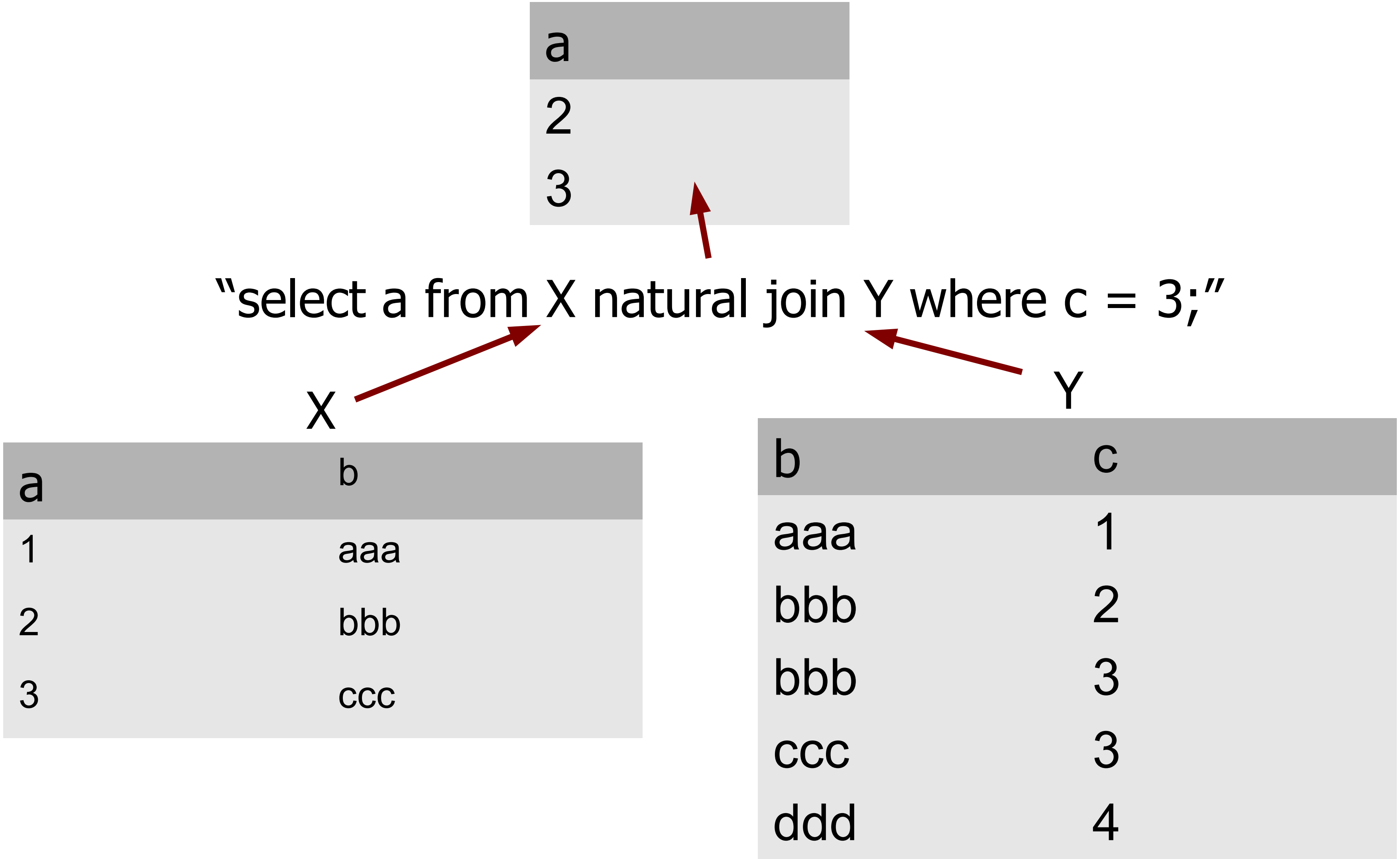
Query processing



a
2
3

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

X

a	b
1	aaa
2	bbb
3	ccc

Y

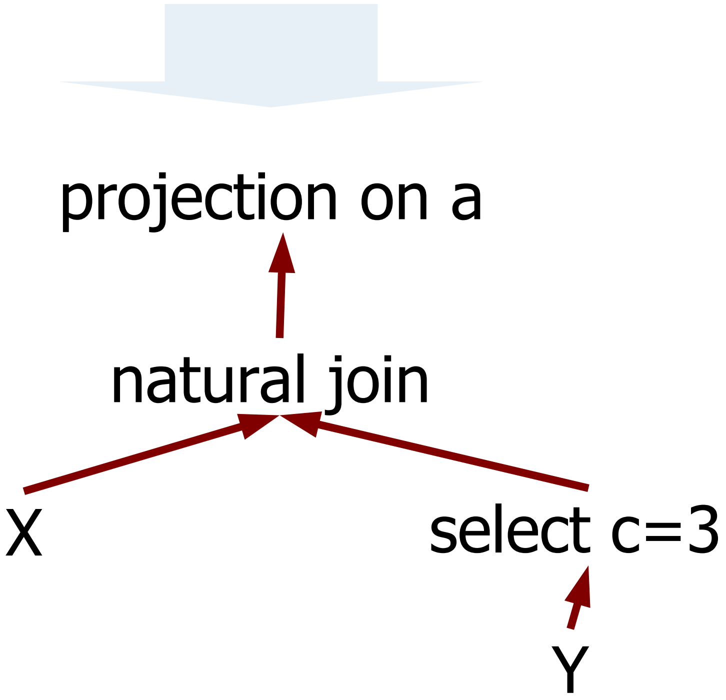
b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Compilation

SQL

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

Relational algebra



projection on a

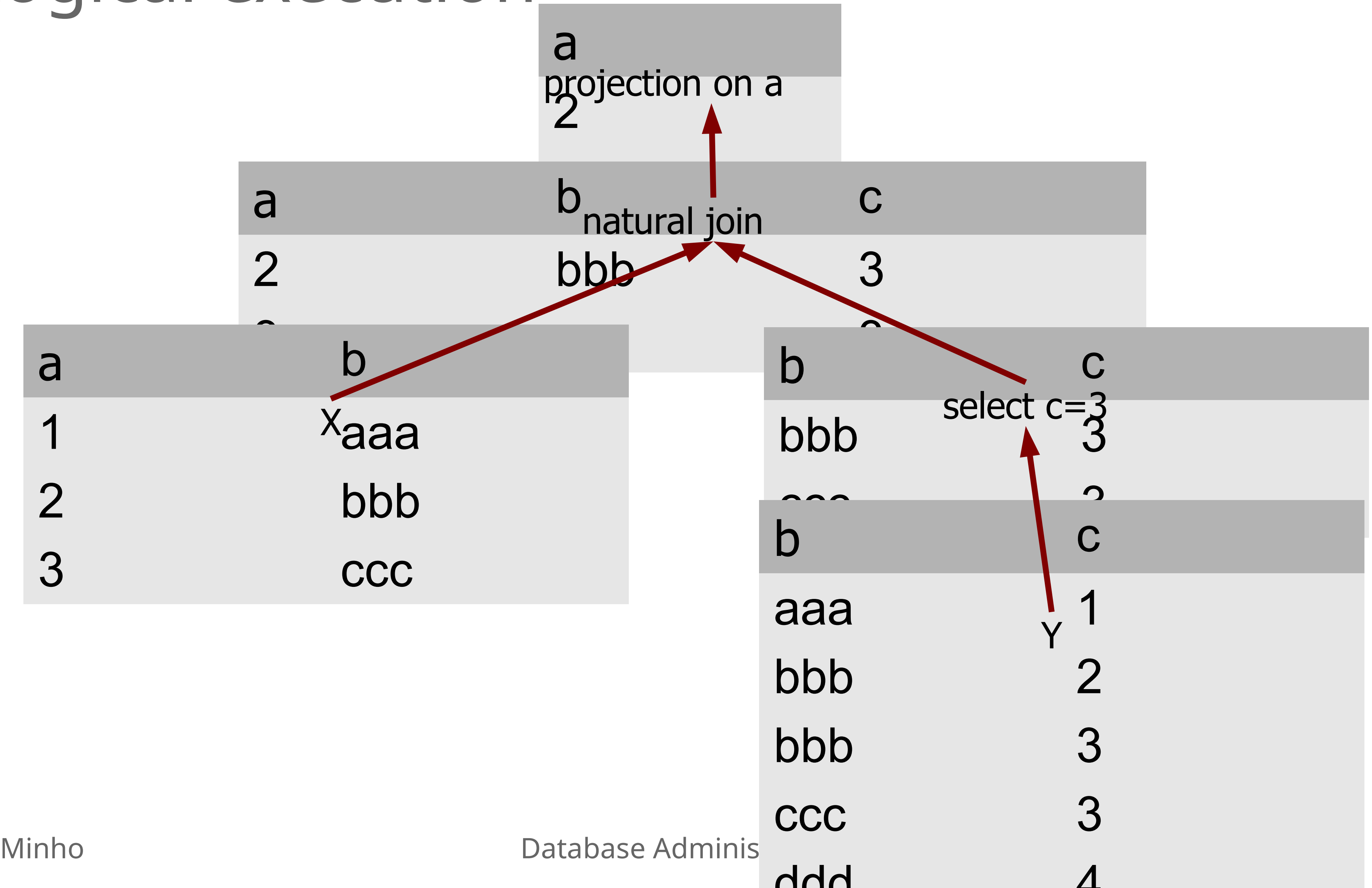
natural join

X

select c=3

Y

Logical execution



a
2

a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

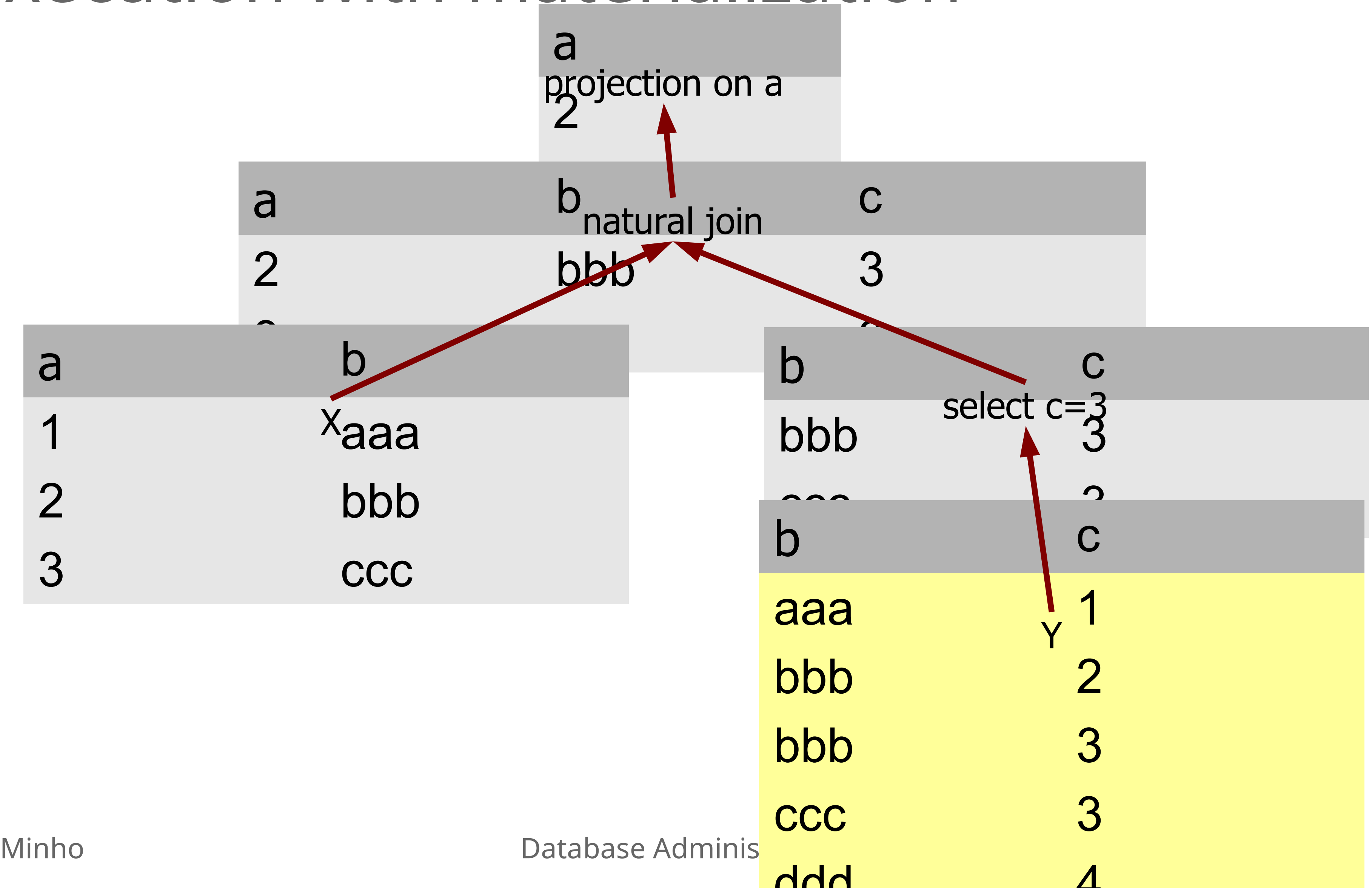
b	c
bbb	3

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

Materialization

- Each operator is a function:
  - Returns a relation
  - Parameters are other relations (possibly, returned from operators)
- Computation order:
  - From leaves to root

Execution with materialization



a
2

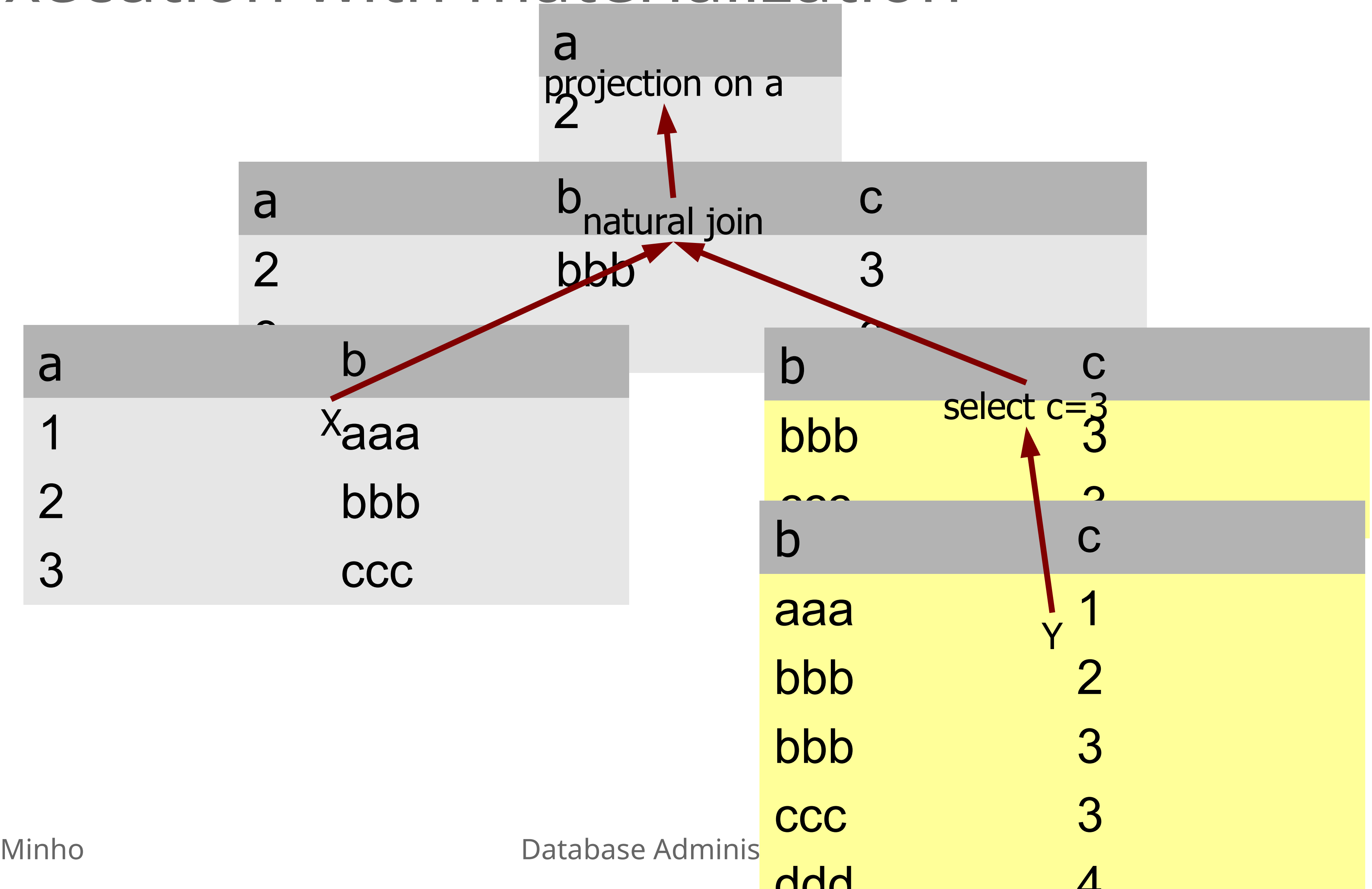
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3

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

Execution with materialization



projection on a

a
2

natural join

a	b	c
2	bbb	3

X

a	b
1	aaa
2	bbb
3	ccc

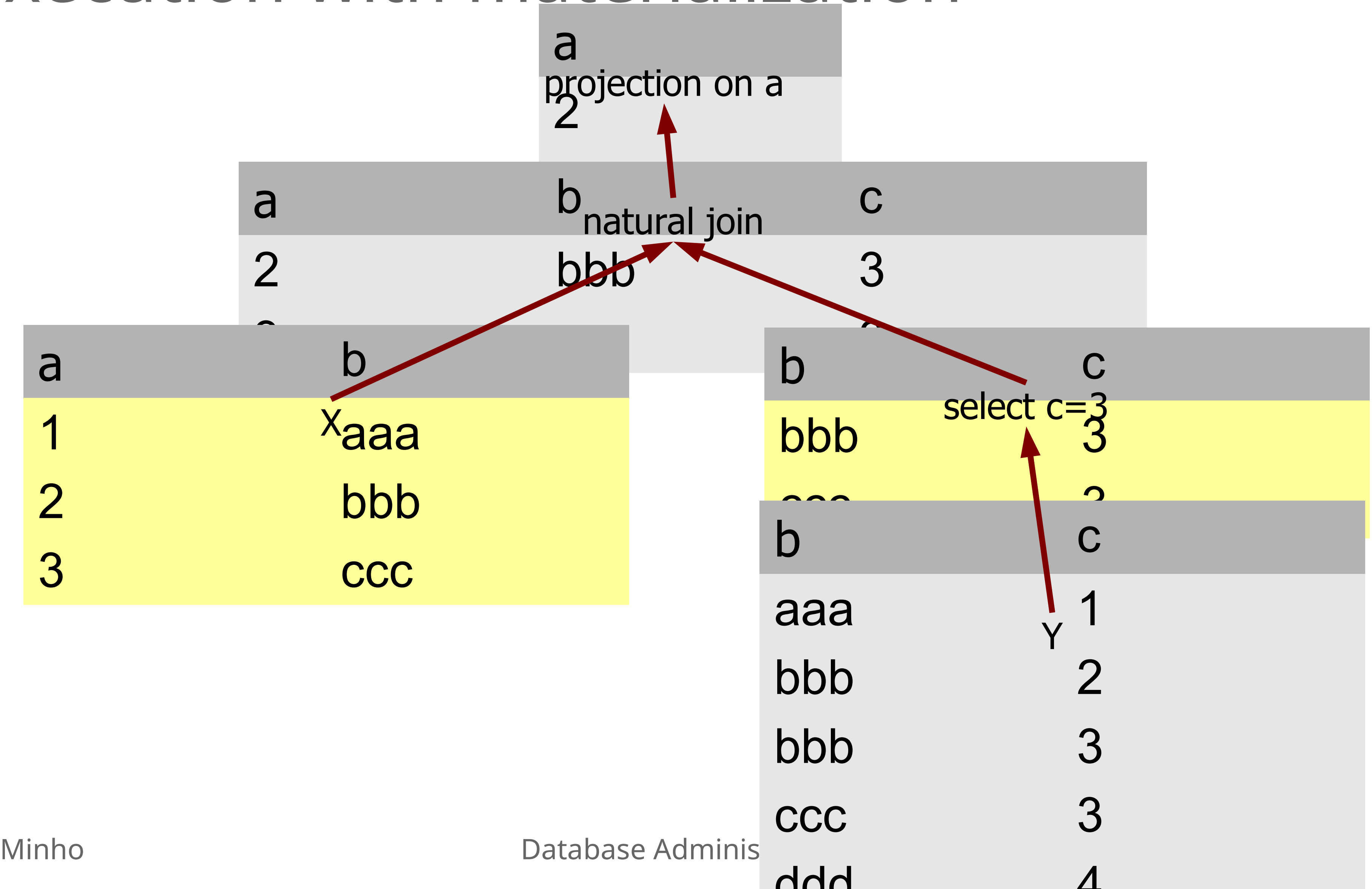
select c=3

b	c
bbb	3

Y

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

Execution with materialization



a
2

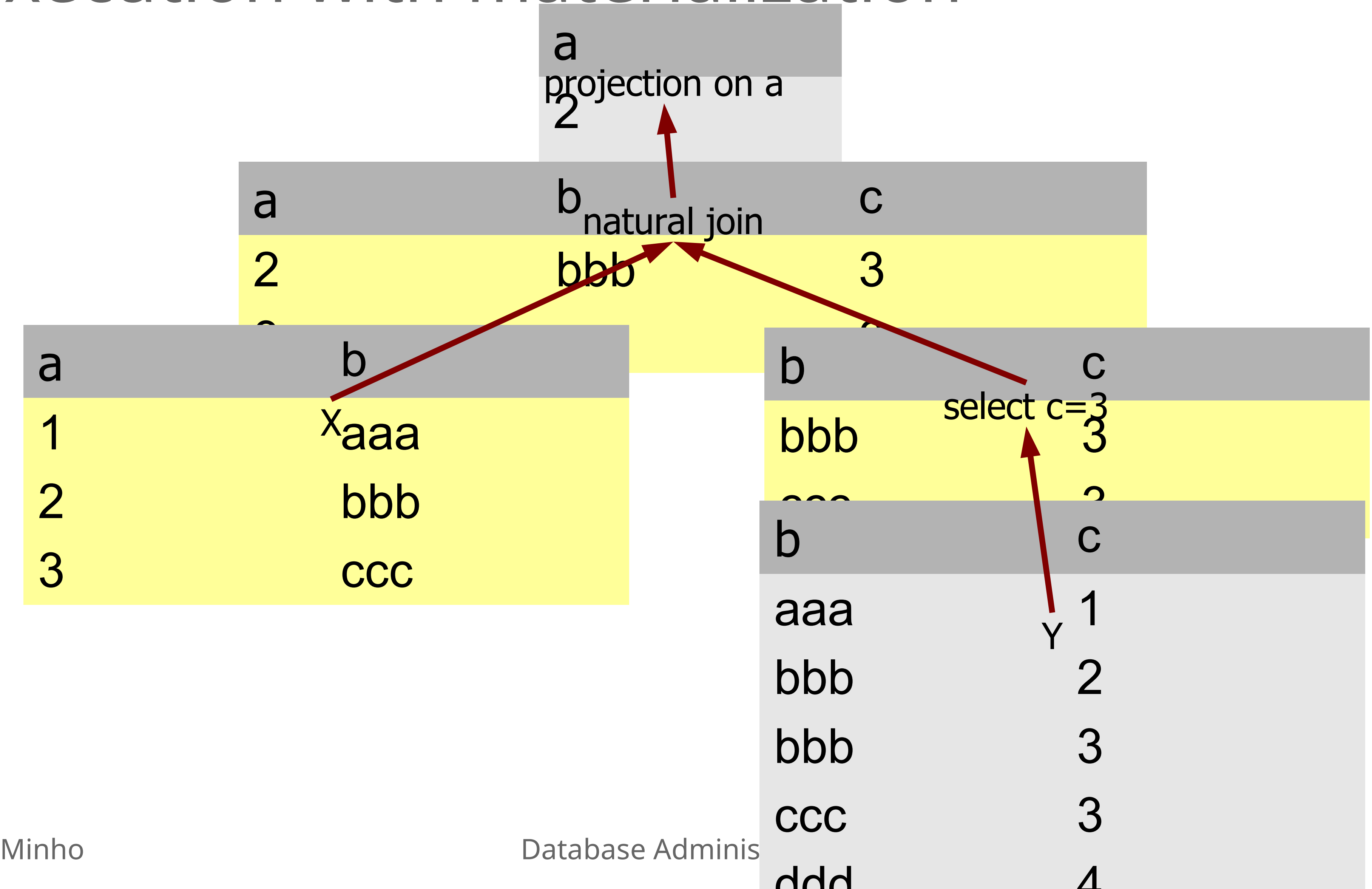
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

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

Execution with materialization



a
2

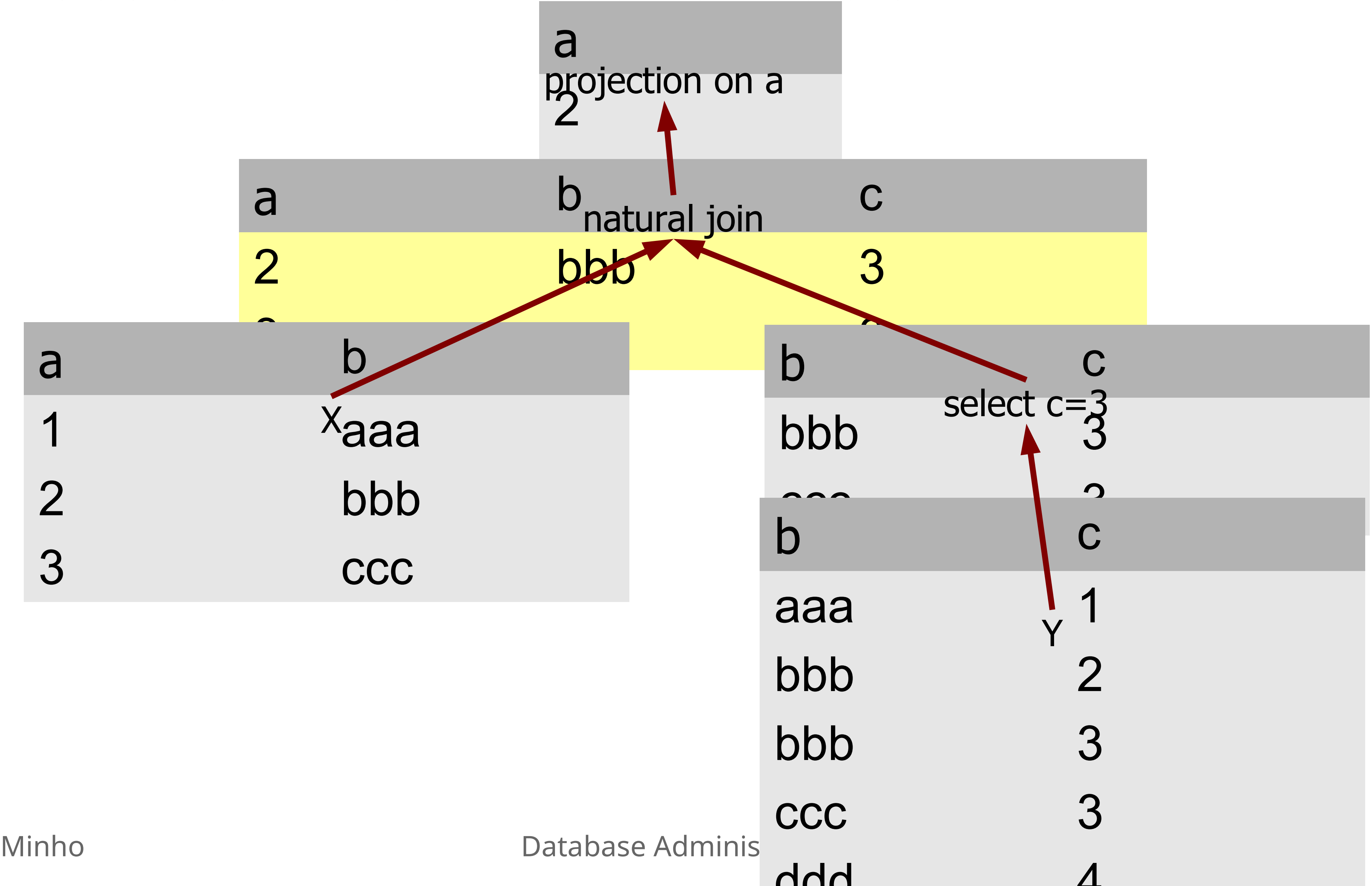
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3

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

Execution with materialization



a
2

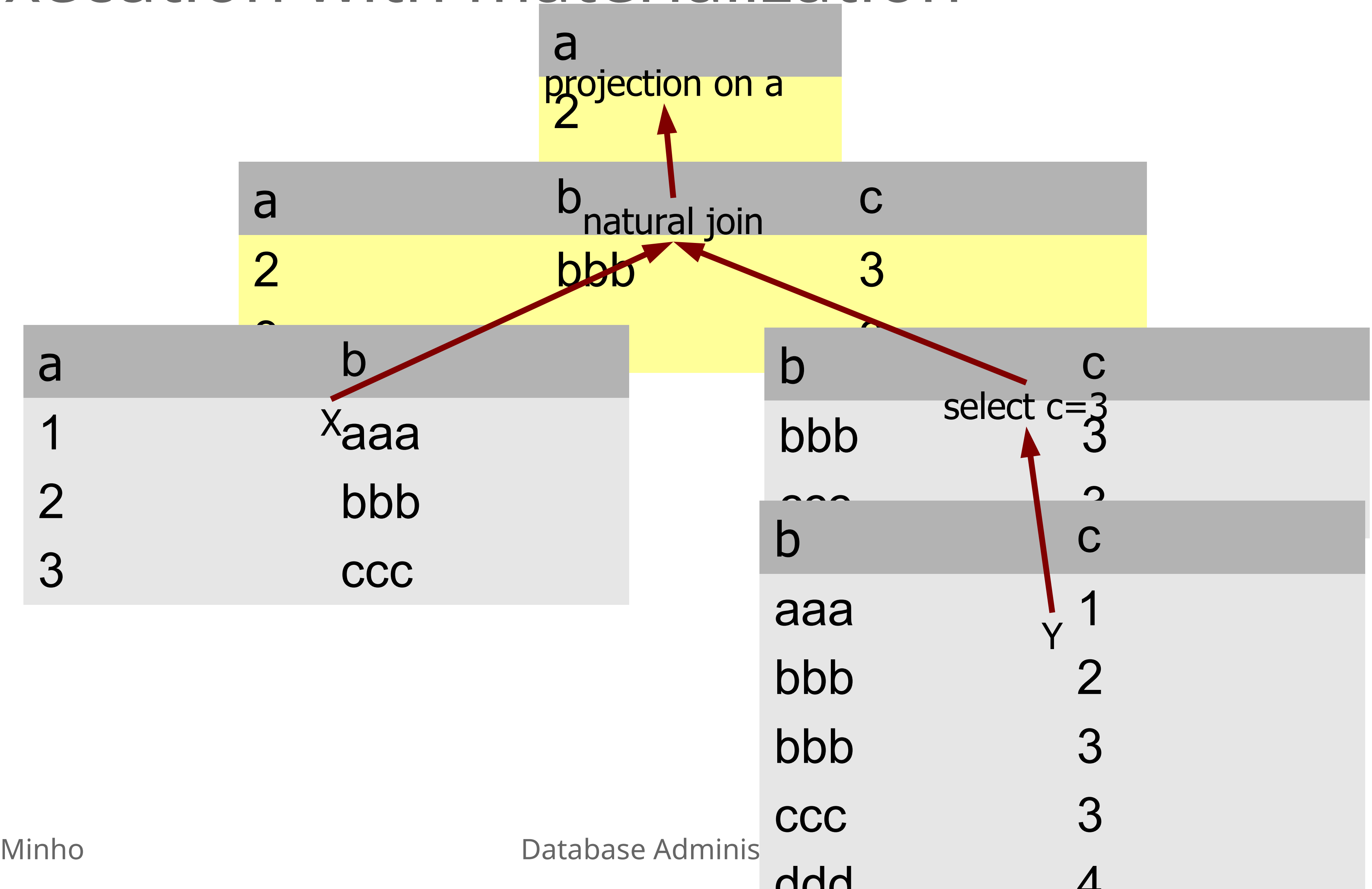
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

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

Execution with materialization



projection on a

a
2

natural join

a	b	c
2	bbb	3

X

a	b
1	aaa
2	bbb
3	ccc

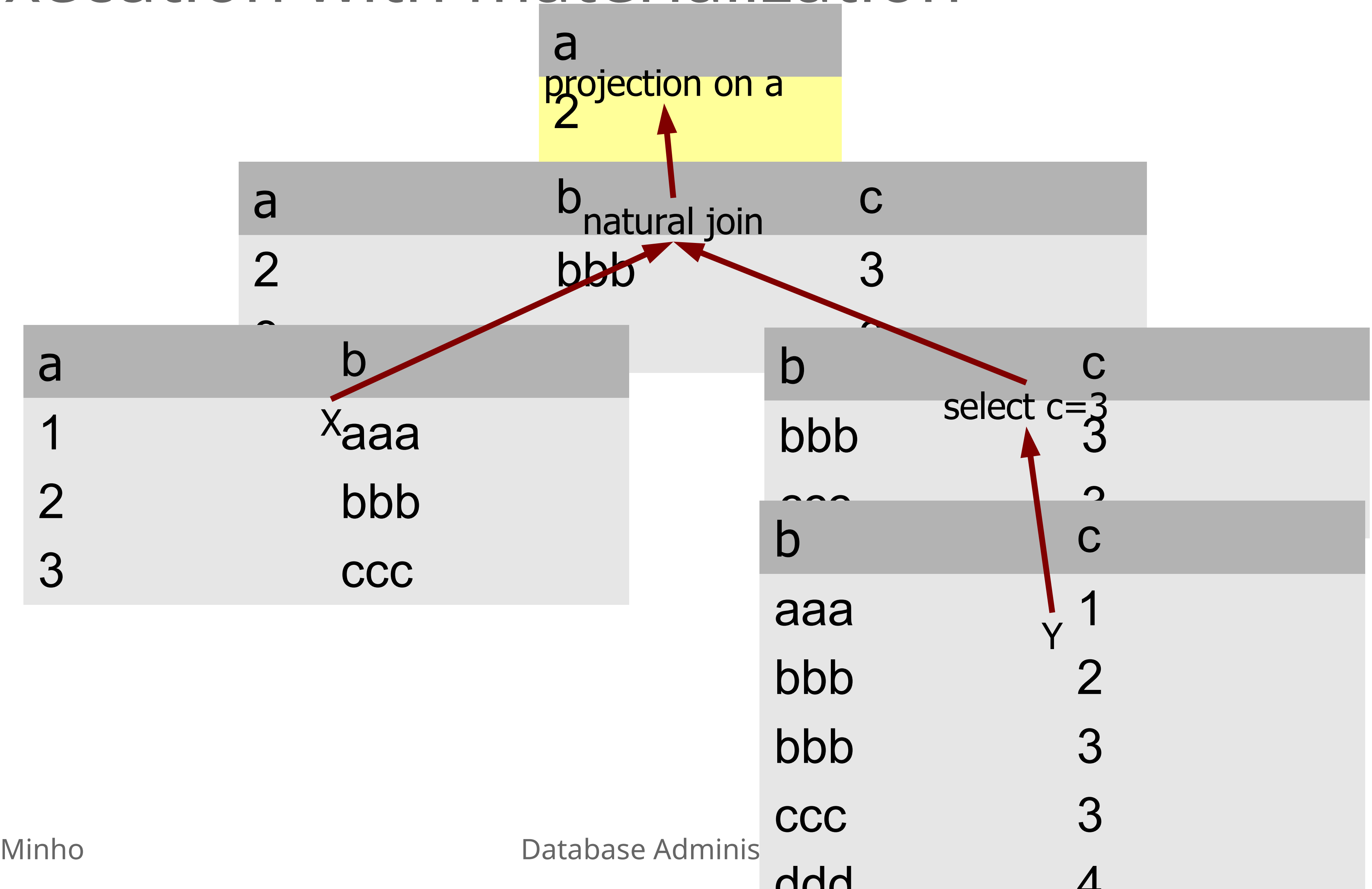
select c=3

b	c
bbb	3

Y

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

Execution with materialization



projection on a

a
2

natural join

a	b	c
2	bbb	3

X

a	b
1	aaa
2	bbb
3	ccc

select c=3

b	c
bbb	3

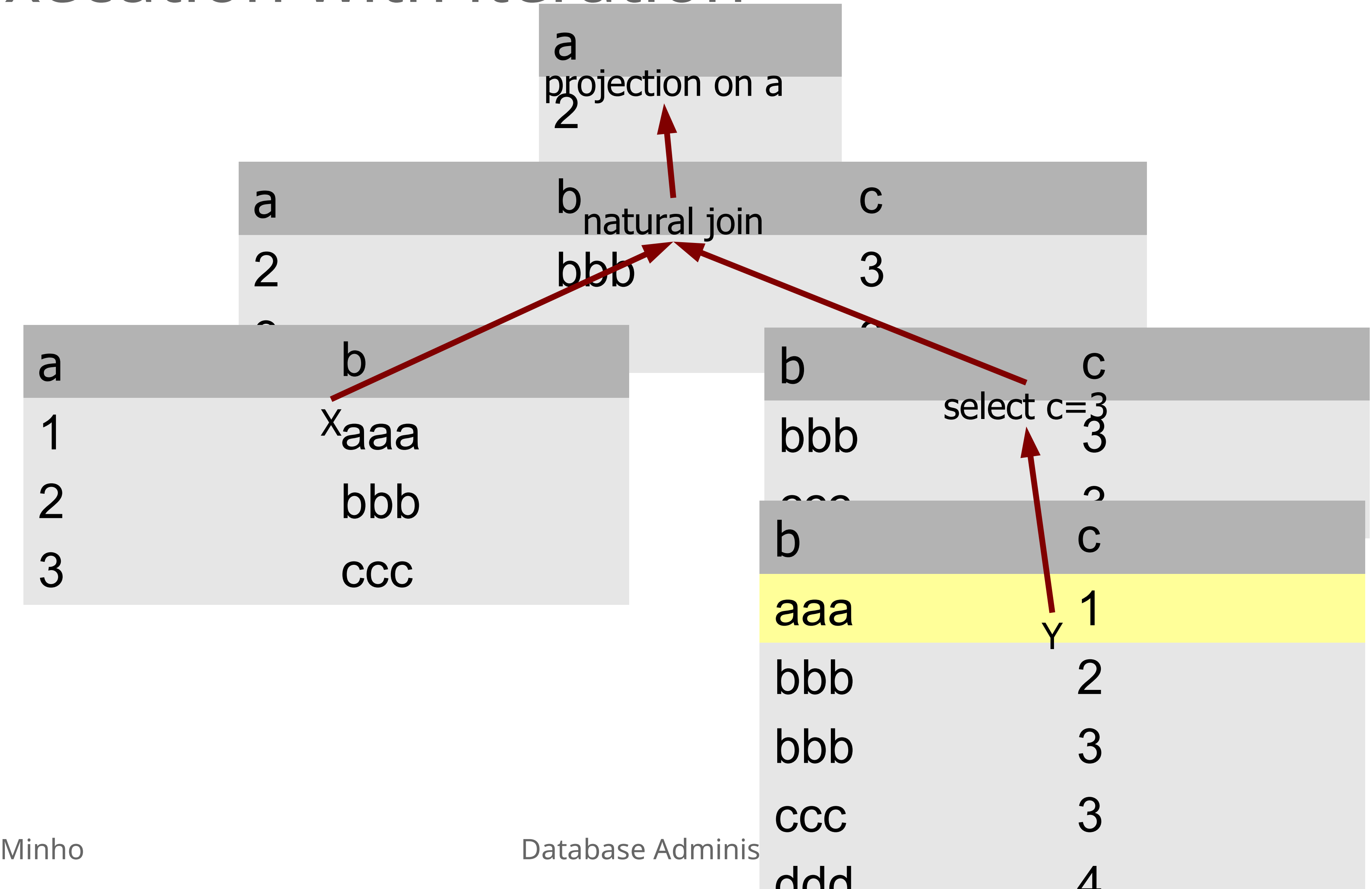
Y

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

Iteration

- Each operator is an object:
  - Interface similar to java.util.Iterator:
    - open() - get ready to return first record
    - next() - return next record
    - close() - no more records required
  - Constructor parameters:
    - Other operator objects
- Computation order:
  - From leaves to root, for each record

Execution with iteration



a
2

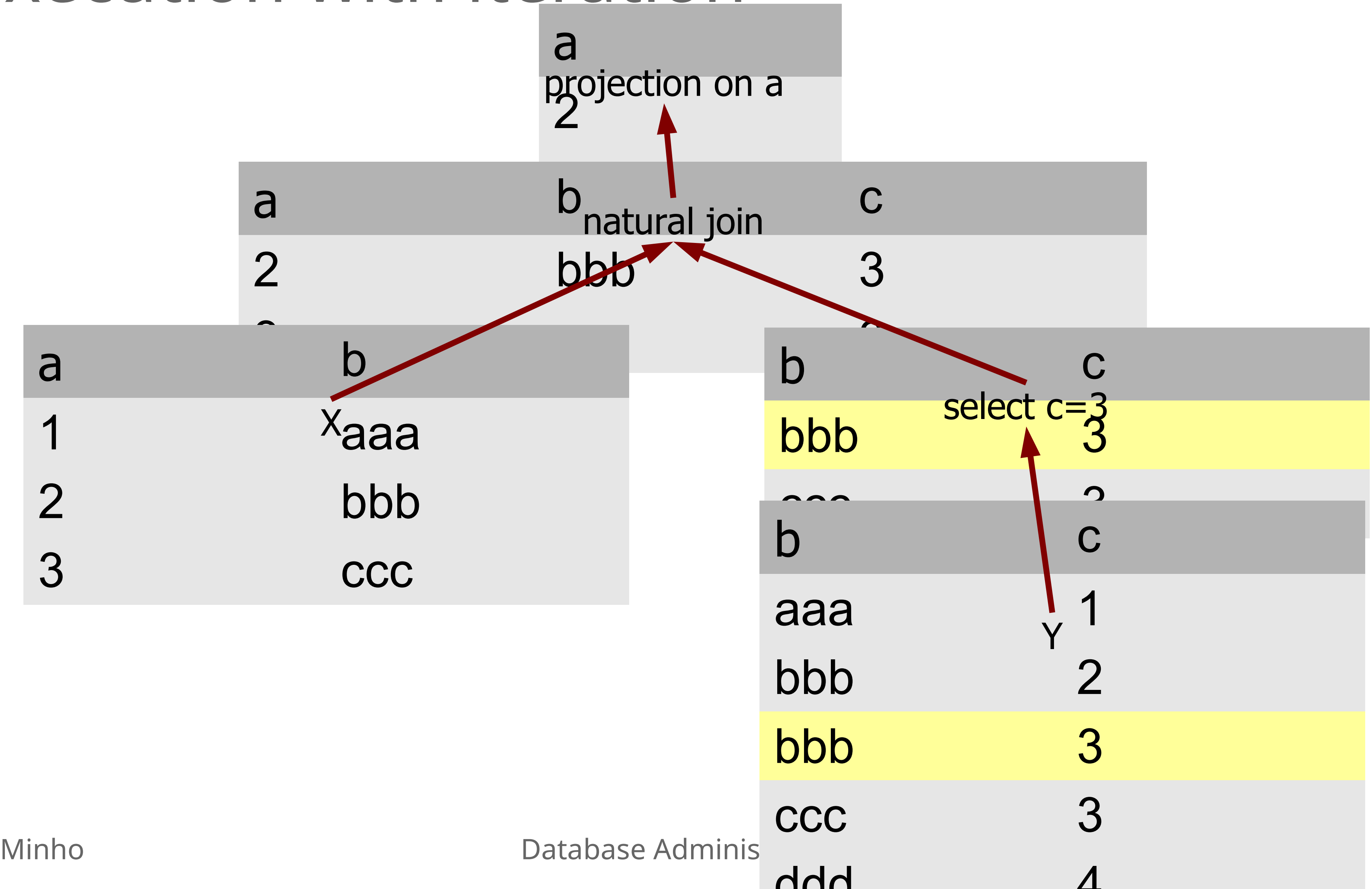
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

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

Execution with iteration



projection on a

a
2

natural join

a	b	c
2	bbb	3

X

a	b
1	aaa
2	bbb
3	ccc

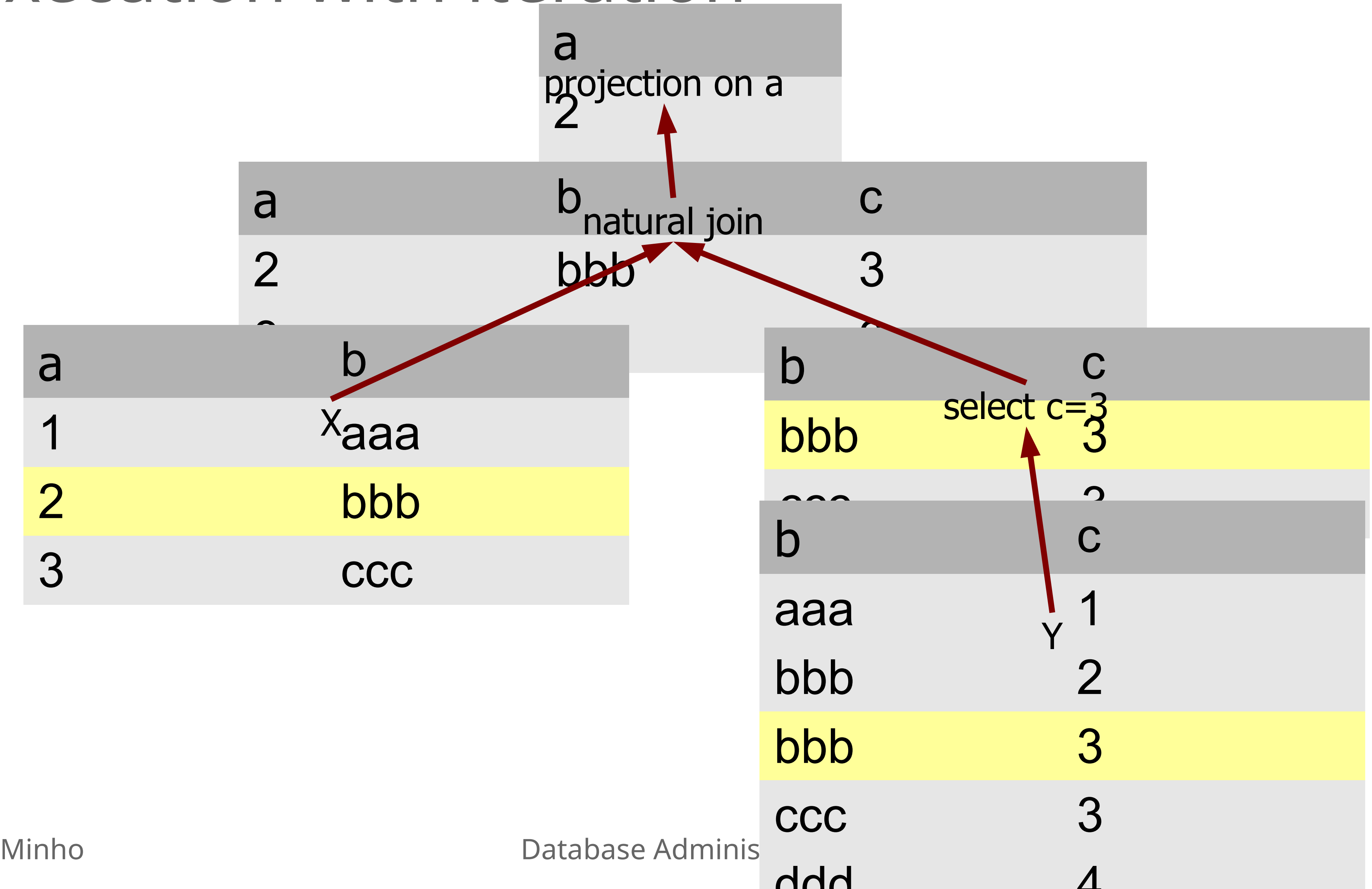
select c=3

b	c
bbb	3

Y

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

Execution with iteration



a
2

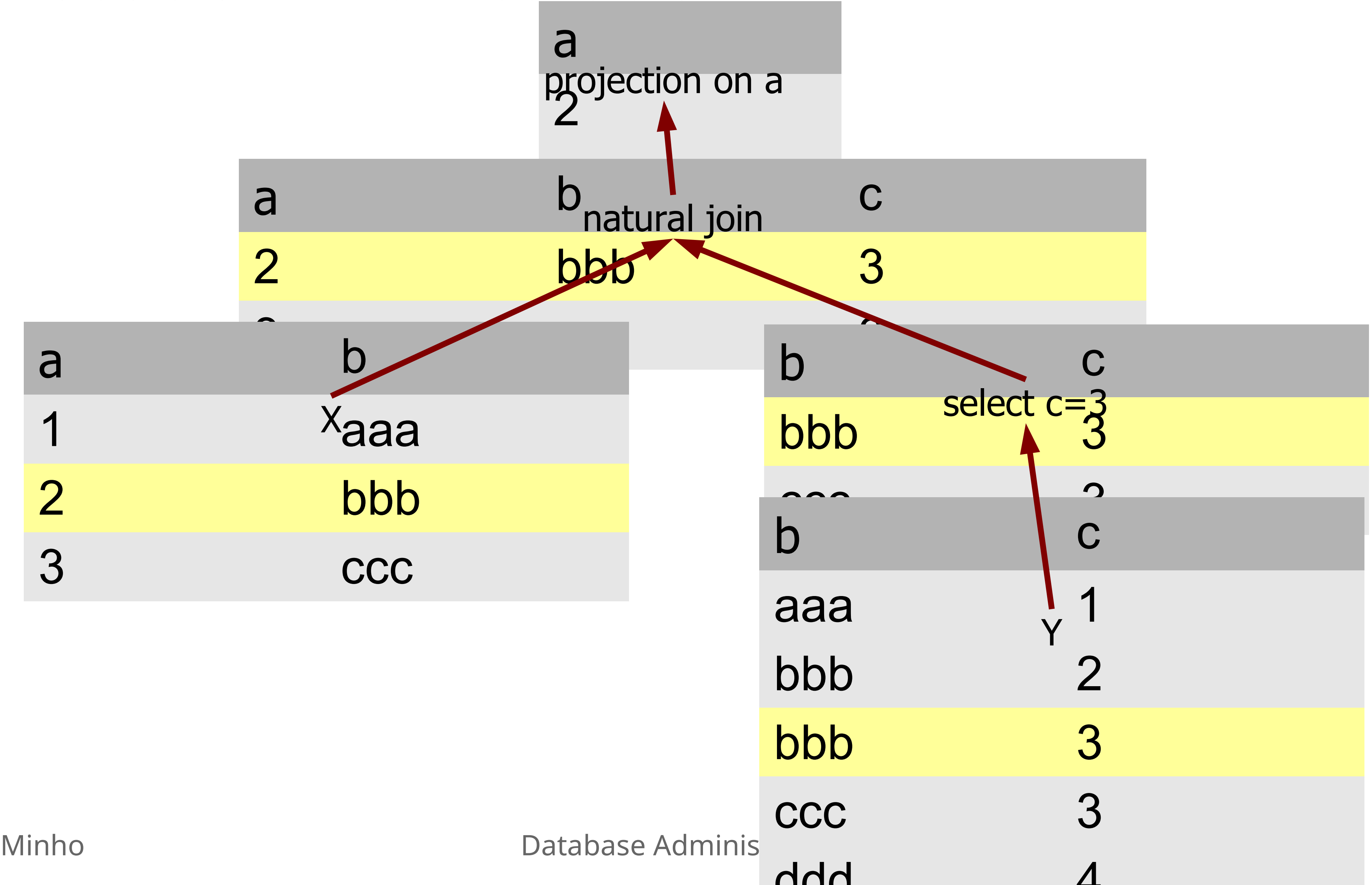
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3

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

Execution with iteration



projection on a

a
2

natural join

a	b	c
2	bbb	3

X

a	b
1	aaa
2	bbb
3	ccc

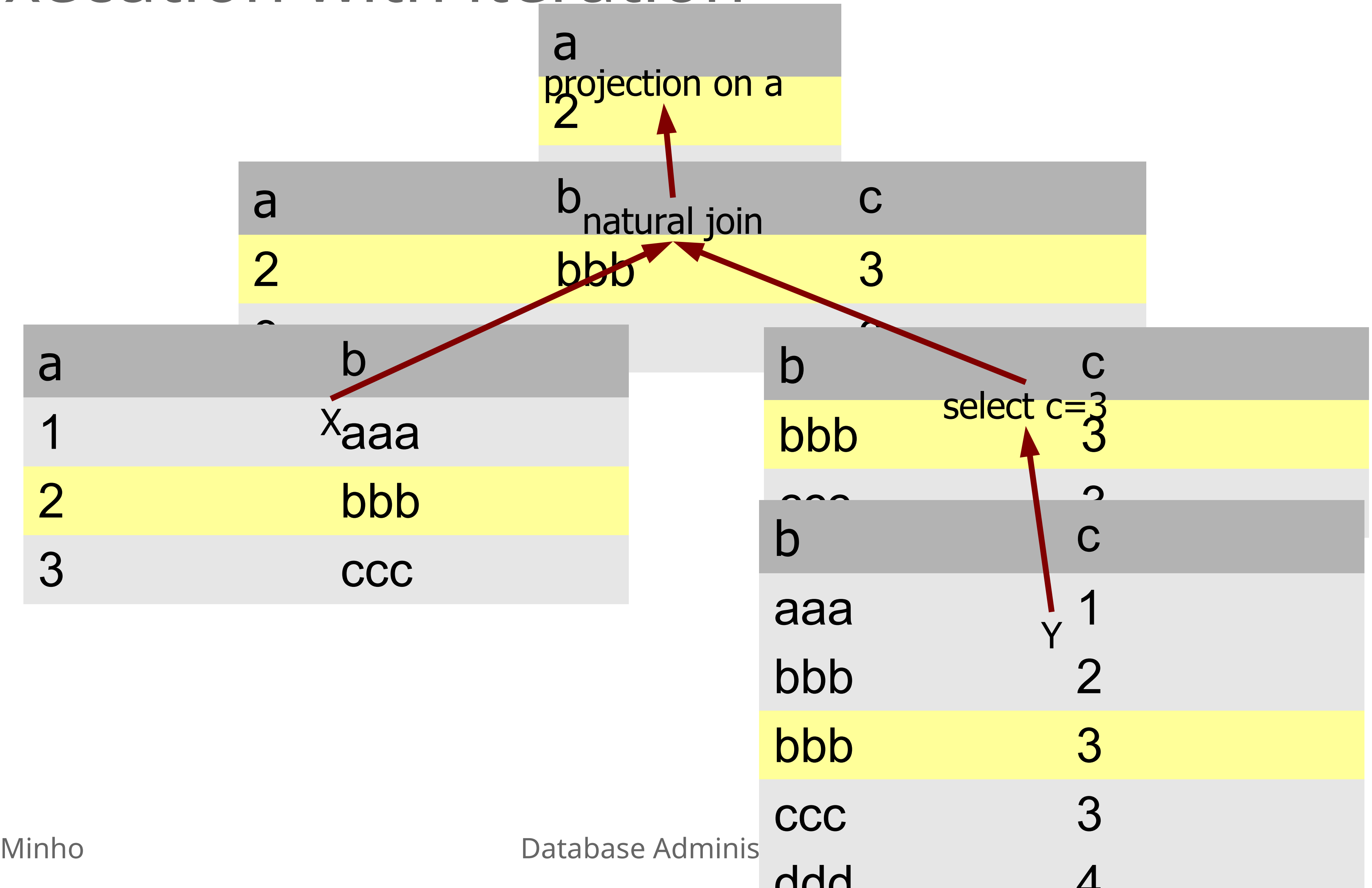
select c=3

b	c
bbb	3

Y

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

Execution with iteration



a
2

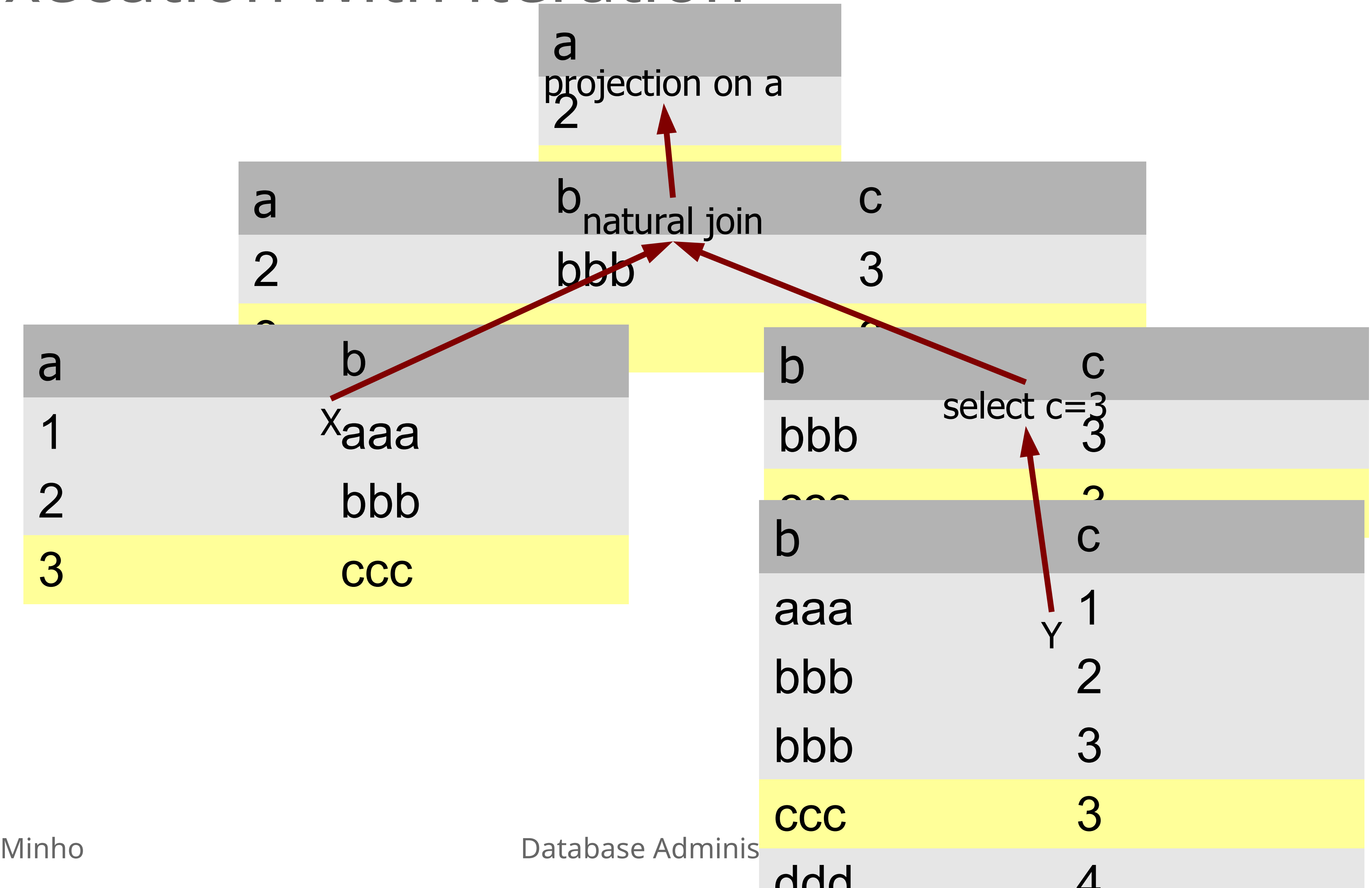
a	b	c
2	bbb	3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

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

Execution with iteration



a
2

a	b	c
2	bbb	3
3		3

a	b
1	aaa
2	bbb
3	ccc

b	c
bbb	3
ccc	3

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

Materialization vs Iteration

- Iteration avoids caching entire relations
- Materialization avoid reading records more than once
- Can mix both:
  - A materialization operator obtains all records upon first invocation of open
  - Returns records from cached copy on iteration

Roadmap

- What physical operators exist for each logical operation?
- 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;”

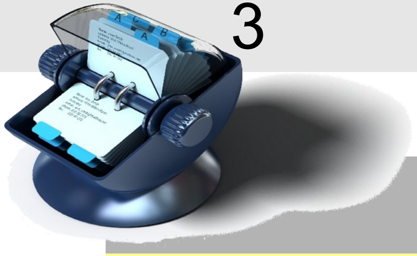
One-pass, full relation, binary

- Union, difference, intersection, product, join:
  - Read and cache the smallest relation
  - Organize for fast look-up (e.g. hash)
  - Read and operate on each record from the largest relation

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

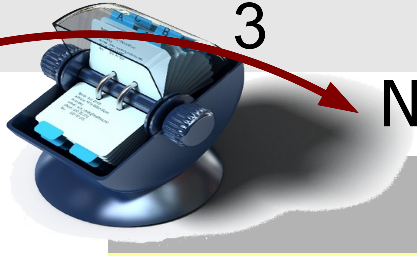
	c
bbb	3
ccc	3

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

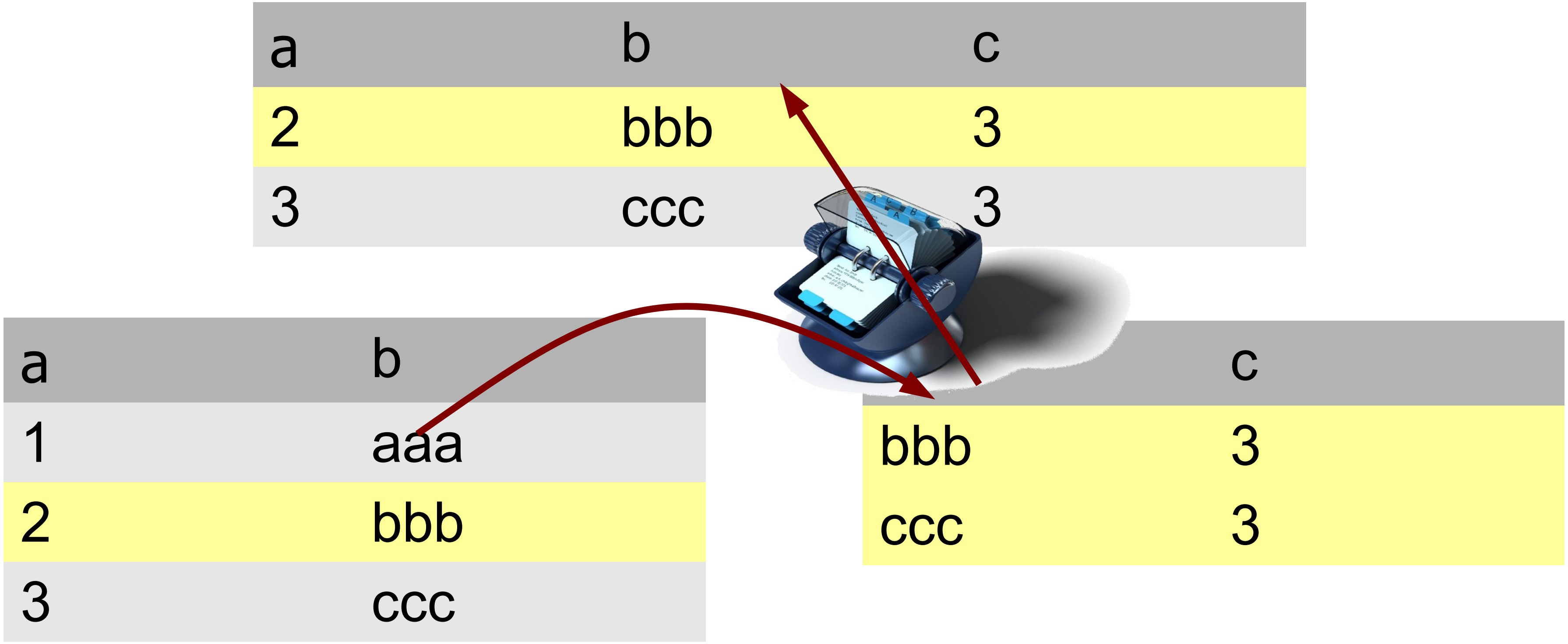


Not found

	c
bbb	3
ccc	3

One-pass, full relation, binary

- Test each record from the largest relation:

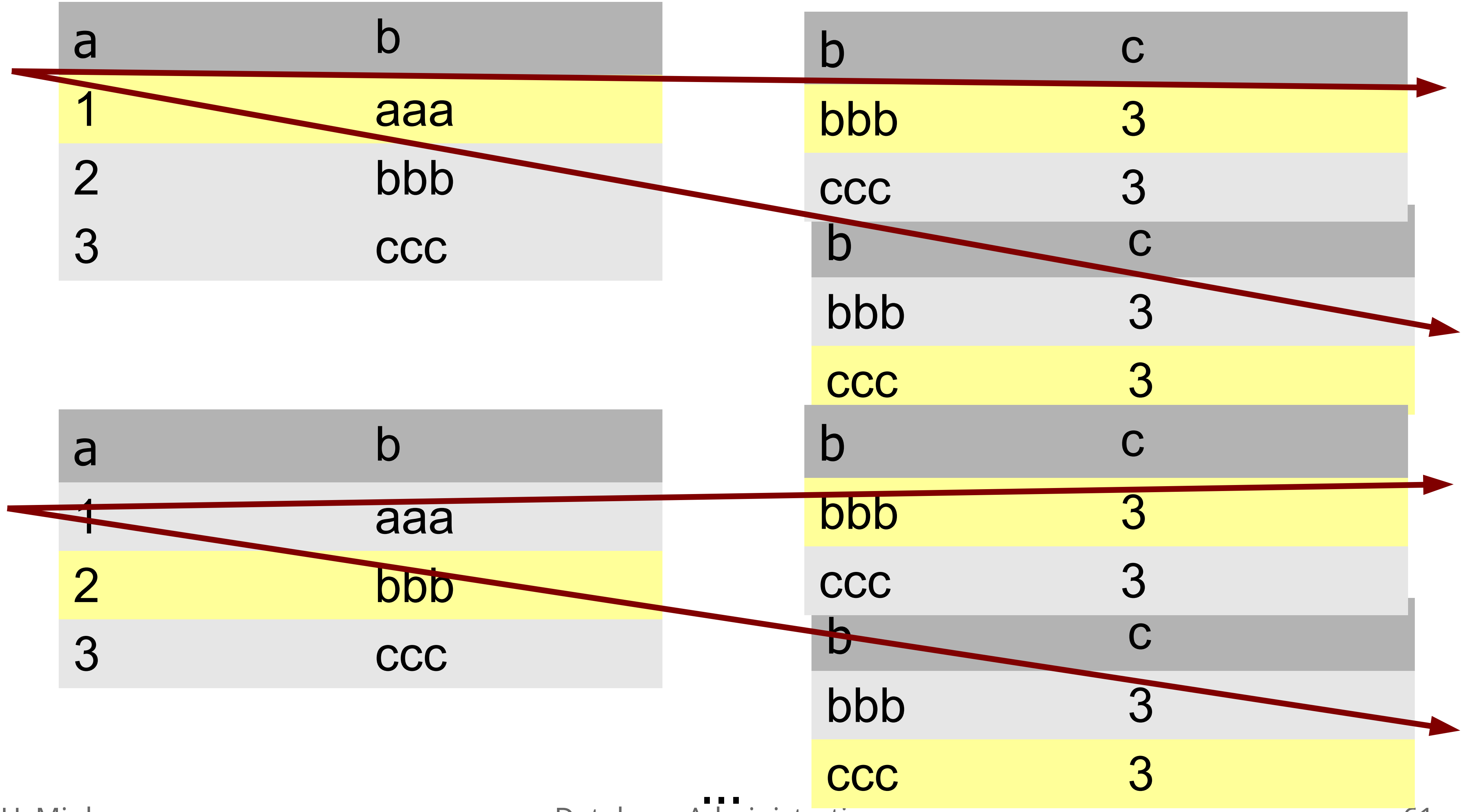


a	b	c
2	bbb	3
3	ccc	3

a	b
1	aaa
2	bbb
3	ccc

	c
bbb	3
ccc	3

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

...

Nested-loop join (NLJ)

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

Block-based NLJ

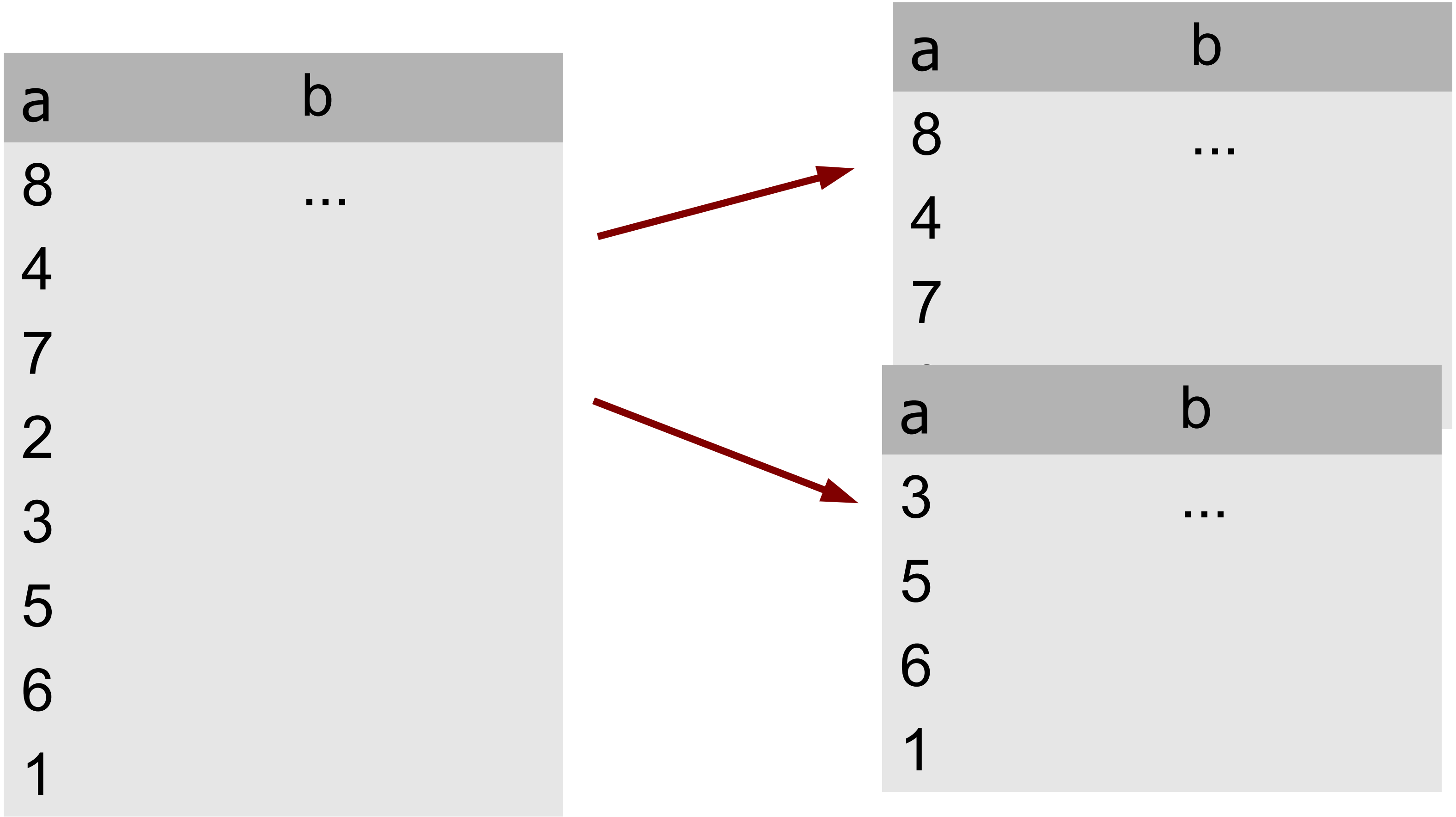
- Much smarter: Execute NLJ by blocks
- Memory requirements:
  - One block from each relation
- Operations:
  - If outer loop has N records / B blocks (B << N)
  - Reads inner relation B times (B << N!)

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	

a	b
3	…
5	
6	
1	

Merge-sort

- Load and sort each of them:

a	b
2	…
4	
7	
8	

a	b
3	…
5	
6	
1	

Merge-sort

- Load and sort each of them:

a	b
2	…
4	
7	
8	

a	b
1	…
3	
5	
6	

Merge-sort

- When iterating, select the next record from the fragment with the next key:

a	b
2	…
4	
7	
8	

a	b
1	…
3	
5	
6	

Merge-sort

- When iterating, select the next record from the fragment with the next key:

a	b
2	...
4	
7	
8	

a	b
1	...
3	
5	
6	

Merge-sort

- When iterating, select the next record from the fragment with the next key:

a	b
2	…
4	
7	
8	

a	b
1	…
3	
5	
6	

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;”

Two-pass, full relation, binary

- Union, difference, intersection, product, join:
  - Read record R1 from sorted relation T1
  - Read record R2 from sorted relation T2
  - If R1 = R2:
    - Use R1 and R2
  - If R1 < R2:
    - R1 does not exist in T2
    - Skip R1
  - If R2 < R1
    - ….

Conclusion

- There are a number of options for executing each query
- More options if we consider other data structures
- Varying performance:
  - Memory requirements
  - Number of iterations
  - Disk accesses
- What is the best one?
- How can it be discovered?