

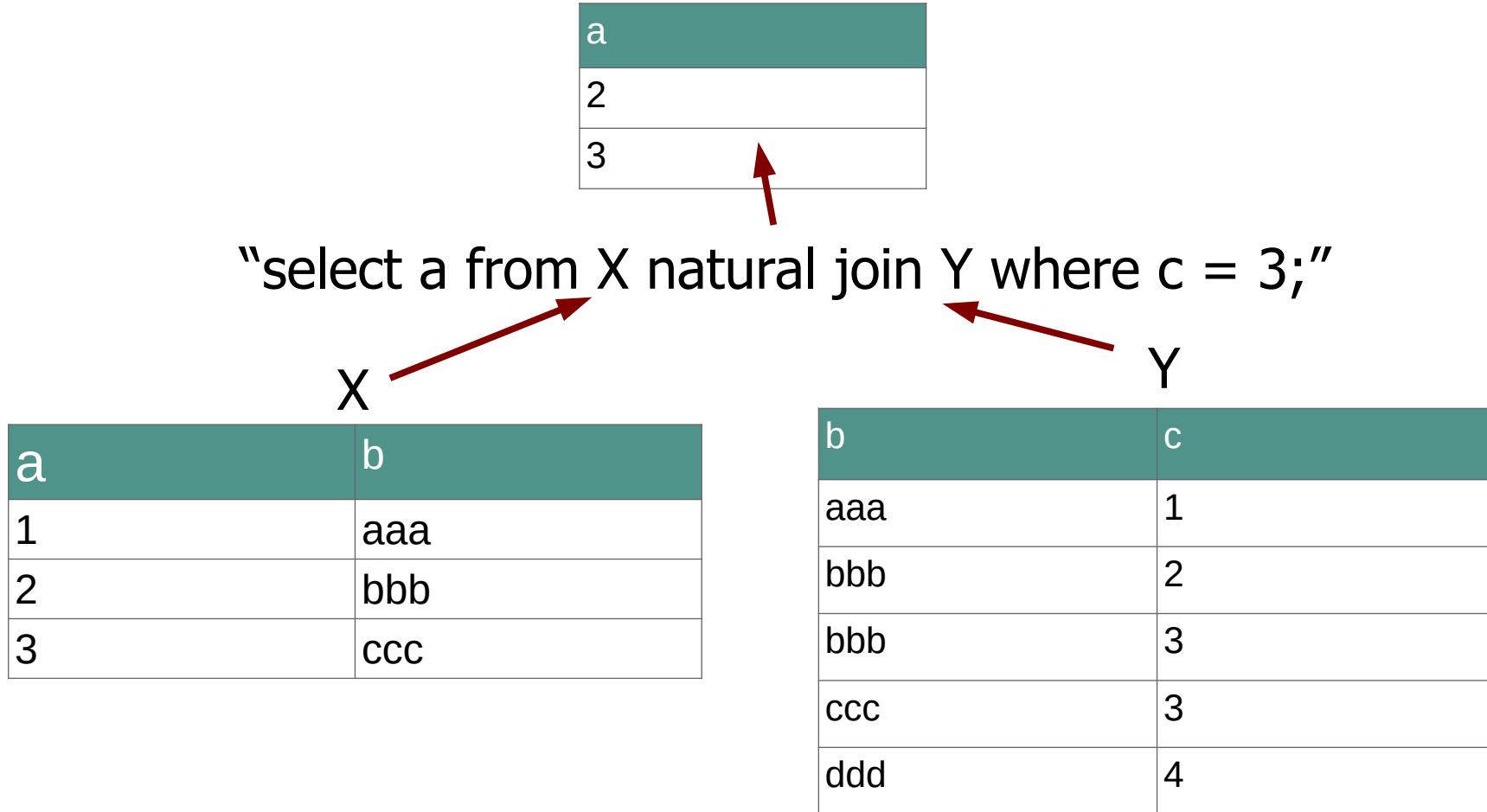
Distributed Data Processing Environments

José Orlando Pereira

Departamento de Informática
Universidade do Minho



Query processing



Relational algebra

- Relation: Set of tuples
- Basic operations:
 - Set operations
 - $\text{SELECT ... WHERE } \underline{\text{condition}}$ → Selection (σ)
 - $\text{SELECT } \underline{\text{columns}} \text{ FROM ...}$ → Projection (π)
 - $\text{SELECT ... FROM } \underline{x} \text{ JOIN } Y$ → Inner join (\bowtie)
- Other operations:
 - Grouping and aggregation (γ)
 - Outer joins (\bowtie_l , \bowtie_r , \bowtie_{lr})

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

Compilation

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



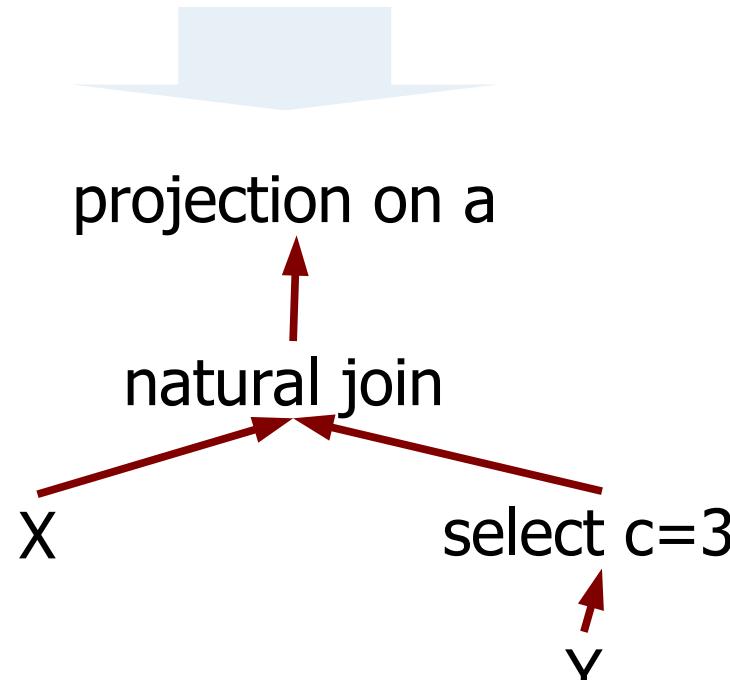
Relational
algebra

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

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)

projection on b

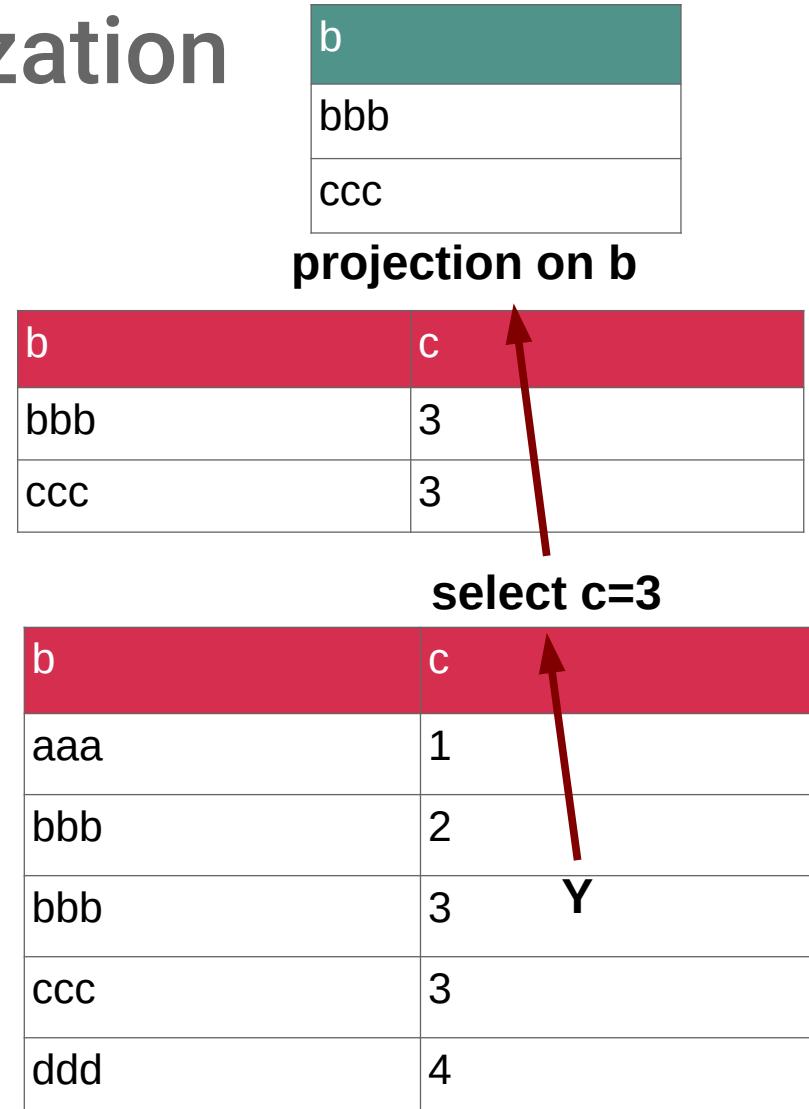
b	c
bbb	3
ccc	3

select c=3

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

Execution with materialization

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

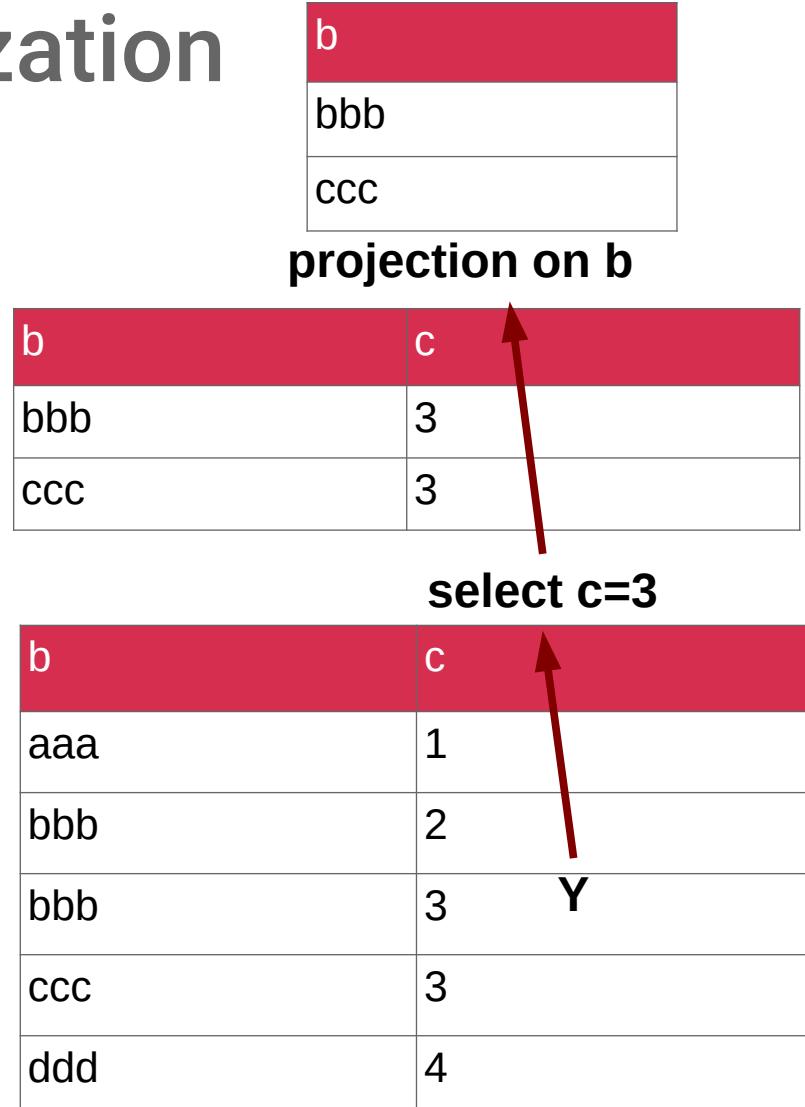


The diagram illustrates the execution of a query using materialization. It shows three stages of data processing:

- Base Table:** A table labeled "b" containing rows "bbb" and "ccc".
- Projection on b:** A temporary table where the row "bbb" is projected into two rows: "bbb" with column "c" value "3" and "ccc" with column "c" value "3".
- select c=3:** The final result table after applying the selection condition. It contains rows "aaa", "bbb", "bbb", "ccc", and "ddd", with the third "bbb" row marked with a yellow "Y" indicating it is the selected row.

Execution with materialization

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



The diagram illustrates the execution of a query using materialization. It shows three stages of data processing:

- Base Table:** A table labeled "b" with rows "bbb" and "ccc".
- Projection on b:** A table with two columns, "b" and "c". The "b" column contains "bbb" and "ccc". The "c" column contains "3" for both rows. This step is labeled "projection on b".
- Final Result:** A table with two columns, "b" and "c". The "b" column contains "aaa", "bbb", "bbb", "ccc", and "ddd". The "c" column contains "1", "2", "3", "3", and "4" respectively. The row where $c=3$ is highlighted with a yellow background and has a red arrow pointing to it, labeled "select c=3".

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

The diagram illustrates the execution of a query using iteration. It shows three stages of processing:

- Input Table:** A table labeled "b" with two rows: "bbb" and "ccc".
- Projection:** A table labeled "projection on b" where the second column "c" is derived from "b". The first row "bbb" has "3" in the "c" column, and the second row "ccc" also has "3" in the "c" column.
- Intermediate Table:** A table after applying the selection "select c=3". The first row "aaa" is highlighted in red. The second row "bbb" is also highlighted in red and has a red arrow pointing to it from the text "select c=3". The third row "bbb" is also highlighted in red. The fourth row "ccc" and the fifth row "ddd" are not highlighted.

projection on b

b	c
bbb	3
ccc	3

select c=3

b	c
aaa	1
bbb	2
bbb	3
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

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

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

The diagram illustrates the execution of a query with iteration through three stages:

- Input:** A table labeled "b" with two rows: "bbb" and "ccc".
- Projection:** A table labeled "projection on b" where the first column is "b" and the second column is "c". The row "bbb" has value "3" in the "c" column, and the row "ccc" also has value "3" in the "c" column.
- Selection:** A table where the row "bbb" is selected based on the condition "c=3". The selected row is highlighted in red and marked with a yellow checkmark "Y". Other rows ("aaa", "bbb", "ccc", "ddd") are shown in their original state.

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

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 through virtual pointers → Processor pipeline stalls
 - Computation on one value at a time → No SIMD
- Severely impacts analytical workloads!

Hybrid solution: Chunked data



DuckDB

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

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?
- What physical operators exist for each logical operation
- Later: How are physical operators selected?

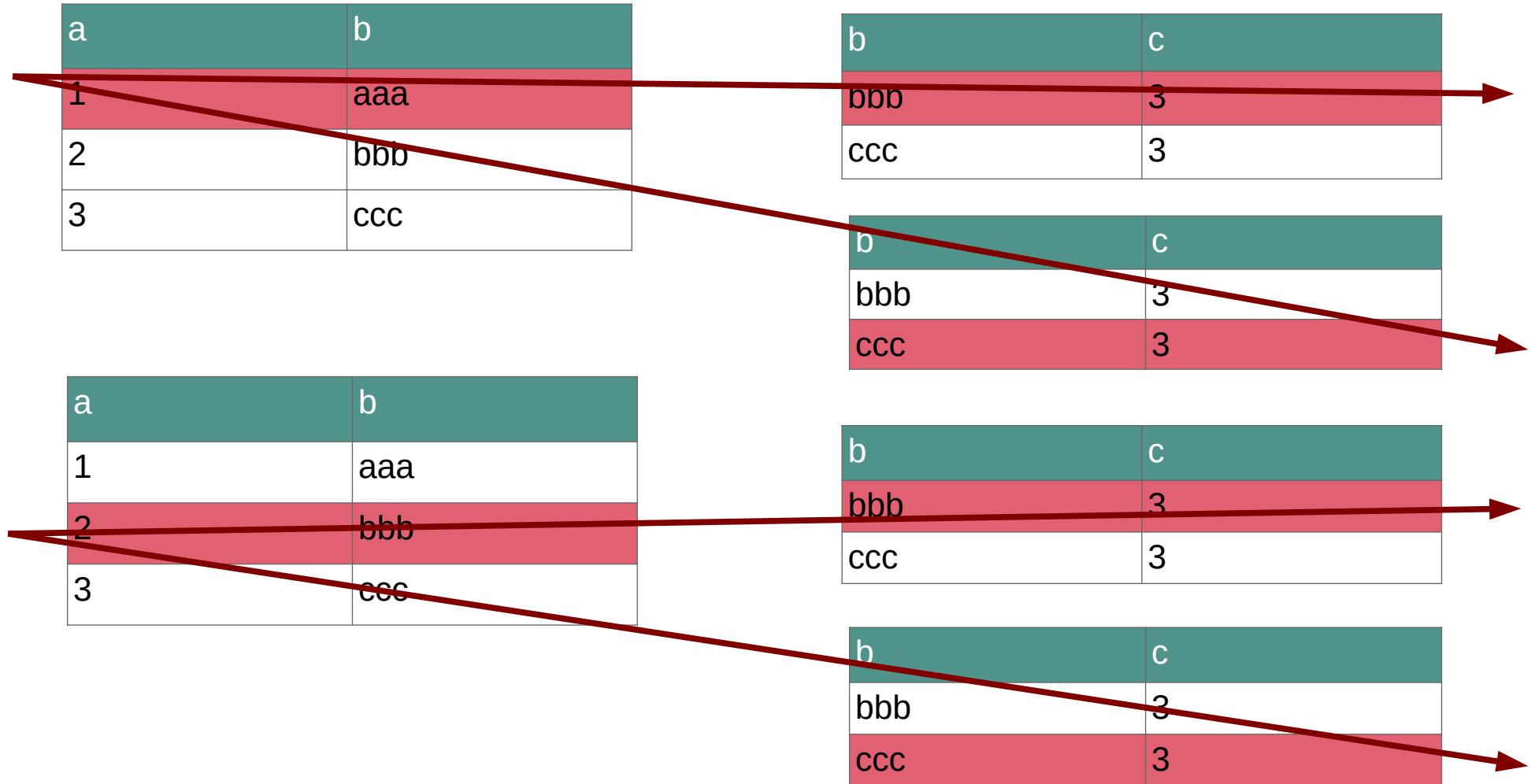
One-pass, record-at-a-time

- 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)



One-pass, full relation, binary

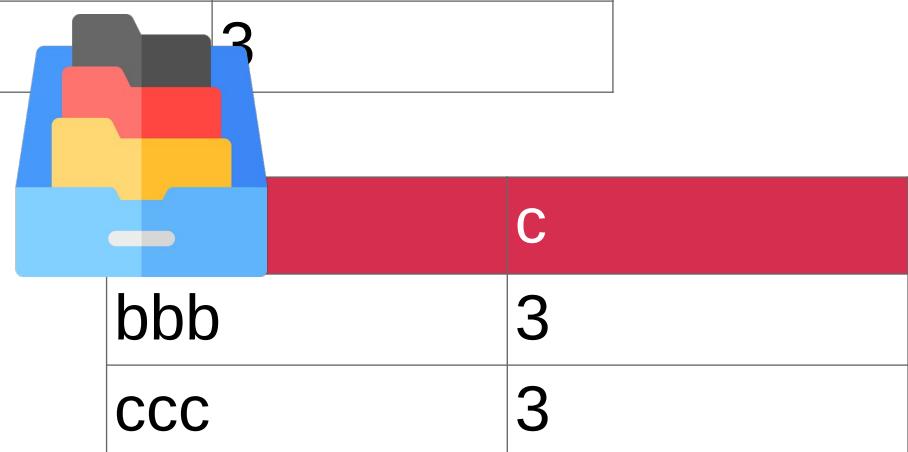
- Avoid reading the inner relation multiple times
 - Read and cache the smallest relation
 - Organize for fast look-up (e.g. hash)
 - Read and operate on each record from the largest relation
- Also applicable to union, difference, intersection, product

One-pass, full relation, binary

- Load smaller table into memory and add search structure:

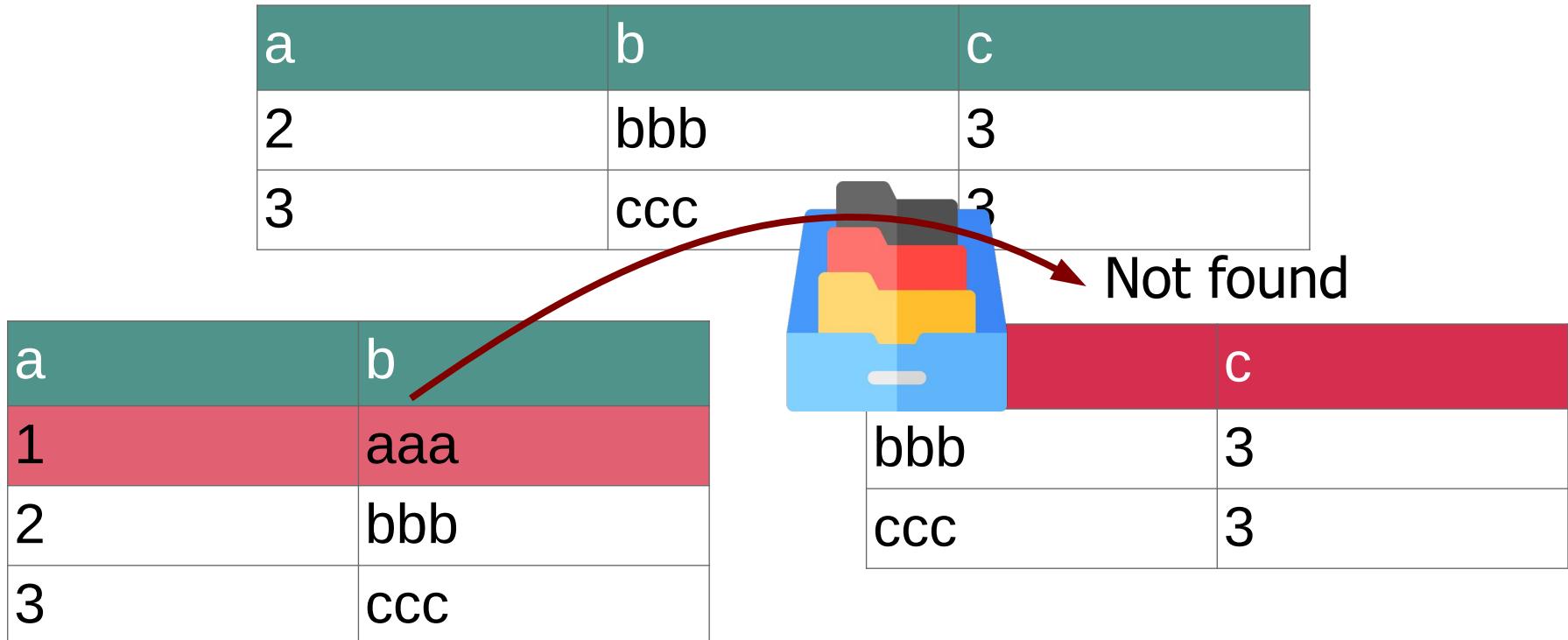
a	b	c
2	bbb	3
3	ccc	3

a	b
1	aaa
2	bbb
3	ccc



One-pass, full relation, binary

- Test each record from the largest relation:



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



a	b	c
	bbb	3
	ccc	3

Nested-loop join (NLJ)

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

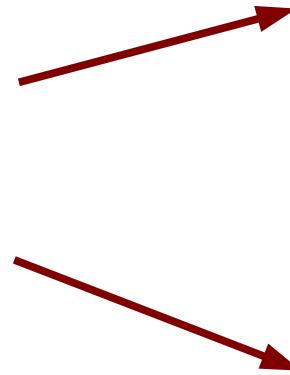
Large relations and sorting

- 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	



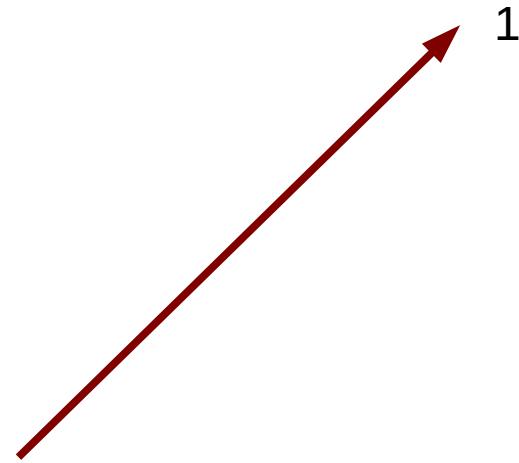
a	b
1	...
3	
5	
6	

Merge-sort

- Select the next element with lowest key:

a	b
2	...
4	
7	
8	

1



a	b
1	...
3	
5	
6	

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	

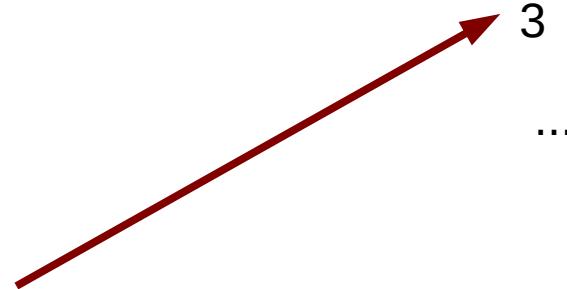
a	b
1	...
3	
5	
6	

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
...	...

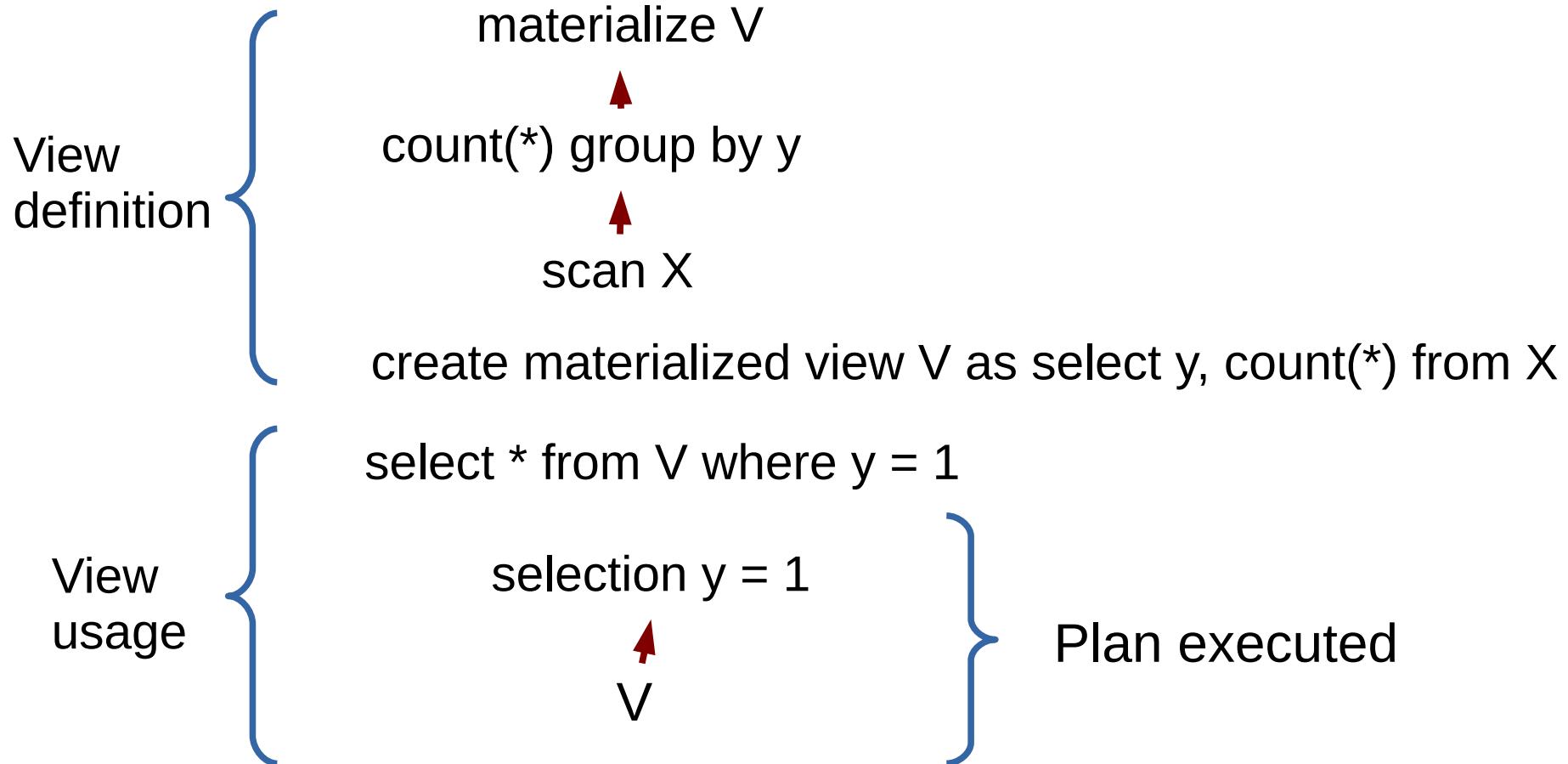
Example

- 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



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