DATABASE FUNDAMENTALS (RECALL/CRASH COURSE)

Database functionalities

Relational Database Management Systems

- Functionality provided
  - What kind of data can I put in? **Relations**
  - How can I get data out of it? **SQL query language**
  - How does it handle concurrent access? **ACID (or less)**
  - How long does a given operation take? **Query optimization**
- Implementation (internals)
  - How does it cope with scale?
    for reads? **Smart storage and indexing structures**
    for writes? **Concurrency control**

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

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

  E.g. every client has a single birthdate
- **Isolation**: 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
- **Durability**: 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)

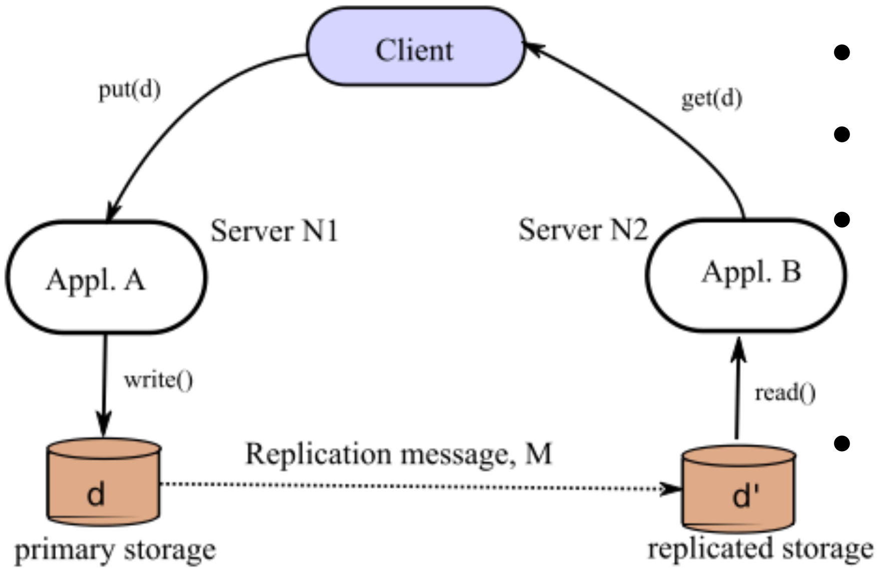
Limits of ACIDity in large distributed systems: the **CAP theorem**

- 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



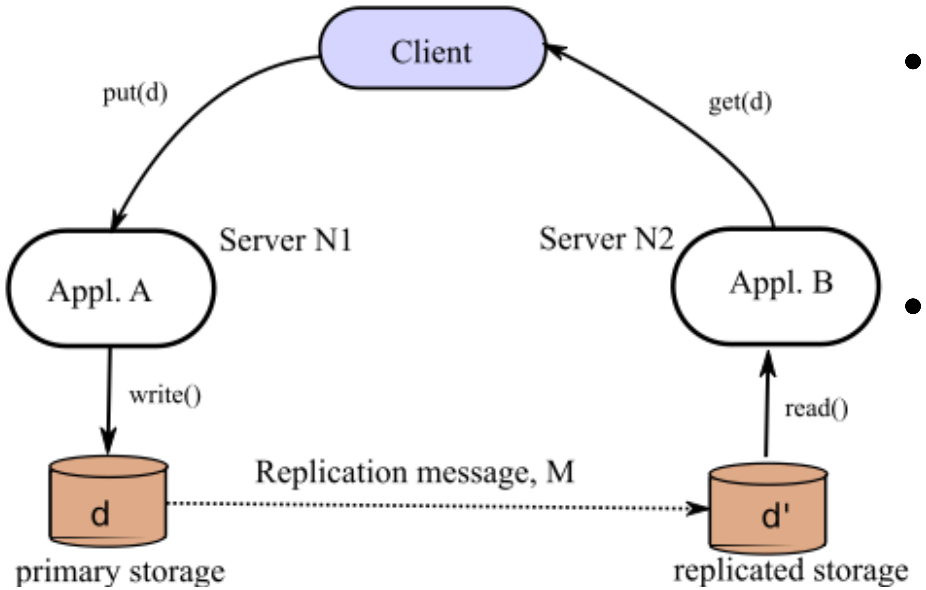
- Primary and replica store
- Applications A and B on servers
- Client writes a new d value through A, which propagates d to the replica (replacing the old d’)
- Subsequently, client reads from B

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?

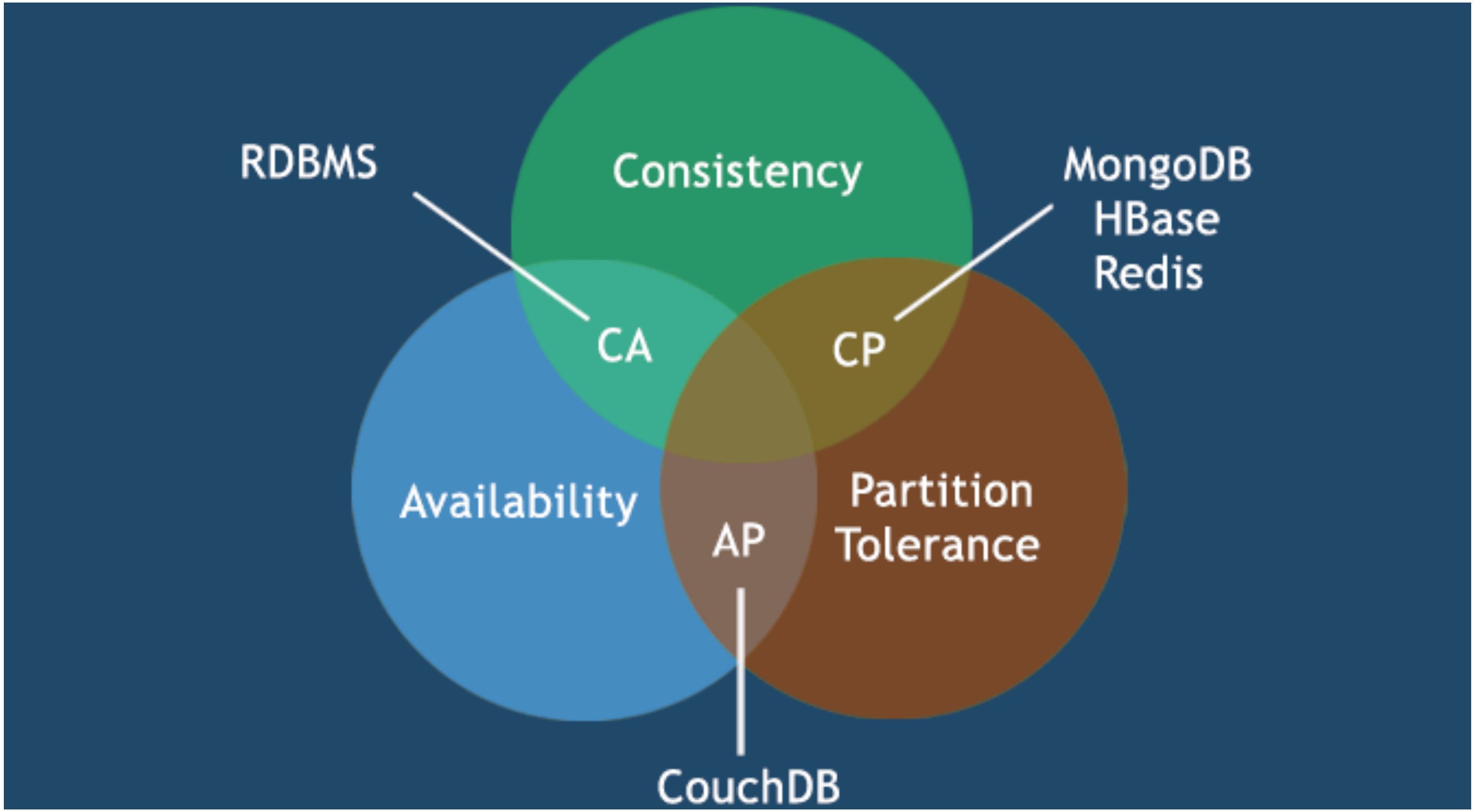


- **Partition tolerance**: we must have it (cannot block if one machine fails)
- Then one must *trade some consistency for availability*

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?

1. Merge copies: find a commonly agreed upon version
   - Concurrent Versioning Systems (CVS, SVN, GIT) do this pretty well but not always
   - Some conflicts remain to be solved by the user
2. Limit the operation set to have fewer conflicts and/or easier to solve
   - E.g., Google Docs solves conflicts by allowing *only style change and add/delete text*
   - E.g., using only *commutative* operations: there is always a way to rearrange a set of operations in a preferred consistent global order
     1. Addition is commutative
     2. Addition with a bounds check is not

DATABASE FUNDAMENTALS (RECALL/CRASH COURSE)

Database architecture

What's in a database?

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

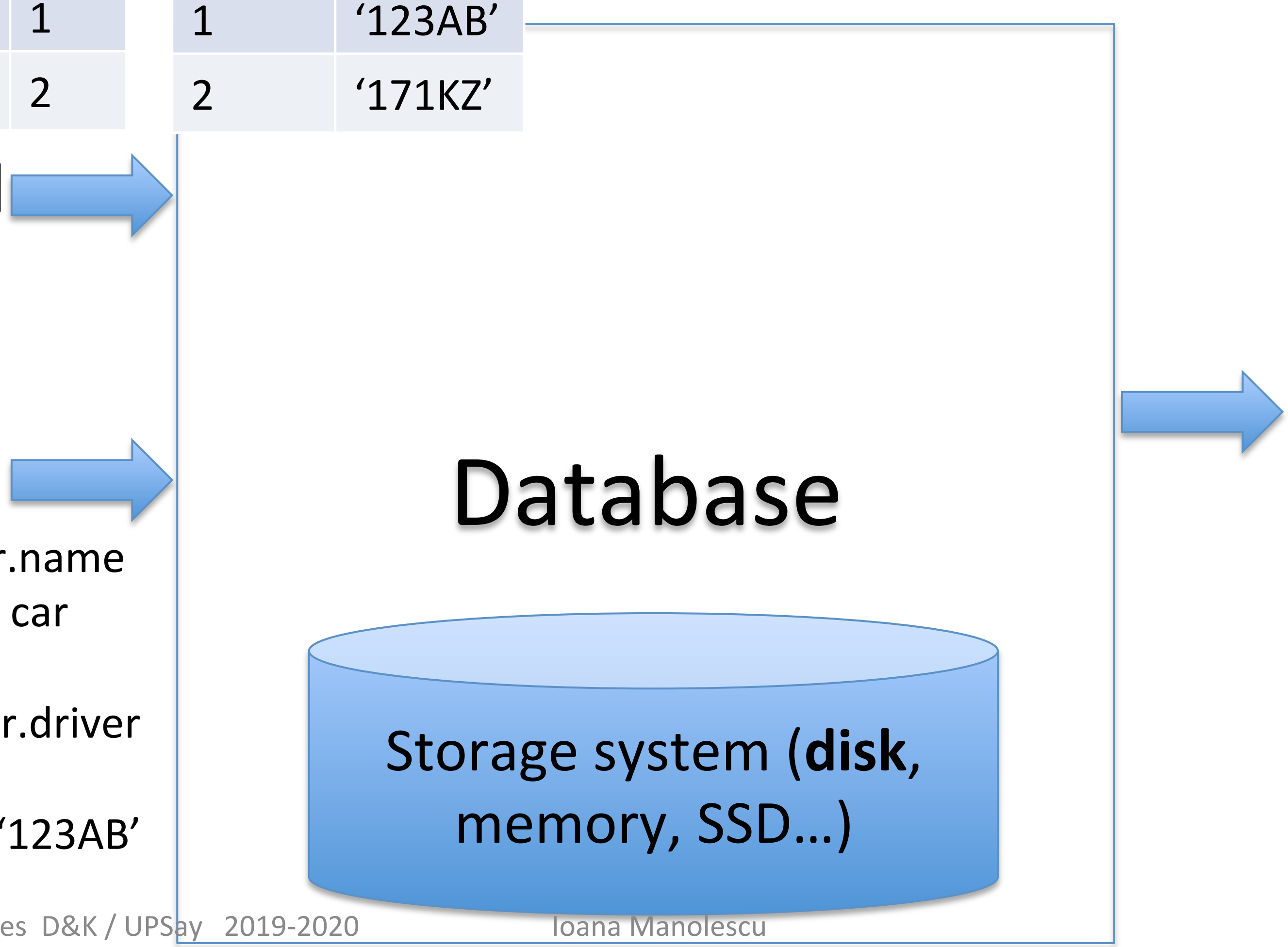
Car

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’



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’

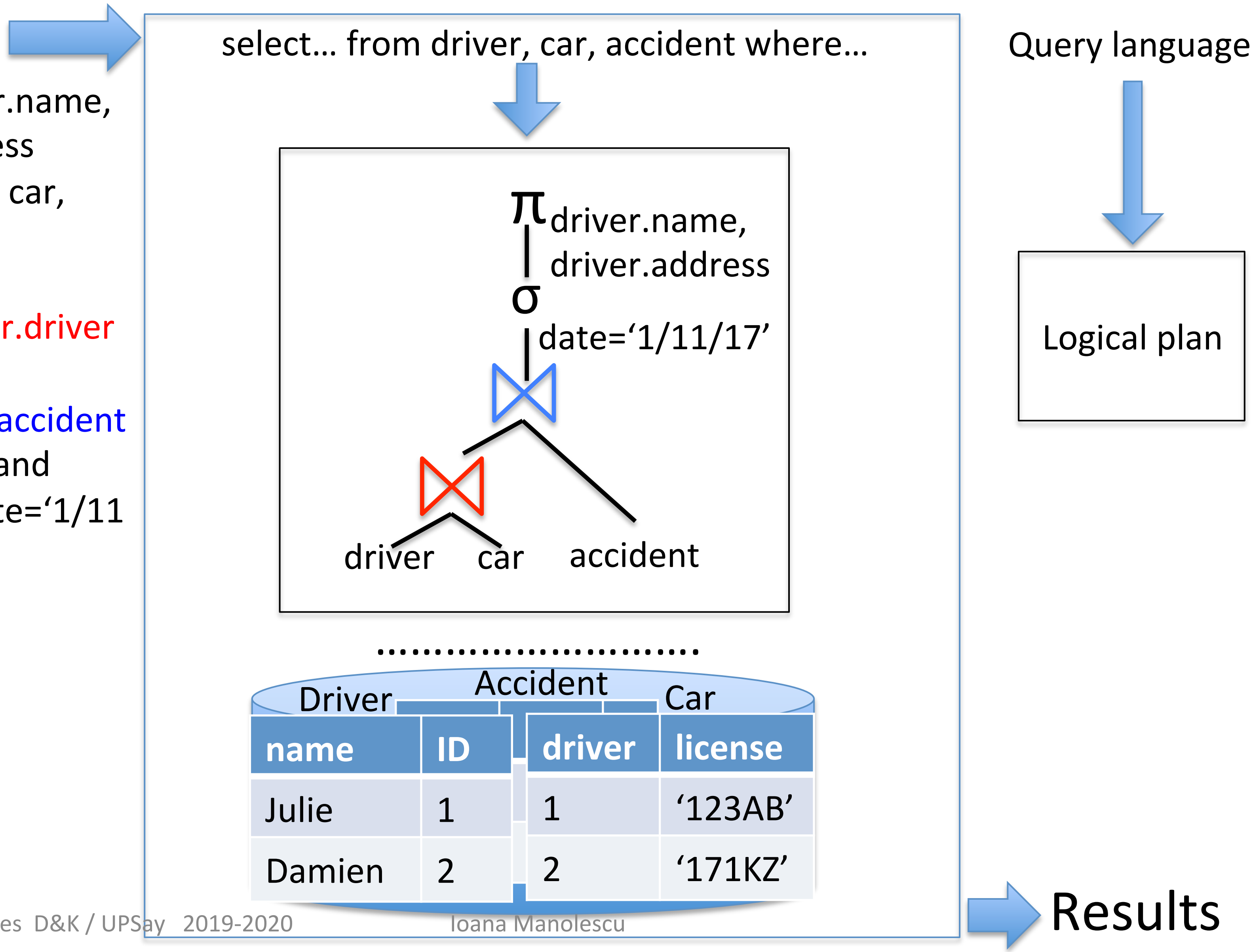
3. Results

name
Julie

What's in a database?

SQL

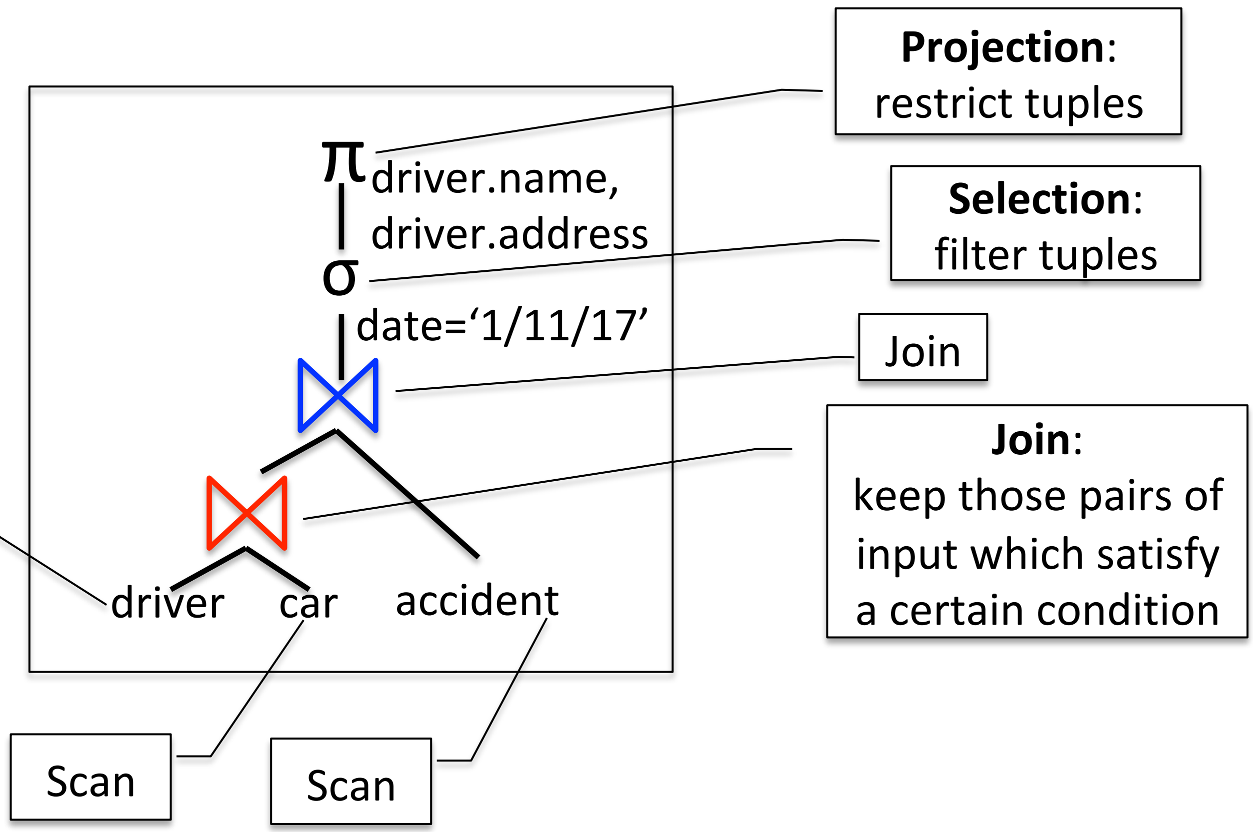
select driver.name, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date='1/11/17'



Logical query plans

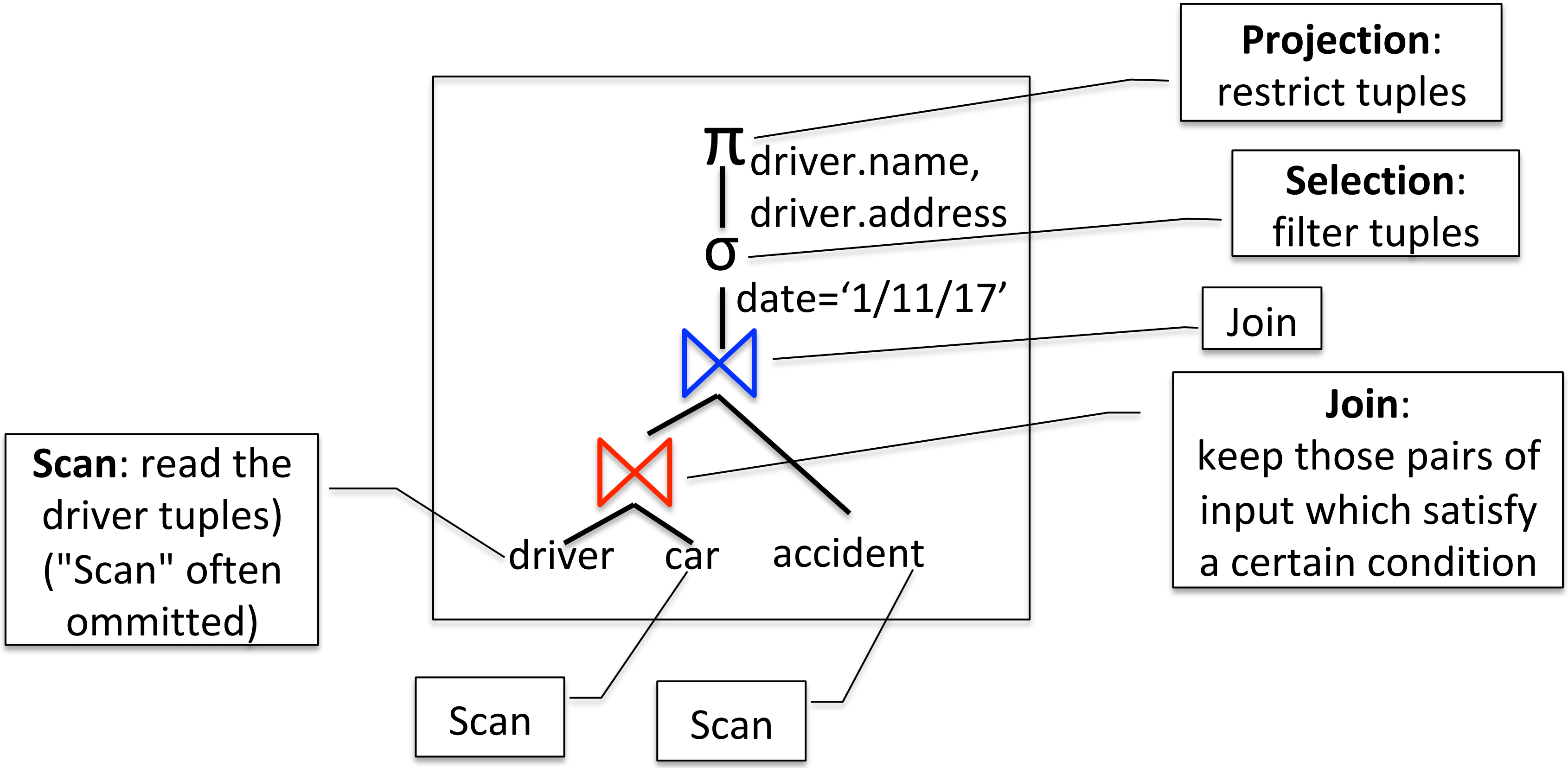
- Trees made of logical operators, each of which specializes in a certain task

SQL:
select driver.name, driver.address
from driver, car, accident
where
driver.ID=car.driver and
car.license=accident.carLicense and
accident.date='1/11/17'



Logical query plans

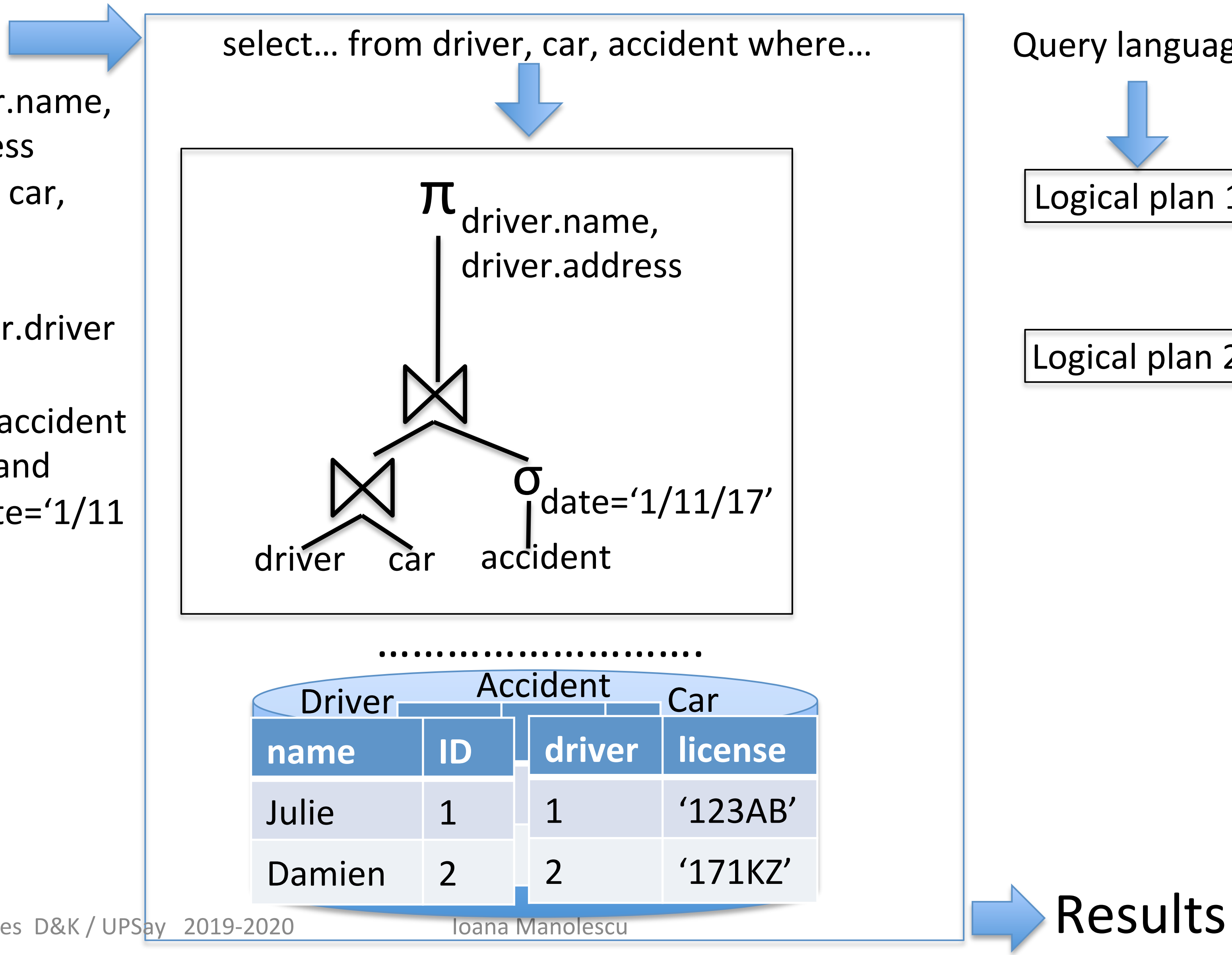
- Trees made of logical operators, each of which specializes in a certain task
- Logical operators: they are defined by their result, not by an algorithm
- Physical operators (see next) implement actual algorithms



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'



name	ID
Julie	1
Damien	2

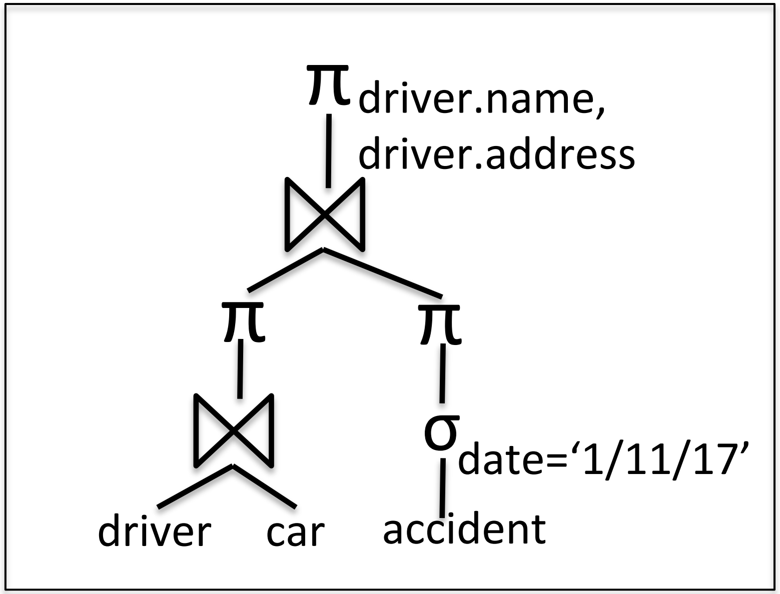
driver	license
1	'123AB'
2	'171KZ'

What's in a database?

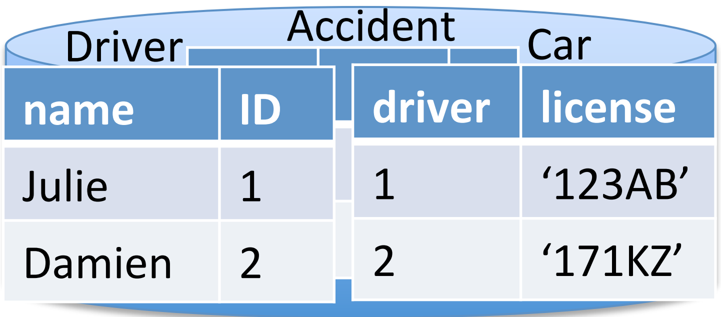
SQL

select driver.name, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date=‘1/11/17’

select… from driver, car, accident where…



……………………………



name	ID
Julie	1
Damien	2

driver	license
1	‘123AB’
2	‘171KZ’

Query language

Logical plan 1

Logical plan 2

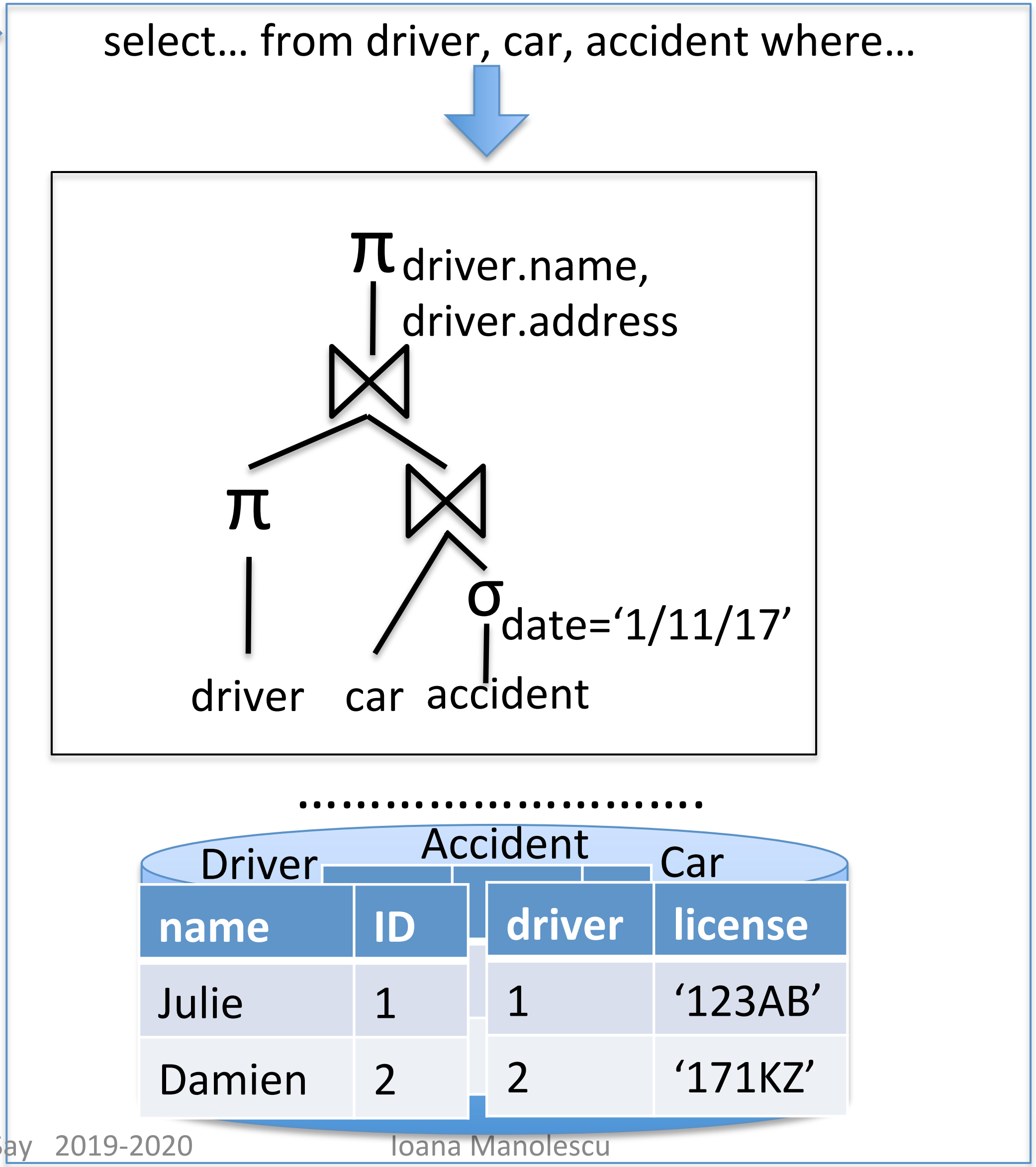
Logical plan 3

Results

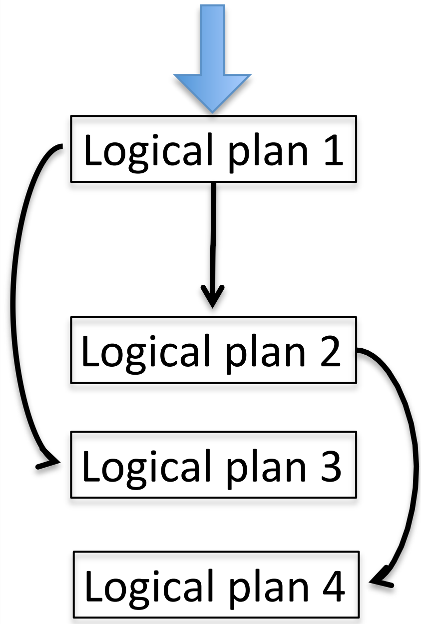
What's in a database?

SQL

select driver.name, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date=‘1/11/17’



Query language



Results

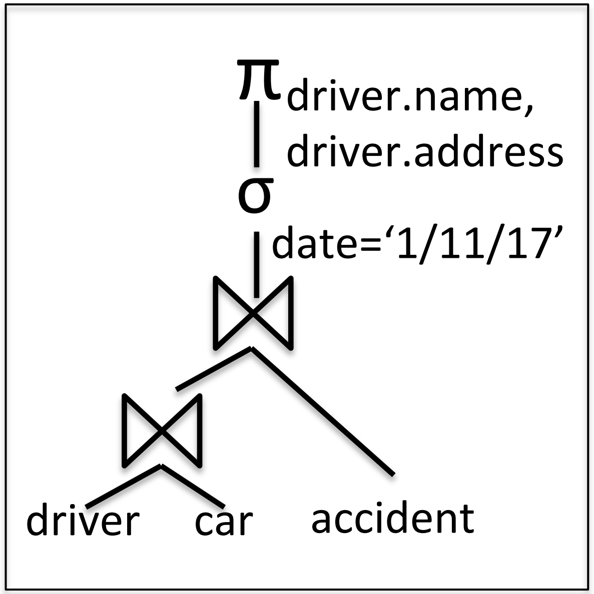
Logical query optimization

- 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? »



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

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

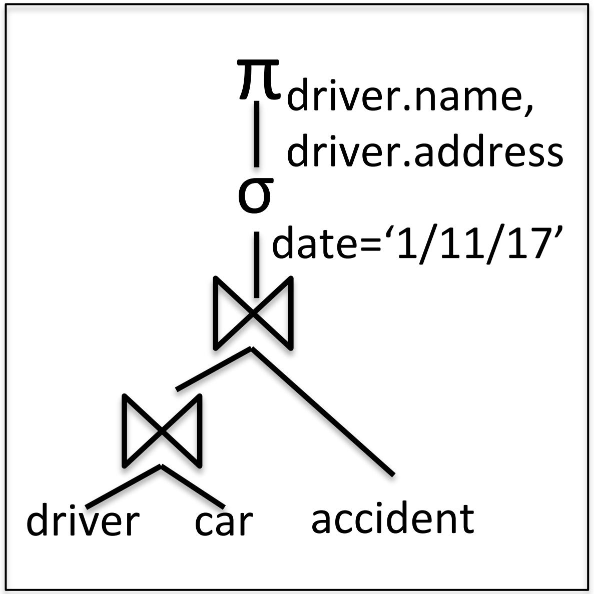
Plan **cost** = the sum of the costs of all operators in a plan

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? »

Pessi-mistic (worst-case) estim.



Scan costs: cs x $(10^6+10^6+10^3)$

Scan cardinality estimations: 10^6, 10^6, 10^3

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

Driver-car join cardinality estimation: 10^6

Driver-car-accident join cost estim.: cj x $(10^6 \times 10^3 = 10^9)$

Driver-car-accident join cardinality estimation: 2×10^3

Selection cost estimation: cf x (2×10^3)

Selection cardinality estimation: 10

Projection (similar), negligible

cs, cj, cf constant

Total cost estimation: cs x $(2\times10^6+10^3)$+ cf x 2×10^3

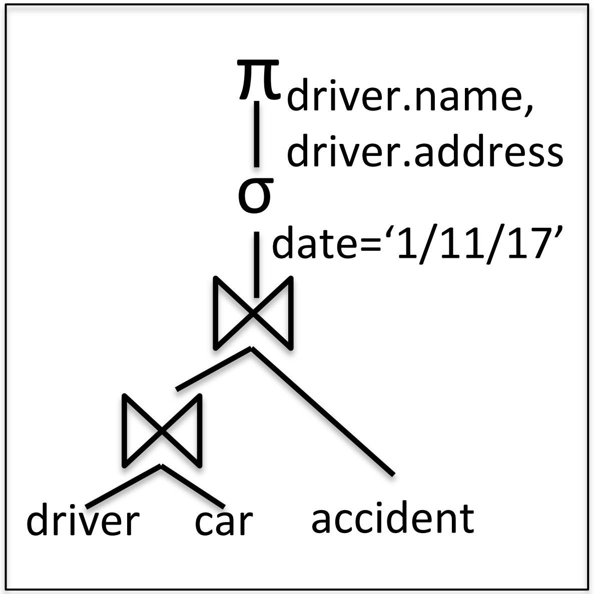
+ cj x $(10^{12} + 2\times10^3)$ ~ cj x 10^{12} ~ $\mathbf{10^{12}}$

Logical query optimization example

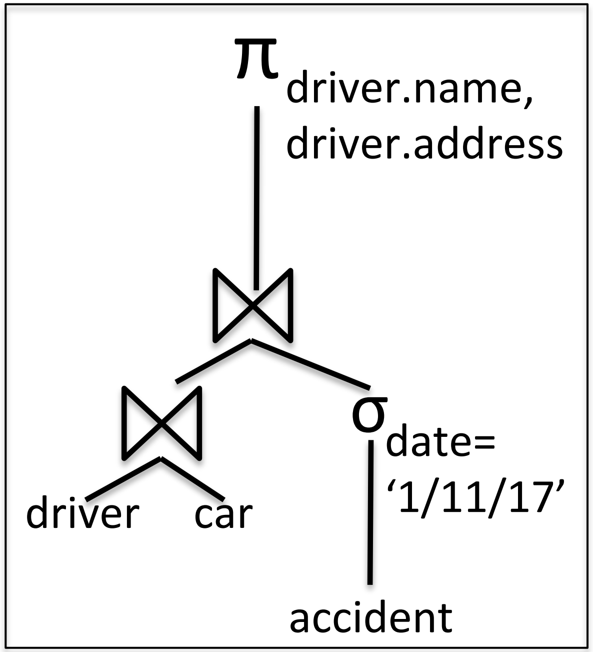
1.000.000 cars, 1.000.000 drivers, 1.000 accidents, 2 cars per accident, 10 accidents on 1/11/17

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

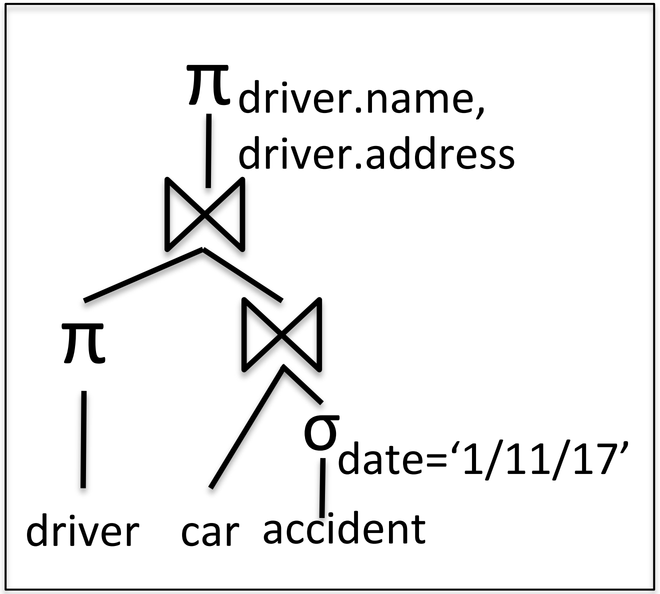
Three plans, same scan costs (neglected below); join costs dominant



$10^9 + 10^{12} \sim \mathbf{10^{12}}$



$10^9 + 10^7 \sim \mathbf{10^9}$



$10^7 + 2*10^7 \sim \mathbf{3*10^7}$

Logical query optimization example

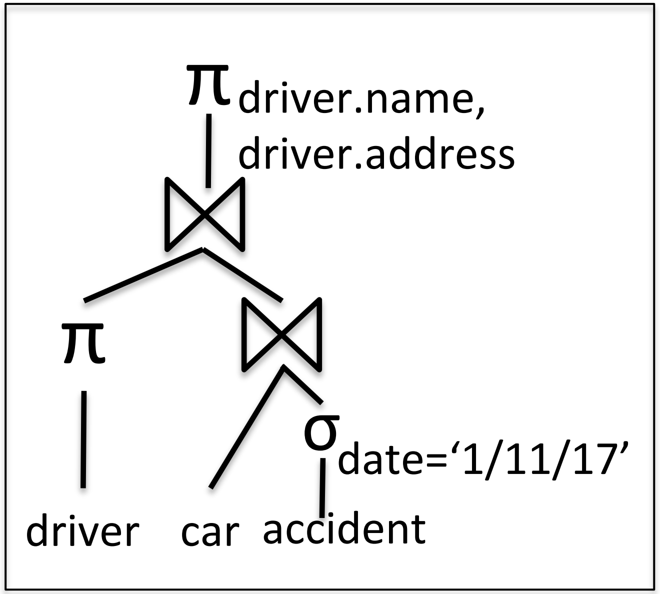
1.000.000 cars, 1.000.000 drivers, 1.000 accidents, 2 cars per accident, 10 accidents on 1/11/17

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

Three plans, same scan costs (neglected below); join costs dominant

The best plan reads only the accidents that have to be consulted

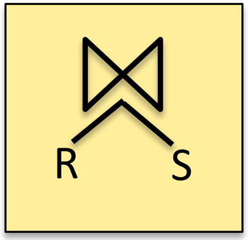
- **Selective data access**
- Typically supported by an **index**
  - Auxiliary data structure, built on top of the data collection
  - Allows to access directly objects satisfying a certain condition



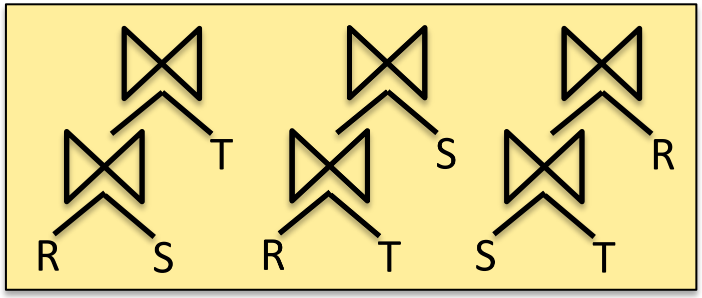
$10^7 + 2*10^7 \sim \mathbf{3*10^7}$

Join ordering is the main problem in logical query optimization

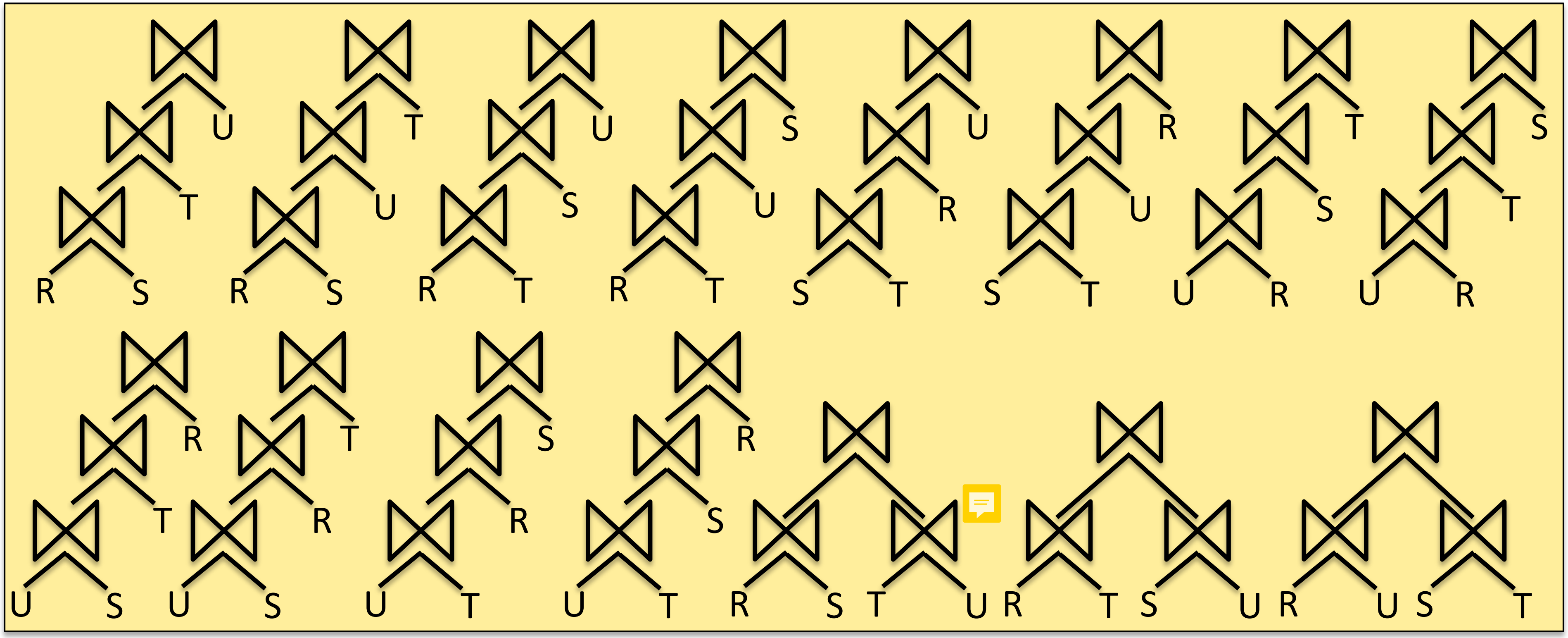
N=2:



N=3:



N=4:



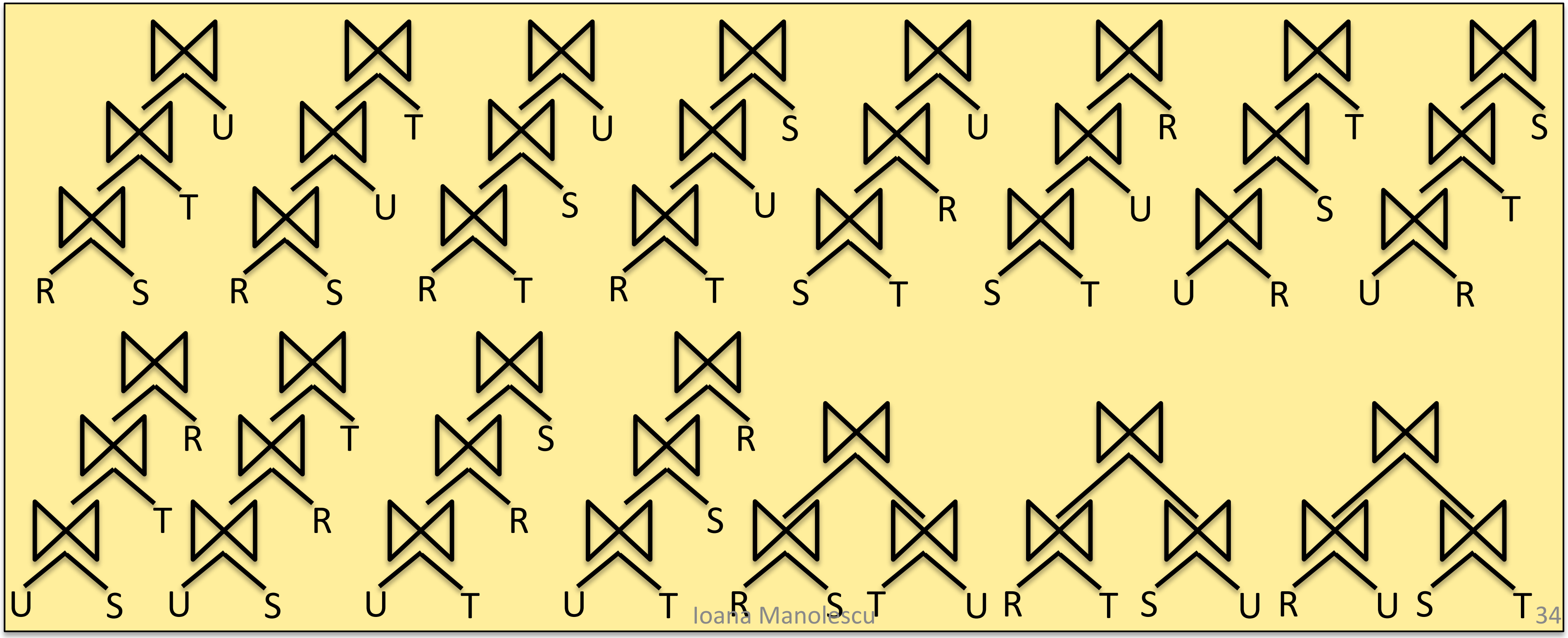
Join ordering is the main problem in logical query optimization

$$Plans(n+1) = (n+1) * Plans(n) + ½ * \Sigma_{i=1}^{(n/2)} Plans(i)*Plans(n+1-i)$$

High (exponential) complexity → many heuristics

- Exploring only left-linear plans etc.

N=4:



Logical query optimization needs statistics

Exact statistics (on base data):

– 1.000.000 cars, 1.000.000 drivers, 1.000 accidents

Approximate / estimated statistics (on intermediary results)

– "1.75 cars involved in every accident"

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

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

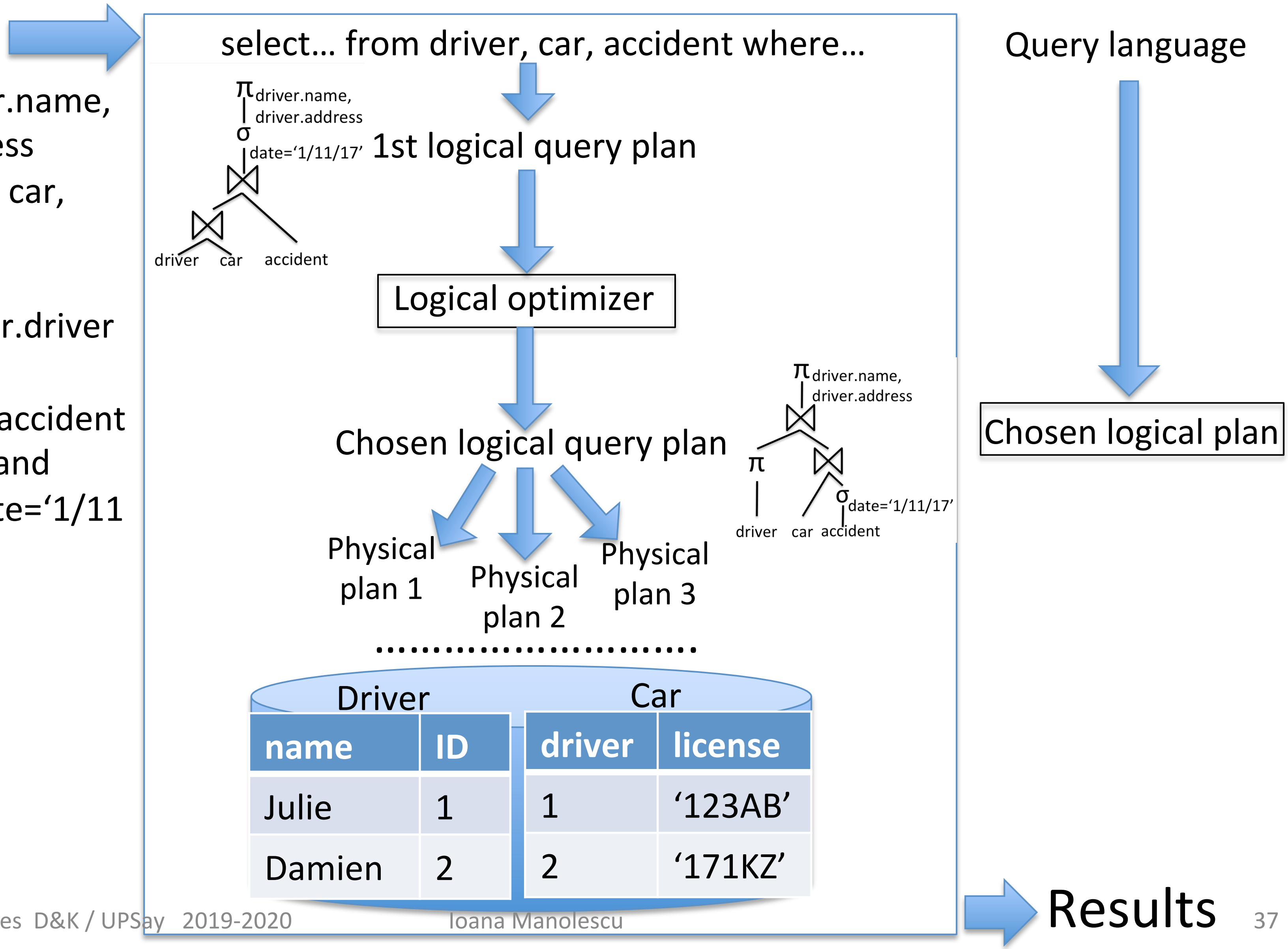
- The **uniform distribution** assumption is frequently wrong
  - Real-world distribution are skewed (popular/frequent values)
- The **independent distribution** assumption is sometimes wrong
  - « Total » counter-example: *functional dependency*
  - Partial but strong enough to ruin optimizer decisions: *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, driver.address
from driver, car, accident
where driver.ID=car.driver and car.license=accident.carLicense and accident.date='1/11/17'



Driver

name	ID
Julie	1
Damien	2

Car

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

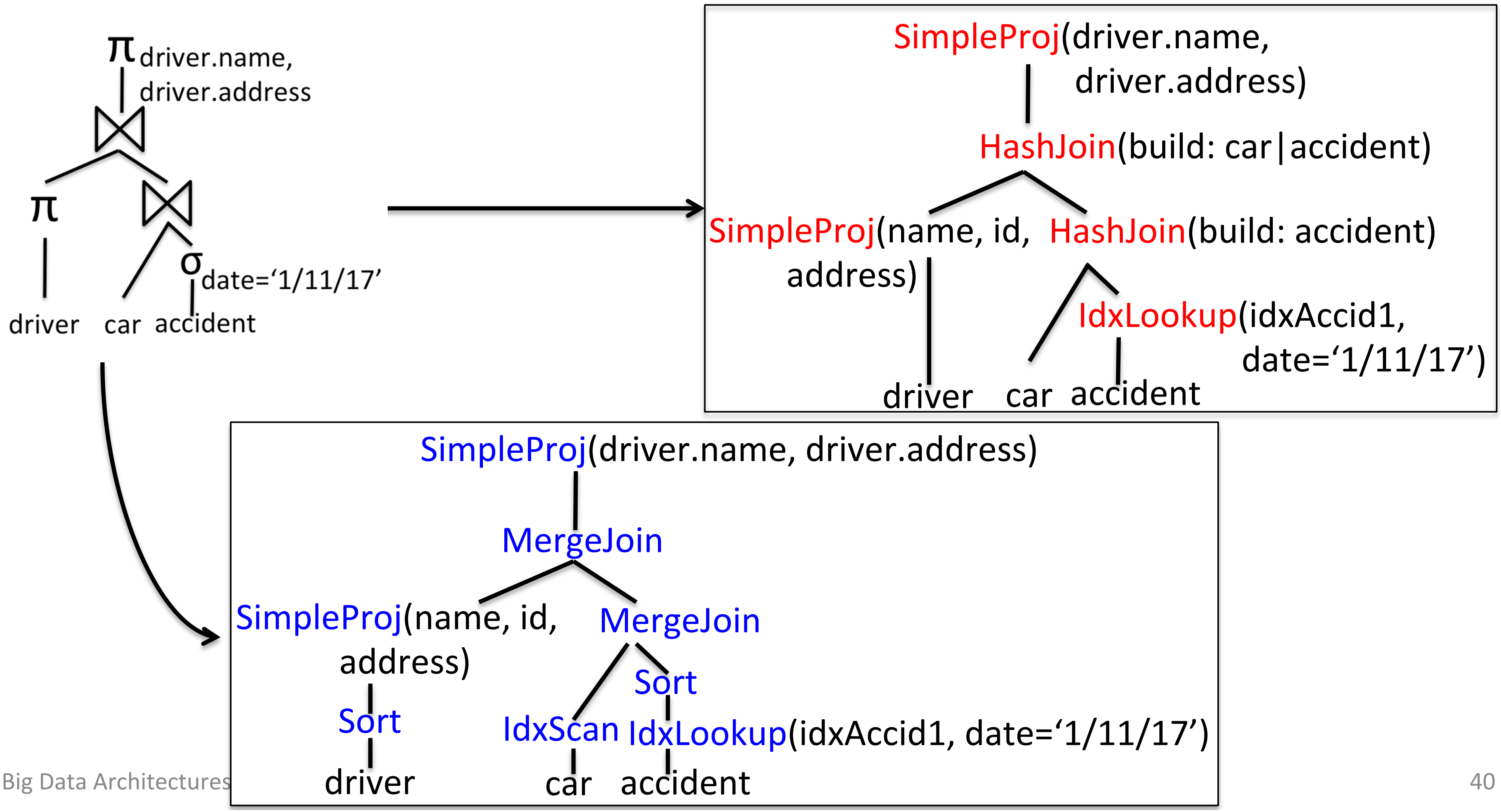
```
While (!endOf(R)) { t ← R.next; put(hash(t.a), t); }
While (!endOf(S)) { t ← S.next;
                    matchingR = get(hash(S.b));
                    output(matchingR x t);
                  }
```

O(|R|+|S|)

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

Physical optimization

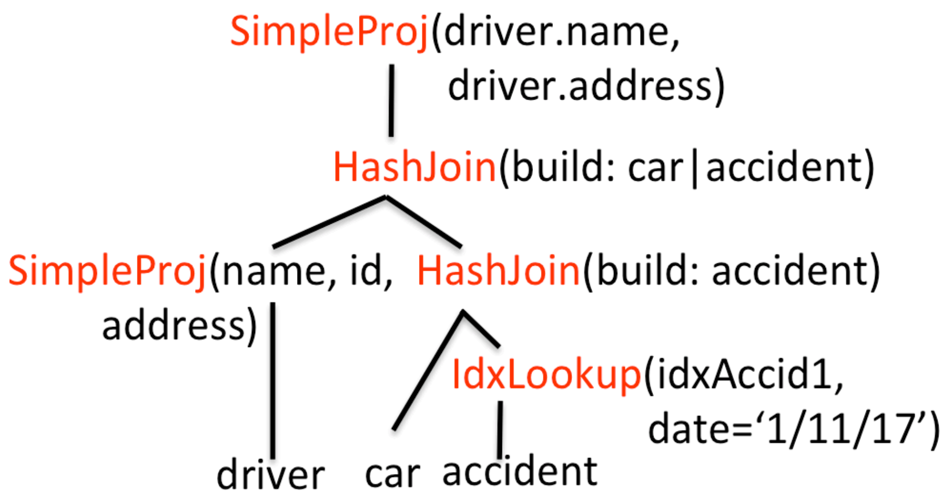
Possible physical plans produced by physical optimization for our sample logical plan:



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'

select… from driver, car, accident where…

1st logical query plan

Query optimizer

Logical optimizer

Physical optimizer

Chosen physical plan

Driver

name	ID
Julie	1
Damien	2

Car

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

Query language

Chosen logical plan

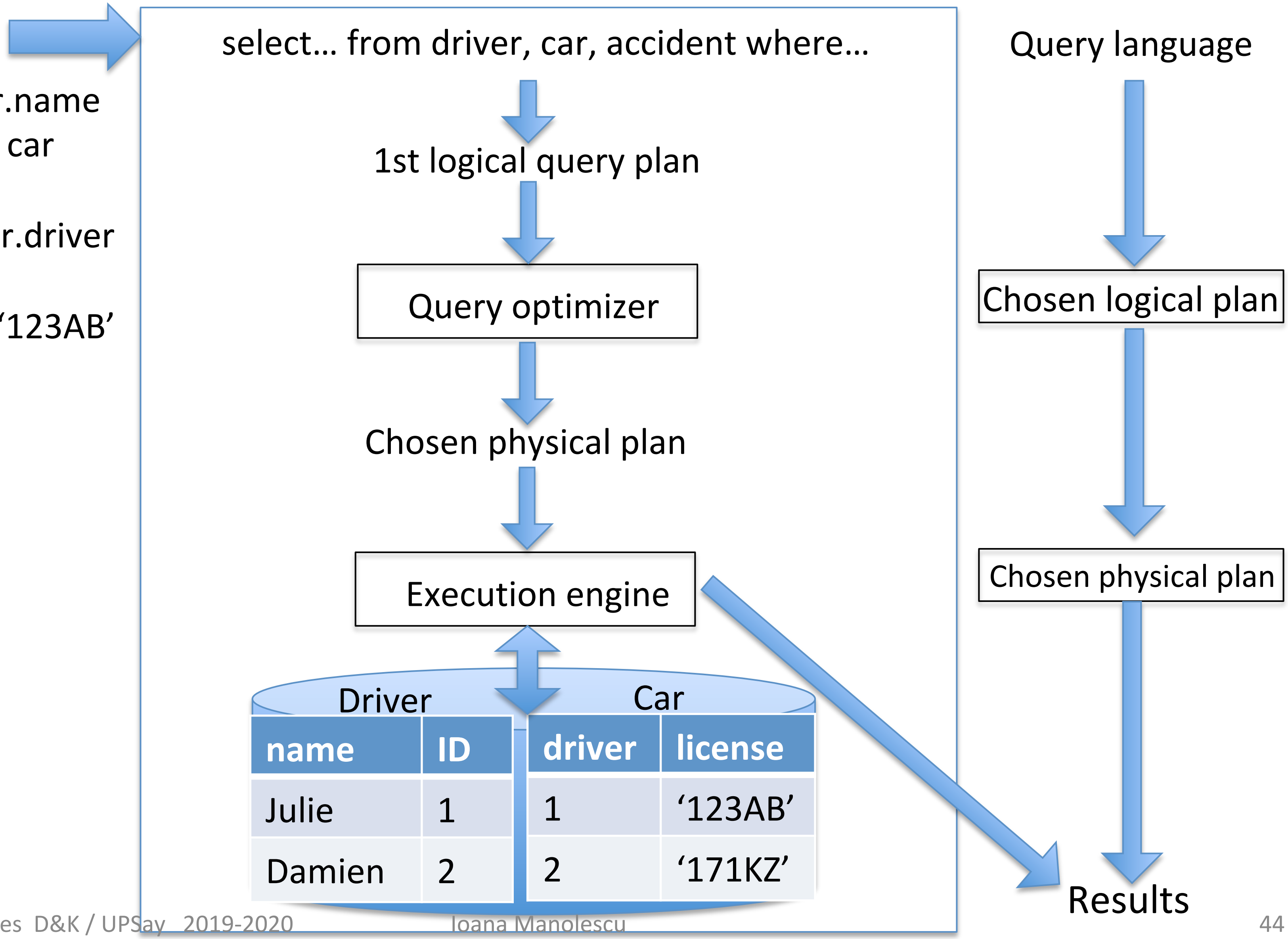
Chosen physical plan

Results

Database internals: query processing pipeline

SQL

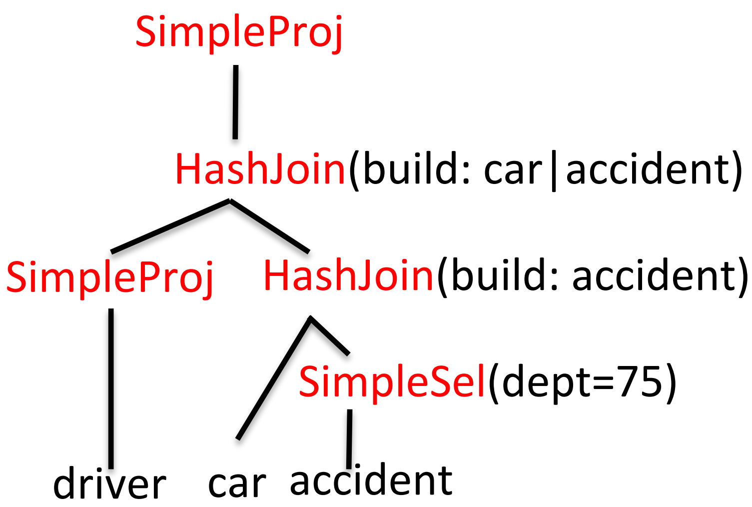
select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license='123AB'



Advanced query optimization techniques: Dynamic Query Optimization

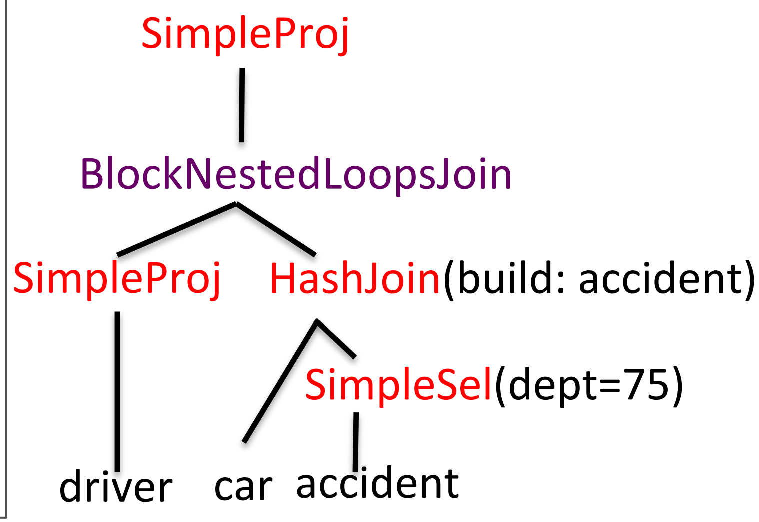
- Sizes (cardinalities) of intermediary results are estimated, which may lead to estimation errors
- A cardinality estimation error may lead to chosing a logical plan and a set of physical operators that perform significantly different from expectation (especially for the worse)

Initially chosen plan:



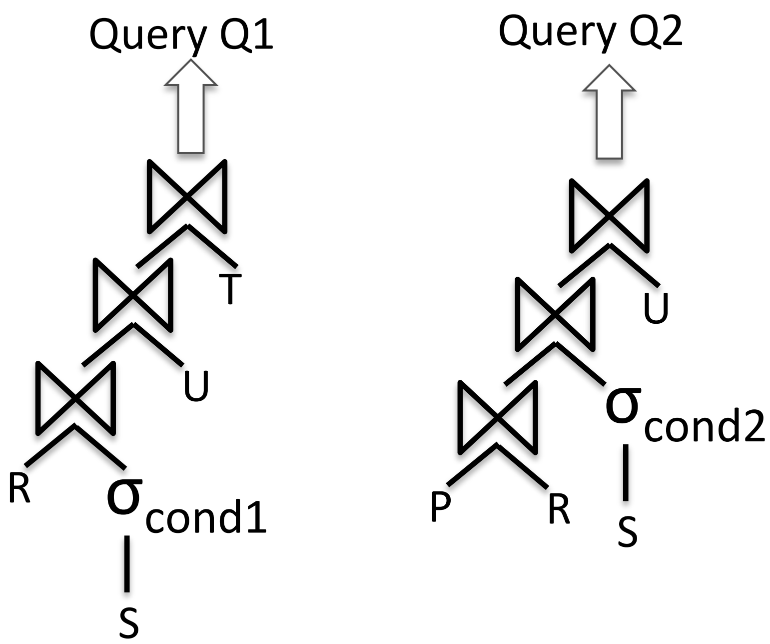
At execution time, we see that the lower HashJoin output is larger than expected: memory insufficient to build

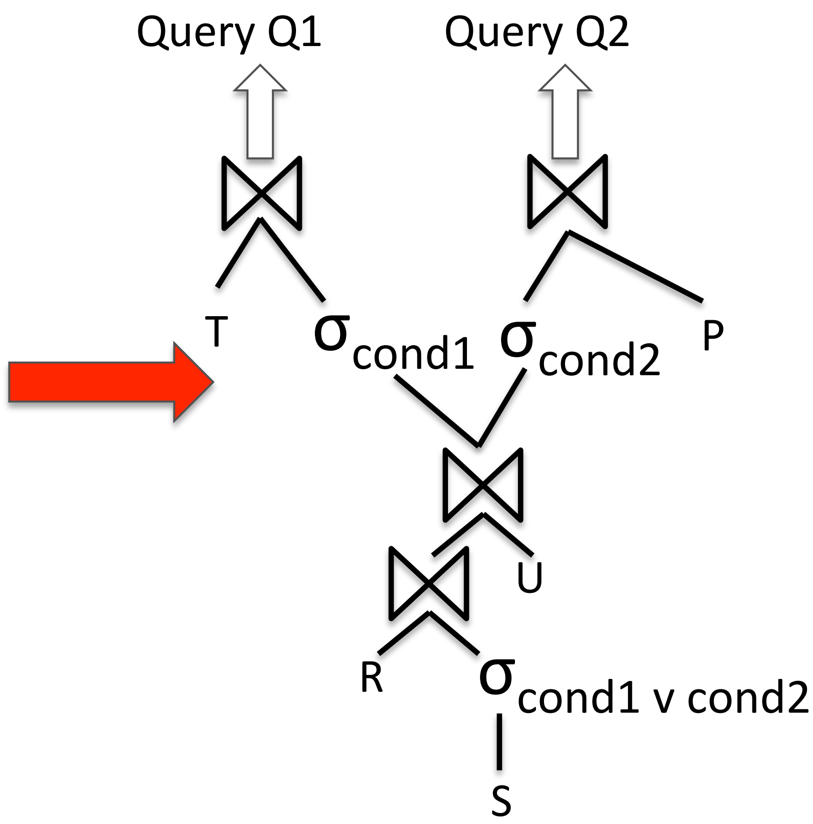
Modified plan:



Advanced query optimization techniques: Multi-Query Optimization

Multiple queries sharing sub-expressions can be optimized together into a single plan with **shared subexpressions**





followed by choice of physical operators

Assume join conditions are the same for R-S and S-U

What's in a database?

SQL update

insert into driver values ('Thomas', 3);
update car set driver=3 where license='123AB';

Database

Accident

Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	‘123AB’
2	‘171KZ’

Database

Accident

Driver

name	ID
Julie	1
Damien	2
Thomas	3

Car

driver	license
3	‘123AB’
2	‘171KZ’

Database updates

- A set of operations atomically executed (either all, or none) is called a **transaction**
- There may be some **dependencies** between the operations of a transaction
  - First read the bank account balance
  - Then write that value reduced by 50€
- A total order over the operations of several concurrent transaction is called a **scheduling**
- The DB component that receives all incoming transactions and decides what operation will be executed when (i.e., <u>global order</u> over the operations of all transactions) is called a **scheduler**

Database updates

- The scheduler is in charge of ordering all operations so that they will appear executed one after the other (serially)

```
T1: BEGIN   A=A+100,   B=B-100   END
T2: BEGIN   A=1.06*A,   B=1.06*B   END
```

```
T1:  A=A+100,  B=B-100,
T2:                             A=1.06*A,   B=1.06*B
```

```
T1:  A=A+100,                    B=B-100
T2:              A=1.06*A,                  B=1.06*B
```

BIG DATA ARCHITECTURES:

WHAT NEEDS TO CHANGE?

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

- Volume requires **distribution**
  - Of the data storage; of query evalution
  - Distribution makes **ACID difficult** (CAP theorem) ✔
    - Complicates concurrency control
    - Replication, eventual consistency
  - Distribution requires efficient, easy-to-use **parallelism**
  - Distribution raises issues of **control** which can lead to single point of failure → **decentralization**
- Velocity requires efficient algorithms
  - Optimize for throughput (rather than response time)
  - Stream processing, in-memory architectures
  - Process-then-store (or process-then-discard)

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

- Variety requires support for
  - **flexible data models**: key-values, JSON, graphs...
  - **different schemas**, and translation mechanisms between the schemas
    - Data integration
  - **several data models** being used together
    - Mediators, Data lakes
- Veracity requires support for reconciliation, data cleaning etc.
  - Similar to single-database setting, but adding source, source confidence, and provenance information

Roadmap for the rest of the course

1. Analysis of large-scale (in particular, distributed) Big Data platforms
   - Focus on: distribution of data and query processing, concurrency control
2. A selection of NoSQL platforms
   - Choice of most popular ones in their class
   - To illustrate variety of data models, and some distribution choices