

Query Optimisation (Part 2)

Big Data Engineering
(formerly Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

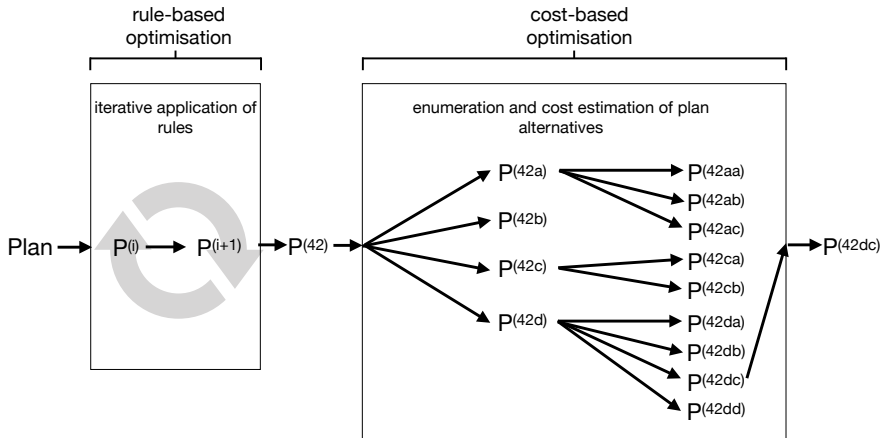
June 1, 2023

Cost-Based Optimisation

3. transformed logical plan
↓ cost-based optimisation
4. physical plan

- (good) query optimiser enumerate a large number of possible plans and try to estimate the runtime of these plans using cost models
- only the plan with the expected shortest runtime is then executed

Difference to Rule-Based Optimisation (see Part 1)



- in **both** components, plans are created based on rules
- difference: cost-based optimisation uses cost estimates in order to decide whether certain transformations are applied

Different Dimensions of Cost-Based Optimisation

Decisions on a **logical** level:

- i. Which join order to take?

Examples: Which join order has less runtime: $R \bowtie (S \bowtie T)$ or $(R \bowtie S) \bowtie T$ or ...?

Decisions on a **physical** level:

- ii. Which physical operator to use?

Examples: Hash-based join, sort-based join or XY join?
(cf. logical vs physical operator discussion)

- iii. Use index or not? If yes, which index to use?

Examples: Scan relation or use binary search? Which type of index to use? Hash-based, tree-based or ... ?
(cf. [Picasso-Notebook](#))

- iv. Which resources to use for which sub-plan?

Examples: How many threads to use where? Which part of the plan gets how much computing time/main memory?

iii. Use Index or Not?

To calculate the results of a query of the form $\sigma_P(R)$, we basically have only two options:

Brute-Force (aka Scan)

We examine each tuple in R and check if P is true, and if it is, we add the tuple to the result set.

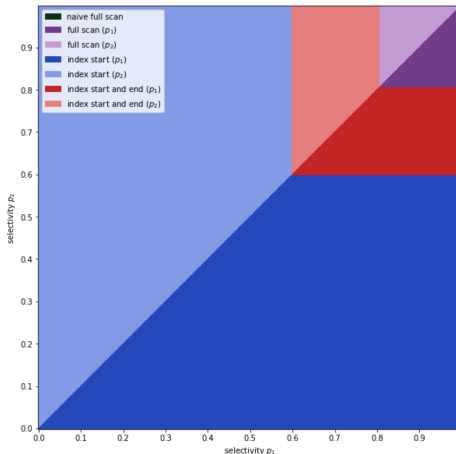
VS

Index (aka Index Scan)

We organise the content of R such that to evaluate the query we do not have to examine each individual tuple to determine whether a tuple belongs to the result of a query.

Picasso Plan Diagrams in Python

```
In [35]: fig, ax = plt.subplots(figsize=(10, 10))  
  
         plot_plan_diagram(plans, plan_labels, 300, ax, color_list)
```



[https://github.com/BigDataAnalyticsGroup/
bigdataengineering/blob/master/Picasso.ipynb](https://github.com/BigDataAnalyticsGroup/bigdataengineering/blob/master/Picasso.ipynb)

i. Join Order: Tree Structure of the Plan

The dimension “join order” consists of different subproblems:

Tree Structure of the Plan

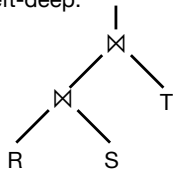
For n input relations, there are C_{n-1} many binary trees with n leaf nodes.

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \sim \frac{4^n}{n^{1.5}\sqrt{\pi}}, n \geq 0 \quad (\text{Catalan number})$$

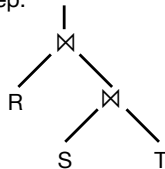
The Catalan numbers beginning with C_0, C_1, \dots are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, ...

Three Input Relations: $C_2 = 2$ Plans

left-deep:

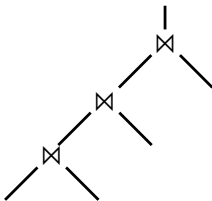


right-deep:

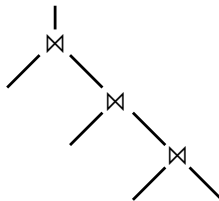


Four Input Relations: $C_3 = 5$ Plans

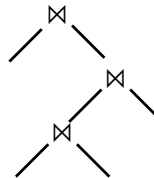
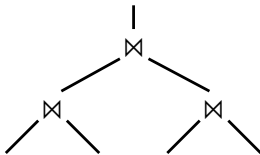
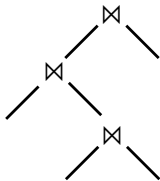
left-deep:



right-deep:



bushy:



i. Join Order: Order of Input Relations

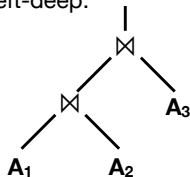
Order of Input Relations

For n input relations there are $n!$ many ways to arrange them and then assign them to the different tree structures of a plan.

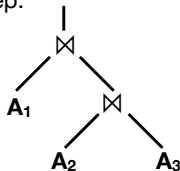
If we consider the plan structure **and** the order of the input relations, we are already at $C_{n-1} \cdot n!$ many plans.

Plan Structure and Order of Input Relations

left-deep:



right-deep:



6 input sequences in each case:

A_1	A_2	A_3
R	S	T
R	T	S
S	R	T
S	T	R
T	R	S
T	S	R

$n = 3$ relations: $C_{n-1} \cdot n! = C_2 \cdot 3! = 2 \cdot 6 = 12$ different plans (theoretically...)

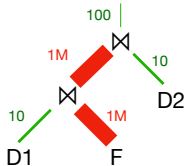
Quick Question

Reality check: Does it even make sense to list all possible join orders?

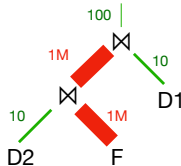
If no join condition is defined between two relations, these relations must be connected by a cross product! And that will 'most likely' be (too) expensive. Why should we even consider this possibility in the enumeration?

Well, is that true?

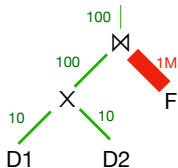
Plans with Cross Products



Costs_{Intermediate results} = 1,000,100



Costs_{Intermediate results} = 1,000,100



Costs_{Intermediate results} = 200

For this example, the plan with the cross product is dramatically better than the ones only considering joins along foreign keys.

Be Careful:

Ignoring cross products in planning may lead to suboptimal plans!

Join Selectivity

Join Selectivity

By join selectivity we mean the ratio of the size of the join result to the size of the cross product of the input relations:

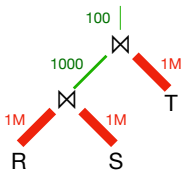
$$sel_{R \bowtie S} = \frac{|R \bowtie S|}{|R \times S|} \leq 1$$

If for R and S no join predicate exists, it follows $sel_{R \bowtie S} = sel_{R \times S} = 1$.

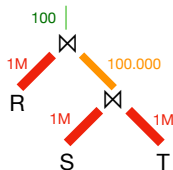
Example:

$|R| = |S| = |T| = 10^6$, $sel_{R \bowtie S} = 10^{-9}$, $sel_{S \bowtie T} = 10^{-7}$,

$sel_{(R \bowtie S) \bowtie T} = sel_{R \bowtie (S \bowtie T)} = 10^{-16}$.



Costs_{Intermediate results} = 1100



Costs_{Intermediate results} = 100.100

These two join orders have different costs.

Predicates

Expressions defined in WHERE or ON, for example, are transformed to CNF and we refer to the “literals” in the following as predicates of a query.

Reminder

A formula in CNF is a conjunction of clauses. Clauses are disjunctions of literals.

Example: (Formula in CNF)

$$\underbrace{(e = 42 \vee c < 3)}_{\text{Clause}} \wedge \underbrace{a \cdot b = 42}_{\text{Clause}} \wedge \underbrace{a + b = c}_{\text{Clause}}$$

Literal Literal Literal Literal Literal

Arity of a Predicate

Arity of a Predicate

Let R_p be the set of relations whose attributes are referenced in the predicate p . Then, the arity of p is defined as follows:

$$||p|| = |R_p|$$

In other words, the arity denotes the number of attributes **from different relations** referenced in the predicate.

Examples:

$$||R.e = 42|| = 1$$

$$||R.c = T.a|| = 2$$

$$||T.a \cdot R.b = 42|| = 2$$

$$||R.a + T.b = S.c|| = 3$$

All examples make the assumption that the variables come from different relations.

Special Cases of Arity

Filter Predicate

Any predicate fp with $||fp|| = 1$ is called filter predicate.

Join Predicate

Any predicate jp with $||jp|| = 2$ is called join predicate.

In the following, we will only consider predicates with arity 1 or 2.

Examples:

$||R.c < 3|| = 1$ (filter predicate)

$||R.c = T.a|| = 2$ (join predicate)

Join Graph (aka Query Graph)

Join Graph (aka Query Graph)

The join graph $G = (V, JP, FP)$ of a query Q with predicates P consists of:

1. The set of relations $V = \{R_1, \dots, R_k\}$ (vertices),
2. the set of join predicates JP (annotation of the edges)

$$JP = \left\{ (R, jp, R') \mid jp \in P \wedge \|jp\| = 2 \wedge attr(jp) \subseteq ([R] \circ [R']) \right\}, \text{ and}$$

3. the set of filter predicates FP (annotation of the vertices)

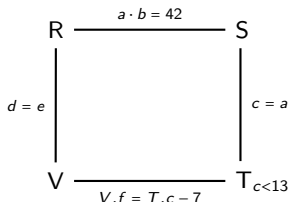
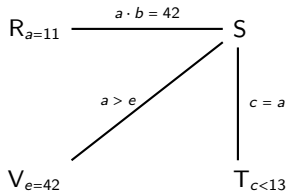
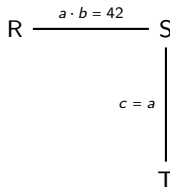
$$FP = \left\{ (R, fp) \mid fp \in P \wedge \|fp\| = 1 \wedge attr(fp) \subseteq [R] \right\}.$$

For predicates with arity > 2 , the definition of the join graph can be extended to a hyper-graph.

Examples for Join Graphs

Examples:

three join graphs on the same schema generated by three different queries

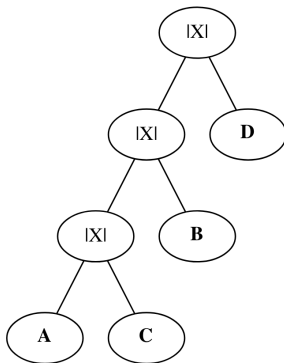


To avoid misunderstandings, predicates can also be written with dot-notation, e.g. $V.f = T.c - 7$. Vertices with filter predicates can alternatively be notated with a filter operator: $\sigma_{c<13}(T)$ instead of $T_{c<13}$.

Plan Enumeration in Python

Find the cheapest plan for our join graph.

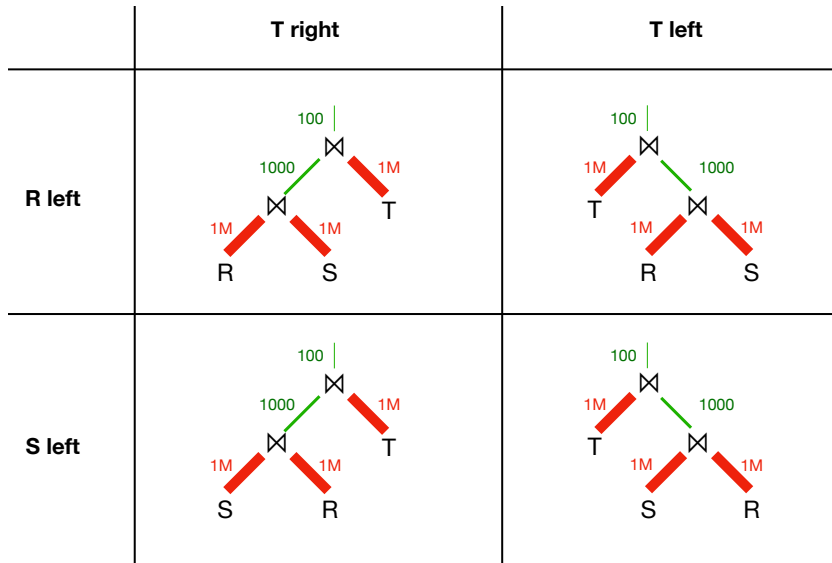
```
In [15]: plan = find_cheapest_plan_exhaustive(the_join_graph, CostFunctions.size_of_inputs)
display(gv.Source(draw_query_plan(plan)))
cost = compute_cost(plan, the_join_graph, CostFunctions.size_of_inputs)
print(f'The cheapest plan is {plan} with cost {cost:,}.')
```



The cheapest plan is ('A', 'C', 'B', 'D') with cost 61,000.

GitHub: [Plan Enumeration.ipynb](#)

Join Order and Commutativity



These four plans have the same costs in this cost model:

Cost_{Intermediate results} = 1100.

HashJoin Algorithm

HashJoin (Relations R and S , Join Predicate $JP := r.x == s.y$)

1. HashMap hm ;
2. Relation $result = \{\}$;

Add all tuples from the left input R into the HashMap:

3. For all r in R :
4. $hm.insert(r.x, r)$;

Search all tuples from the right input S in the HashMap:

5. For all s in S :
6. $RES = hm.probe(s.y)$;

If the HashMap has entries for the key $s.y$:

7. If $RES \neq \emptyset$:

Add partial result RES to total result:

8. $result = result \cup (RES \times \{s\})$;
9. Return $result$;

So what?

The HashJoin in different cost models:

Cost model: Total runtime

$$\text{Costs}_{\text{Total runtime}}(\text{HashJoin}) \approx c_1 \cdot |R| + c_2 \cdot |S|$$

There is a linear cost in the size of the input relation for building the HashMap on R. Then another linear cost in the size of S for all tuple lookups from S in the HashMap.

- The exact values of c_1 and c_2 depend on many factors: load factor of the HashMap, type of the HashMap, data distribution, etc.¹
- If c_1 and c_2 are similar in size, HashJoin(R,S) or HashJoin(S,R) makes not much difference. Then the costs of the join are approximately **symmetric**.
- We have ignored the selectivity of the join here. A combination of $\text{Costs}_{\text{Total runtime}}$ with $\text{Costs}_{\text{Intermediate results}}$ is reasonable.

¹Further Reading: Stefan Richter, Victor Alvarez, Jens Dittrich.

A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. PVLDB/VLDB 2016.

The HashJoin in Main Memory (Storage Costs Only)

Cost model: HashJoin in main memory

$$\text{Costs}_{\text{Main memory}}(\text{HashJoin}) \approx c_3 \cdot c_4 \cdot |R|.$$

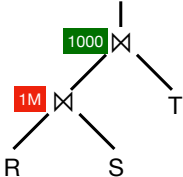
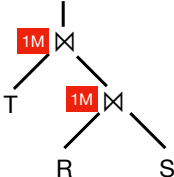
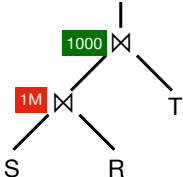
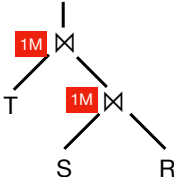
c_3 := Overhead of the HashMap for storing an entry.

c_4 := Size of a tuple in R .

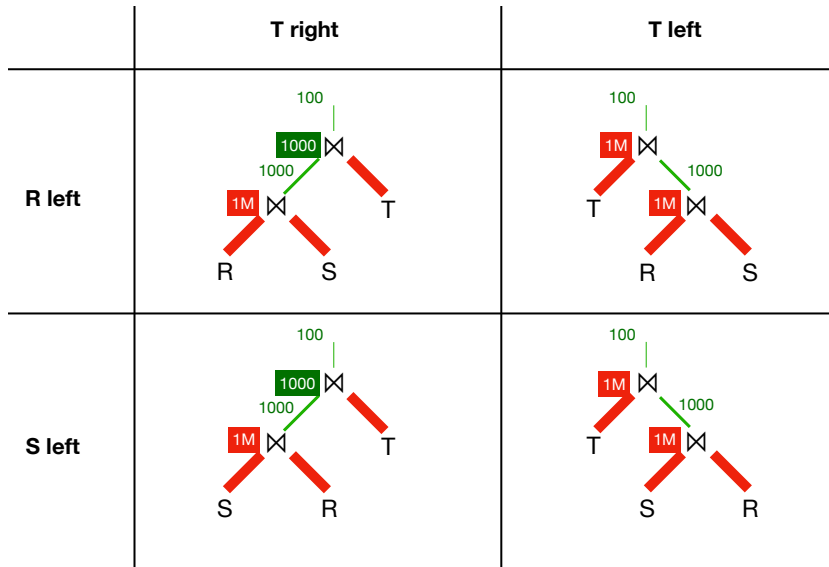
There are linear costs for building the HashMap on input relation R . Then again ..., nope, that's it!

- The size of S plays **no** role in this cost model.
- It is advantageous in this cost model to take the smaller input relation as the left input relation.
- The cost of the HashJoin is **asymmetric** in this cost model.

Join Order and Main Memory Costs

	T right	T left
R left		
S left	 <p>Costs_{Main memory} = $c_3 \cdot c_4 \cdot 1.001.000!$</p>	 <p>Costs_{Main memory} = $c_3 \cdot c_4 \cdot 2.000.000!$</p>

Both Cost Models Combined



Both Cost Models Combined

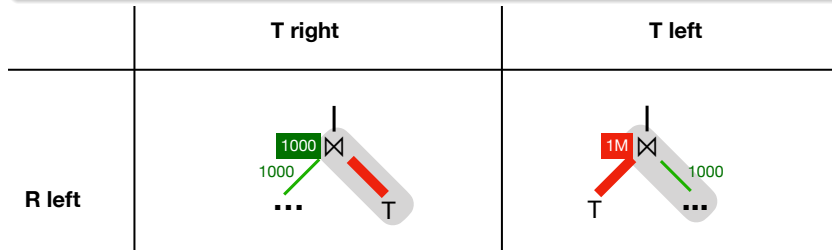
Cost model: Combination of main memory and intermediate results

$$\begin{aligned}\text{Costs}_{\text{Combined}}(\textit{HashJoin}) := \\ c_5 \cdot \text{Costs}_{\text{Main memory}}(\textit{HashJoin}) \\ + c_6 \cdot \text{Costs}_{\text{Intermediate results}}(\textit{HashJoin}).\end{aligned}$$

c_5, c_6 : Constant factors

Cost Model vs Real Costs: Comparison of Model and Reality...

If T is joined from the right, we have modelled that this does not cause any additional **storage cost** for T. Does that reflect reality?



In both cases, the right-hand input of the join, (which contains 1 million tuples for “T right”), is passed as parameter, so “somehow” moved through main memory!

HashJoin (Relations L and R, Join Predicate $JP := l.x == r.y$)

...

HashJoin Algorithm: Costs in Main Memory

What does it mean when relations are passed **as parameter** to a function like L and R here (left and right input)?

- Are the complete relations materialised in main memory (or on another storage medium)?
- What if one (or both) of the relations is (are) very large?

What happens in the HashJoin algorithm?

```
234 ▾ class Equi_Join_HashBased(Equi_Join):  
235 ▾     def _dot(self, graph, prefix):  
236         return super()._dot(graph, prefix, "⋈_HashBased{},{}")  
237  
238 ▾     def evaluate(self):  
239         l_eval_input = self.l_input.evaluate()  
240         r_eval_input = self.r_input.evaluate()
```

The evaluate in line 240 would be reason enough, to draw the join with “T right” also in red! Because here all intermediate results of the right input are first collected (materialised in main memory)!

⇒ The cost model from above is in contradiction to the implementation!

Remedy? Option 1: We Change the Model

Old model from above:

Cost model: HashJoin in main memory

$$\text{Costs}_{\text{Main memory}}(\text{HashJoin}) \approx c_3 \cdot c_4 \cdot |R|.$$

New model:

Cost model: HashJoin in main memory including costs for right input relation

$$\begin{aligned} \text{Costs}_{\text{Main memory complete}}(\text{HashJoin}) = \\ \text{Costs}_{\text{Main memory}}(\text{HashJoin}) + c_5 \cdot |S|. \end{aligned}$$

$c_5 :=$ Size of a tuple in S .

Option 2: We Change Reality

Observation

Currently, the relations are materialised for each call. However, this is actually unnecessary. When to materialise depends on the operator.

Examples: The input is read completely and the result is materialised before a result is passed on to the next operator!

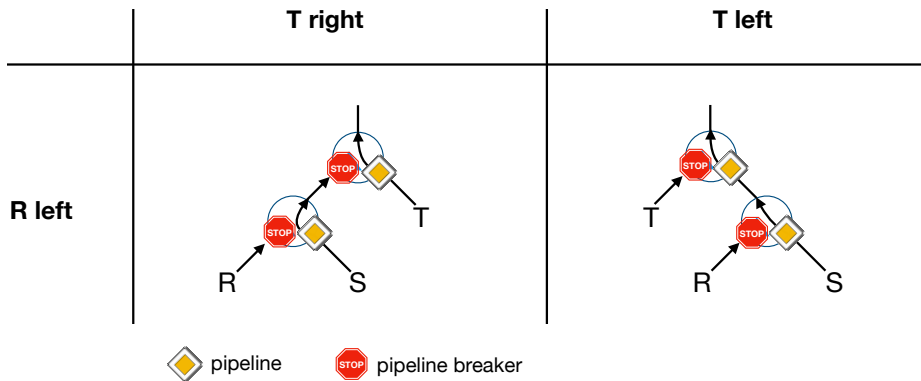
- Selection and projection:

But: actually, each tuple can be filtered or projected independently, i.e. we could pass it on directly!


- Aggregation with `max()`:

But: We have to look at all the tuples to decide what the maximum is. However, we do not need to cache all the input tuples. We could compare each tuple directly with the current maximum.

Pipelines vs Pipeline Breakers

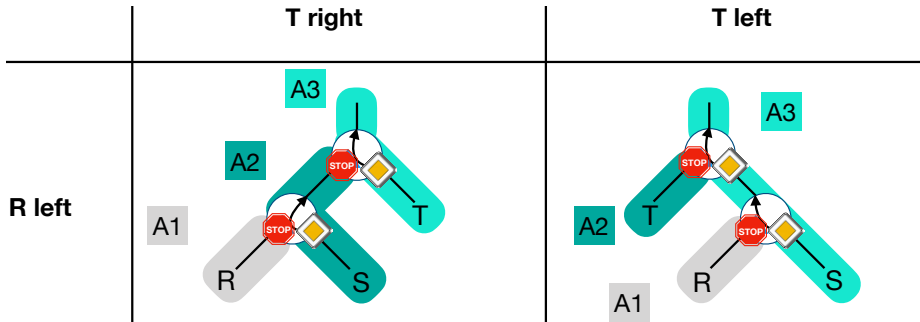


Pipeline Breaker

At certain points in the physical plan data **must** be materialised. We call these places pipeline breakers. 

e.g. in the HashJoin for filling the hash table

Plan Sections



Pipeline Breakers and Plan Sections



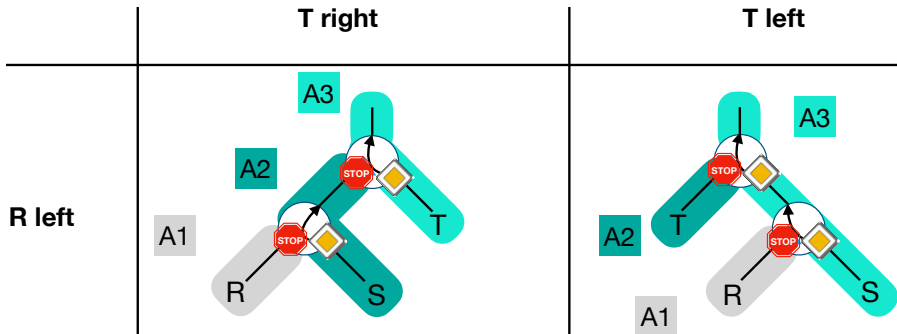
Pipeline breakers naturally partition a physical plan into plan sections.

Pipelining



Data can be streamed (pipelined) within a plan section, without interrupting the data flow.

Plan Sections and Pipelining



Section A1: Relation R is completely materialised in the hash table

Section A2: Materialises only $R \bowtie S$ in the second hash table

Section A3: Materialises nothing at all

⇒ **good plan!**

(regarding pipelining)

if $\text{sizeof}(R \bowtie S) < \text{sizeof}(T)$

Section A1: Just like on the left

Section A2: Relation T is completely materialised in the hash table

Section A3: Just like on the left

⇒ **good plan!**

(regarding pipelining)

if $\text{sizeof}(R \bowtie S) > \text{sizeof}(T)$

Code Generation

4. physical plan

↓ code generation

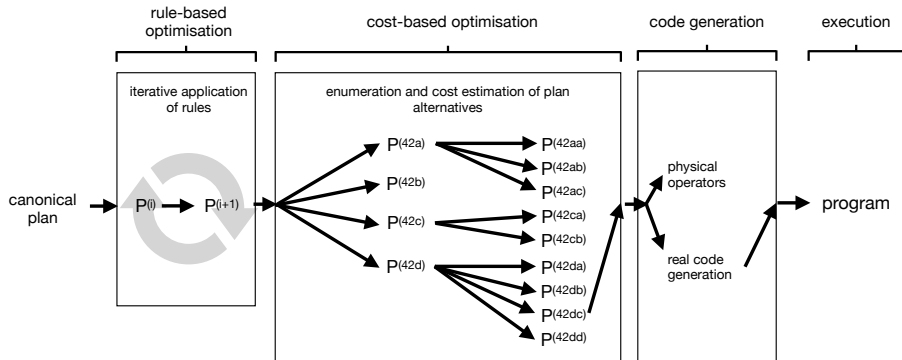
5. executable code

- in the last step we generate executable code from the physical plan
- typically C/C++, LLVM or WebAssembly is generated

Further Reading:

I. Haffner, J. Dittrich. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. EDBT 2023.

Optimisation, Code Generation, and Execution: Overview



Plan Interpretation vs Code Generation

Plan Interpretation

For some systems, code generation is omitted: then the physical operators are converted 1:1 into programming language constructs, e.g. by calls to a library of physical operators. This is what we call *plan interpretation*.

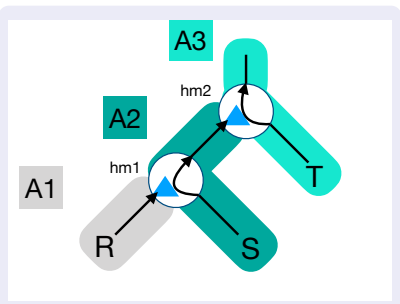
This is exactly what happens in our implementation in the notebook.

Code Generation

Other systems generate program code in this step that breaks the imaginary boundaries of the physical operators. The physical operators are **not necessarily** converted 1:1 into programming language constructs. This is what we call *code generation*.

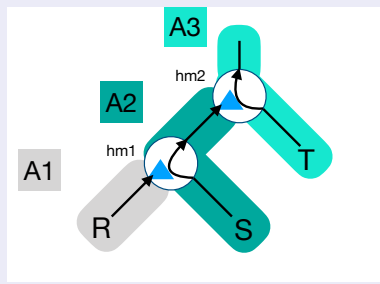
Real Code Generation (Pseudo-Code)

1. `HashMap hm1; HashMap hm2;`
- A1: Add all tuples from input R into the HashMap `hm1`:
2. For all r in R :
3. `hm1.insert(r.x, r);`
- A2: Search all tuples from the right input S in the HashMap `hm1`:
4. For all s in S :
5. `RES = hm1.probe(s.y);`
6. For all z in $(RES \times \{s\})$:
7. `hm2.insert(z. ... , z);`
- A3: Search all tuples from the right input T in the HashMap `hm2`:
8. For all t in T :
9. `RES = hm2.probe(t...);`
10. `yield (RES \times {t});`

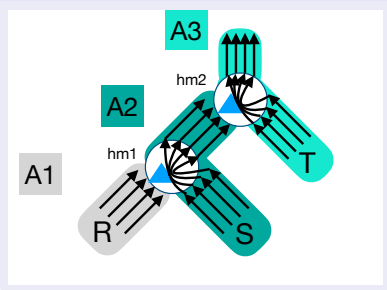


Multi-Threading

Single-Threaded



Multi-Threaded (4 Threads)



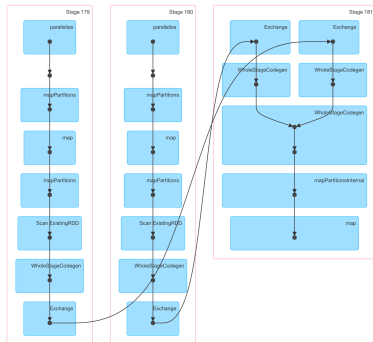
- The code generation can be easily extended in this case to support multi-threading (in general, this can become arbitrarily complex...).
- Here the input relations are **horizontally partitioned**.
- Each partition is then processed in a separate thread.
- The only points of contact between the threads are the two (hopefully) thread-safe hash tables.

Multi-Threading and Plan Sections in Spark

```
== Analyzed Logical Plan ==
last_name: string, prob#24: float
Project [last_name#14, prob#24]
+- Filter ((last_name#14 = Tarantino) && (genre#23 = Mystery))
   +- Filter (id#12 = director_id#22)
      +- Join Cross
         :- LogicalRDD [id#12, first_name#13, last_name#14], false
         +- LogicalRDD [director_id#22, genre#23, prob#24], false

== Optimized Logical Plan ==
Project [last_name#14, prob#24]
+- Join Cross, (id#12 = director_id#22)
   :- Project [id#12, last_name#14]
   :- Filter (last_name#14 = Tarantino)
      +- LogicalRDD [id#12, first_name#13, last_name#14], false
   +- Project [director_id#22, prob#24]
      +- Filter (genre#23 = Mystery)
         +- LogicalRDD [director_id#22, genre#23, prob#24], false

== Physical Plan ==
*(5) Project [last_name#14, prob#24]
+- *(5) SortMergeJoin [id#12], [director_id#22], Cross
   :- *(2) Sort [id#12 ASC NULLS FIRST], false, 0
   :- +- Exchange hashpartitioning(id#12, 200)
   :-    +- *(1) Project [id#12, last_name#14]
   :-       +- *(1) Filter (last_name#14 = Tarantino)
   :-          +- Scan ExistingRDD[id#12,first_name#13,last_name#14]
   +- *(4) Sort [director_id#22 ASC NULLS FIRST], false, 0
   +- +- Exchange hashpartitioning(director_id#22, 200)
   +-    +- *(3) Project [director_id#22, prob#24]
   +-       +- *(3) Filter (genre#23 = Mystery)
   +-          +- Scan ExistingRDD[director_id#22,genre#23,prob#24]
```



- GitHub: [Spark.ipynb](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/MLlibExample.scala)
- Spark is a popular framework for processing large datasets.
- In Spark, plan sections are called *Stages*.
- Within a stage, data is divided by horizontal partitioning.
- One thread is used per partition.

What Do I Do if the Queries Are Too Slow?

Question 1

How do we actually get from SQL to an executable program in principle?

Through automatic heuristic- and cost-based optimisation.

Summary: Cost-Based Optimisation

3. transformed logical plan
↓ cost-based optimisation
4. physical plan

What are the main tasks of cost-based optimisation?

1. Enumeration and evaluation of plan alternatives (requires statistics and cost estimates)
2. Choice of join order
3. Choice of physical operators
4. Planning the pipelining (blocking vs. non-blocking operators)
5. Planning multi-threading and possibly other resources

Summary: Code Generation

- 4. physical plan
↓ code generation
- 5. executable code

What are the main tasks of code generation?

Make a basic decision: how to execute the plan?









Either:

- A Interpretation: A tree of physical operators is suitably traversed and executed.
- B (Real) Code Generation: A program that calculates the result of the query is created, compiled, and executed. Operator boundaries are removed in that process.

Attention

We have only scratched the surface for all these topics so far.... There are many interesting techniques in the field:

Further Material

	Query Planning and Optimization Prof. Dr. Jens Dittrich 0:51
	14.500 Query Optimizer Overview Prof. Dr. Jens Dittrich 6:29
	14.502 Challenges in Query Optimization: Rule-Based Optimization Prof. Dr. Jens Dittrich 13:57
	14.503 Challenges in Query Optimization: Join Order, Costs, and Index Access Prof. Dr. Jens Dittrich 14:53
	14.514 Cost-Based Optimization, Plan Enumeration, Search Space, Catalan Numbers, Identical Plans Prof. Dr. Jens Dittrich 20:42
	14.516a Dynamic Programming: Core Idea, Requirements, Join Graph Prof. Dr. Jens Dittrich 12:08
	14.516b Dynamic Programming Example without Interesting Orders, Pseudo-Code Prof. Dr. Jens Dittrich 13:41
	14.516c Dynamic Programming Optimizations: Interesting Orders, Graph Structure Prof. Dr. Jens Dittrich 10:18

YouTube Videos of Prof. Dittrich on Query Optimisation (English)

And:

- Jens Dittrich. Patterns in Data Management.
<https://bigdata.uni-saarland.de/datenbankenlernen/book.pdf>
- Core lecture Database Systems