Distributed Data Processing Environments

José Orlando Pereira

Departamento de Informática
Universidade do Minho

Query processing

a
2
3

“select a from X natural join Y where c = 3;”

X

a	b
1	aaa
2	bbb
3	ccc

Y

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Relational algebra

- Relation: Set of tuples
- Basic operations:
  - Set operations
  - SELECT … WHERE <u>condition</u> → Selection (σ)
  - SELECT <u>columns</u> FROM … → Projection (π)
  - SELECT … FROM <u>x JOIN Y</u> → Inner join (⋈)
- Other operations:
  - Grouping and aggregation (γ)
  - Outer joins (⟗, ⟕, ⟖)

Most operators in SQL systems work on multi-sets / “bags”!

Compilation

SQL { "select a from X natural join Y where c = 3;"

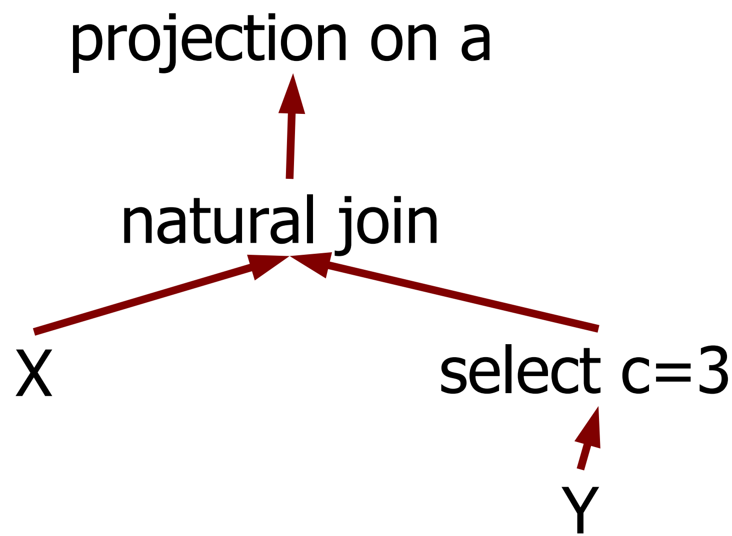
Relational algebra

$$\pi_a(\sigma_{c=3}(X \bowtie Y))$$

Compilation

SQL “select a from X natural join Y where c = 3;”

Relational algebra



Roadmap

- How are physical operators implemented and composed?

- What physical operators exist for each logical operation

- Later: How are physical operators selected?

Execution with materialization

- Bottom up:
  - Start from the leafs (stored tables)

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Execution with materialization

- Bottom up:
  - Start from the leafs (stored tables)
- Compute intermediate results

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Execution with materialization

- Bottom up:
  - Start from the leafs (stored tables)
- Compute intermediate results
- Until the final result can be delivered to the user

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Consequences

- Efficient use of current CPU architectures when combined with columnar layouts
  - Vectorization
- Large intermediate results that need to be stored
  - Might not fit completely in memory
- Potentially wasted work
  - e.g. SELECT … LIMIT 10

Execution with iteration

- Top down:
  - What is needed for a row in the result?
  - Recursively visit each intermediate result
  - Eventually start reading the data

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Execution with iteration

- Top down:
  - What is needed for a row in the result?
  - Recursively visit each intermediate result
  - Eventually start reading the data
- The intermediate result is computed for each row

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Execution with iteration

- Top down:
  - What is needed for a row in the result?
  - Recursively visit each intermediate result
  - Eventually start reading the data
- The intermediate result is computed for each row

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3 **Y**
ccc	3
ddd	4

Execution with iteration

- Top down:
  - What is needed for a row in the result?
  - Recursively visit each intermediate result
  - Eventually start reading the data
- The intermediate result is computed for each row

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Execution with iteration

- Top down:
  - What is needed for a row in the result?
  - Recursively visit each intermediate result
  - Eventually start reading the data
- The intermediate result is computed for each row

b
bbb
ccc

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Y

Consequences of iteration

- Minimizes memory needed for large intermediate results
- Minimizes work with LIMIT clause
- Not applicable to operators that must observe all rows before knowing what is the first to output
  - ORDER BY
  - GROUP BY on an unsorted input
  - …

Consequences of iteration

- Close to worst case scenario for data movement and parallelism!
  - Poor locality → Impacts caching / NUMA
  - Short code segments interleaved with dereferencing though virtual pointers → Processor pipeline stalls
  - Computation on one value at a time → No SIMD
- Severely impacts <u>analytical</u> workloads!

Hybrid solution: Chunked data

- Iterate over “chunks”:
  - Records → Records of arrays
- Exploit columnar layout: SIMD
- Can be combined with operator fusion

DuckDB

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Roadmap

- How are physical operators implemented and composed?

- <u>What physical operators exist for each logical operation</u>

- Later: How are physical operators selected?

One-pass, record-at-a-time

- Operators:
  - Sequential scan
  - Selection
  - Projection
- Memory requirements:
  - No more than one record required
  - Always possible

One-pass, full relation, unary

- Duplicate elimination:
  - Cache unique records
  - “select distinct * from X;”
- Grouping and aggregation:
  - Cache groups
  - “select count(*) from X group by b;”
- Sorting:
  - Cache all records and sort in memory
  - “select * from X order by b;”

Nested-loop join (NLJ)

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

b	c
bbb	3
ccc	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

b	c
bbb	3
ccc	3

...

One-pass, full relation, binary

- Avoid reading the inner relation multiple times
  - Read and cache the smallest relation
  - Organize for fast look-up (e.g. hash)
  - Read and operate on each record from the largest relation
- Also applicable to union, difference, intersection, product

One-pass, full relation, binary

- Load smaller table into memory and add search structure:

a	b	c
2	bbb	3
3	ccc	3

a	b
1	aaa
2	bbb
3	ccc

	c
bbb	3
ccc	3

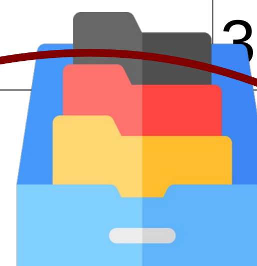
Icons by Flaticon.com.

One-pass, full relation, binary

- Test each record from the largest relation:

a	b	c
2	bbb	3
3	ccc	3

a	b
1	aaa
2	bbb
3	ccc



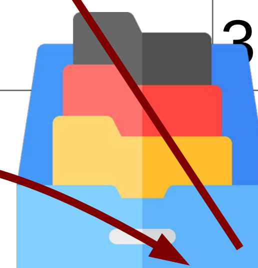
Not found

	c
bbb	3
ccc	3

One-pass, full relation, binary

- Test each record from the largest relation:

a	b	c
2	bbb	3
3	ccc	3



a	b
1	aaa
2	bbb
3	ccc

	c
bbb	3
ccc	3

Icons by Flaticon.com.

Nested-loop join (NLJ)

- Memory requirements:
  - One record from each relation
- Operations:
  - If outer loop has N records
  - Reads inner relation N times

Large relations and sorting

- Algorithms using sorted data are more efficient (e.g. than nested loops)
- How to sort data that does not fit in memory?

Merge-sort

- Split data in chunks that fit in memory:

a	b
8	...
4	
7	
2	
3	
5	
6	
1	

a	b
8	...
4	
7	
2	

a	b
3	...
5	
6	
1	

Merge-sort

- Load and sort each of them

a	b
8	...
4	
7	
2	

a	b
2	...
4	
7	
8	

a	b
3	...
5	
6	
1	

Merge-sort

- Load and sort each of them

a	b
8	...
4	
7	
2	

a	b
2	...
4	
7	
8	

a	b
3	...
5	
6	
1	

a	b
1	...
3	
5	
6	

Merge-sort

- Select the next element with lowest key:

a	b
2	...
4	
7	
8	

a	b
1	...
3	
5	
6	

1

Merge-sort

- Select the next element with lowest key:

a	b
2	...
4	
7	
8	

1

2

a	b
1	...
3	
5	
6	

Merge-sort

- Select the next element with lowest key:

a	b
2	...
4	
7	
8	

a	b
1	...
3	
5	
6	

1

2

3

...

Two-pass, full relation, unary

- First pass is sorting
- Duplicate elimination:
  - Cache last record
  - “select distinct * from X;”
- Grouping and aggregation:
  - Cache last group
  - “select count(*) from X group by b;”

Example

- Assumptions:
  - ~50%, y=1
  - ~50%, y=2
  - a few, y=3
- Query:
  - select count(*) from X where y = 1;
- Not efficient for frequent queries

z	y
d	**1**
c	2
g	**1**
k	2
h	3
a	**1**
b	**1**
f	2
d	2
k	**1**
j	2
l	**1**
...	...

Example

- Keep results cached when original table is updated:

y	count
1	**773647263**
2	765732332
3	1

- Use with:
  - select * from counts where y = 1;

Materialized views

View definition

materialize V

count(*) group by y

scan X

create materialized view V as select y, count(*) from X

View usage

select * from V where y = 1

selection y = 1

V

Plan executed

Summary

- A SQL system does:
  - Transform the statement to relational algebra
  - Selects physical operators
  - Executes the resulting program
- Different execution strategies:
  - Iteration is not good for analytical workloads
- Different physical operators:
  - Each with performance tradeoffs
- Materialization is key for analytical performance