DATABASE FUNDAMENTALS (RECALL/CRASH COURSE)

Database functionalities

Database Management Systems

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

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**

DATABASE FUNDAMENTALS (RECALL/CRASH COURSE)

Queries and their processing

How are queries processed?

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

SQL → Database → Results

name
Julie

How are queries processed?

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

How are queries processed?

Driver

name	ID
Julie	1
Damien	2

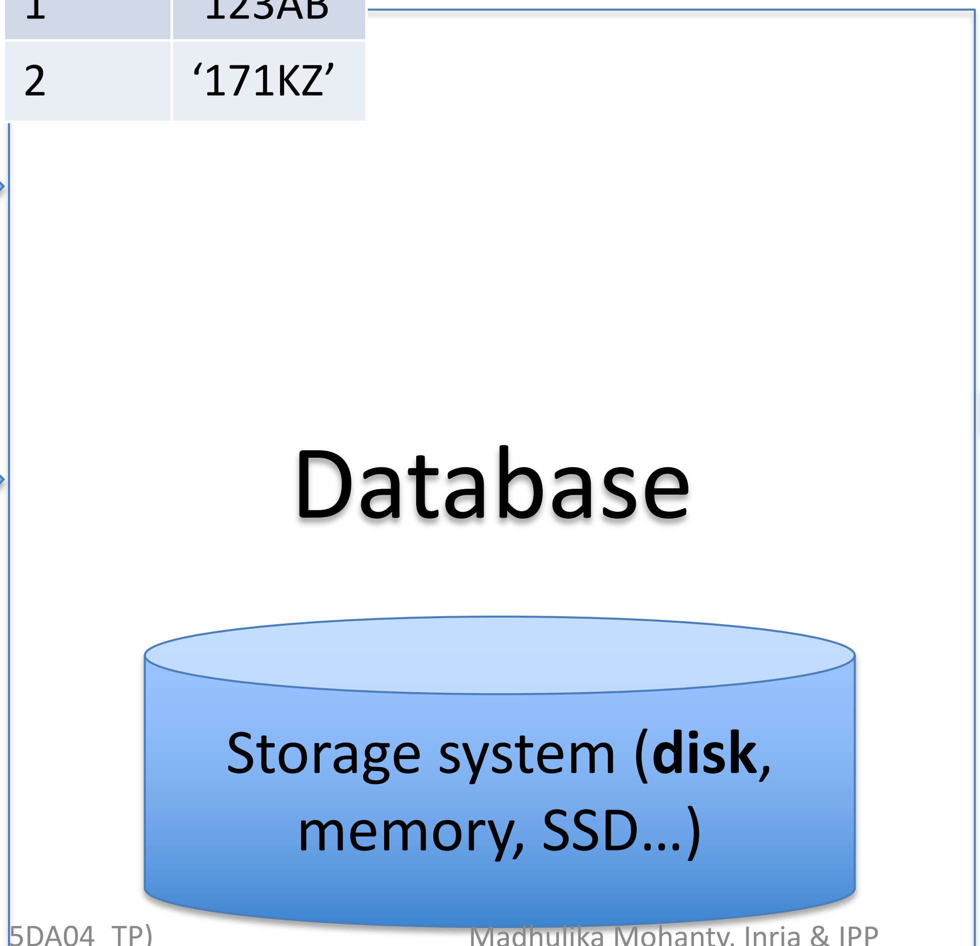
Car

driver	license
1	‘123AB’
2	‘171KZ’

1. Load

2. SQL

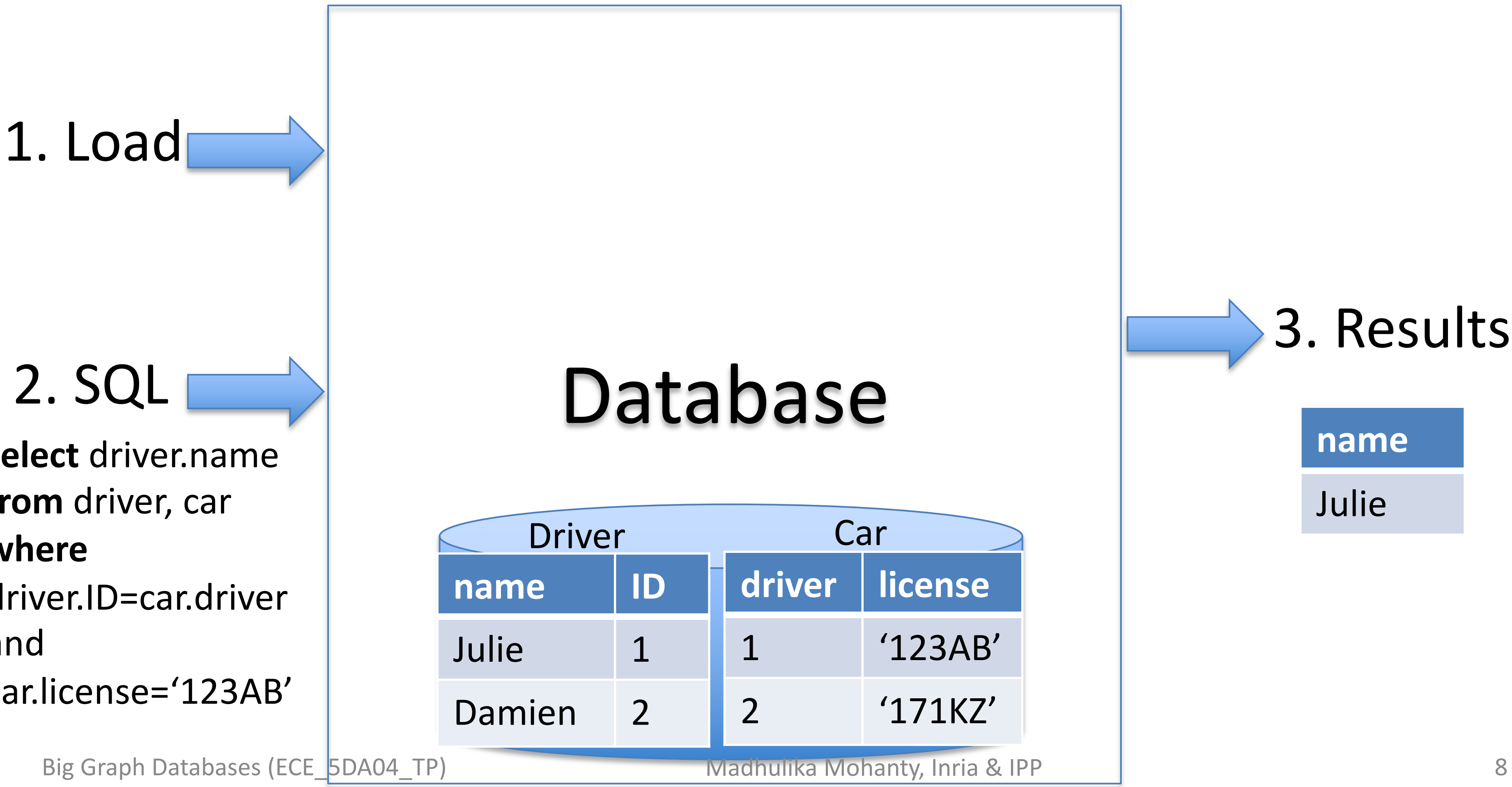
select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license=‘123AB’



3. Results

name
Julie

How are queries processed?



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

Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	‘123AB’
2	‘171KZ’

name
Julie

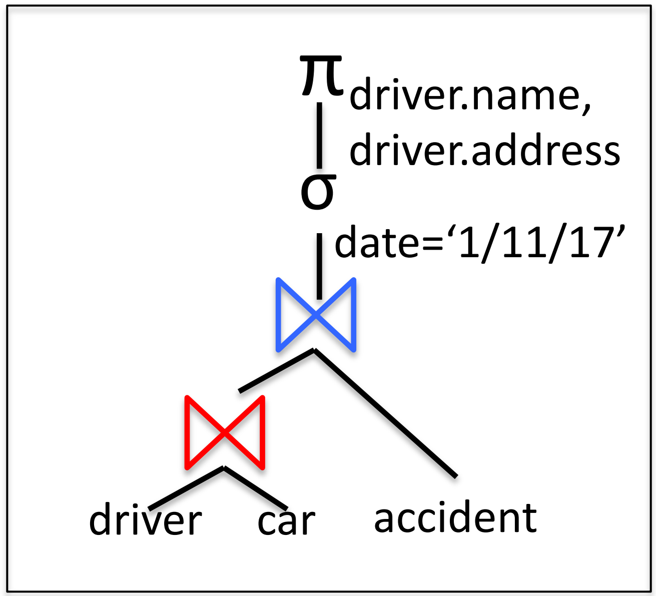
How are queries processed?

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…

Query language



Logical plan

Driver

name	ID
Julie	1
Damien	2

Accident

Car

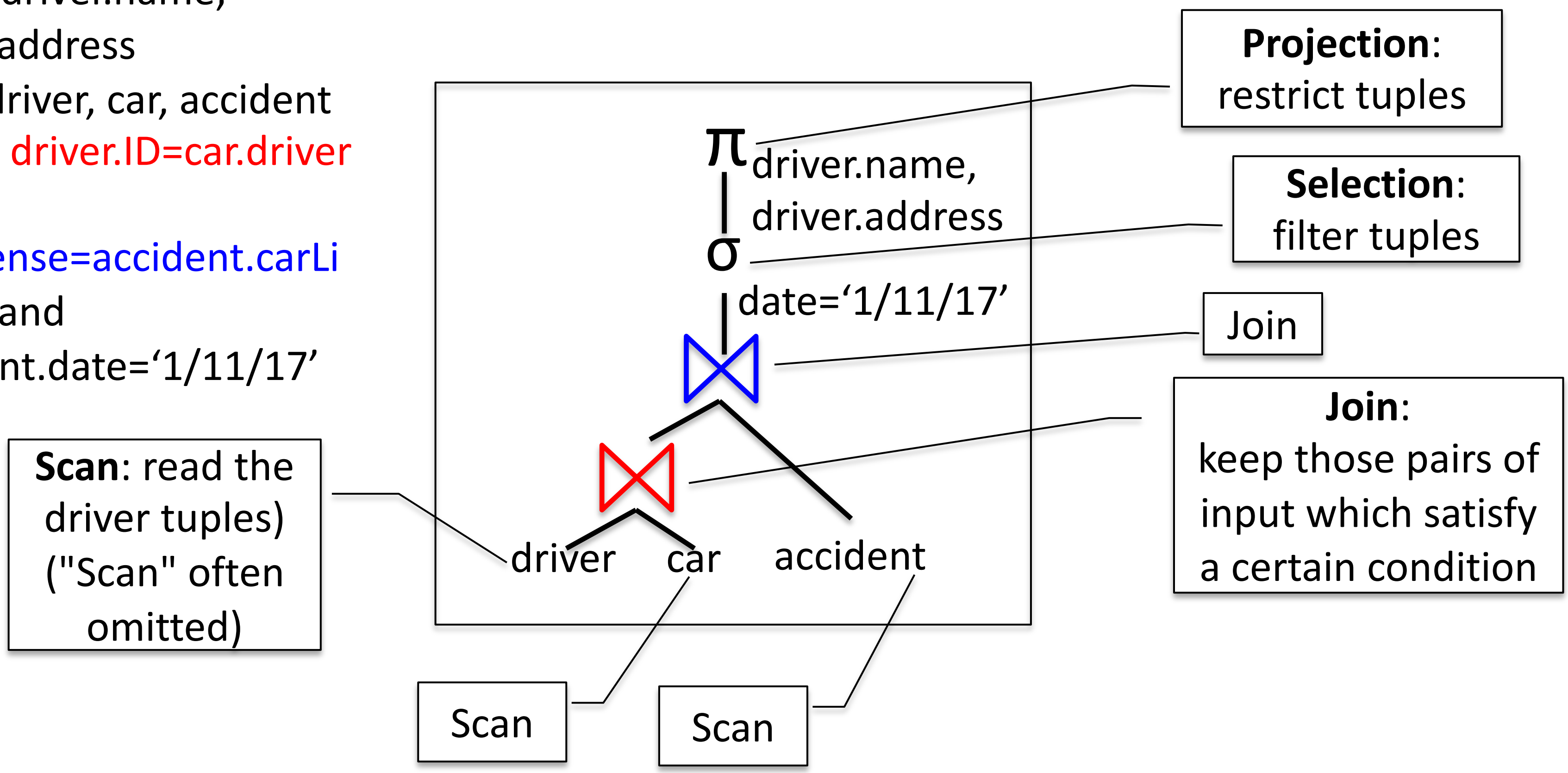
driver	license
1	'123AB'
2	'171KZ'

Results

Logical query plans

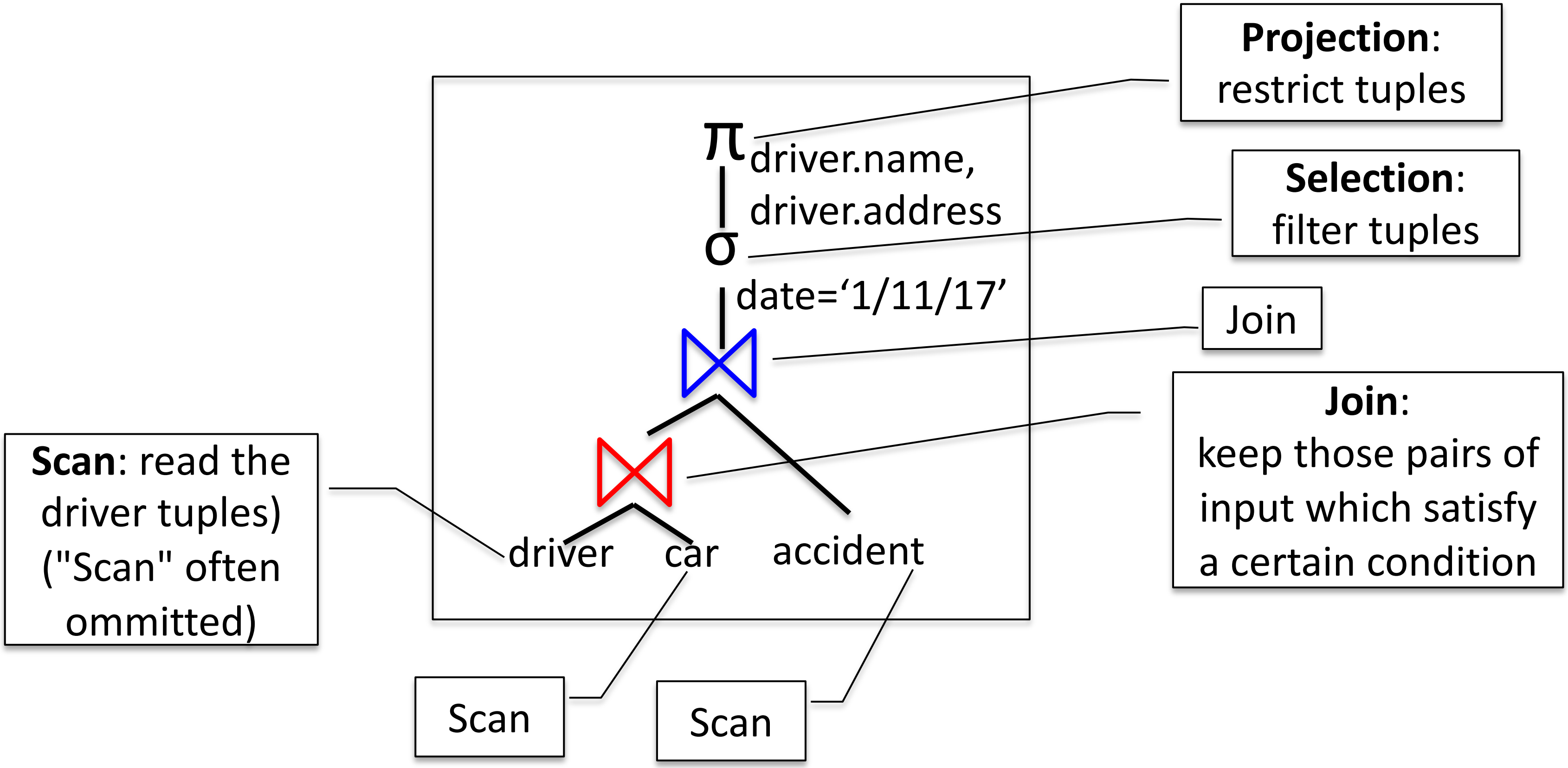
- Trees made of logical operators, each of which specializes in a certain task

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 (a bit later) implement actual algorithms

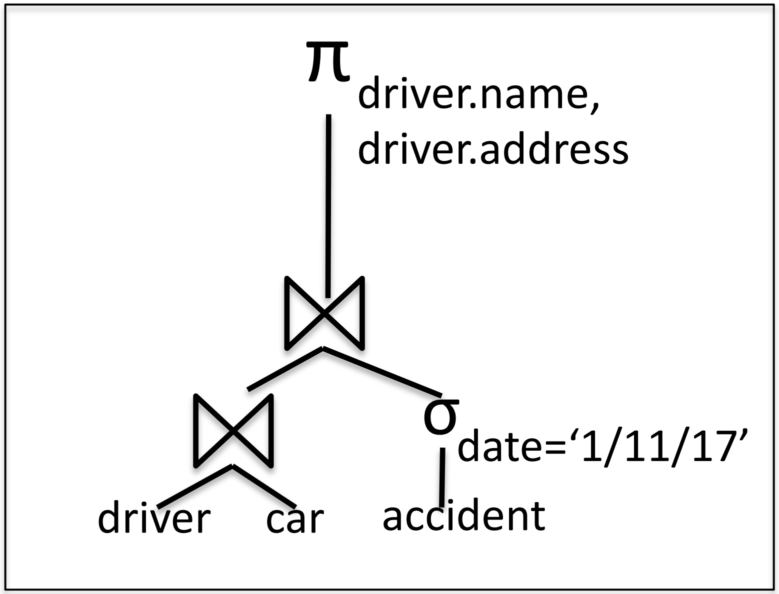


How are queries processed?

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 1

Logical plan 2

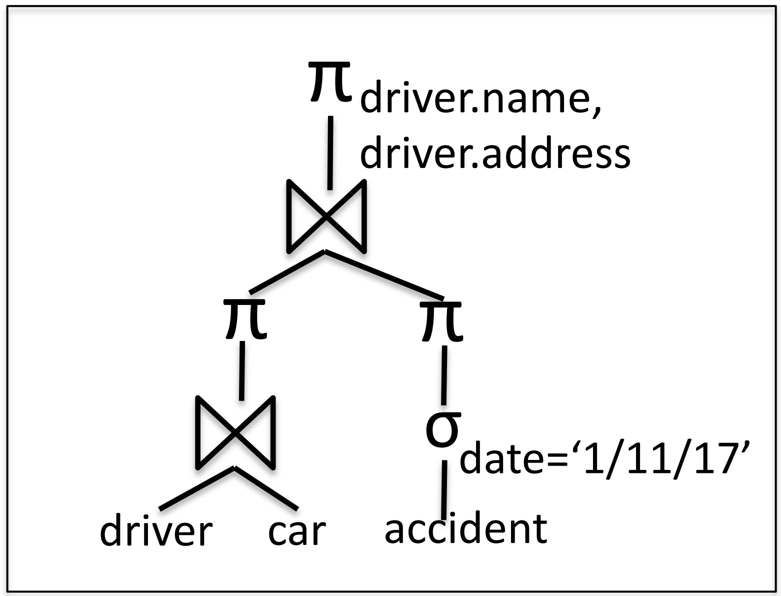
Results

How are queries processed?

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…



……………………………

Accident

Driver

name	ID
Julie	1
Damien	2

Car

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

Query language

Logical plan 1

Logical plan 2

Logical plan 3

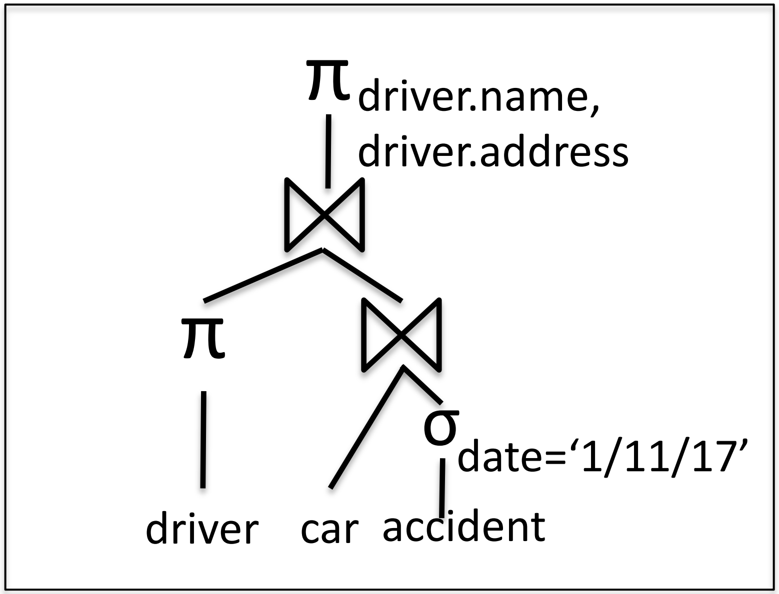
Results

How are queries processed?

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…



……………………………

Accident

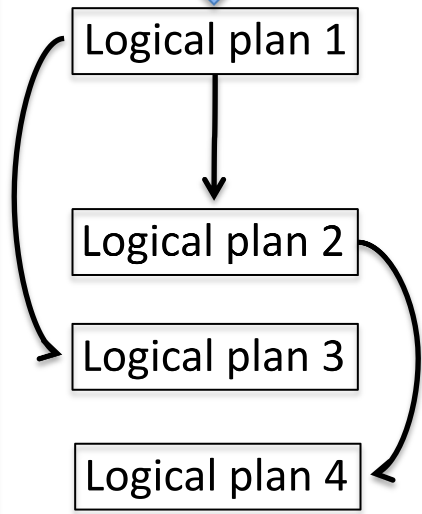
Driver

name	ID
Julie	1
Damien	2

Car

driver	license
1	‘123AB’
2	‘171KZ’

Query language



Results

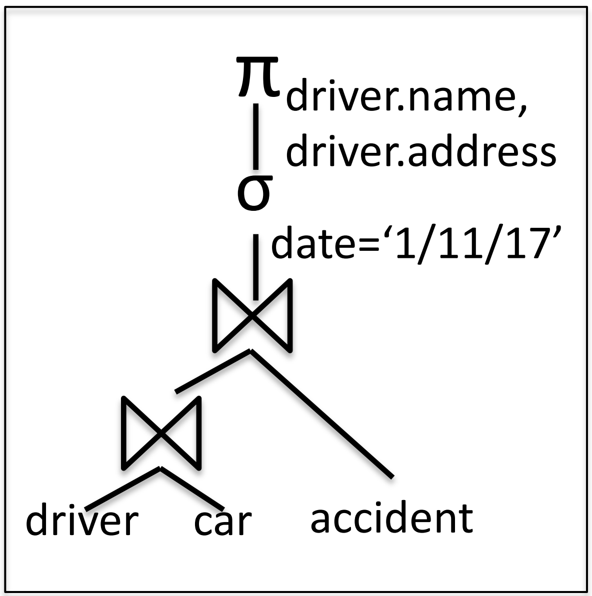
Logical query optimization

- 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? »



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

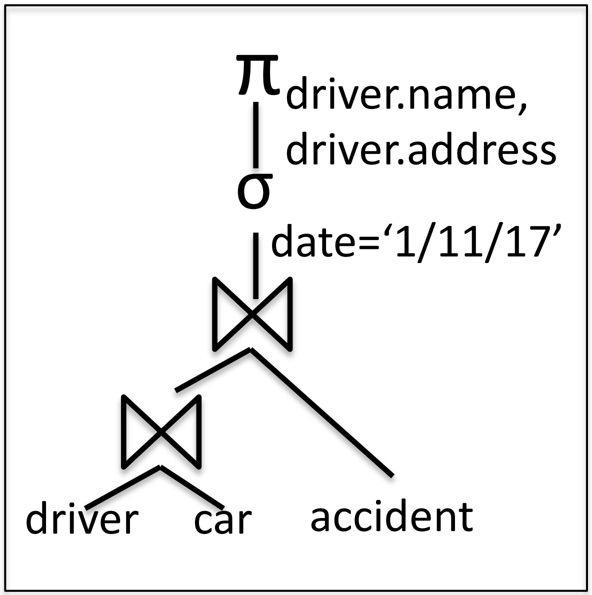
Plan **cost** = the sum of the costs of all operators in a plan

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.



cs, cj, cf constant

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

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

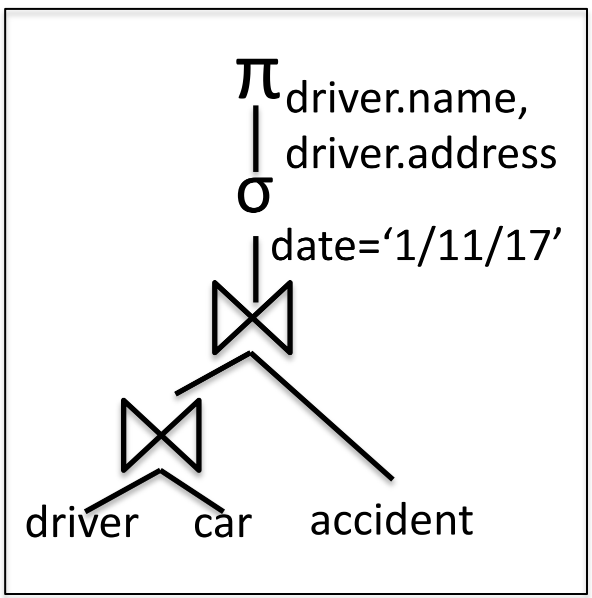
+ cj x $(10^{12} + 2\times10^3)$ ~ cj x 10^{12} ~ $\mathbf{10^{12}}$

Logical query optimization example

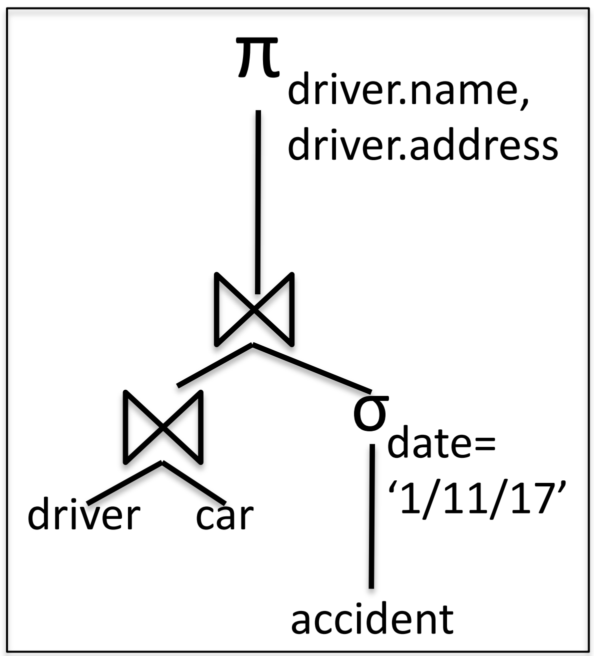
1.000.000 cars, 1.000.000 drivers, 1.000 accidents, 2 cars per accident, 10 accidents on 1/11/17

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

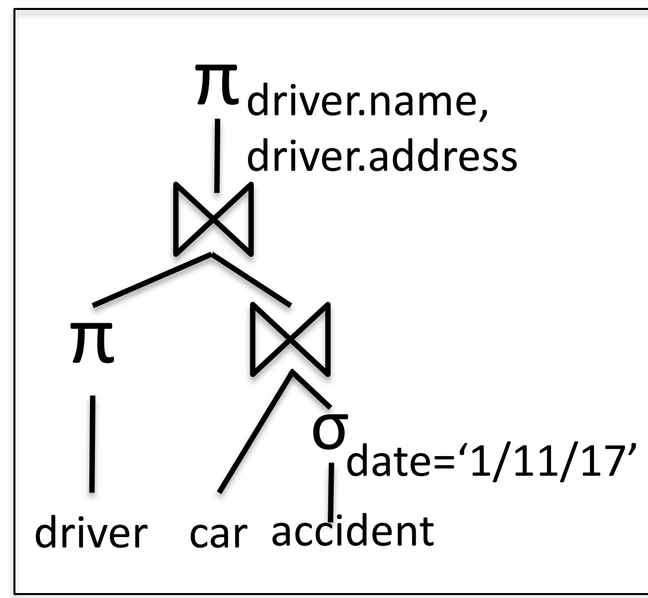
Three plans, same scan costs (neglected below); join costs dominant



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



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



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

Logical query optimization example

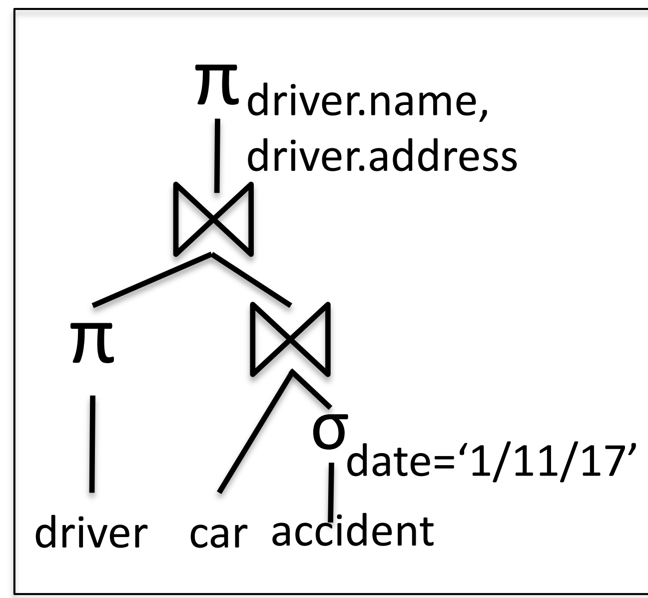
1.000.000 cars, 1.000.000 drivers, 1.000 accidents, 2 cars per accident, 10 accidents on 1/11/17

« 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

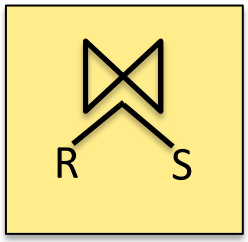
- **Selective data access**
- Typically supported by an **index**
  - Auxiliary data structure, built on top of the data collection
  - Allows to access directly objects satisfying a certain condition



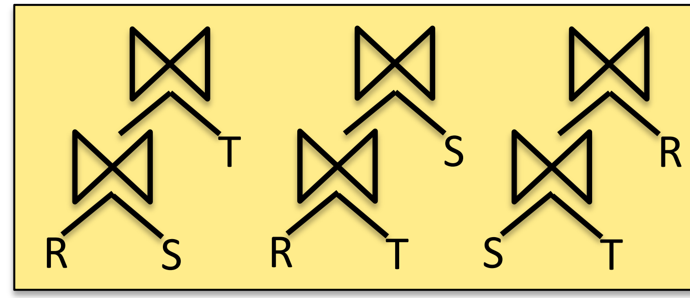
$10^7 + 2*10^7 \sim \mathbf{3*10^7}$

Join ordering is the main problem in logical query optimization

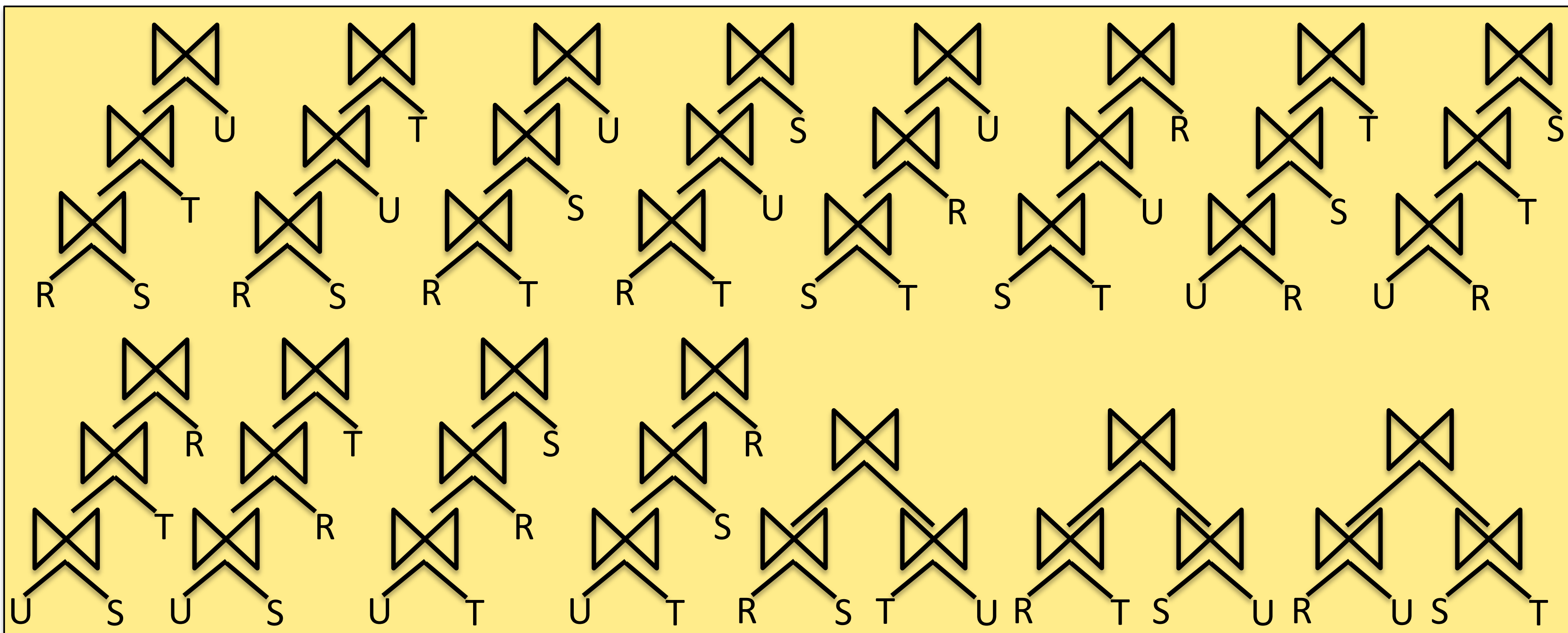
N=2:



N=3:



N=4:

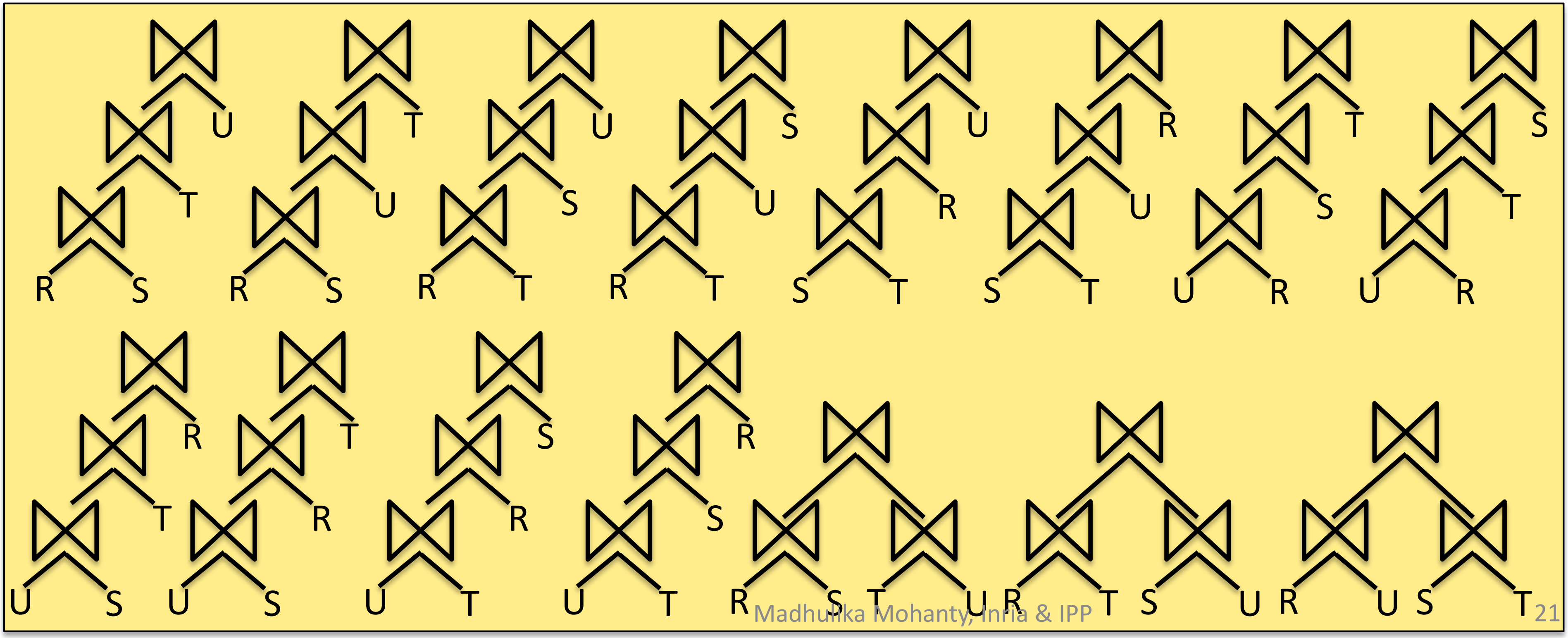


Join ordering is the main problem in logical query optimization

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

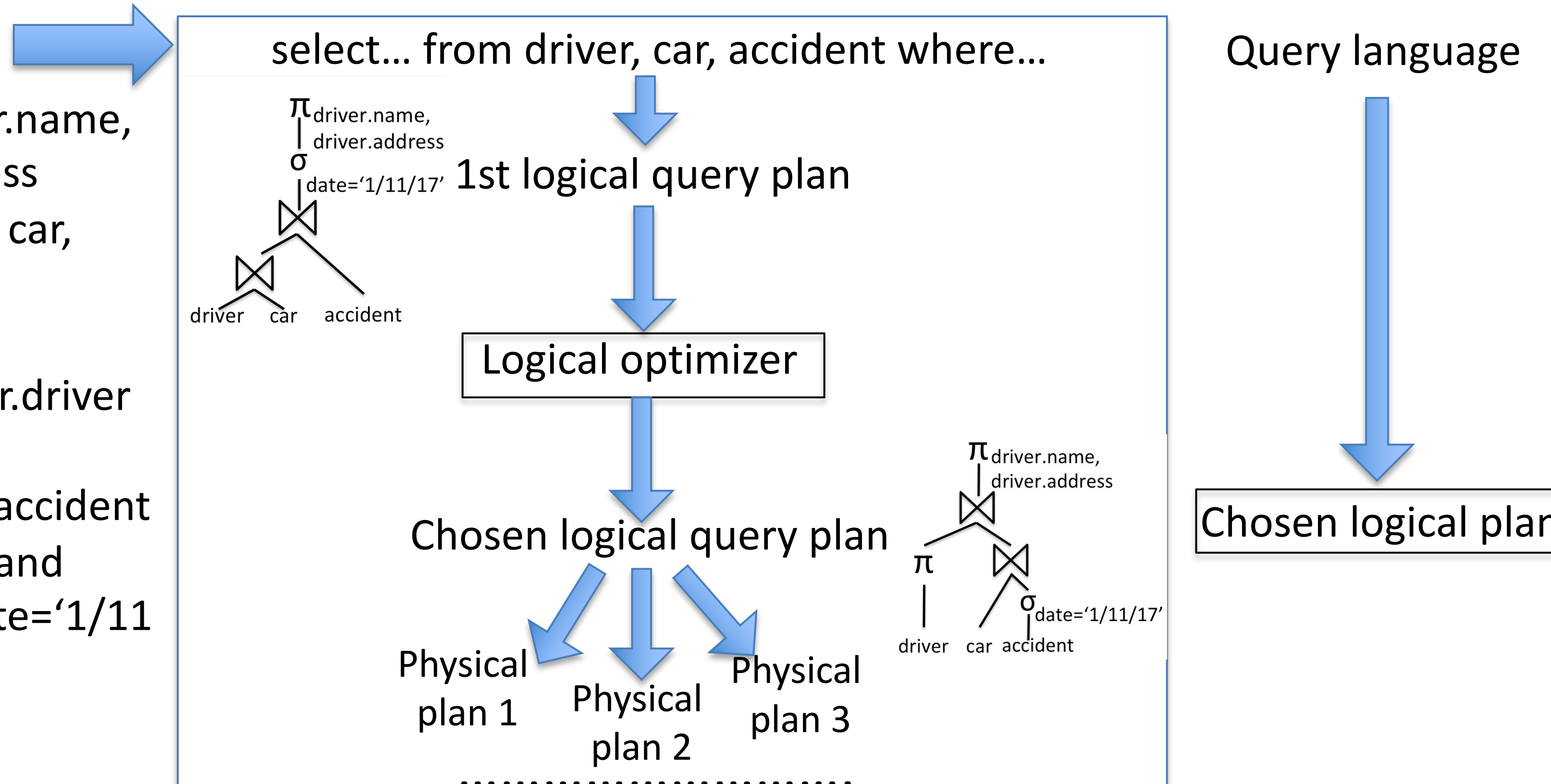
- The **uniform distribution** assumption is frequently wrong
  - Real-world distribution are skewed (popular/frequent values)
- The **independent distribution** assumption is sometimes wrong
  - « Total » counter-example: *functional dependency*
  - Partial but strong enough to ruin optimizer decisions: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

Results

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

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 O(|R|+|S|)

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

Also:

Block nested loops join

Index nested loops join

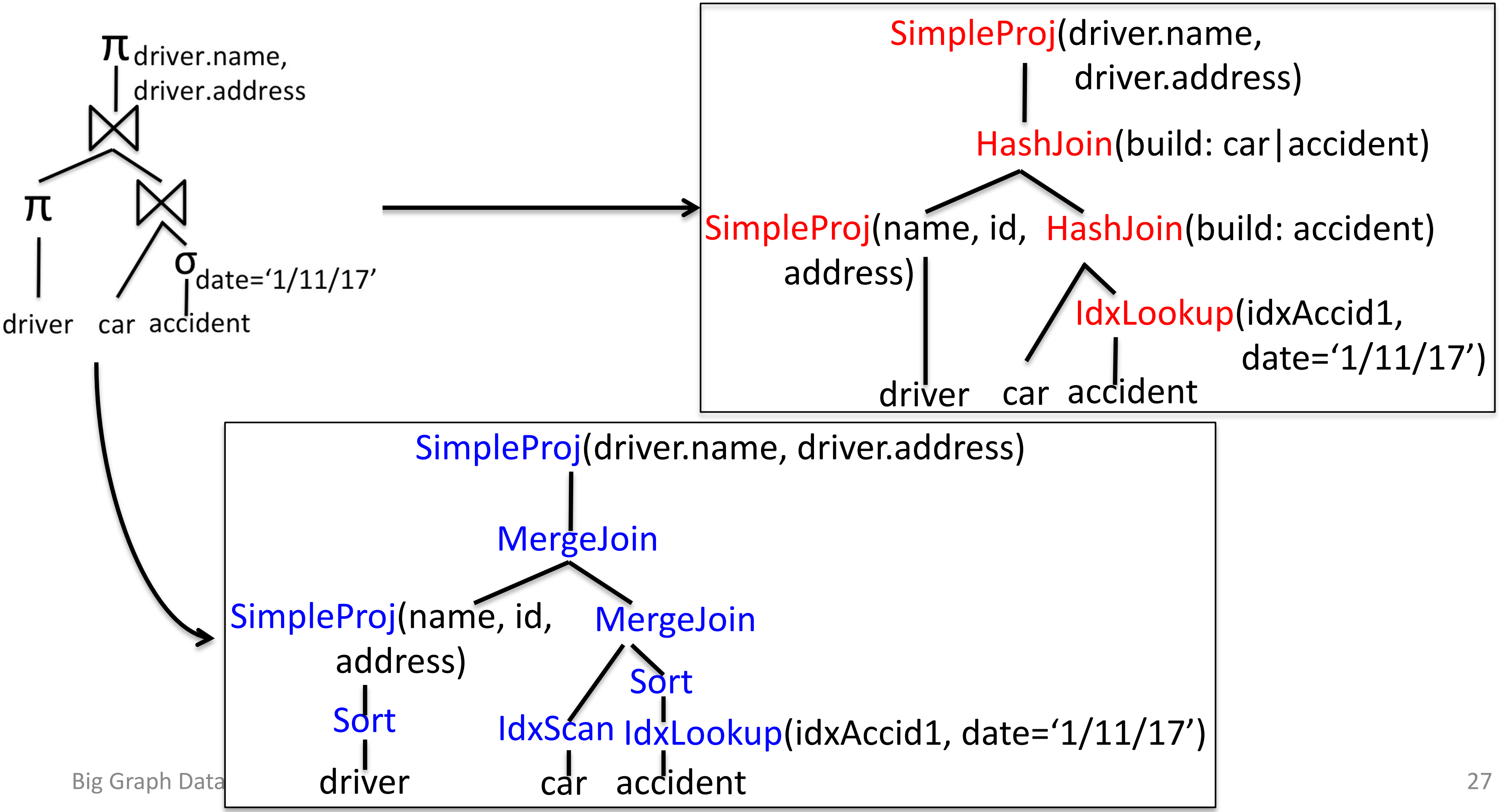
Hybrid hash join

Hash groups / teams

…

Physical optimization

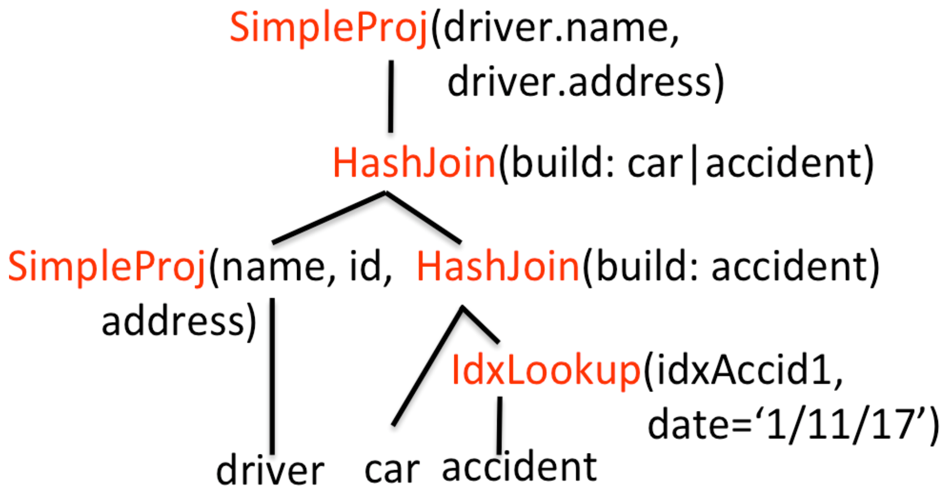
Possible physical plans produced by physical optimization for our sample logical plan:



Physical plan performance

Metrics characterizing a physical plan

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



Query optimizers in action

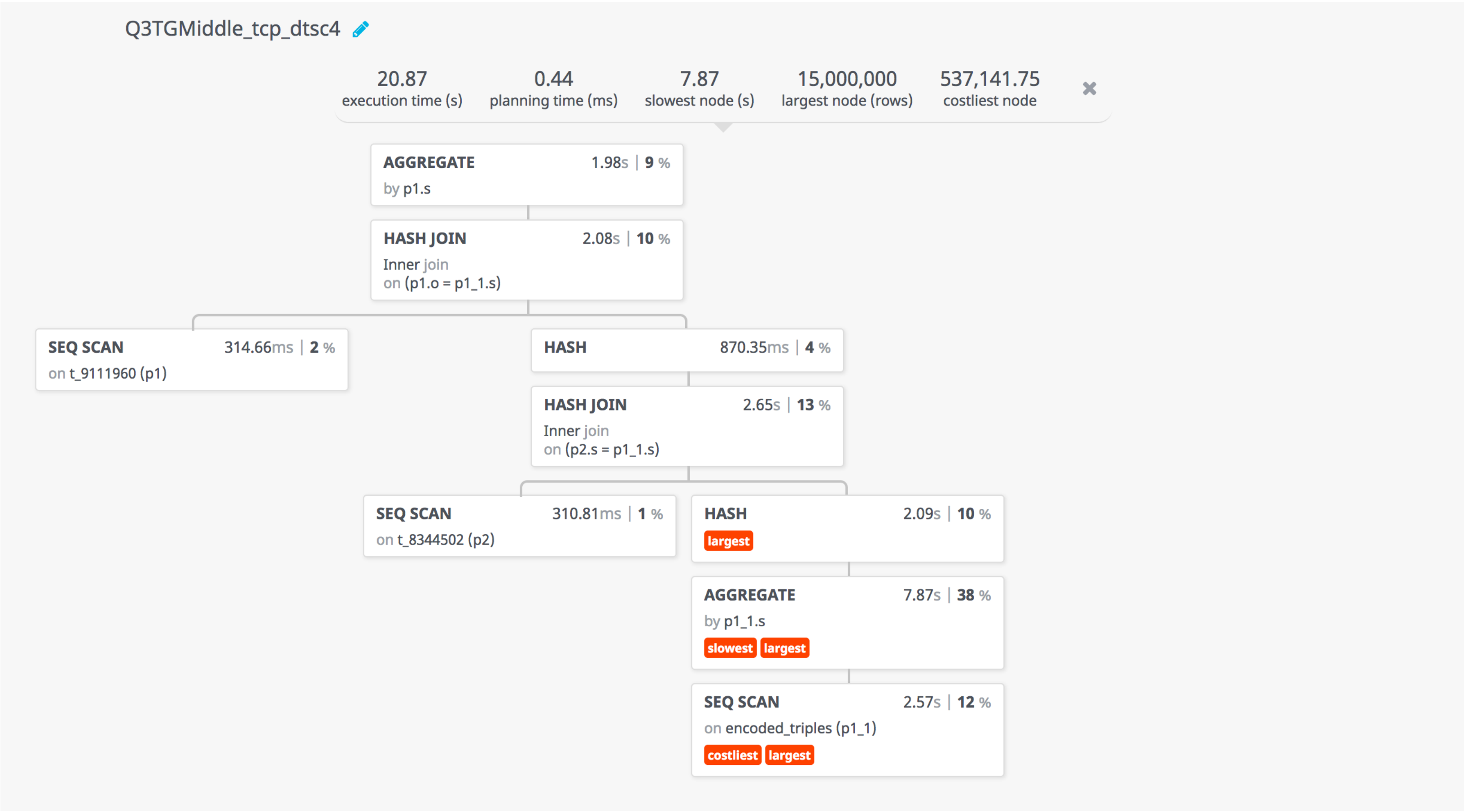
Most database management systems have an « explain » functionality → physical plans. Below sample Postgres output:

```
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)
```

Inspecting query plans

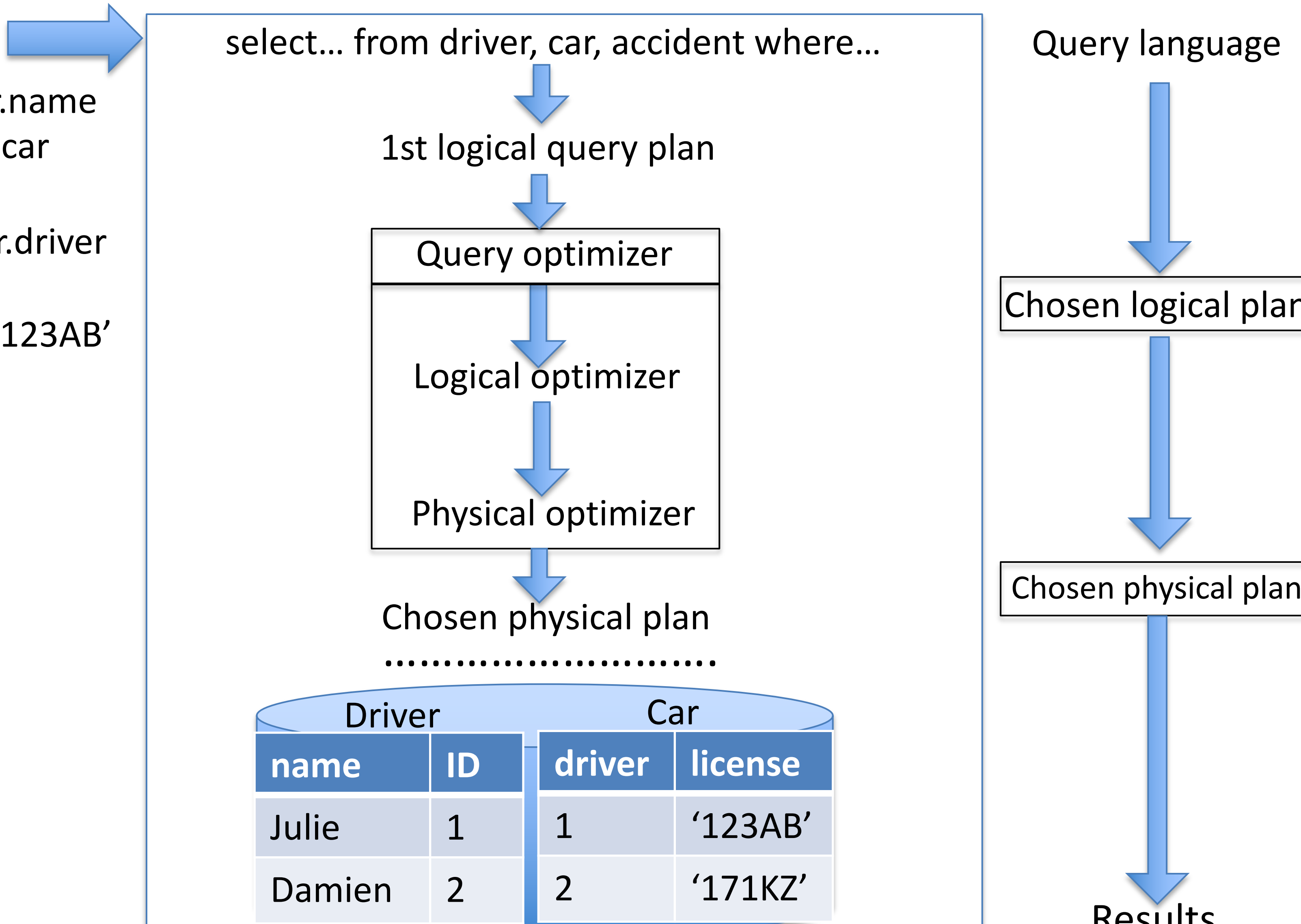
- Can use Dalibo:



Database internal: physical plan

SQL

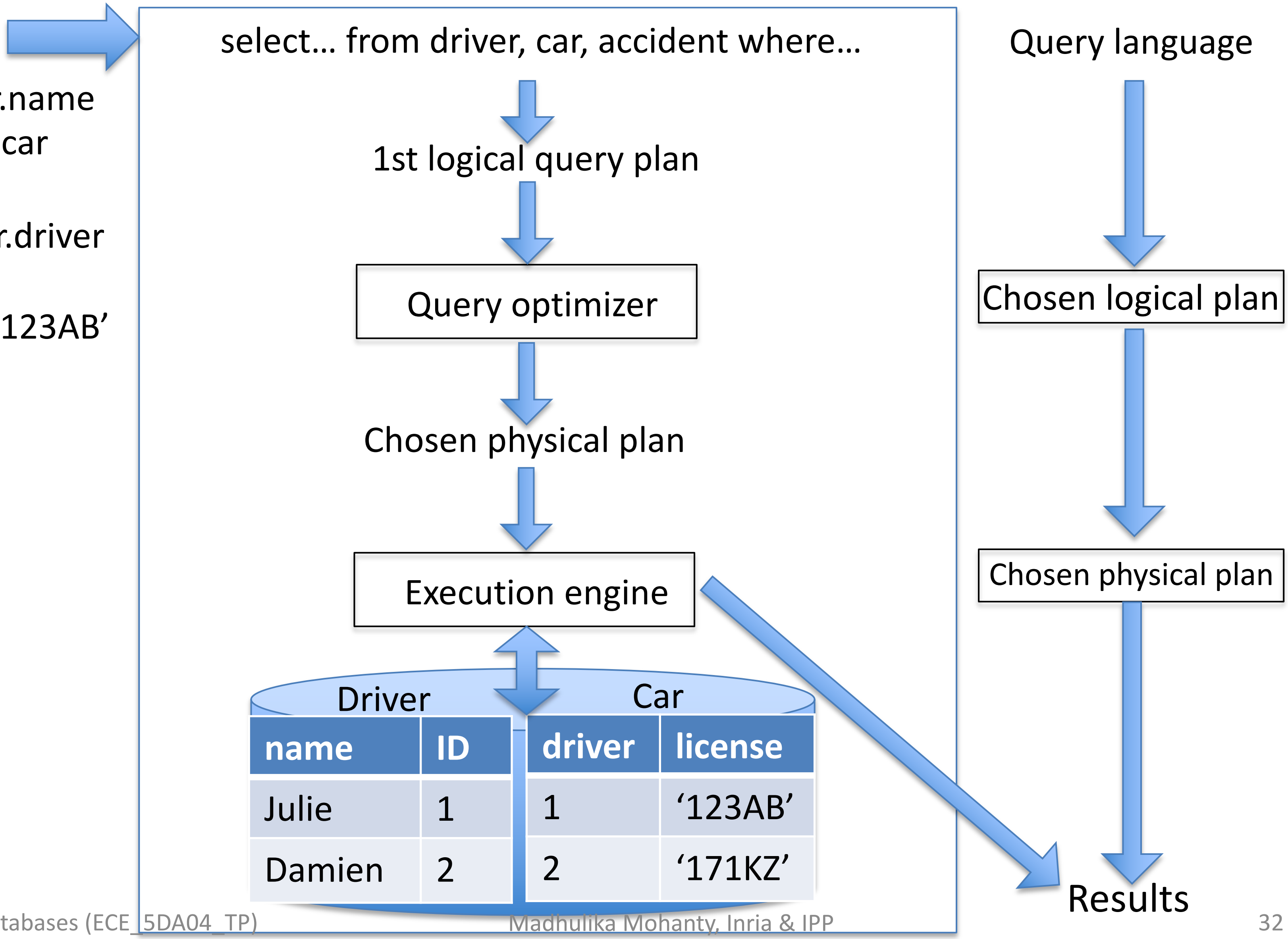
select driver.name
from driver, car
where
driver.ID=car.driver
and
car.license='123AB'



Database internals: query processing pipeline

SQL

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



DATABASE FUNDAMENTALS (RECALL/CRASH COURSE)

Updating the database

What's in a database?

SQL update

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

Database

Driver

name	ID
Julie	1
Damien	2

Accident

Car

driver	license
1	‘123AB’
2	‘171KZ’

Database

Driver

name	ID
Julie	1
Damien	2
Thomas	3

Accident

Car

driver	license
3	‘123AB’
2	‘171KZ’

Fundamental database features

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

Fundamental properties of database stores: 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 »

DATABASE FUNDAMENTALS (RECALL/CRASH COURSE)

Takeaway

Main principles behind correct and scalable data management...

... core of the database management systems:

1. **Declarative query language** allows users to just state what they want
2. For one query there are several **logical plans**; for each, several **physical plans**
   – Optimizer picks **best plan**
3. **ACID** properties crucial for "faith in the system" ("my salary, payments, and social security are within a reliable system")