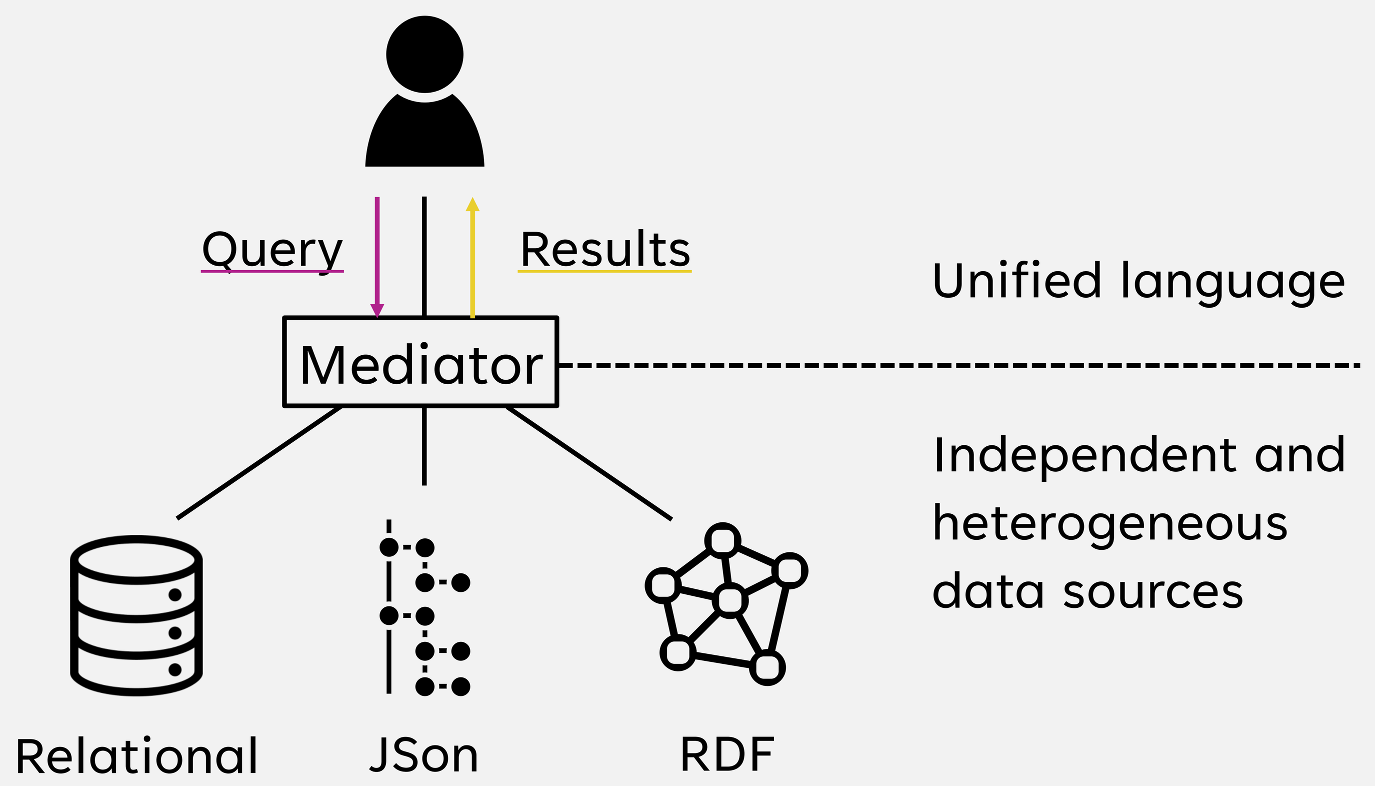
Join optimization in data integration systems

Maurin Gilles, Maxime Buron

Data integration

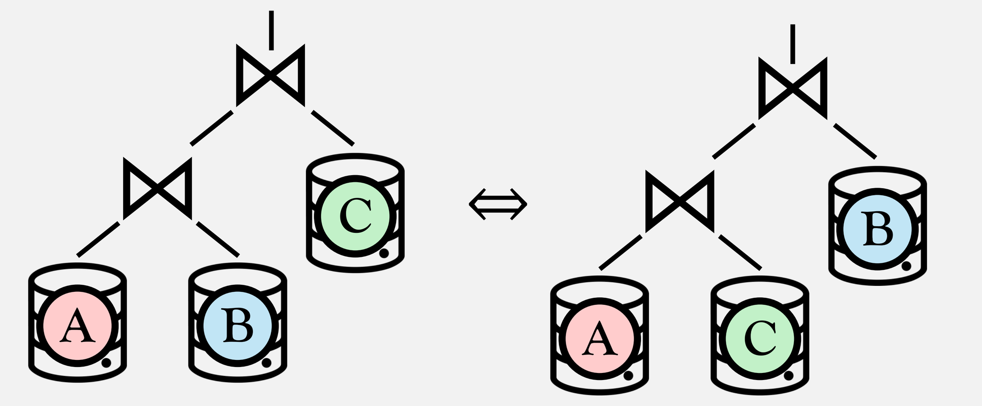


Query plans

User queries are parsed as **logical** expressions over relational algebra. They describe in which order the operators are executed.

I need all x values that appear in A, B and C at the same time.

$$\pi_{A.x}((A \bowtie_{A.x=B.x} B) \bowtie_{A.x=C.x} C) \Leftrightarrow \pi_{A.x}((A \bowtie_{A.x=C.x} C) \bowtie_{A.x=B.x} B)$$



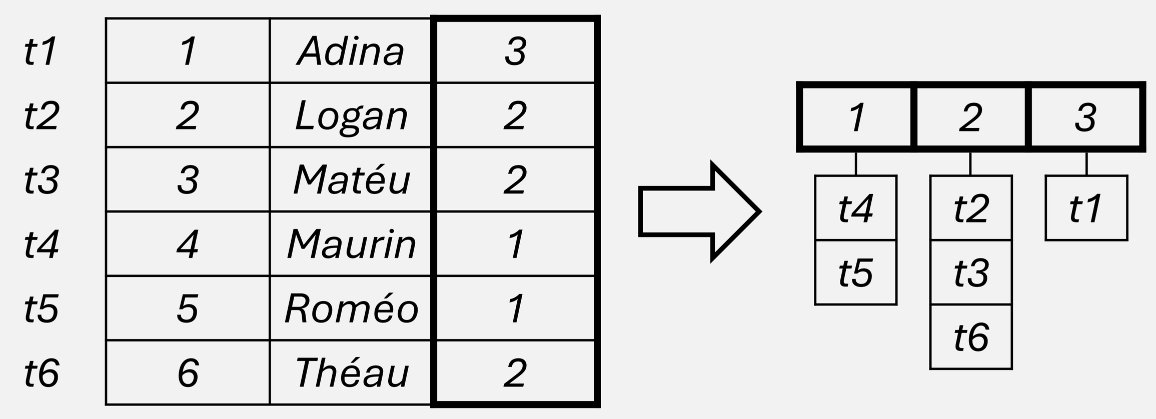
Each query can be resolved by a space of equivalent plans

Search for a "best plan"

Moreover, there are several implementations for each operator

Pipelined hashjoin

Use of **hashtables** on specific columns.



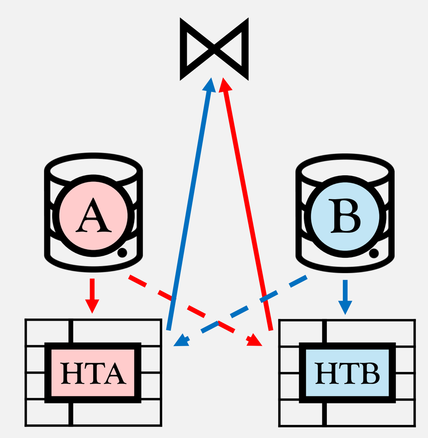
Algorithm.

Table **T** : A or B, alternates at each iteration

While (remaining tuples in A or B)

1. Take a **tuple from T**
2. **Insert** this tuple in the **hashtable of T**
3. **Search** for matches of this tuple in the **other hashtable**
4. If results are found, return them

Visualisation.



Advantages.

> **No initialization time**, compared to usual hashjoin. The size of the sources doesn't impact the time we wait before the first results are found.
> Under certain conditions, **hashtables can be used for multiple joins**.
> It is **symetrical**, thus reducing the space of equivalent plans.

Implementation.

We have implemented the pipelined hashjoin on a java project called **tatooine**. It allowed us to make experiments by simulating multiple plans with multiple sources.

Results.

As expected, the pipelined hashjoin is faster to find the first tuples. However, our implementation is about twice slower than the usual hashjoin in most cases.

Why we use it: We use pipelined hashjoins because it enables to apply **plan transformations during execution** by keeping the same hashtables.

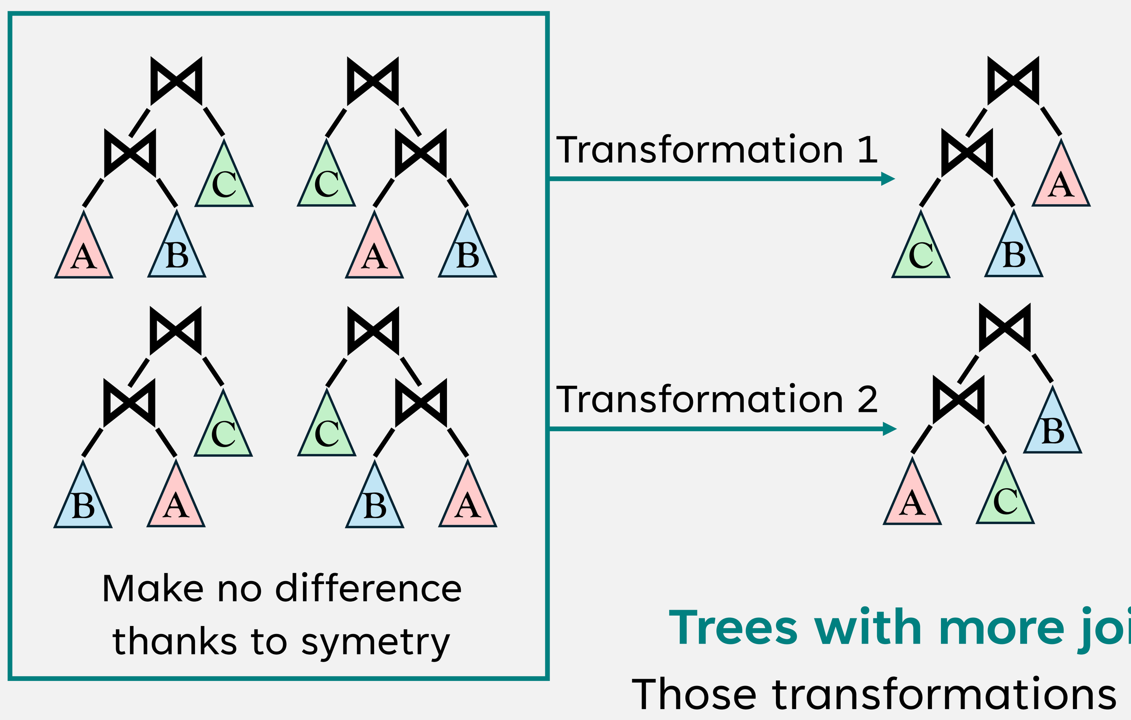
Plan transformation

We propose to explore the space of join trees by applying local transformations, which can be applied during the execution under certain hypotheses.

Hypotheses.

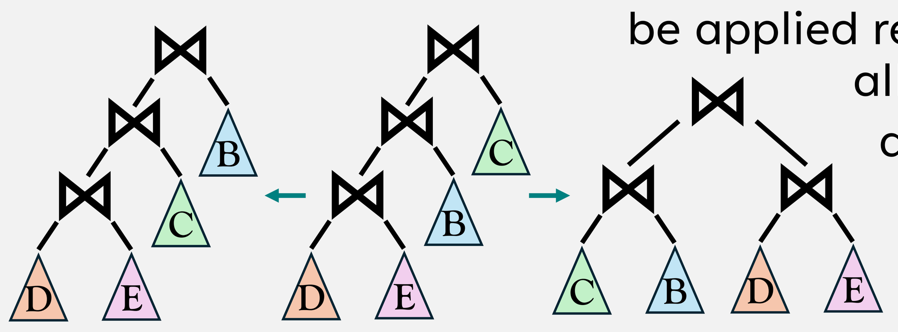
> Joins are implemented using pipelined hashjoin.
> Join tree represents a star query, meaning the join conditions are equalities on the same attributes : *(A.x = B.y = C.z)*

Two transformations.



Trees with more joins.

Those transformations can be applied recursively to find all the plans easily accessible from a certain one.



Optimizing plans

Selectivity.

$$\alpha_{AB} = \frac{|A \bowtie B|}{|A||B|} \Leftrightarrow |A \bowtie B| = |A||B|\alpha_{AB}$$

Where $|P|$ is the **cardinality** of a subplan, and $\alpha_{AB} \in [0,1]$

Cost model.

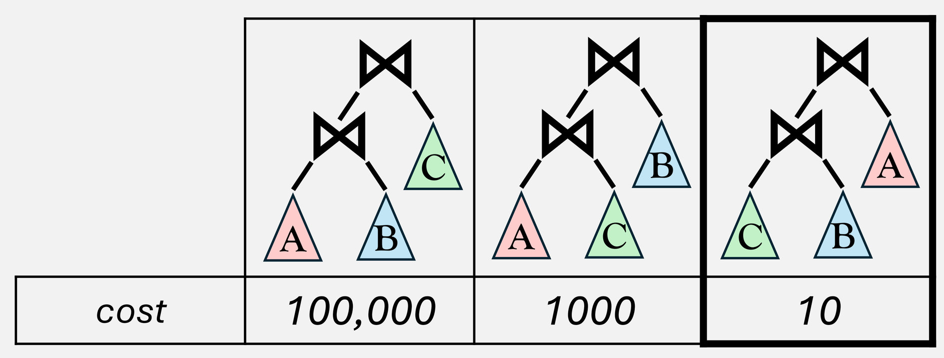
The objective is to execute a plan having a minimal **cost** : the total amount of tuples generated during the execution.

$$cost = \sum |P_i \bowtie P_j| \quad \text{for } P_i \bowtie P_j \text{ internal join in the tree.}$$

Example.

Let $|A| = 10$; $|B| = 10{,}000$; $|C| = 10{,}000$

$\alpha_{A,B} = 1$; $\alpha_{A,C} = 0.01$; $\alpha_{B,C} = 0.0000001$



Optimization cycle.

Measure selectivity during execution → Find a better plan → Apply transformation

Future work

- Fix performance issues in the implementation.
- Find plan transformations for joins with join conditions other than star queries.
- Take into account the properties of the different sources in the cost model.
- Implement the cost model and the plan transformations during execution.
- Determine when the selectivity measured during execution is reliable enough, and when its difference with estimated selectivity justify a plan transformation.