FROM DATABASES TO ARCHITECTURES FOR BIG DATA MANAGEMENT

DATABASE FUNDAMENTALS (RECALL/INTRODUCTION)

Database functionalities

Which of these is/acts like a database?

From the user's perspective

MySQL	**Excel**	**Oracle**	**Hadoop**	**Google**	**GMail**
Facebook	**Twitter**	**Emacs**	**Skype**	**Firefox**	**Python**

Which of these is/acts like a database?

From the user's perspective

MySQL ✓	**Excel** ✗	**Oracle** ✓	**Hadoop** ✗	**Google**	**GMail**
Facebook ✓	**Twitter** ~	**Emacs** ✗	**Skype** ✗	**Firefox** ~	**Python** ~

Twitter, Skype and Firefox include / are built on database servers

- **Twitter**: no delete; small data items
- **Skype**: local database+index of all conversations, mirrorring the one from Microsoft. May get corrupted ☹
- **Firefox**: includes a tiny SQL server for the bookmarks

Fundamental database properties (1)

- **Data storage**
  - Protection against unauthorized access, data loss
- Ability to at least **add** to and **remove** data to the database
  - Also: **updates**; **active behavior** upon update (triggers)
- Support for **accessing** the data
  - Declarative query languages: say what data you need, not how to find it

Fundamental database properties: ACID

- **A****tomicity**: either all operations involved in a transactions are done, or none of them is
  - E.g. bank payment
- **C****onsistency**: application-dependent constraint

  E.g. every client has a single birthdate
- **I****solation**: concurrent operations on the database are executed as if each ran alone on the system
  - E.g. if a debit and a credit operation run concurrently, the final result is still correct
- **D****urability**: data will not be lost nor corrupted even in the presence of system failure during operation execution

Jim Gray, ACM Turing Award 1998 for « fundamental contributions to databases and transaction management »

ACID properties

- **Atomicity**: per transaction (cf. boundaries)
- **Consistency**: difference in the expressive power of the constraints
- Illustrated below for relational databases, **create table** statement:

CREATE TABLE tbl_name (create_definition,...) [table_options] [partition_options]

create_definition: *col_name column_definition* |
[**CONSTRAINT** [*symbol*]] **PRIMARY KEY** [*index_type*] (*index_col_name*,...) [*index_option*] ... |
{INDEX|KEY} [*index_name*] [*index_type*] (*index_col_name*,...) [*index_option*] ... |
[**CONSTRAINT** [*symbol*]] **UNIQUE** [**INDEX**|**KEY**] [*index_name*] [*index_type*]
(*index_col_name*,...) [*index_option*] (…) |
CHECK (*expr*)

column_definition: *data_type* [**NOT NULL** | **NULL**] [DEFAULT *default_value*]
[AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY] (…)

ACID properties

Consistency (continued)

- SQL constraint syntax (within create table):

[**CONSTRAINT** [*symbol*]] **FOREIGN KEY** [*index_name*]
(*index_col_name*, ...)
    REFERENCES *tbl_name* (*index_col_name*,...)
    [**ON DELETE** *reference_option*]
    [**ON UPDATE** *reference_option*]
reference_option: **RESTRICT | CASCADE | SET NULL | NO ACTION**

- Key-value store: REDIS
  - a data item can have only one value for a given property
- Key-value store: DynamoDB
  - The value of a data item can be constrained to be unique, *or* allowed to be a set
- Hadoop File System (HDFS): no constraints

ACID properties

- **Isolation**: concurrent operations on the database are executed as if each ran alone on the system
  - Watch out for: read-write (RW) or write-write (WW) conflicts
  - Conflict granularity depends on the data model
- **An example of advanced isolation support: SQL**
  - E.g. SQL

Isolation Level	Dirty Read	Non Repeatable Read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Snapshot	No	No	No
Serializable	No	No	No

  - High isolation conflicts with high transaction throughput
  - E.g. HDFS: a file is never modified (written only once and integrally)

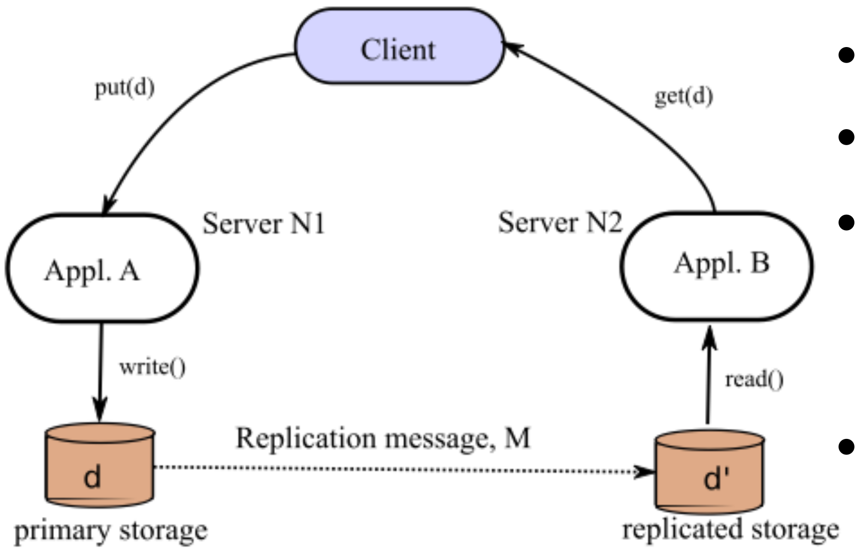
Limits of ACIDity in large distributed systems: the **CAP theorem**

- Eric Brewer, « Symposium on Principles of Distributed Computing », 2000 (conjecture)
- Proved in 2002

- No distributed system can simultaneously provide

1. **Consistency** (all nodes see the same data at the same time)
2. **Availability** (node failures do not prevent survivors from continuing to operate)
3. **Partition tolerance** (the system continues to operate despite arbitrary message loss)

CAP theorem by example



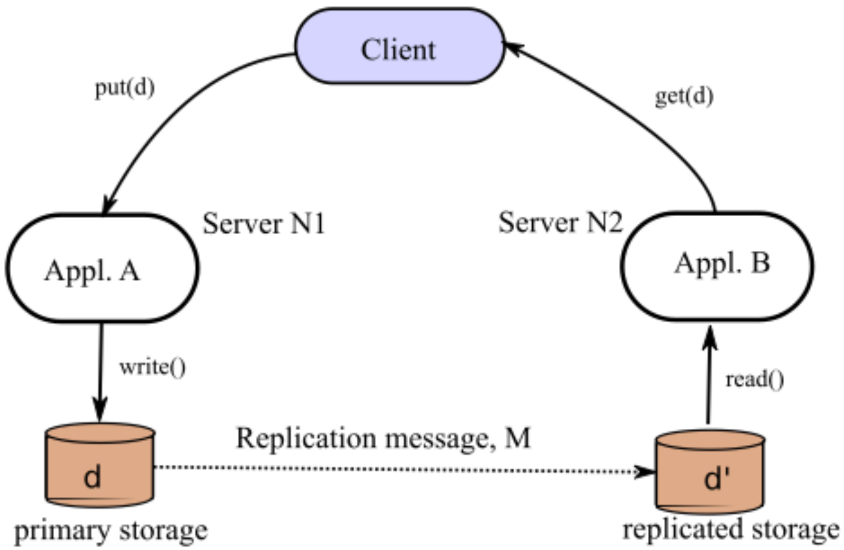
- Primary and replica store
- Applications A and B on servers
- Client writes a new d value through A, which propagates d to the replica (replacing the old d')
- Subsequently, client reads from B

What if a failure occurs in the system?

Communication missed between primary and replica

1. If we want **Partition tolerance** (let the system function) → the Client reads old data (**no Consistency**)
2. If we want **Consistency**, e.g. make the write+replica msg an atomic transaction (to avoid missed communications) → **no Availability** (we may wait for the msg forever if failure)

CAP theorem: what can we do?

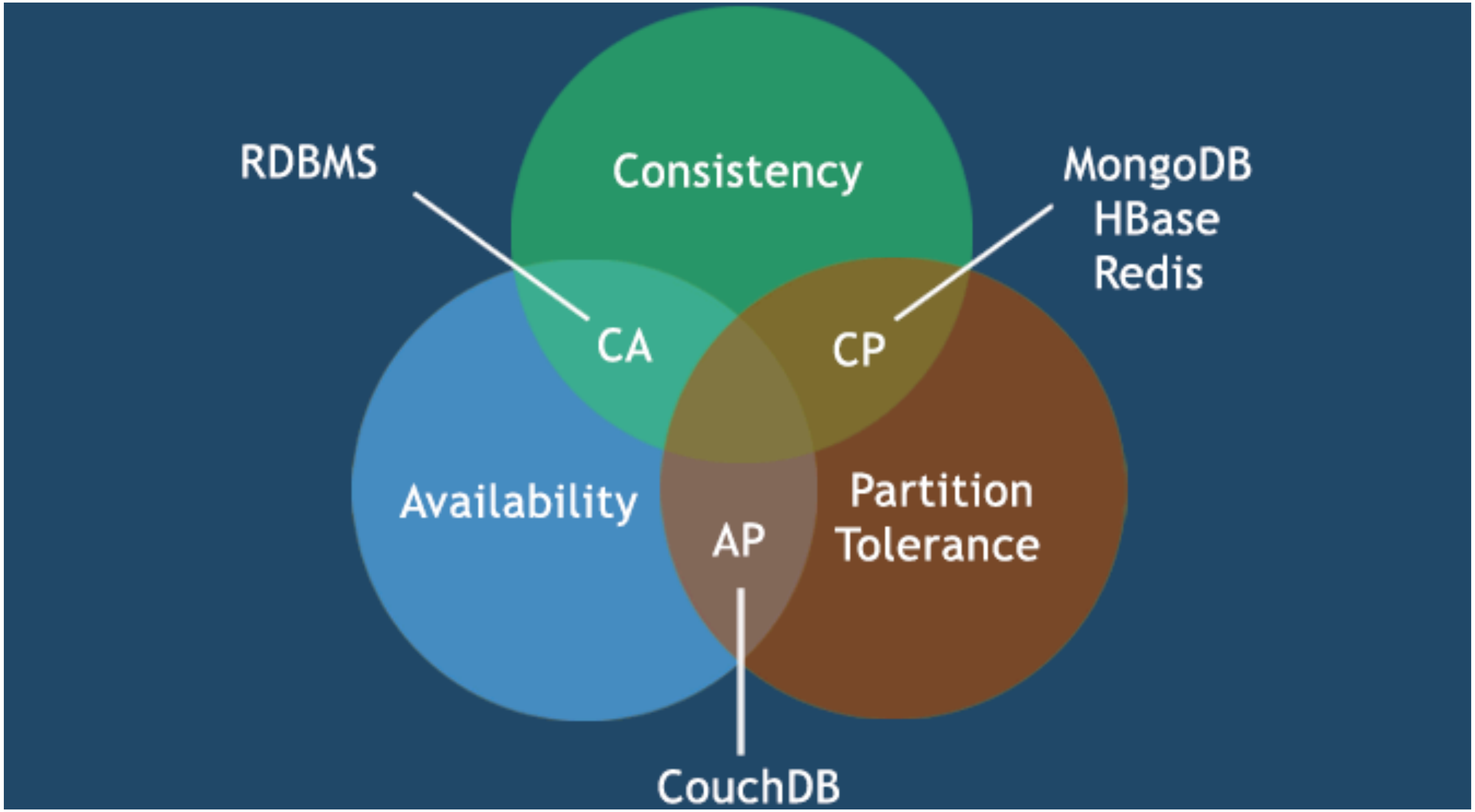


- **Partition tolerance**: we must have it (cannot block if one machine fails)
- Then one must *trade some consistency for availability*

Eventual consistency model:

- The replication message is asynchronous (non-blocking)
- N1 keeps sending the message until acknowledge by N2 (*eventually* the replica and primary store are consistent)
- In the mean time, the client works on inconsistent data (« I had already removed this from the basket once! »)

NoSQL systems vs. CAP theorem



Modern systems (e.g. NoSQL) arose exactly because partition tolerance is a must in large-scale distributed systems

More on CAP theorem

- ACID properties focus on consistency: business databases (sales, administration...)
- **BASE**: Basically Available, Soft state, Eventually consistent
  – Modern NoSQL systems are typically BASE
- "Partition" in fact corresponds to a **timeout** (when do we decide that we waited enough)
  – Different nodes in the system may have different opinion on whether there is a partition
  – Each node can go in "partition mode"

Choices in the ACID-BASE spectrum

- Yahoo! PNUTS: give up strong consistency to avoid high latency. The master copy is always "nearby" the user
- Facebook: the master copy is always remote, however updates go directly to the master copy and *this is also where users' reads go for 20 seconds*. After that, the user traffic reverts to the closer copy.

What do to in case of inconsistency?

- Merge copies: find a commonly agreed upon version
- Concurrent Versioning System (CVS) does this pretty well but not always (some conflicts remain to be solved by the user)
- Google Docs always solves conflicts by allowing only style change and add/delete text
  - Retrieving consistency is easier if the operations set is limited
  - E.g., using only commutative operations: there is always a way to rearrange a set of operations in a preferred consistent global order
    - Addition is commutative
    - Addition with a bounds check is not

DATABASE FUNDAMENTALS (RECALL/INTRODUCTION)

Database internals

(We illustrate for relational databases, as they are the most mature)

What's in a database?

SQL

Database

Results

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

name
Julie

What's in a database?

Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	‘123AB’
2	‘171KZ’

1. Load

2. SQL

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

Database

3. Results

name
Julie

What's in a database?

Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	‘123AB’
2	‘171KZ’

1. Load

2. SQL

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

Database

Storage system (**disk**, memory, SSD…)

3. Results

name
Julie

What's in a database?

1. Load

2. SQL

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

Database

Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	‘123AB’
2	‘171KZ’

3. Results

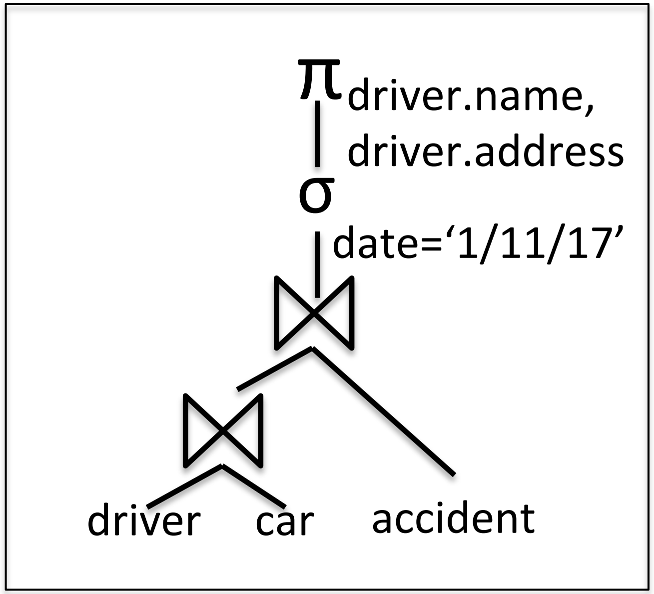
name
Julie

What's in a database?

SQL

select driver.name, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date='1/11/13'

select… from driver, car, accident where…



……………………………

Driver

name	ID
Julie	1
Damien	2

Accident

Car

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

Query language

Logical plan

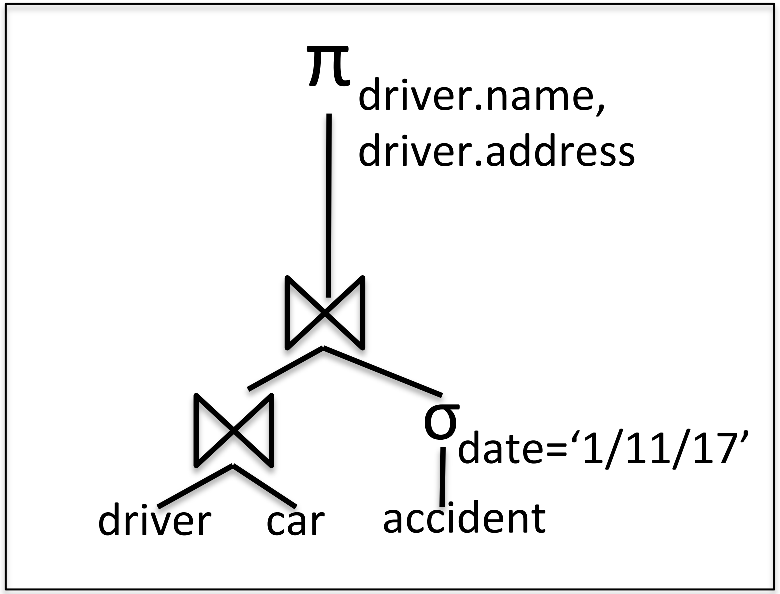
Results

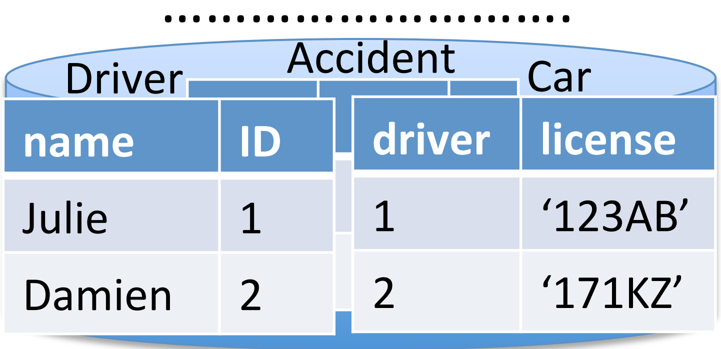
What's in a database?

SQL

select driver.name, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date='1/11/13'

select… from driver, car, accident where…





Query language

Logical plan 1

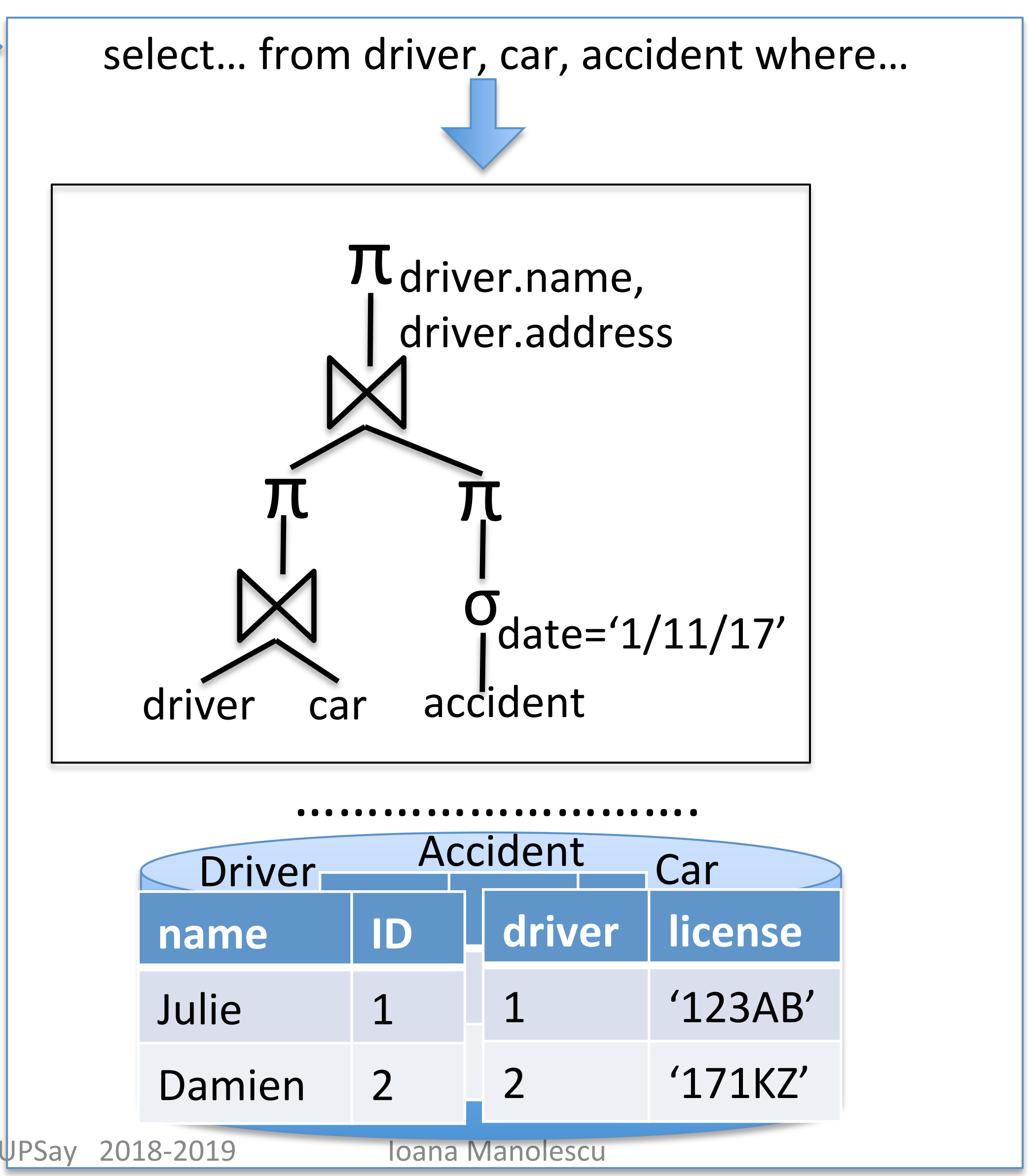
Logical plan 2

Results

What's in a database?

SQL

select driver.name, driver.address **from** driver, car, accident **where** driver.ID=car.driver and car.license=accident.carLicense and accident.date='1/11/13'



Query language

Logical plan 1

Logical plan 2

Logical plan 3

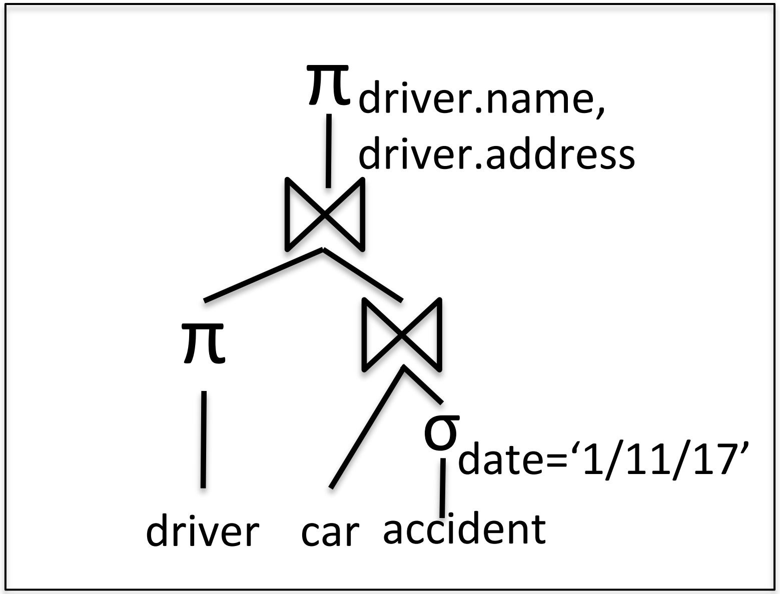
Results

What's in a database?

SQL

select driver.name, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date='1/11/13'

select… from driver, car, accident where…

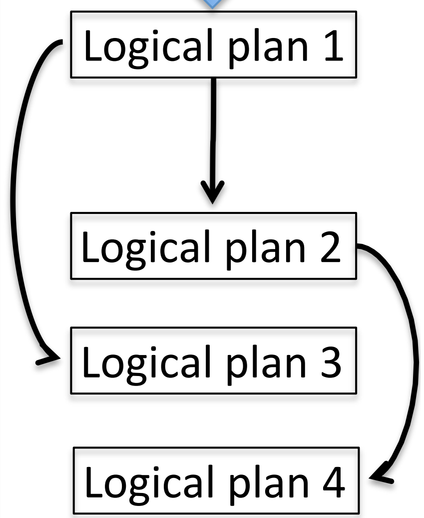


……………………………

Driver | Accident | Car

name	ID	driver	license
Julie	1	1	'123AB'
Damien	2	2	'171KZ'

Query language



Results

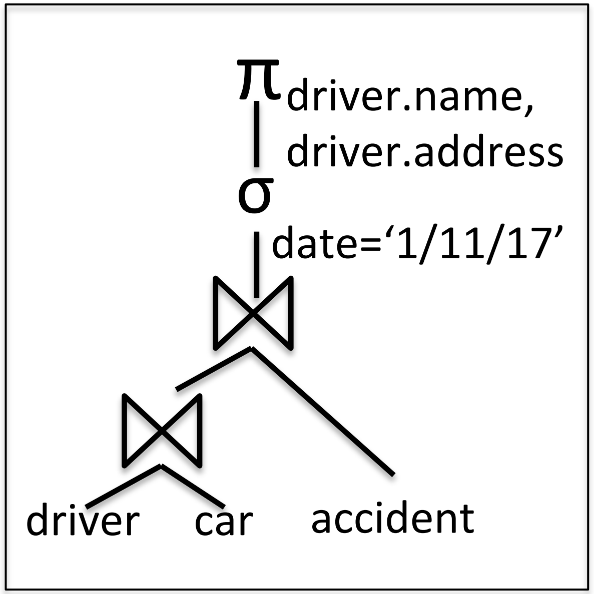
Logical query optimization

- Enumerates logical plans
- All logical plans compute the query result
  - They are **equivalent**
- Some are (much) more **efficient** than others
- **Logical optimization**: moving from a plan to a more efficient one
  - Pushing selections
  - Pushing projections
  - Join reordering: most important source of optimizations

Logical query optimization example

1.000.000 cars, 1.000.000 drivers, 1.000 accidents, 2 cars per accident, 10 accidents on 1/11/17

« Name and address of drivers in accidents on 1/11/2017? »



$10^{12} + 10^{9} + 2000 + 20$ operations ~ $\mathbf{10^{12}} + 10^{9}$

Cost of an operator: depends on the number of tuples (or tuple pairs) which it must process

e.g. c_disk x number of tuples read from disk

e.g. c_cpu x number of tuples compared

Cardinality of an operator's output: how many tuples result from this operator produce

The cardinality of one operator's output determines the cost of its parent operator!

Plan cost: the sum of the costs of all operators in a plan

Total scan costs: $10^6+10^6+10^3$

Driver-car join cost estimation: $10^6 \times 10^6 = 10^{12}$

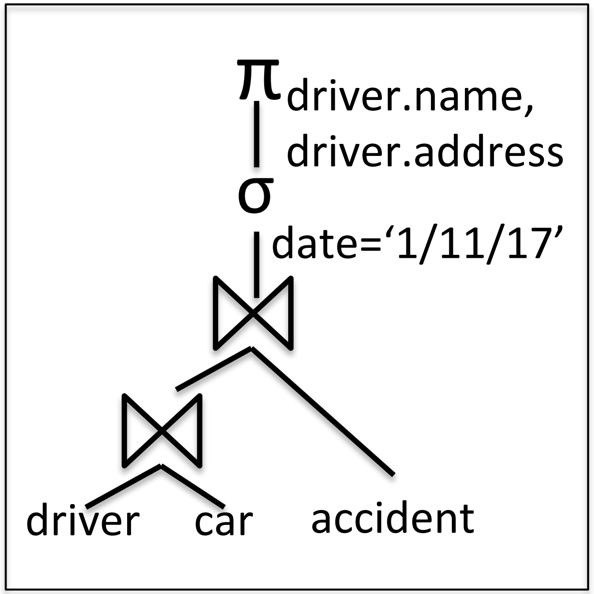
Driver-car join size estimation: 10^9

Driver-car-accident cost estimation: $10^9 \times 10^3 = 10^{12}$

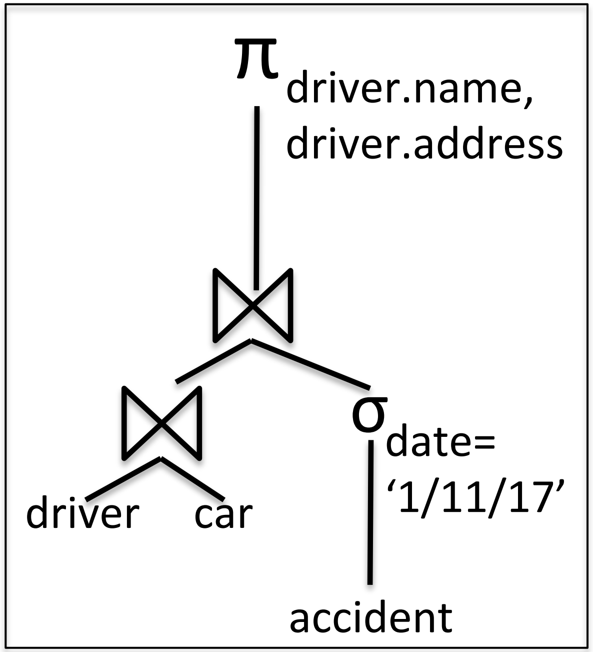
Logical query optimization example

1.000.000 cars, 1.000.000 drivers, 1.000 accidents, 2 cars per accident, 10 accidents on 1/11/17

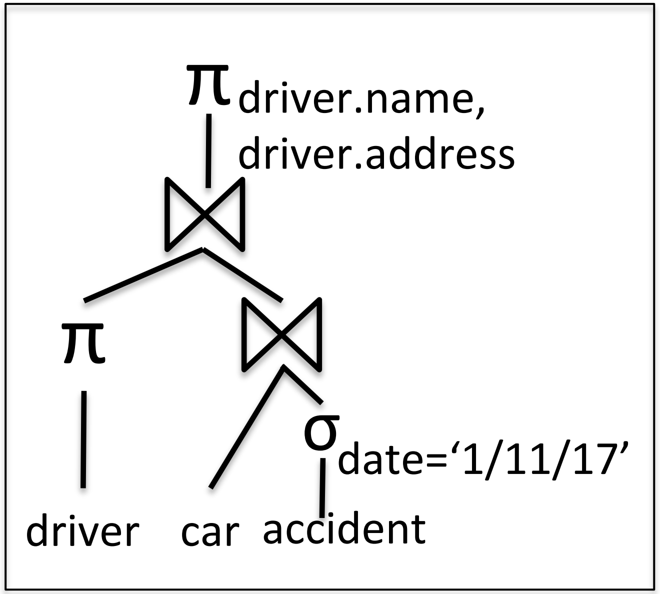
« Name and address of drivers in accidents on 1/11/2017? »



$10^{12} + 10^9 + 2000 + 20$ operations ~ $\mathbf{10^{12}} + 10^9$



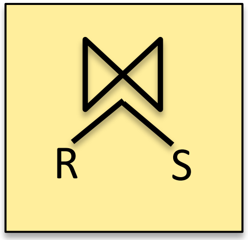
$10^{12} + 1000 + 10^7 + 20$ operations ~ $\mathbf{10^{12}} + 10^3$



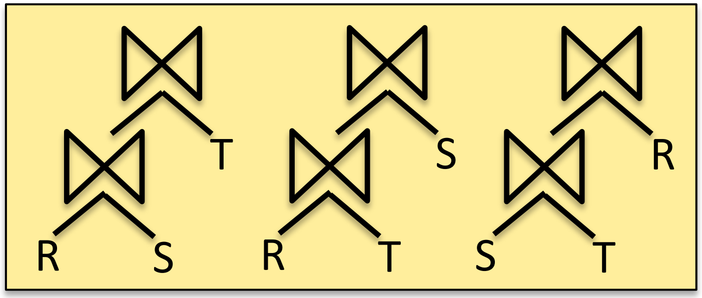
$10^6 + 10^6 + 10^7 + 2*10^7 + 20$ operations ~ $\mathbf{3*10^7}$

Join ordering is the main problem in logical query optimization

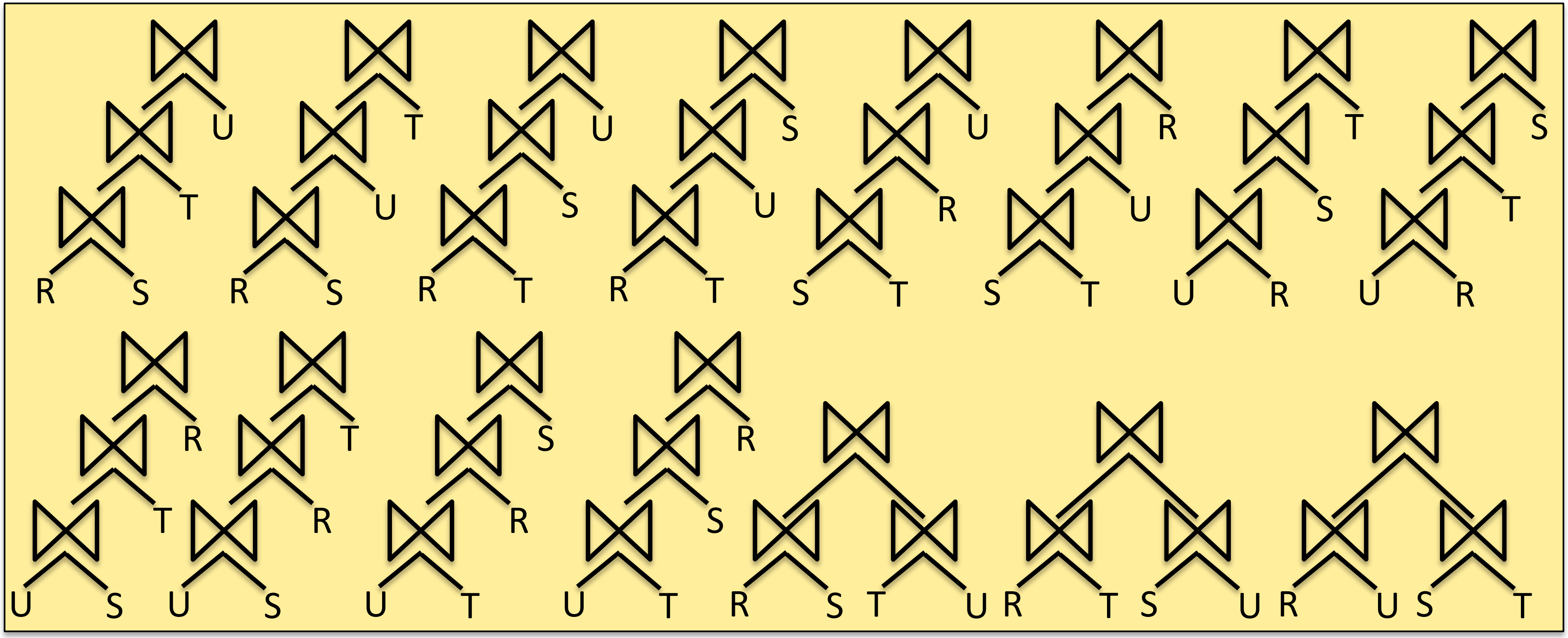
N=2:



N=3:



N=4:



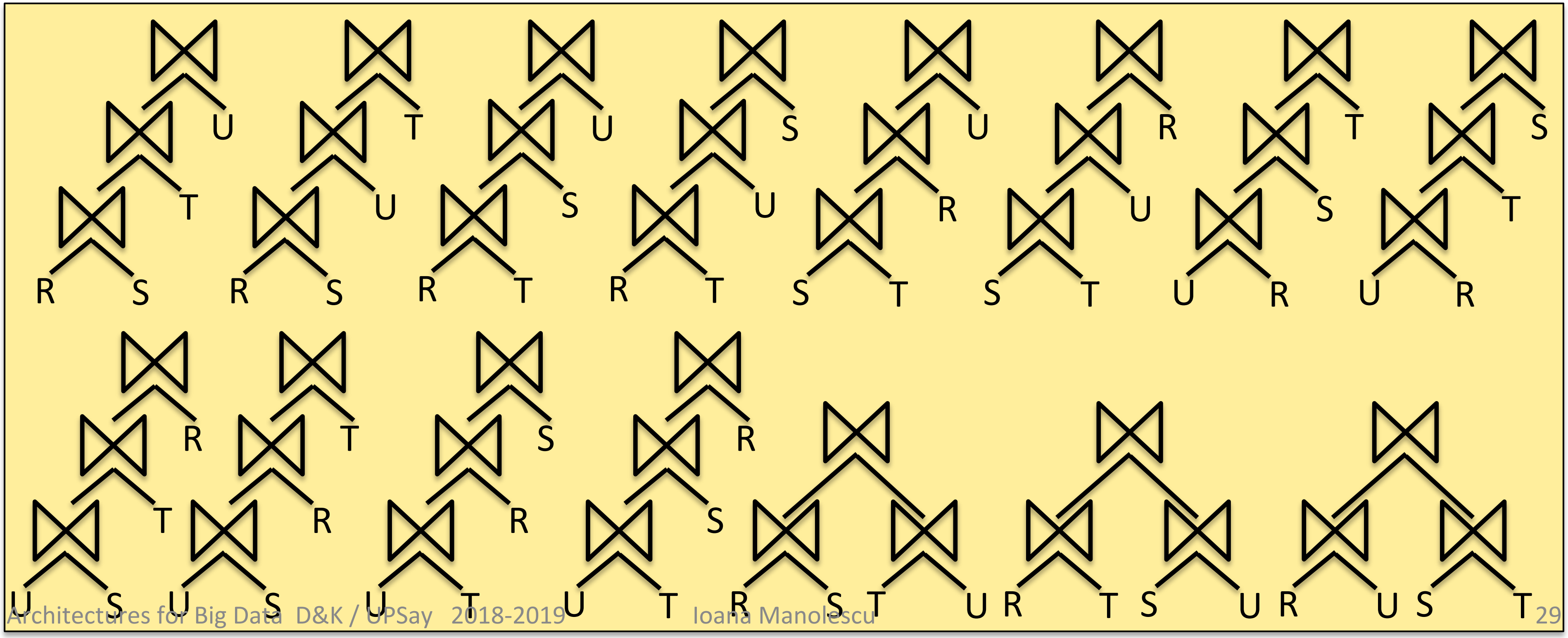
Join ordering is the main problem in logical query optimization

$$Plans(n+1) = (n+1) * Plans(n) + \frac{1}{2} * \sum_{i=1}^{(n/2)} Plans(i)*Plans(n+1-i)$$

High (exponential) complexity → many heuristics

- Exploring only left-linear plans etc.

N=4:



Logical query optimization needs statistics

Exact statistics:

– 1.000.000 cars, 1.000.000 drivers, 1.000 accidents

Approximate / estimated statistics:

– 2 cars per accident, 10 accidents on 1/11/17

Statistics are gathered

- When **loading** the data: take advantage of the scan
- **Periodically** or upon **request** (e.g. `analyze` in the Postgres RDBMS)
- At **runtime**: modern systems may do this to change the data layout (e.g., dynamic indexing – to be seen next)

Statistics on the base data vs. on results of operations not evaluated (yet):

– « On average 2 cars per accident »

- For each column R.a, store:
  |R|, |R.a| (number of distinct values), min{R.a}, max{R.a}
- Assume **uniform distribution** in R.a
- Assume **independent distribution**
  – of values in R.a vs values in R.b; of values in R.a vs values in S.c
- + simple probability computations

More on statistics

- For each column R.a, store:
  |R|, |R.a| (number of distinct values), min{R.a}, max{R.a}
- Assume **uniform distribution** in R.a
- Assume **independent distribution**
  - of values in R.a vs values in R.b; of values in R.a vs values in S.c

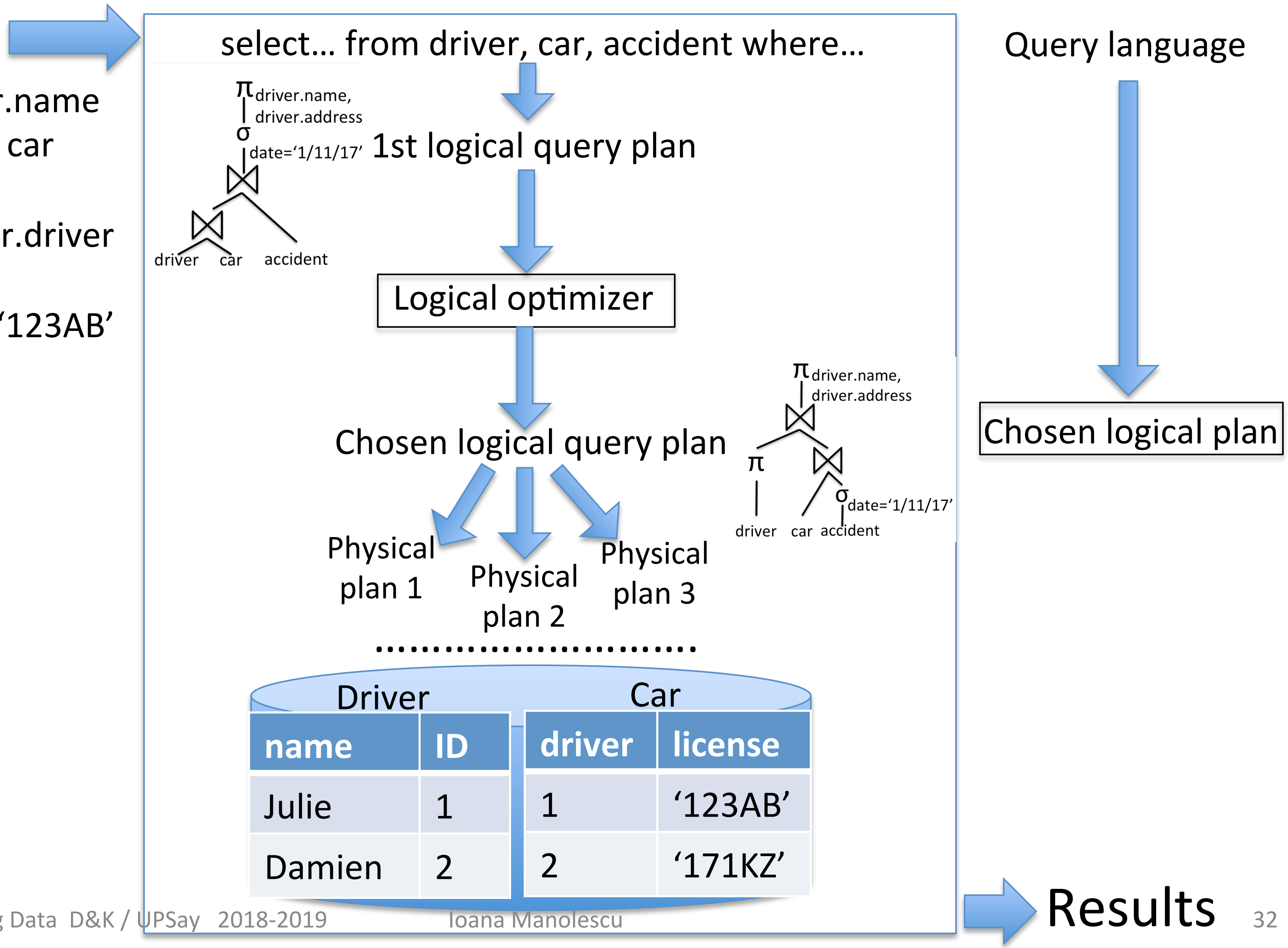
- The **uniform distribution** assumption is frequently wrong
  - Real-world distribution are skewed (popular/frequent values)
- The **independent distribution** assumption is sometimes wrong
  - « Total » counter-example: *functional dependency*
  - Partial but strong enough to ruin optimizer decisions: *correlation*

- Actual optimizers use more sophisticated statistic informations
  - **Histograms**: equi-width, equi-depth
  - Trade-offs: size vs. maintenance cost vs. control over estimation error

Database internal: query optimizer

SQL

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



name	ID
Julie	1
Damien	2

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

Physical query plans

Made up of **physical operators** =

algorithms for implementing logical operators

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

Nested loops join:

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

Merge join: // requires sorted inputs

```
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);
                  }
```

Physical query plans

Made up of **physical operators** =

algorithms for implementing logical operators

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

Nested loops join: O(|R|x|S|)

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

Merge join: // requires sorted inputs O(|R|+|S|)

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

Hash join: // builds a hash table in memory

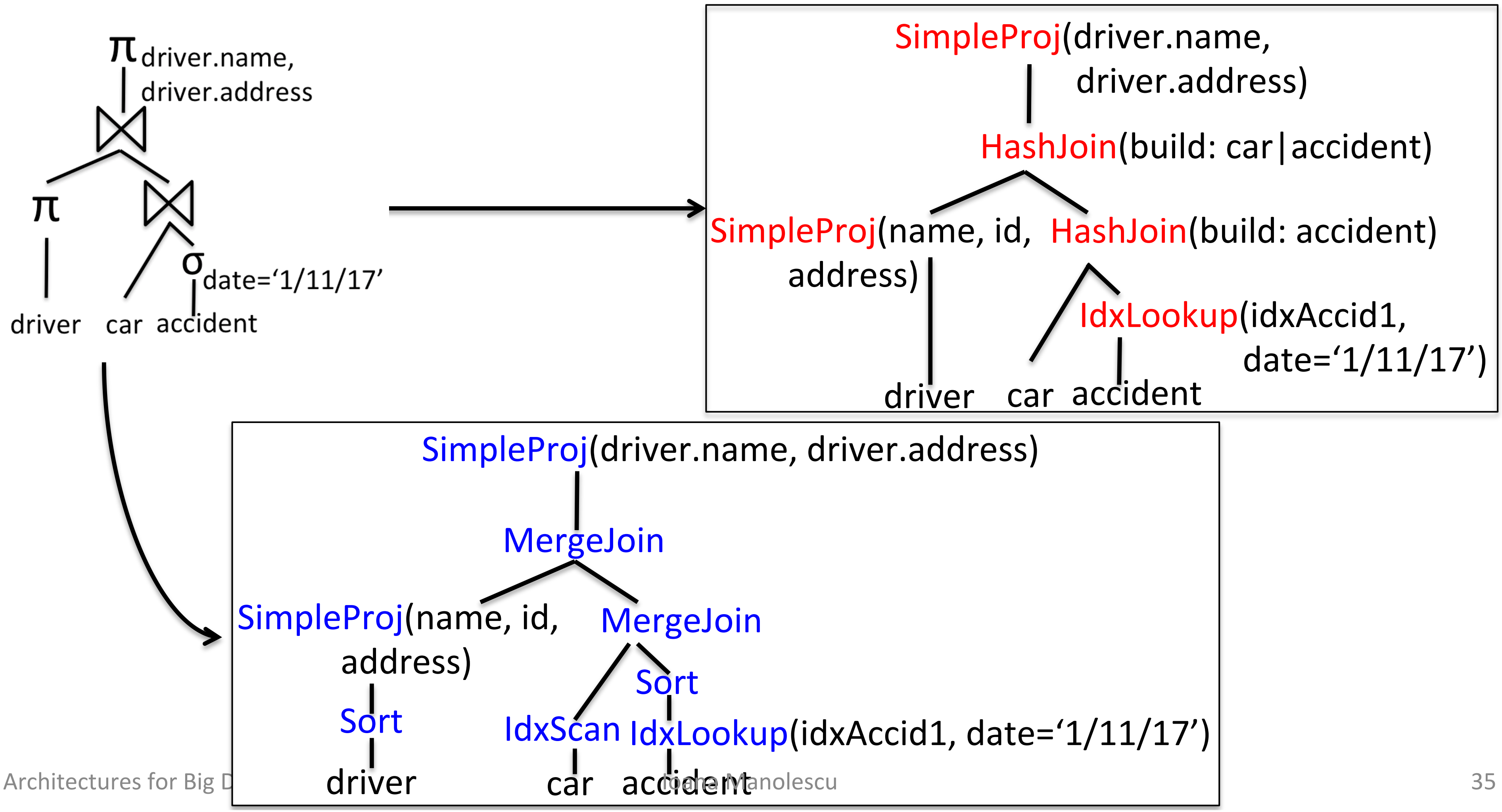
```
While (!endOf(R)) { t ← R.next; put(hash(t.a), t); }
While (!endOf(S)) { t ← S.next;
                    matchingR = get(hash(S.b));
                    output(matchingR x t);
                  }
```

O(|R|+|S|)

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

Physical optimization

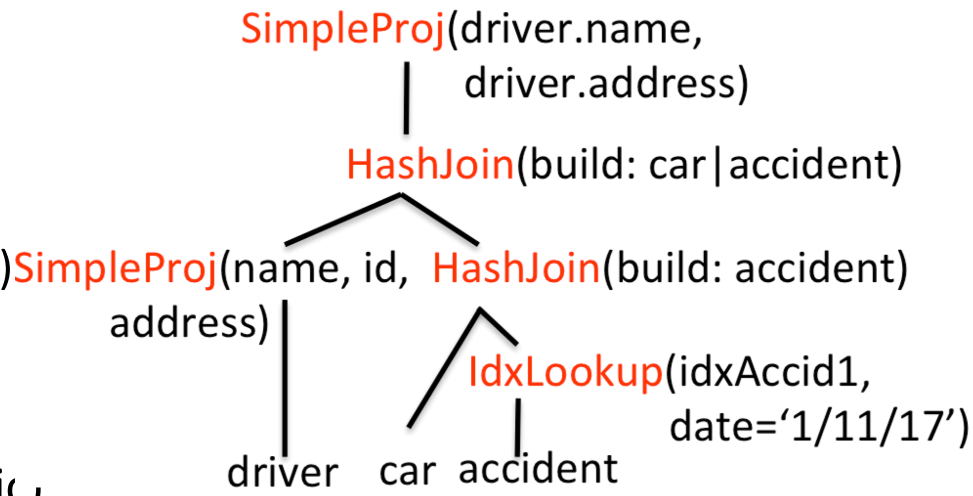
Possible physical plans produced by physical optimization for our sample logical plan:



Physical plan performance

Metrics characterizing a physical plan

- **Response time**: between the time the query starts running to the we know it's end of results
- **Work** (resource consumption)
  - How many **I/O** calls (blocks read)
    - Scan, IdxScan, IdxAccess; Sort; HybridHash (or spilling HashJoin)
  - How much **CPU**
    - All operators
  - (Distributed plans: **network** traffic)
- **Total work**: work made by all operators



Query optimizers in action

Most database management systems have an « explain » functionality → physical plans. Below sample Postgres output:

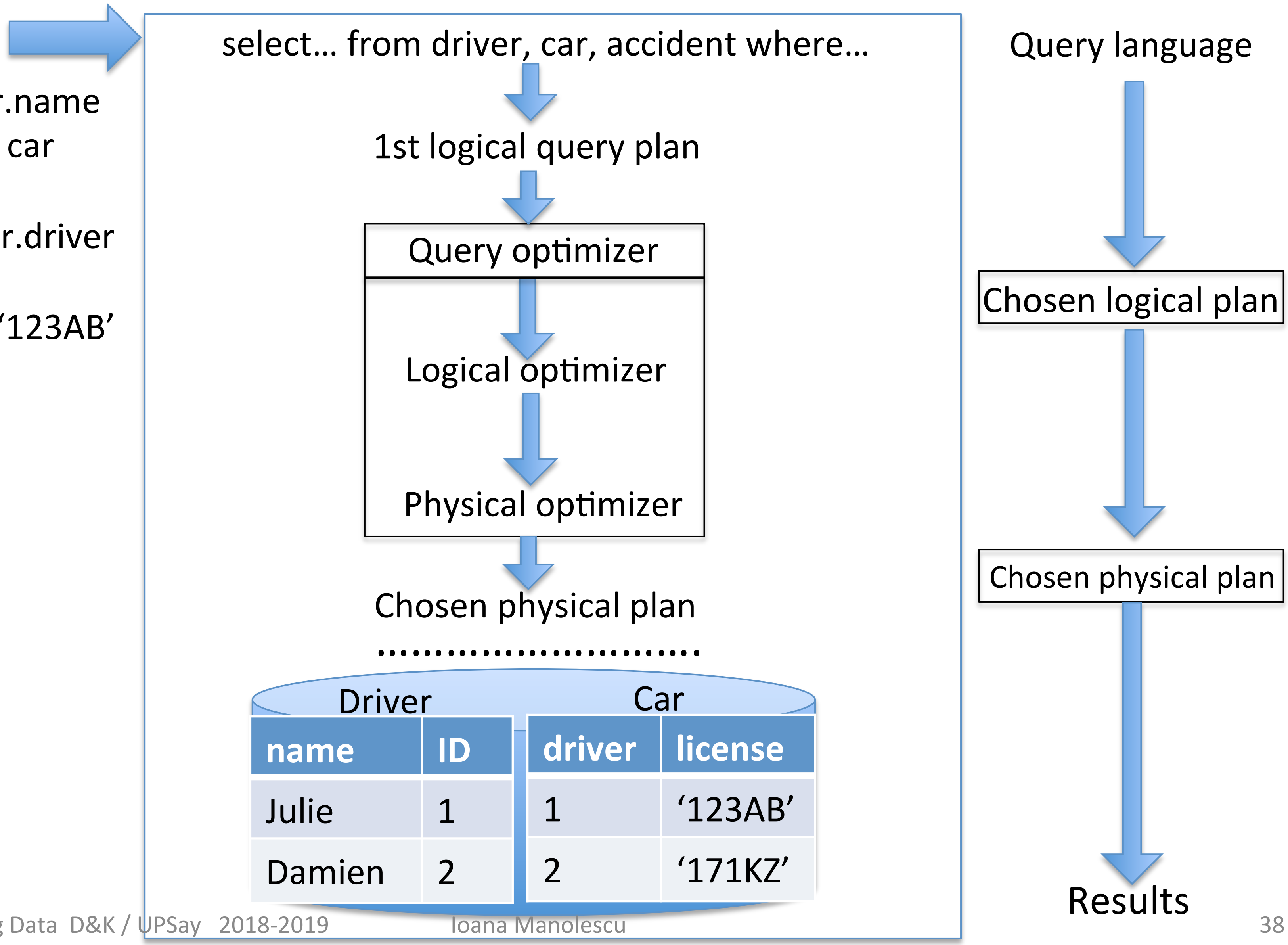
```
EXPLAIN SELECT * FROM tenk1;
                        QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
                                        QUERY PLAN
--------------------------------------------------------------------------------------
Hash Join (cost=232.61..741.67 rows=106 width=488)
   Hash Cond: ("outer".unique2 = "inner".unique2)
   -> Seq Scan on tenk2 t2 (cost=0.00..458.00 rows=10000 width=244)
   -> Hash (cost=232.35..232.35 rows=106 width=244)
         -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244)
         Recheck Cond: (unique1 < 100)
         -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
```

Database internal: physical plan

SQL

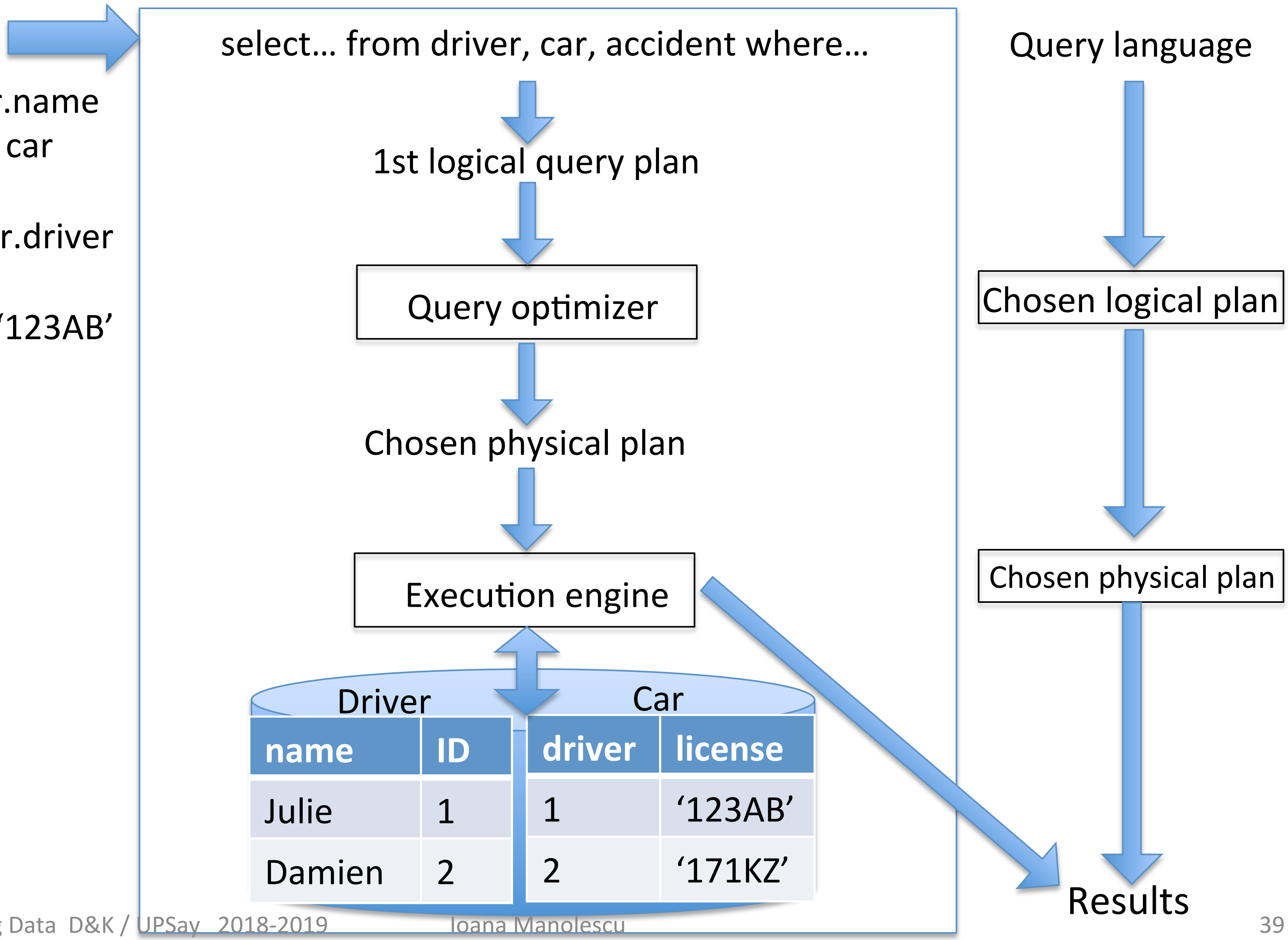
select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license='123AB'



Database internals: query processing pipeline

SQL

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



ARCHITECTURES FOR BIG DATA MANAGEMENT:

WHAT NEEDS TO CHANGE?

What is the impact of Big Data properties on database architectures?

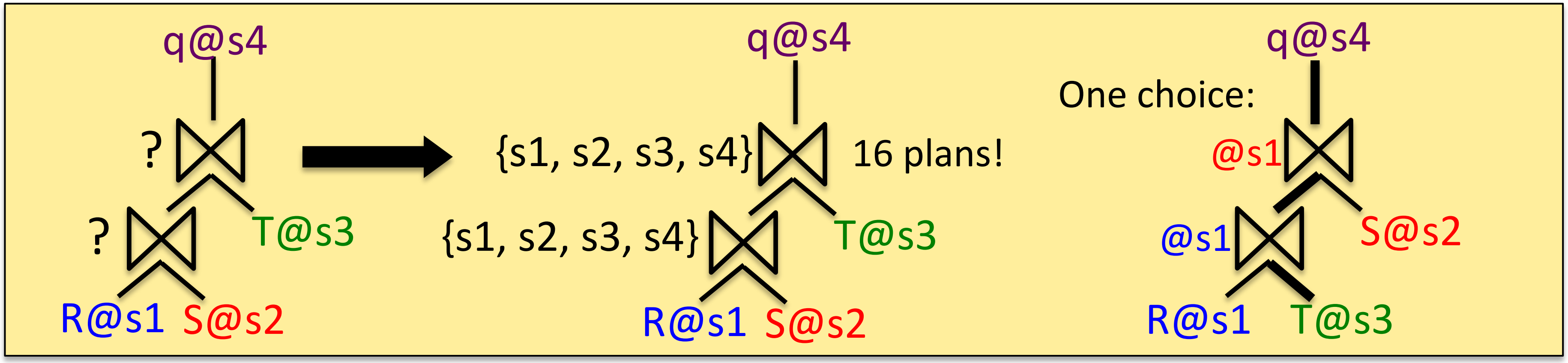
- Volume and Velocity require **distribution**
  - Of the data; of query evalution
  - Distribution makes **ACID difficult**
  - Distribution requires efficient, easy-to-use parallelism
- Variety requires support for
  - **flexible data models**: key-values, JSON, graphs...
  - **different schemas**, and translation mechanisms between the schemas
  - **several data models** being used together

Distributed databases

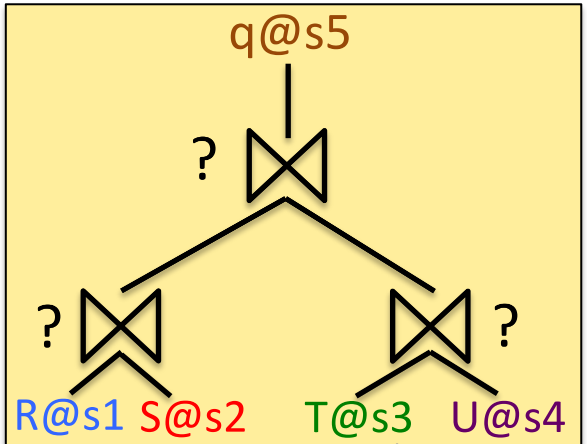
- Oldest distributed architecture ('70s)
- Illustrate/introduce the main priciples
- Data is distributed among many *nodes* (*sites*, *peers*...)
  - **Data catalog**: information on which data is stored where
    - Explicit : « All Paris sales are stored in Paris ».
      Ex: Relational table fragmentation (horizontal, vertical etc.)
      Catalog stored at a master/central server.
    - Implicit: « Data is distributed by the value of the city »
      (« somewhere »)
      Catalog split across all sites (P2P) or at a master (Hadoop FS)
- Queries are distributed (may come from any site)
- Query processing is distributed
  - Operators may run on different sites → network transfer
  - Another layer of complexity to the optimization process

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

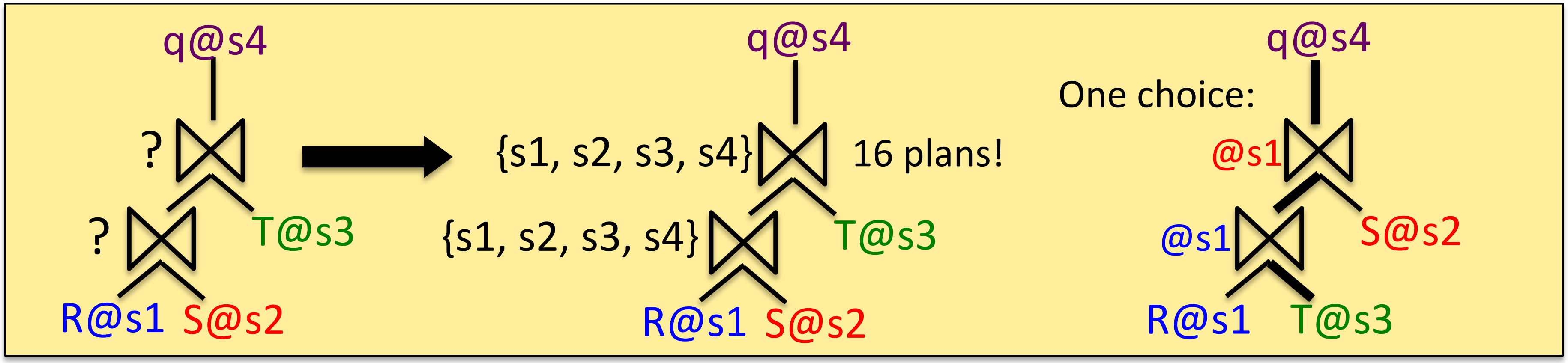


Plan pruning criteria if all the sites and network connections have equal performance

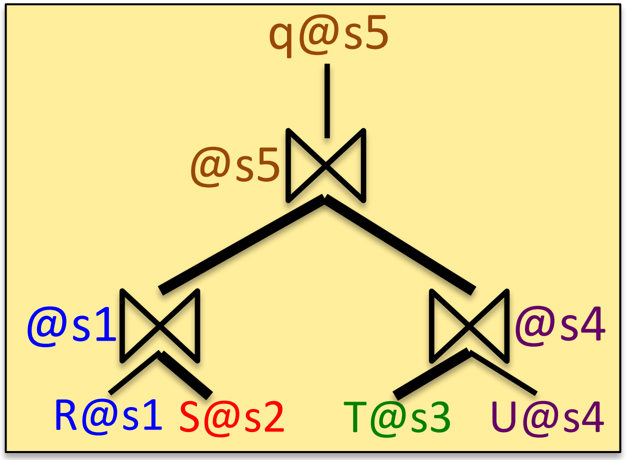
- *Ship the <u>smaller</u> collection*

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

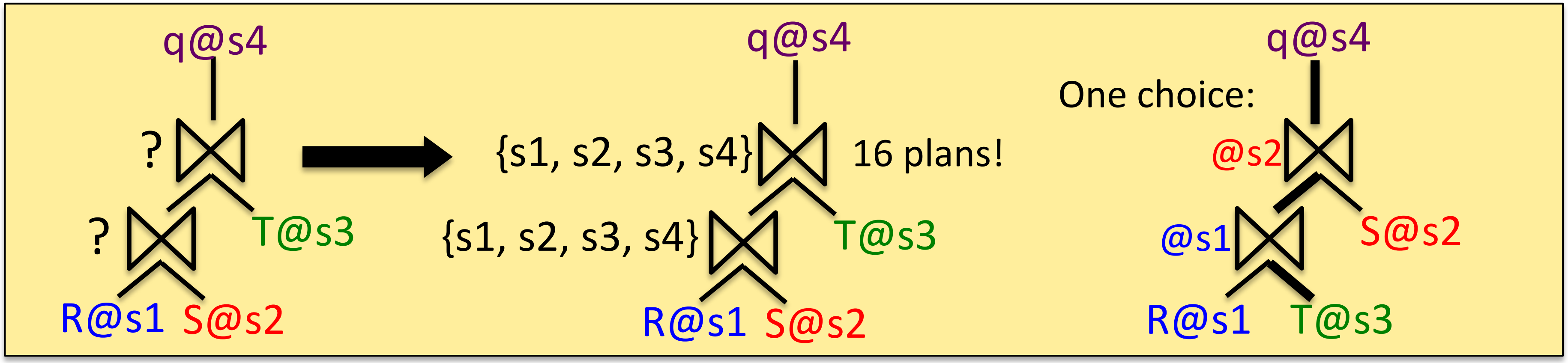


Plan pruning criteria if all the sites and network connections have equal performance:

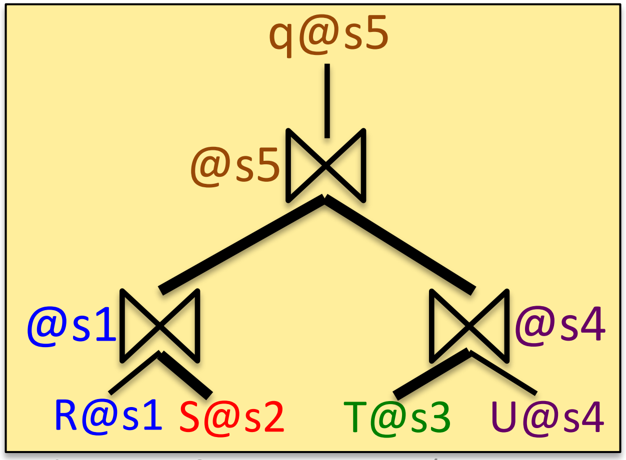
- *Ship the smaller collection.*
- *Transfer to join partner or the query site*

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 smaller collection.*
- *Transfer to join partner or the query site*

This plan illustrates total effort != response time

Dimensions of distributed systems

- **Data model**:
  - Relations, trees (XML, JSON), graphs (RDF, others…), nested relations
  - Query language
- **Heterogeneity** (DM, QL): none, some, a lot
- **Scale**: small (~10-20 sites) or large (~10.000 sites)
- **ACID** properties
- **Control**:
  - Single master w/complete control over N slaves (Hadoop/HDFS)
  - Sites publish independently and process queries as directed by single master/*mediator*
  - Many-mediator systems, or peer-to-peer (P2P) with *super-peers*
  - Sites completely independent (P2P)

Distributed relational databases

- **DM**: relations; **language**: SQL; **ACID**: cf. SQL standard
- **Heterogeneity**: none
- **Control**:

Servers 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
```

Site3 decides what to import from site1, site2 (« hard links »)

Site1, site2 are independent servers

Also: replication policies, distribution etc. (usually with one or a few masters)

- **Size**: small