections have equal performance:

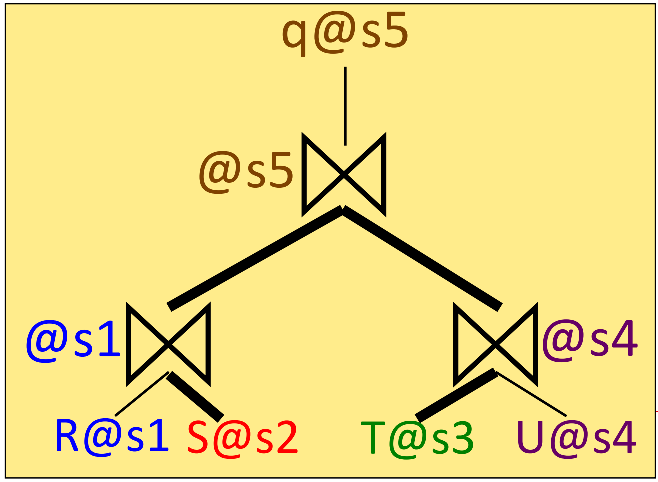
- *Ship the <u>smaller</u> collection*
- *Transfer to join partner or the query site*

Distributed query optimization

Example 1: R@s1, S@s2, T@s3, q@s4



Example 2: R@s1, S@s2, T@s3, U@s4, q@s5



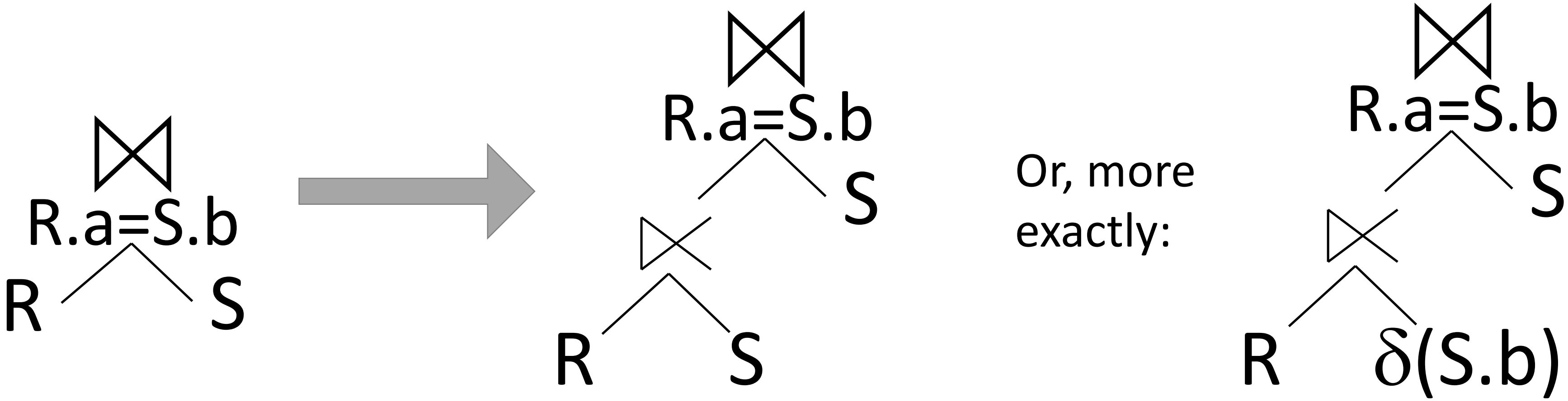
Plan pruning criteria if all the sites and network connections have equal performance:

- *Ship the <u>smaller</u> collection.*
- *Transfer to join partner or the query site*

This plan illustrates total effort != response time

Distributed query optimization technique: semijoin reducers

R join S = (R semijoin S) join S

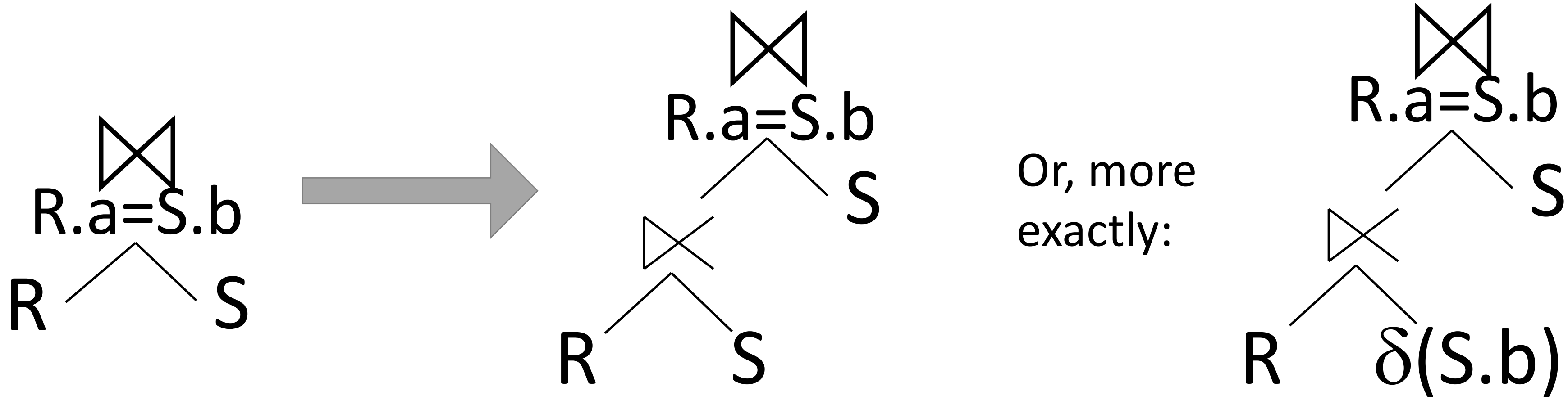


Useful in distributed settings to reduce transfers: *if the distinct S.b values* are smaller than *the non-joining R tuples*

Example: 1.000.000 tuples in R, 1.000.000 tuples in S, 900.000 distinct values of R.a, 10 distinct values of S.b

Distributed query optimization technique: semijoin reducers

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-joining R tuples*

Symetrical alternative: R join S = R join (S semijoin R)

This gives one more alternative in every join → search space explosion

Heuristics [Stocker, Kossmann et al., ICDE 2001]

Data warehouse

A data warehouse is a **large database** with a **single consolidated schema**.

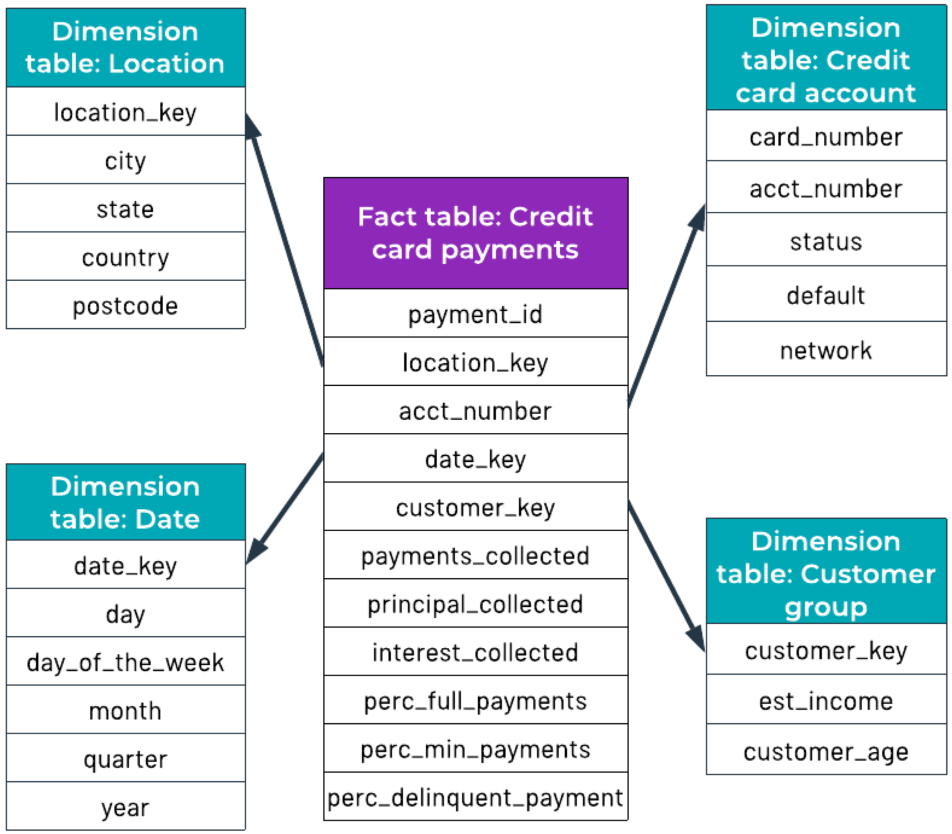
It is built within one organization with strong control (or a tight collaboration).

Typically, a warehouse schema contains:

- A very large table, called **fact table**. Each fact is characterized by several dimensions.
- A set of small(er) **dimension tables** which describe dimension values.
- A dimension value may be shared by many facts → avoid redundancy in the fact table.

Usually called a **star schema**.

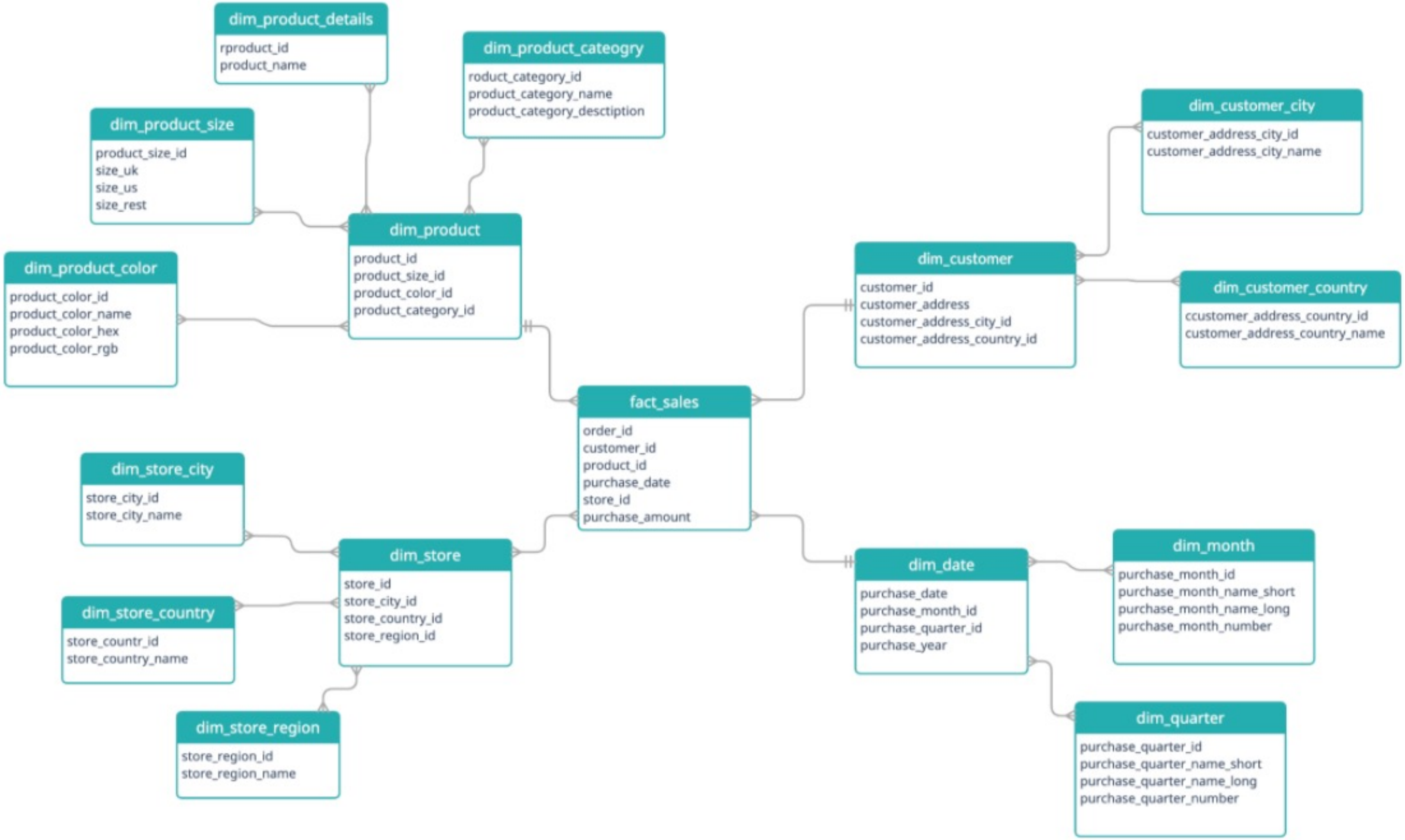
Data from many sources can go through **Extract-Transform-Load (ETL)** processes to feed the DW.



Data warehouse

A data warehouse is a large database with a single consolidated schema.

Further splitting the dimension tables leads to a **snowflake schema**.



Data integration systems (aka mediation systems)

A number of databases ("**data sources**"), **independently built** and operated by independent organizations, must be **used together**

- E.g., providers of goods or services that <u>sell something together</u> via a Web site
  - E-commerce: buy P from S1 or from S2
  - Travel: buy a trip = hotel + restaurant + rental car from S1 and S2 and S3
- E.g., large scientific studies where different labs <u>gather separate experiment data</u> for a joint study
  - Health: patient cohorts followed in separate hospitals
  - Climate: measures of ocean water, resp. air temperature and wind

Data integration systems (aka mediation systems)

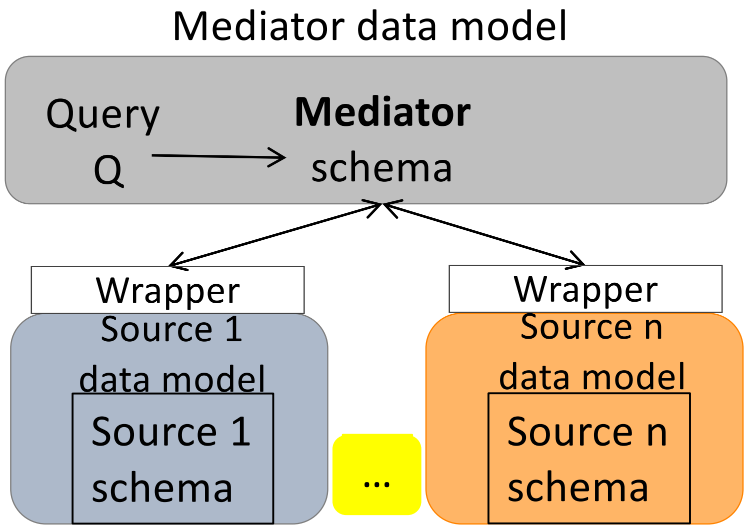
A number of databases ("**data sources**"), **independently built** and operated by independent organizations, must be **used together**

Each data source has its own schema

The data integration system shows **a single schema** to users/applications, hiding the complexity of: different schema, possibly distributed databases…

No data actually follows the integrated schema!

Mappings (logical formulae) relate the source schemas to the integrated (or mediator) schema.

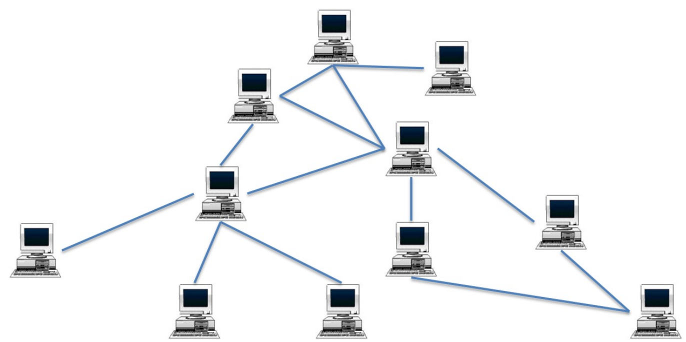


Peer-to-peer databases

Decentralized, highly distributed, symetric architectures

A **peer** (or **node**) may publish (share) some data, independently from others.

While the peer is part of the P2P network, other peers can query its data.



To enable other peers to find the data, need to **advertise**

- Propagate to other nodes information about each node's data

P2P networks are **dynamic** (peers may join or leave); peer <u>churn</u>

Data must be re-advertised to reflect

- Changes in the data
- Changes in the peer network

Data lakes

Data files and/or databases accumulate within large organizations

- E.g., Open Data from a large city, a region, or a country: administration and commerces, restaurants, census, school information…
- E.g., sales data within a company: online sale logs, advertisement campaigns, market forecasts, consolidated sale numbers…
- E.g., all weather data from a climate studies lab

Large numbers of files (1000s); large or small

Heterogeneous data models; often CSV, TSV, .XLS, … Documents also possible.

No single schema: too hard or impractical to design, also because of lack of centralized application focus or control.

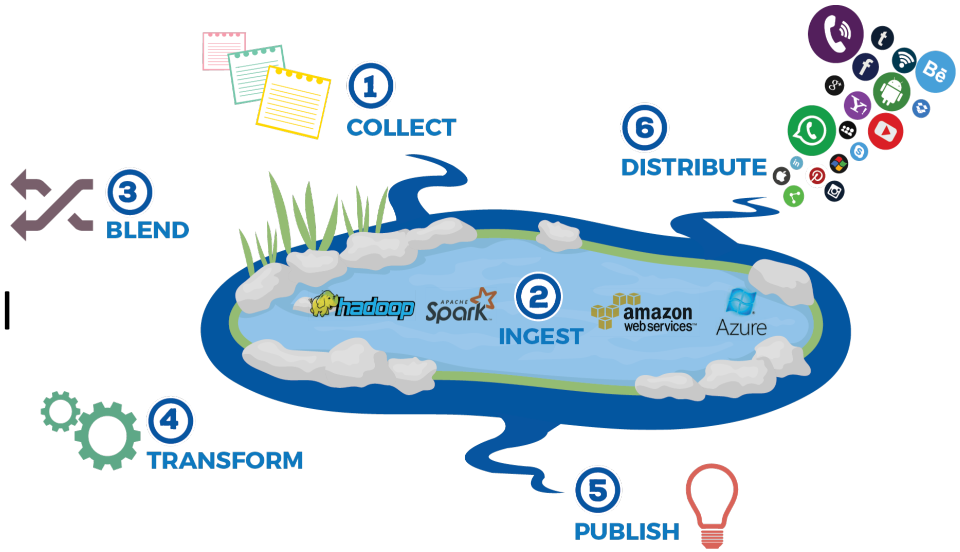
Data lakes

Data files and/or databases accumulate within large organizations

Large numbers of files (1000s); large or small

Heterogeneous data models; often CSV, TSV, .XLS, … Documents also possible.

No single schema: too hard or impractical to design, also because of lack of centralized application focus or control.



INGEST: into large-scale, possibly cloud-hosted storage, for massively parallel processing (see next)

BLEND: combine two or more datasets

BLEND and TRANSFORM are reminiscent of ETL (Extract, Transform, Load) in Data Warehouses

How to exploit a data lake?

Questions to address:

1. **Dataset search**: find a dataset by keyword search, format, date, …
2. **Dataset annotation**: attach ontology concepts to a dataset
3. Find datasets **compatible with this one**:

- Same-type, same-meaning attribute(s) → They may join.
  Difficulty: different attribute names, different data types…
- Same schema → They may be unioned.
- Share some attributes → They may be joined and/or their projections may be unioned, etc.

The datasets understood sufficiently well (« clean tables ») may be moved from the lake to a warehouse

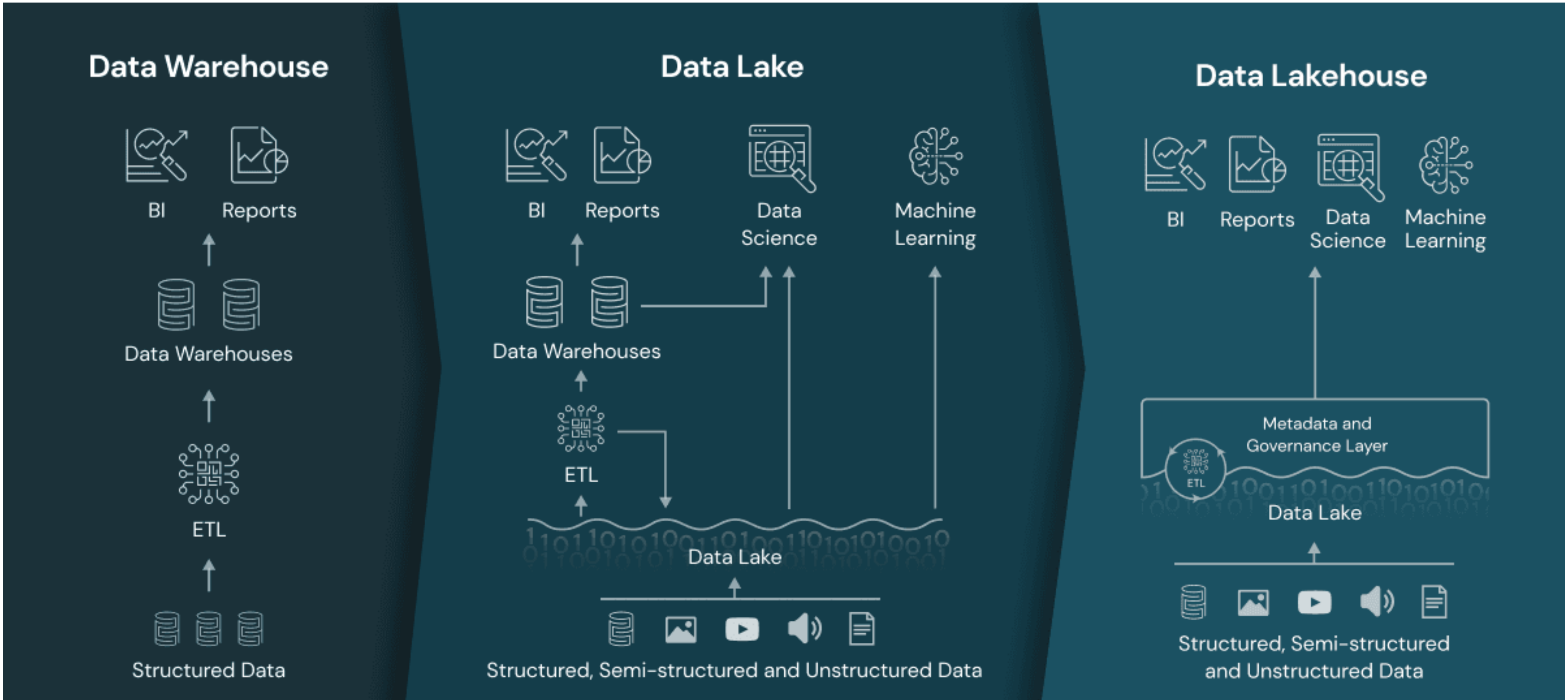
Data lake without proper analysis means: « data swamp »

Data lakehouse

A more recent brand of systems aiming to provide **data warehouse-style processing in a lake-style environment**

Metadata: file names, descriptions, tags…

Governance: access rights, predefined workflows for some data processes

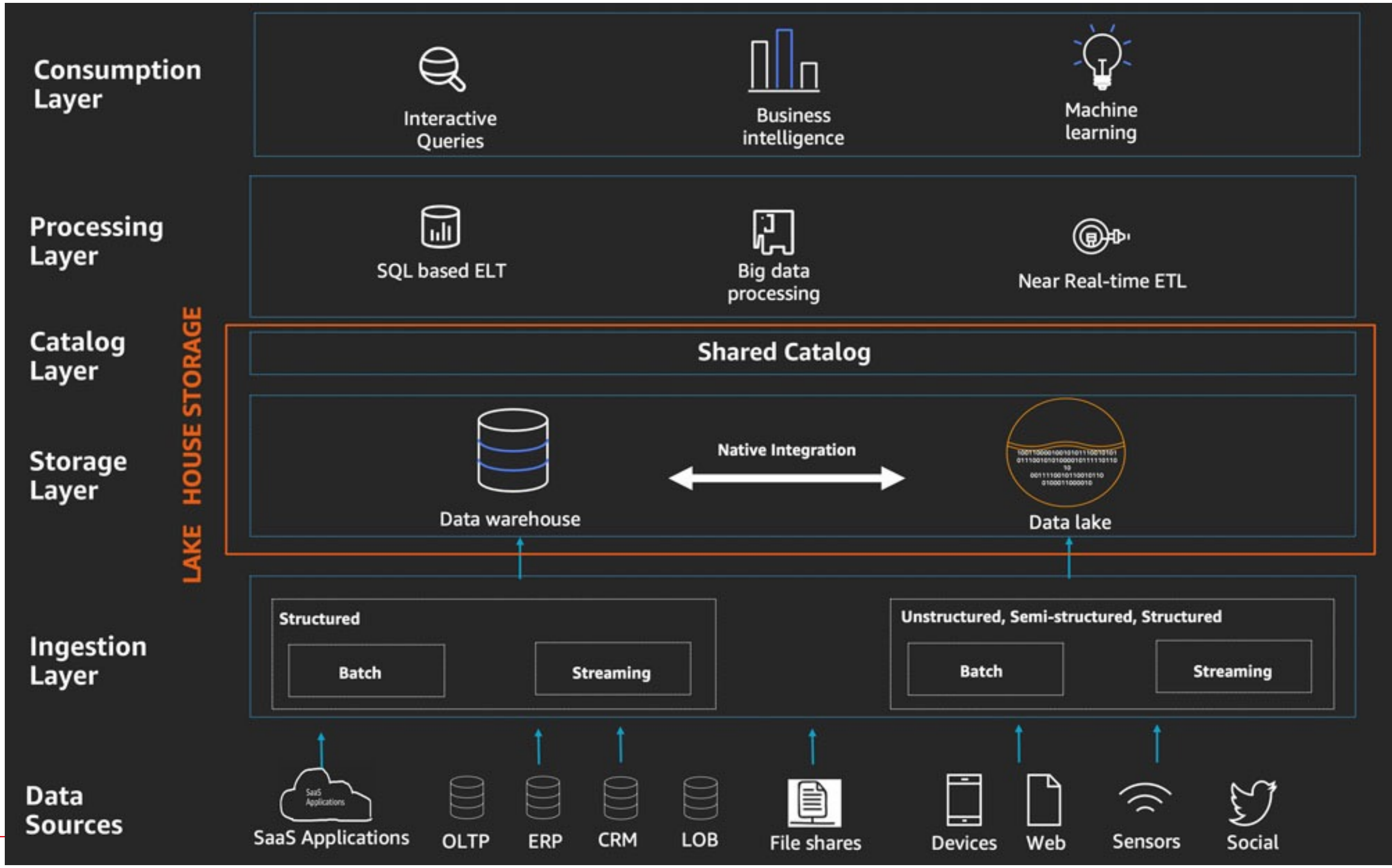


https://www.databricks.com/glossary/data-lakehouse

Data lakehouse

A more recent brand of systems aiming to provide **data warehouse-style processing in a lake-style environment**

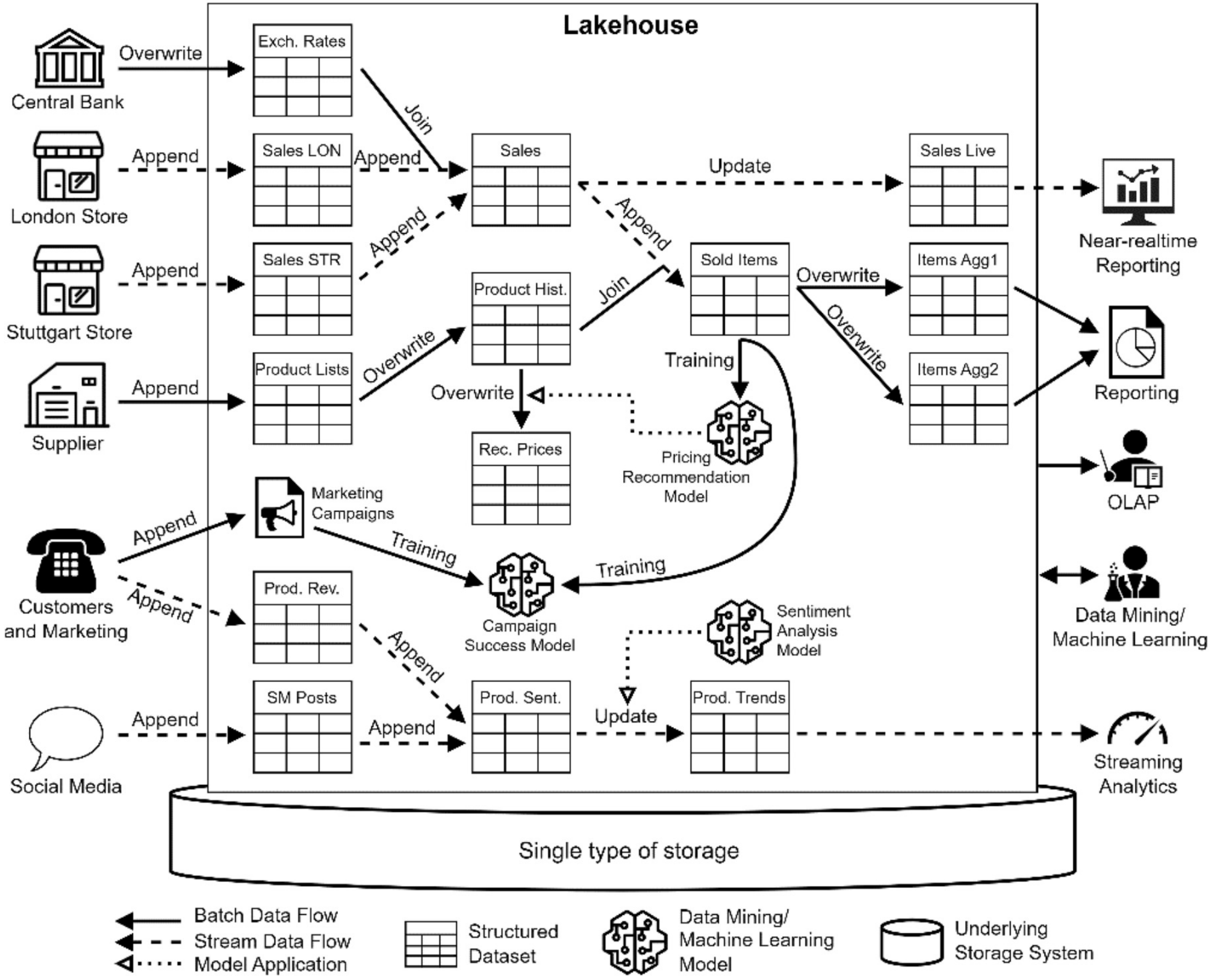
OLTP: online transaction processing   CRM: customer relationship management
ERP: Enterprise Resource Planning (HR, manufacturing, supply chain, finance, accounting; generic)
LOB: Line of Business (specific to the company)



https://aws.amazon.com/fr/blogs/big-data/build-a-lake-house-architecture-on-aws/

Sample lakehouse

Schneider et al., 2024 https://doi.org/10.1007/s42979-024-02737-0

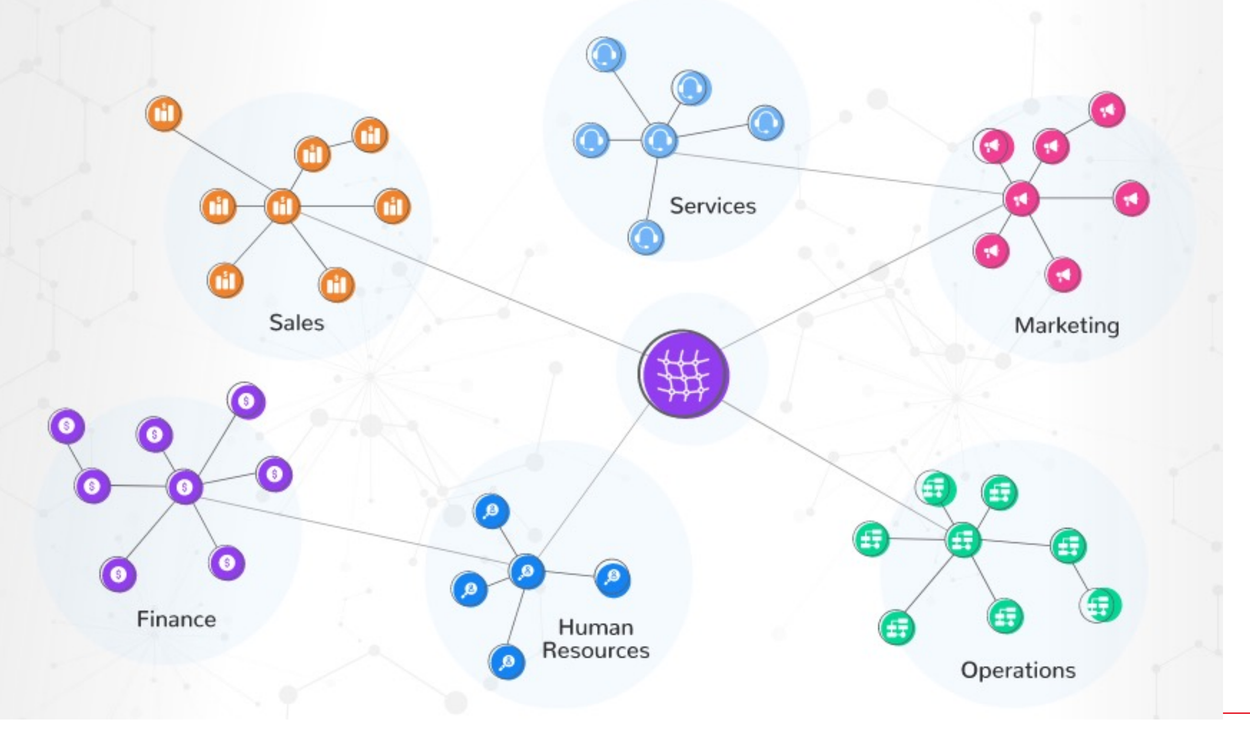


Data mesh

Most recent category of systems, some tractions in industry (Netflix, Paypal, Amazon…)

Four core principles:

1. Domain ownership: domain (application) specialists decide what data to store, how it should be structured, described, etc.



Data mesh

Four core principles:

1. **Domain ownership**: domain (application) specialists decide what data to store, how it should be structured, described, etc. E.g., personnel, financial, marketing…
2. **Data as a product**: each dataset, original or derived, should :
   - Be discoverable
   - Be addressable
   - Be trustworthy
   - Have self-describing semantics and syntax
3. **Self-serve data platform**: easy for domain teams to add/modify/work on data
4. **Federated computational governance** across the domain teams + technical infrastructure

Recent term, durability unclear.
Main emphasis is on organizational, not technical aspects.
Organization is important, too.

Cloud computing and cloud data management

Cloud computing

Idea: delegate large-scale storage and large-scale computing to remote centers

Run by the (only) enterprise using them: private clouds

- Large companies can afford the cost to own and operate a cloud service: La Poste, Orange, ...

Run by a company who rents out storage and computing services: commercial clouds

- Main players: Amazon (has basically *created* the industry), Google, Microsoft

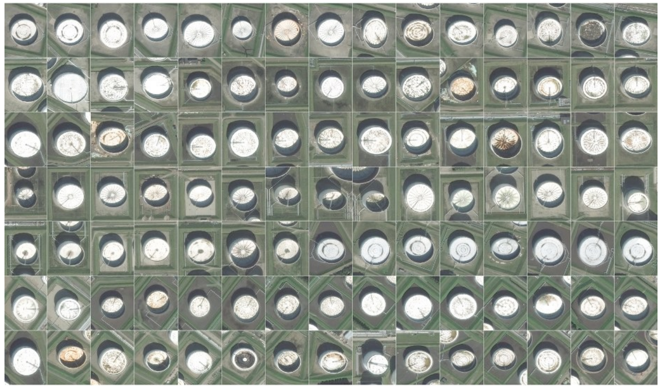
Advantages of cloud computing

Allow companies to focus on their main business
not on IT

Allow scaling the resource usage up and down according to the needs

Examples:

- Shops with more clients as Christmas approaches
- Tax statements, built once a year
- Satellite image data processing company which needs significant computing resources (only) when it has an order from a client



https://www.wired.com/2015/03/orbital-insight/

How cloud services work (1/3)

Data storage at scale

Users upload files to be hosted on cloud provider's servers

Data is <u>replicated</u> for reliability and quick access

The service is paid by the GB and day

- Total cost = sum(file size x file storage time)

Computing

Users typically buy virtual computers (virtual machines, VM)

Service paid by the duration of use of the VM

Each VM is <u>hosted</u> by some physical computer in the cloud provider's cluster

If a physical machine fails, the VM is recreated elsewhere and the work restarts

How cloud services work (1/3)

Data storage at scale

Users upload files to be hosted on cloud provider's servers

Data is replicated for reliabil

The service is paid by the GB

▶ Total cost = sum(file size x file

Computing

Users typically buy virtual co

Service paid by the duration

Each VM is hosted by some p
provider's cluster

If a physical machine fails, the VM will be recreated elsewhere and the work will restart

The separation of storage and computing, in the way they are provided and purchased, is called **disaggregation**.

It is a radical departure from the previous database management architectures. It is specific to the cloud environment.

How cloud services work (2/3)

Computing (continued)

There are typically different sizes (capacities) of virtual machines

- E.g., Small (**S**), Medium (**M**), Large (**L**), Extra-Large (**XL**); nowadays hundreds of sizes (or instance types)
- The difference is in the *computing speed (# of cores and their speed), memory size, network connectivity...*

Fast storage of small-granularity data, typically in memory in the cloud

For: metadata (catalog, user management, ...)

Key-value stores, document stores

Pay per operation (put, get)

Other services

E.g. messaging **queues** to synchronize different applications

How cloud services work (3/3): cloud computing models

Infrastructure-as-a-service (IaaS)

The vendor provides access to computing resources such as servers, storage and networking.

Clients use their own platforms and applications within a service provider's infrastructure.
They do not host but they *develop, deploy and administer* in the cloud.

Platform-as-a-service (Paas)

The vendor provides: storage and other computing resources, prebuilt tools to develop, customize and test their own applications.

Clients do not host and mostly do not administer either.

They still *develop and deploy* in the cloud.

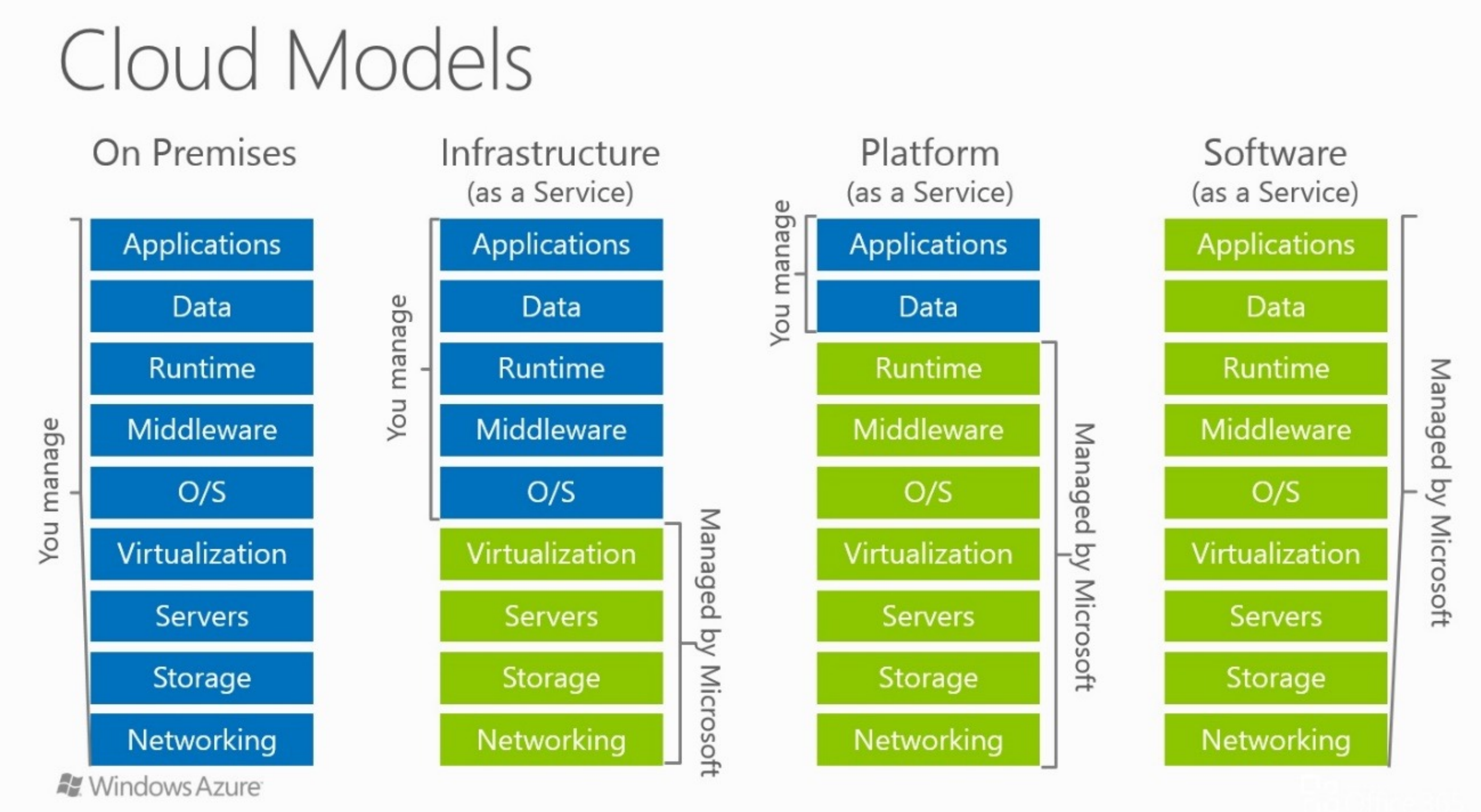
How cloud services work (3/3): cloud computing models

Software-as-a-service (SaaS)

The vendor provides: storage and other computing resources; software and applications via a subscription model (or pay-per-use...)

Clients *access* the applications remotely.
They do not store, host, develop nor administer.

Example of Microsoft Azure



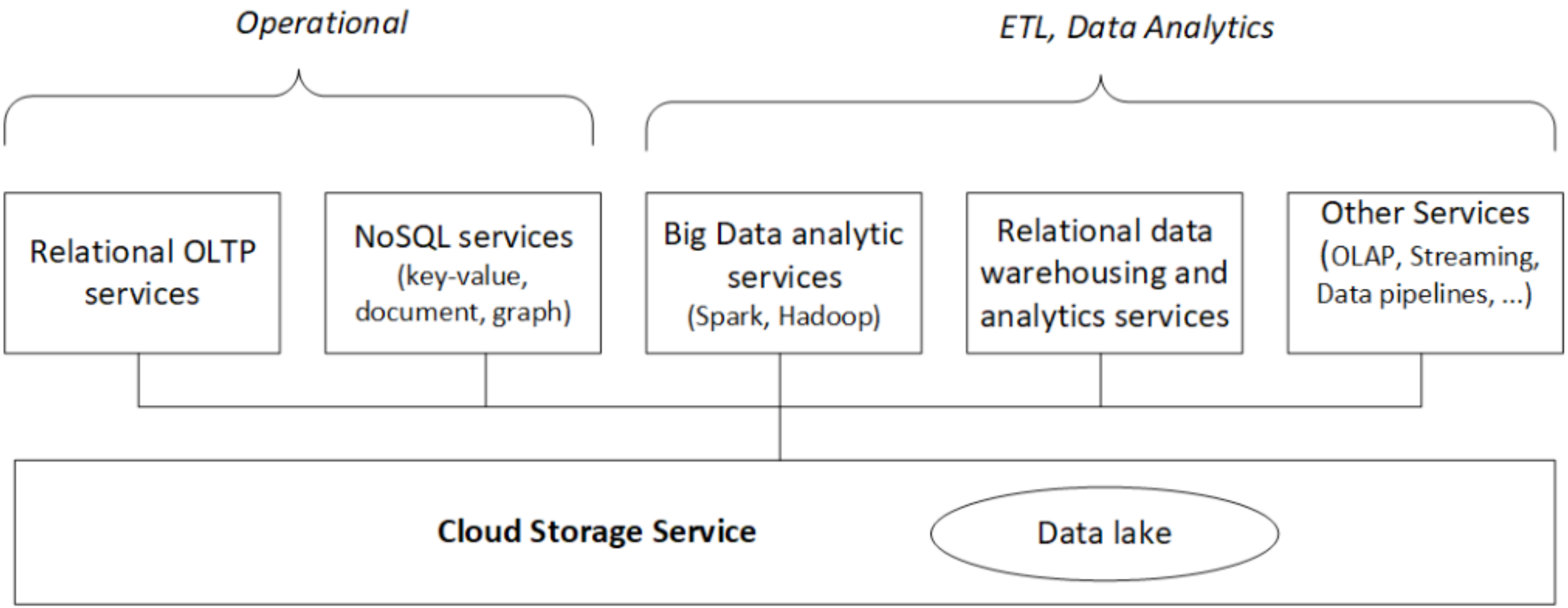
https://docs.microsoft.com/fr-fr/azure/cloud-adoption-framework/strategy/monitoring-strategy

Cloud services

	Amazon web services	Google Cloud Platform	Windows Azure
File storage service	Amazon Scalable Storage Service (S3)	Google Cloud Storage	Windows Azure BLOB Storage
Virtual machines	Amazon Elastic Compute Cloud (EC2)	Google Compute Engine	Windows Azure Virtual Machines
Fine-granularity data store	Amazon DynamoDB	Google High Replication Datastore	Windows Azure Tables
Queue service	Amazon Simple Queue Service (SQS)	Google Task Queues	Windows Azure Queues

Major vendors still actively publishing new features/tools!

Cloud database services



V. Narasayya and S. Chaudhuri (Microsoft). « Cloud Data Services: Workloads, Architectures, and Multi-tenancy », Foundations and Trends in Data Management, 2021.

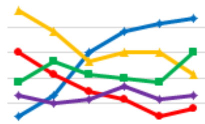
Relational database ranking

DB-Engines Ranking of Relational DBMS

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

This is a partial list of the complete ranking showing only relational DBMS.

Read more about the method of calculating the scores.



trend chart

☐ include secondary database models

165 systems in ranking, December 2023

Cloud-native systems

Rank Dec 2023	Rank Nov 2023	Rank Dec 2022	DBMS	Database Model	Score Dec 2023	Score Nov 2023	Score Dec 2022
1.	1.	1.	Oracle	Relational, Multi-model	1257.41	-19.62	+7.10
2.	2.	2.	MySQL	Relational, Multi-model	1126.64	+11.40	-72.76
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	903.83	-7.59	-20.52
4.	4.	4.	PostgreSQL	Relational, Multi-model	650.90	+14.05	+32.93
5.	5.	5.	IBM Db2	Relational, Multi-model	134.60	-1.40	-12.02
6.	↑ 7.	6.	Microsoft Access	Relational	121.75	-2.74	-12.08
7.	↑ 8.	↑ 8.	Snowflake	Relational	119.88	-1.12	+5.11
8.	↓ 6.	↓ 7.	SQLite	Relational	117.95	-6.63	-14.49
9.	9.	9.	MariaDB	Relational, Multi-model	100.43	-1.66	-0.50
10.	10.	10.	Microsoft Azure SQL Database	Relational, Multi-model	83.04	-0.13	+1.06
11.	11.	↑ 13.	Databricks	Multi-model	80.31	+3.09	+19.57
12.	12.	↓ 11.	Hive	Relational	69.41	+0.77	-8.49
13.	13.	↑ 14.	Google BigQuery	Relational	62.17	+2.85	+6.48
14.	14.	↓ 12.	Teradata	Relational, Multi-model	55.69	-1.63	-10.19
15.	15.	15.	FileMaker	Relational	54.18	+1.75	+0.33
16.	16.	16.	SAP HANA	Relational, Multi-model	48.80	-0.32	-1.40
17.	17.	17.	SAP Adaptive Server	Relational, Multi-model	40.66	-0.83	-2.10
18.	↑ 19.	↑ 19.	Firebird	Relational	27.93	+2.29	+3.70
19.	↓ 18.	↑ 20.	Microsoft Azure Synapse Analytics	Relational	26.64	-0.16	+4.39
20.	20.	↓ 18.	Amazon Redshift	Relational	22.23	+0.86	-3.31

https://db-engines.com/en/ranking

Relational database ranking

Also offered an cloud services

Cloud-native systems

☐ include secondary database models — 165 systems in ranking, December 2023

Rank Dec 2023	Rank Nov 2023	Rank Dec 2022	DBMS	Database Model	Score Dec 2023	Score Nov 2023	Score Dec 2022
1.	1.	1.	Oracle	Relational, Multi-model	1257.41	-19.62	+7.10
2.	2.	2.	MySQL	Relational, Multi-model	1126.64	+11.40	-72.76
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	903.83	-7.59	-20.52
4.	4.	4.	PostgreSQL	Relational, Multi-model	650.90	+14.05	+32.93
5.	5.	5.	IBM Db2	Relational, Multi-model	134.60	-1.40	-12.02
6.	↑ 7.	6.	Microsoft Access	Relational	121.75	-2.74	-12.08
7.	↑ 8.	↑ 8.	Snowflake	Relational	119.88	-1.12	+5.11
8.	↓ 6.	↓ 7.	SQLite	Relational	117.95	-6.63	-14.49
9.	9.	9.	MariaDB	Relational, Multi-model	100.43	-1.66	-0.50
10.	10.	10.	Microsoft Azure SQL Database	Relational, Multi-model	83.04	-0.13	+1.06
11.	11.	↑ 13.	Databricks	Multi-model	80.31	+3.09	+19.57
12.	12.	↓ 11.	Hive	Relational	69.41	+0.77	-8.49
13.	13.	↑ 14.	Google BigQuery	Relational	62.17	+2.85	+6.48
14.	14.	↓ 12.	Teradata	Relational, Multi-model	55.69	-1.63	-10.19
15.	15.	15.	FileMaker	Relational	54.18	+1.75	+0.33
16.	16.	16.	SAP HANA	Relational, Multi-model	48.80	-0.32	-1.40
17.	17.	17.	SAP Adaptive Server	Relational, Multi-model	40.66	-0.83	-2.10
18.	↑ 19.	↑ 19.	Firebird	Relational	27.93	+2.29	+3.70
19.	↓ 18.	↑ 20.	Microsoft Azure Synapse Analytics	Relational	26.64	-0.16	+4.39
20.	20.	↓ 18.	Amazon Redshift	Relational	22.23	+0.86	-3.31
21.	21.	21.	Informix	Relational, Multi-model	20.94	+0.29	-0.97
22.	22.	22.	Spark SQL	Relational	18.87	-0.37	-1.75
23.	23.	↑ 24.	Impala	Relational, Multi-model	17.39	-0.85	-0.43
24.	24.	↑ 28.	ClickHouse	Relational, Multi-model	16.96	+0.98	+3.29
25.	25.	↑ 26.	Presto	Relational	14.81	+1.04	-0.15
26.	↑ 27.	↑ 27.	dBASE	Relational	14.59	+0.92	+0.40
27.	↑ 29.		Apache Flink	Relational	13.44	+0.19	
28.	↓ 26.	↓ 23.	Vertica	Relational, Multi-model	13.30	-0.46	-5.21
29.	↓ 28.	↓ 25.	Netezza	Relational	13.17	-0.44	-3.62
30.	30.	30.	Greenplum	Relational, Multi-model	10.57	-0.17	-0.75
31.	31.	↓ 29.	Amazon Aurora	Relational, Multi-model	9.47	-0.22	-2.14
32.	32.	↓ 31.	H2	Relational, Multi-model	8.71	-0.33	+0.04
33.	33.	↓ 32.	Oracle Essbase	Relational	8.09	+0.31	-0.25
34.	34.	↑ 35.	Microsoft Azure Data Explorer	Relational, Multi-model	6.93	-0.06	+0.25

https://db-engines.com/en/ranking

Cloud and Big Data management

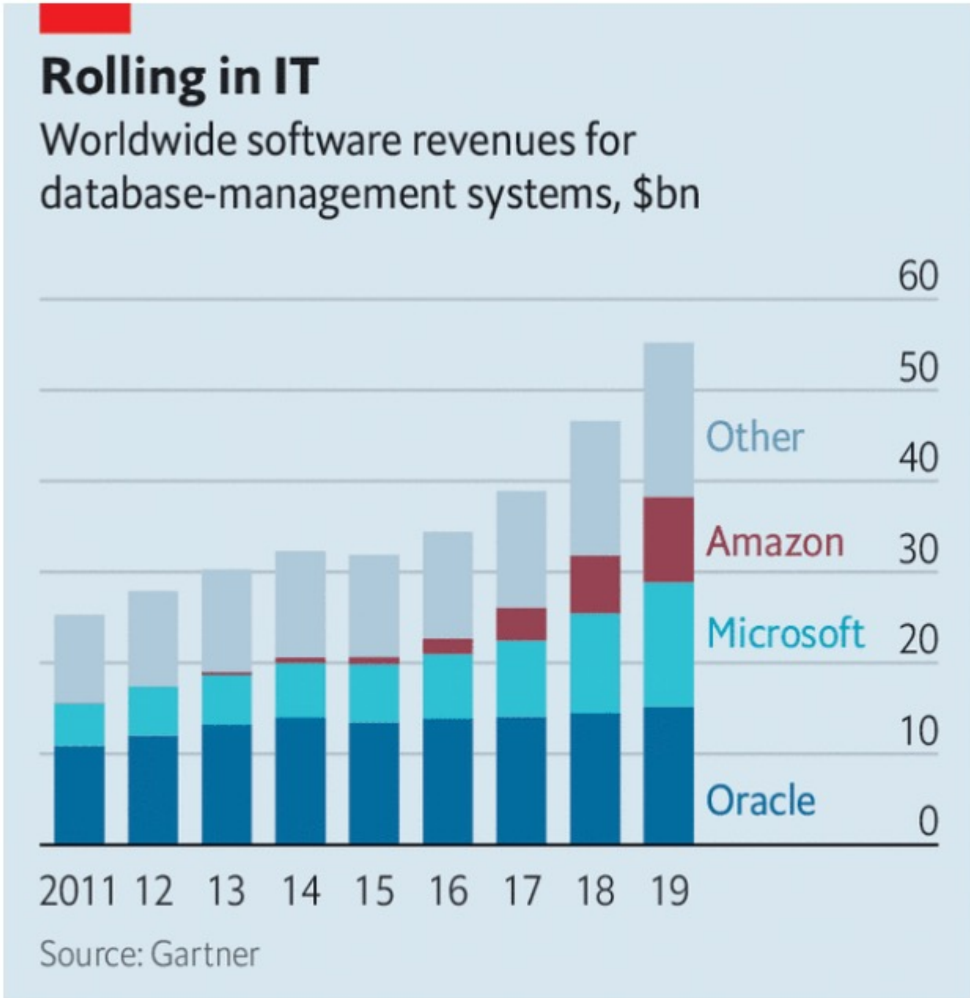
economist.com

Steam engine in the cloud - How Snowflake raised $3bn in a record software IPO I Business

Sep 15th 2020

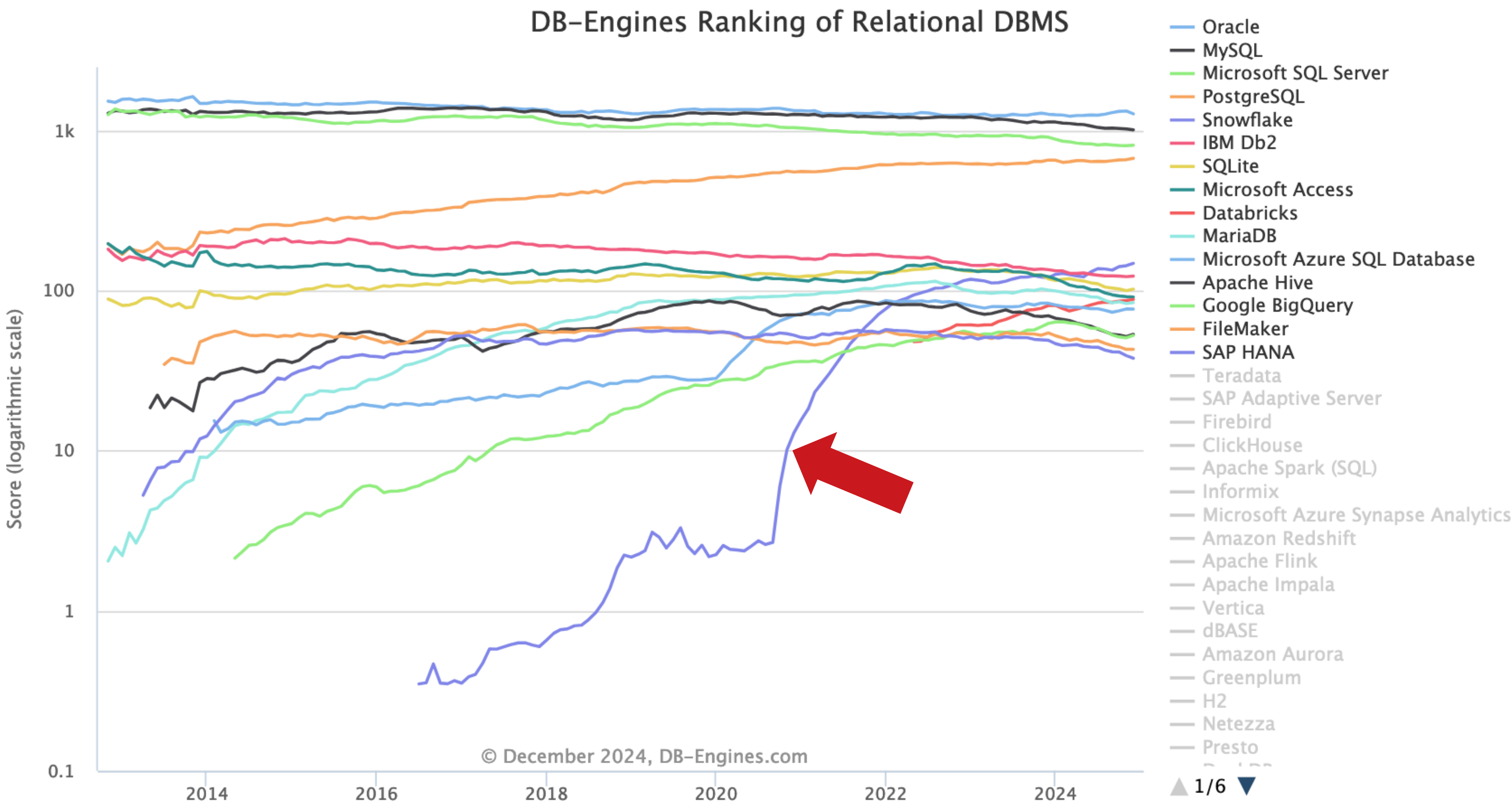
5-6 minutes

But competition in the database business is heating up

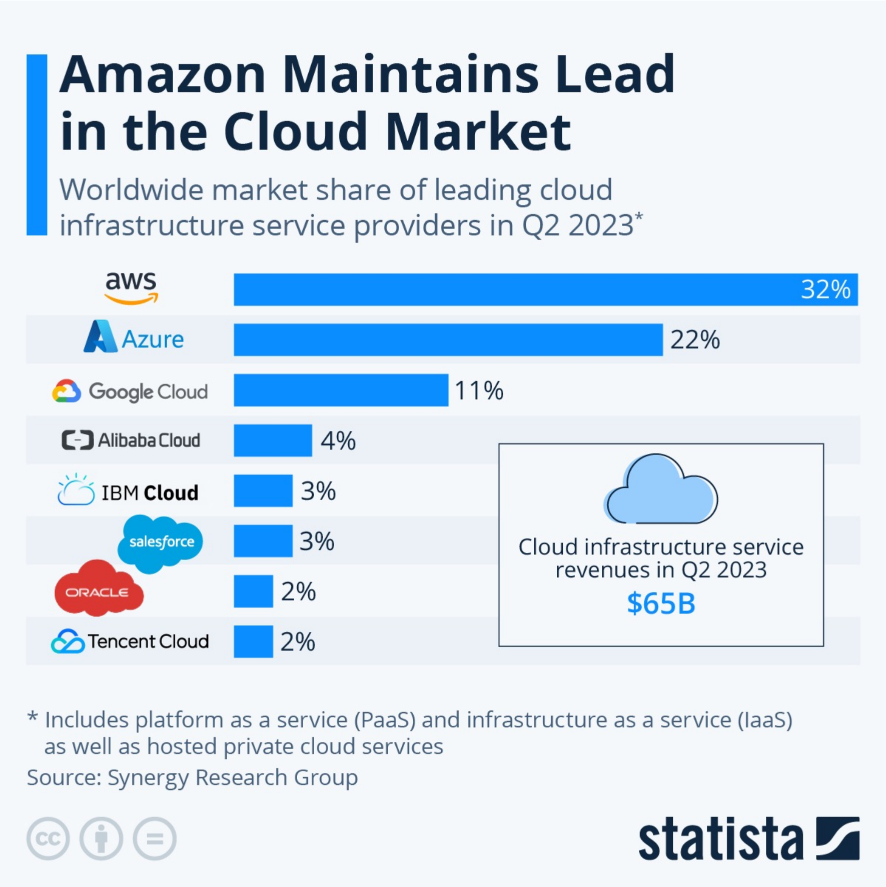


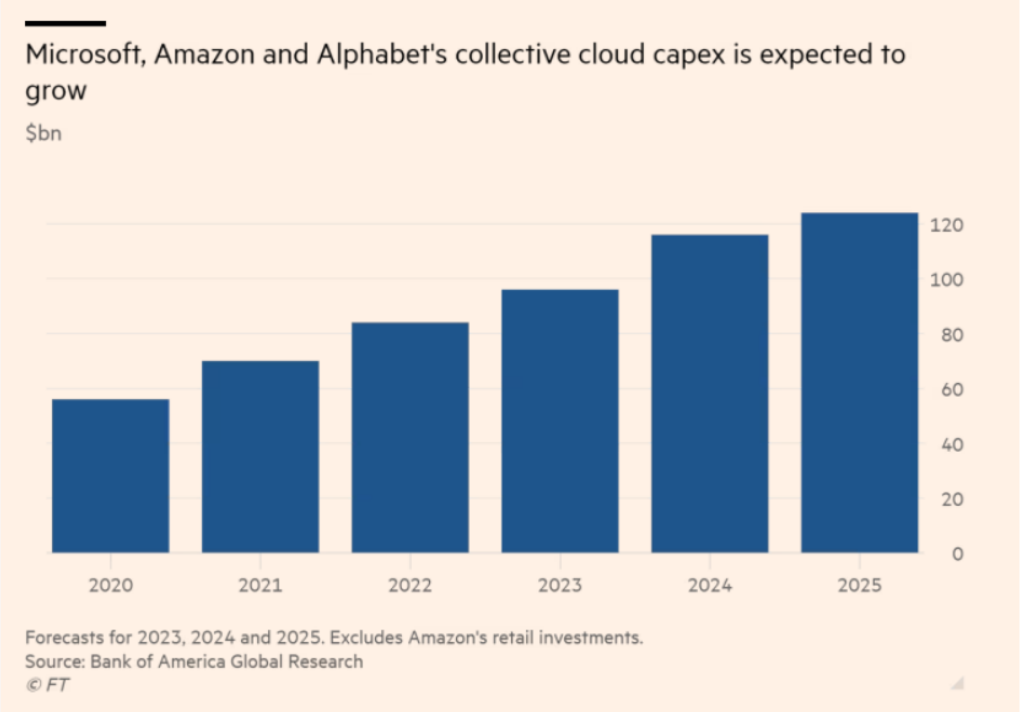
https://www.economist.com/business/2020/09/15/how-snowflake-raised-3bn-in-a-record-software-ipo

Relational database ranking, December 2024



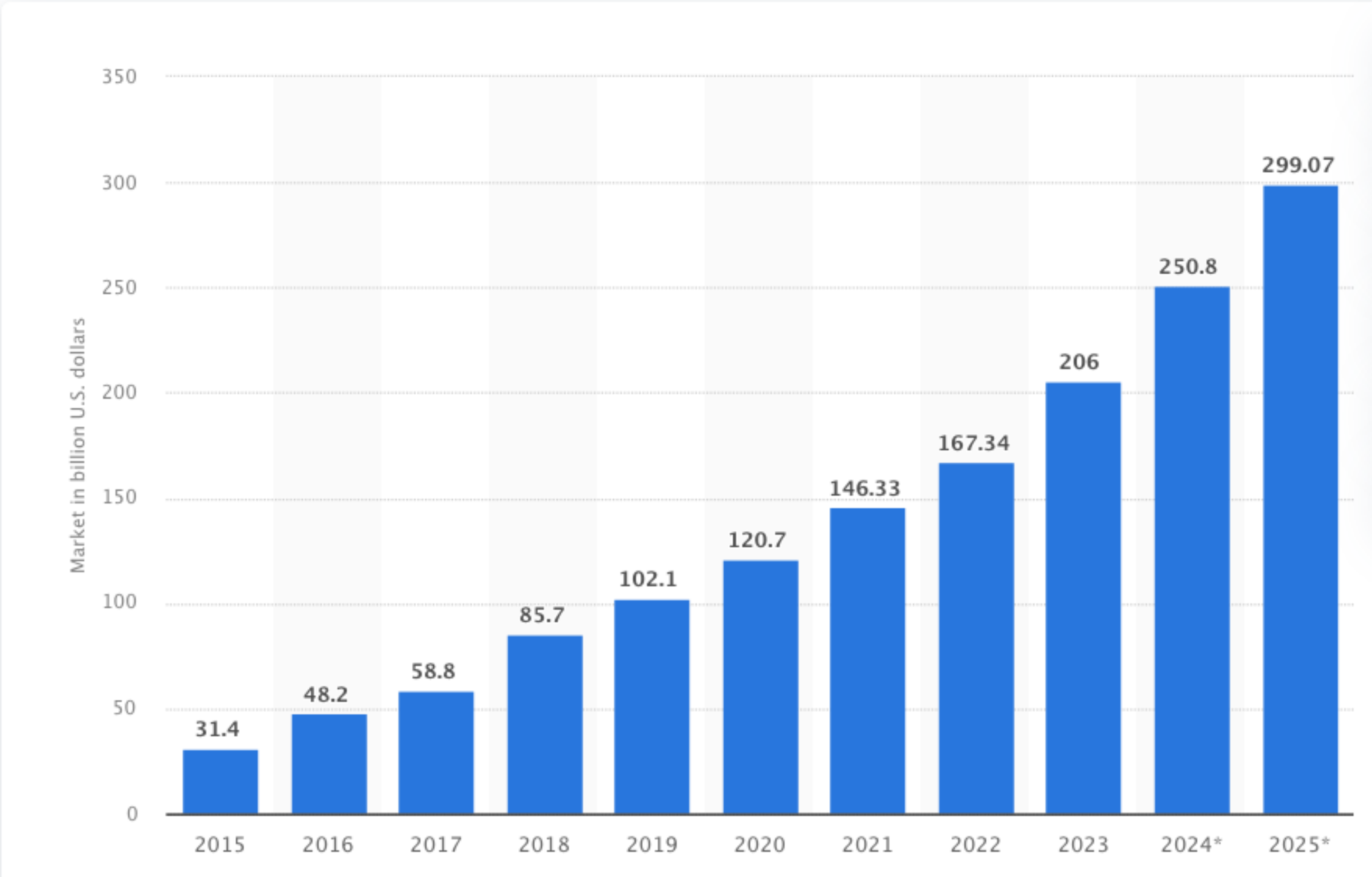
State of the Cloud Computing industry





State of the Cloud Computing industry

Public cloud application SaaS end-user spending

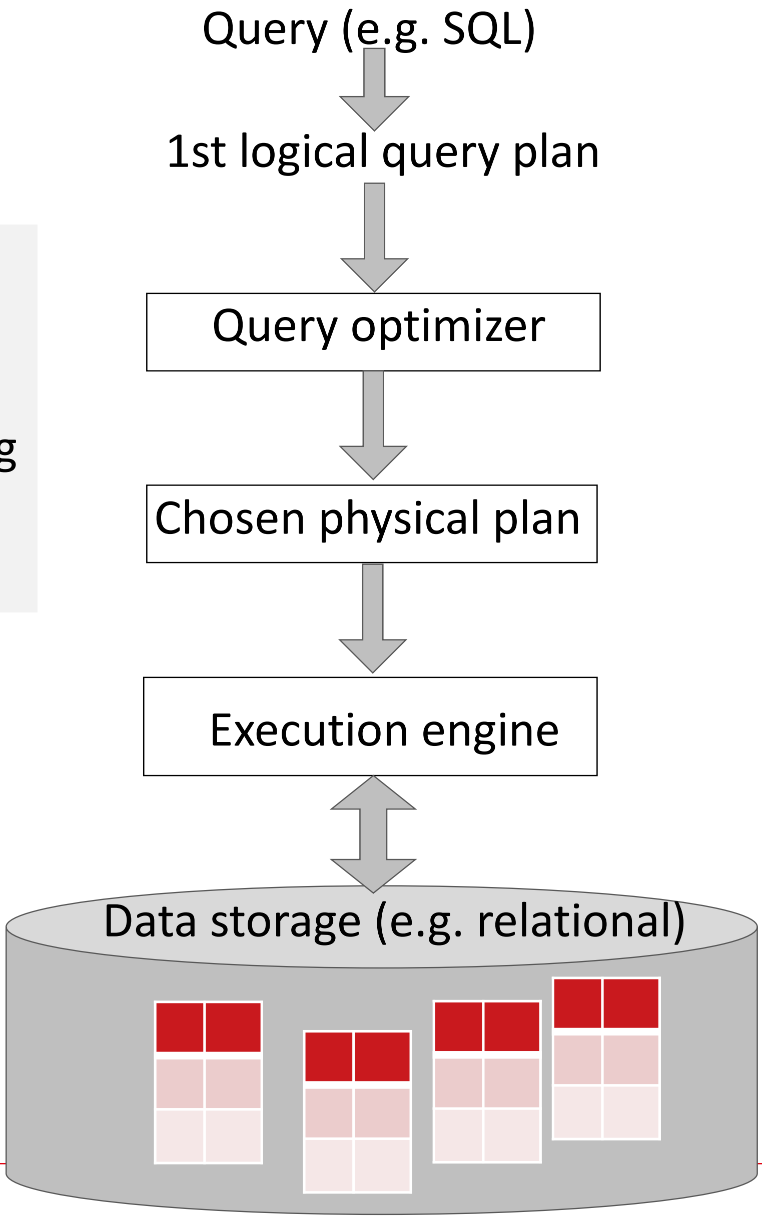


https://www.statista.com/statistics/505243/worldwide-software-as-a-service-revenue/

Cloud data management: Principles and architectures

How to build a data management platform in the cloud?

Recall: classical query processing in a database



Moving to large-scale distribution

Store (distribute) the data in a distributed file system

How to split it?

How to provide efficient access to this data?

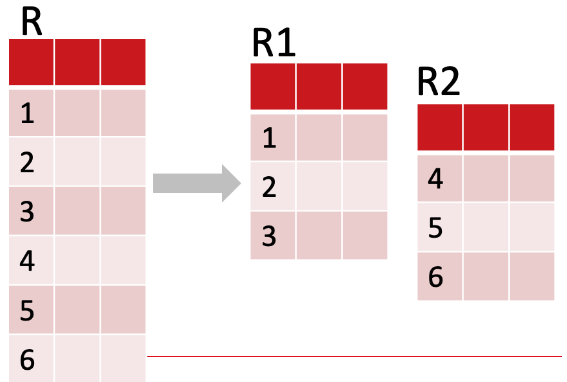
Process queries in cloud

How to evaluate operators over distributed data, in a distributed architecture?

How to optimize queries?

Distributing a large table

Goal: **distribute a table into several fragments** (or tablets, splits…) to leverage distributed storage



R=R1 U R2 U … U Rn

When there are many fragments, this horizontal distribution is also called sharding (shard: small fragment, typically of wood)

Good properties that the distribution could ensure:

- ❑ Relatively uniform distribution of data volume across the machines
- ❑ Finding easily where each record is stored

Distributing a large table via hashing

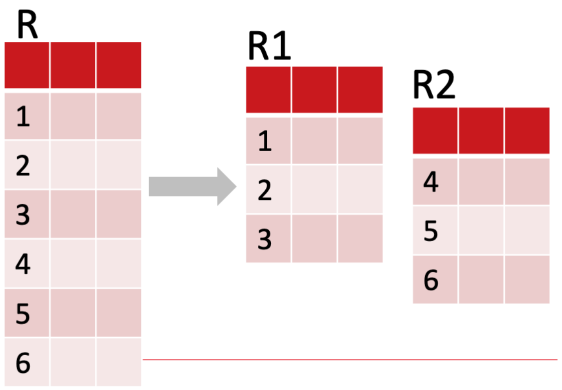
Let R be a table sharded into R1 U R2 U … U Rn.

Assume the key for **R** is a.

Assume available a **hash function** which, given an input, returns an output in the 0… 2^{n-1}, for some integer n.

Then, for each tuple r from R:

- Compute h(r.a) = k
- Tuple r will be part of shard number k
- When looking for an R tuple, we know it is on machine number k=h(a)



Hashing ensures (with high probability) uniform distribution

Also, it facilitates searching by the key

Massively parallel data data management using Map/Reduce

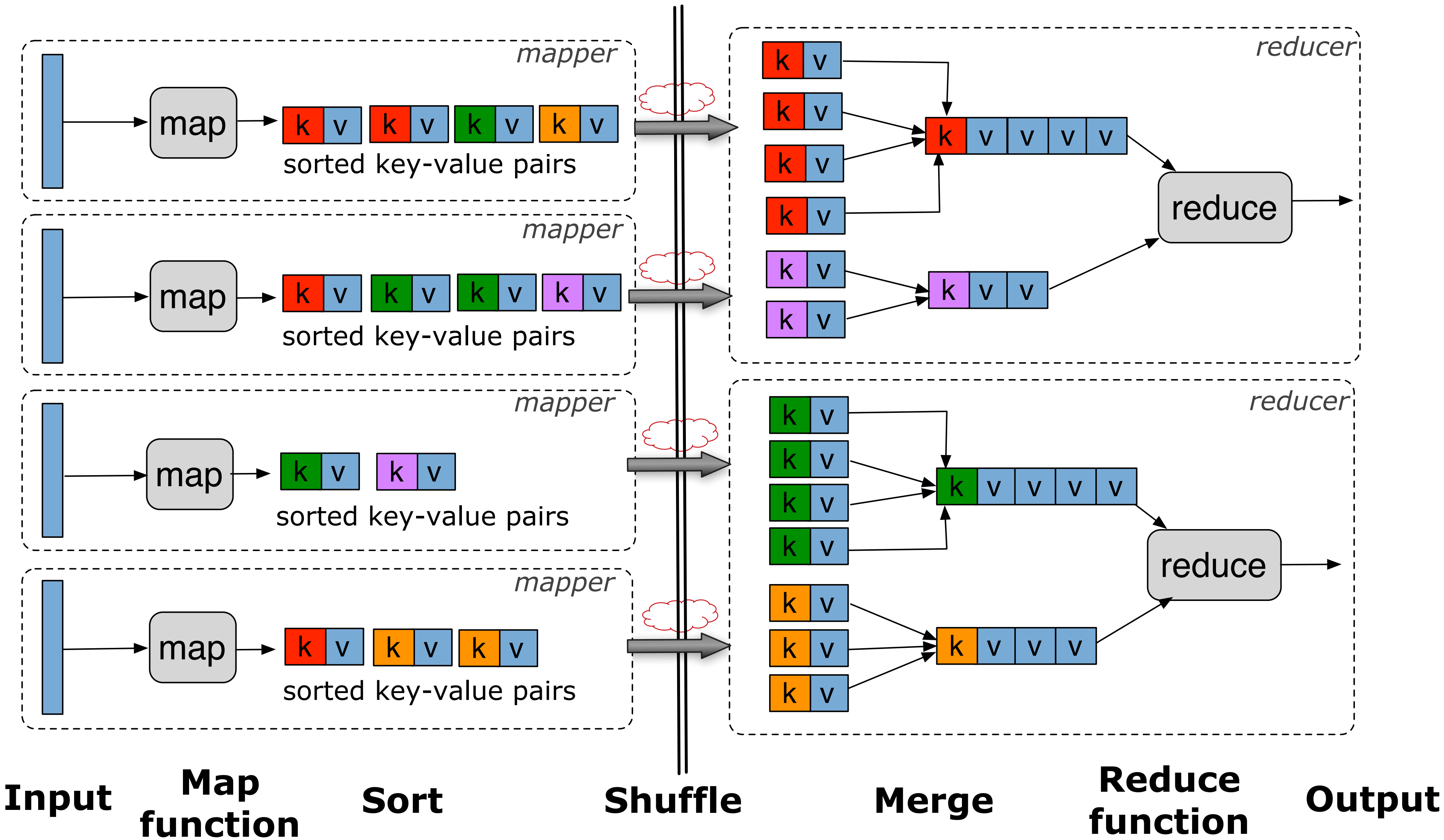
Cloud platforms provide **distributed file systems**, in which we can store very large collections of data.

Popular framework (~2010-…) for processing very large amounts of data stored on multiple machines, in a massively parallel way: **Map-Reduce**

Idea:

- Ask users to describe their desired computations by defining a set of functions
- Implement in a common framework:
  - Calling the function on all the data fragments
  - Gathering the results
  - Intra-node communication, etc.

Map/Reduce outline



Sample queries and their implementation in Map-Reduce

Assume Customer, Order are large, distributed tables

```
1. SELECT city
FROM customer c
WHERE c.name='Anne'
```

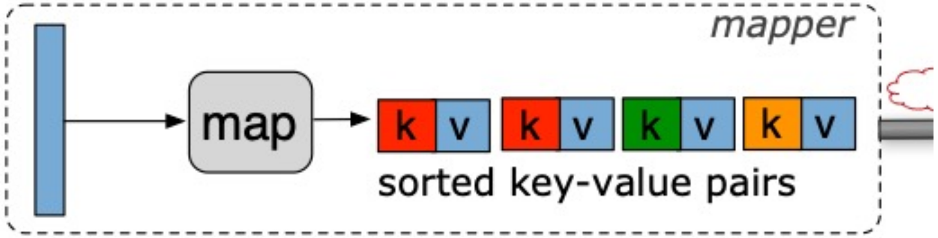
```
2. SELECT MONTH(c.start_date), COUNT(*)
FROM customer c
GROUP BY MONTH(c.start_date)
```

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

```
4. SELECT c.name, SUM(o.total)
FROM customer c, order o
WHERE c.id=o.cid
GROUP BY c.name
```

Making a selection query more efficient

```
1. SELECT city
FROM customer c
WHERE c.name='Anne'
```

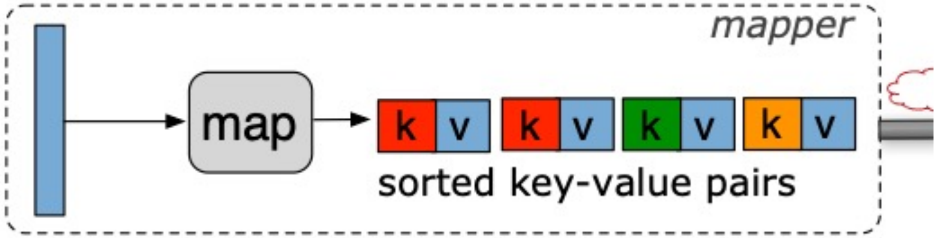


How to reduce the effort involved in reading the data?

- Add header information to each data split, summarizing split attribute values
  - E.g., Split #110 has name in {'Anna',... 'Bruce'} , or [A*...B*]
  - Possible false positives, depending on the values
  - Optimization in Hadoop, leading MapReduce implementation: Enhance the data-read request method of HDFS (Hadoop Distributed File System) into read(customer, attr1=val1, ..., attrn=valn) to avoid reading data that does not match

Making a selection query more efficient

```
1. SELECT city
FROM customer c
WHERE c.name='Anne'
```



How to reduce the effort involved in reading the data?

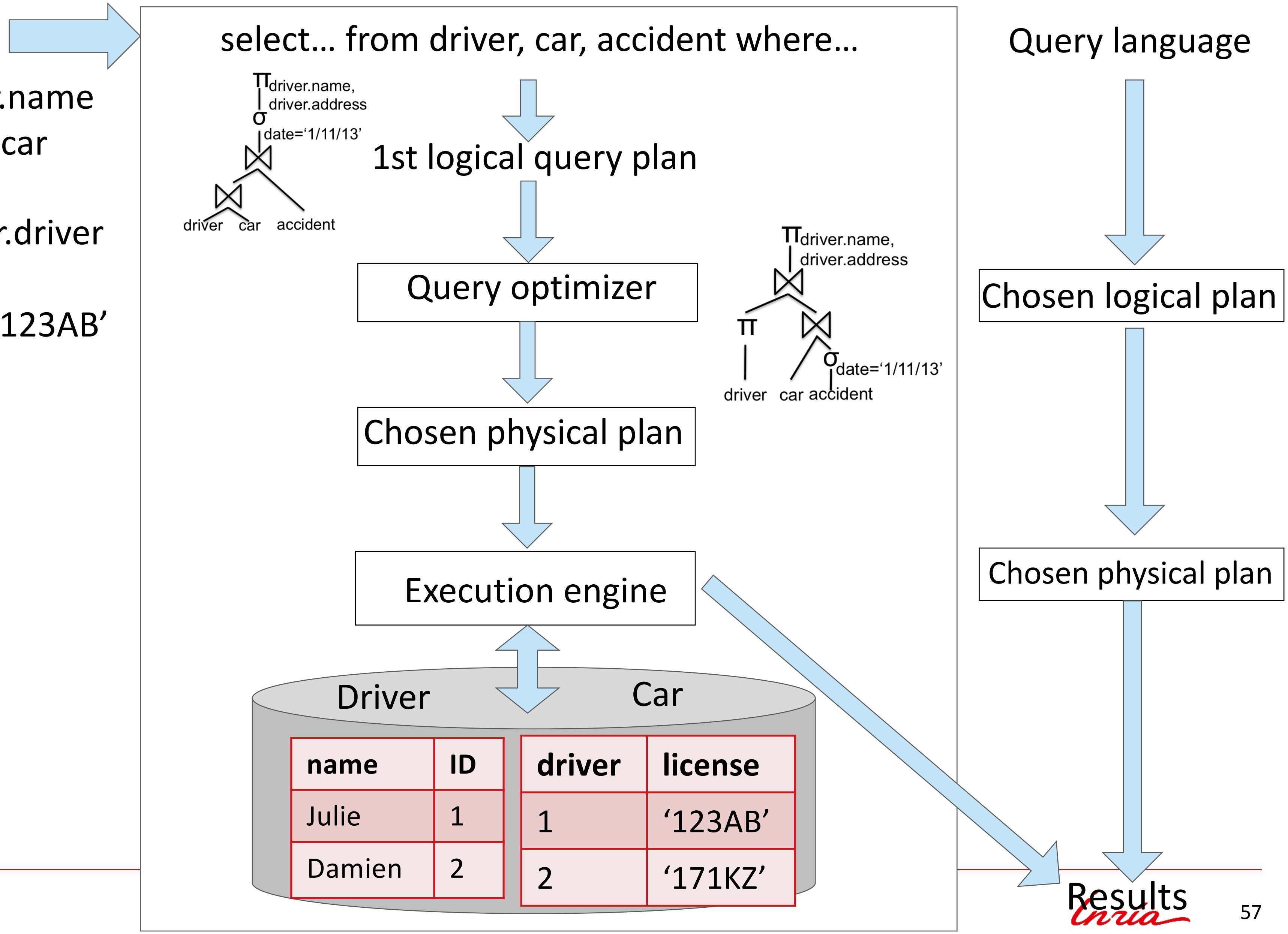
- On each node, build in-memory index of the split on that node, e.g., on c.name
  - For maximum efficiency, the index should be clustered → the split should be stored ordered by c.name
  - Hadoop typical replication factor is 3 → three indexes are possible!
  - Appropriately route queries

Example: c.name, c.age, c.city

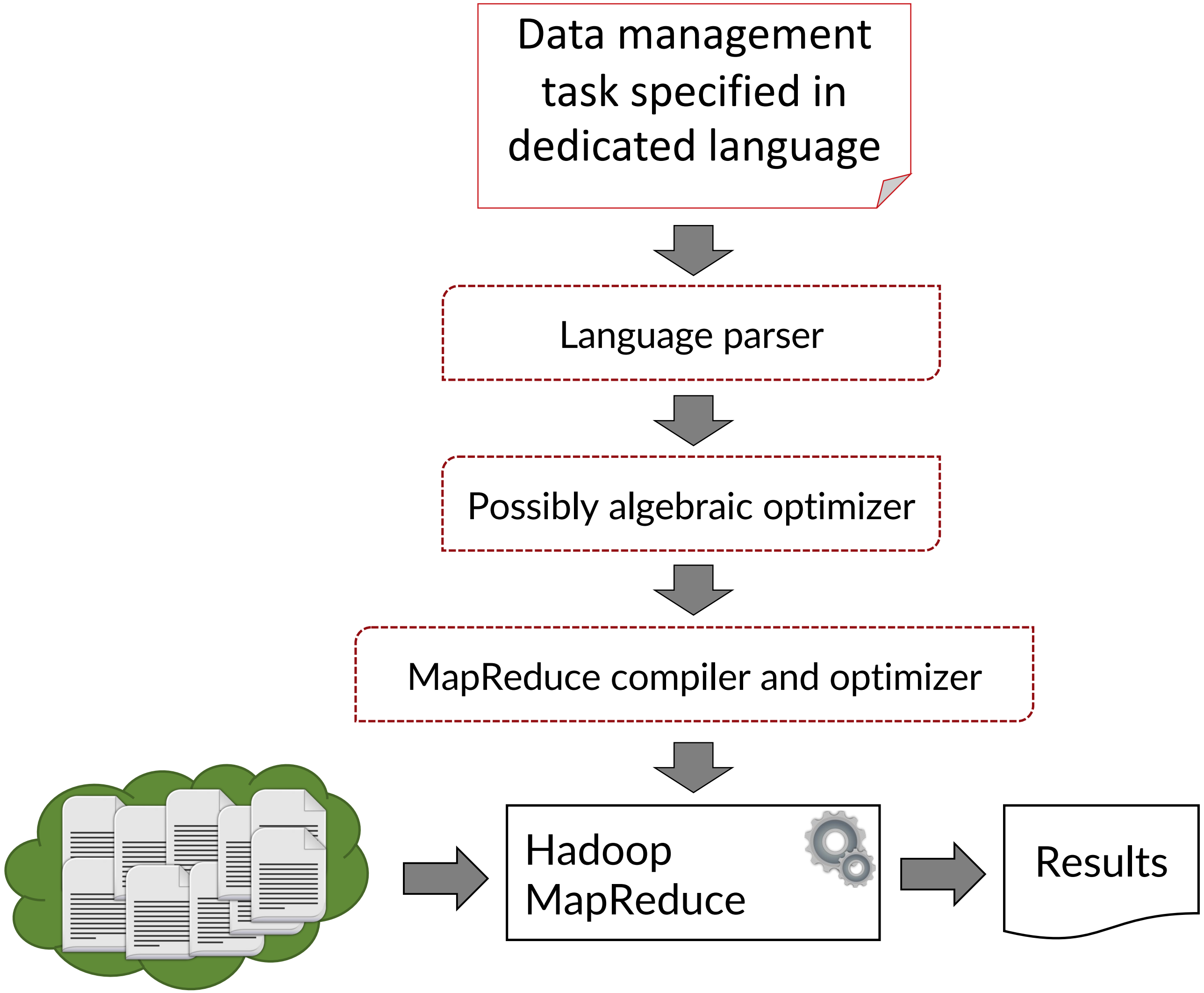
Recall: query processing pipeline in DBMS

SQL

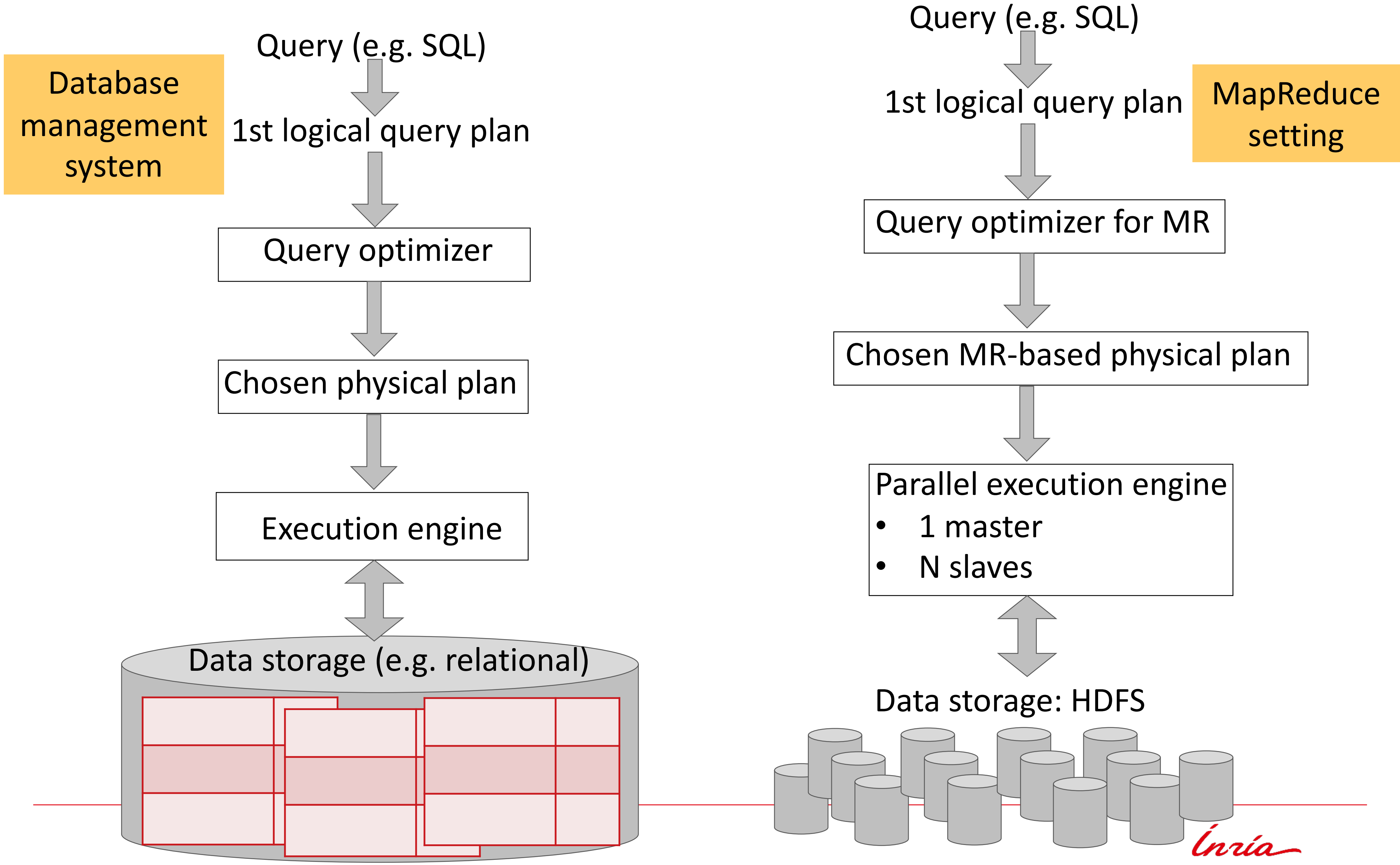
select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license=‘123AB’



Goal: query processing pipeline on top of MapReduce



Architecture: a MR program for every 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 (selection and projection)
- Binary operators (mostly: $\bowtie$ on equality, i.e. equijoin)
- N-ary operators (complex join expressions)

Implementing unary operators on MapReduce

Selection $\sigma_{pred}(R)$:

Map:
foreach t which satisfies pred in the input partition

- Output (hn(t.toString()), t); // hn fonction de hash

Reduce:

- Concatenate all the inputs

Projection $\pi_{cols}(R)$:

Map: foreach t

- Output (hn(t), π_{cols}(t))

Reduce:

- Concatenate all the inputs

Recall: basic physical operators for binary joins in a DBMS

Assume we need to join R, S on 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));
```

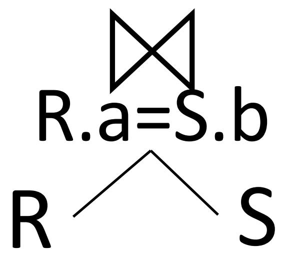
Hash join: // builds a hash table in memory

```
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 (~symetric hash)

Map:

foreach t in R

▶ Output (t.a , («R», t))

foreach t in S

▶ Output (t.b, («S», t))

Reduce:

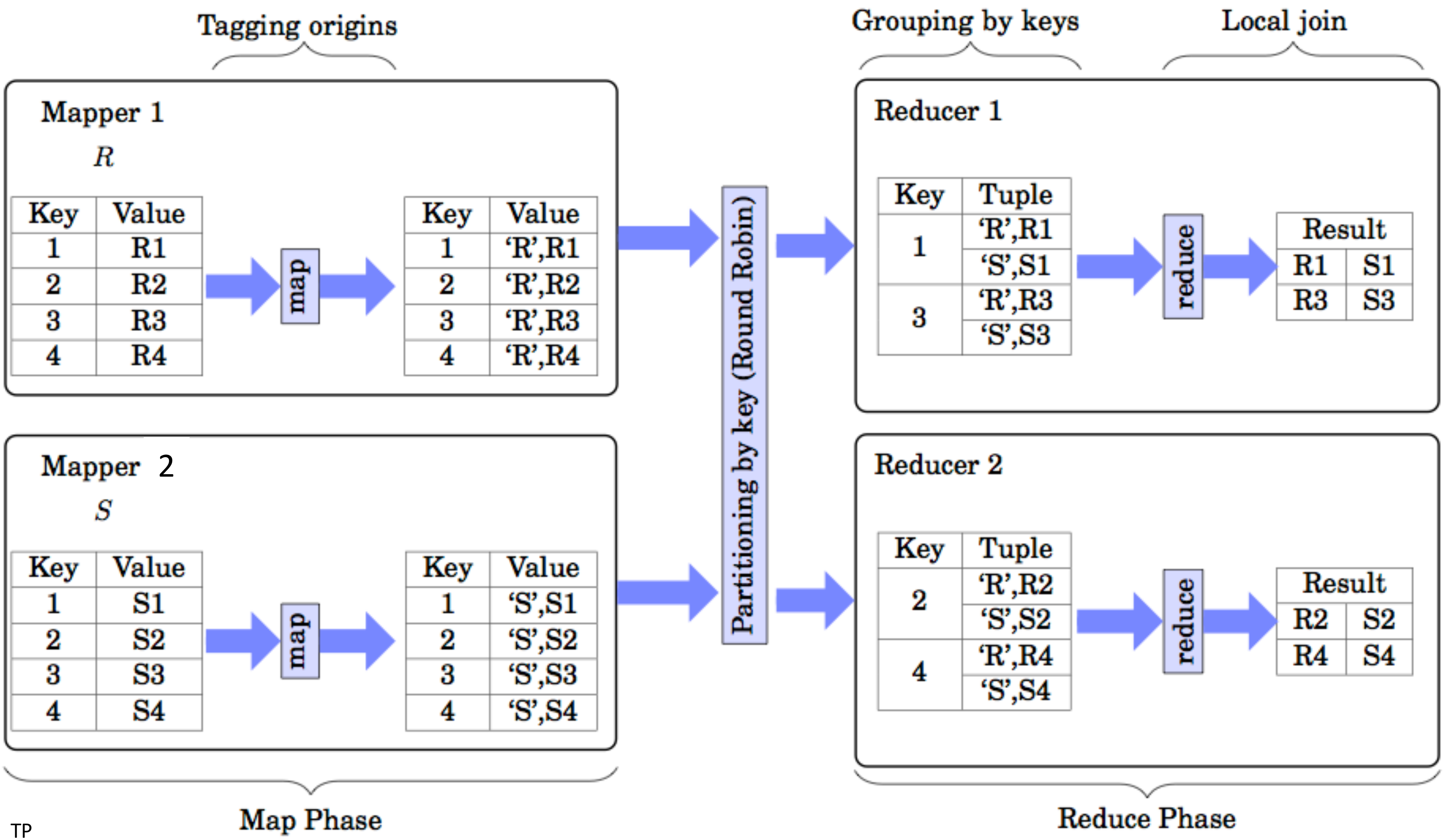
Foreach input key k

▶ Res_k = set of all R tuples on k × set of all S tuples on k

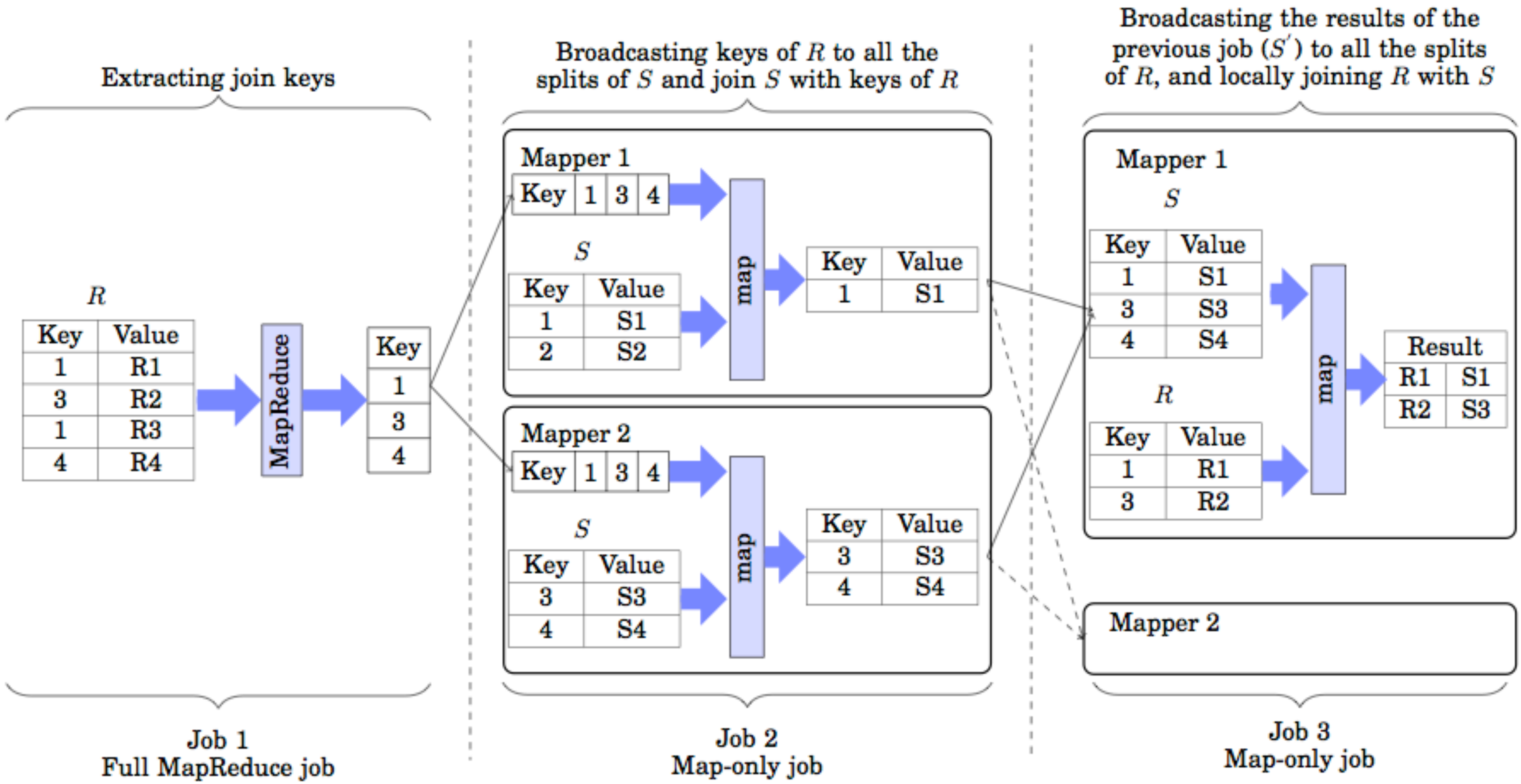
▶ Output Res_k

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

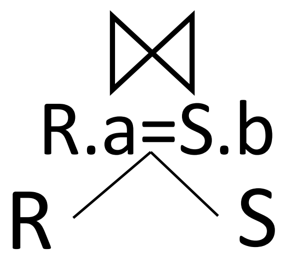
R(rID, rVal) join(rID = SID) S(sID, sVal)



Implementing equi-joins on MapReduce (2): Semijoin-based MapReduce join



Implementing equi-joins on MapReduce (3)



Broadcast (map-only) MapReduce join
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 »)

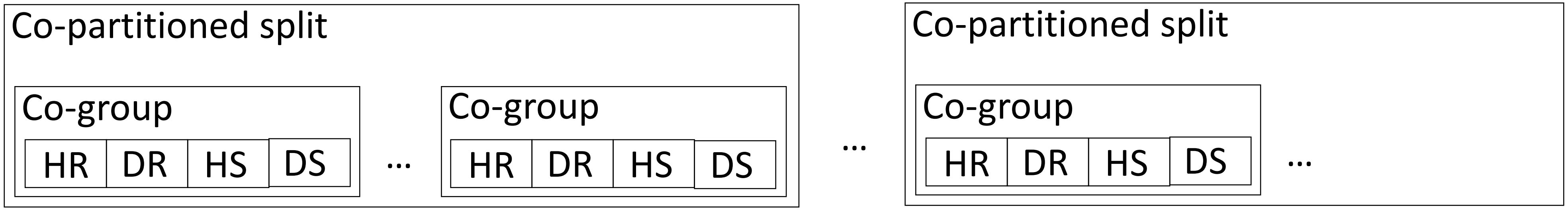
Implementing equi-joins on MapReduce (4)

Trojan Join [Dittrich 2010]

A Map task is sufficient for the join if relations are already co-partitioned by the join key

The split of R with a given join key is already next to the split 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?...

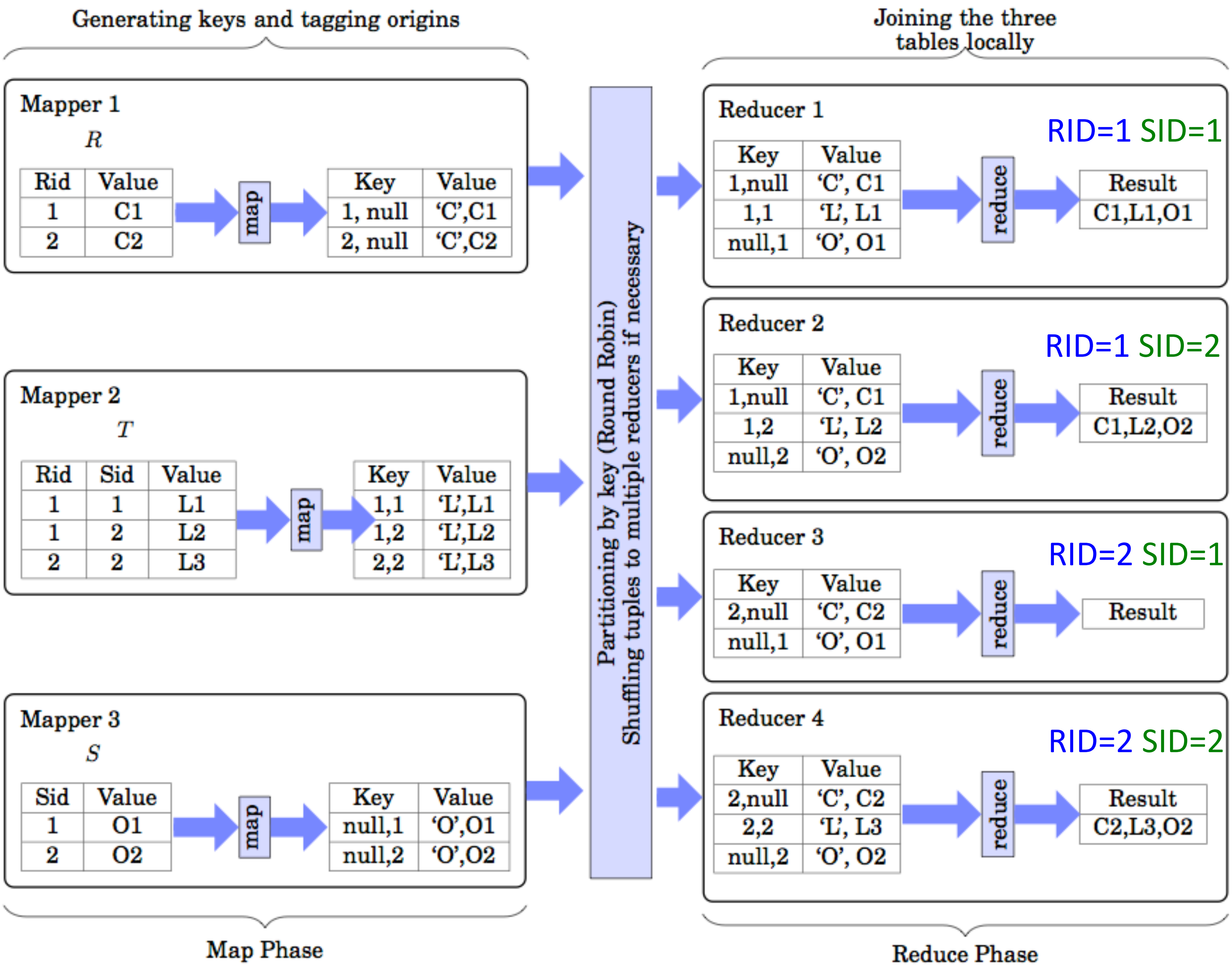
Three relations, two join attributes (RID and SID)

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.

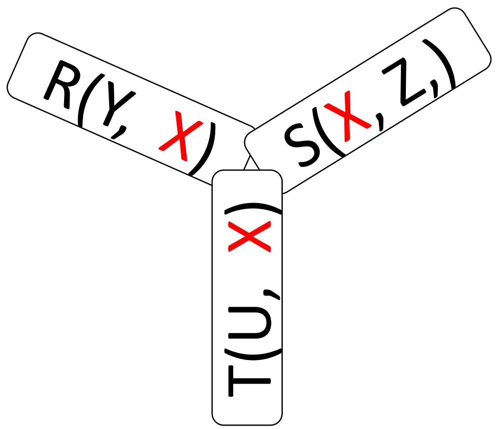
Hash R and S tuples on *partial keys* (RID, null) and (null, SID)

Distribute R and S tuples to each reducer where the non-null component matches (potentially multiple times!)


Generating keys and tagging origins
Joining the three tables locally
Mapper 1
R
Rid | Value
1 | C1
2 | C2
map
Key | Value
1, null | 'C',C1
2, null | 'C',C2
Mapper 2
T
Rid | Sid | Value
1 | 1 | L1
1 | 2 | L2
2 | 2 | L3
map
Key | Value
1,1 | 'L',L1
1,2 | 'L',L2
2,2 | 'L',L3
Mapper 3
S
Sid | Value
1 | O1
2 | O2
map
Key | Value
null,1 | 'O',O1
null,2 | 'O',O2
Partitioning by key (Round Robin)
Shuffling tuples to multiple reducers if necessary
Reducer 1
RID=1 SID=1
Key | Value
1,null | 'C', C1
1,1 | 'L', L1
null,1 | 'O', O1
reduce
Result
C1,L1,O1
Reducer 2
RID=1 SID=2
Key | Value
1,null | 'C', C1
1,2 | 'L', L2
null,2 | 'O', O2
reduce
Result
C1,L2,O2
Reducer 3
RID=2 SID=1
Key | Value
2,null | 'C', C2
null,1 | 'O', O1
reduce
Result
Reducer 4
RID=2 SID=2
Key | Value
2,null | 'C', C2
2,2 | 'L', L3
null,2 | 'O', O2
reduce
Result
C2,L3,O2
Map Phase
Reduce Phase

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?**
  - 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

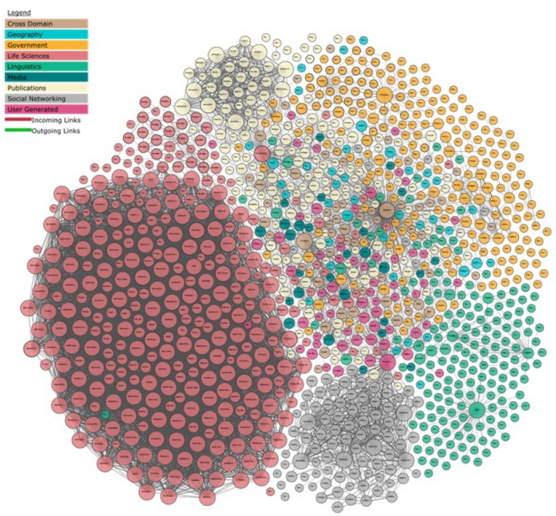
Language parser

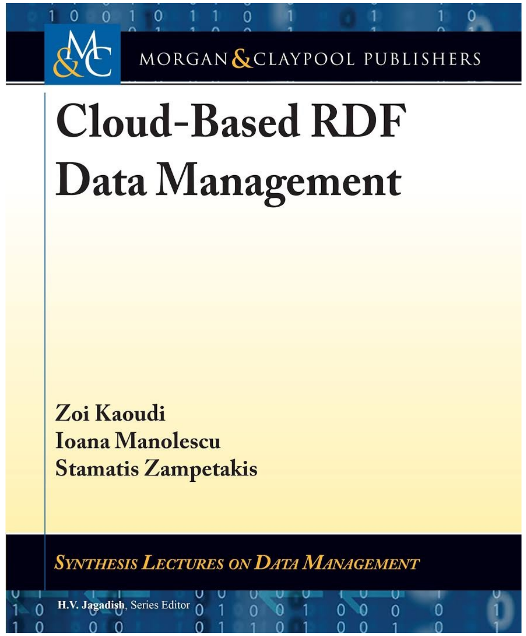
Possibly algebraic optimizer

MapReduce compiler and optimizer

Hadoop MapReduce

Results



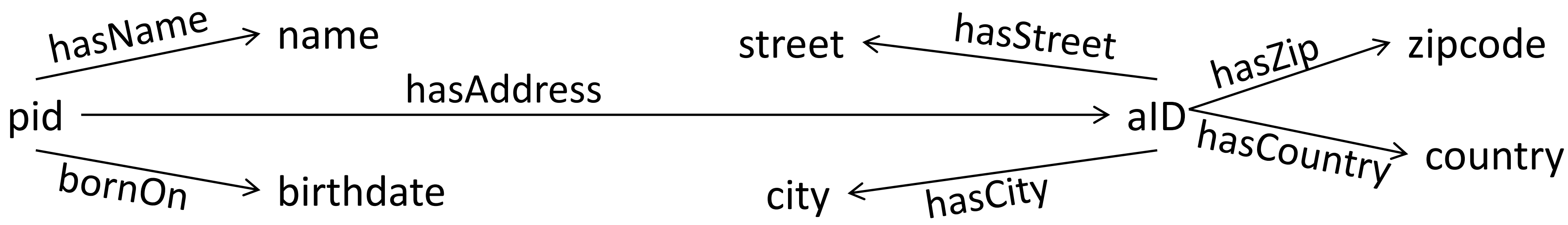


RDF query optimization for MapReduce

RDF queries need more joins than « equivalent » relational ones

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)



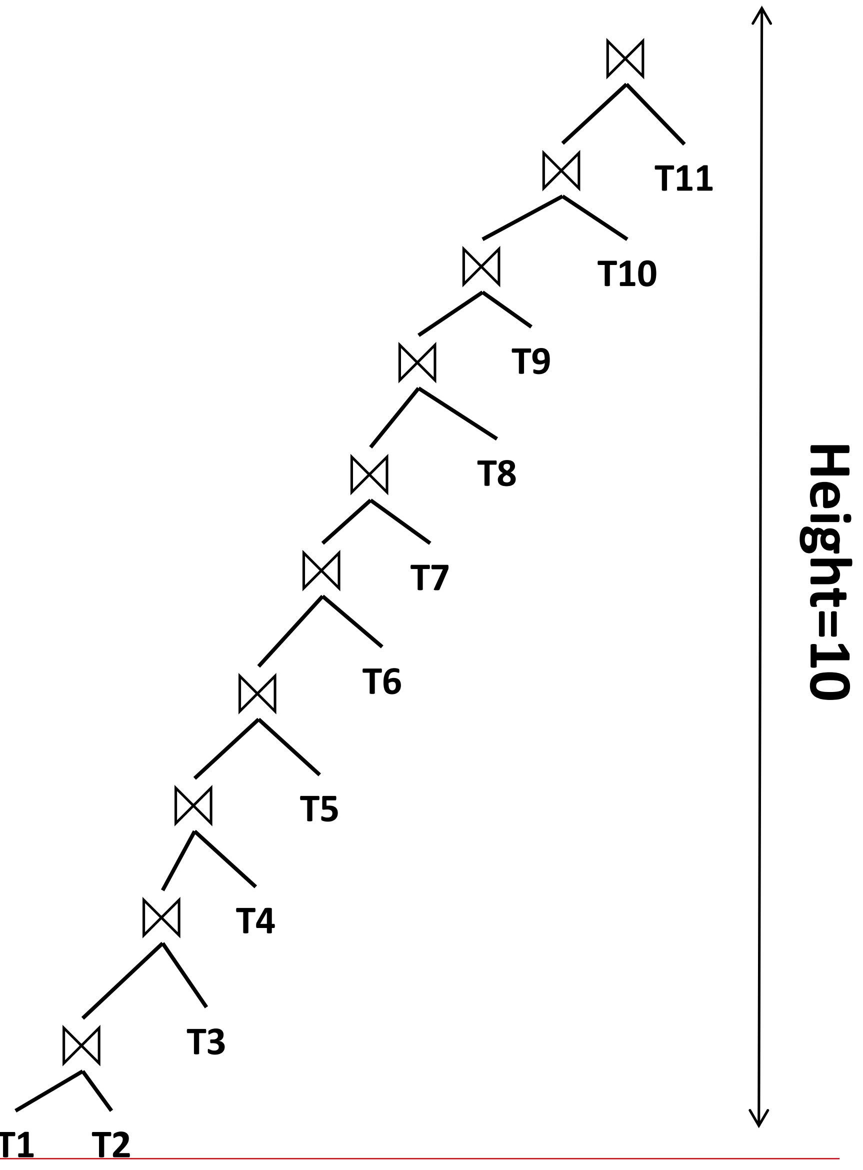
SPARQL query optimization is a stress test for MapReduce platforms

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 plan with binary joins:

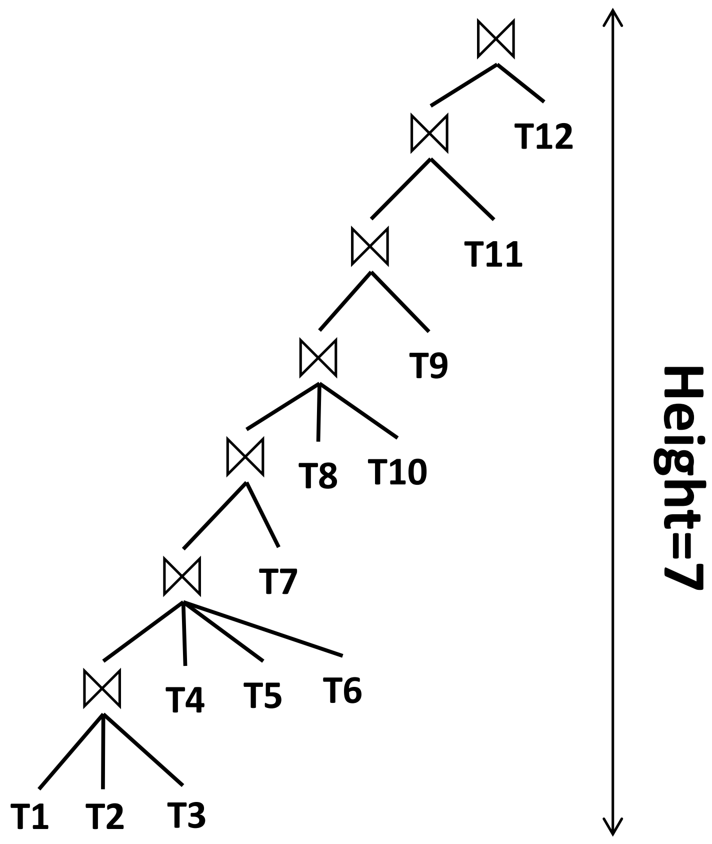


Query plans on MapReduce

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> .}

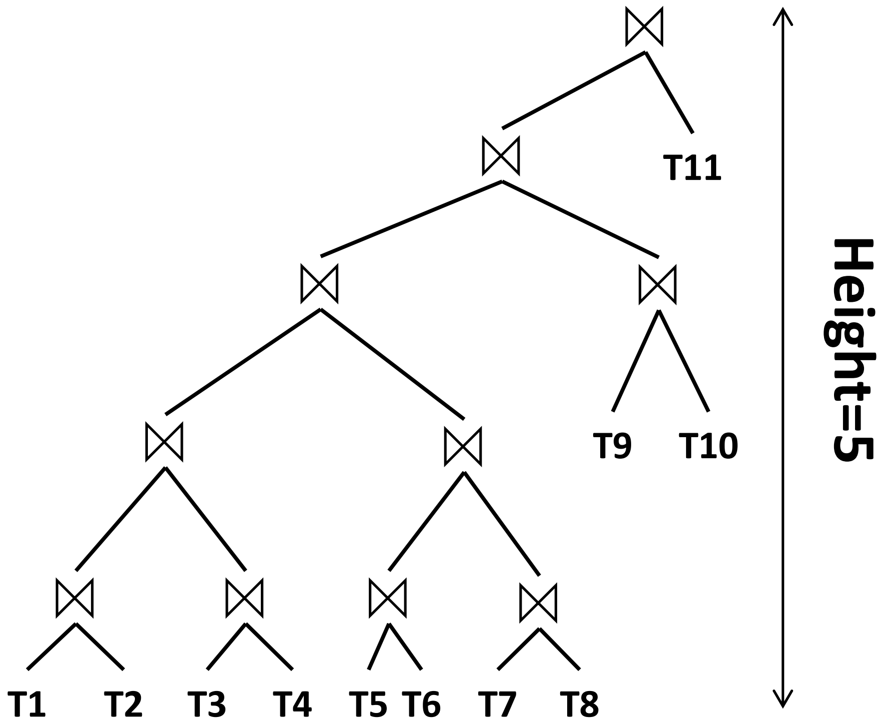
Left deep plan with n-ary joins:



Query plans on MapReduce

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> .}

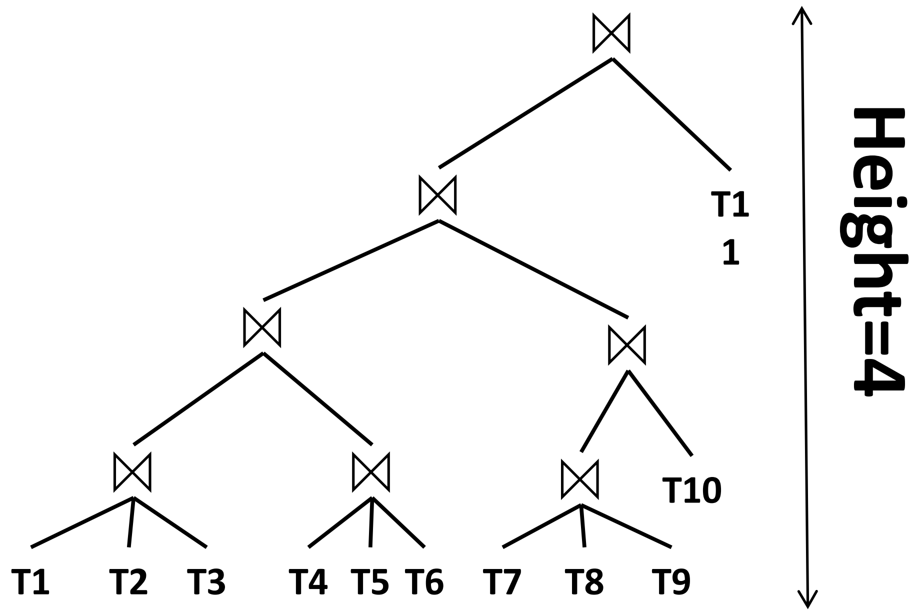
Bushy plan with binary joins:



Query plans on MapReduce

```
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> .}
```

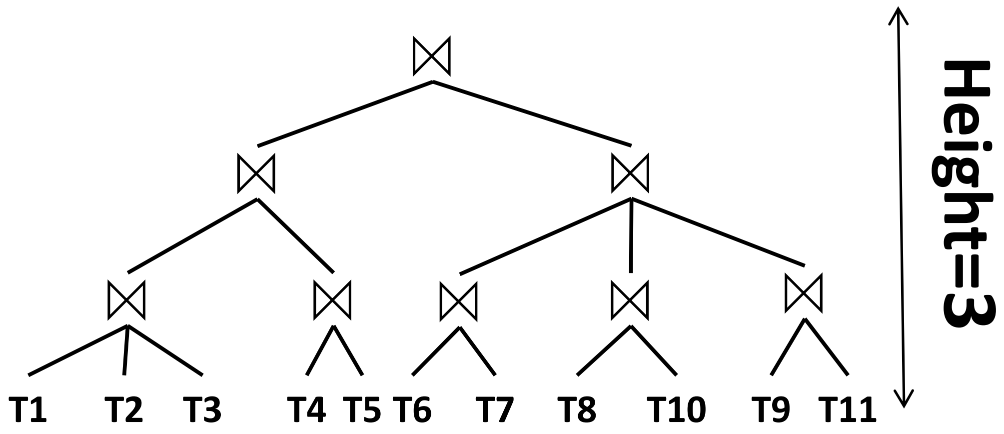
Bushy plan with n-ary joins only at leaves:



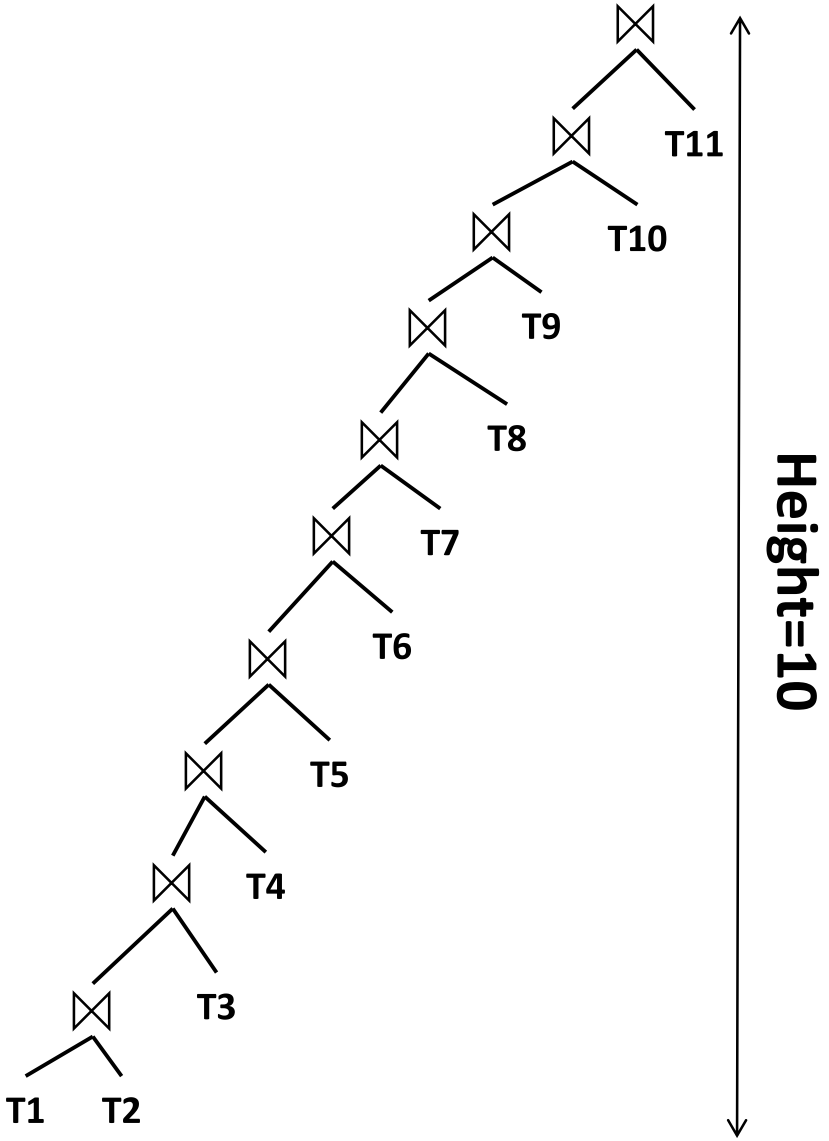
Query plans on MapReduce

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> .}

Bushy plan with n-ary joins:



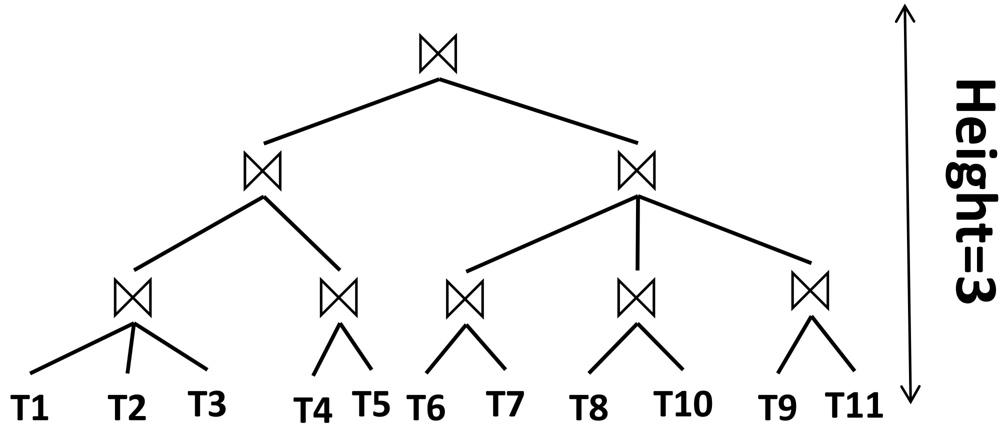
Query plans on MapReduce



Each join layer leads to one or more MR jobs (1 job = 1 map + 1 reduce)

The plan height = the number of successive jobs

Impacts execution time!



How to build flat plans with n-ary joins?

N-ary joins not studied in relational database management, because:

- Fewer joins in all, and thus, fewer star (n-ary) joins
- Limited memory to be shared between few binary joins → little interest in fitting n-ary joins…

Idea for SPARQL (Basic Graph Pattern) queries:

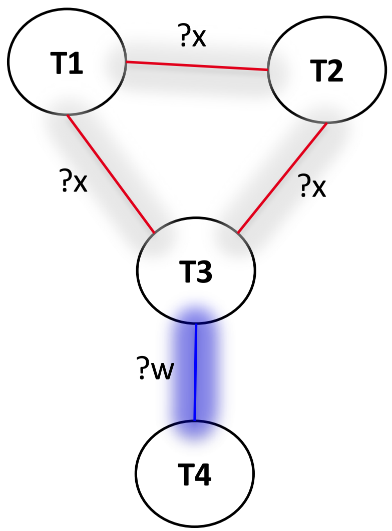
- Identify **cliques** = subsets of *n >= 2* triples sharing a common variable.
  - Pick a clique, use an *n*-ary join to combine these triples
  - Then find another clique in the query thus simplified, and similarly join them, etc.
  - …
- Until all triples have been joined

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

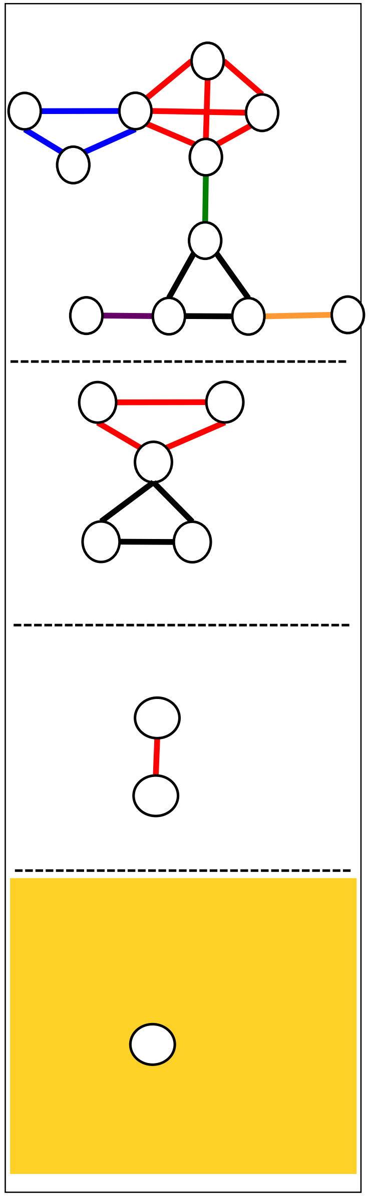


Variable graph

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

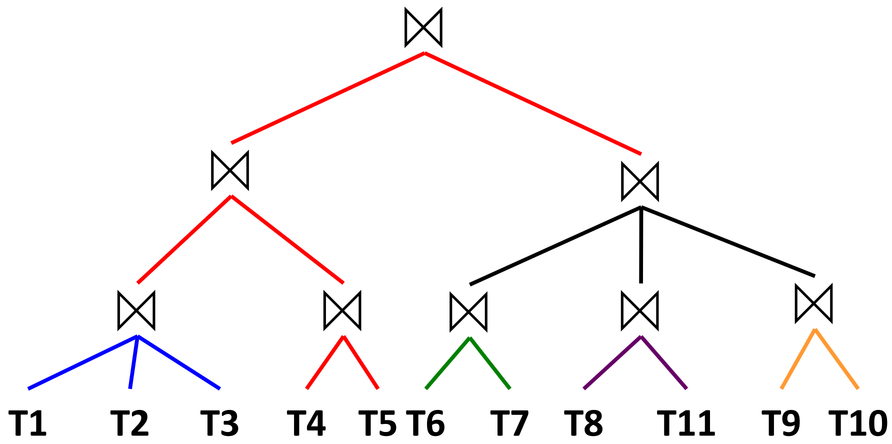
CliqueSquare: optimization with *n*-ary joins

States

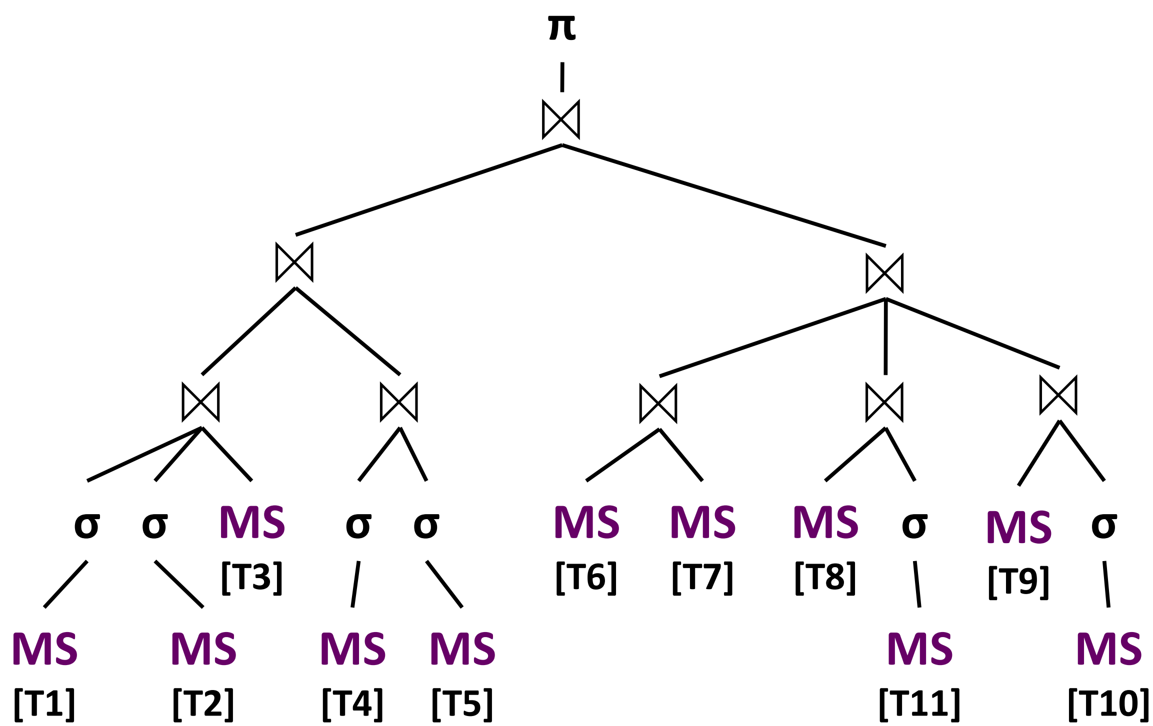


Each **node** of a graph corresponds to a **clique** of nodes of the previous graph.

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

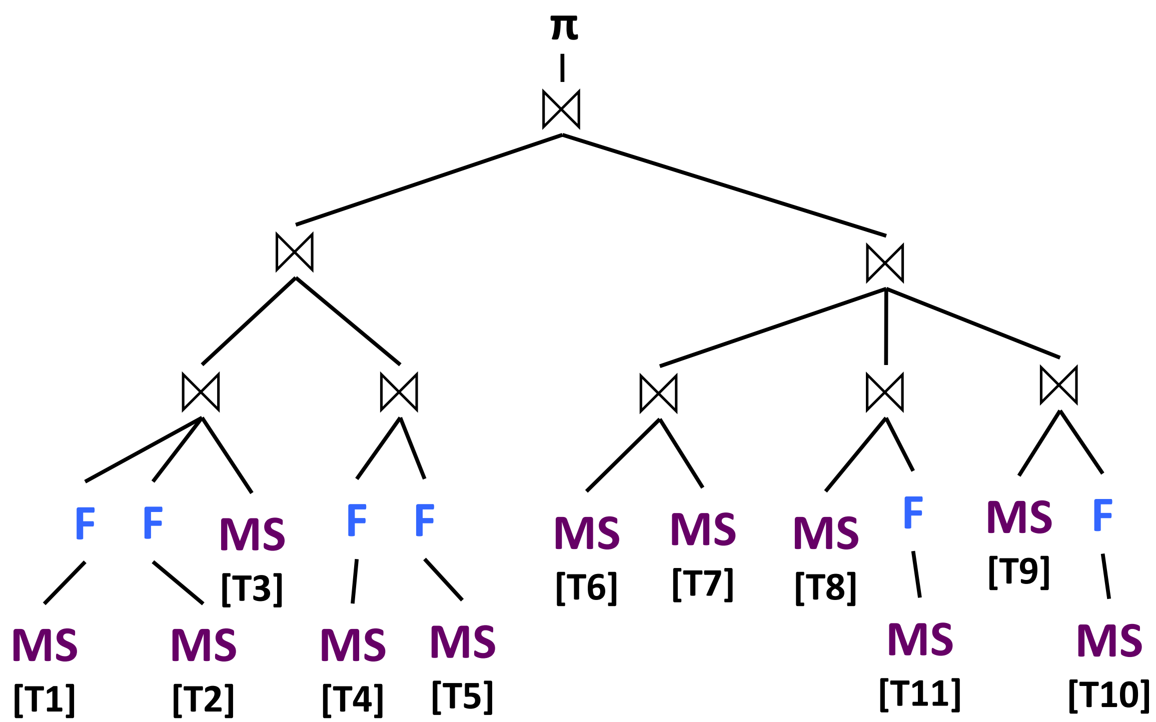


From the logical plan to a MapReduce physical plan



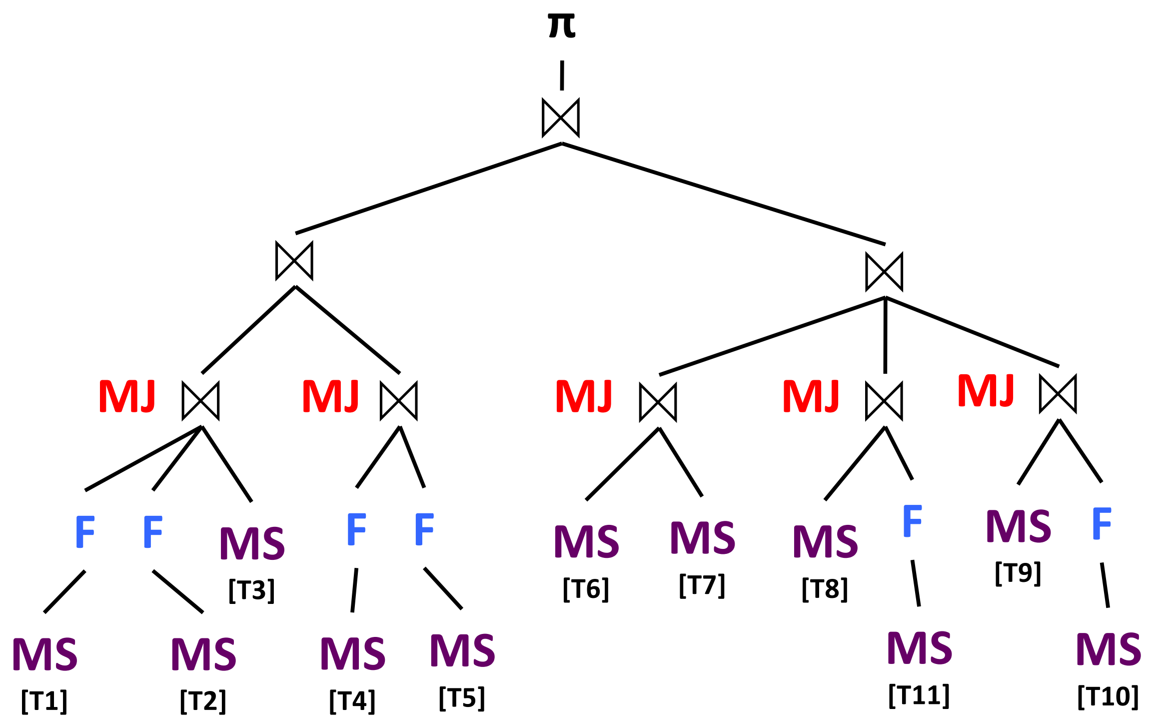
- Reading the triples from HDFS requires a Map Scan (**MS**) operator

From the logical plan to a MapReduce physical plan



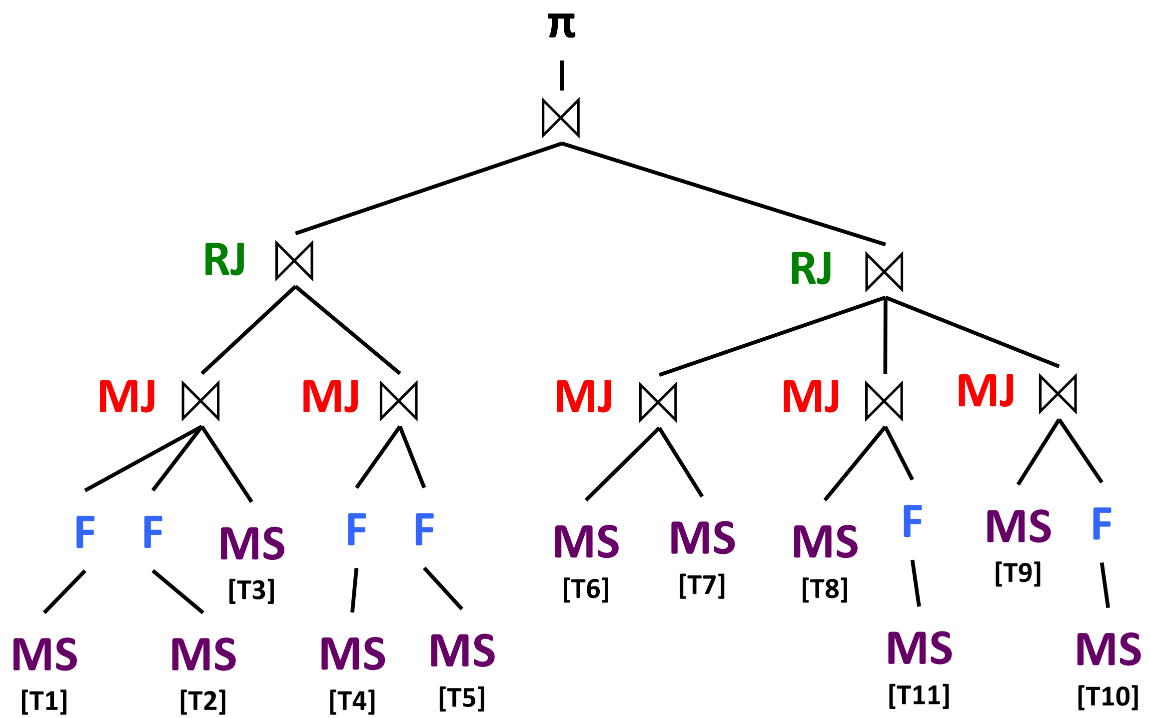
- Logical selections (**σ**) are translated to physical selections (**F**)

From the logical plan to a MapReduce physical plan



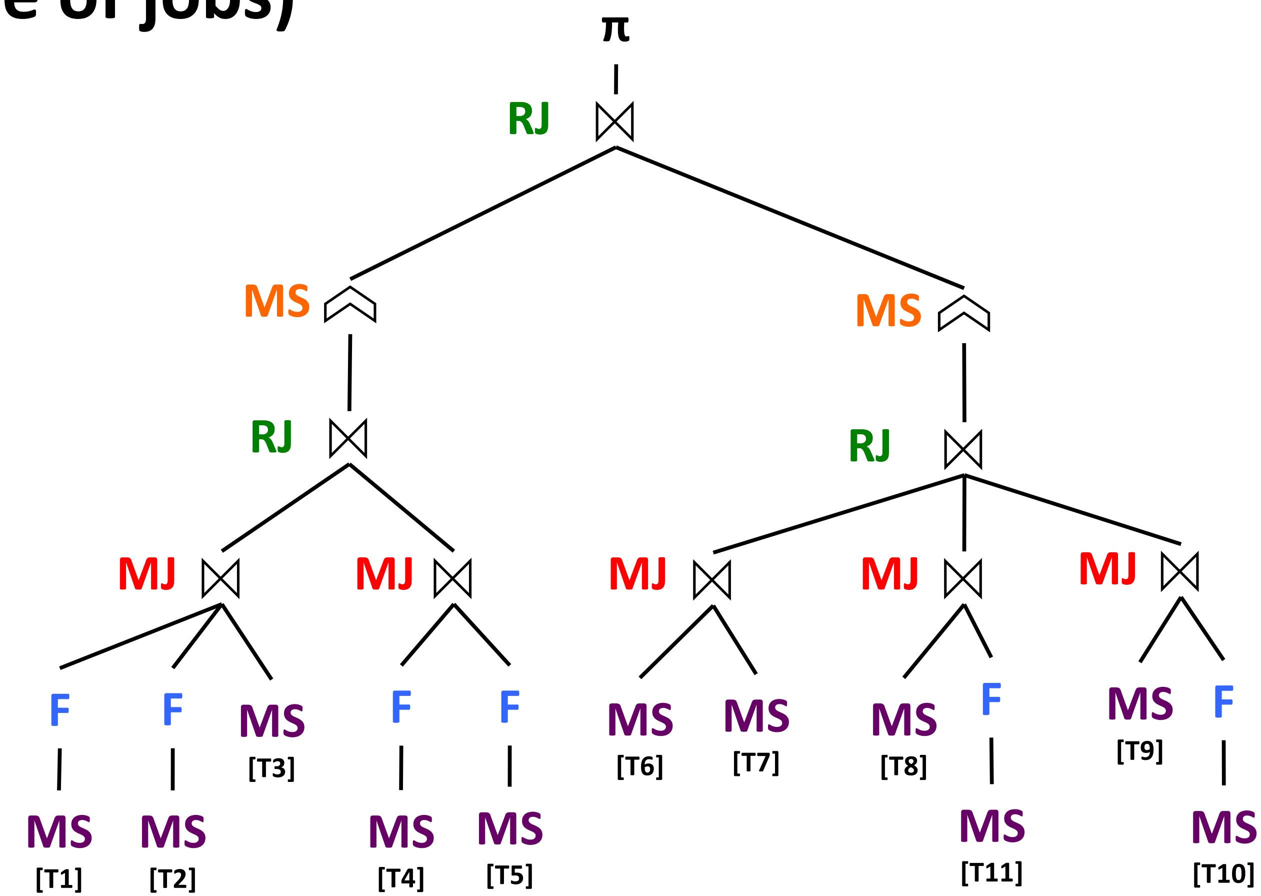
➢ First level joins are translated to Map side joins (**MJ**) taking advantage of the **data partitioning** (triples stored three times, hashed by subject, property, object)

From the logical plan to a MapReduce physical plan



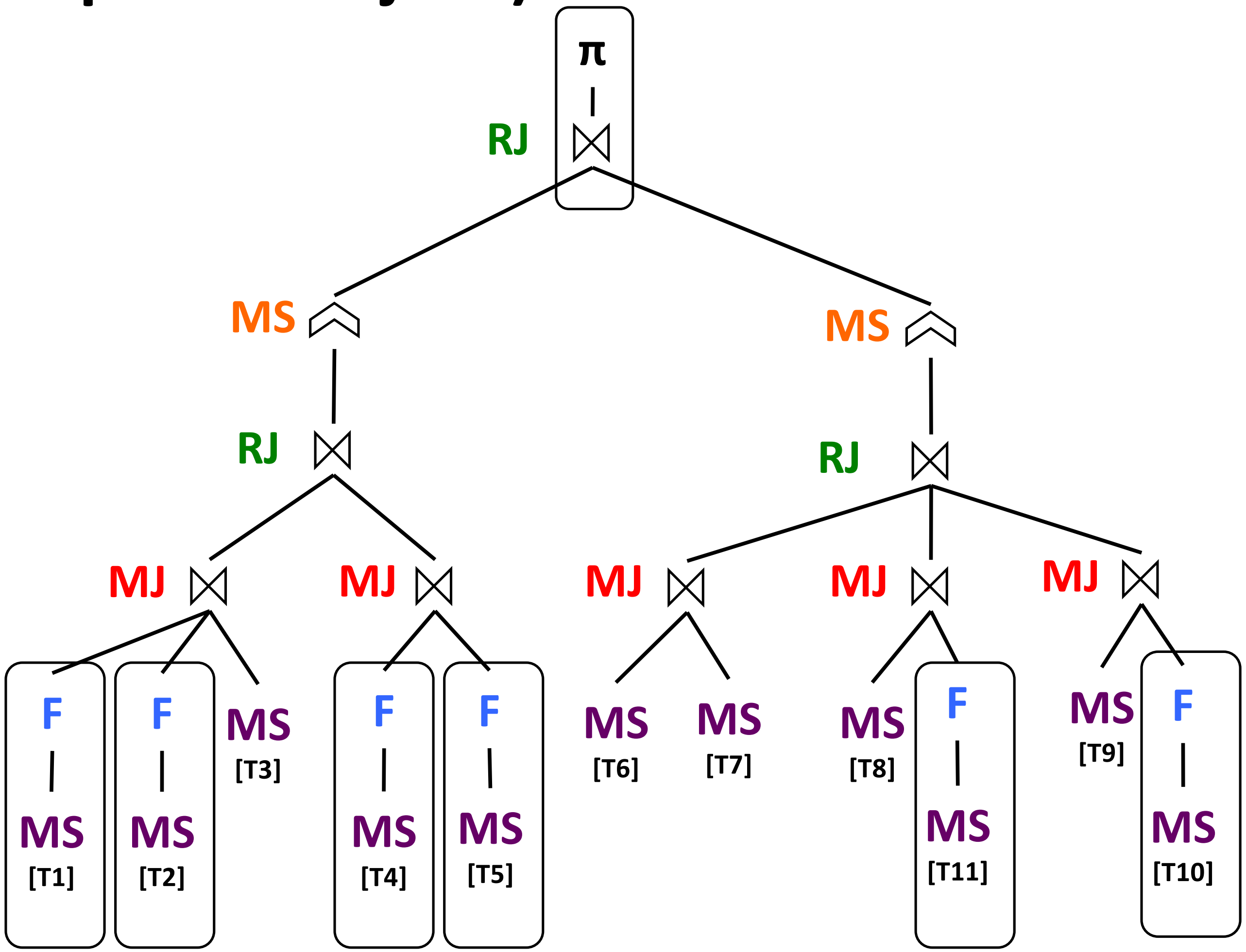
➢ All subsequent joins are translated to Reduce side joins (**RJ**)

From a MapReduce physical plan to a MapReduce program (sequence of jobs)



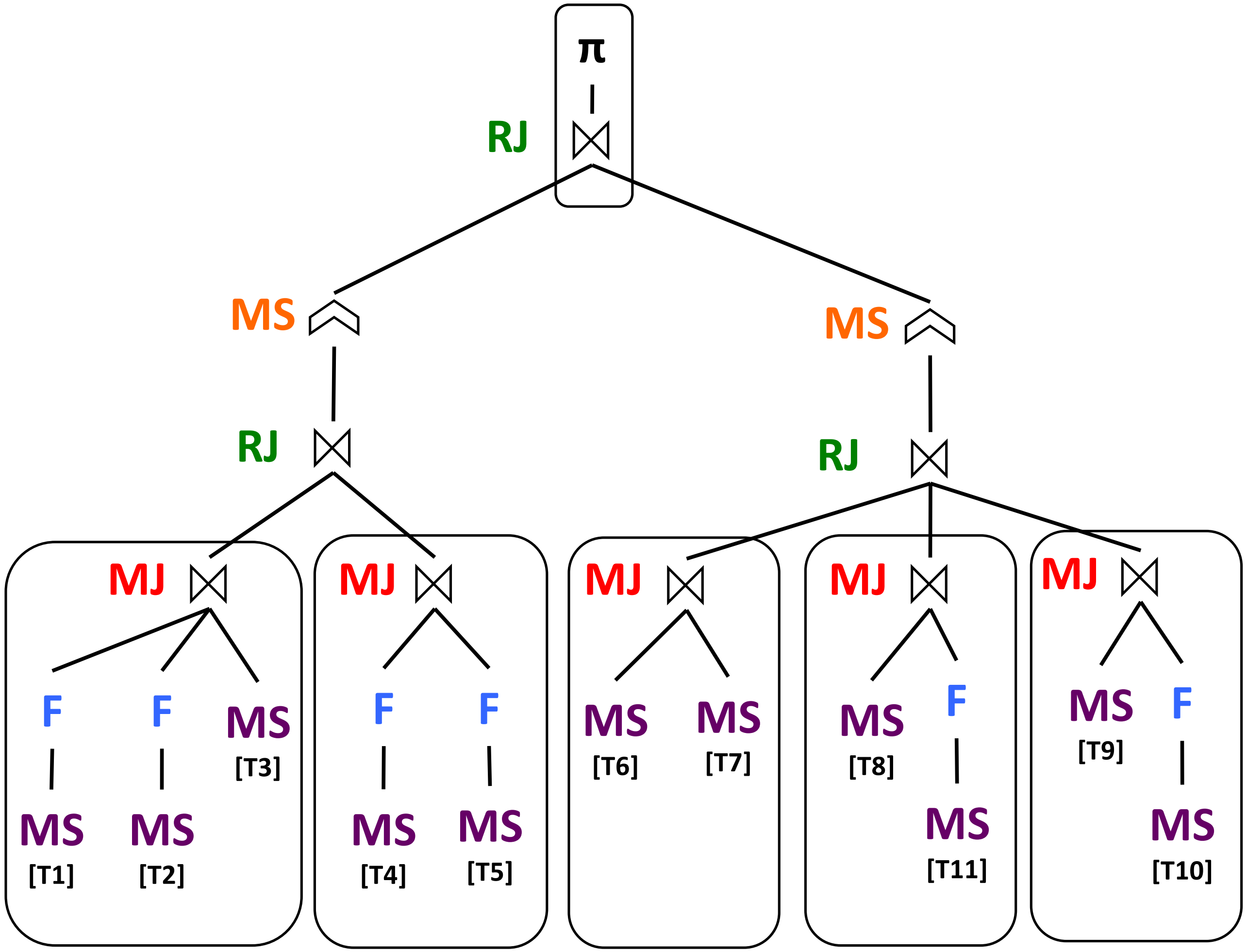
- Group the physical operators into Map/Reduce **tasks** and **jobs**

From a MapReduce physical plan to a MapReduce program (sequence of jobs)



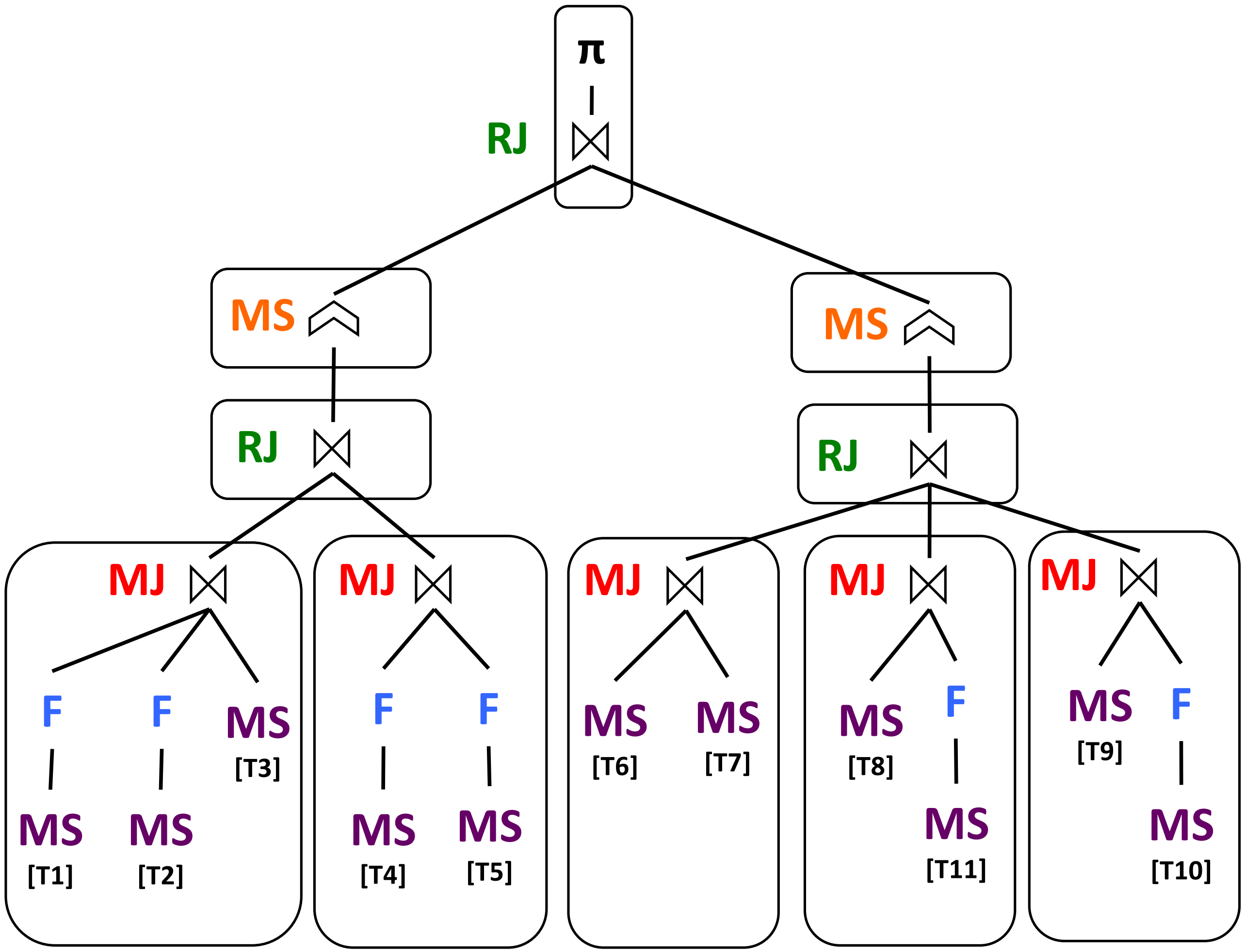
➢ Selections (F) and projections (π) belong to the same task as their child operator

From a MapReduce physical plan to a MapReduce program (sequence of jobs)



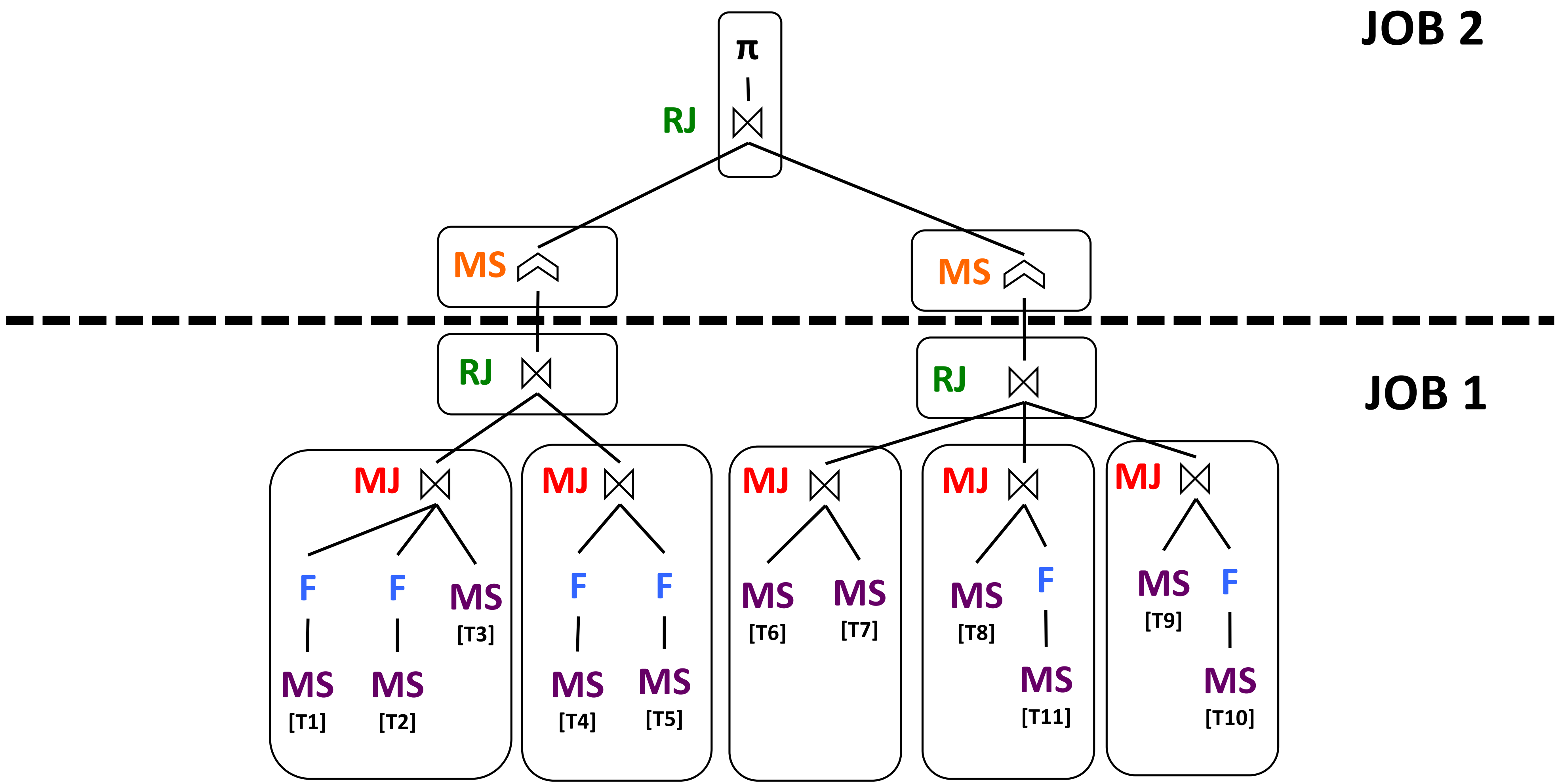
- Map joins (**MJ**) along with all their descendants are executed in the same task

From a MapReduce physical plan to a MapReduce program (sequence of jobs)



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

MapReduce program (jobs)



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

Questions?