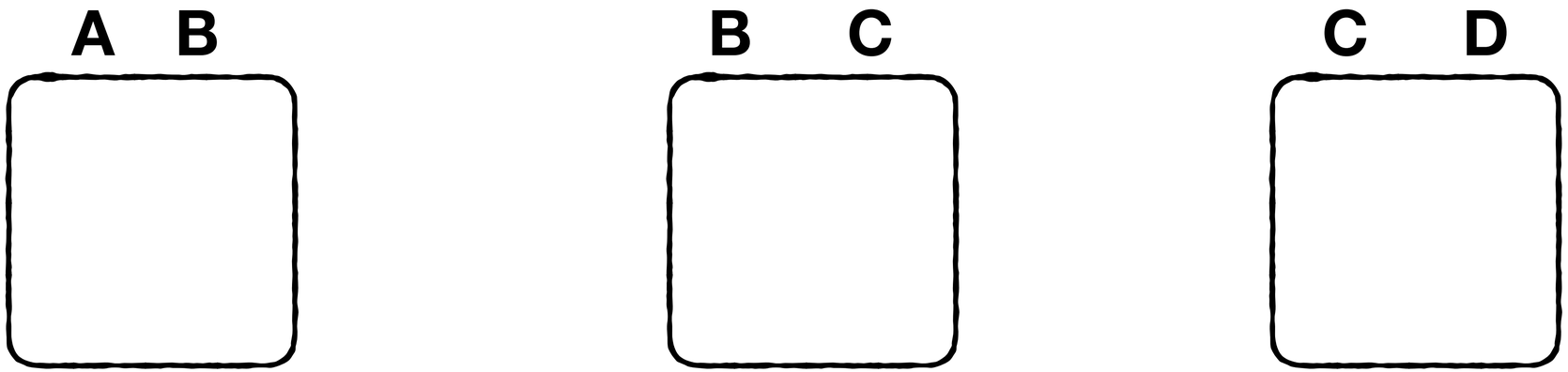
Query Evalaution

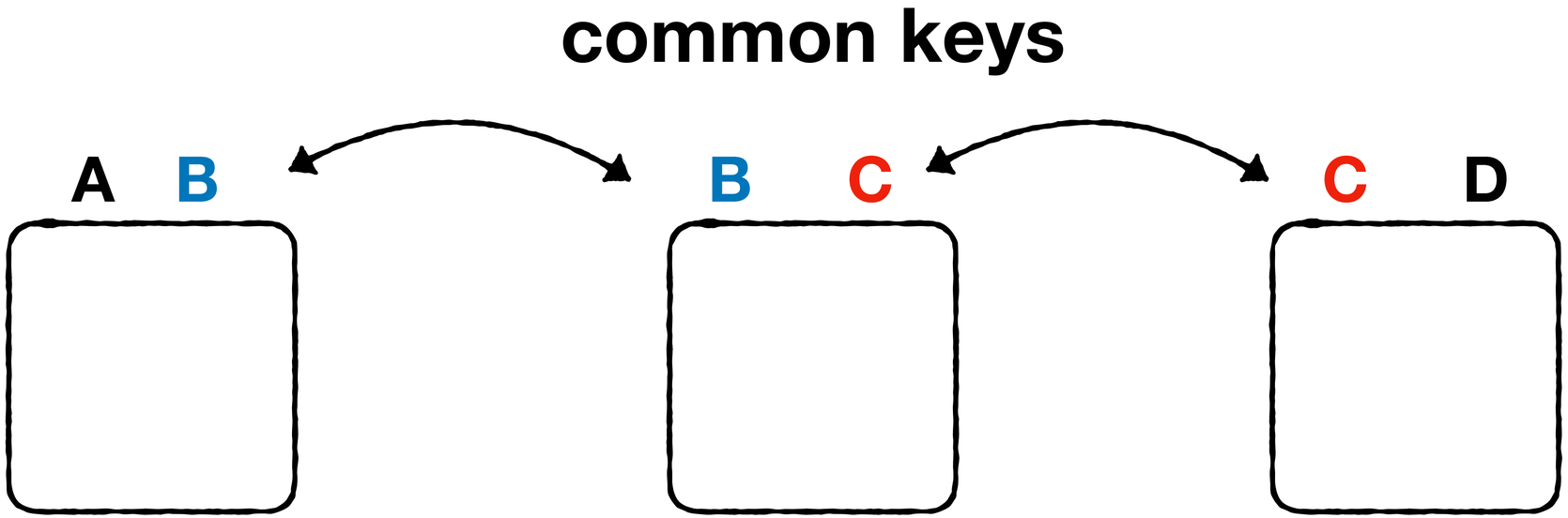
CSC443H1 Database System Technology

Niv Dayan

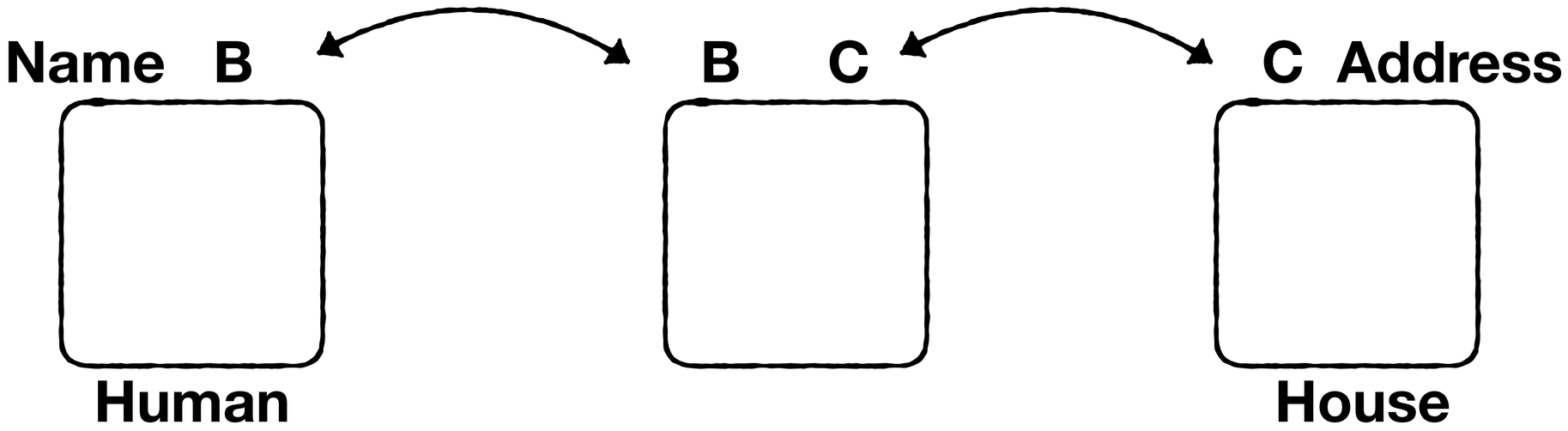
Consider three tables



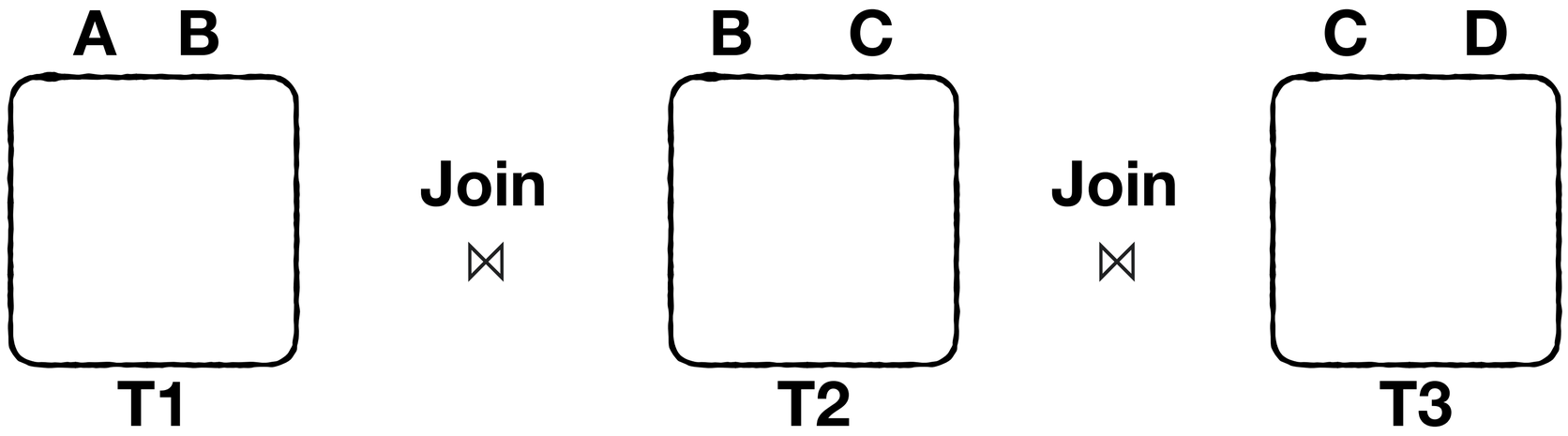
Consider three tables



Consider three tables

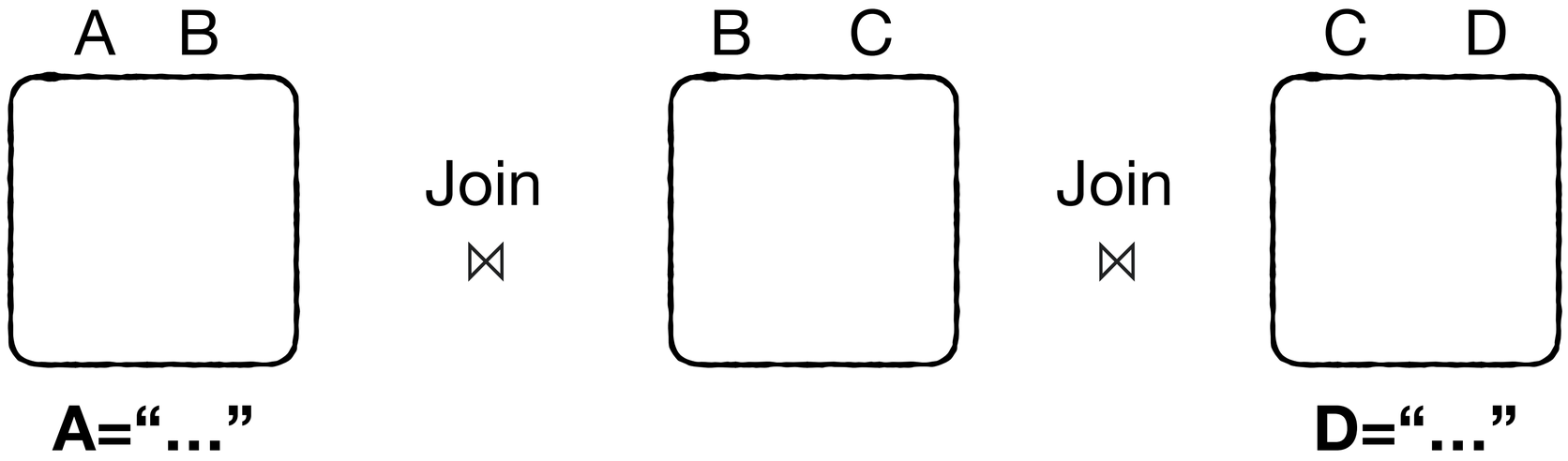


Consider three tables



Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C

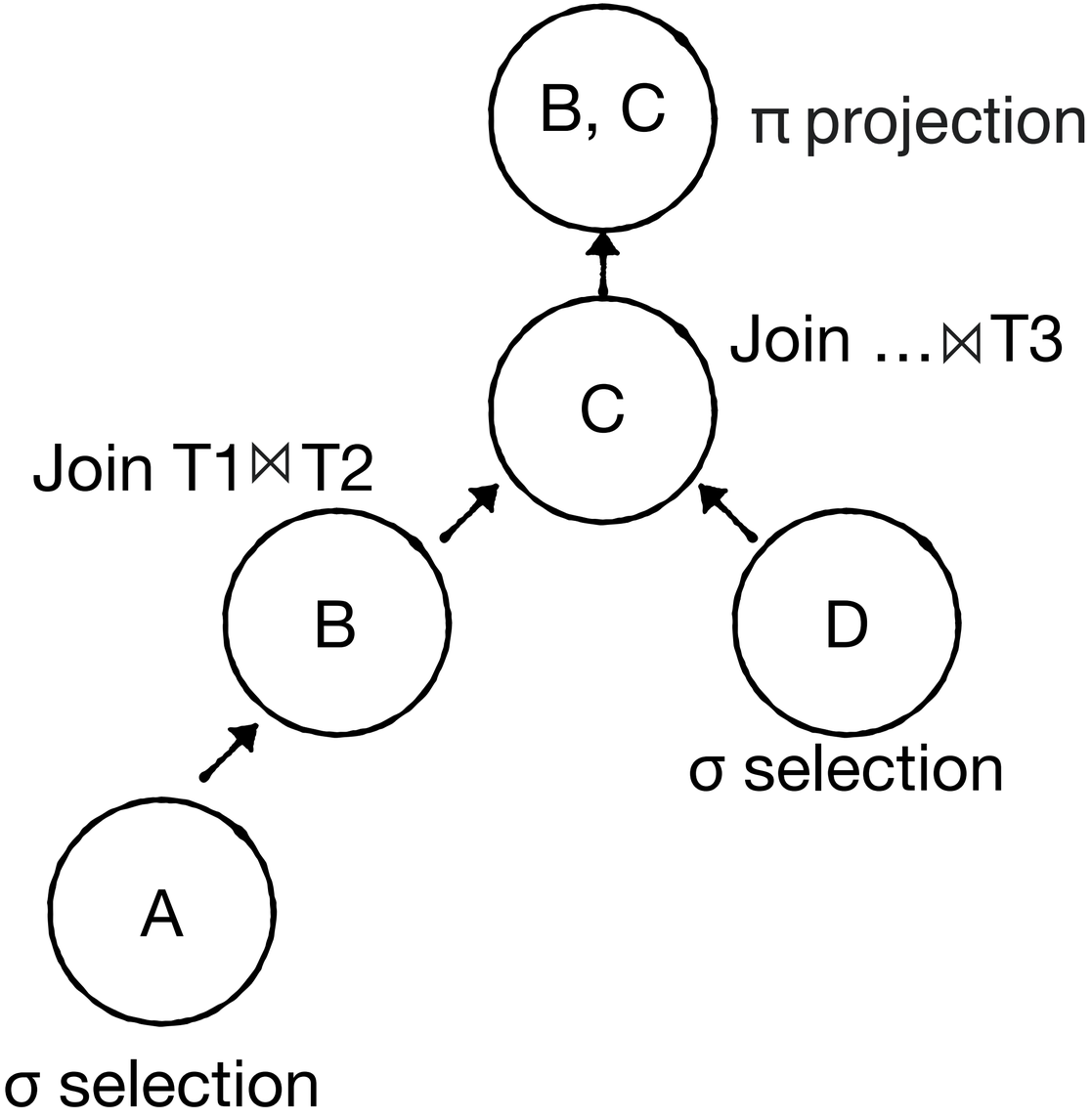
Consider three tables



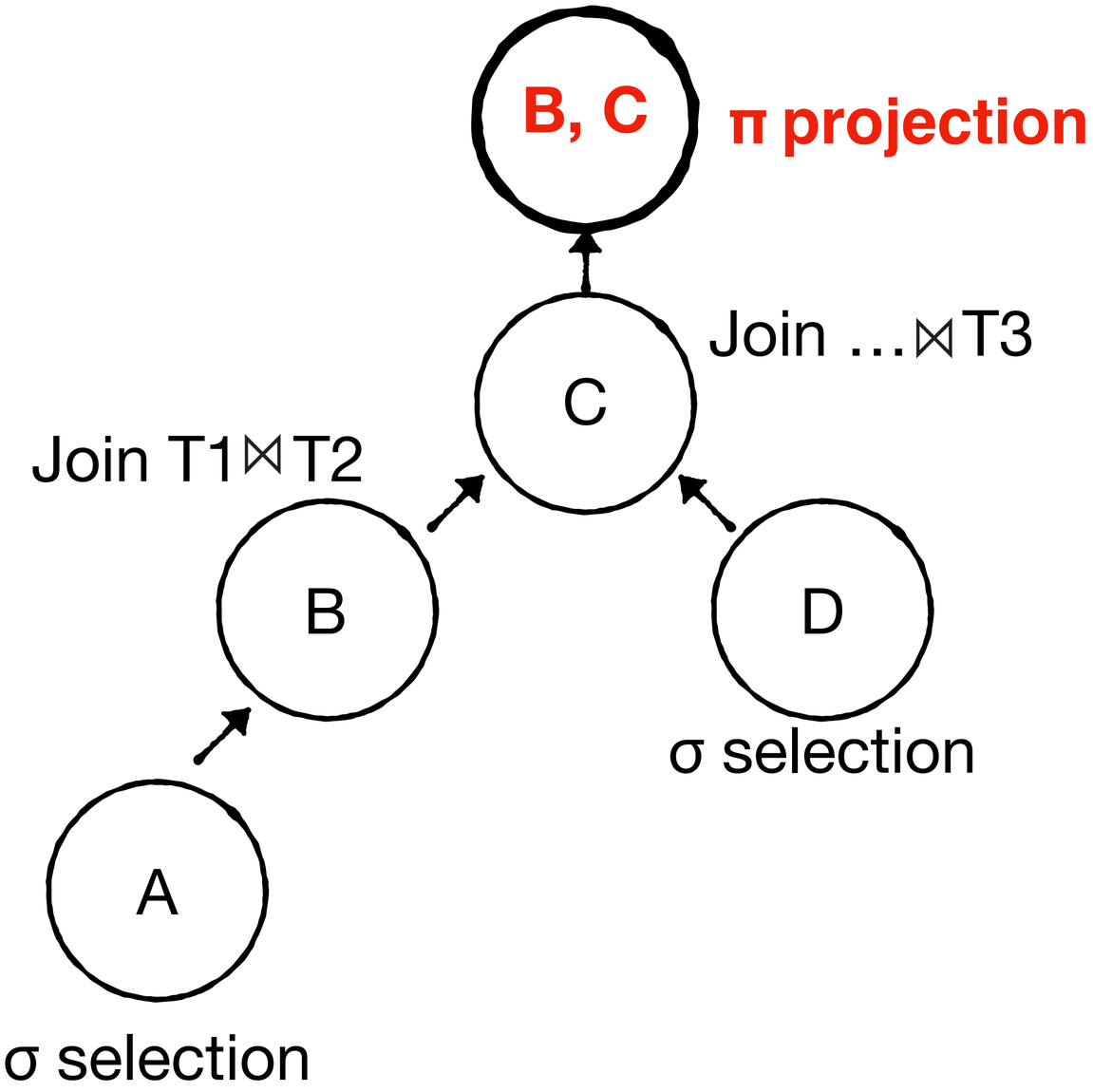
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"

Converted into a query plan of logical operators

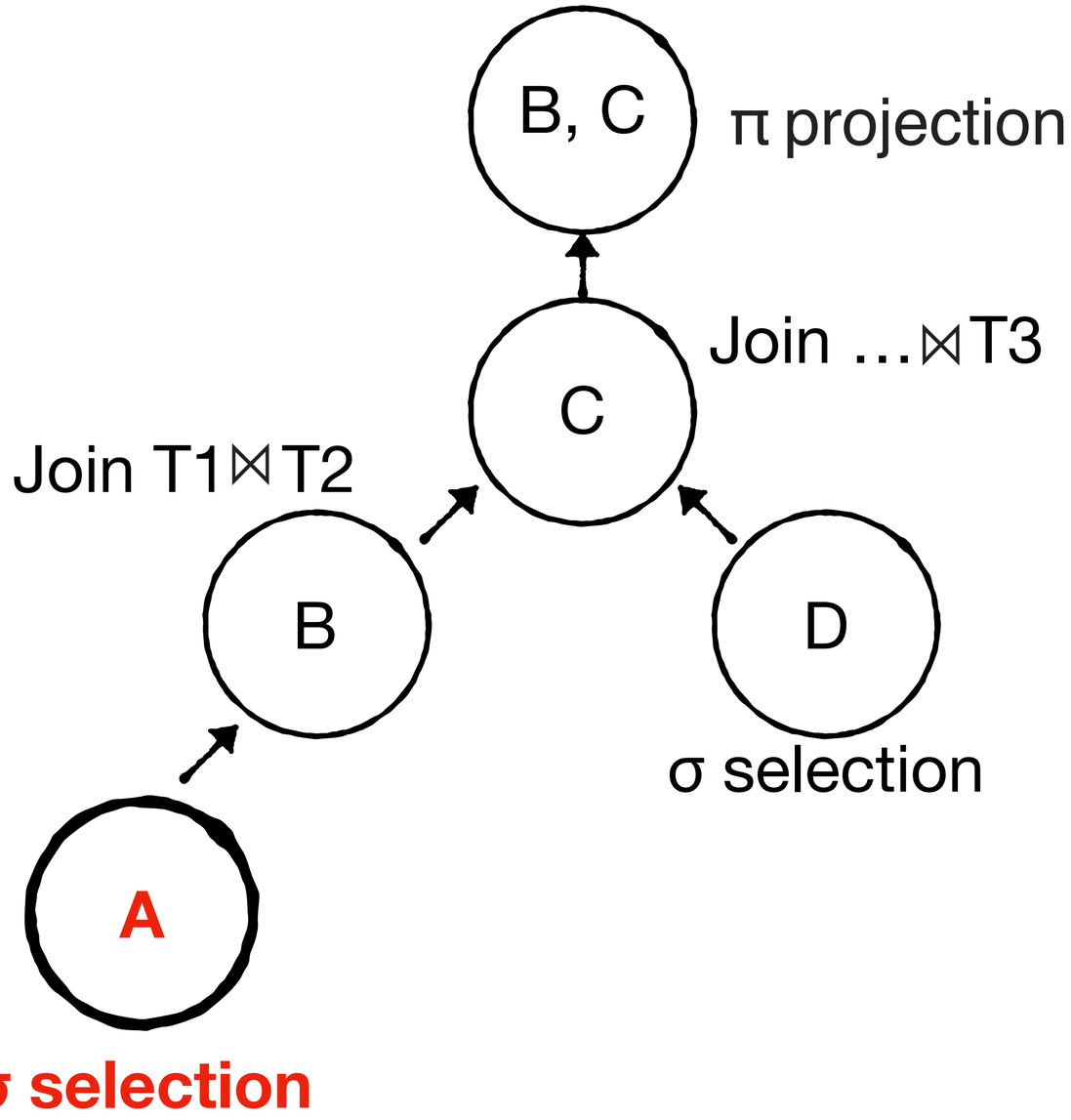
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"



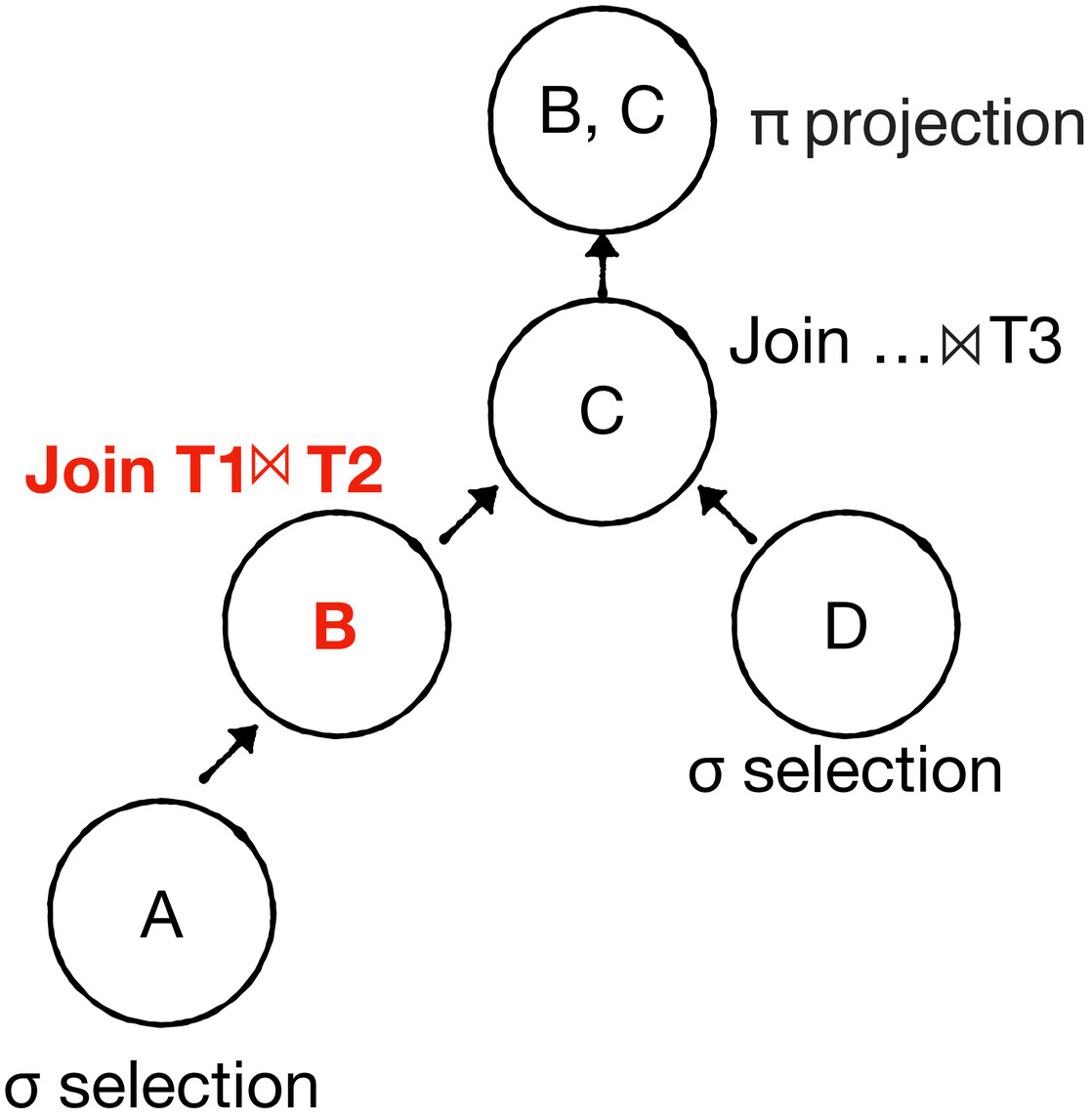
Select **A, D** from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"



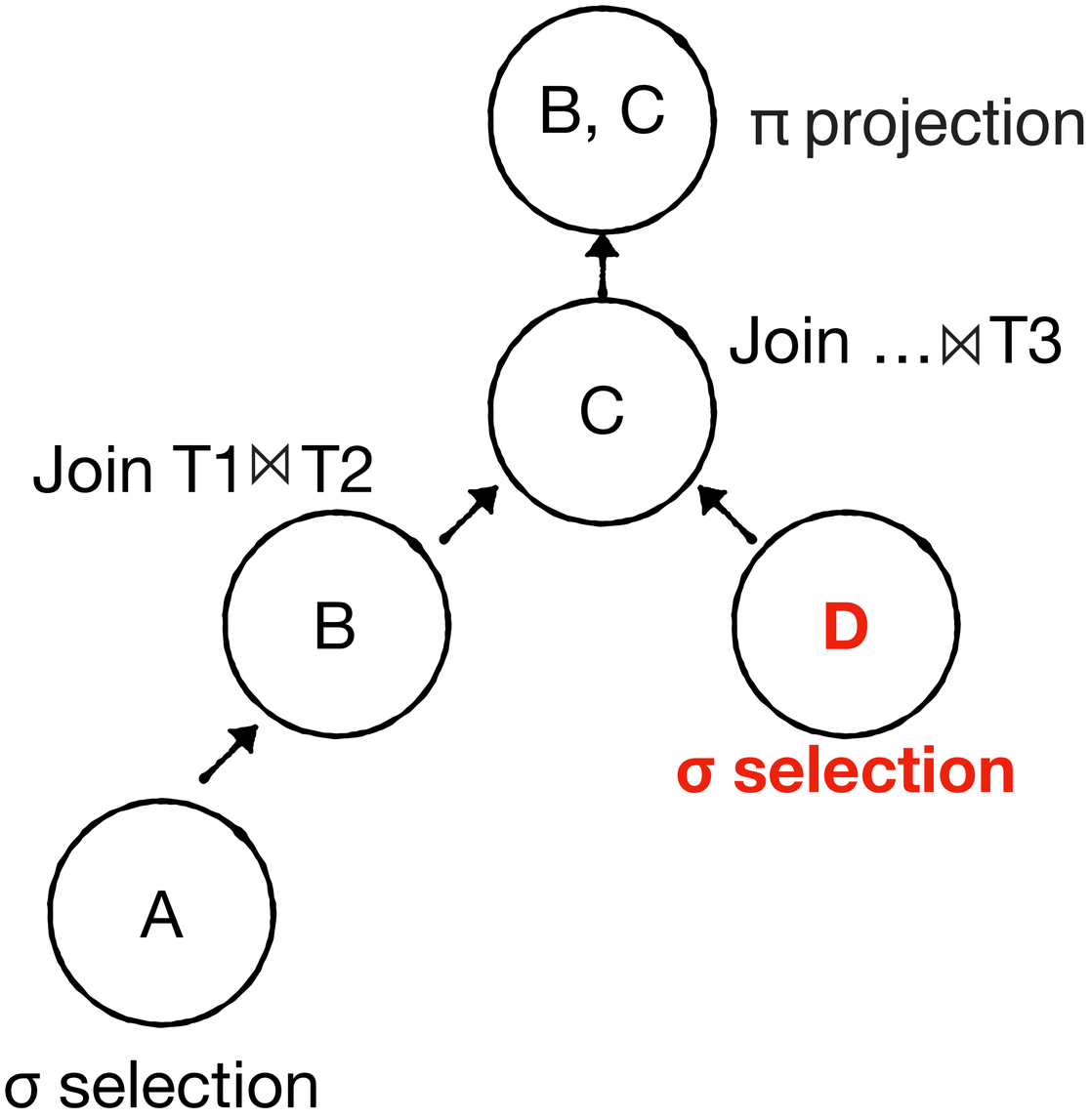
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and **A="…"** and D="…"



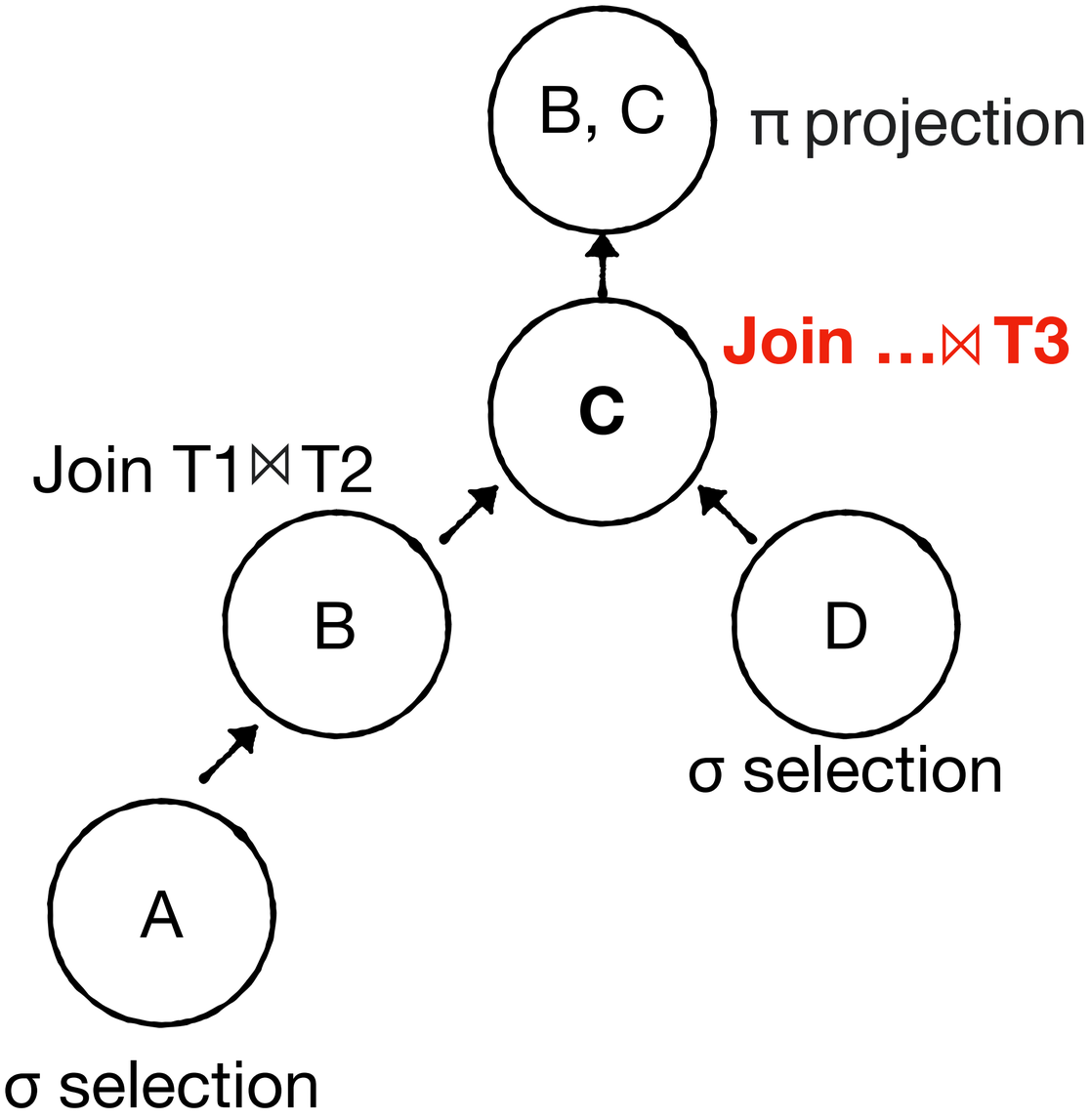
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"



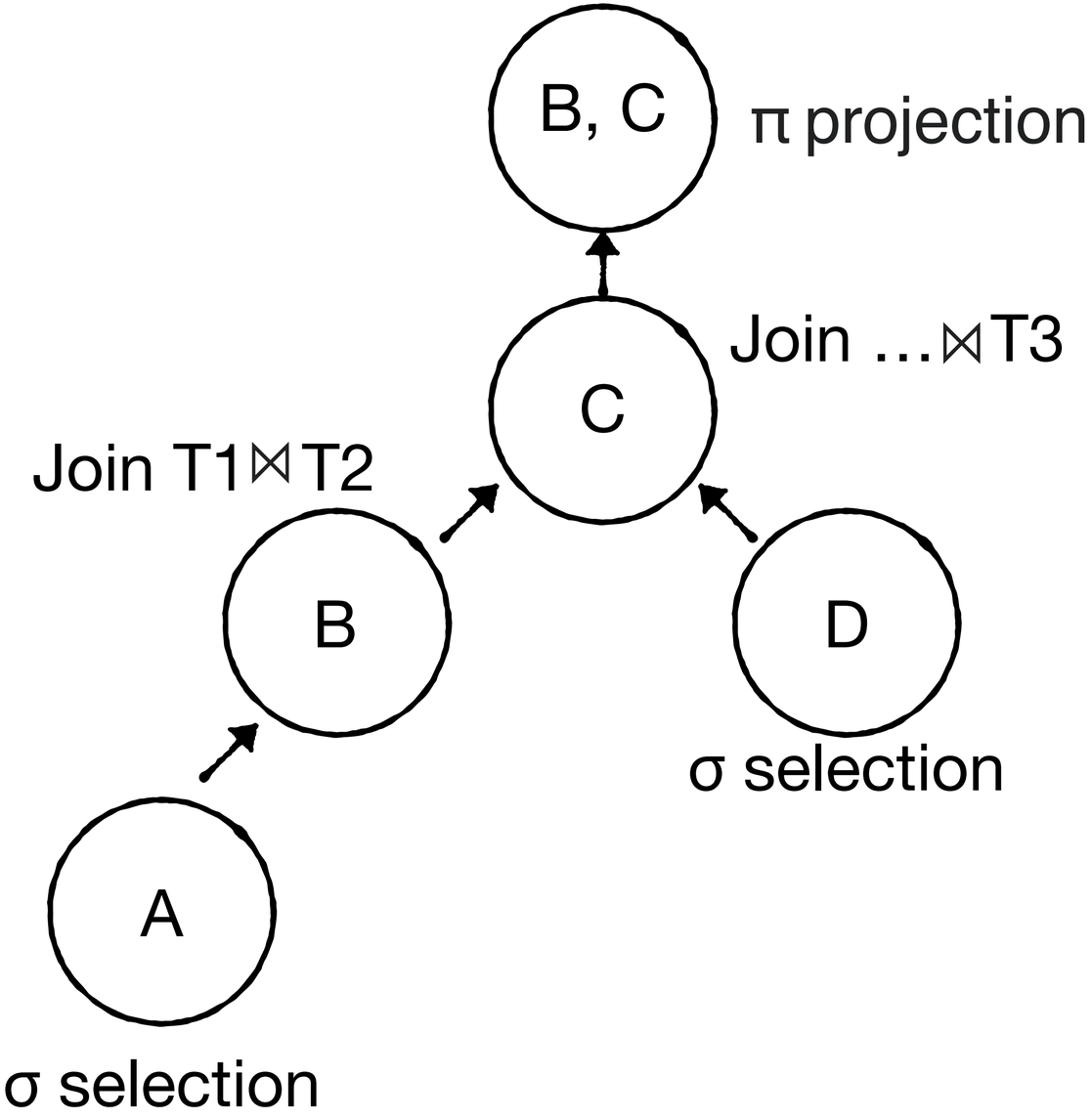
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and **D="…"**



Select A, D from T1, T2, T3 where
T1.B = T2.B and **T2.C = T3.C**
and A="…" and D="…"



All implement an iterator interface (the glue)

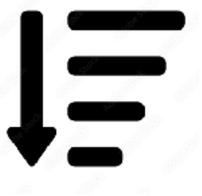


Selection

Projection

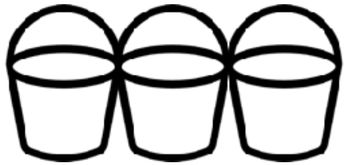
Join

Order by



Distinct

Group by



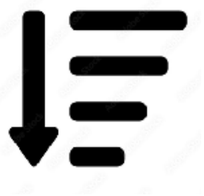
Each can be implemented using different algorithms

Selection

Projection

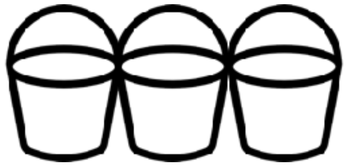
Join

Order by



Distinct

Group by



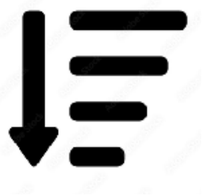
Each can be implemented using different algorithms

Selection

Projection

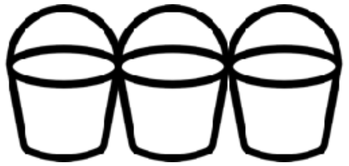
Join

Order by



Distinct

Group by

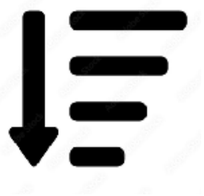


different trade-offs for different contexts

Most content

Selection

Order by

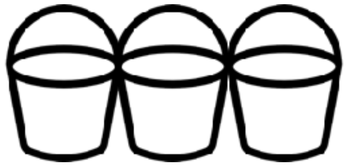


Projection

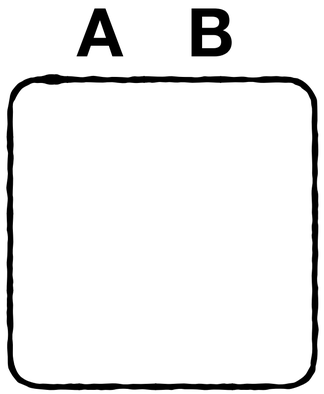
Distinct

Join

Group by



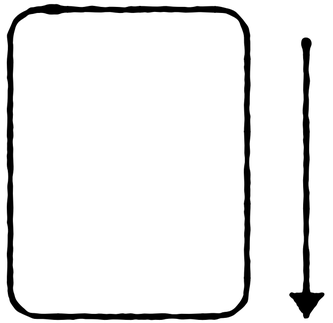
Selection



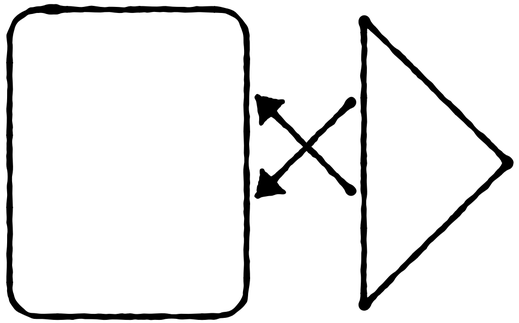
Select * from … where **A = “…”**

Select * from … where **A = “…”**

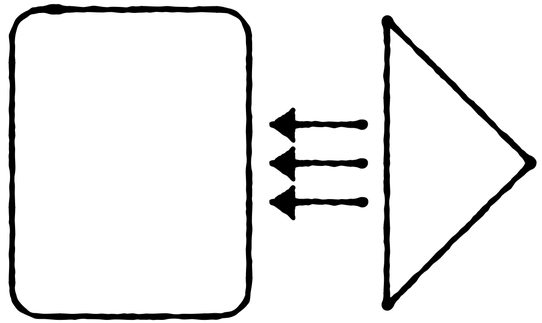
Scan



Unclustered Index

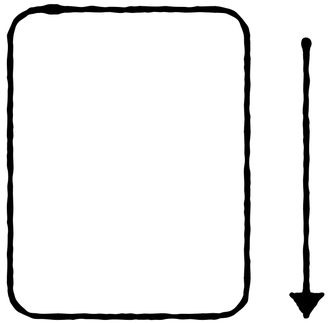


Clustered Index



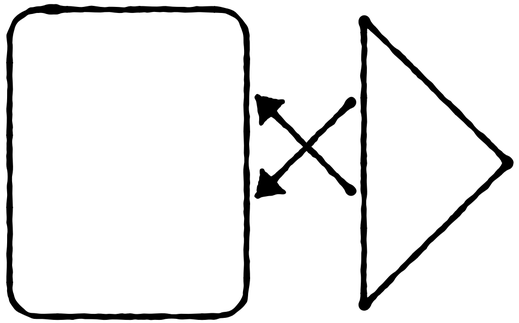
Select * from … where **A = “…”**

Scan

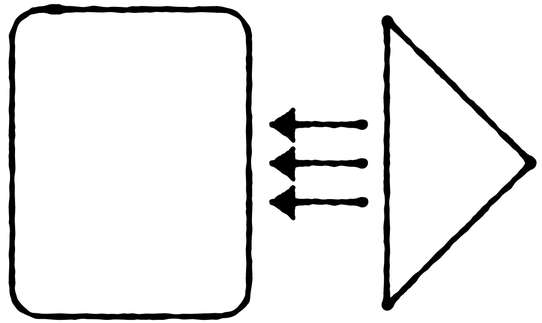


O(N/B)

Unclustered Index

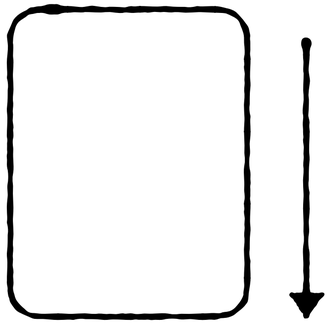


Clustered Index



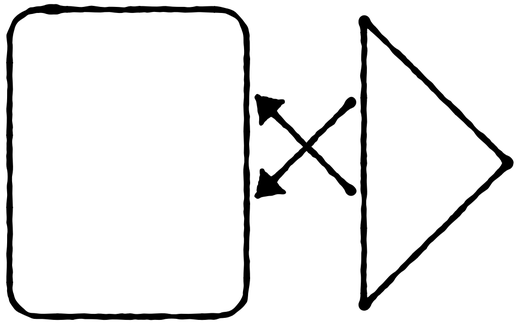
Select * from … where **A = “…”**

Scan



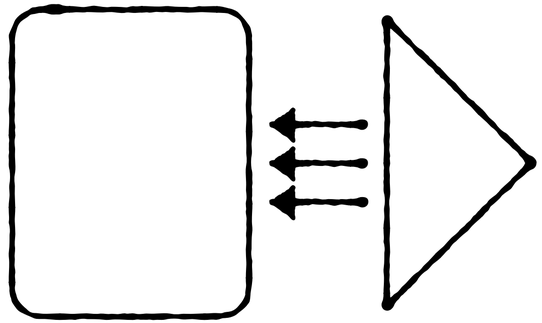
O(N/B)

Unclustered Index



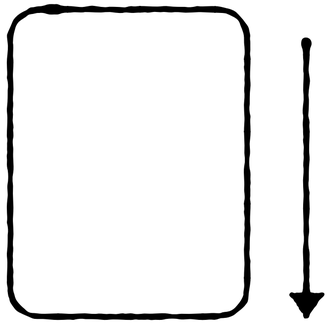
$O(\log_B(N)+S)$

Clustered Index



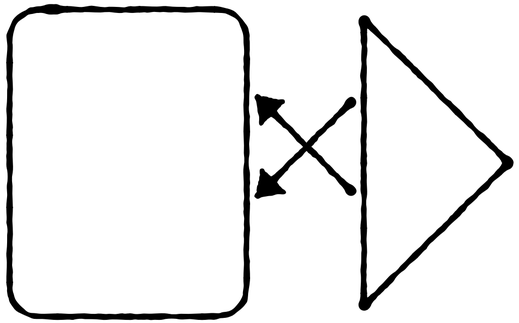
Select * from … where **A = “…”**

Scan



O(N/B)

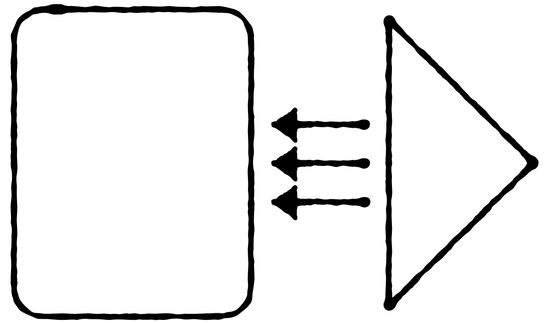
Unclustered Index



O(**log$_B$(N)**+S)

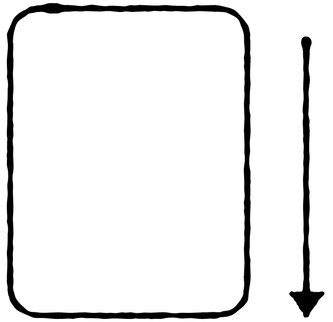
Assuming B-tree

Clustered Index



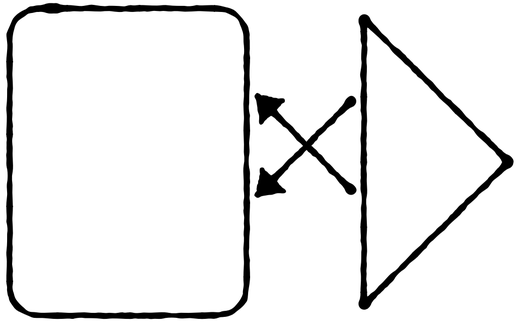
Select * from … where **A = "…"**

Scan



O(N/B)

Unclustered Index

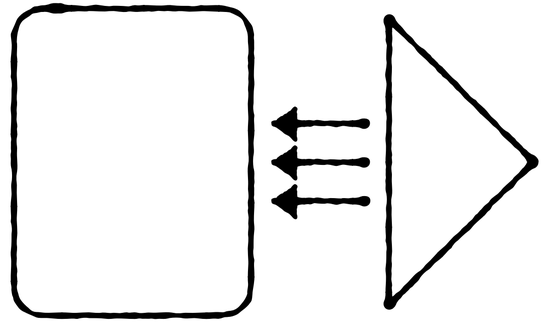


$O(\log_B(N)+\mathbf{S})$

Assuming B-tree

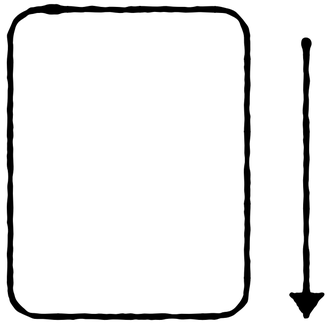
 # qualifying tuples

Clustered Index



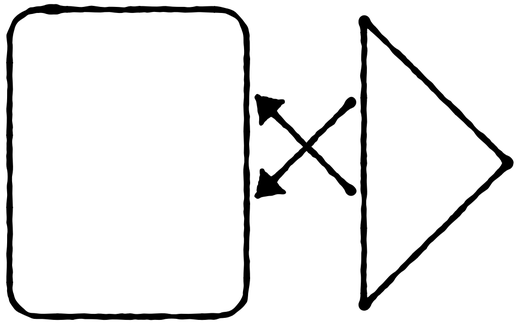
Select * from … where **A = “…”**

Scan



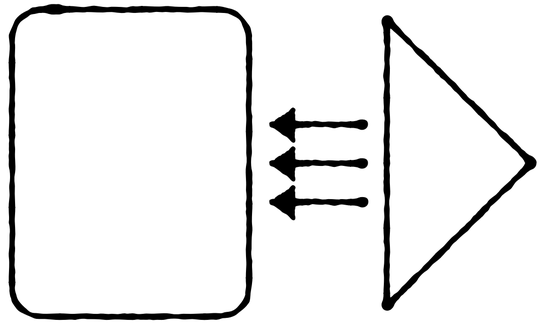
O(N/B)

Unclustered Index



$O(\log_B(N)+S)$

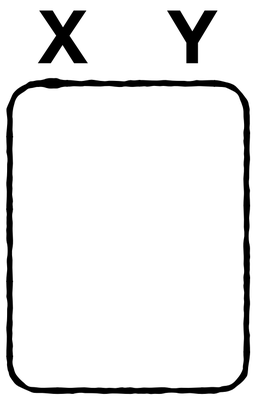
Clustered Index



$O(\log_B(N)+S/B)$

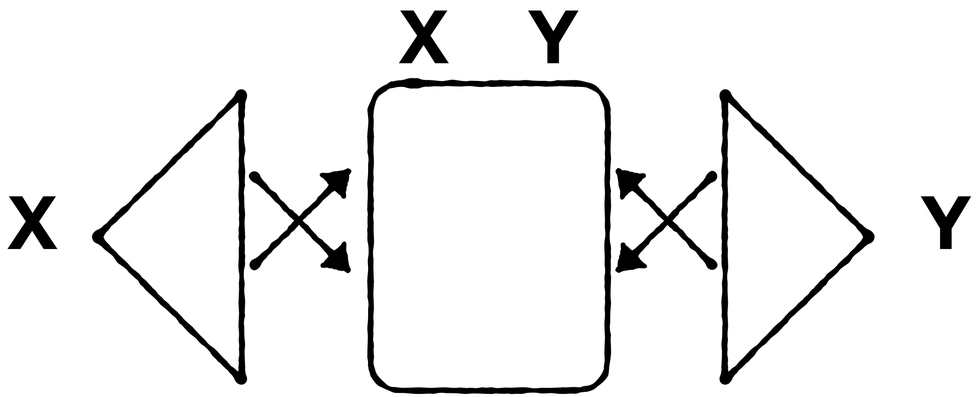
Select * from … where **X = “…” and Y=“…”**

How about when we have multiple selection conditions?

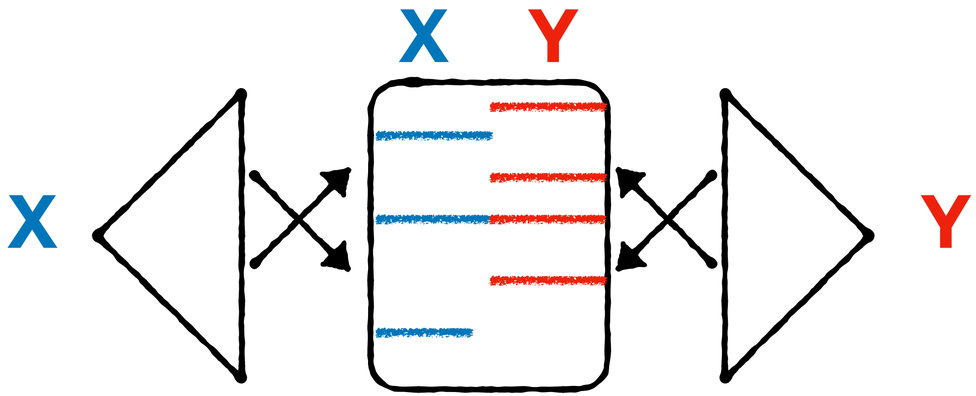


Select * from … where **X = “…” and Y=“…”**

Assume two unclustered indexes



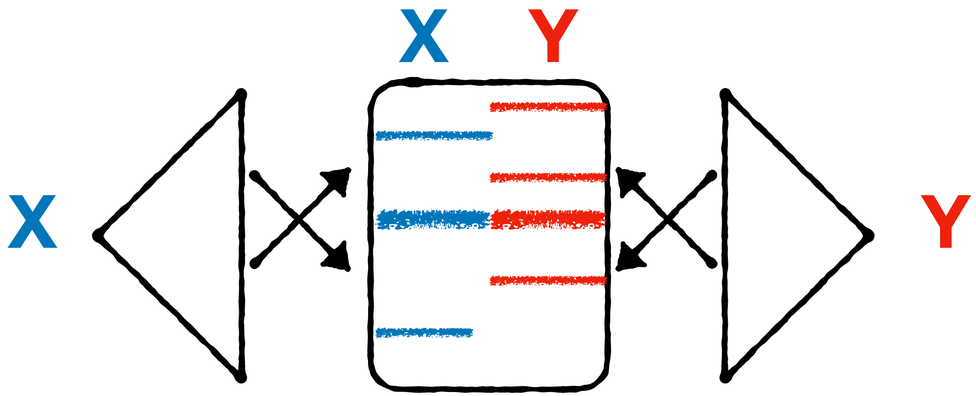
Select * from … where **X = “i” and Y=“j”**



Let $|X_i|$ and $|Y_j|$ denote the number of matching rows to A and to B, resp.

e.g., $|X_i| = 3$ and $|Y_j|=4$

Select * from … where **X = “i” and Y=“j”**

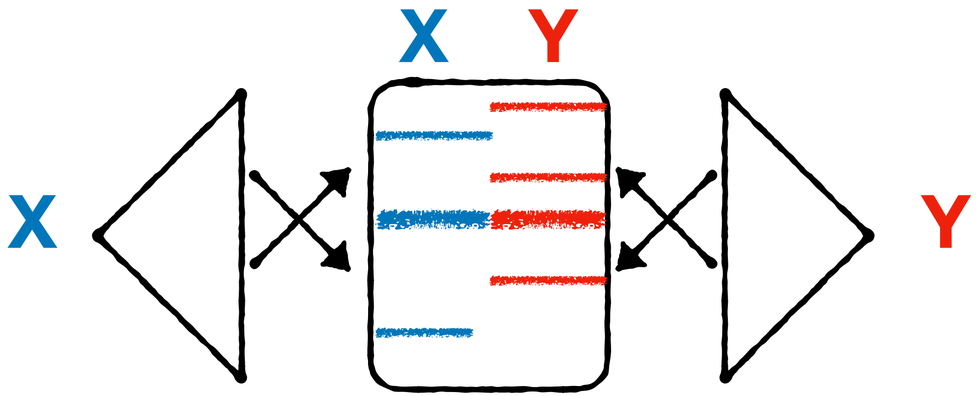


Let $|X_i|$ and $|Y_j|$ denote the number of matching rows to A and to B, resp.

e.g., $|X_i| = 3$ and $|Y_j|=4$

The query is looking for the intersection: $|X_i \cap Y_j| = 1$

Select * from … where **X = “i” and Y=“j”**



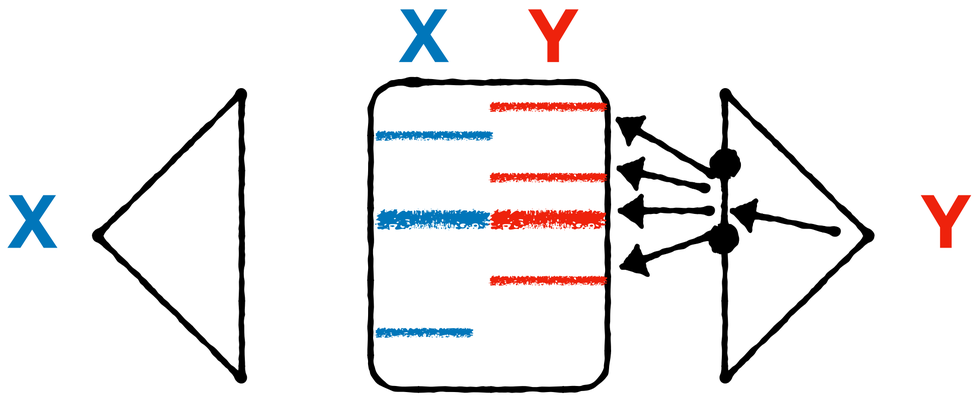
Let $|X_i|$ and $|Y_j|$ denote the number of matching rows to A and to B, resp.

e.g., $|X_i| = 3$ and $|Y_j|=4$

The query is looking for the intersection: $|X_i \cap Y_j| = 1$

Some ideas?

Select * from … where **X = “i” and Y=“j”**

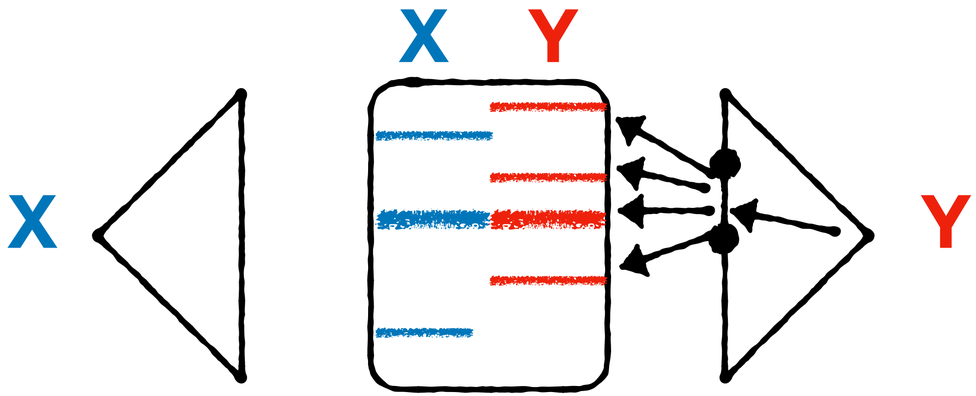


Algorithm 1: **Search index Y**

For each row in table, check if X matches

Cost:

Select * from … where **X = “i” and Y=“j”**

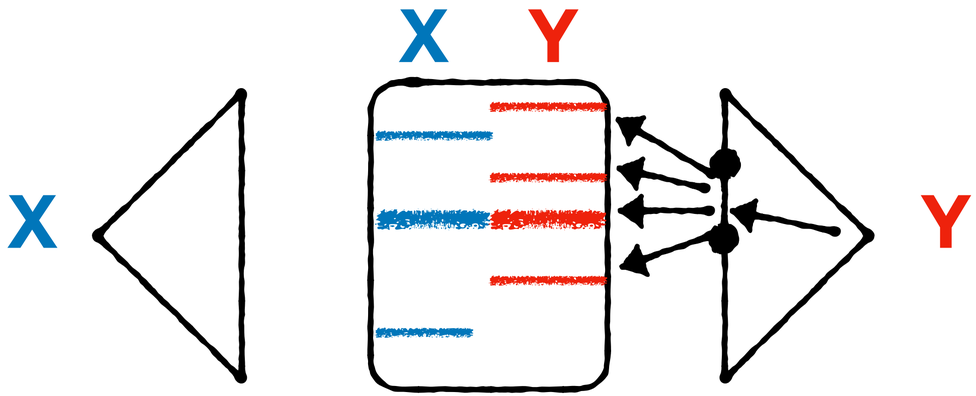


Algorithm 1: Search index Y

For each row in table, check if X matches

Cost: $\log_B(N) + |Y_j|/B + |Y_j|$ **I/O**

Select * from … where **X = “i” and Y=“j”**

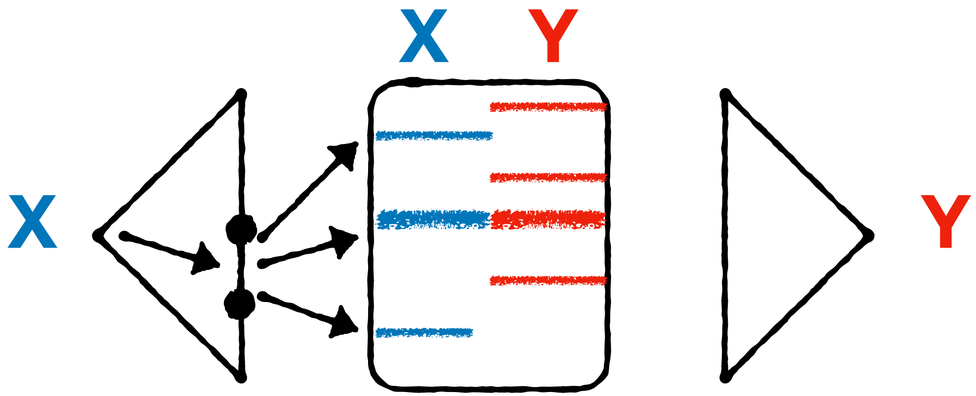


Algorithm 1: Search index Y

For each row in table, check if X matches

Cost: $\log_B(N) + |Y_j|$ **I/O**

Select * from … where **X = “i” and Y=“j”**

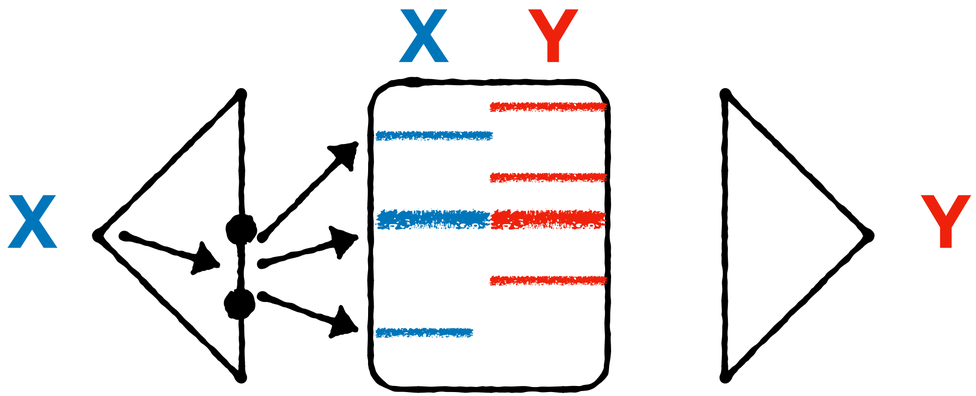


Algorithm 2: **Search index X**

For each row in table, check if Y matches

Cost:

Select * from … where **X = “i” and Y=“j”**

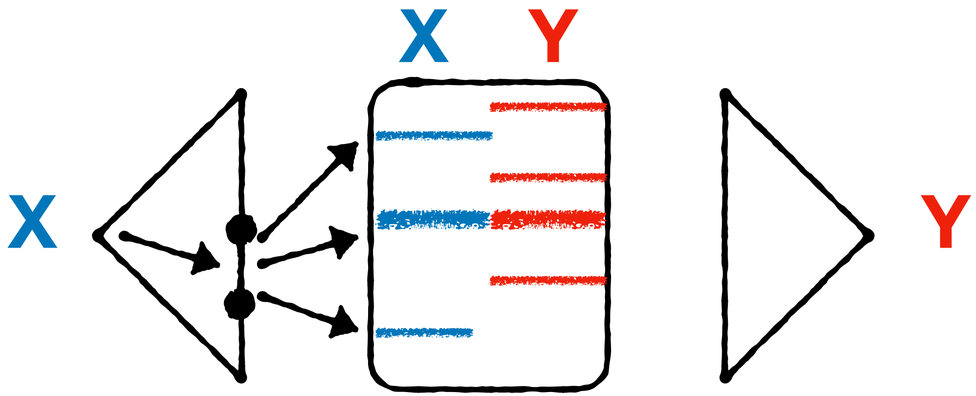


Algorithm 2: Search index X

For each row in table, check if Y matches

Cost: $\mathbf{log_B(N) + |X_i|/B + |X_i|}$ **I/O**

Select * from … where **X = “i” and Y=“j”**

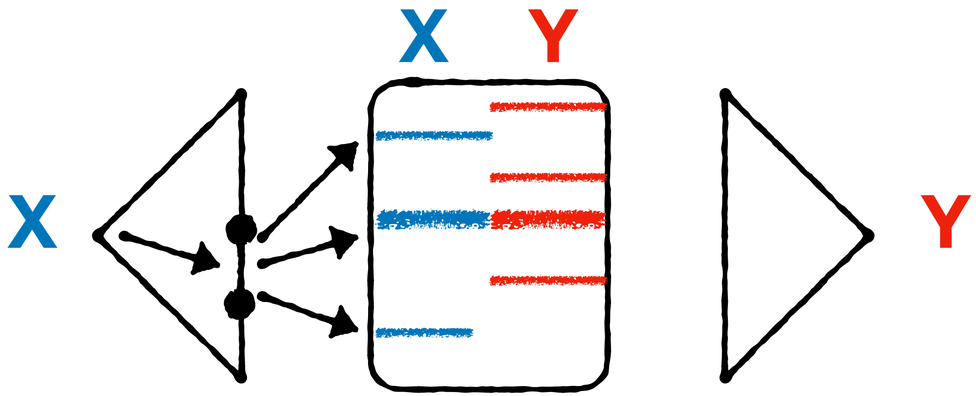


Algorithm 2: Search index X

For each row in table, check if Y matches

Cost: **$\log_B(N) + |X_i|$ I/O**

Select * from … where **X = “i” and Y=“j”**



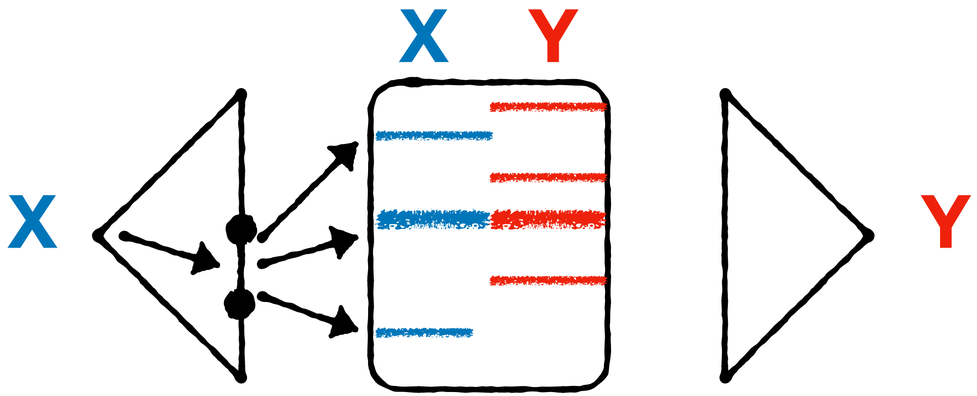
Algorithm 2: Search index X

For each row in table, check if Y matches

Cost: $\log_B(N) + |X_i|$ I/O

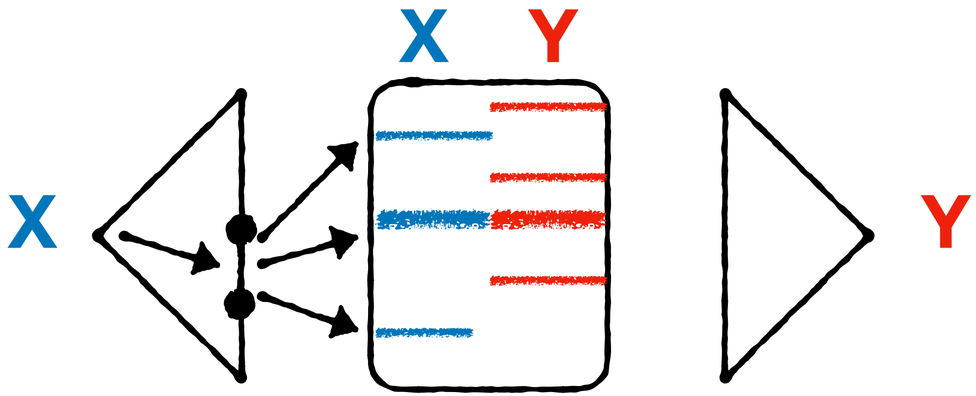
Less expensive since $|X_i| < |Y_j|$ ($|X_i|=3$ and $|Y_j| = 4$)

Select * from … where **X = “i” and Y=“j”**



Principle: filter based on most selective predicate first

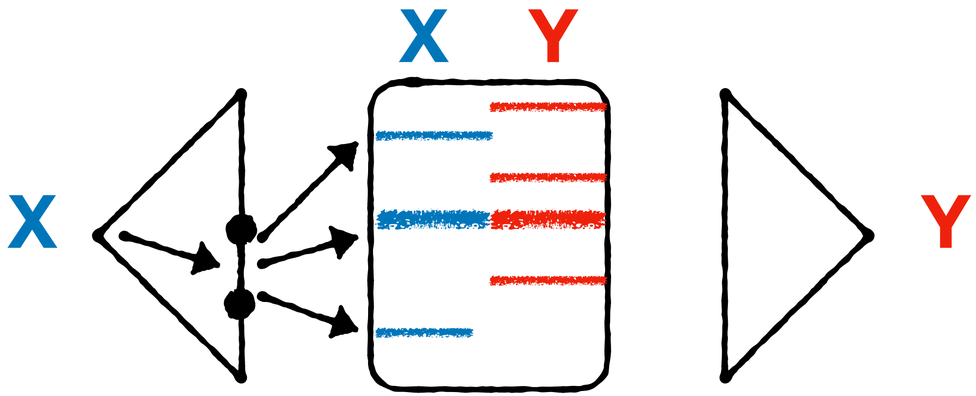
Select * from … where **X = “i” and Y=“j”**



Principle: filter based on most selective predicate first

This requires frequency estimation, e.g., estimating $|X_i|$ or $|Y_j|$

Select * from … where **X = “i” and Y=“j”**

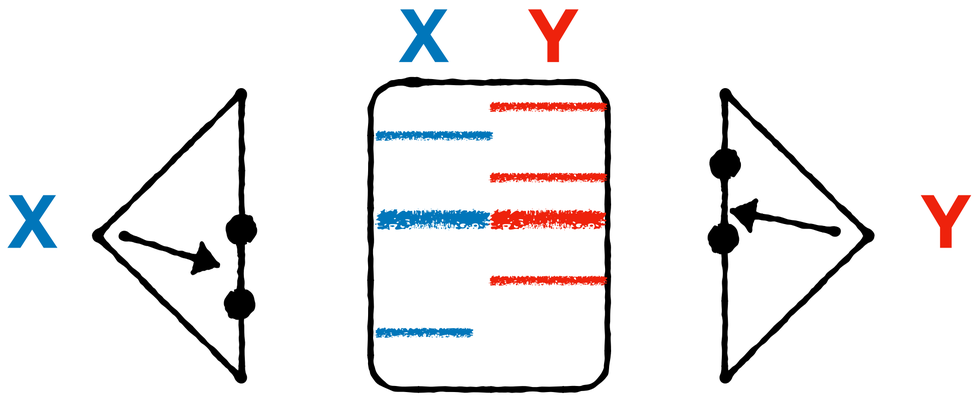


Principle: filter based on most selective predicate first

This requires frequency estimation, e.g., estimating $|X_i|$ or $|Y_j|$

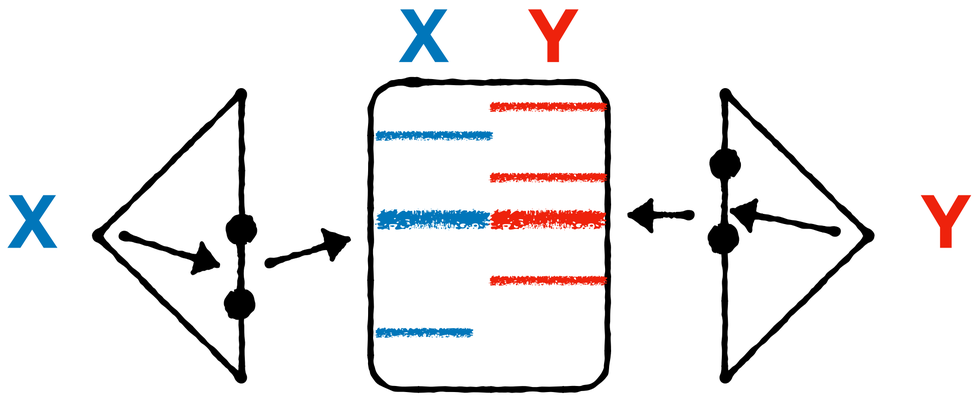
Is there yet another alternative?

Select * from … where **X = “i” and Y=“j”**



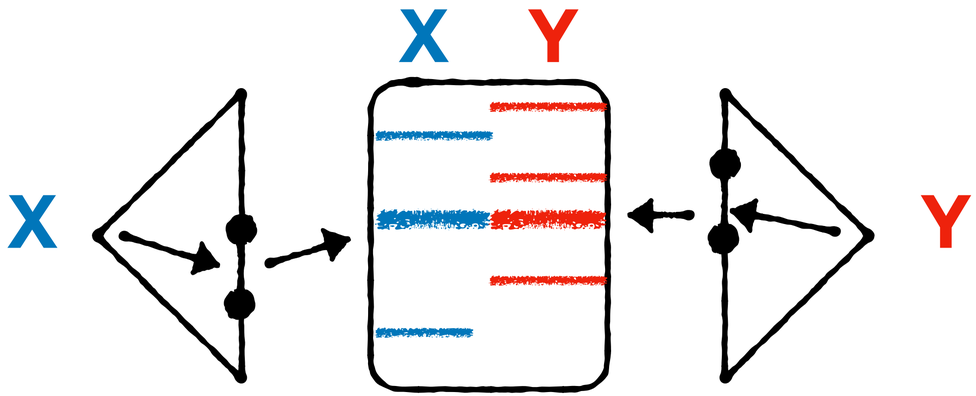
Algorithm 3: Search index A and B for matching row IDs

Select * from … where **X = “i” and Y=“j”**



Algorithm 3: Search index A and B for matching row IDs

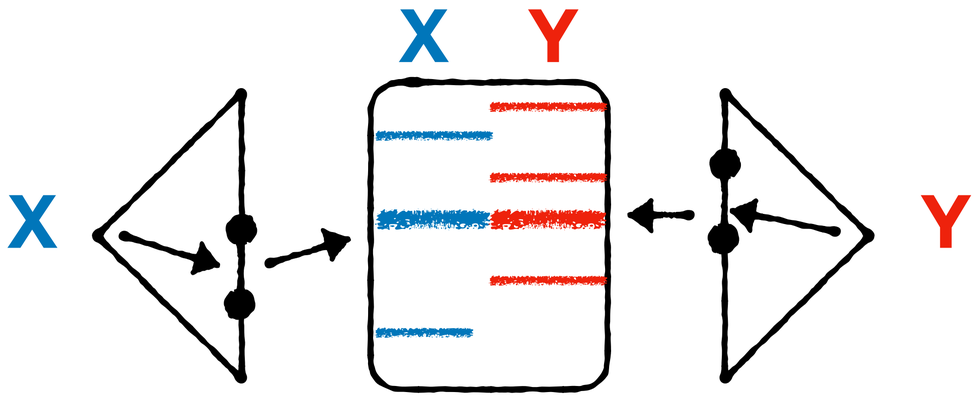
Select * from … where **X = “i” and Y=“j”**



Algorithm 3: Search index A and B for matching row IDs

Access table for intersection

Select * from … where **X = “i” and Y=“j”**

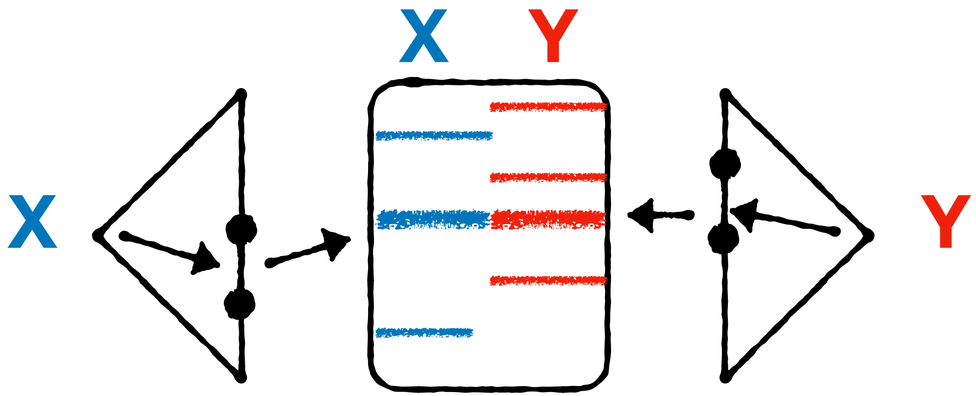


Algorithm 3: Search index A and B for matching row IDs

Access table for intersection

Cost:

Select * from … where **X = “i” and Y=“j”**

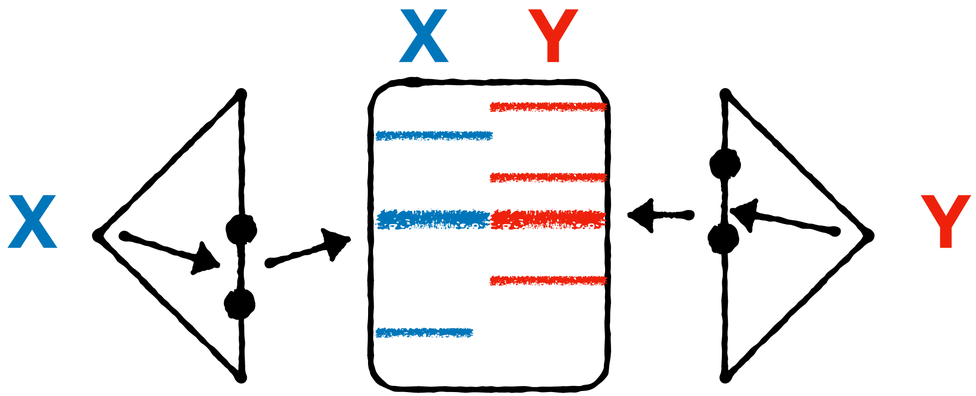


Algorithm 3: Search index A and B for matching row IDs

Access table for intersection

Cost: $\mathbf{2 \cdot \log_B(N) + |X_i|/B + |Y_j|/B + |X_i \cap Y_j|}$ **I/O**

Select * from … where **X = “i” and Y=“j”**



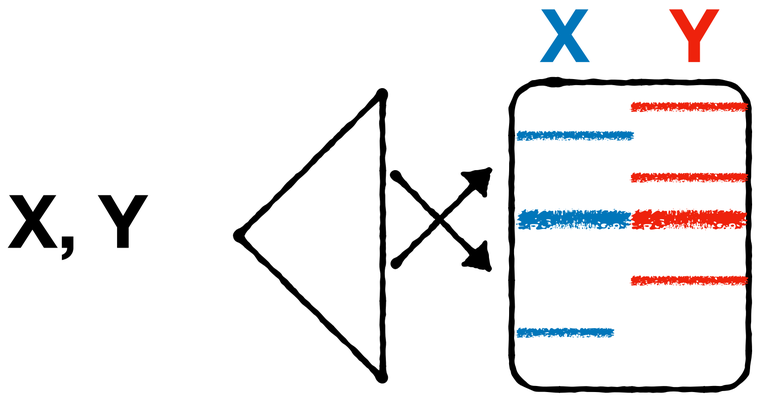
Algorithm 3: Search index A and B for matching row IDs

Only search table for intersection

Cost: $2 \cdot \log_B(N) + |X_i|/B + |Y_j|/B + |X_i \cap Y_j|$ I/O

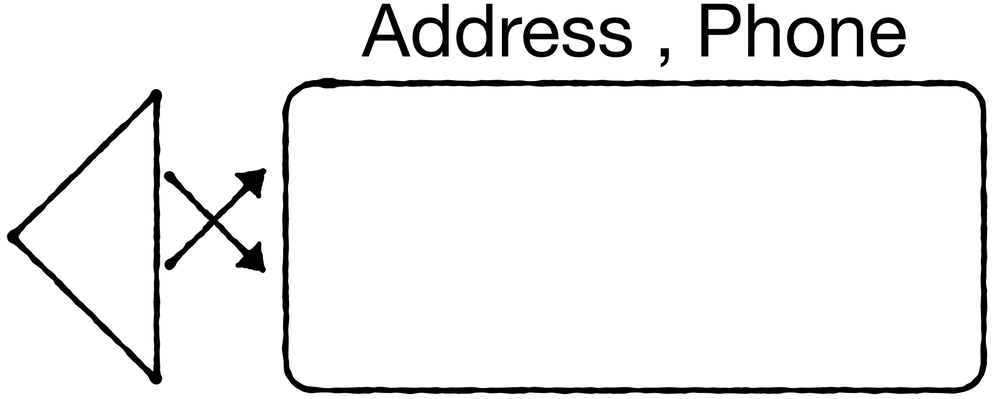
Better if $|X_i \cap Y_j|$ is small relative to $|X_i|$, $|Y_j|$

Select * from … where **X = “i” and Y=“j”**



What if we combine these indexes into a composite index?

Select * from … where **surname = “Lovelace” and firstname=“Ada”**



Surname, first name

…

Long, Justin

Lovelace, Abby

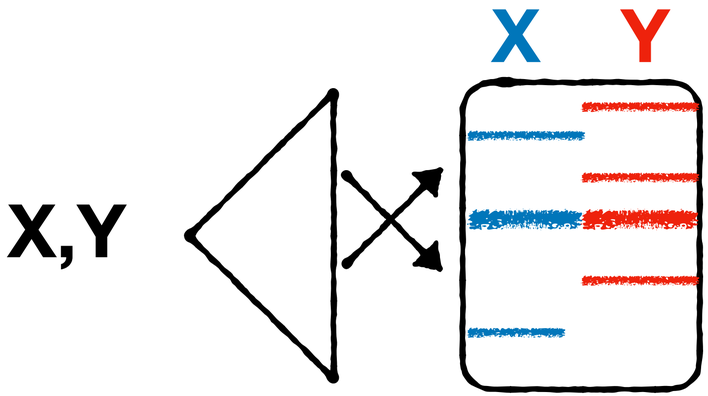
Lovelace, Ada

Lovelace, Adam

Lopez, Jenniffer

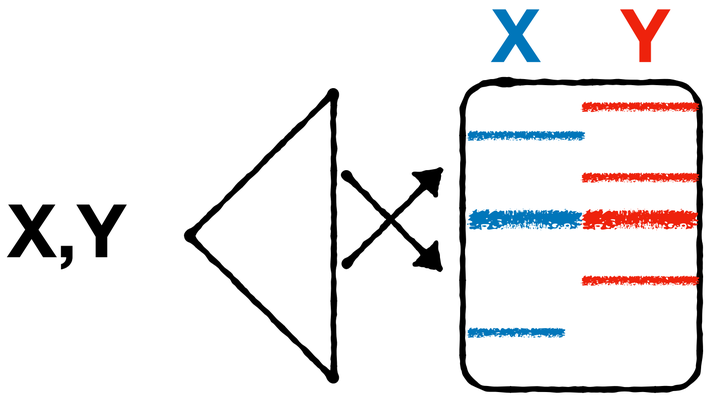
…

Select * from … where **X = “…” and Y=“…”**



Algorithm 4: Search composite index A and B for matching row IDs

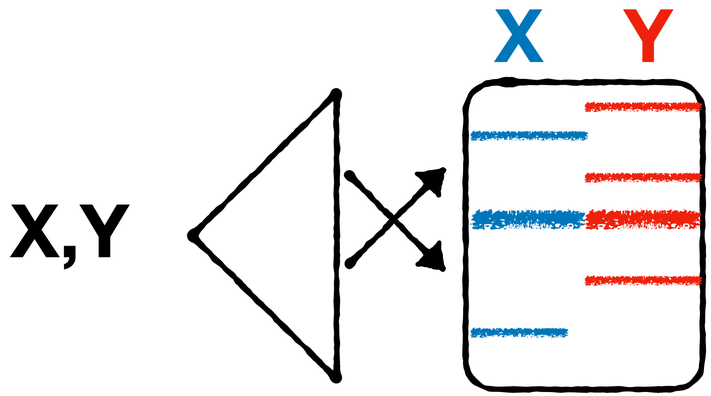
Select * from … where **X = “…” and Y=“…”**



Algorithm 4: Search composite index A and B for matching row IDs

Cost: $\log_B(N) + |X_i \cap Y_j|$ **I/O**

Select * from … where **X = “…” and Y=“…”**

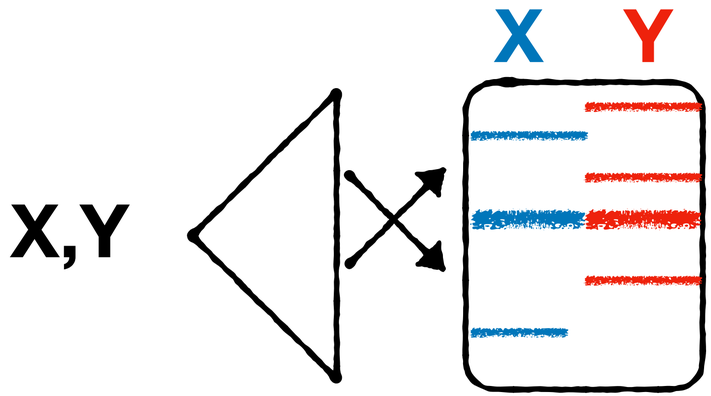


Algorithm 4: Search composite index A and B for matching row IDs

Cost: $\log_B(N) + |X_i \cap Y_j|$ I/O

Cheapest option so far!

Select * from … where **X = “…” and Y=“…”**



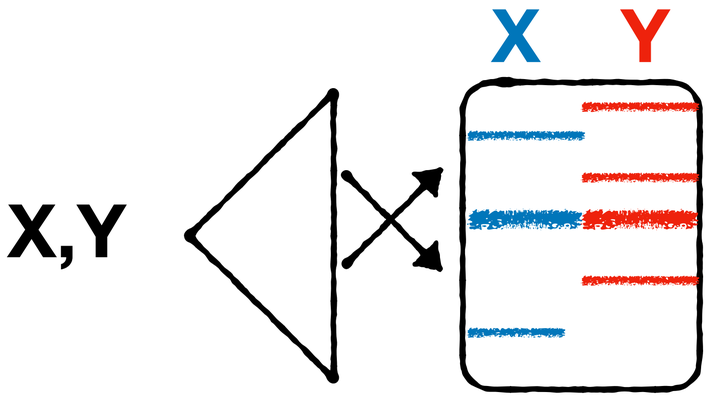
Algorithm 4: Search composite index A and B for matching row IDs

Cost: $\log_B(N) + |X_i \cap Y_j|$ I/O

Cheapest option so far!

Downsides?

Select * from … where **Y=“…”**



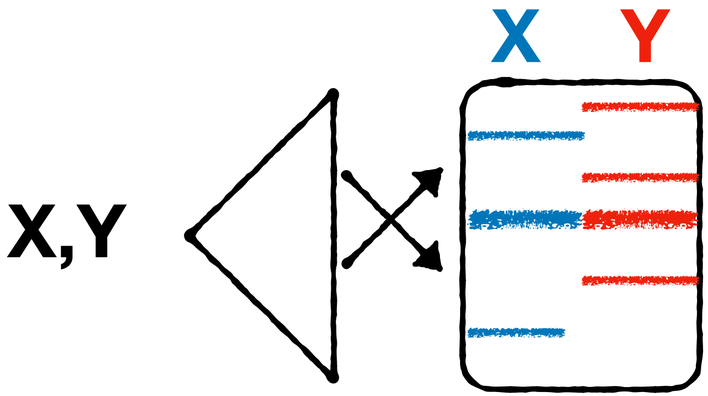
Algorithm 4: Search composite index A and B for matching row IDs

Cost: $\log_B(N) + |X_i \cap Y_j|$ I/O

Cheapest option so far!

Downsides? Cannot handle queries just based on Y

Select * from … where **Y=“…”**



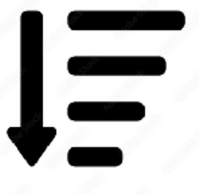
Composite indexes can achieve better performance for some queries but are less generic. Choose them carefully :)

Selection

Projection

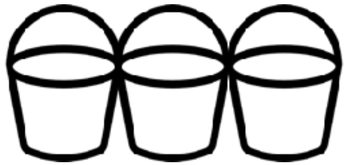
Join

Order by



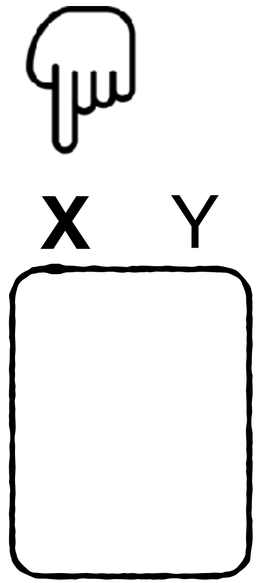
Distinct

Group by



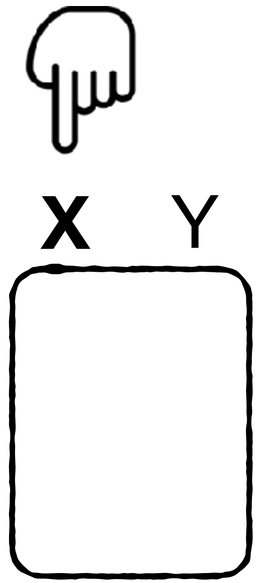
Projection

Select X from table



Projection

Select X from table



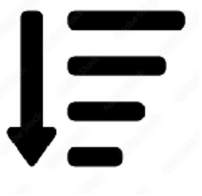
Trivial in row-stores. More interesting in column-stores (next week).

Selection

Projection

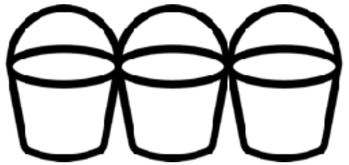
Join

Order By



Distinct

Group by

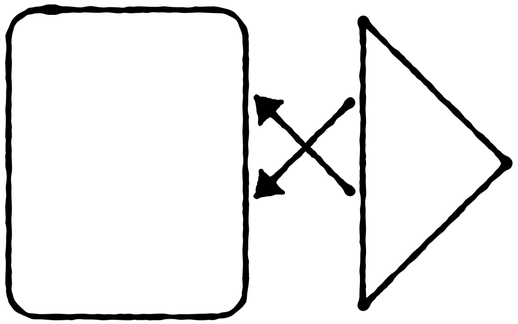


Select … From … where … **Order By** col1, col2, ... **ASC | DESC**

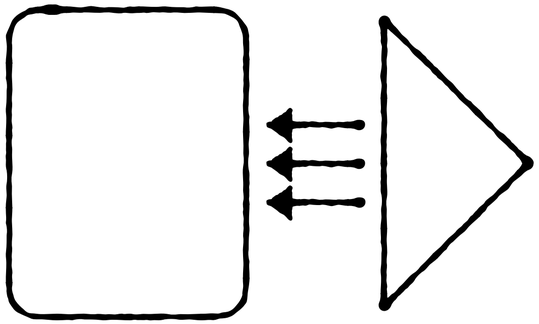
Select … From … where … Order By col1, col2, ... ASC | DESC

If we have an applicable index

Unclustered index scan



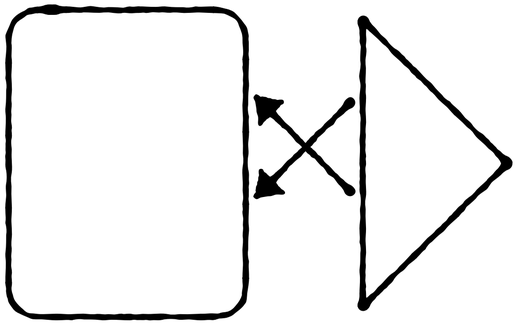
Clustered index scan



Select … From … where … Order By col1, col2, ... ASC | DESC

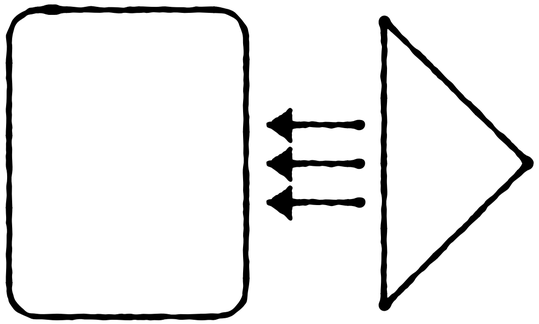
If we have an applicable index

Unclustered
index scan



(Good if result set is small)

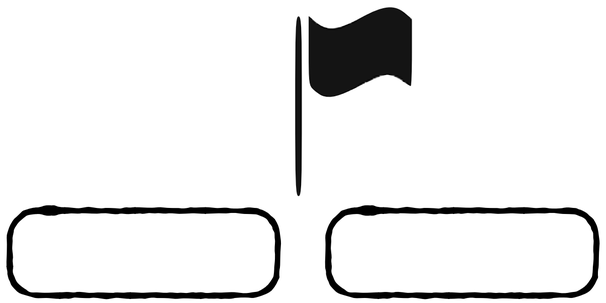
Clustered
index scan



(Good in all cases)

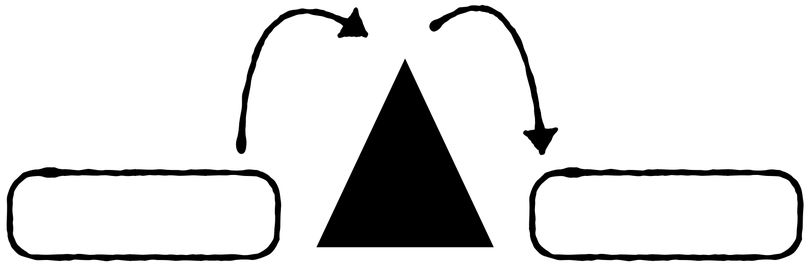
Select … From … where … Order By col1, col2, ... ASC | DESC

Quick-Sort



If result set fits in memory

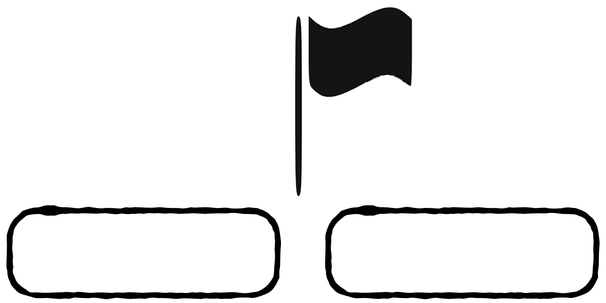
Heap-Sort



If quick-sort takes longer than expected

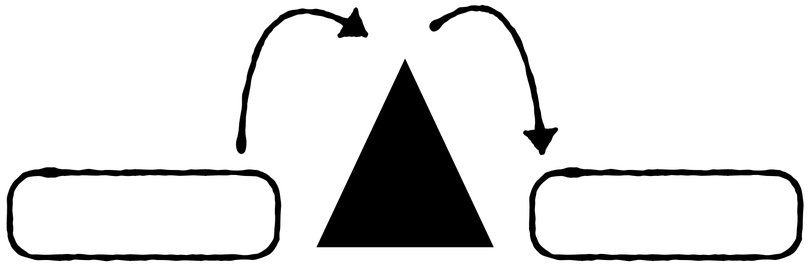
Select … From … where … Order By col1, col2, ... ASC | DESC

Quick-Sort



If result set fits in memory

Heap-Sort

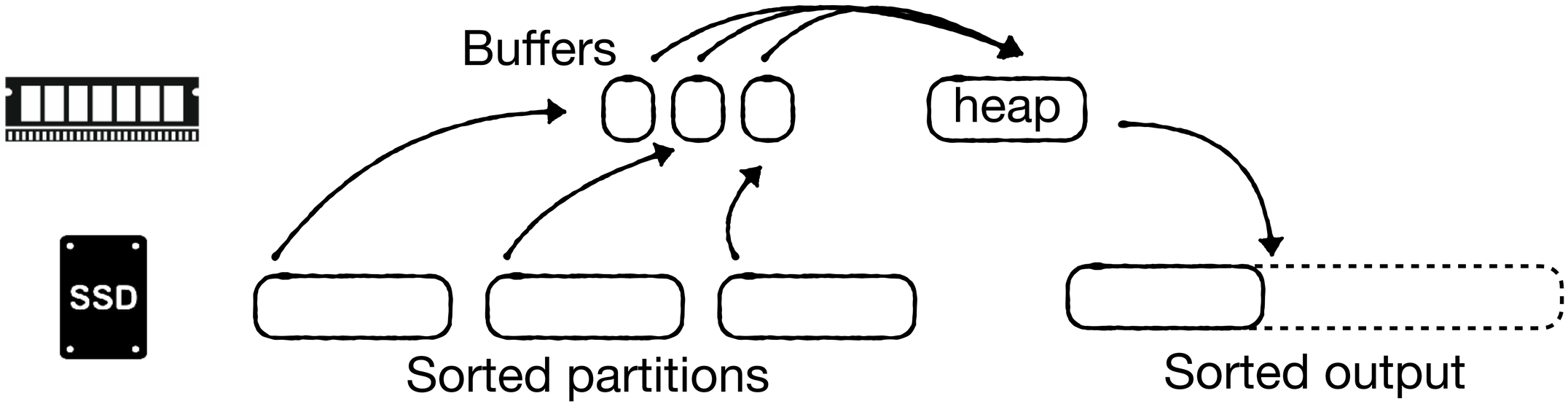


If quick-sort takes longer than expected

Introsort :)

Select … From … where … Order By col1, col2, ... ASC | DESC

Or external sort if data does not fit in memory

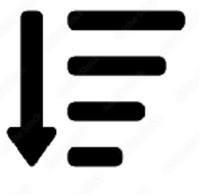


Selection

Projection

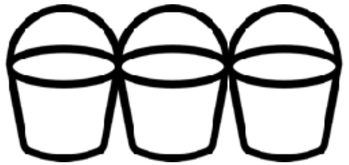
Join

Order By



Distinct

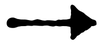
Group by



Select **Distinct** c1, c2, ... FROM …;

Select **Distinct** address FROM people;

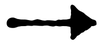
Name	Address
Bert	48 1st St.
Ernie	48 1st St.
	...



Address
48 1st St.
...

Select **Distinct** address FROM people;

Name	Address
Bert	48 1st St.
Ernie	48 1st St.
	...



Address
48 1st St.
...

How to implement?

Select **Distinct** c1, c2, ... FROM …;

Sort & eliminate adjacent identical items

Select **Distinct** c1, c2, ... FROM …;

Sort & eliminate adjacent identical items

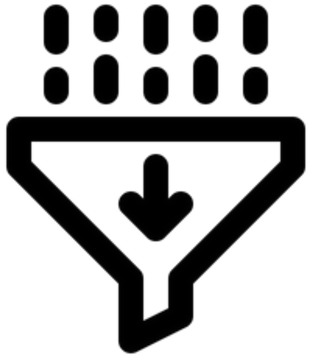
Quick-Sort
Heap-Sort
External Sort
Index scan
Etc.

Select **Distinct** c1, c2, ... FROM …;

Sort & eliminate adjacent identical items

Quick-Sort
Heap-Sort
External Sort
Index scan
Etc.

Hash to identify identical items

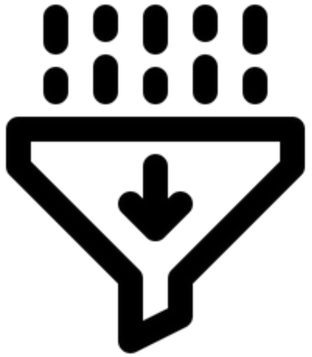


Select **Distinct** c1, c2, ... FROM …;

Sort & eliminate adjacent identical items

Quick-Sort
Heap-Sort
External Sort
Index scan
Etc.

Hash to identify identical items



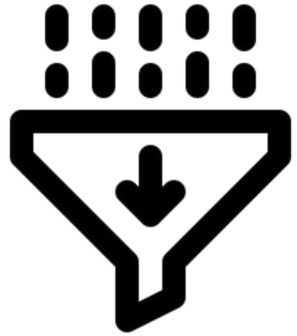
Linear probing
Chaining
Extendible hashing
Cuckoo hashing
Etc.

Select Distinct c1, c2, ... FROM …;

Sort & eliminate adjacent identical items

CPU: $O(N \log_2 N)$

Hash to identify identical items



O(N)

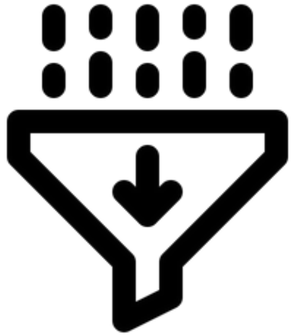
Select **Distinct** c1, c2, ... FROM … **order by** c1, c2 …;

Sort & eliminate adjacent identical items

CPU: $O(N \log_2 N)$

Better if we later need to sort anyways

Hash to identify identical items



O(N)

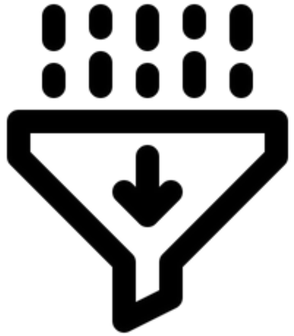
Select **Distinct** c1, c2, ... FROM …;

Sort & eliminate adjacent identical items

CPU: O(N $\log_2$ N)

Better if we later need to sort anyways

Hash to identify identical items



O(N)

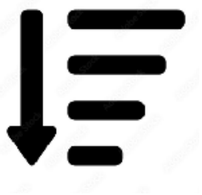
Good if user is fine with unordered output

Selection

Projection

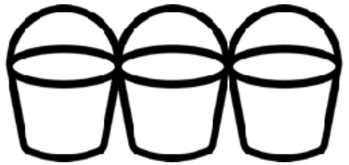
Join

Order by



Distinct

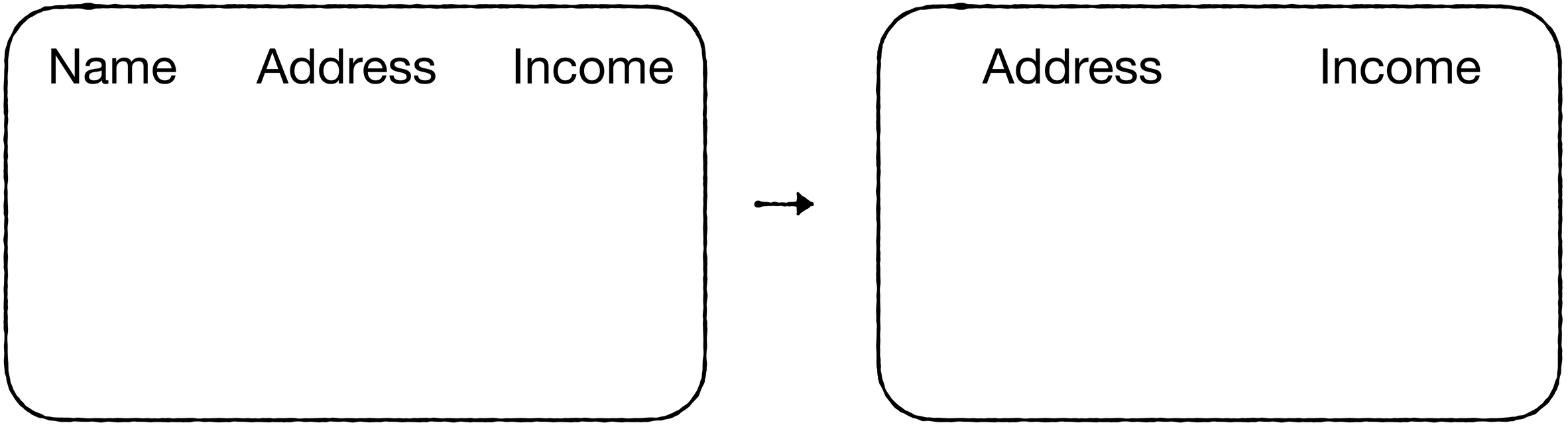
Group By



Select c1, c2, ... FROM … **Group By**

Select address, sum(income) FROM people **Group By** address

(Income per household)



Select address, sum(income) FROM people **Group By** address

(Income per household)

Name	Address	Income
Bert	48 1st St.	**100K**
Ernie	48 1st St.	**100K**

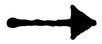
→

Address	Income
48 1st St.	**200K**

Select address, sum(income) FROM people **Group By** address

(Income per household)

Name	Address	Income
Bert	48 1st St.	**100K**
Ernie	48 1st St.	**100K**



Address	Income
48 1st St.	**200K**

How to execute?

Select address, sum(income) FROM people Group By address

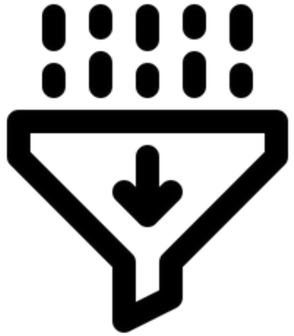
Index/Sort by Category

e.g., B-tree (address -> sum(income))

Select address, sum(income) FROM people Group By address

Index/Sort by Category

Hash by Category



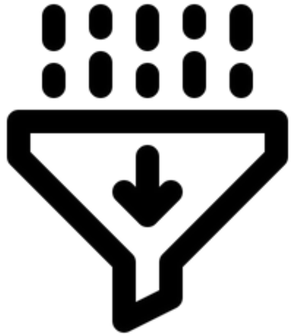
e.g., also (address -> sum(income))

Select c1, c2, ... FROM … Group By

Index/Sort by Category

CPU: $O(N \log_2 N)$

Hash by Category



O(N)

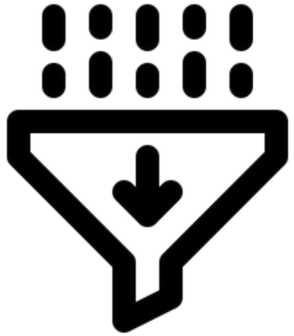
Select c1, c2, ... FROM … Group By

Index/Sort by Category

CPU: $O(N \log_2 N)$

Better if we later need to sort anyways

Hash by Category



$O(N)$

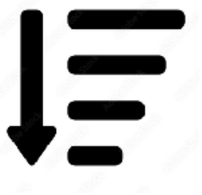
Good if user is fine with unordered output

Selection

Projection

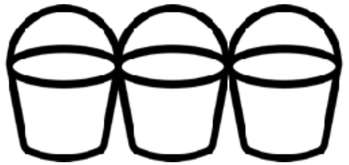
Join

Order by

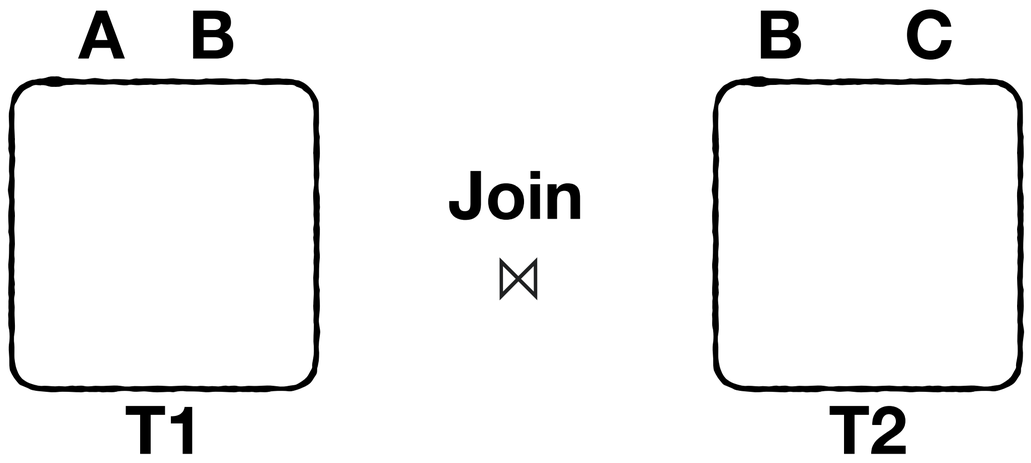


Distinct

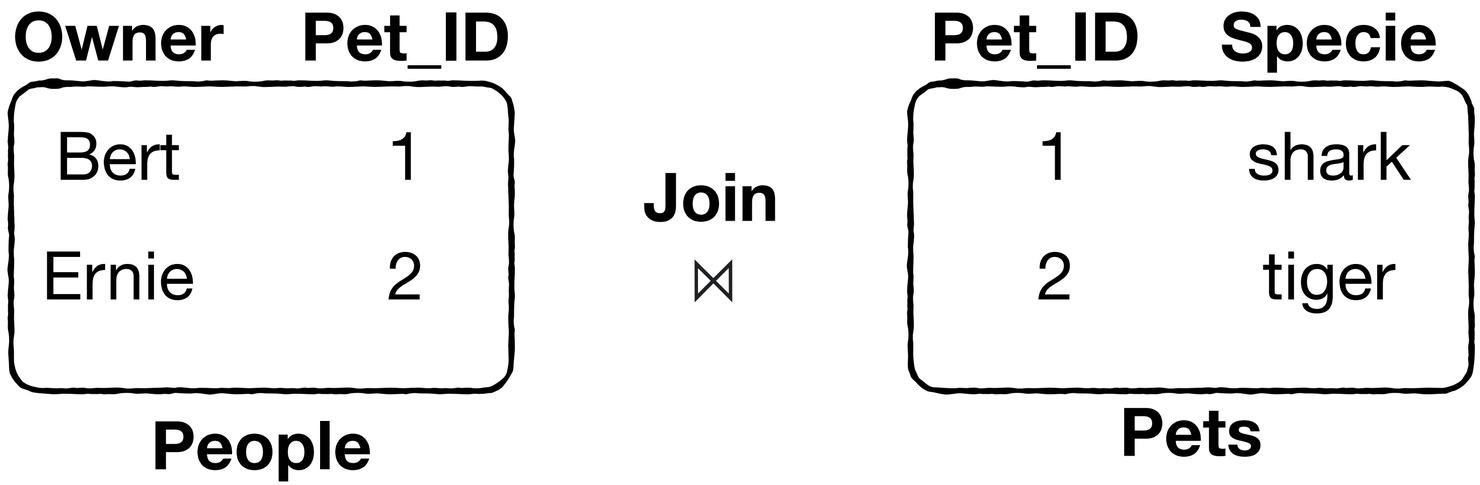
Group By



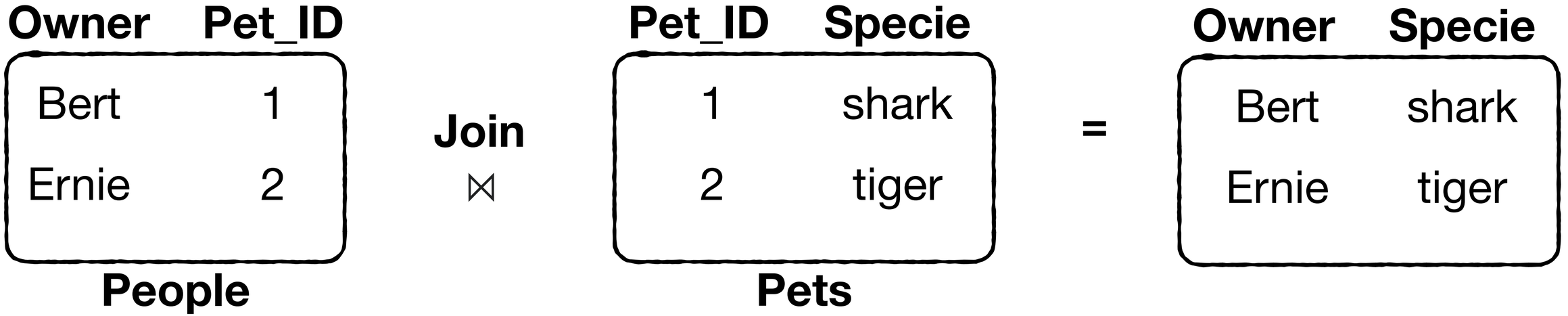
Join: Select … FROM T1, T2 **where T1.B = T2=B**



Select **Owner**, **Specie** FROM People, Pets **where T1.Pet_ID = T2=Pet_ID**



Select **Owner**, **Specie** FROM People, Pets **where T1.Pet_ID = T2=Pet_ID**



Join Algorithms

Nested Loop

Block Nested Loop

Index-Join

Sort-merge Join

Grace hash Join

Join Algorithms

Nested Loop

Straw man

Block Nested Loop

Index-Join

Sort-merge Join

Grace hash Join

Join Algorithms

Nested Loop

Block Nested Loop

Index-Join

Sort-merge Join

Grace hash Join

Good when one relation almost or entirely fits in memory

Join Algorithms

Nested Loop

Block Nested Loop

Index-Join

Sort-merge Join

Grace hash Join

There are index/es on the join keys

Join Algorithms

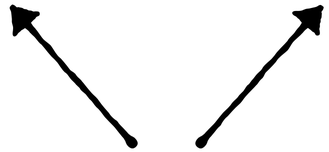
Nested Loop

Block Nested Loop

Index-Join

Sort-merge Join

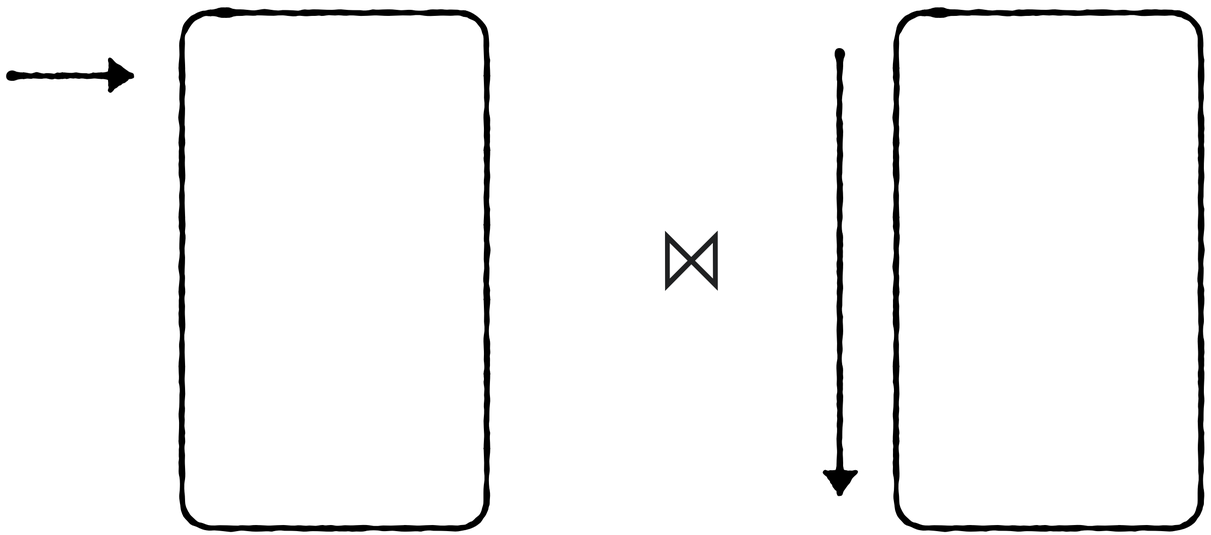
Grace hash Join



Both relations must bigger than memory

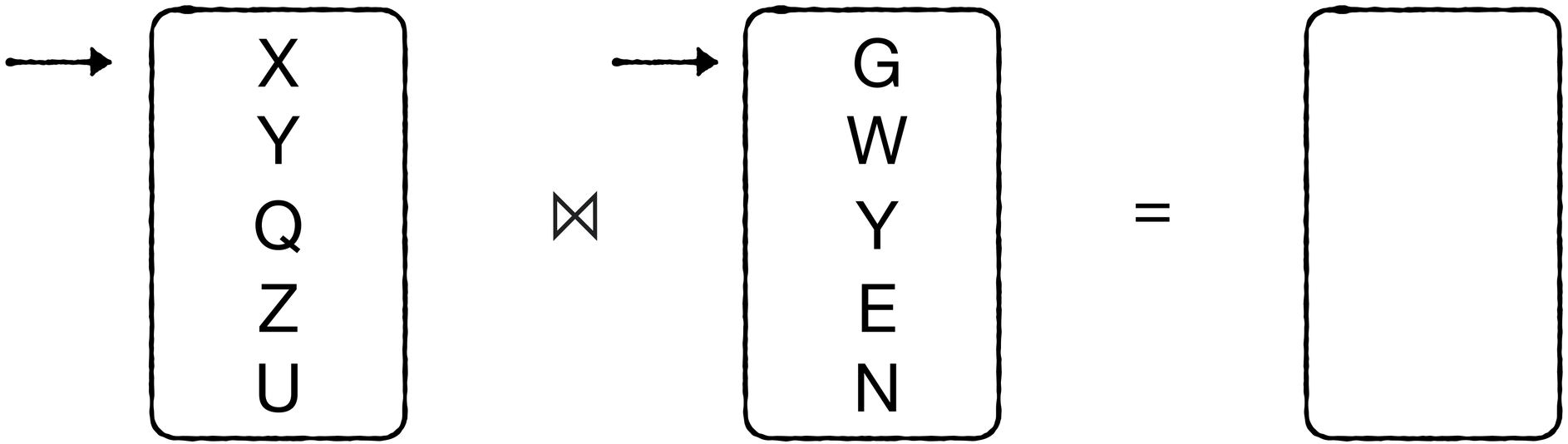
Nested Loop

For each entry in one relation, scan whole other relation



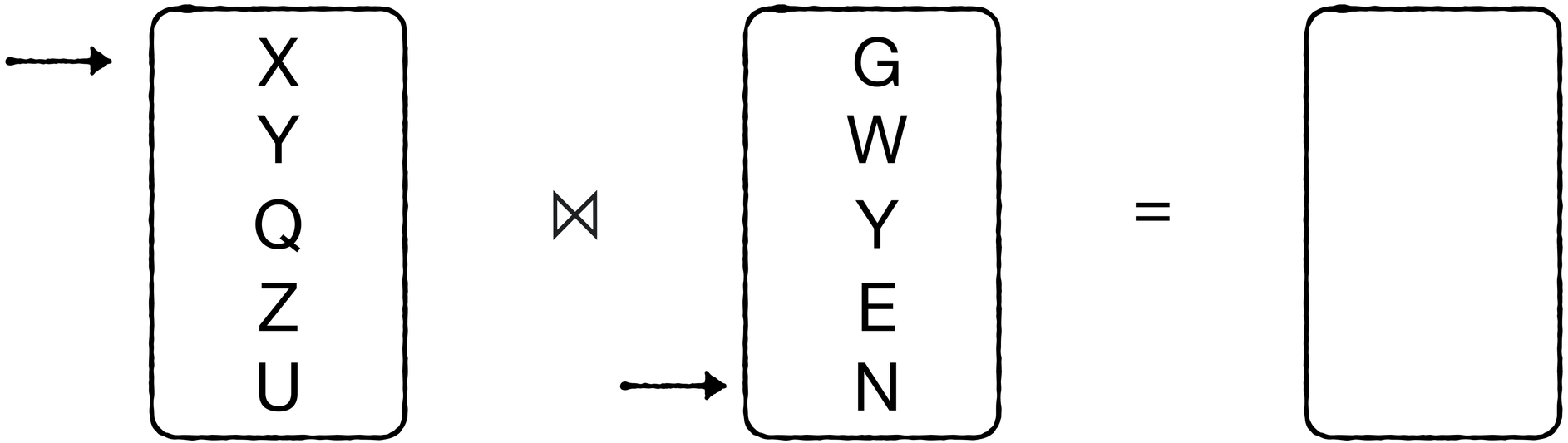
Nested Loop

For each entry in one relation, scan whole other relation



Nested Loop

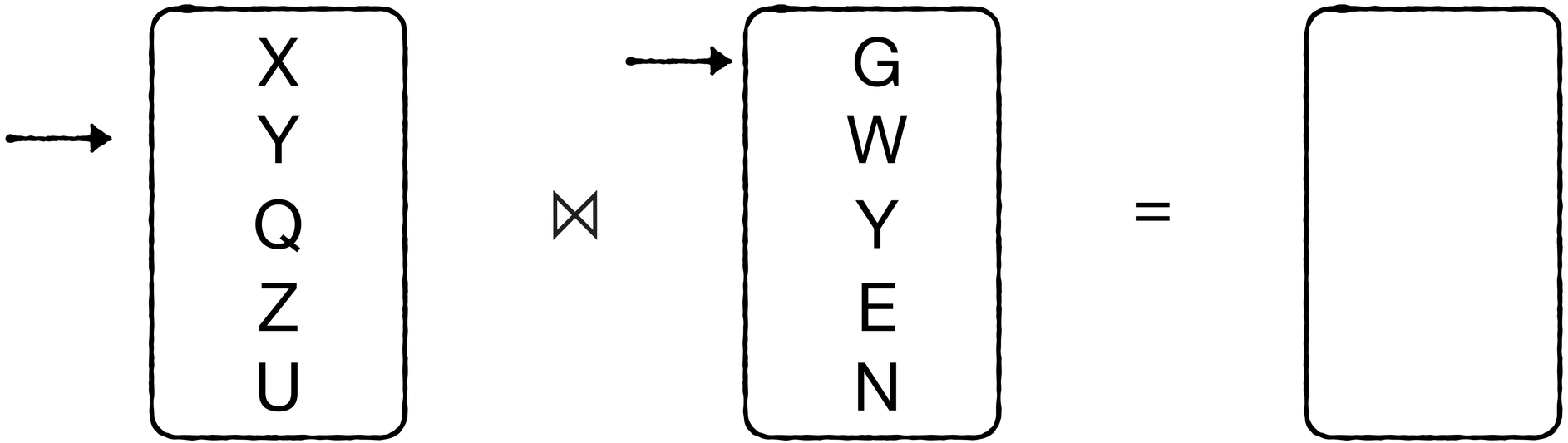
For each entry in one relation, scan whole other relation



No match for X

Nested Loop

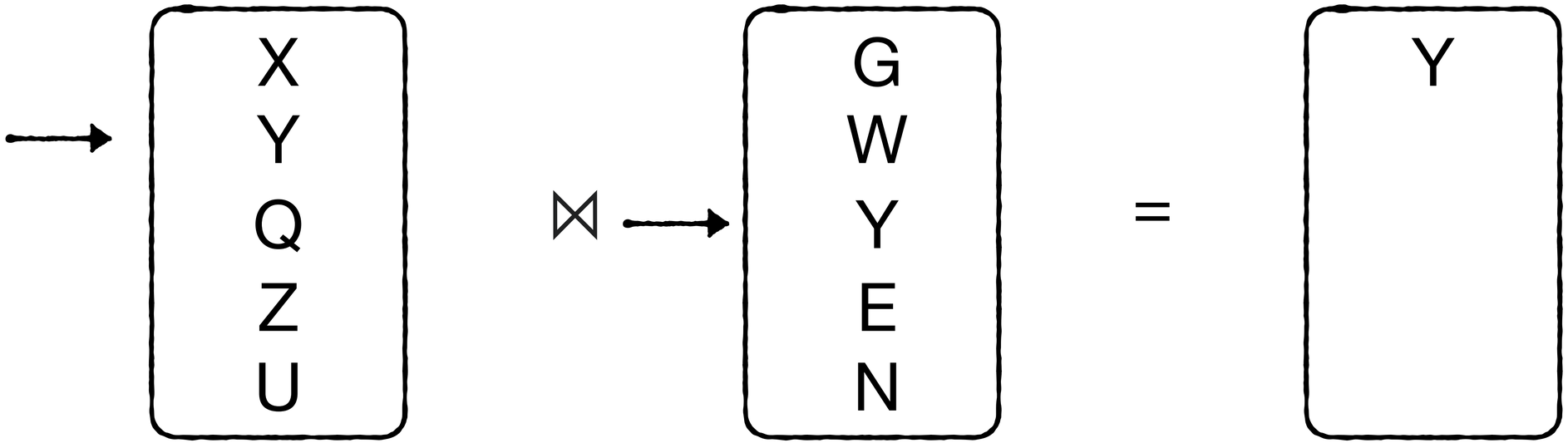
For each entry in one relation, scan whole other relation



No match for X

Nested Loop

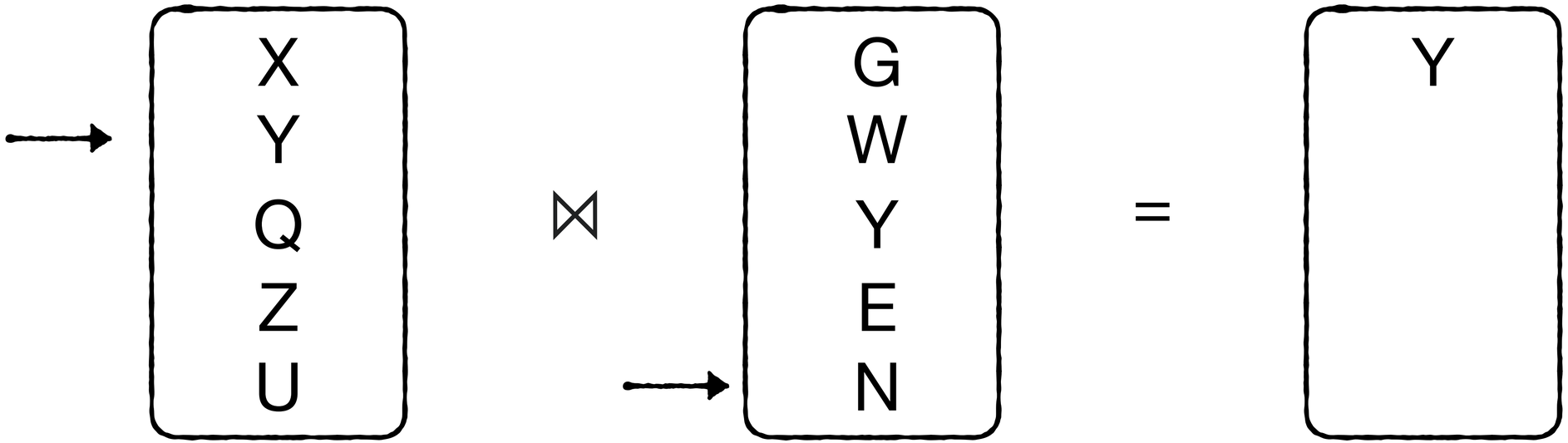
For each entry in one relation, scan whole other relation



Match for Y

Nested Loop

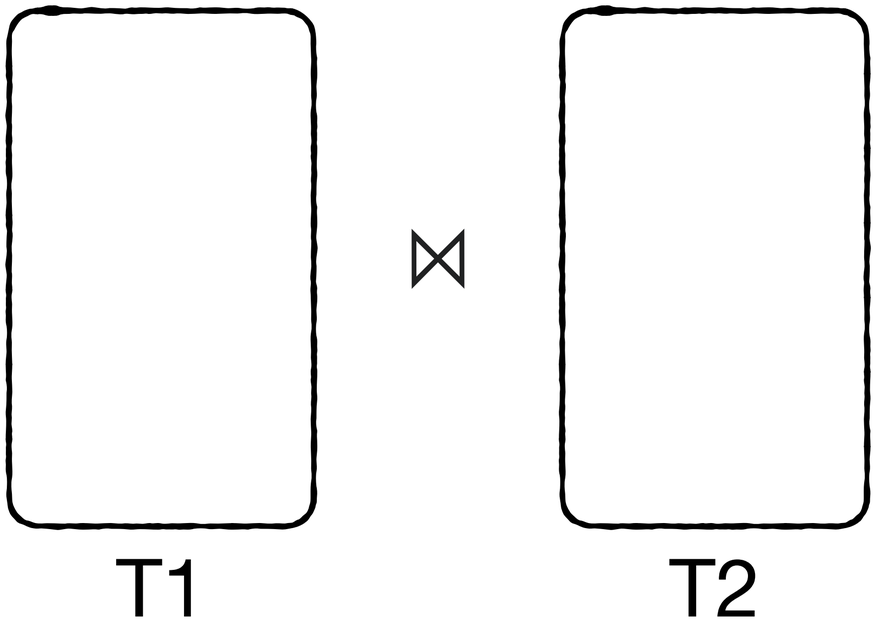
For each entry in one relation, scan whole other relation



No more matches for Y

Nested Loop

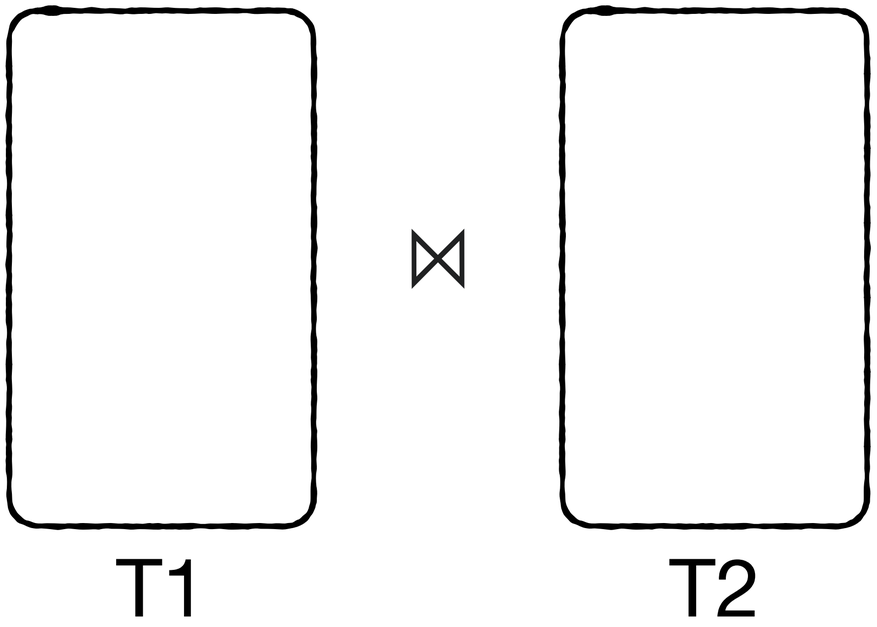
For each entry in one relation, scan whole other relation



Cost?

Nested Loop

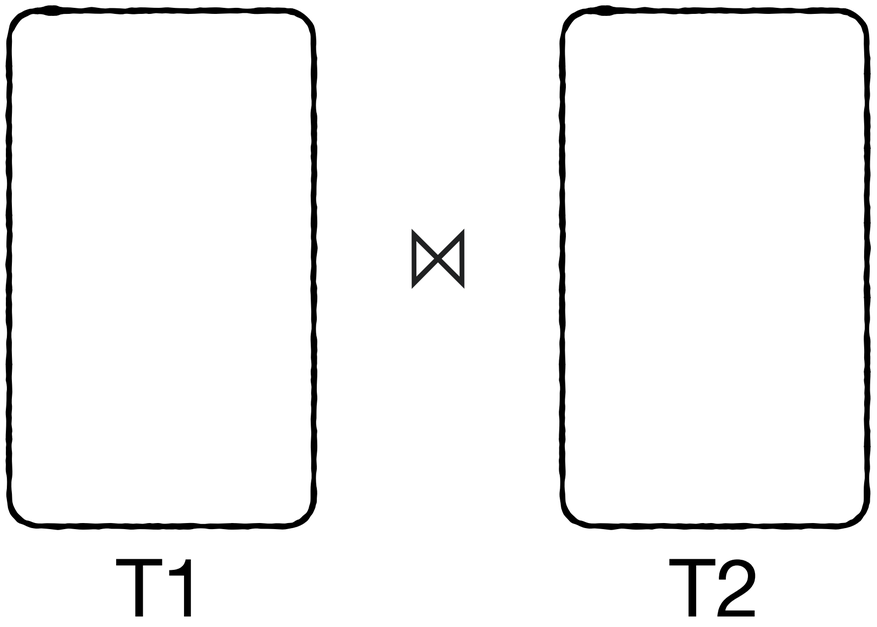
For each entry in one relation, scan whole other relation



Cost: O(|T1| · |T2|/B) I/O

Nested Loop

For each entry in one relation, scan whole other relation

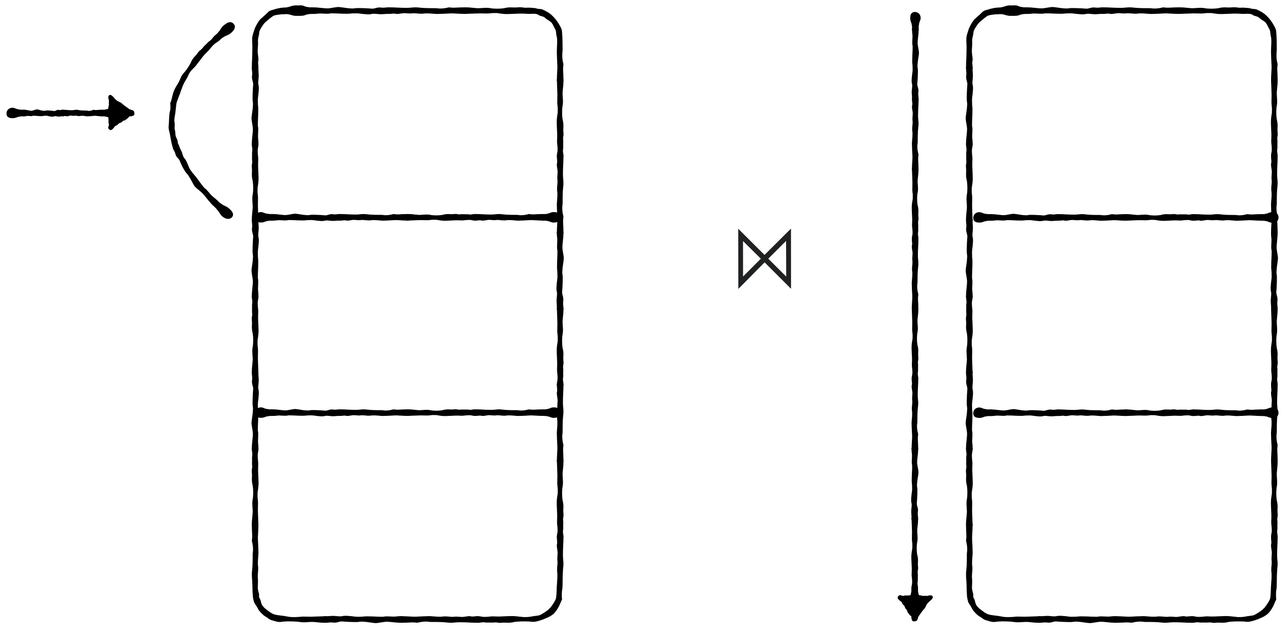


Cost: $O(|T1| \cdot |T2|/B)$ I/O

What's a simple improvement?

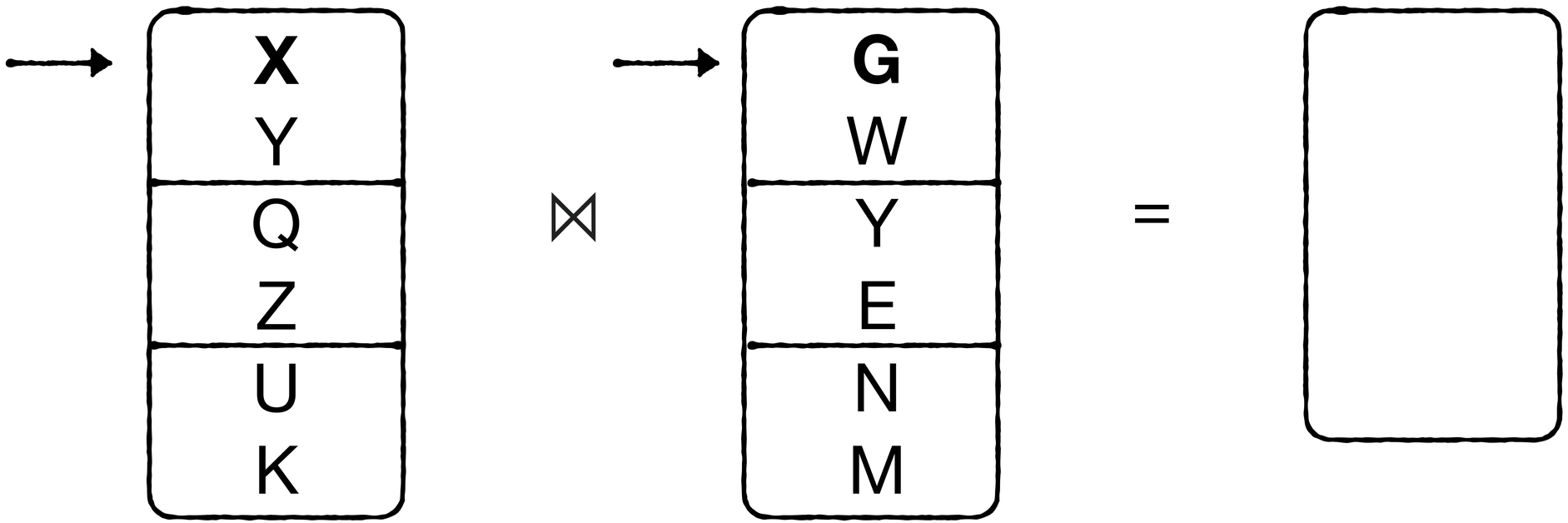
Block Nested Loop

For each **block** in one relation, scan whole other relation



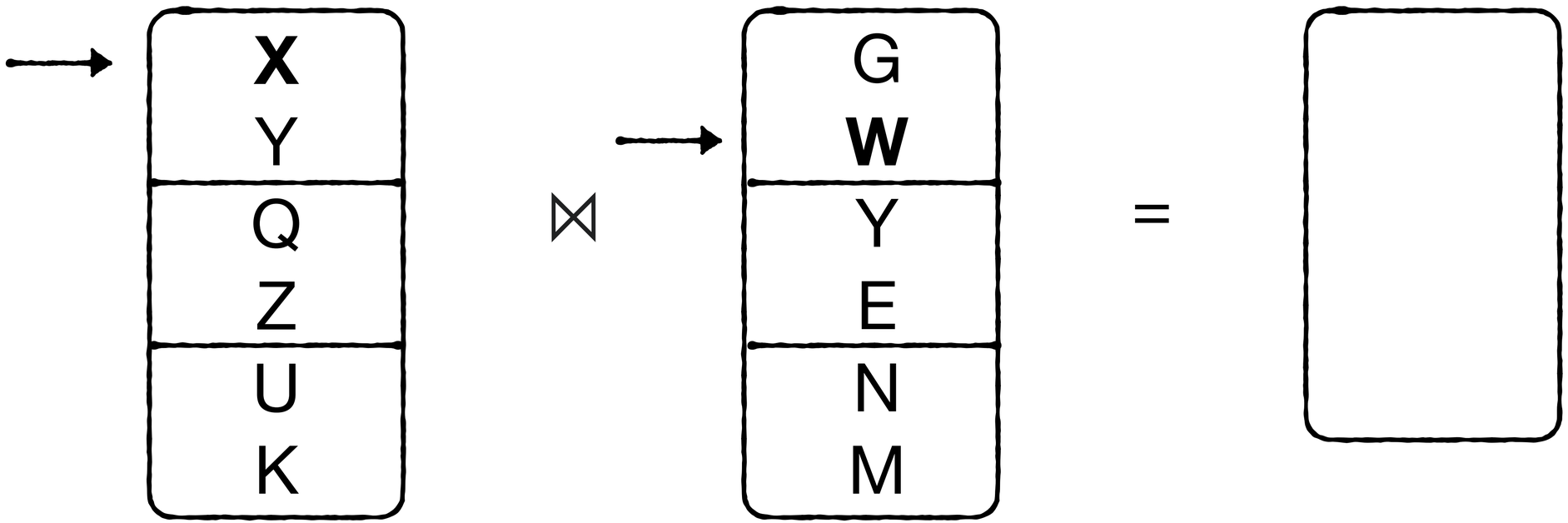
Block Nested Loop

For each block in one relation, scan whole other relation



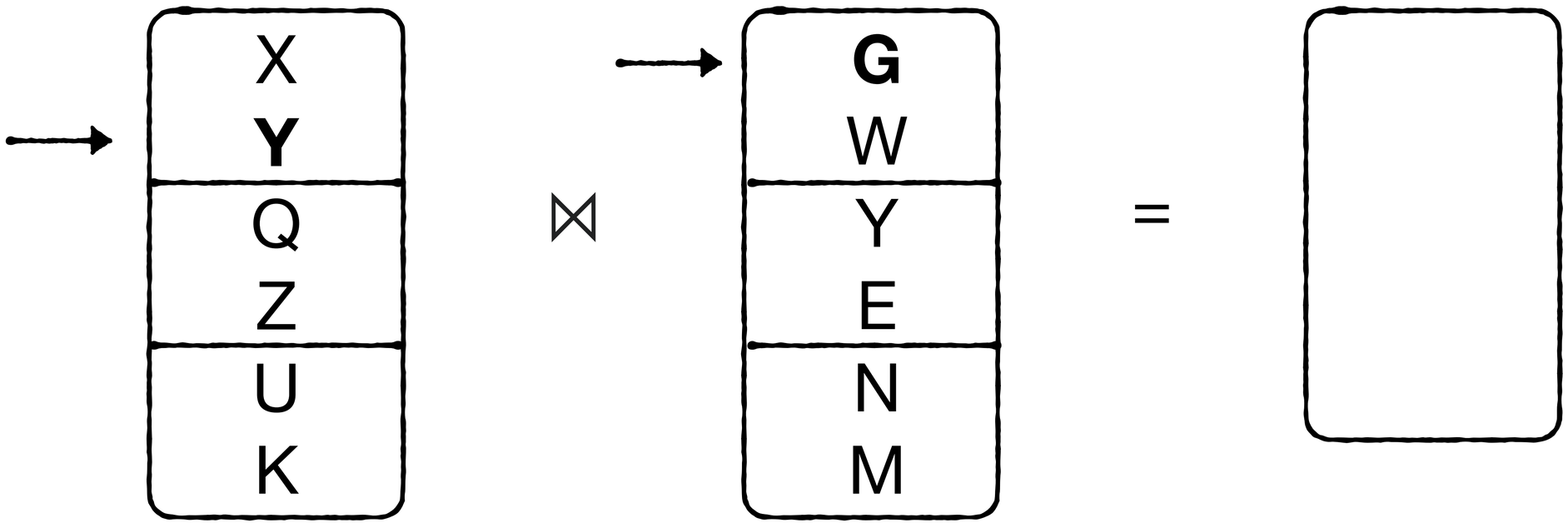
Block Nested Loop

For each block in one relation, scan whole other relation



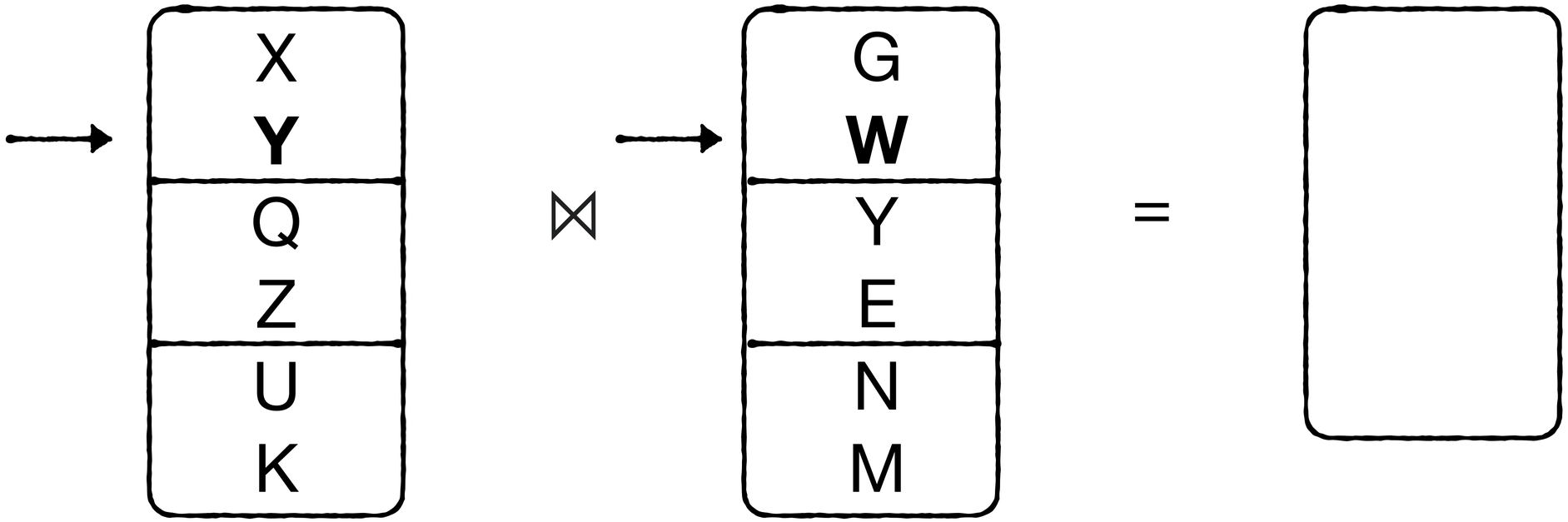
Block Nested Loop

For each block in one relation, scan whole other relation



Block Nested Loop

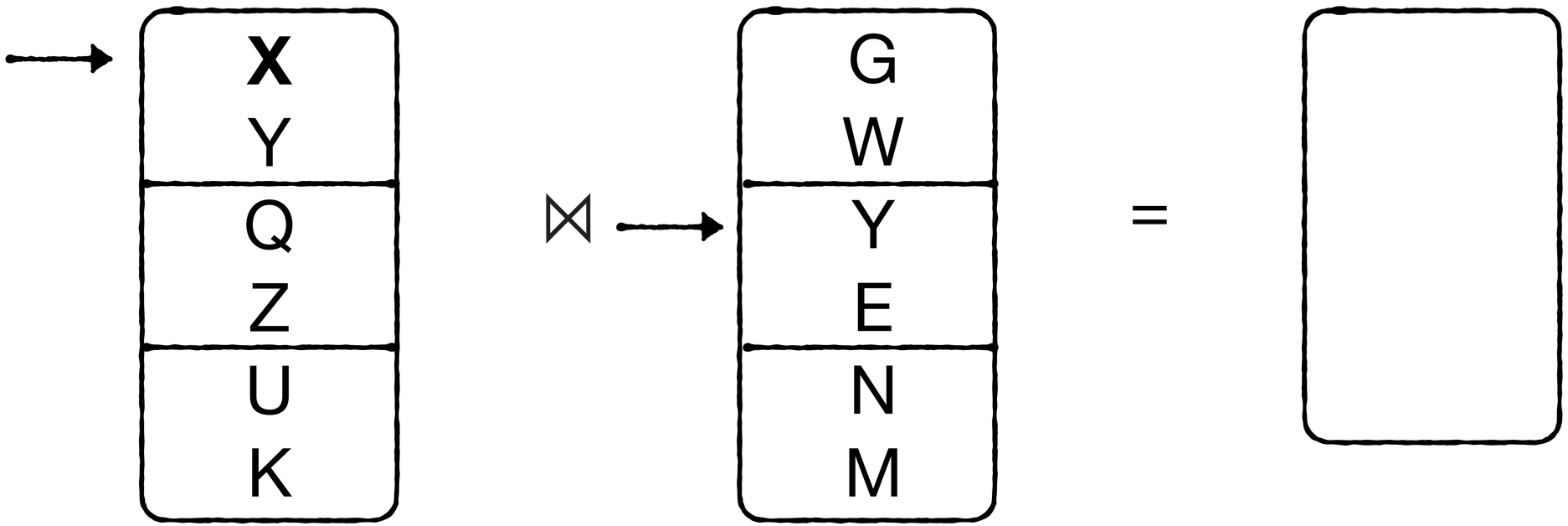
For each block in one relation, scan whole other relation



No match in first block

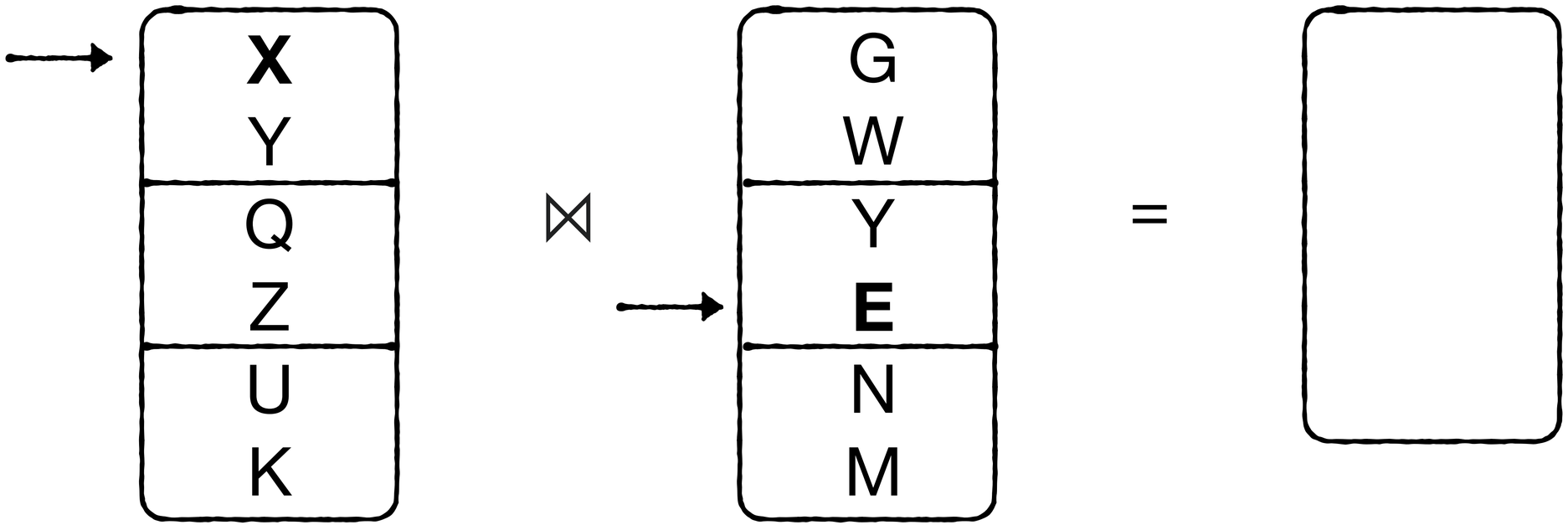
Block Nested Loop

For each block in one relation, scan whole other relation



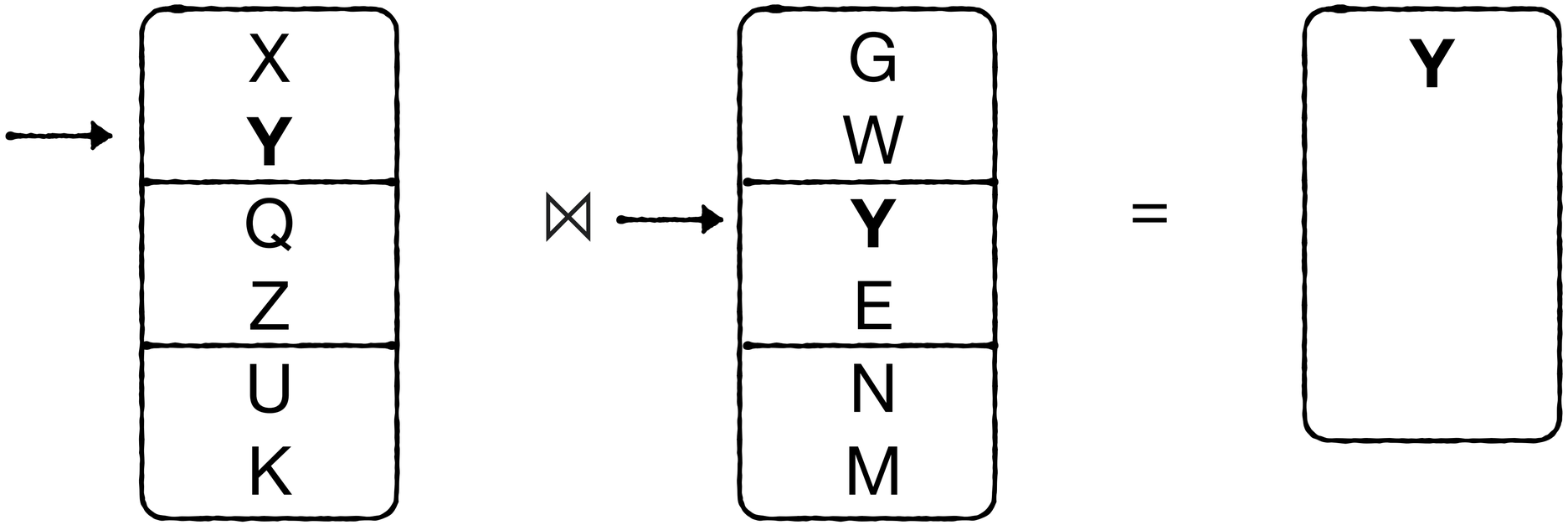
Block Nested Loop

For each block in one relation, scan whole other relation



Block Nested Loop

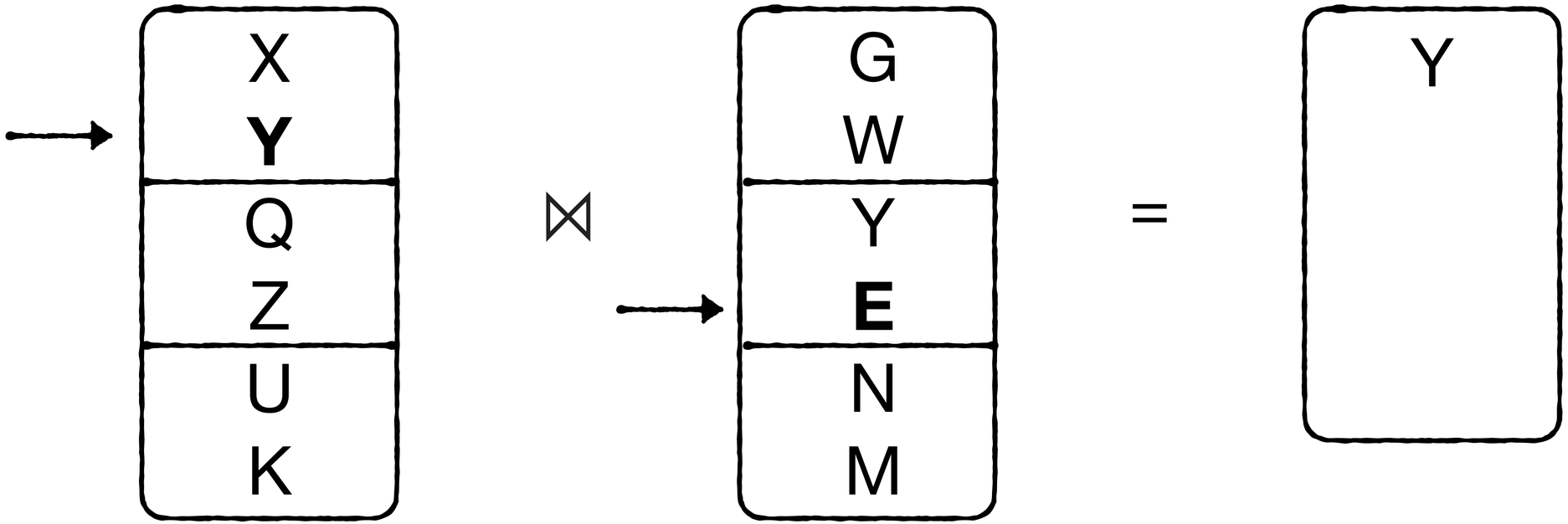
For each block in one relation, scan whole other relation



Match!

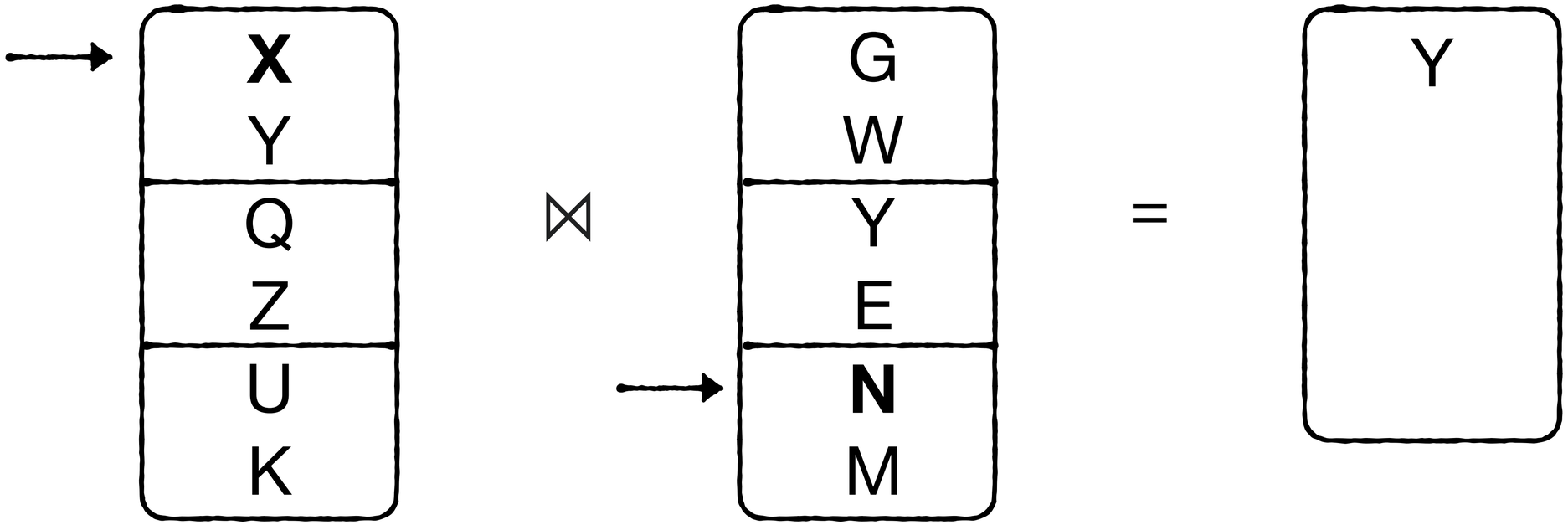
Block Nested Loop

For each block in one relation, scan whole other relation



Block Nested Loop

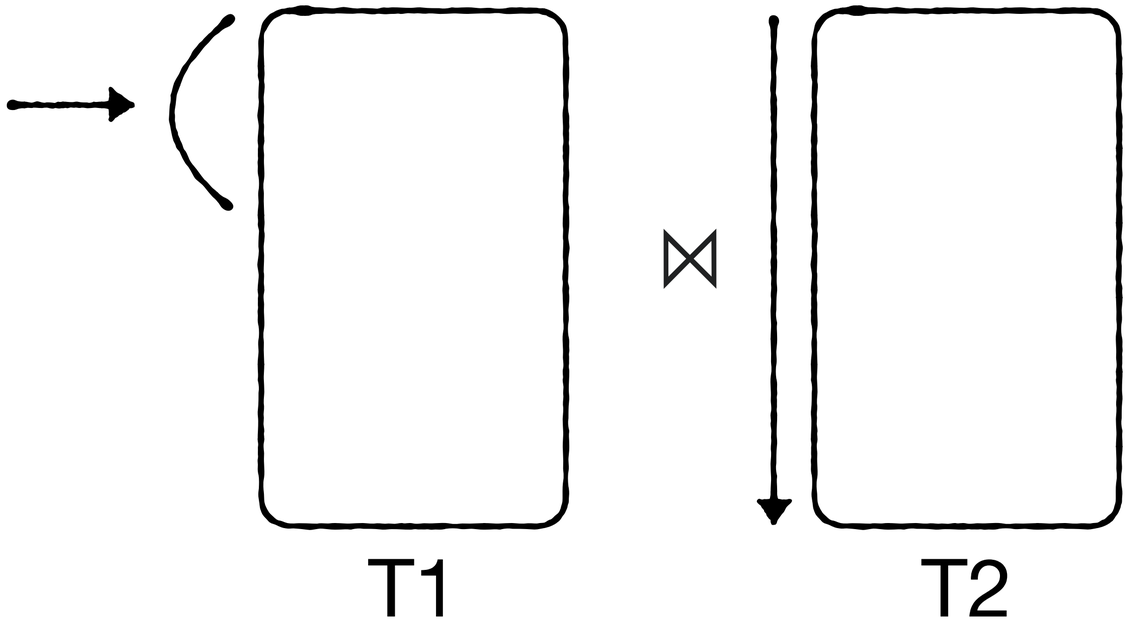
For each block in one relation, scan whole other relation



And so on…

Block Nested Loop

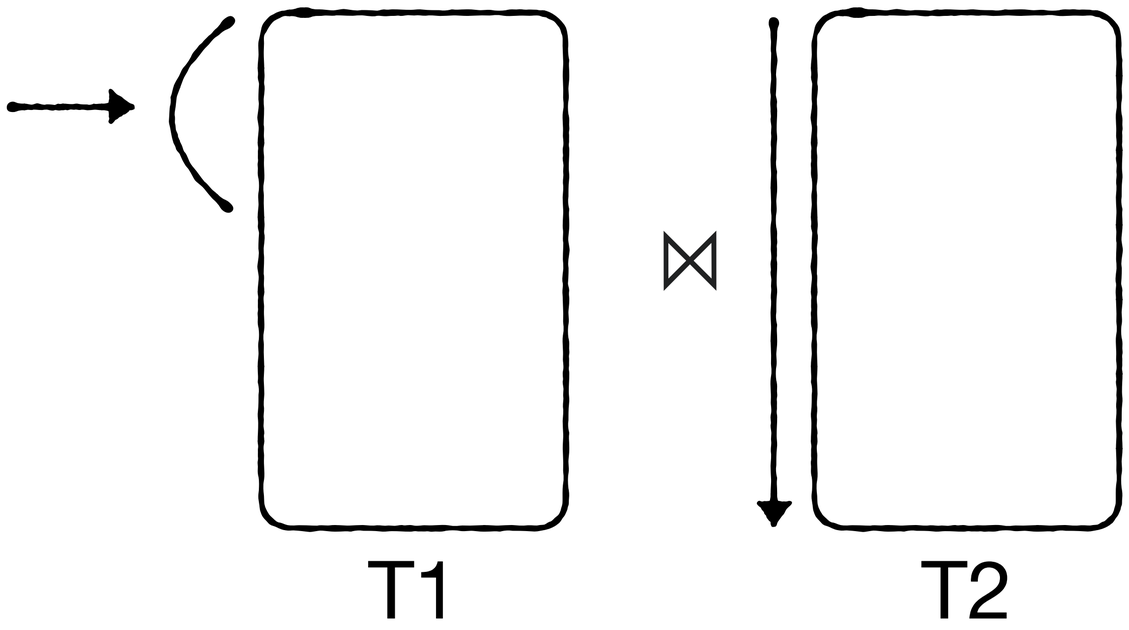
For each block in one relation, scan whole other relation



Cost?

Block Nested Loop

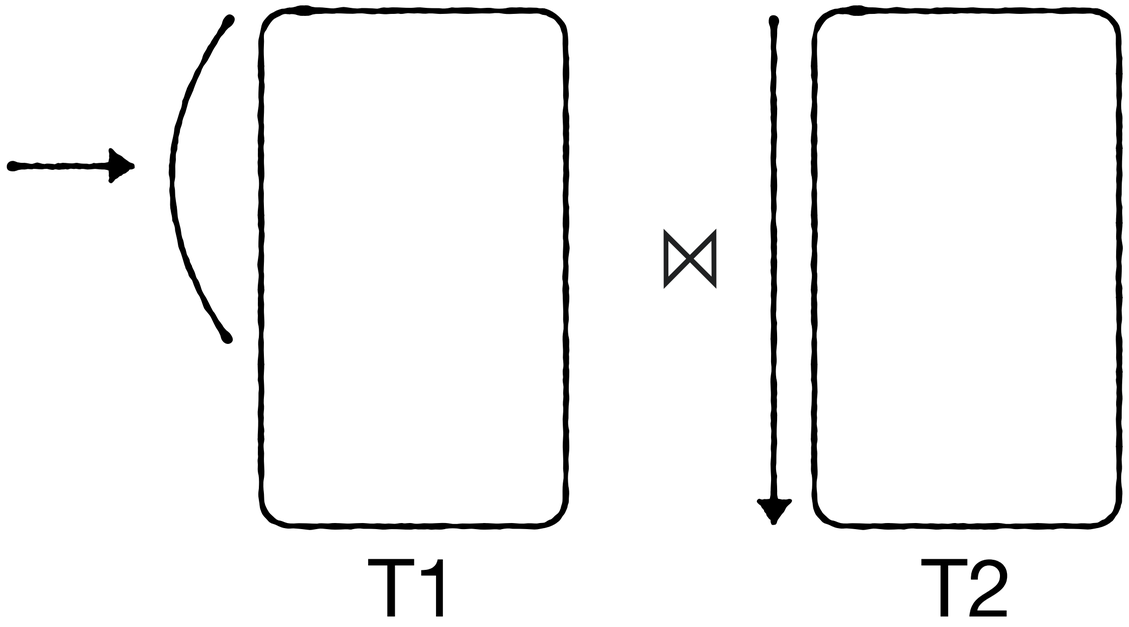
For each block in one relation, scan whole other relation



Cost: O(|T1|/B · |T2|/B) I/O

Block Nested Loop

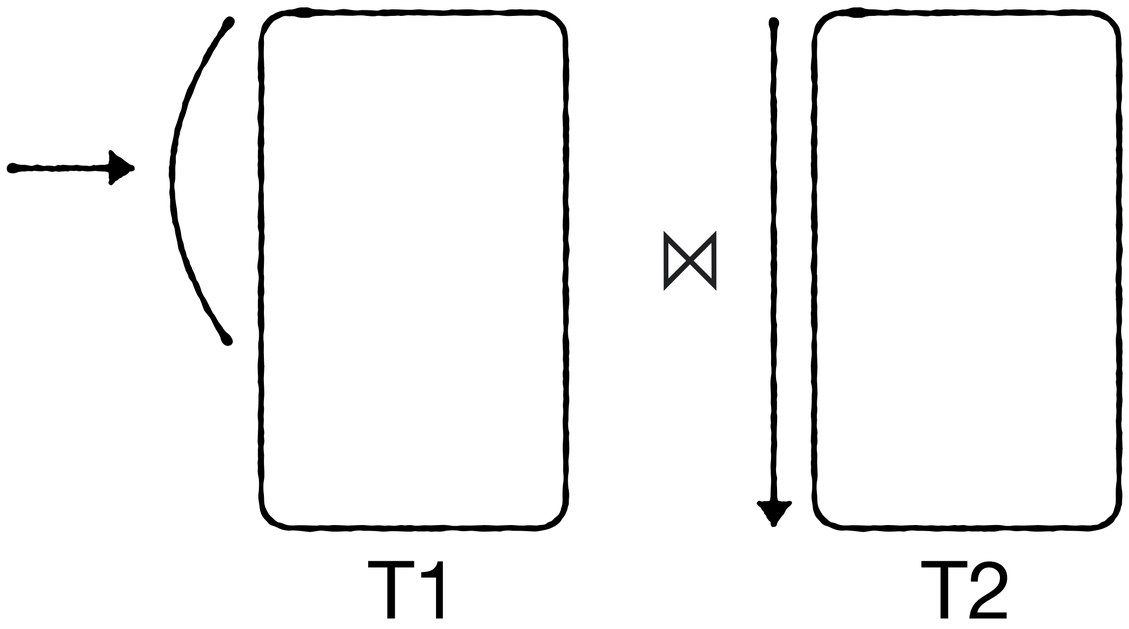
What if we read Q pages from T1 for each scan of T2?



Cost: O(|T1|/B · |T2|/B) I/O

Block Nested Loop

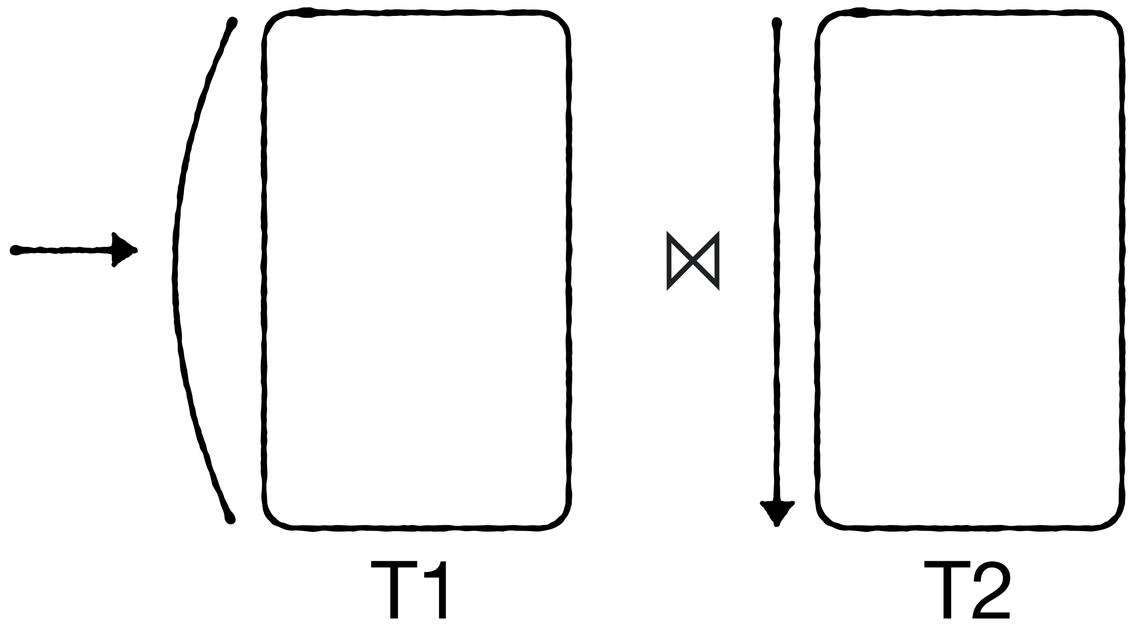
What if we read Q pages from T1 for each scan of T2?



Cost: O(**|T1|/(B · Q)** · |T2|/B) I/O

Block Nested Loop

What if we read Q pages from T1 for each scan of T2?



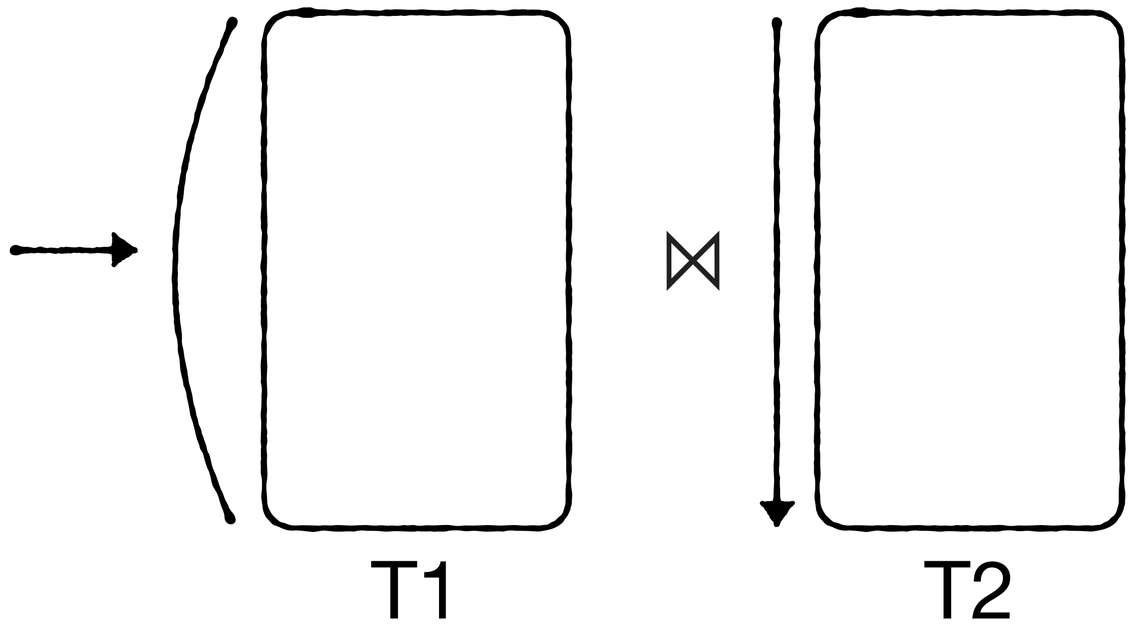
Cost: $O(|T1|/(B \cdot Q) \cdot |T2|/B)$ I/O

What if Q = min(|T1|/B, |T2|/B)?

(One table fits in memory)

Block Nested Loop

What if we read Q pages from T1 for each scan of T2?



Cost: O(|T1|/(B · Q) · |T2|/B) I/O

What if Q = min(|T1|/B, |T2|/B)?

(One table fits in memory)

Cost: O(|T1|/B + |T2|/B) I/O

Join Algorithms

Nested Loop

Block Nested Loop

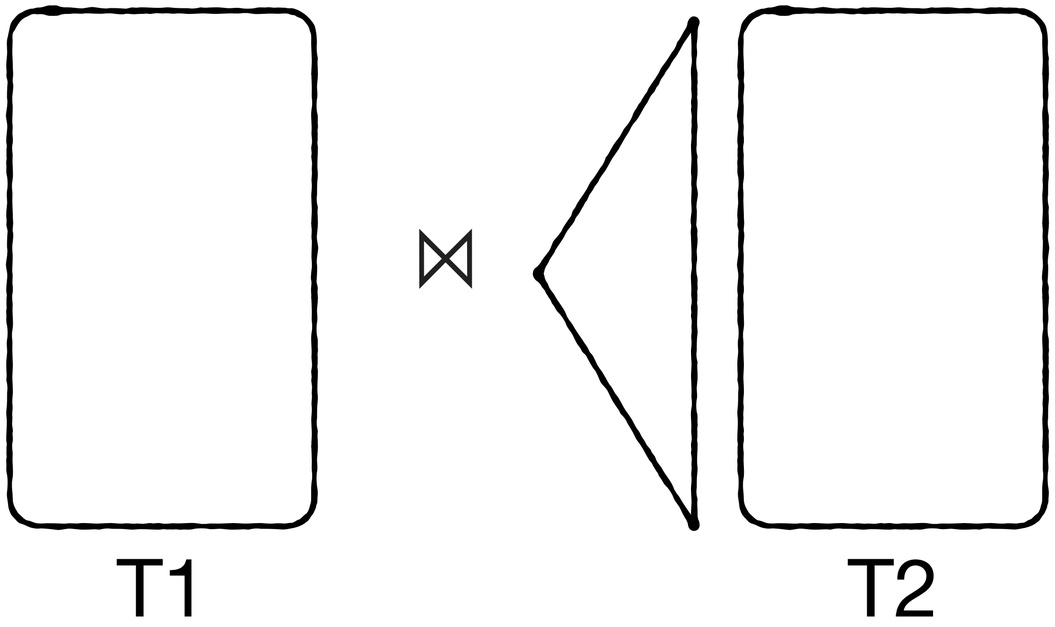
Index-Join

Sort-merge Join

Grace hash Join

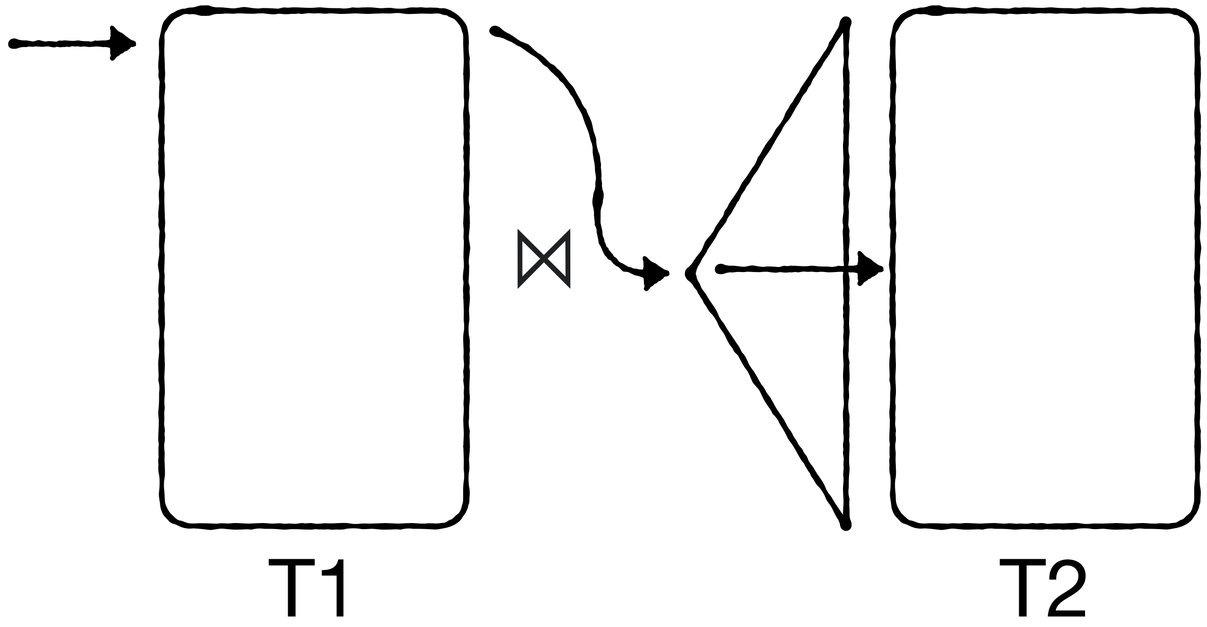
Index-Join

Suppose we have index on one of the relations



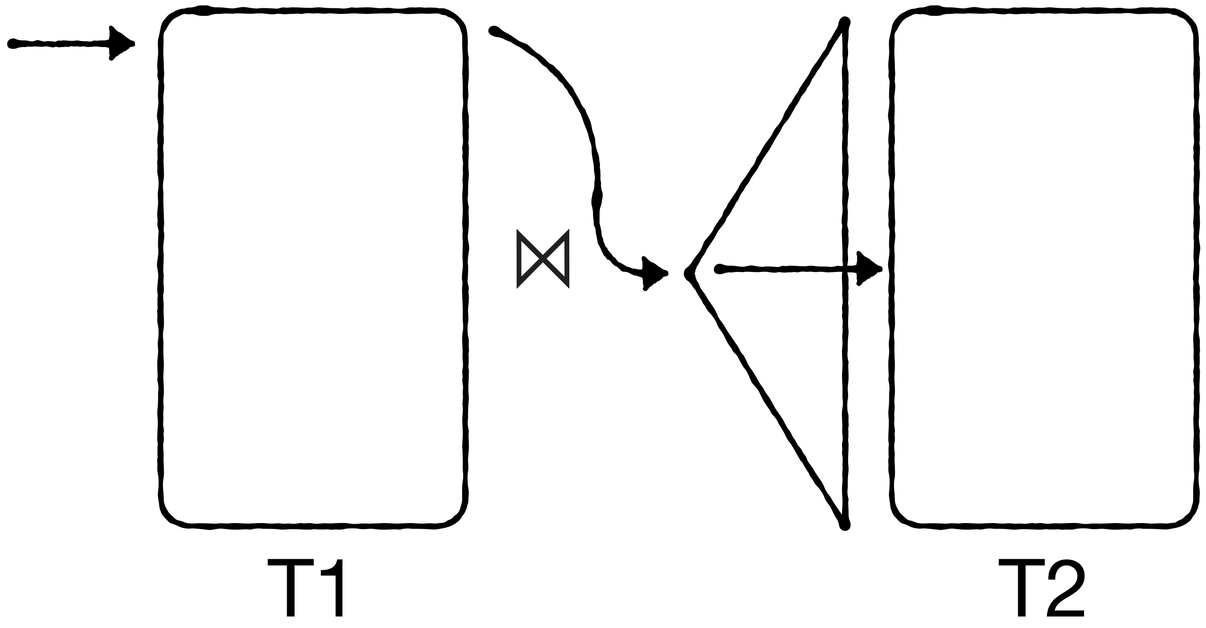
Index-Join

For each entry in one relation, search index for other relation



Index-Join

For each entry in one relation, search index for other relation

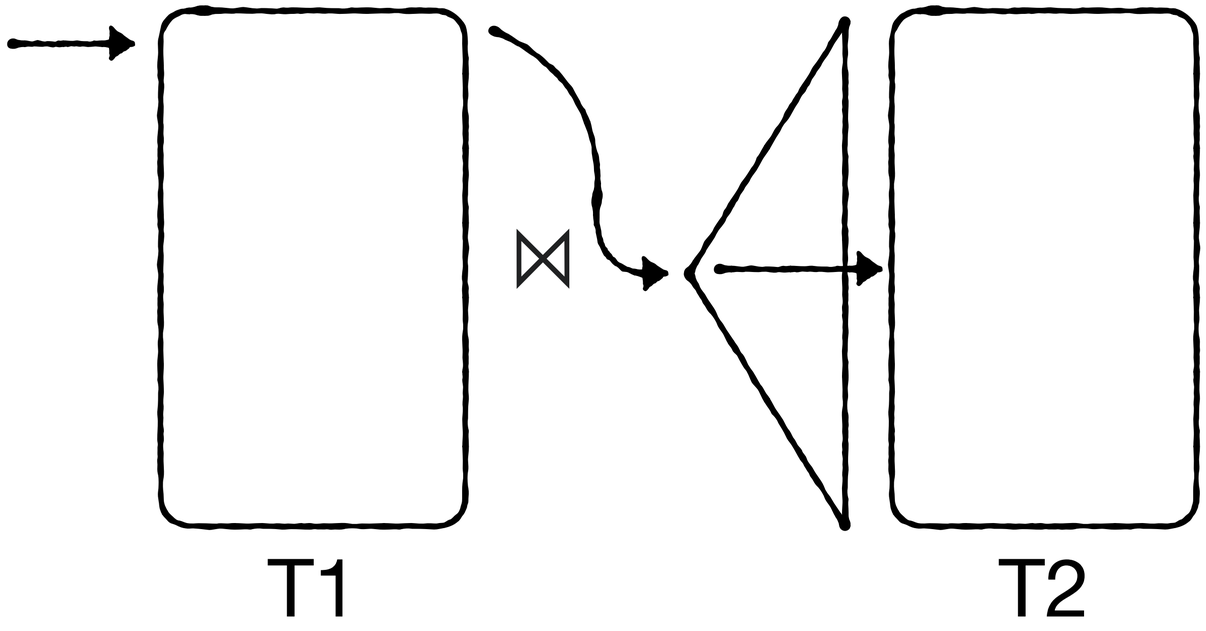


Cost?

(Assuming B-tree, clustered or unclustered)

Index-Join

For each entry in one relation, search index for other relation



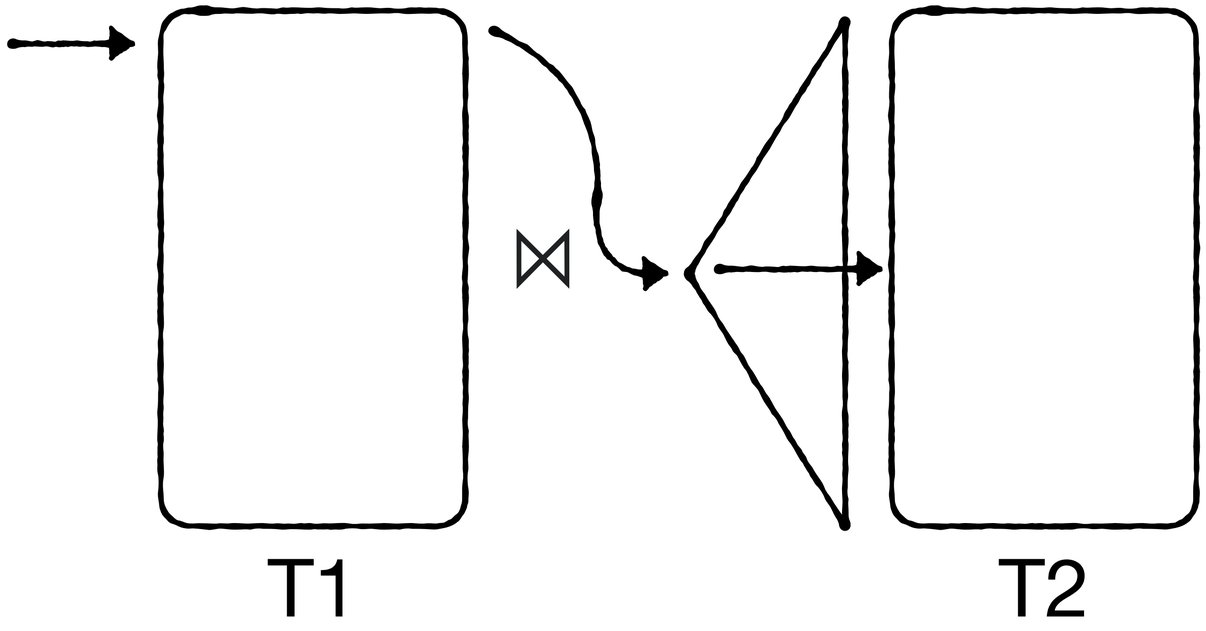
Cost:

$O(|T1|/B + |T1| \cdot \log_B |T2|)$ I/O

(Assuming B-tree, clustered or unclustered)

Index-Join

For each entry in one relation, search index for other relation



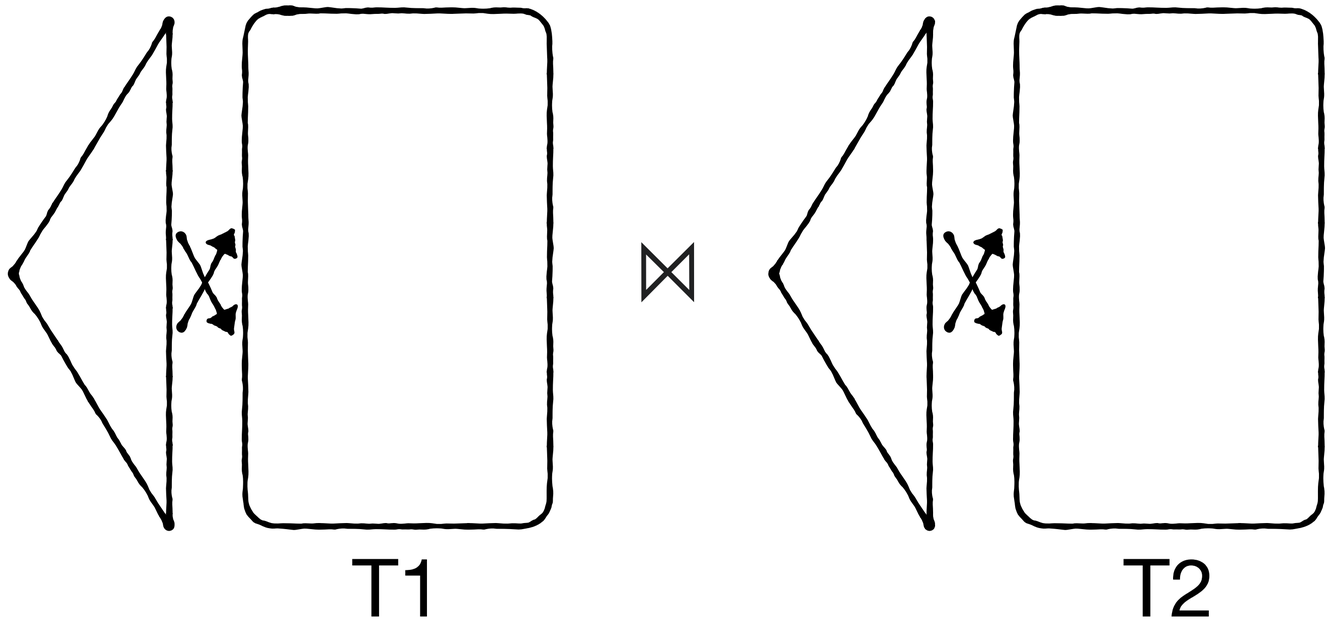
Cost:

$O(|T1| \cdot \log_B |T2|)$ I/O

(Assuming B-tree, clustered or unclustered)

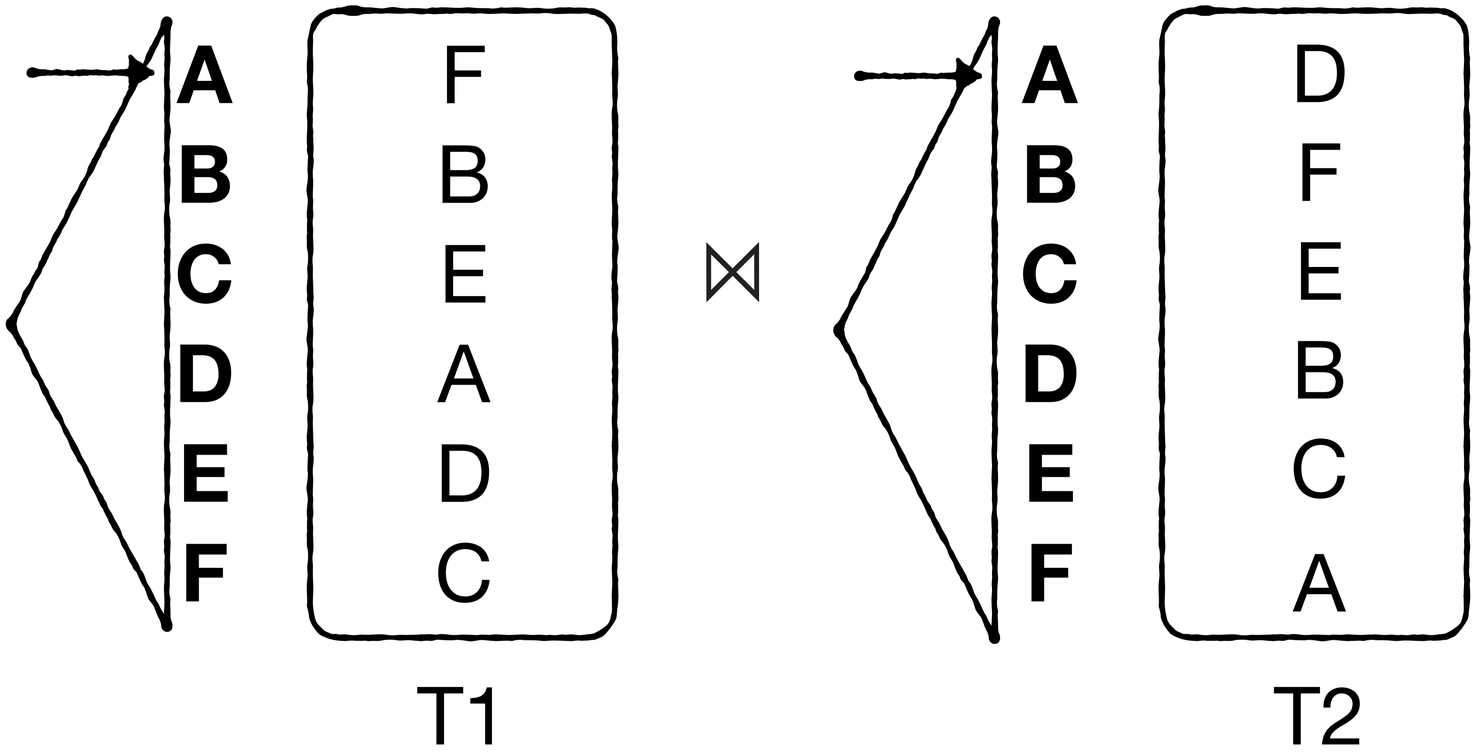
Index-Join

What if both relations have an unclustered index on the join key?



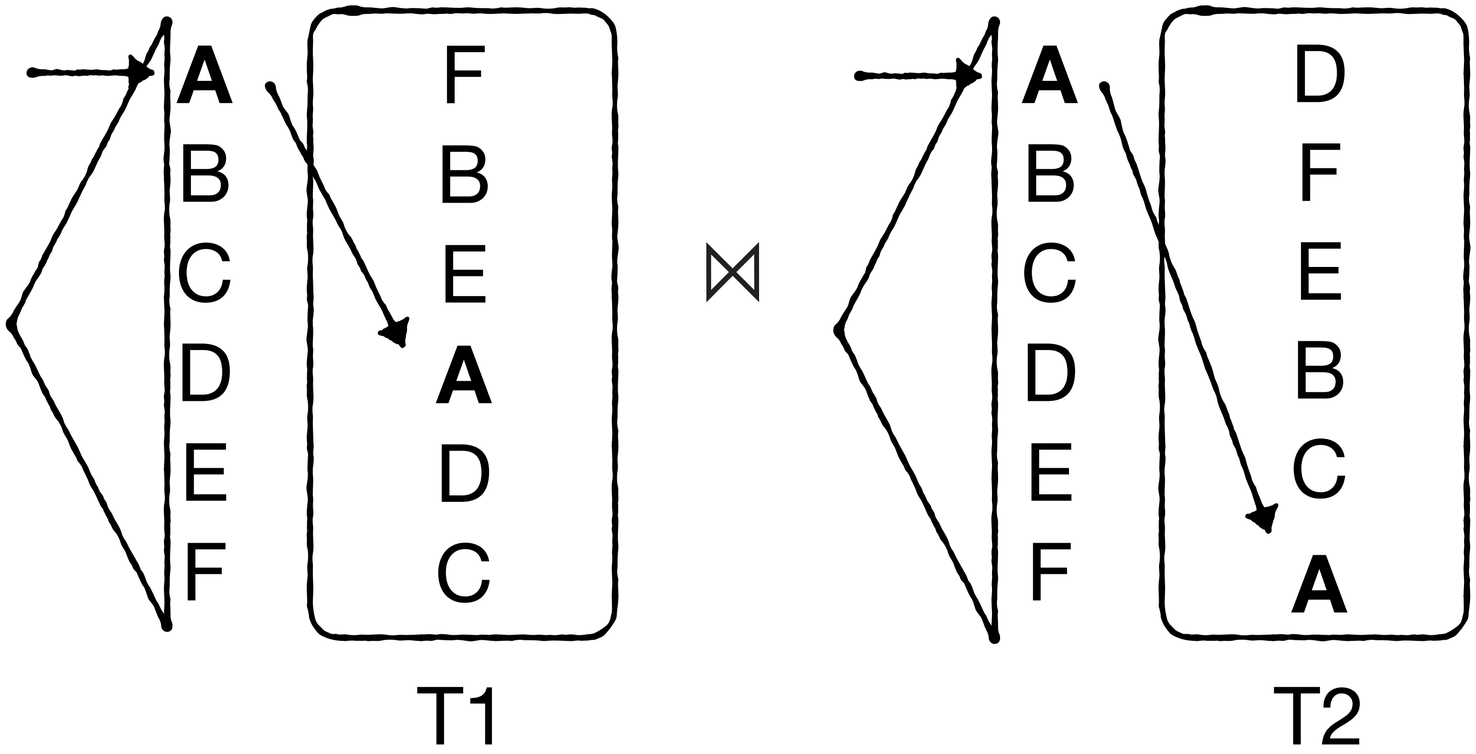
Index-Join

What if both relations have an unclustered index on the join key?



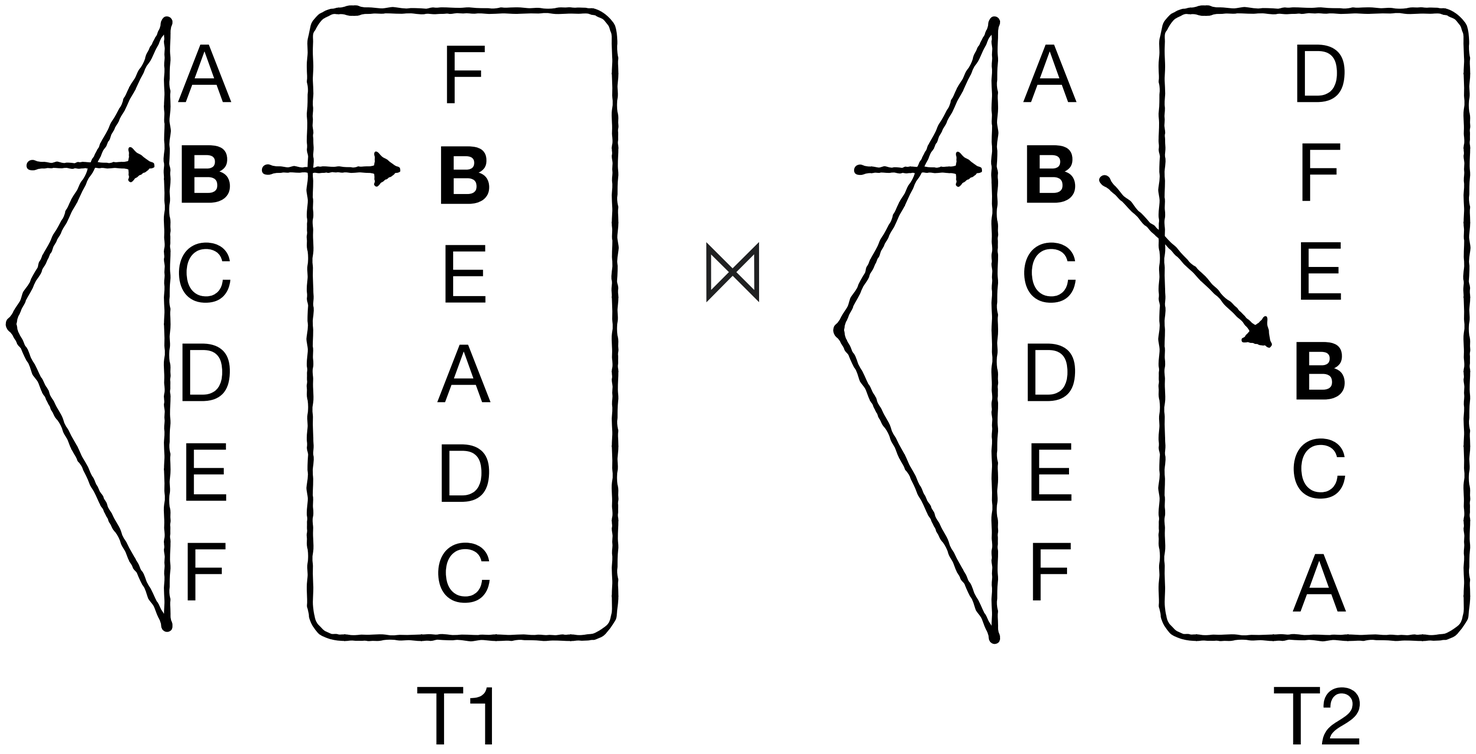
Index-Join

What if both relations have an unclustered index on the join key?



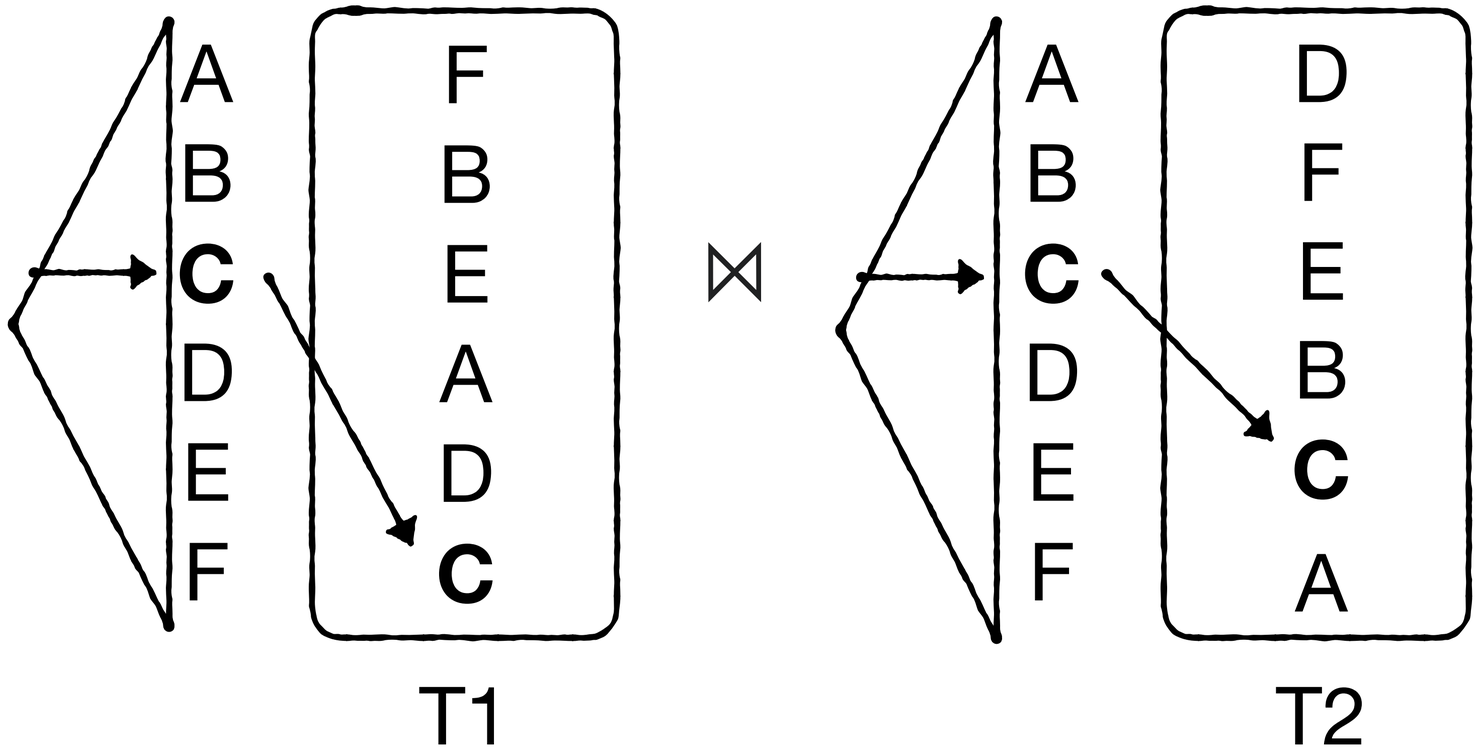
Index-Join

What if both relations have an unclustered index on the join key?



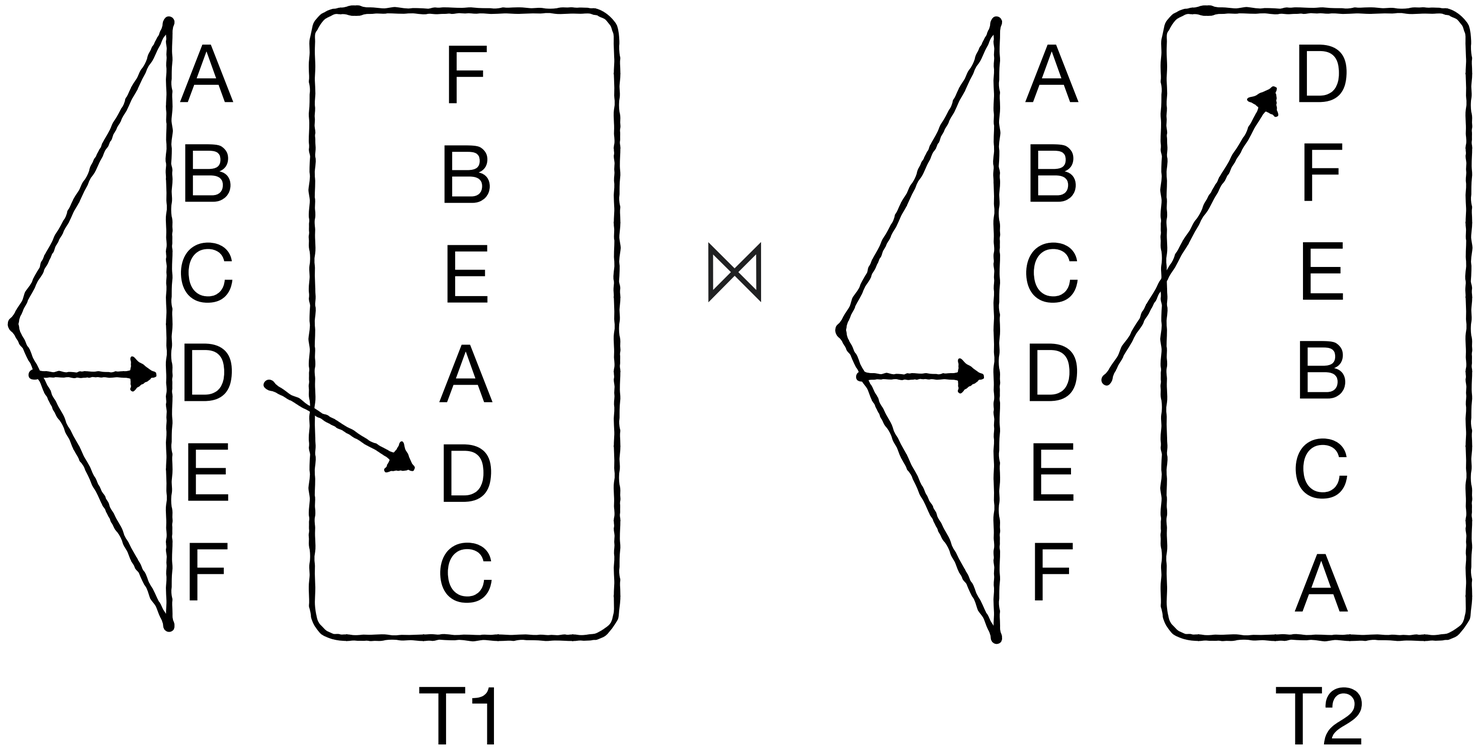
Index-Join

What if both relations have an unclustered index on the join key?



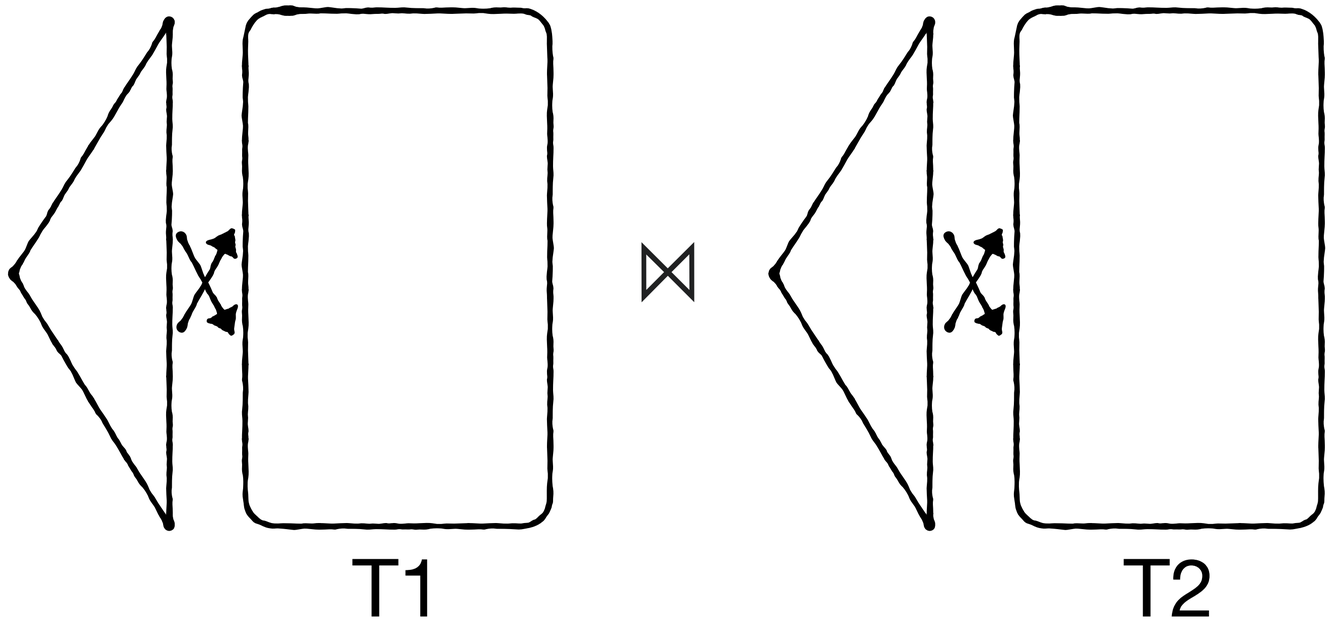
Index-Join

What if both relations have an unclustered index on the join key?



Index-Join

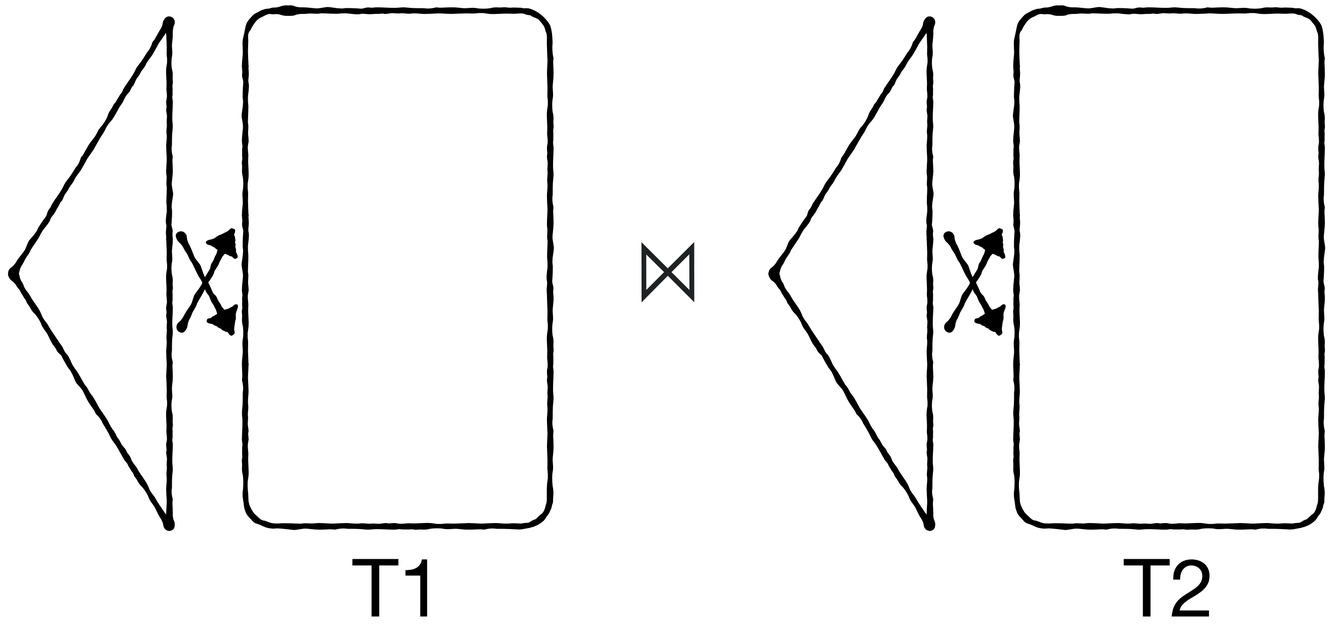
What if both relations have an unclustered index on the join key?



Cost?

Index-Join

What if both relations have an unclustered index on the join key?

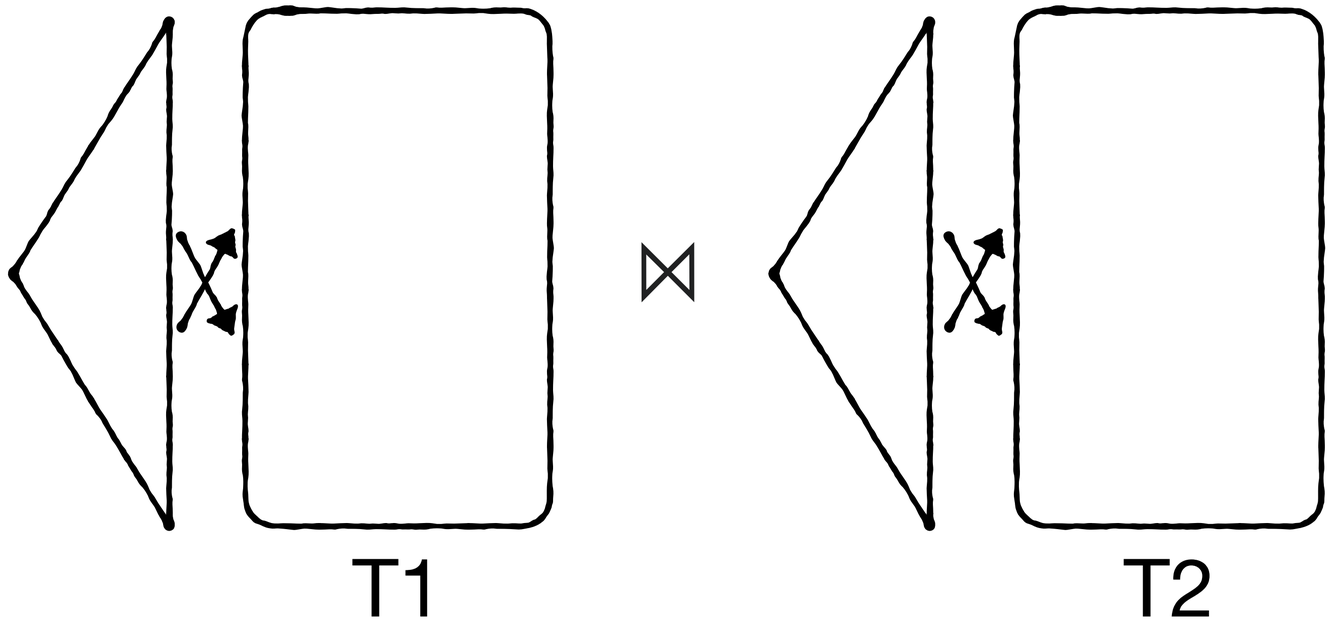


Cost:

O(|T1|/B + |T1| + |T2|/B + |T2|)

Index-Join

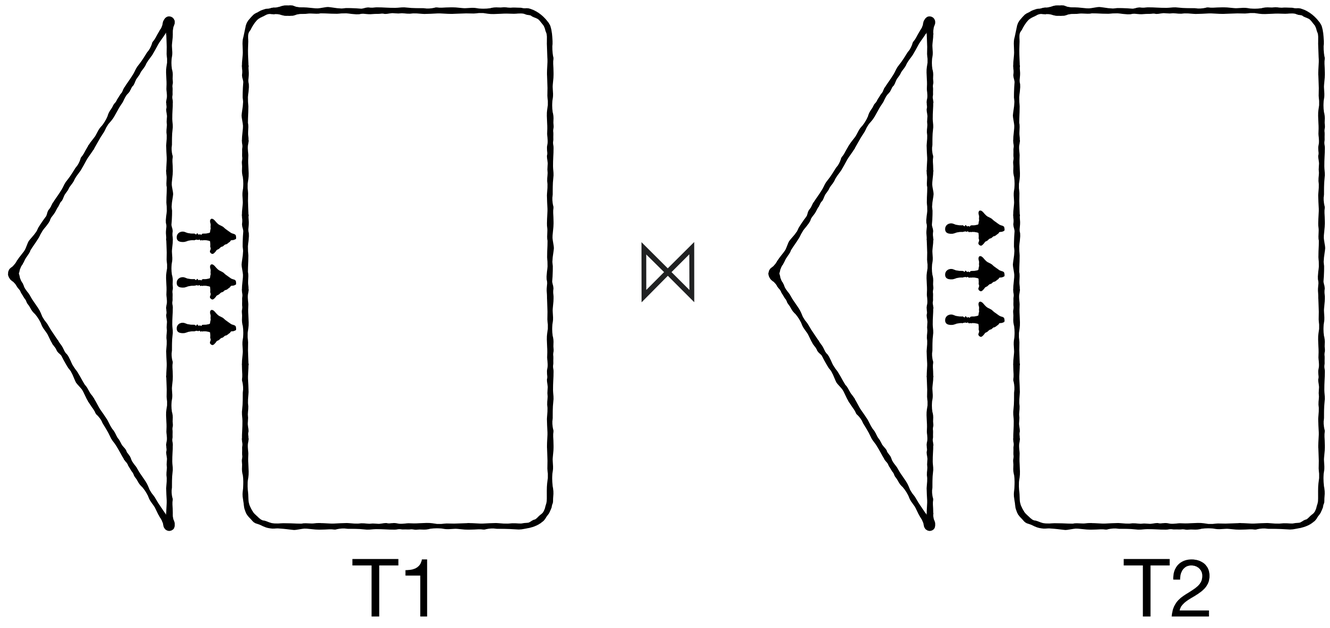
What if both relations have an unclustered index on the join key?



Cost:
O(|T1| + |T2|)

Index-Join

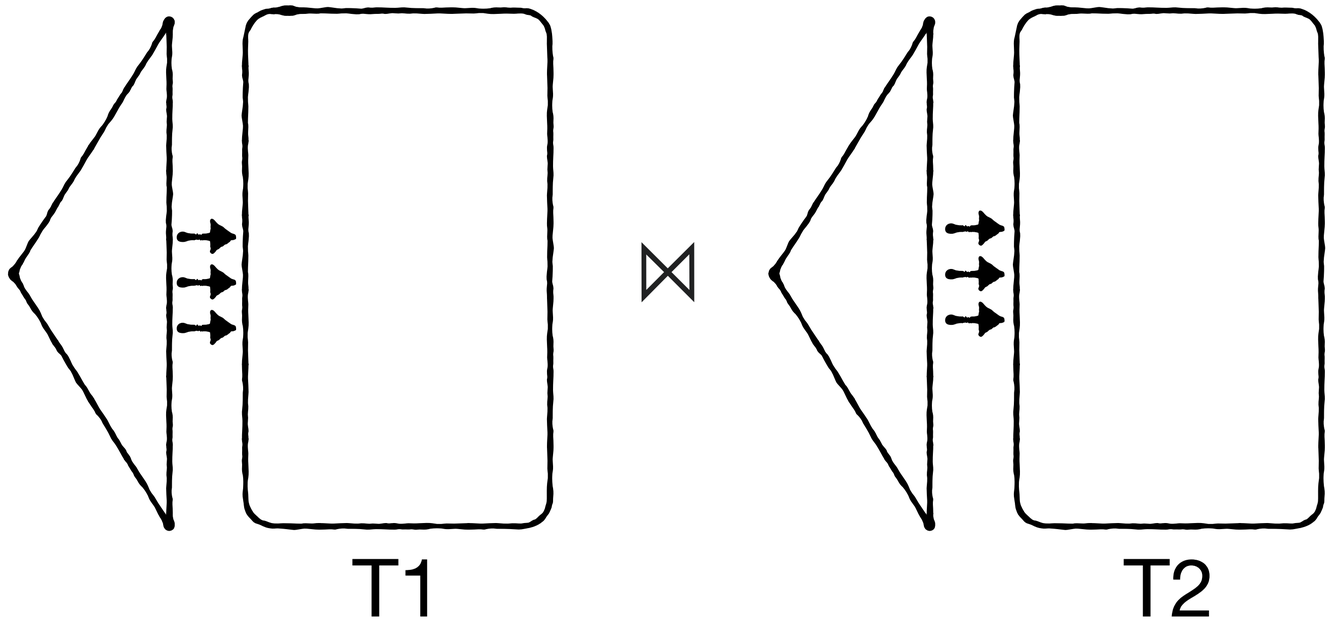
What if both indexes are clustered?



Cost?

Index-Join

What if both indexes are clustered?



Cost: O(|T1|/B + |T2|/B)

Nested Loop

Block Nested Loop

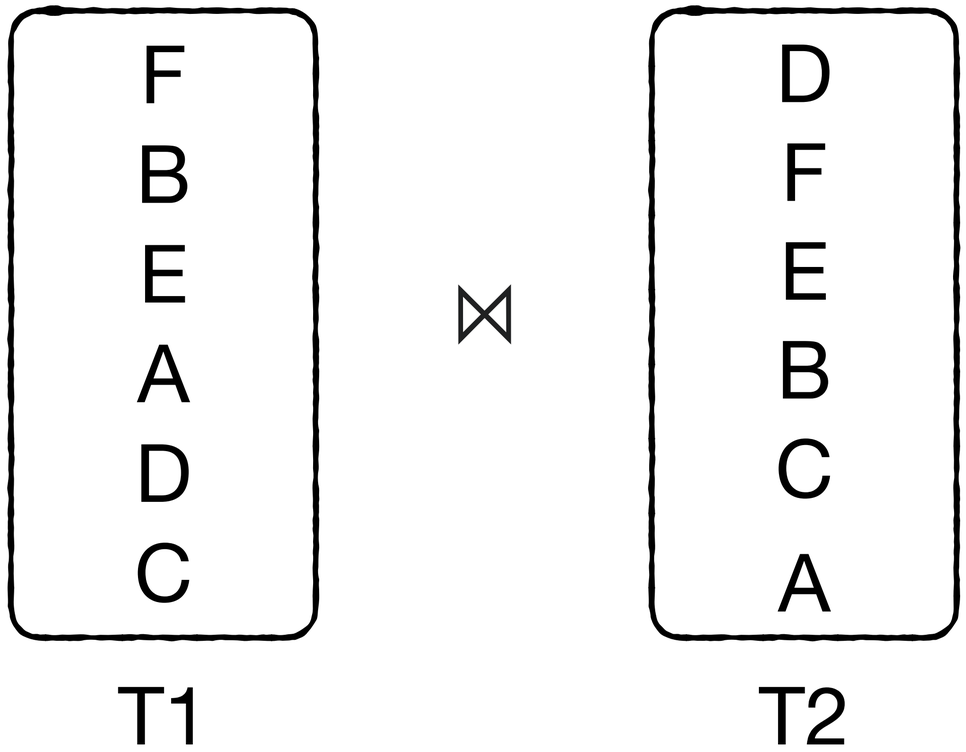
Index-Join

Sort-Merge Join

Grace hash Join

Sort-Merge Join

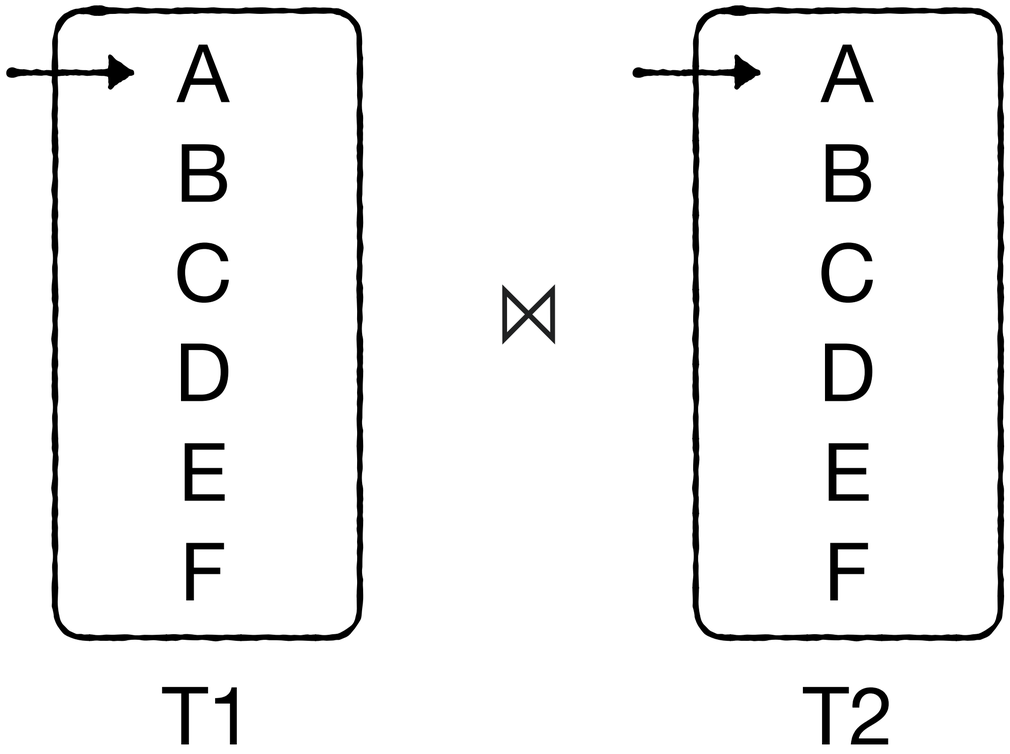
(A) Sort both relations based on join key



Sort-Merge Join

(A) Sort both relations based on join key

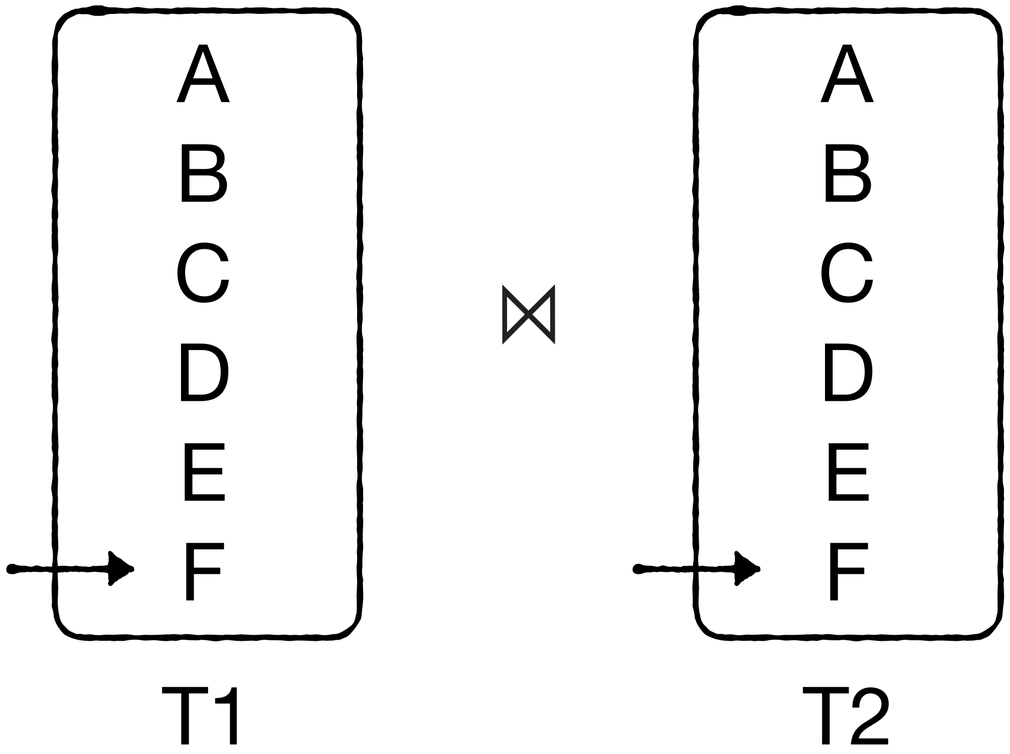
(B) Scan both relations linearly



Sort-Merge Join

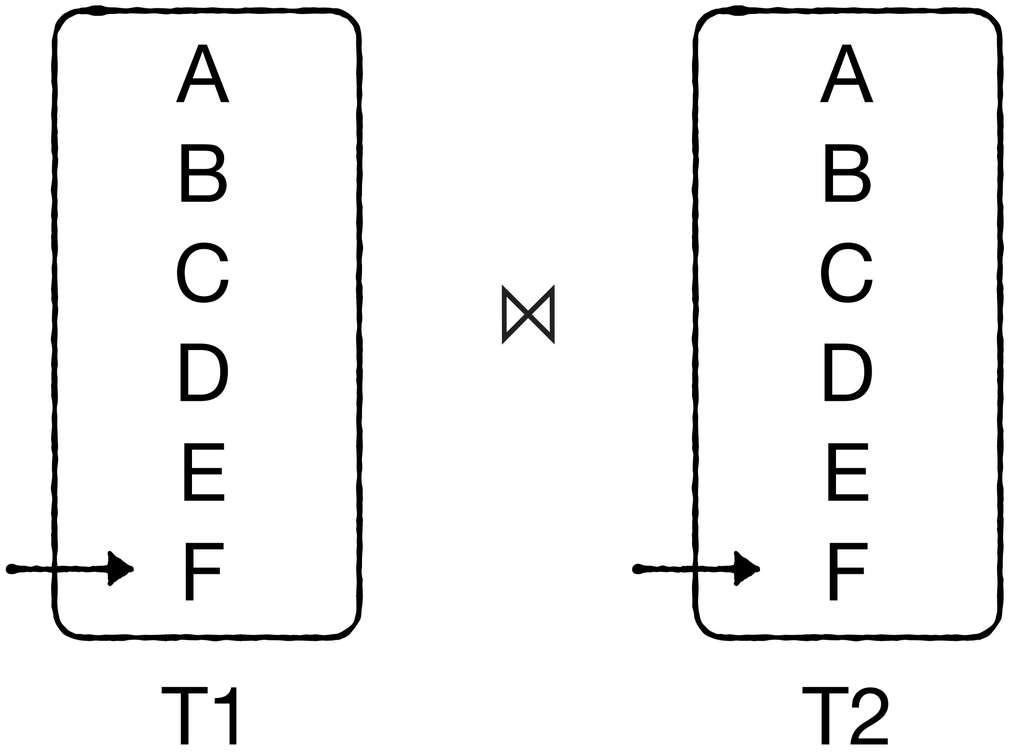
(A) Sort both relations based on join key

(B) Scan both relations linearly



Sort-Merge Join

(A) Sort both relations based on join key
(B) Scan both relations linearly



I/O cost?

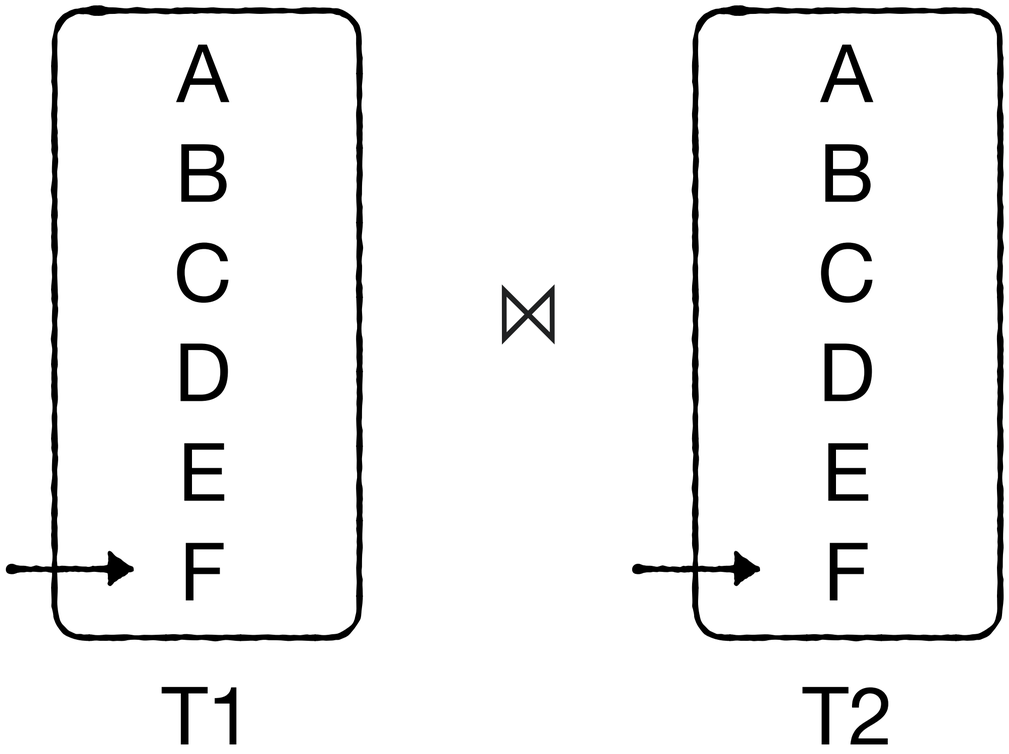
CPU cost?

Memory cost?

Assuming both relations do not fit in memory, and two-pass sorting

Sort-Merge Join

(A) Sort both relations based on join key
(B) Scan both relations linearly



I/O cost? **O(|T1|/B + |T2|/B)**

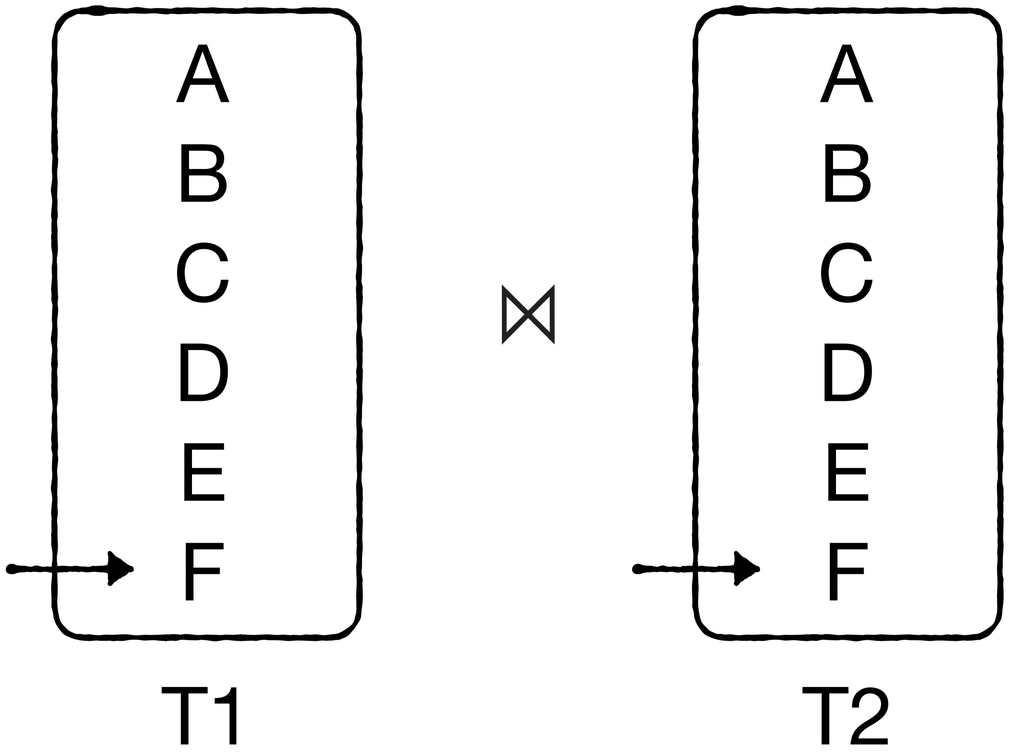
CPU cost?

Memory cost?

Assuming both relations do not fit in memory, and two-pass sorting

Sort-Merge Join

(A) Sort both relations based on join key
(B) Scan both relations linearly



I/O cost? $O(|T1|/B + |T2|/B)$

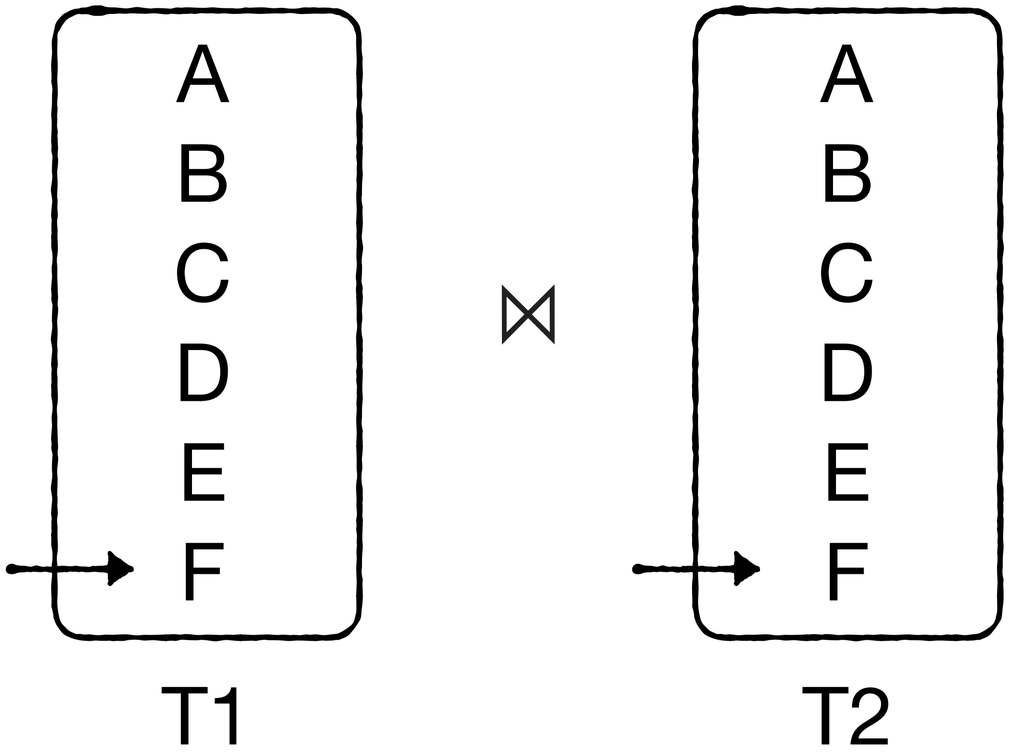
CPU cost? $\mathbf{O(|T1|\cdot\log_2|T1|+|T2|\cdot\log_2|T2|)}$

Memory cost?

Assuming both relations do not fit in memory, and two-pass sorting

Sort-Merge Join

(A) Sort both relations based on join key
(B) Scan both relations linearly



I/O cost? O(|T1|/B + |T2|/B)

CPU cost? $O(|T1|\cdot\log_2|T1|+|T2|\cdot\log_2|T2|)$

Memory cost? $O(\sqrt{\max(|T1|,|T2|)\cdot B})$

Assuming both relations do not fit in memory, and two-pass sorting

Nested Loop

Block Nested Loop

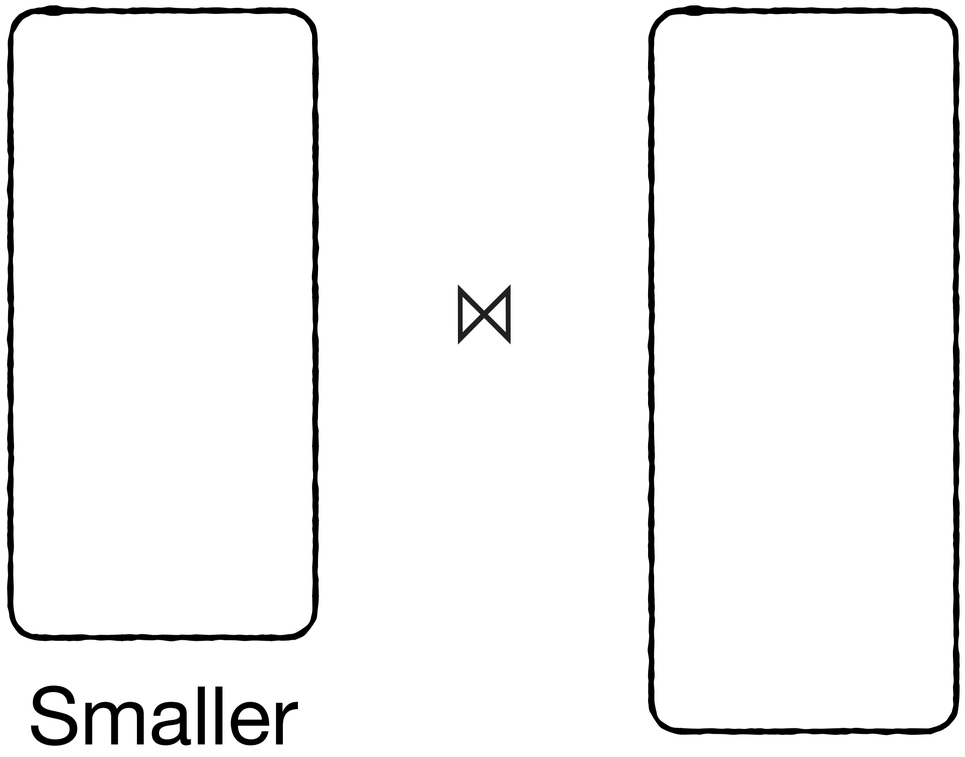
Index-Join

Sort-Merge Join

Grace Hash Join

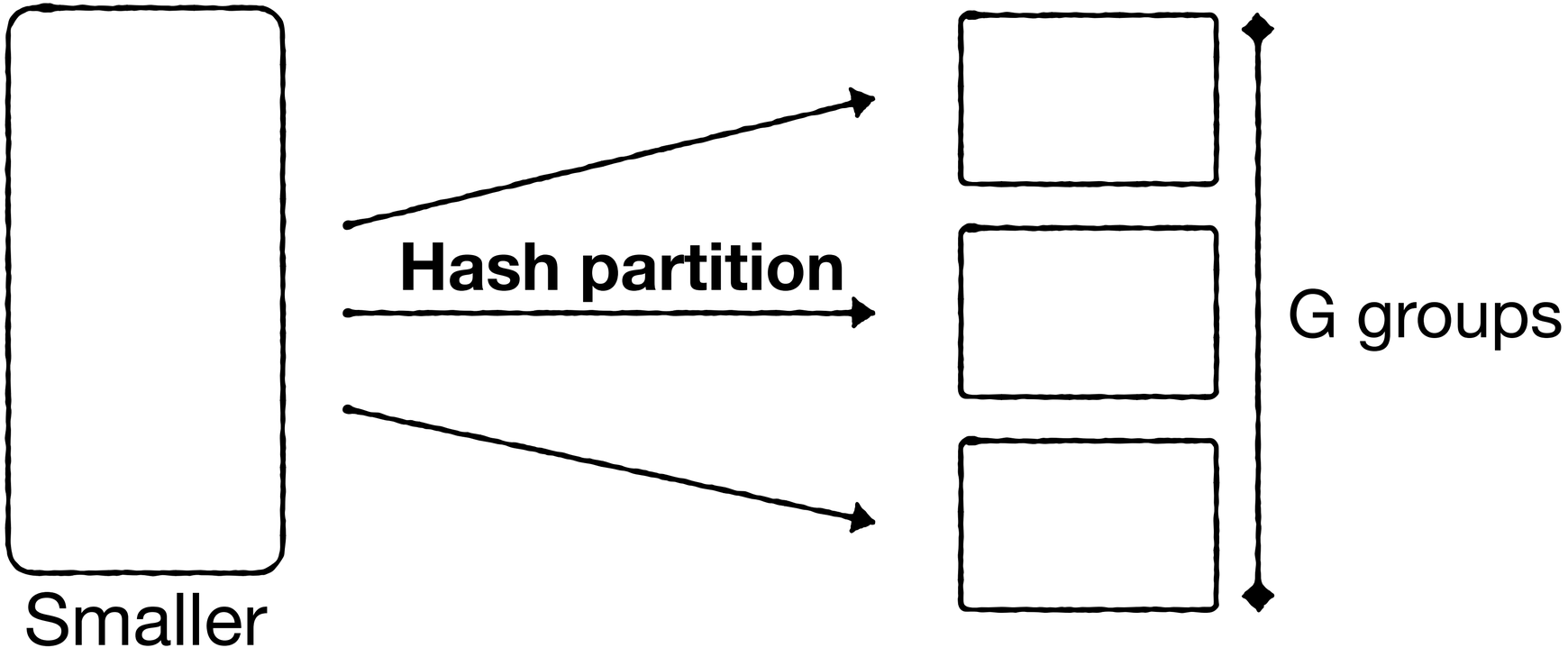
Grace Hash Join

Best for very large relations that do not fit in memory



Grace Hash Join

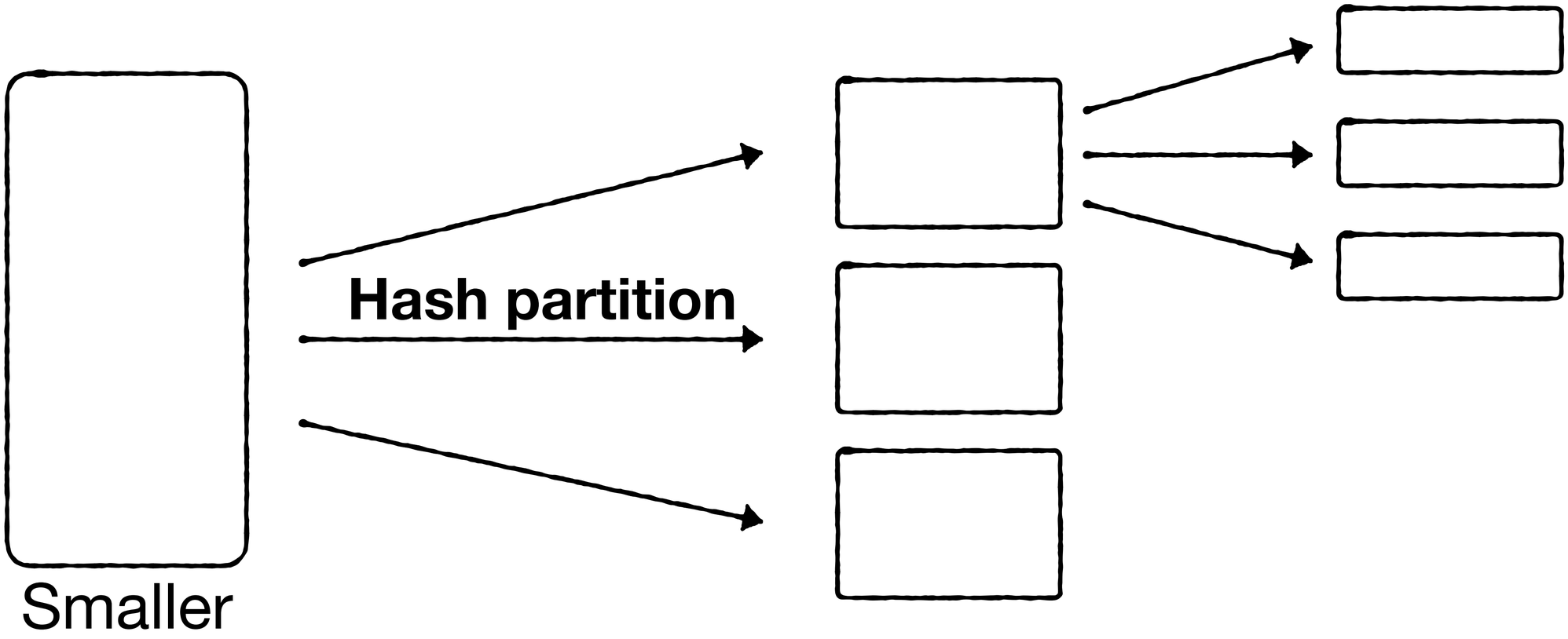
(A) Hash partition smaller table



Grace Hash Join

(A) Hash partition smaller table

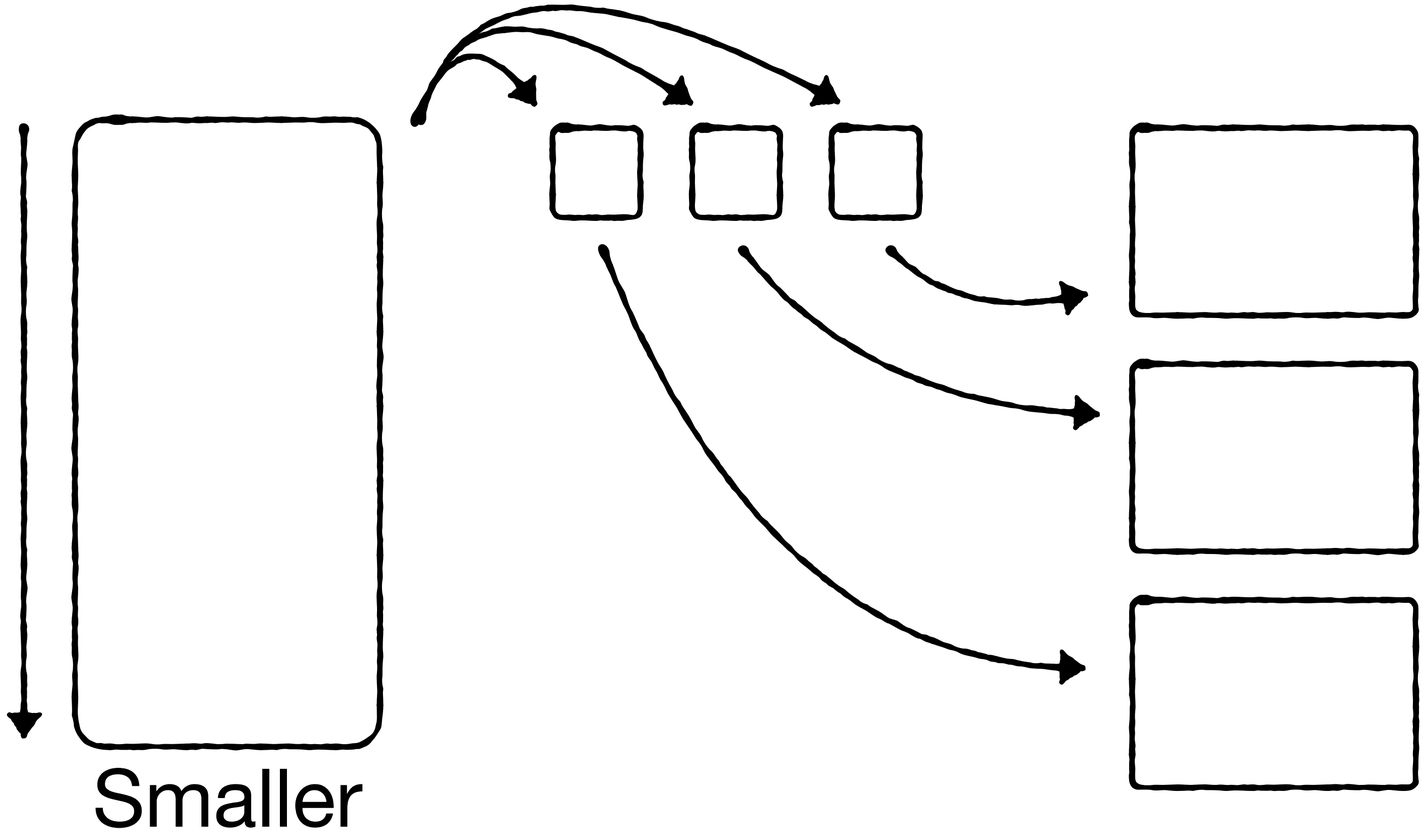
Repartition until each partition fits in memory



Grace Hash Join

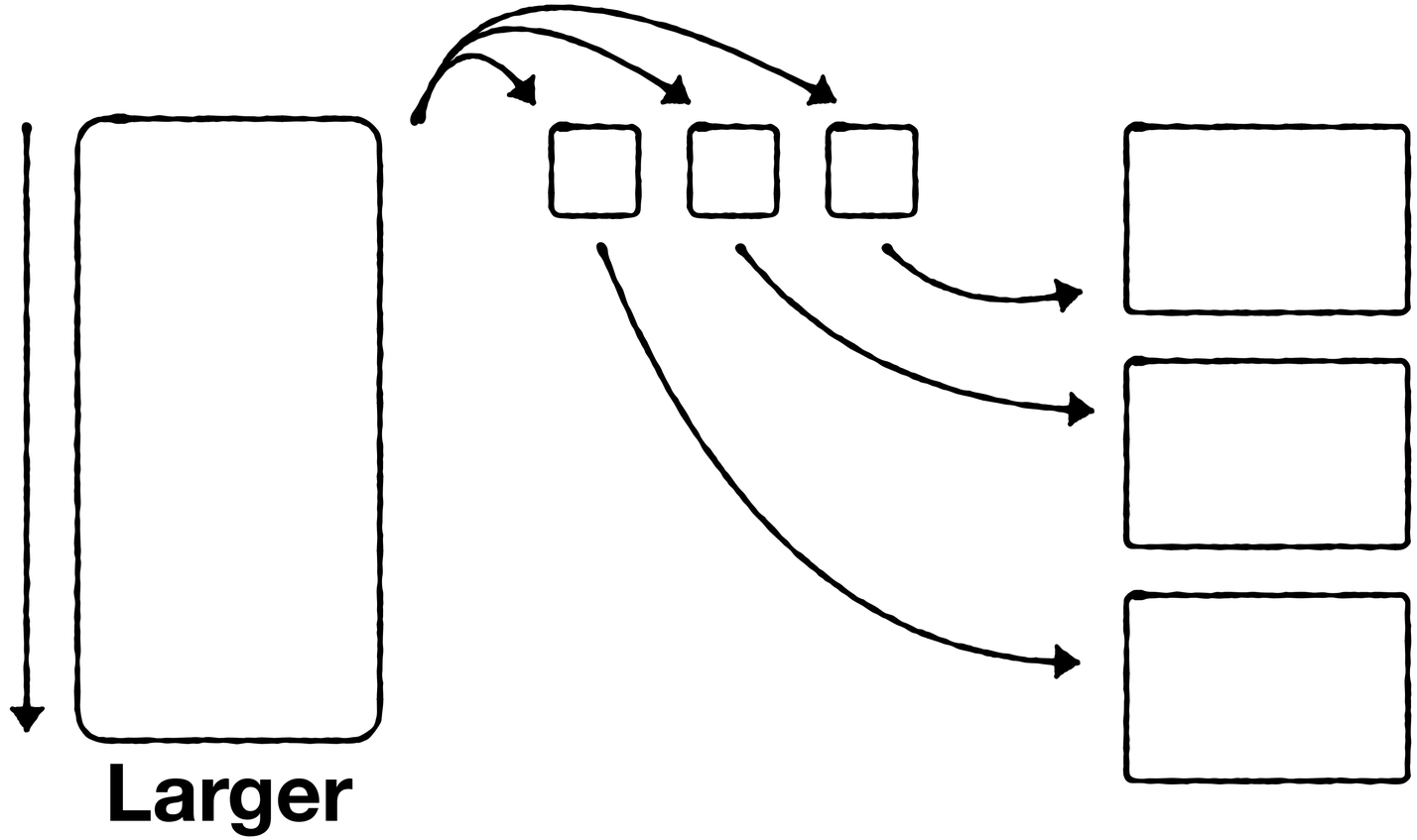
(A) Hash partition smaller table

Requires one pass and multiple buffers



Grace Hash Join

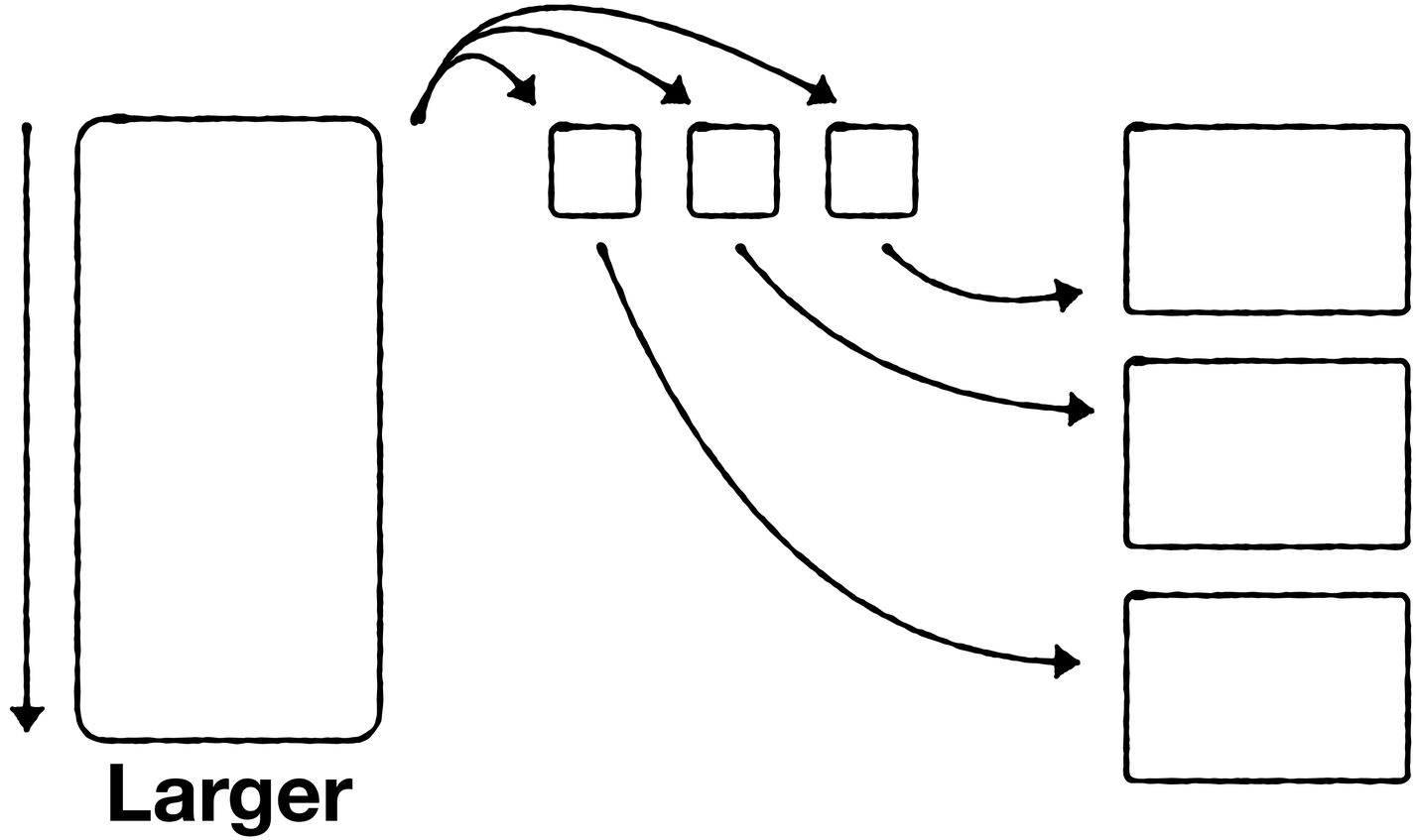
(B) Hash partition larger table using same hash function



Grace Hash Join

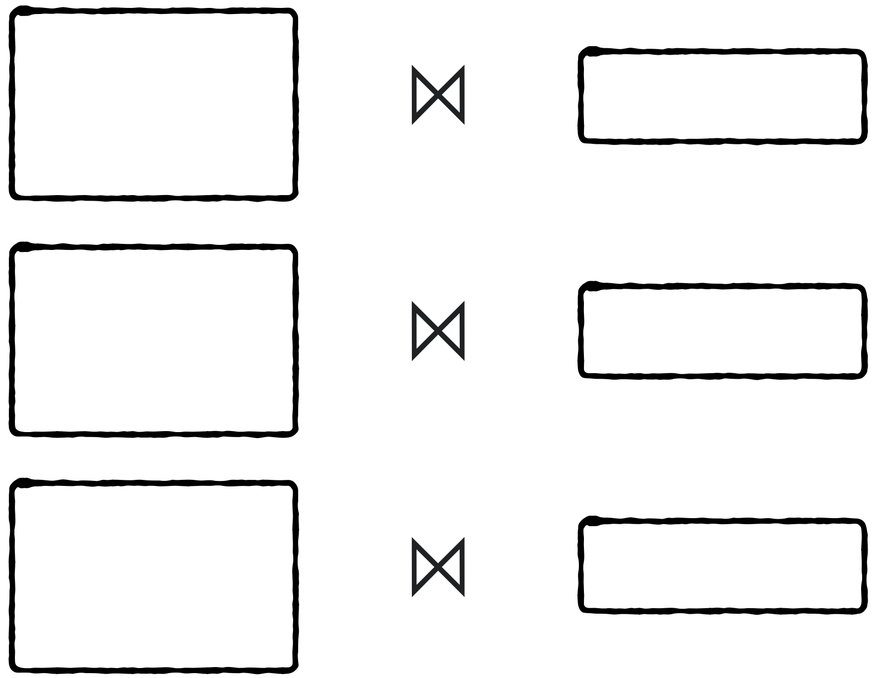
(B) Hash partition larger table using same hash function

For larger table, each partition can be larger than memory.



Grace Hash Join

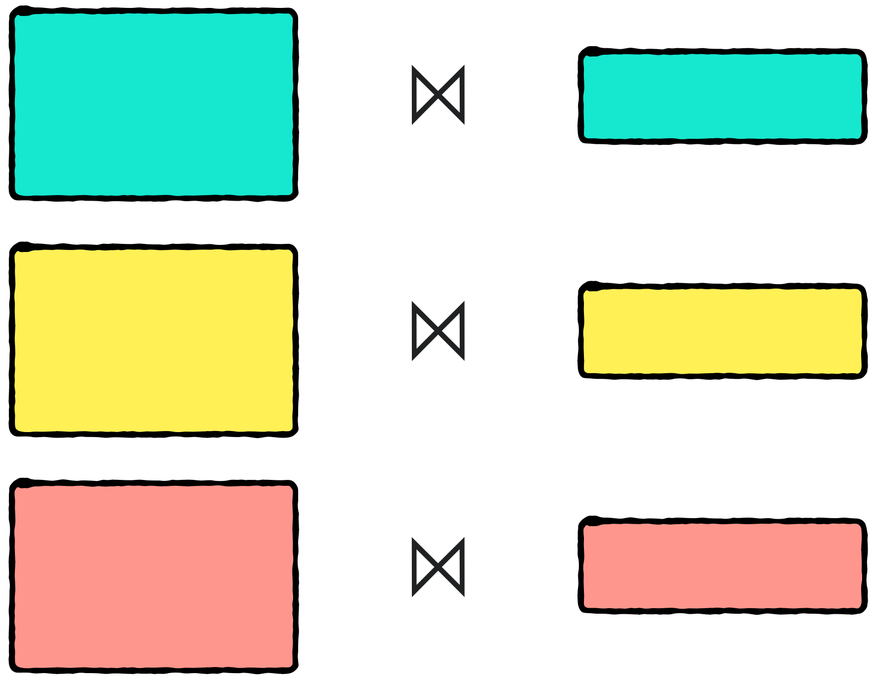
(C) Join each pair of matching partitions independently



Grace Hash Join

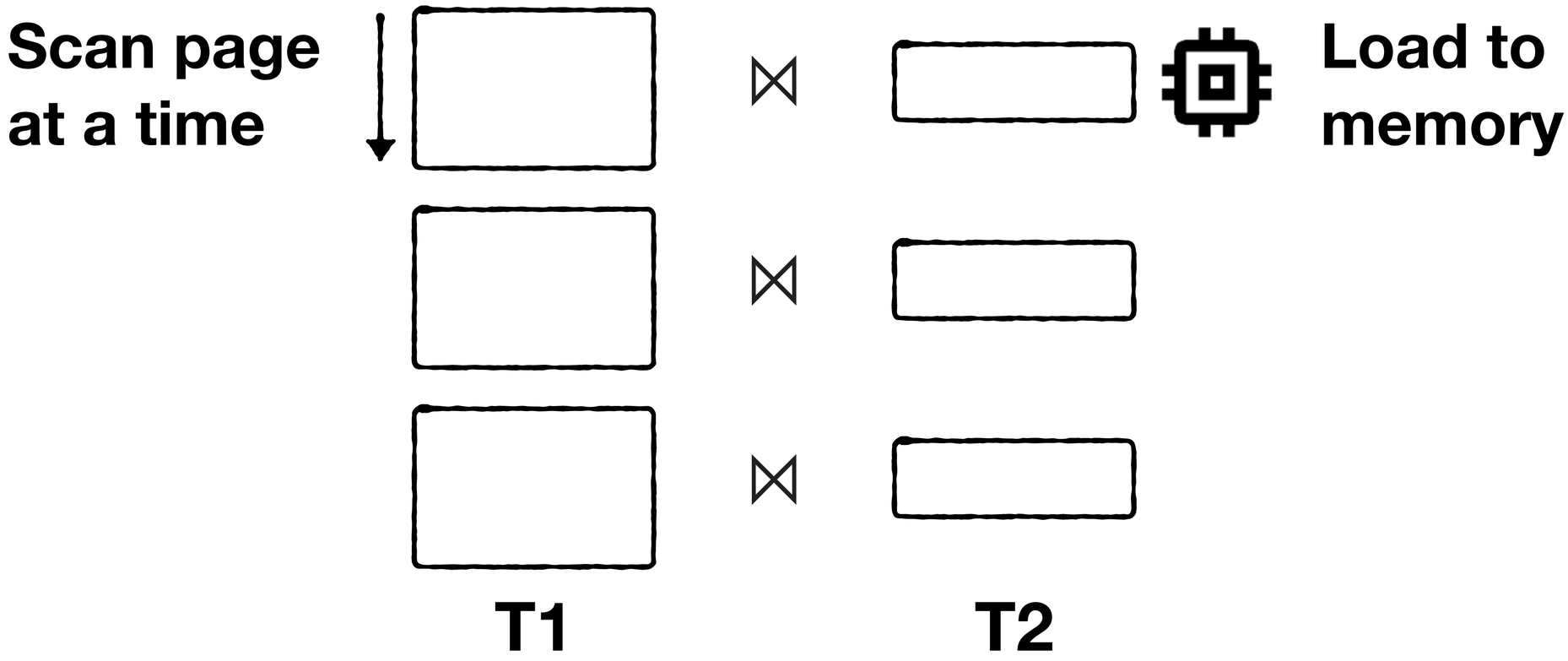
(C) Join each pair of matching partitions independently

Only a pair of matching partitions can have joining keys since we hash partitioned



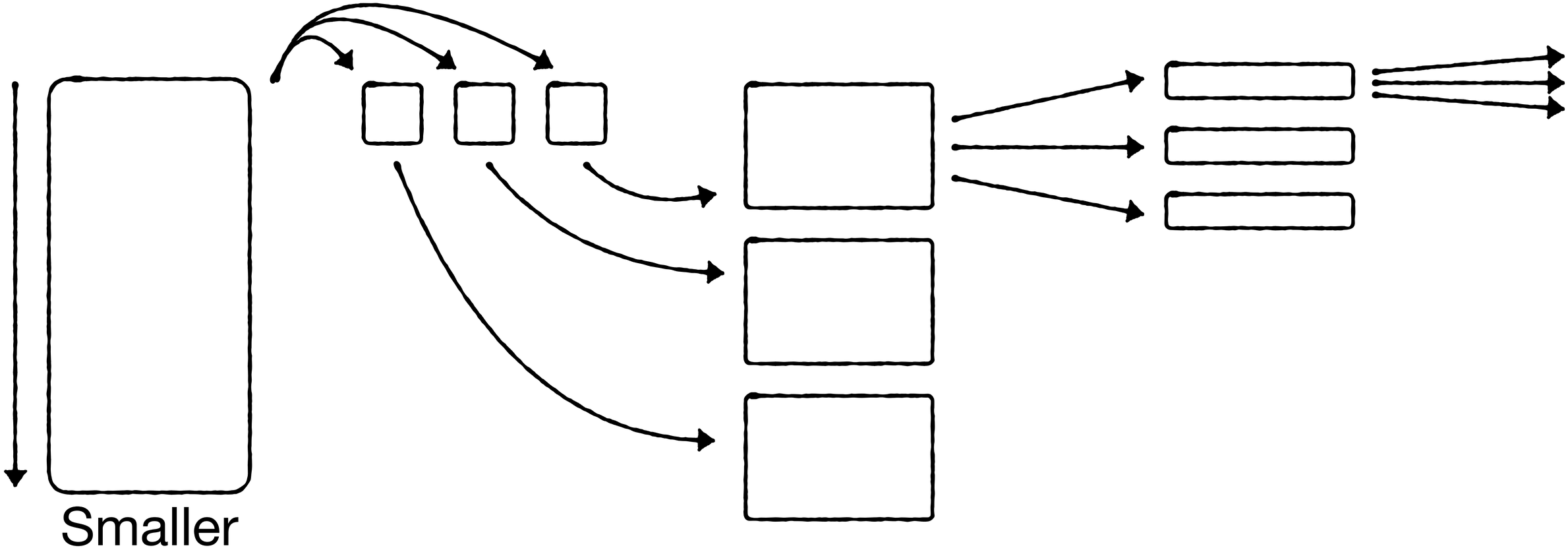
Grace Hash Join

(C) Join each pair of matching partitions independently



Grace Hash Join Analysis

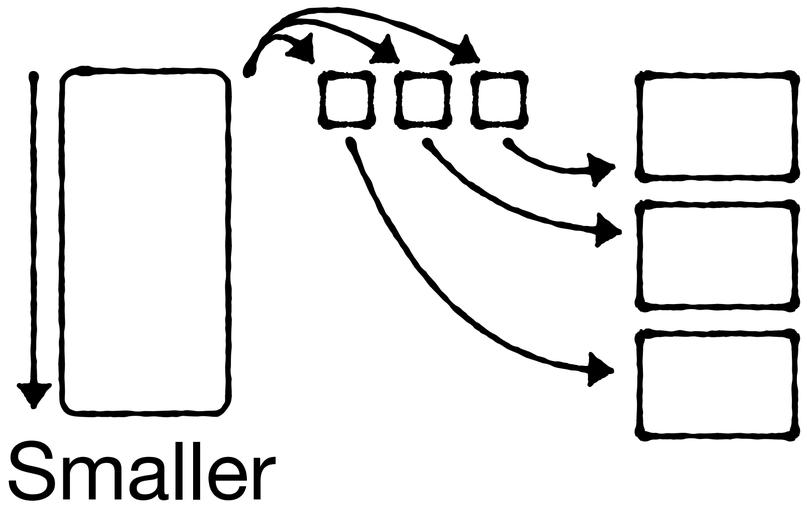
When partitioning smaller table, number of available buffers dictates how many iterations we need until all partitions it in memory



Grace Hash Join Analysis

When partitioning smaller table, number of available buffers dictates how many iterations we need until all partitions it in memory

Assuming M entries fit in memory, # iterations needed is:

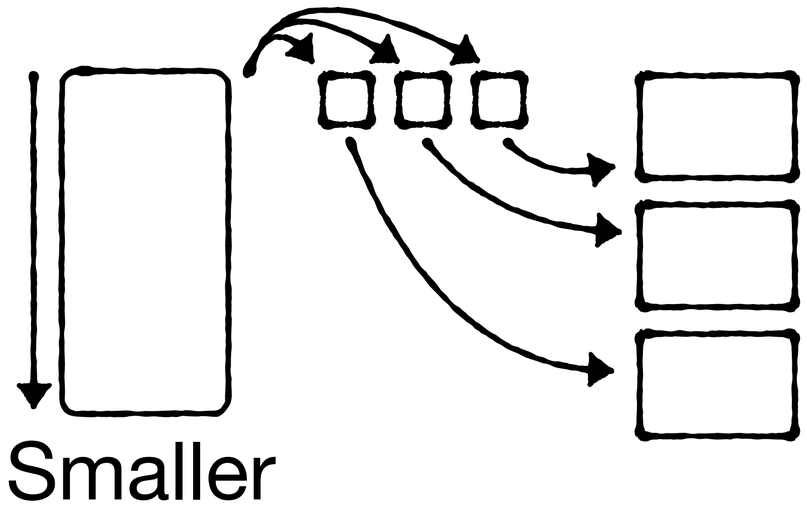


$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{M}\right)+1$$

Grace Hash Join Analysis

When partitioning smaller table, number of available buffers dictates how many iterations we need until all partitions it in memory

Assuming M entries fit in memory, # iterations needed is:



$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{M}\right)+1$$

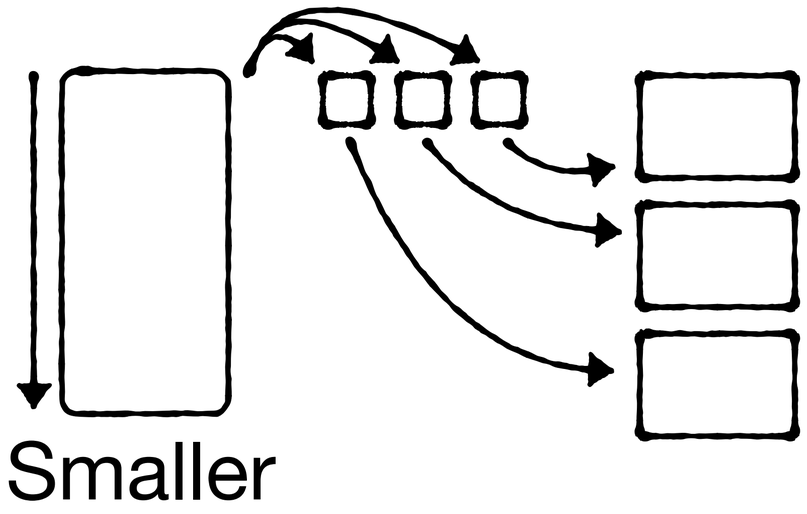
Partitioning fanout

 # partitions we must divide table into such that each fits in memory

Grace Hash Join Analysis

When partitioning smaller table, number of available buffers dictates how many iterations we need until all partitions it in memory

Assuming M entries fit in memory, # iterations needed is:



$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{M}\right) + 1$$

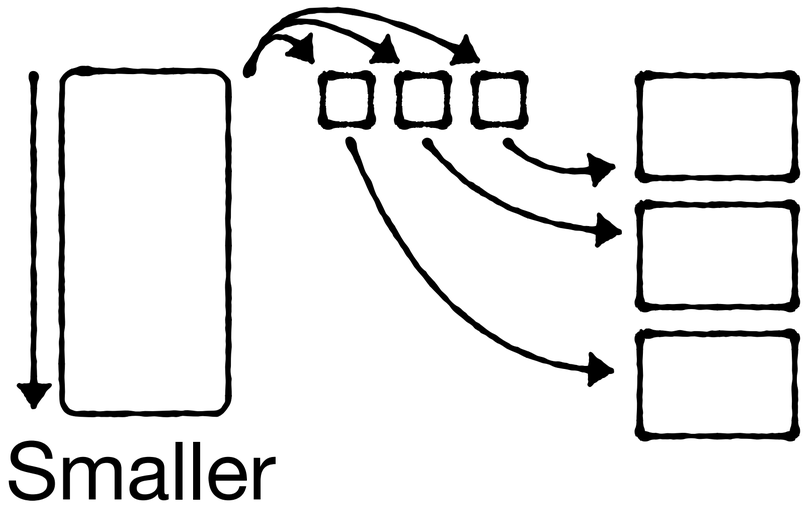
Partitioning Iterations

Joining Iteration

Grace Hash Join Analysis

When partitioning smaller table, number of available buffers dictates how many iterations we need until all partitions it in memory

Assuming M entries fit in memory, # iterations needed is:



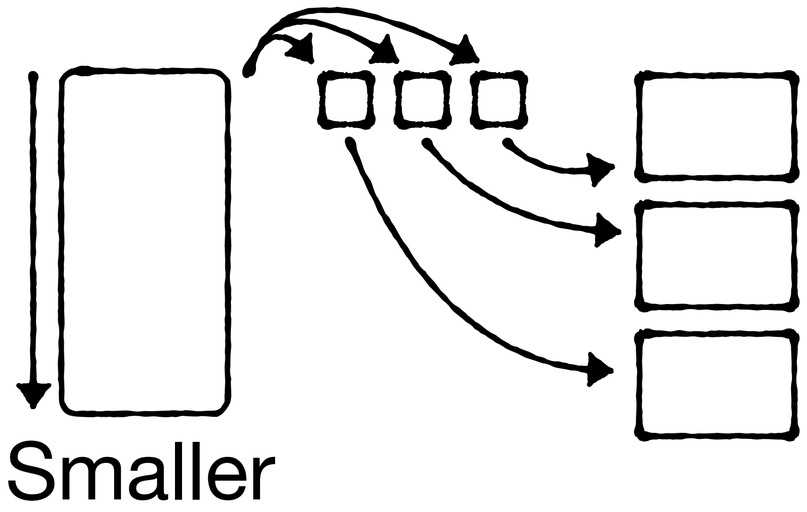
$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{M}\right) + 1 \;=\; \log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{B}\right)$$

Same as external merge sort :)

Grace Hash Join Analysis

When partitioning smaller table, number of available buffers dictates how many iterations we need until all partitions it in memory

Assuming M entries fit in memory, # iterations needed is:

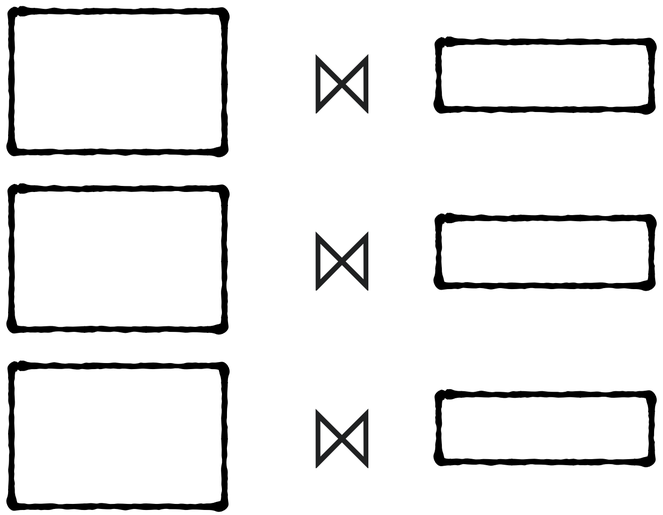


$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{B}\right) = 2 \quad \longrightarrow \quad M = \sqrt{|Smaller| \cdot B}$$

Equate to 2 and solve for M to obtain memory needed to join in two passes

Grace Hash Join Analysis

Overall analysis including both tables, assuming two-pass partitioning



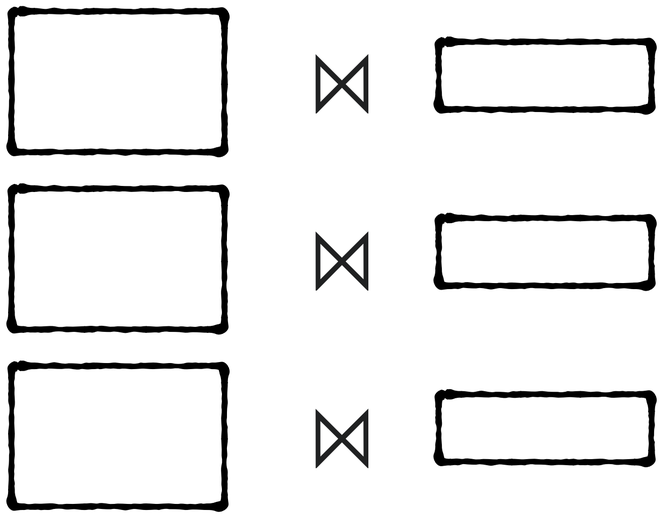
I/O cost? **O(|T1|/B + |T2|/B)**

CPU cost? **O(|T1|+|T2|)**

Memory cost? $O(\sqrt{\min(|T1|,|T2|) \cdot B})$

Grace Hash Join Analysis

Overall analysis including both tables, assuming two-pass partitioning



T1 **T2**

I/O cost? **O(|T1|/B + |T2|/B)**

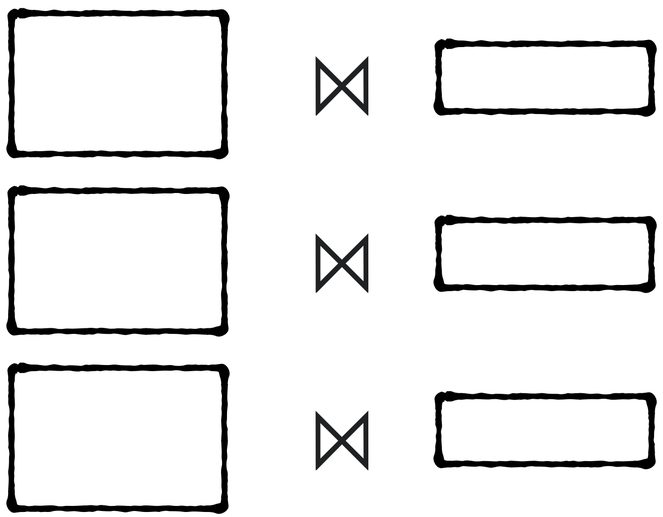
CPU cost? **O(|T1|+|T2|)**

Memory cost? $O(\sqrt{\min(|T1|,|T2|) \cdot B})$

Lower memory footprint and CPU cost than merge-sort join :)

Grace Hash Join Analysis

Overall analysis including both tables, assuming two-pass partitioning



I/O cost? **O(|T1|/B + |T2|/B)**

CPU cost? **O(|T1|+|T2|)**

Memory cost? $O(\sqrt{\min(|T1|,|T2|) \cdot B})$

Lower memory footprint and
CPU cost than merge-sort join :)

Sort join is still better if we have an order by clause

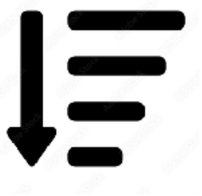
Query Operators

Selection

Projection

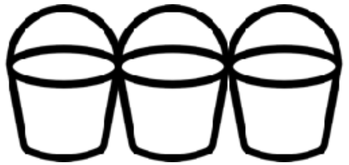
Join

Order by



Distinct

Group by



Query Operators

Cardinality Estimation

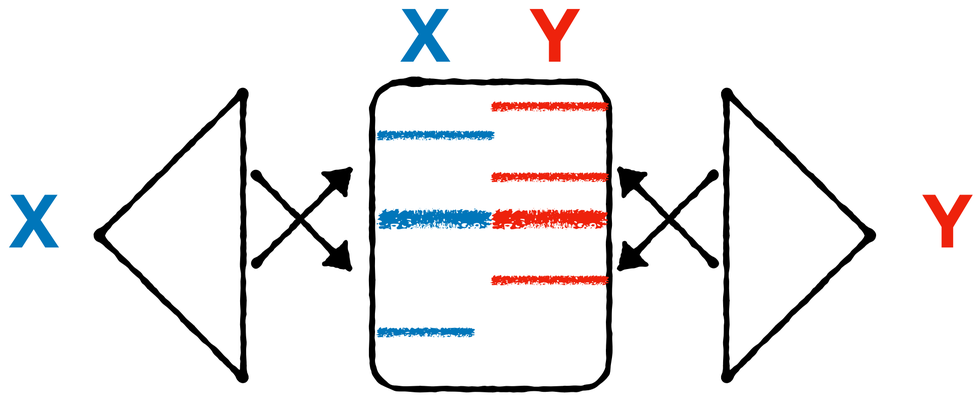
Query Optimization

Query
Operators

Cardinality
Estimation

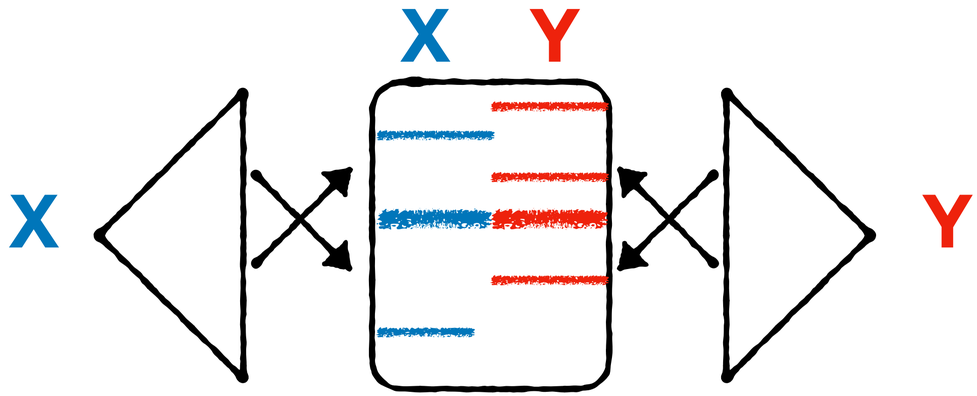
Query
Optimization

Cardinality Estimation



Select * from … where **X = “i” and Y=“j”**

Cardinality Estimation



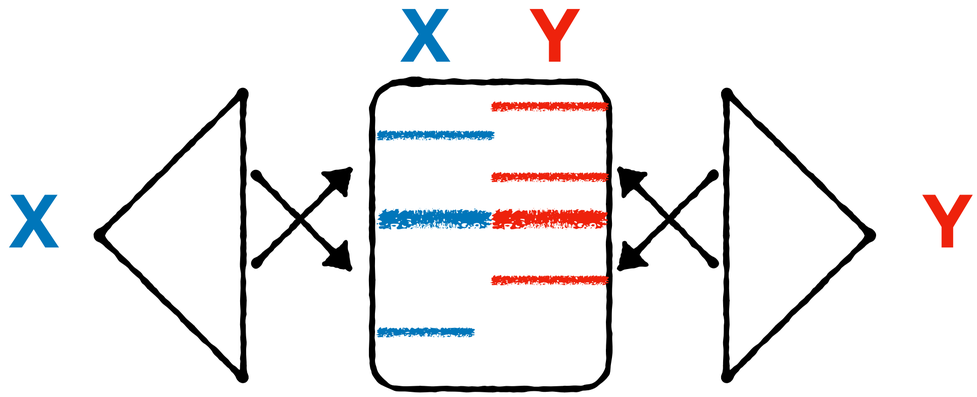
Select * from … where **X = “i” and Y=“j”**

Algo 1: **Search index Y** $\log_B(N) + |Y_j|$ I/O

Algo 2: **Search index X** $\log_B(N) + |X_i|$ I/O

Algo 3: **Search both** $2 \cdot \log_B(N) + |X_i|/B + |Y_j|/B + |X_i \cap Y_j|$ I/O

Cardinality Estimation

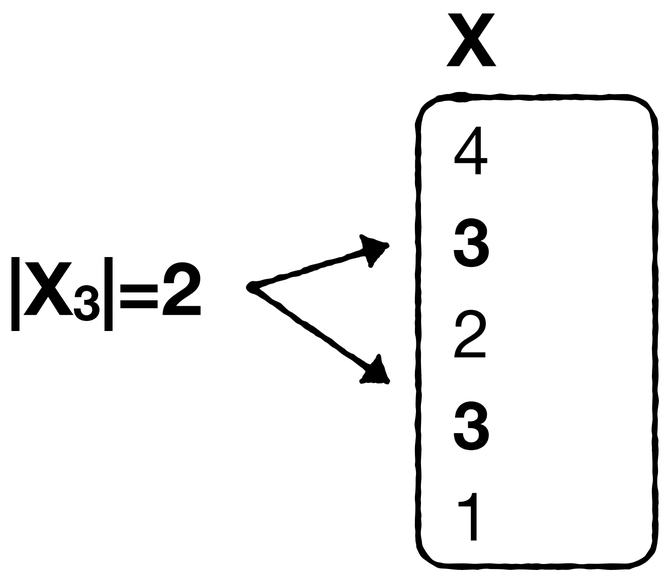


Select * from … where **X = “i” and Y=“j”**

Algo 1:	Search index Y	$\log_B(N) + \lvert Y_j \rvert$ I/O
Algo 2:	Search index X	$\log_B(N) + \lvert X_i \rvert$ I/O
Algo 3:	Search both	$2 \cdot \log_B(N) + \lvert X_i \rvert/B + \lvert Y_j \rvert/B + \lvert X_i \cap Y_j \rvert$ I/O

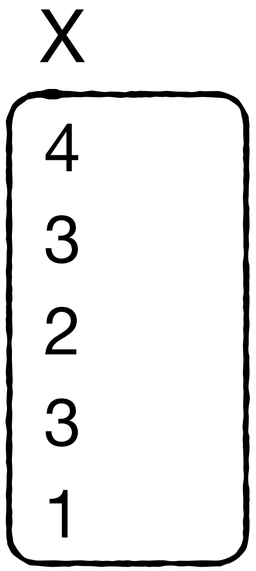
Determining best plan requires estimating $|X_i|$, $|Y_j|$ and $|X_i \cap Y_j|$

Cardinality Estimation



How many rows have a particular value X_i?

Cardinality Estimation



$|X_3|=2$

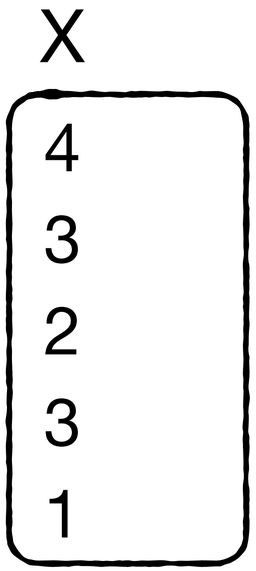
How many rows have a particular value X_i?

Approach 1: Estimate X_i as N / |X|

 # rows

 # unique X values

Cardinality Estimation



$|X_3|=2$

$|X|=4$

$N=5$

How many rows have a particular value X_i?

Approach 1: Estimate X_i as N / |X| = 5/4

 # rows

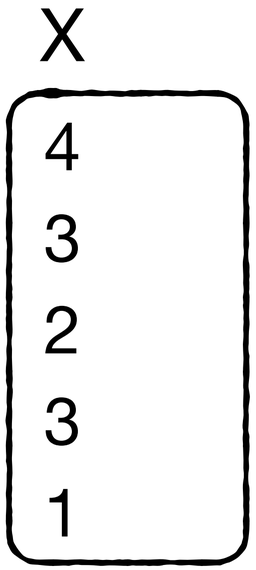
 # unique X values

Cardinality Estimation

$|X_3|=2$

$|X|=4$

$N=5$

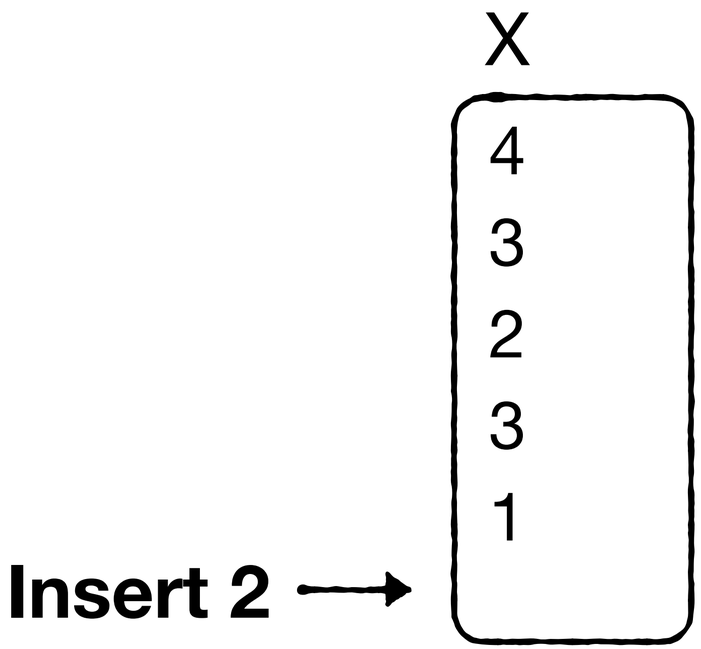


How many rows have a particular value X_i?

Approach 1: Estimate X_i as $N / |X| = 5/4$

Problem: estimating |X| is non-trivial as well!

Cardinality Estimation



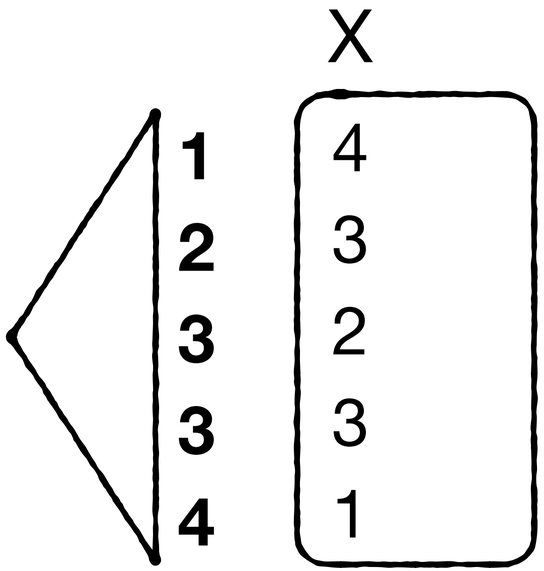
How many rows have a particular value X_i?

Approach 1: Estimate X_i as N / |X| = 5/4

Problem: estimating |X| is non-trivial as well!

e.g., during insertion, we do not know if new value is unique or not

Cardinality Estimation

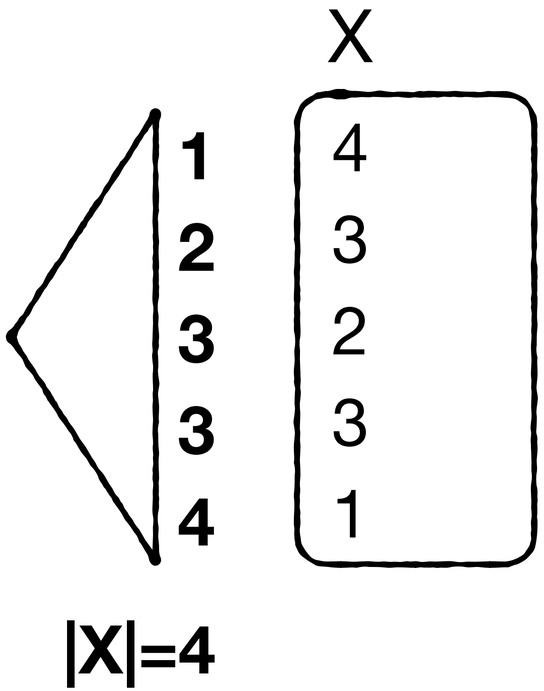


How many rows have a particular value X_i?

Approach 1: Estimate X_i as N / |X| = 5/4

If we have an index on X, it becomes easy to tell if any new insert/delete/update adds or removes a unique value

Cardinality Estimation



|X|=4

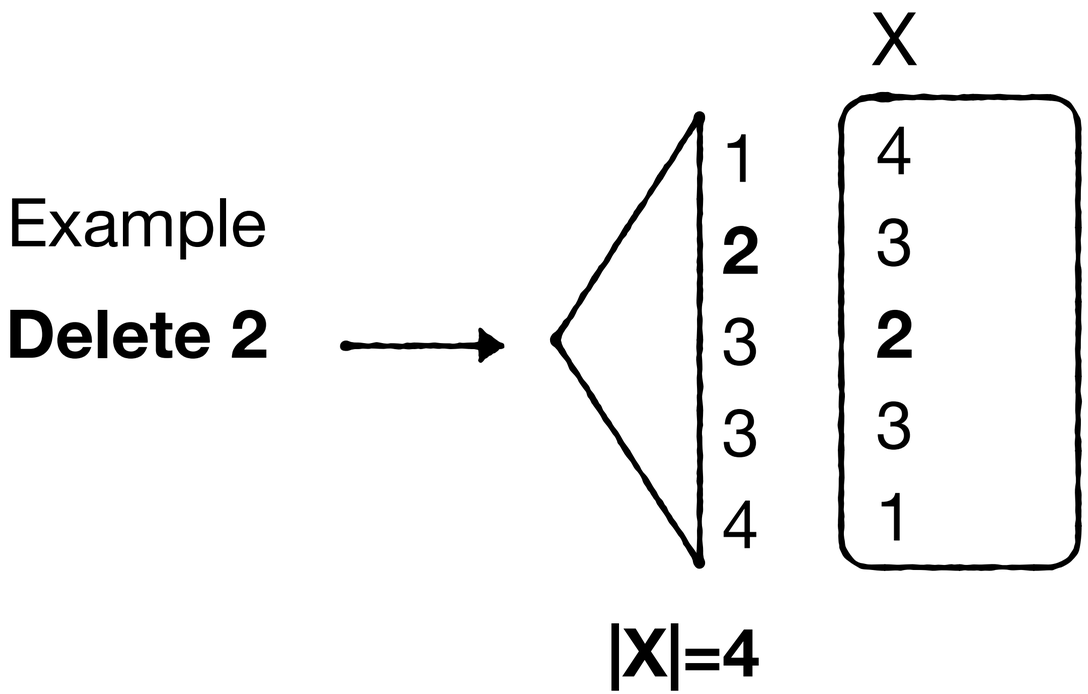
How many rows have a particular value X_i?

Approach 1: Estimate X_i as N / |X| = 5/4

If we have an index on X, it becomes easy to tell if any new insert/delete/update adds or removes a unique value

So we can maintain a cardinality counter for each index

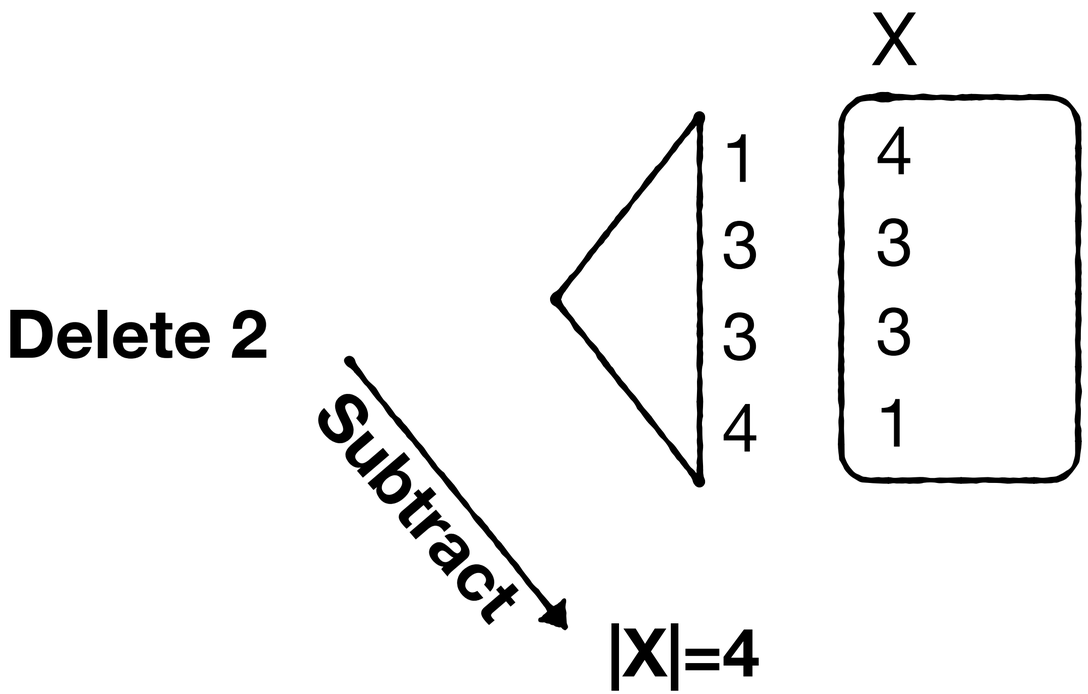
Cardinality Estimation



If we have an index on X, it becomes easy to tell if any new insert/delete/update adds or removes a unique value

So we can maintain a cardinality counter for each index

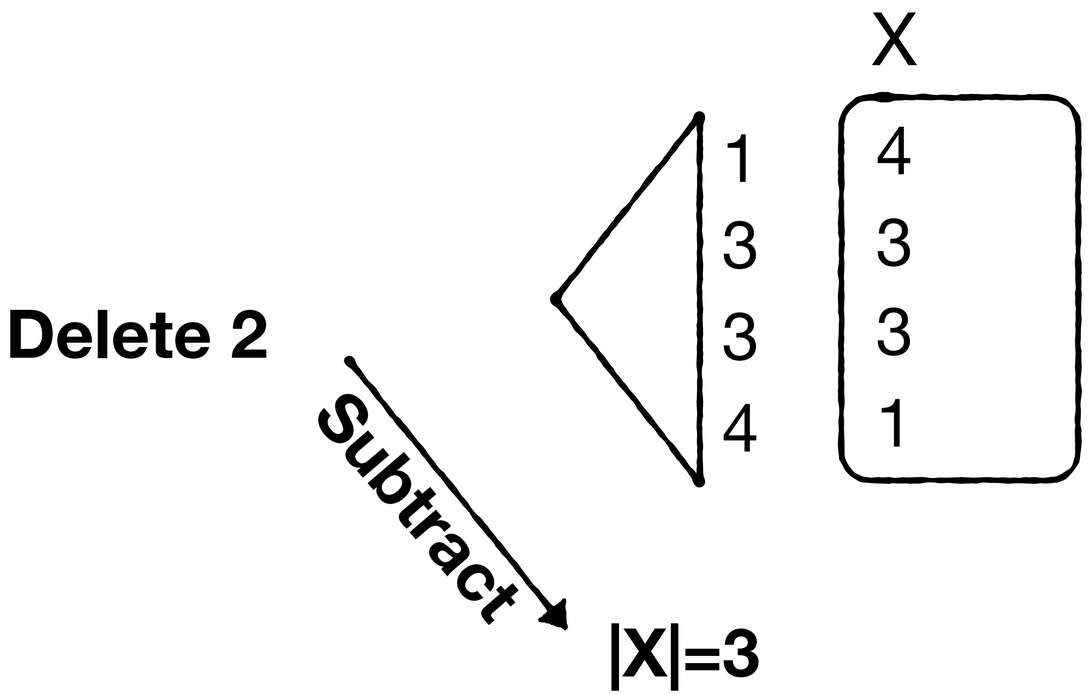
Cardinality Estimation



If we have an index on X, it becomes easy to tell if any new insert/delete/update adds or removes a unique value

So we can maintain a cardinality counter for each index

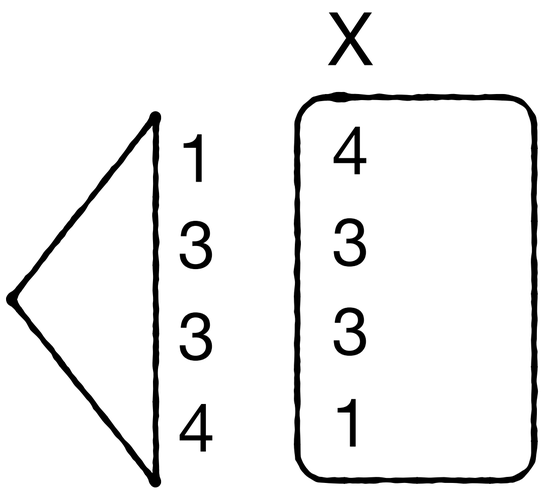
Cardinality Estimation



If we have an index on X, it becomes easy to tell if any new insert/delete/update adds or removes a unique value

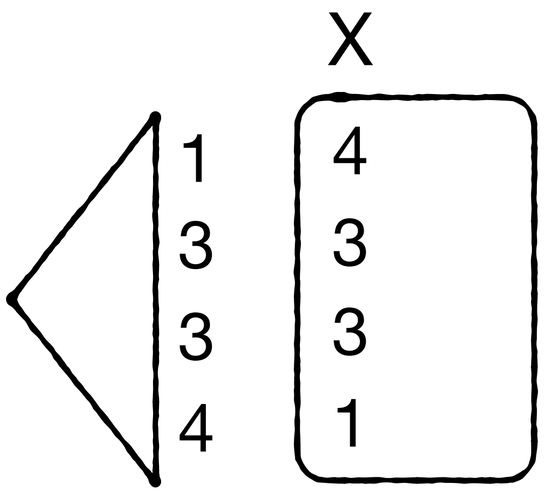
So we can maintain a cardinality counter for each index

Cardinality Estimation



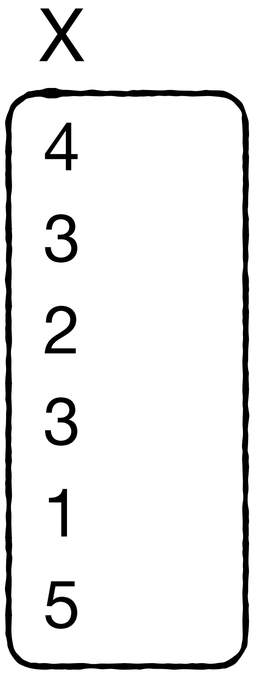
What if there is no index?

Cardinality Estimation



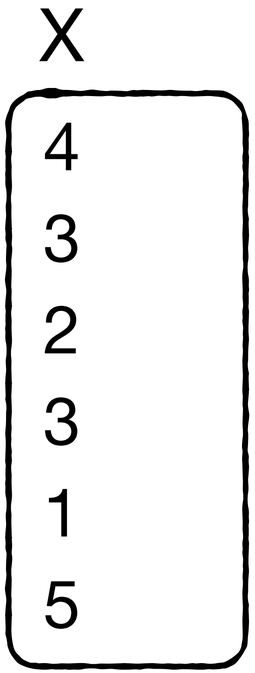
What if there is no index?

periodically scan and count #unique entries



estimate any $|X_i|$ as $N/|X|$

Assumes counts of all values are uniform

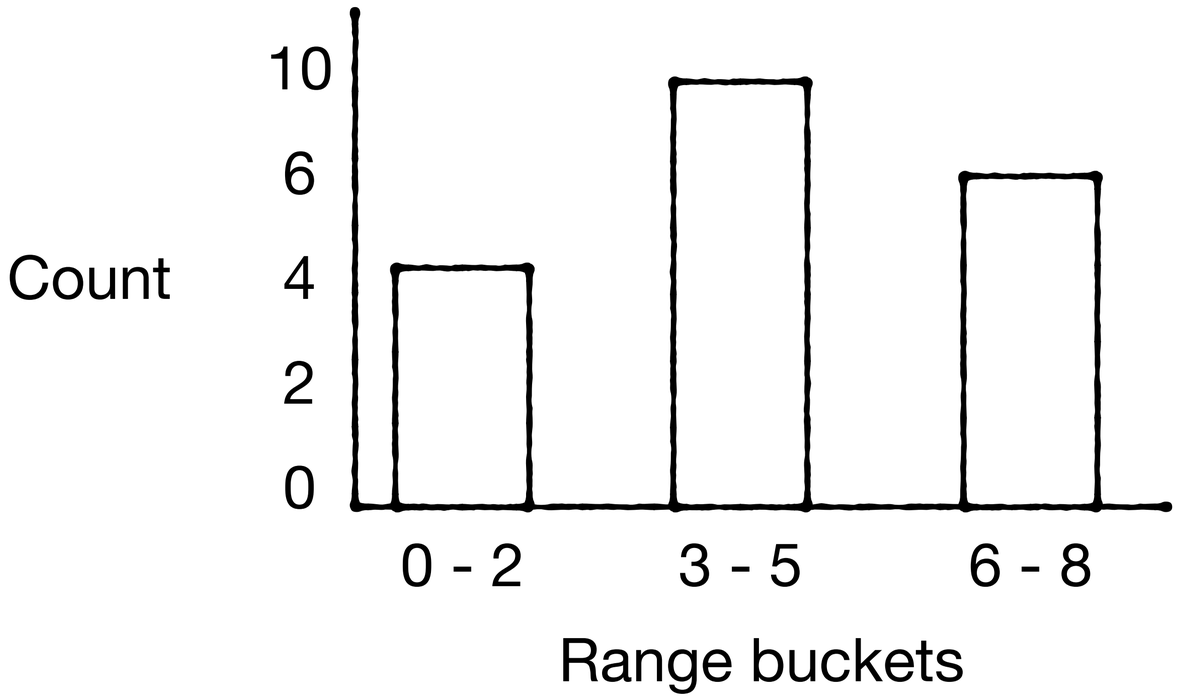


estimate any $|X_i|$ as $N/|X|$

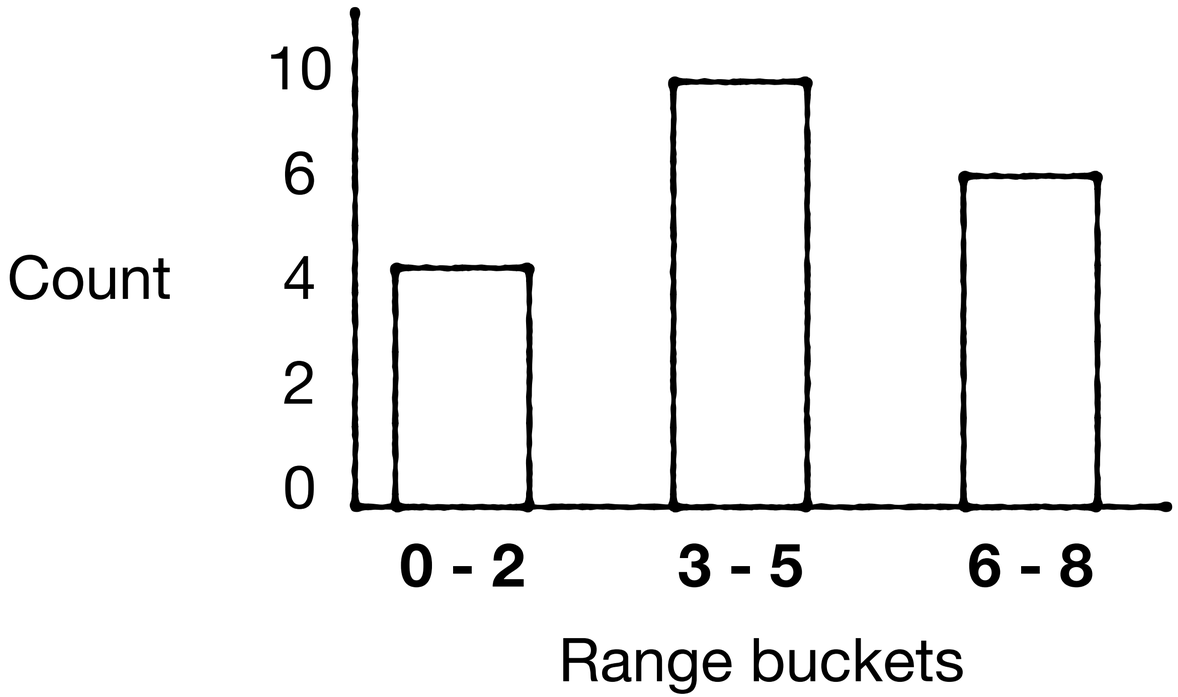
Assumes counts of all values are uniform

Can we do better?

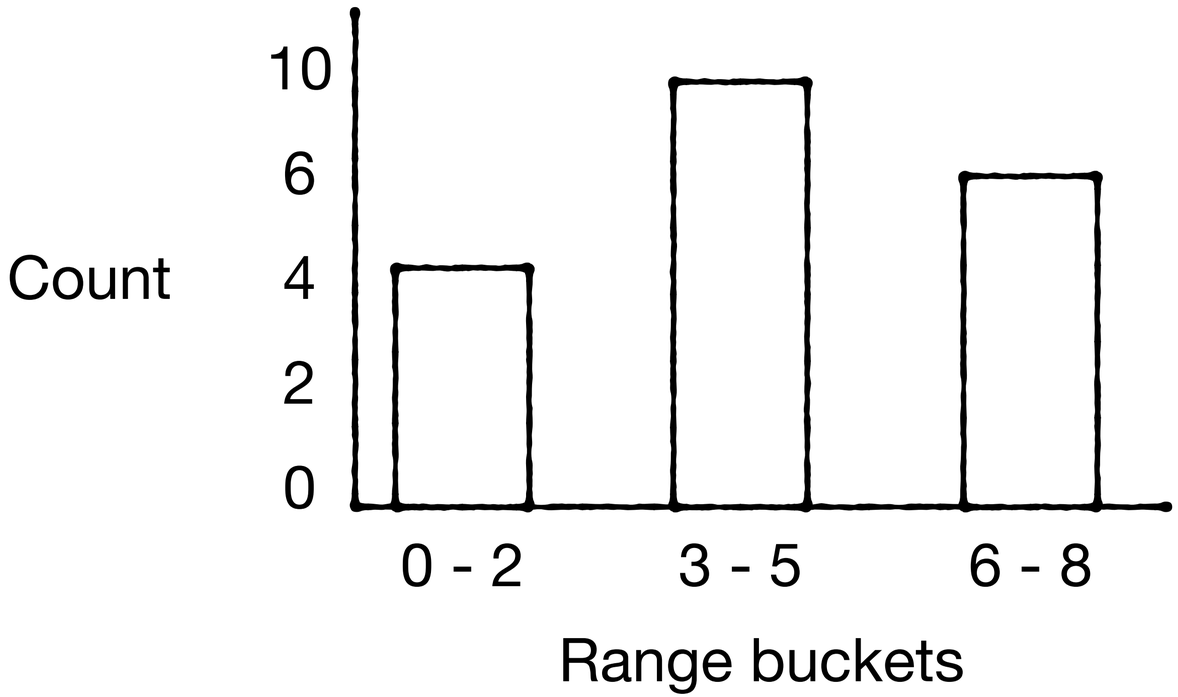
Histograms



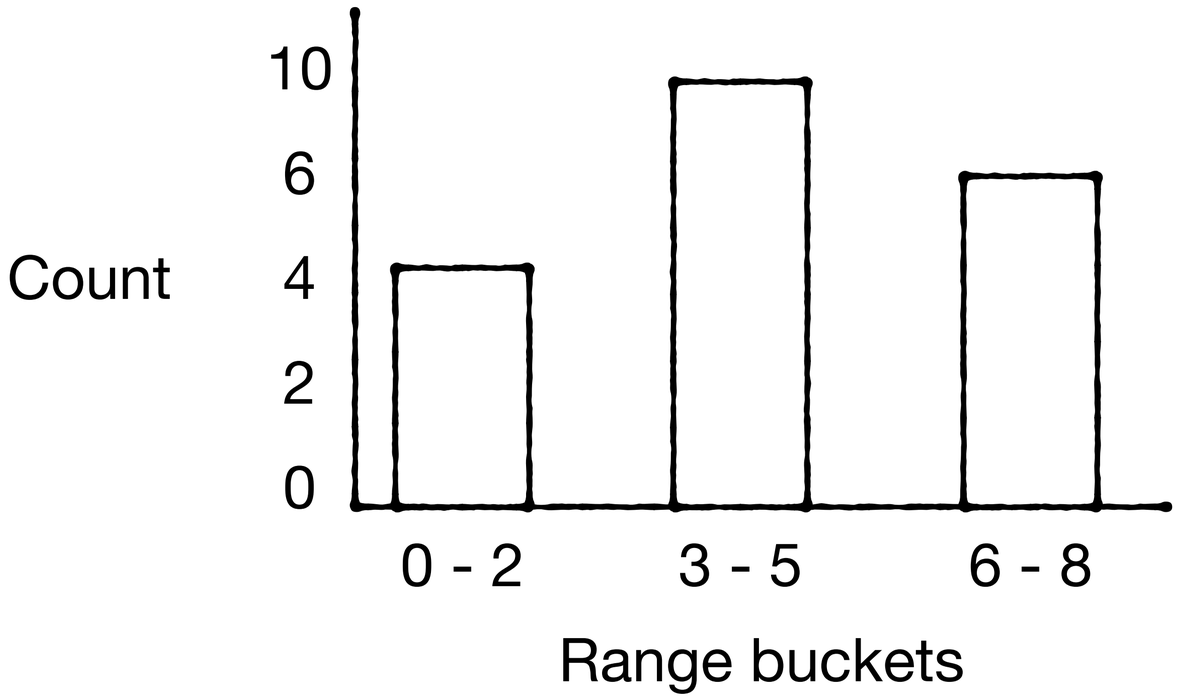
Histograms



Histograms



Estimate $|X_i|$ **as bucket count / bucket range**



Estimate $|X_i|$ **as bucket count / bucket range**

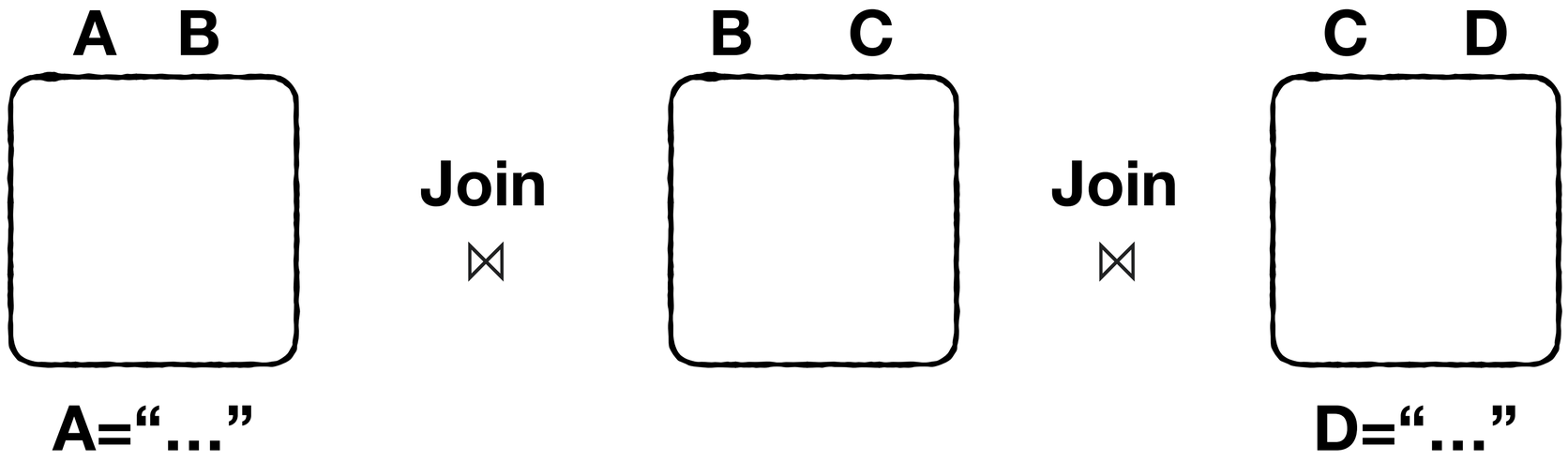
Estimate $|X_4|$ **= as 10/3=3.3**

Query
Operators

Cardinality
Estimation

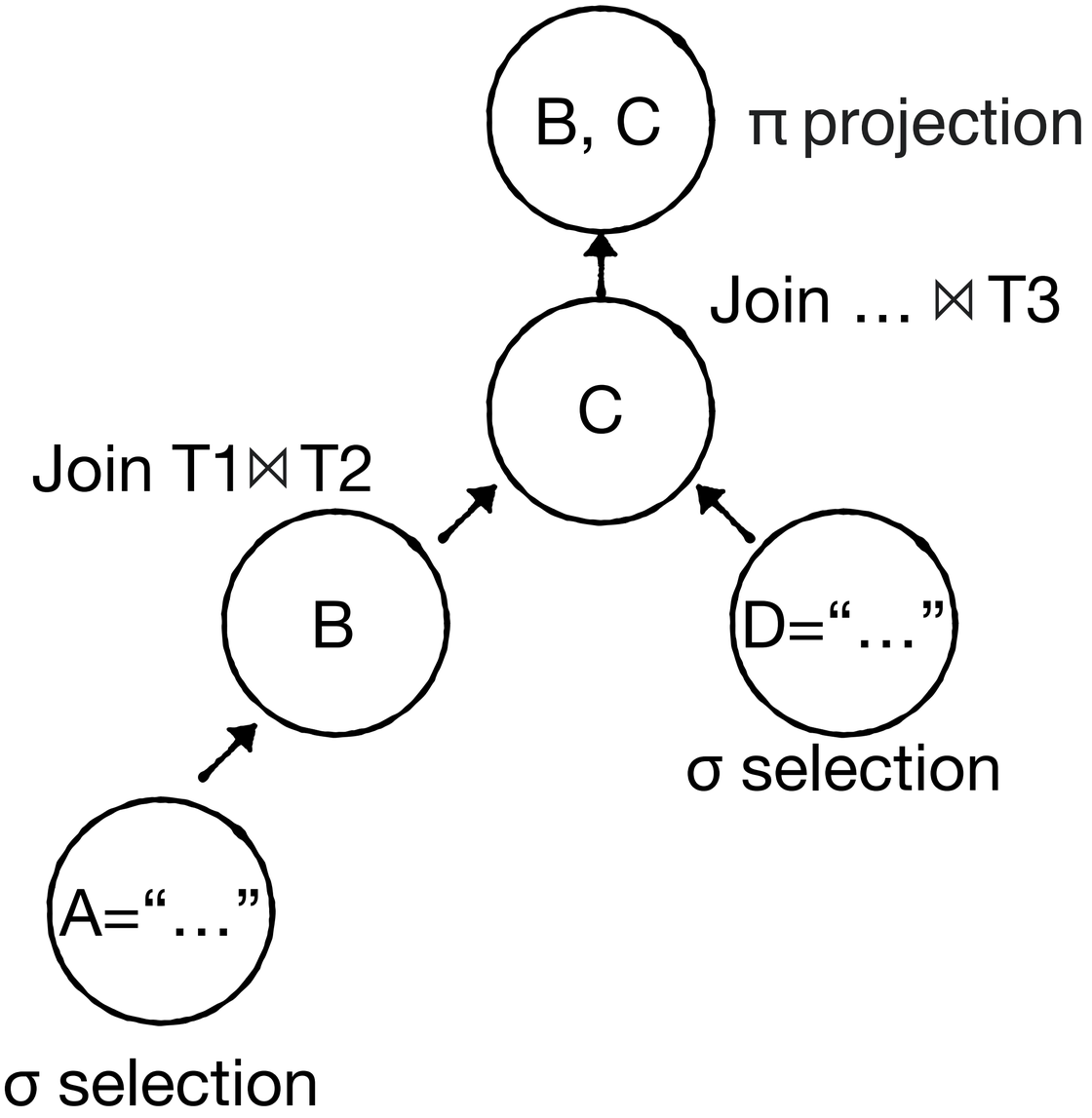
Query
Optimization

Query Optimization

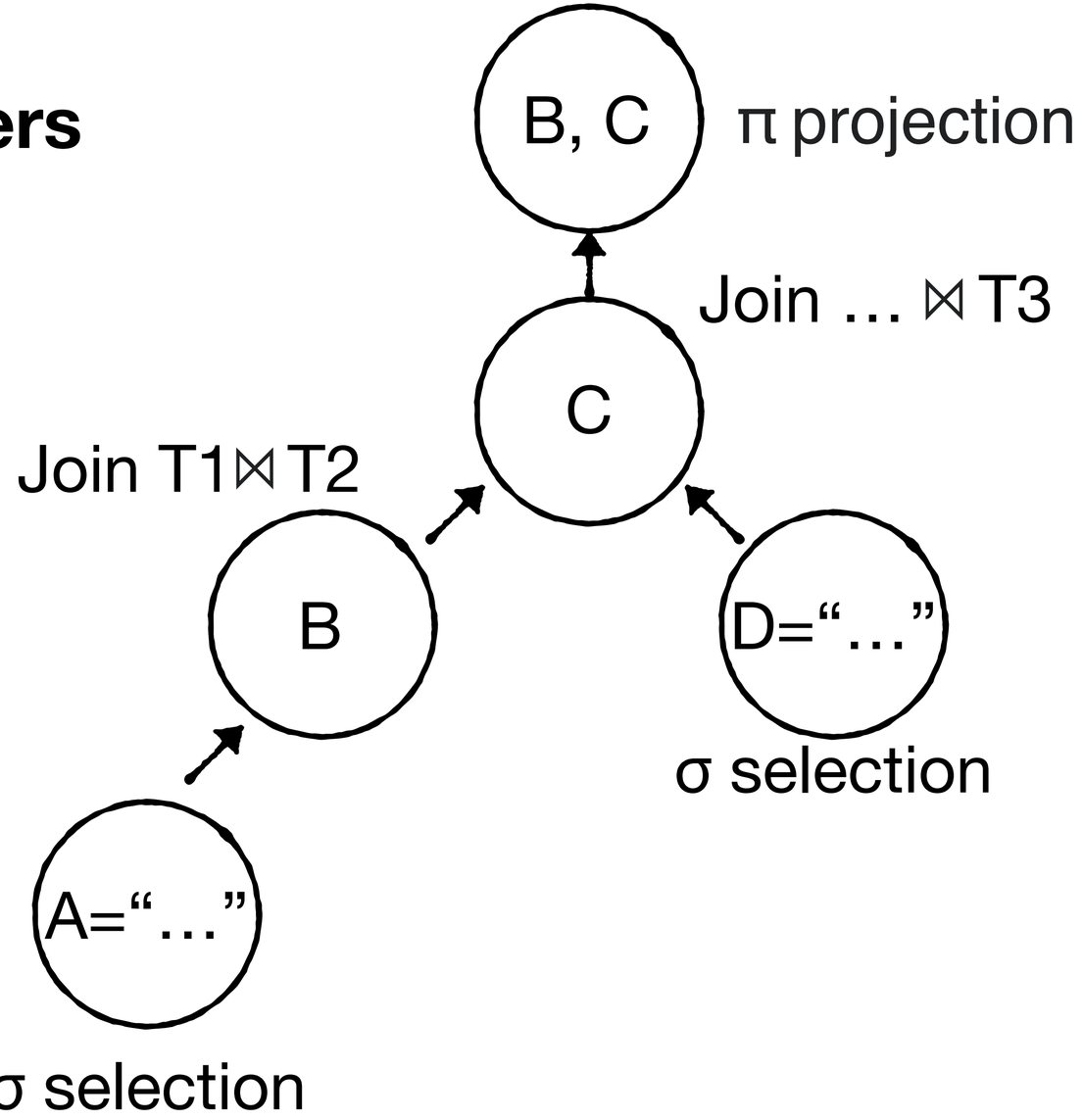


Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"

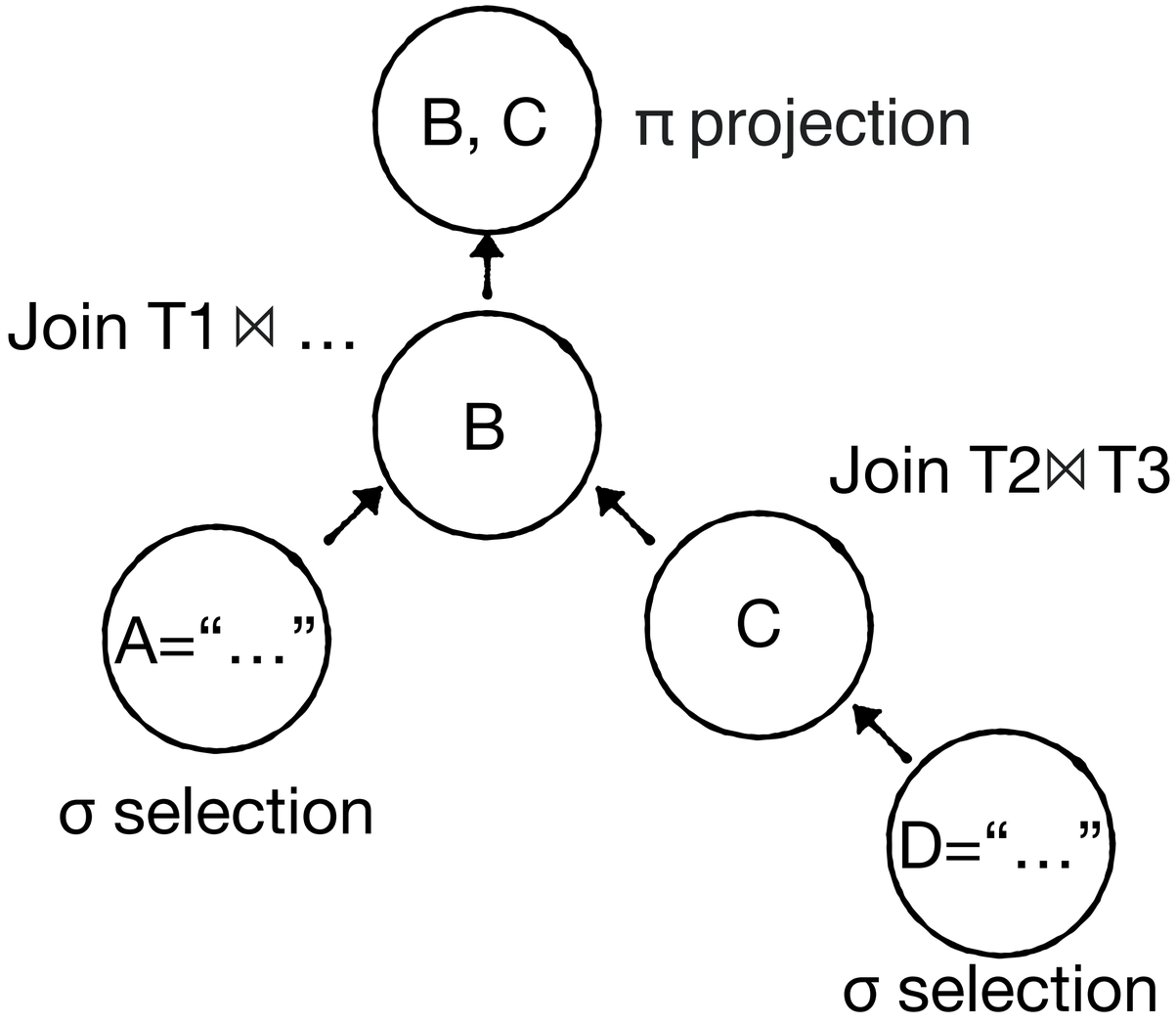
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"



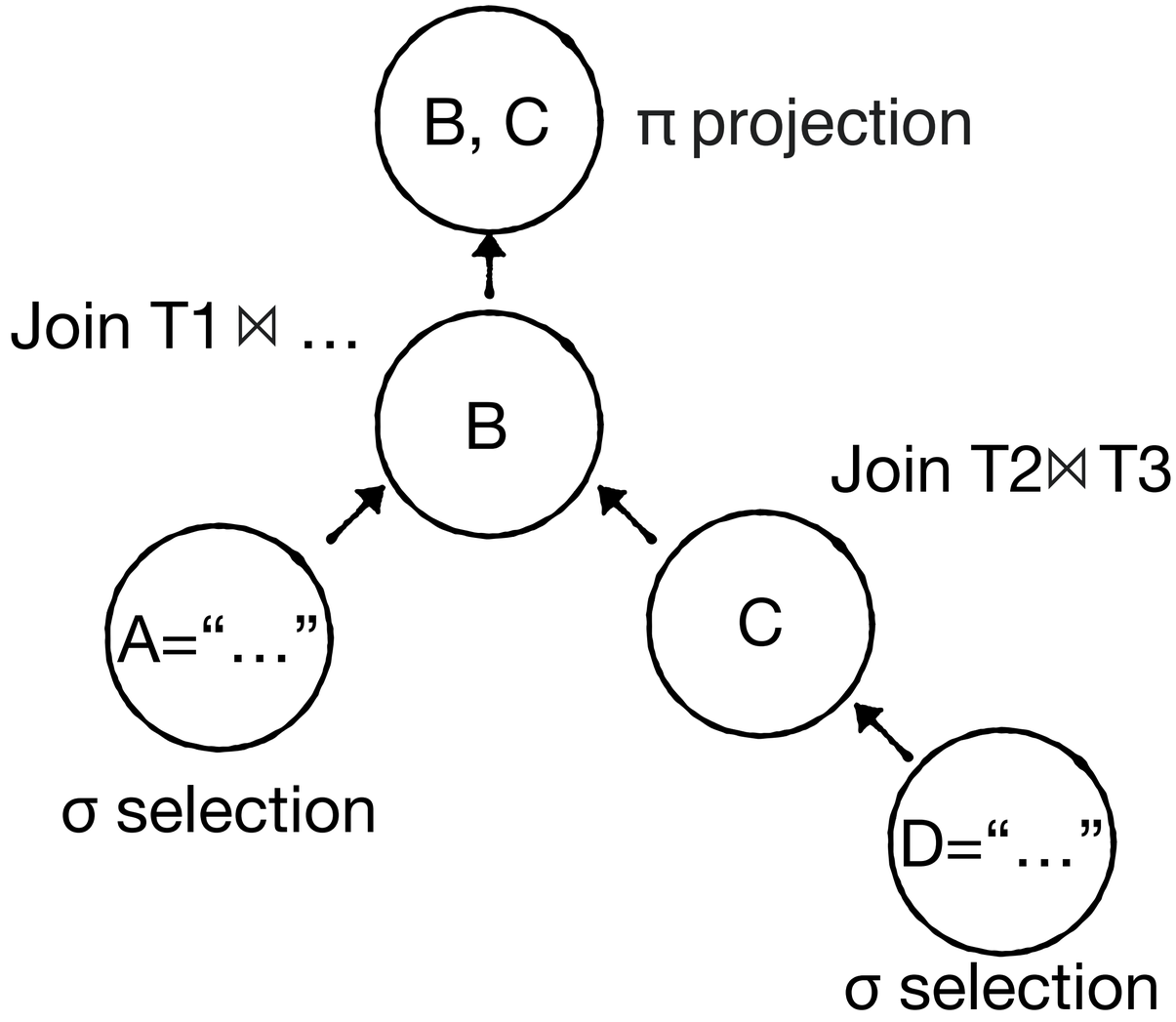
Can join tables in different orders



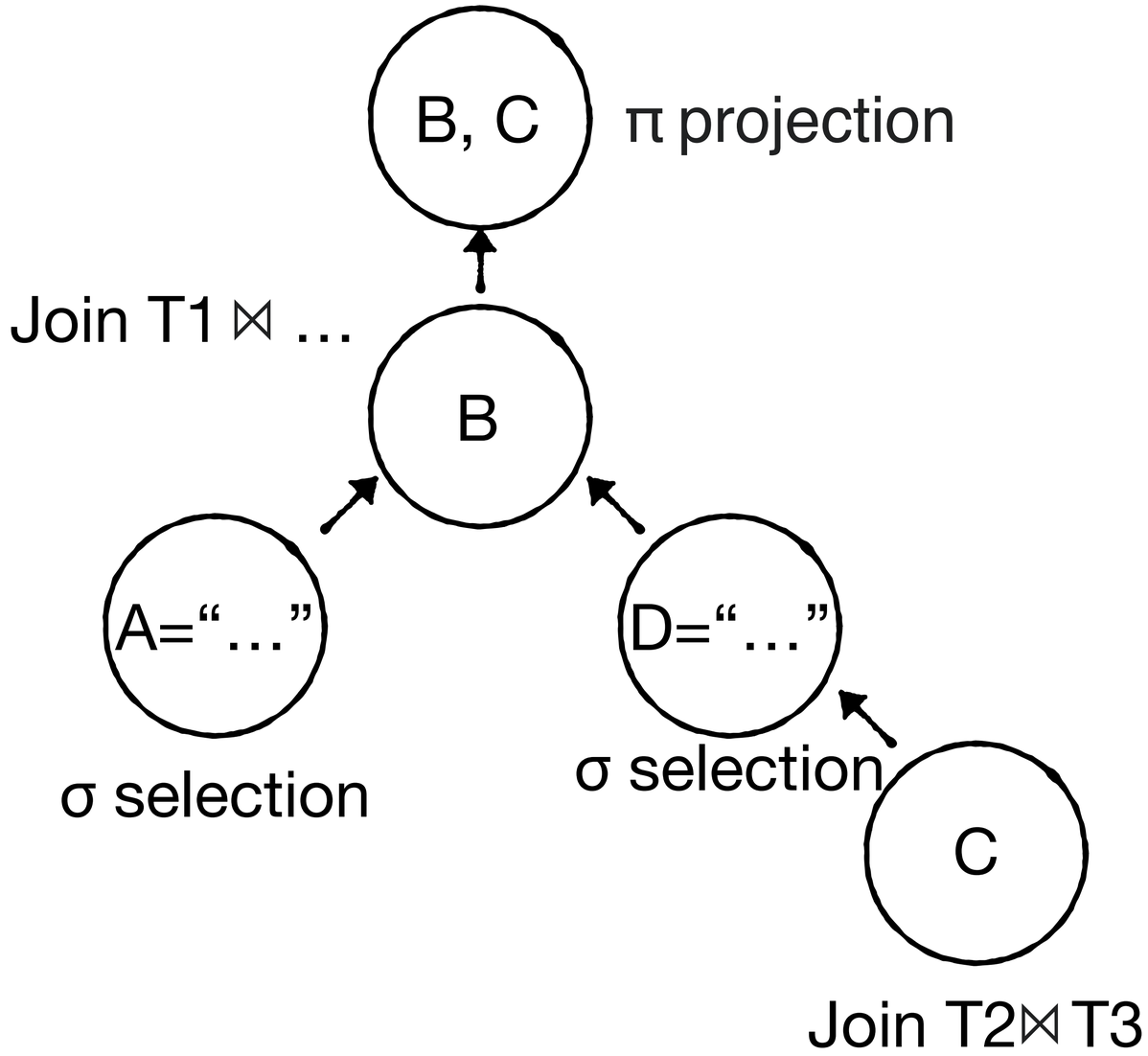
Can join tables in different orders



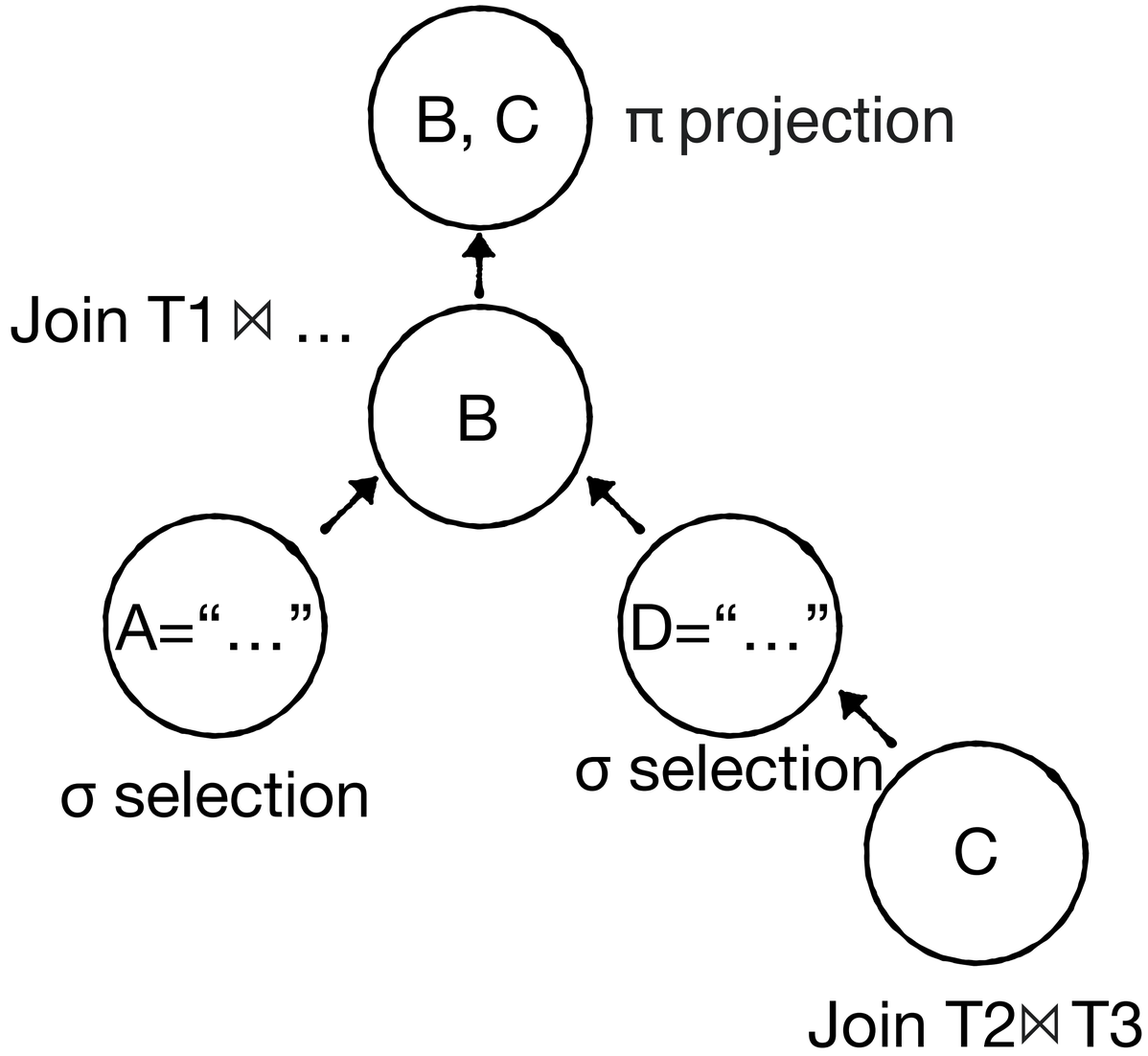
Can select in different orders

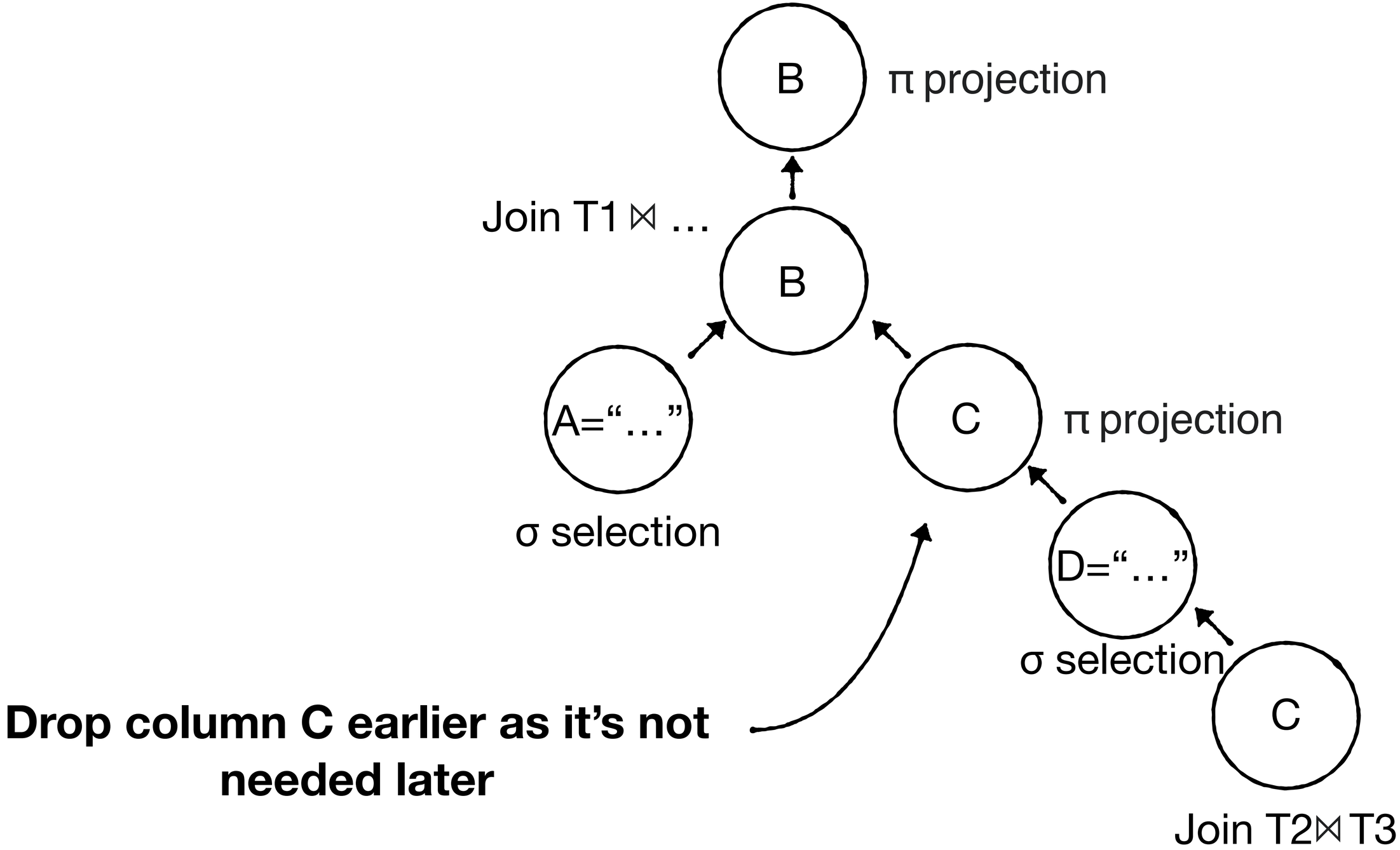


Can select in different orders



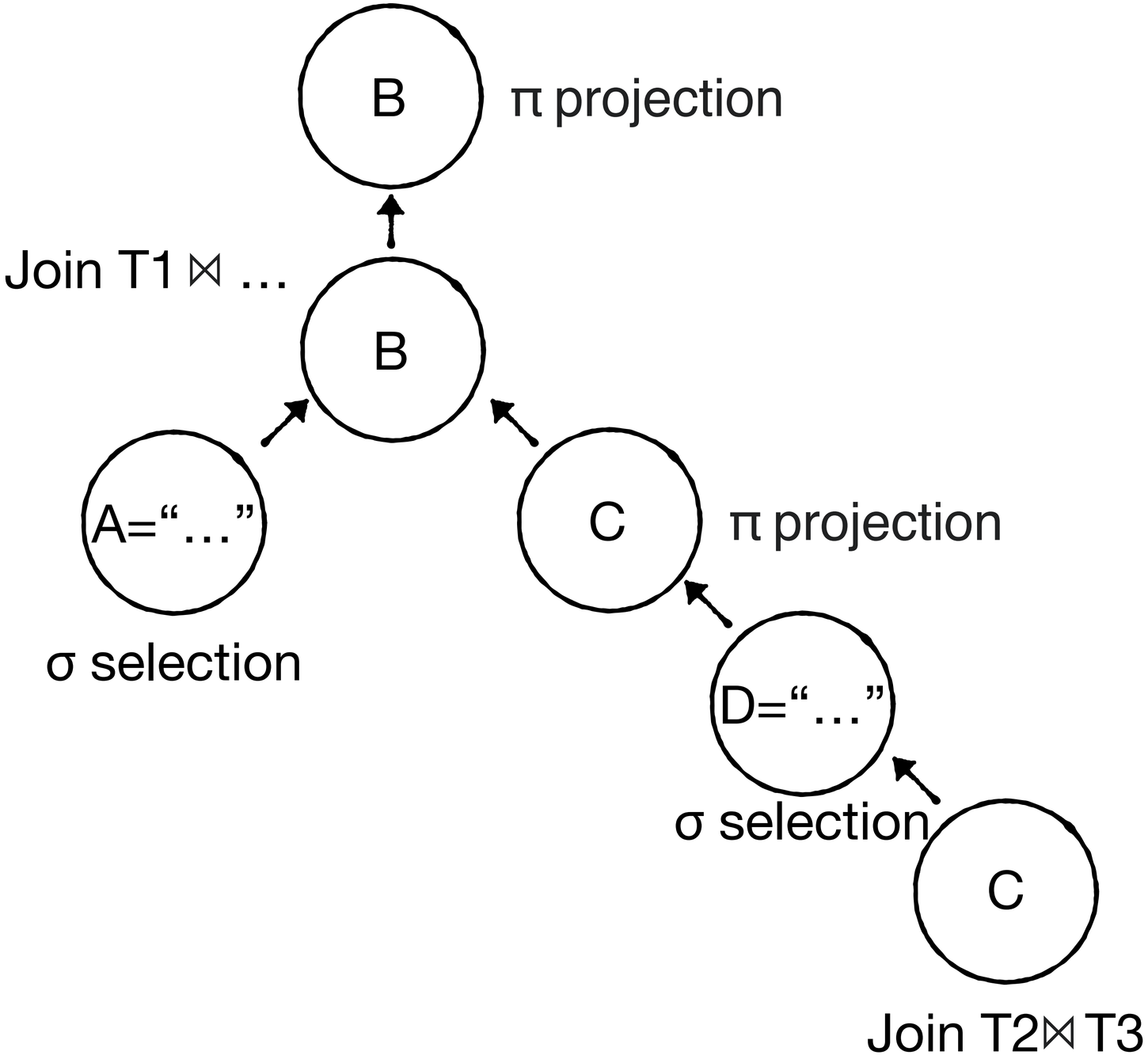
Can project columns in different orders as long as selections or joins do not rely on them



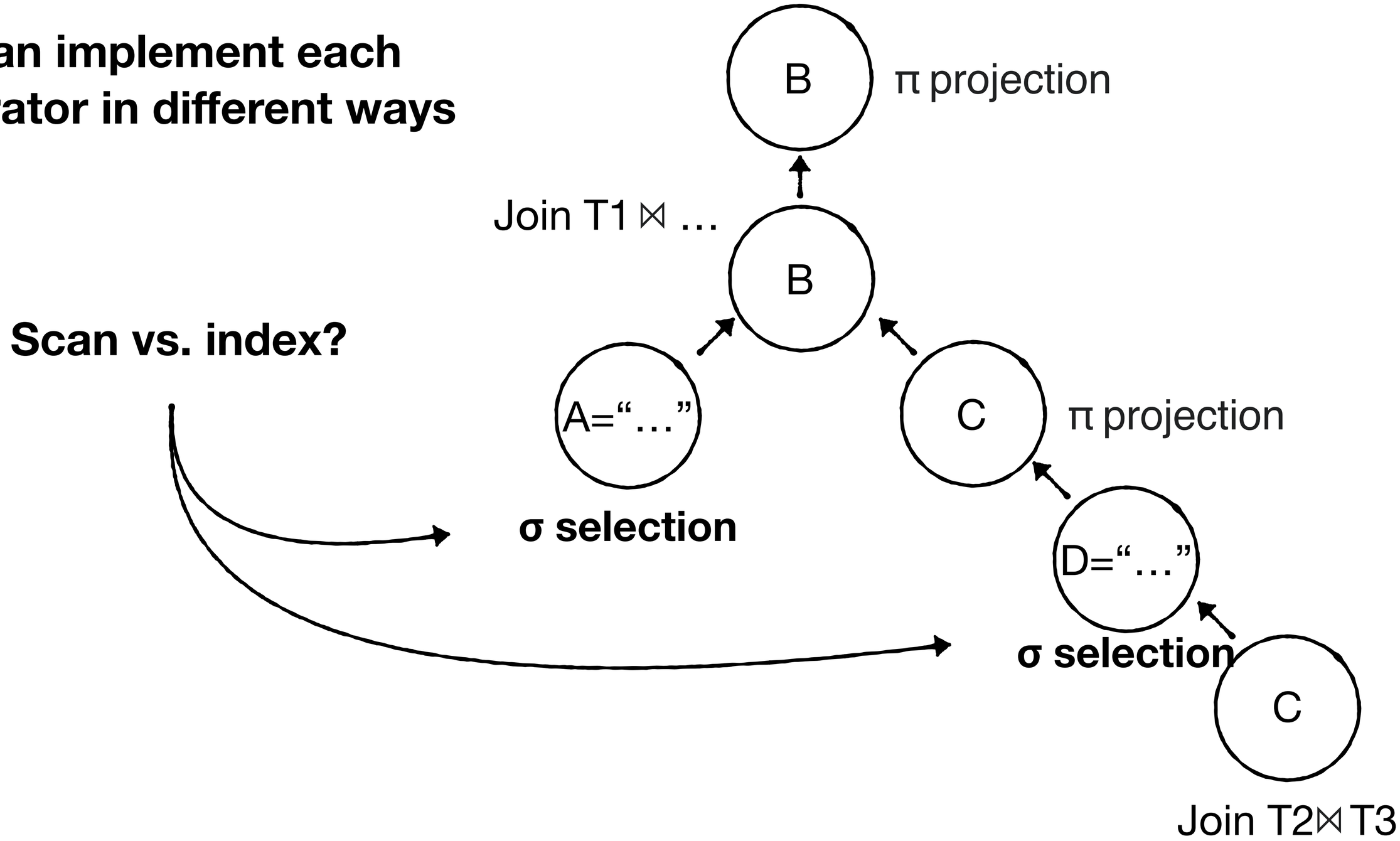


Drop column C earlier as it’s not needed later

Can implement each operator in different ways

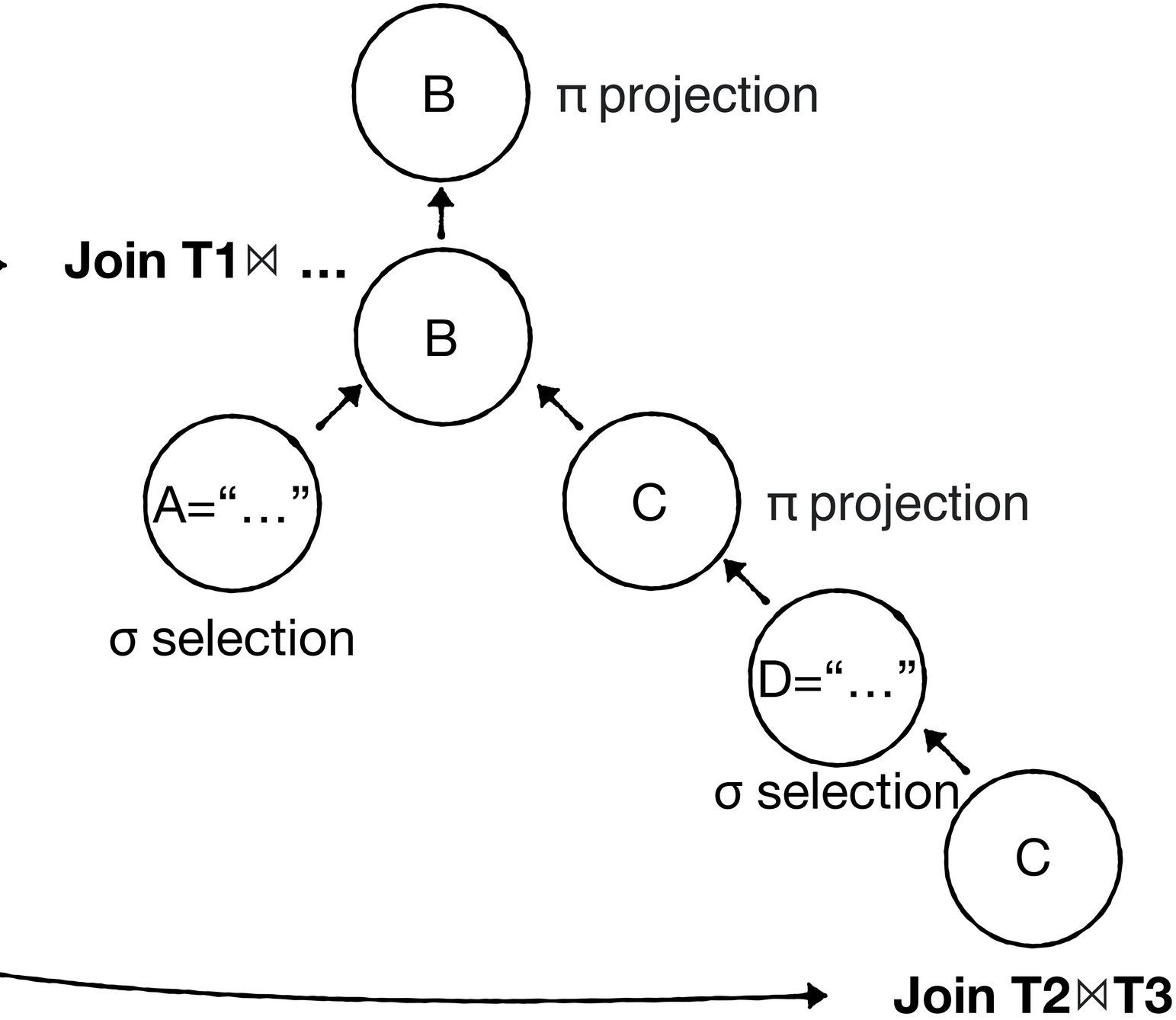


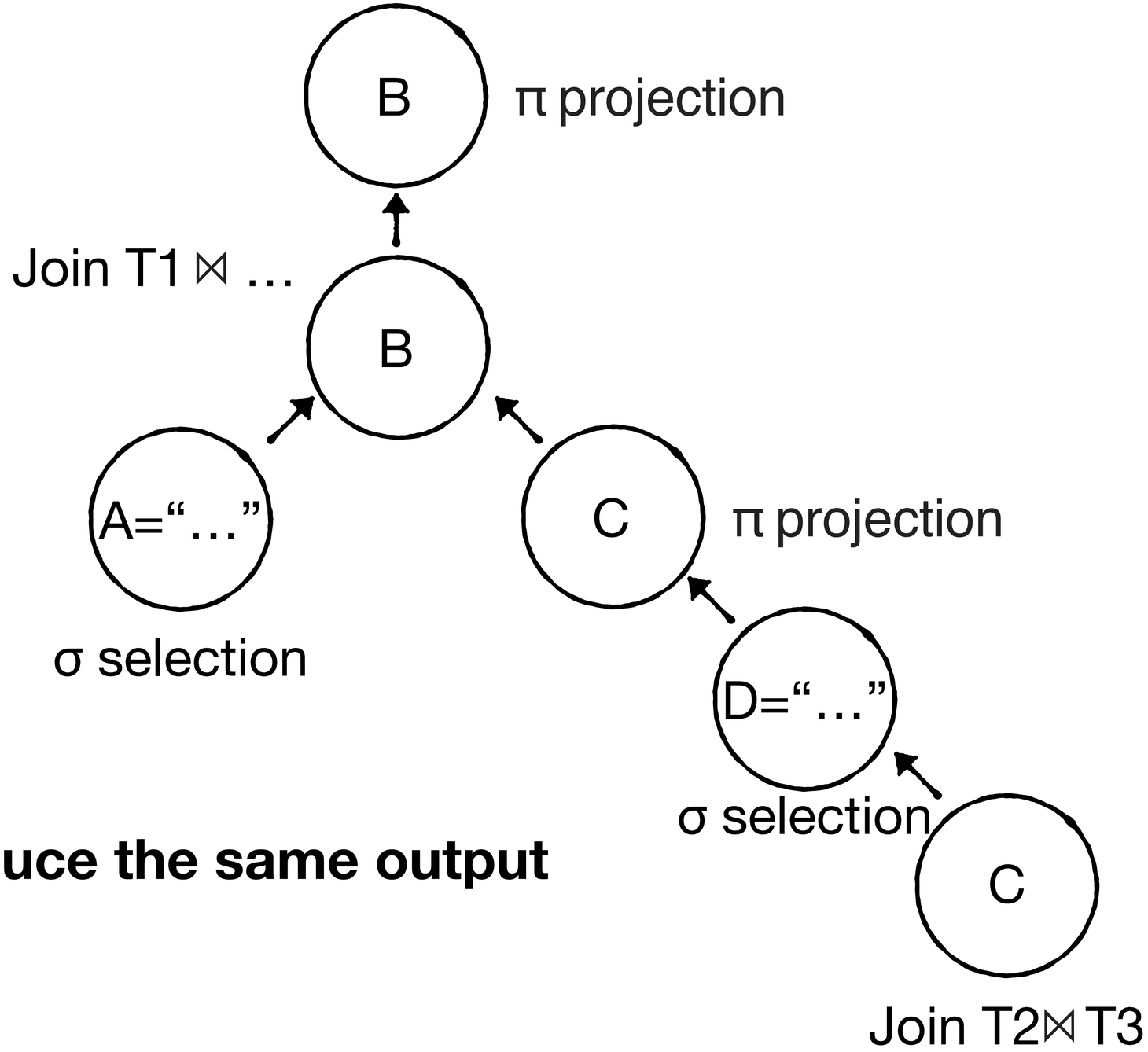
Can implement each operator in different ways



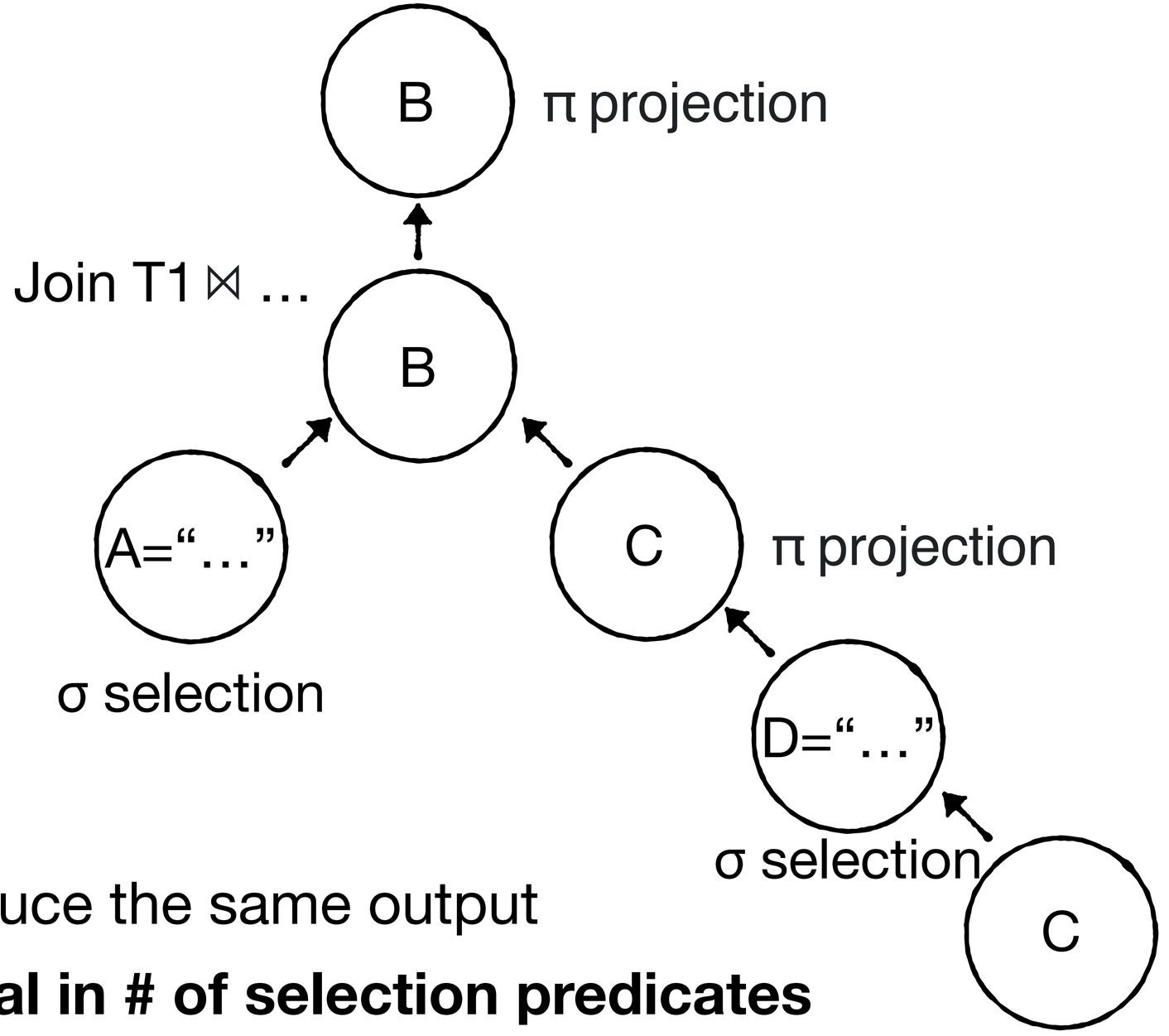
Can implement each operator in different ways

Block nested loop?
Sort-merge?
Grace Hash?
Index?





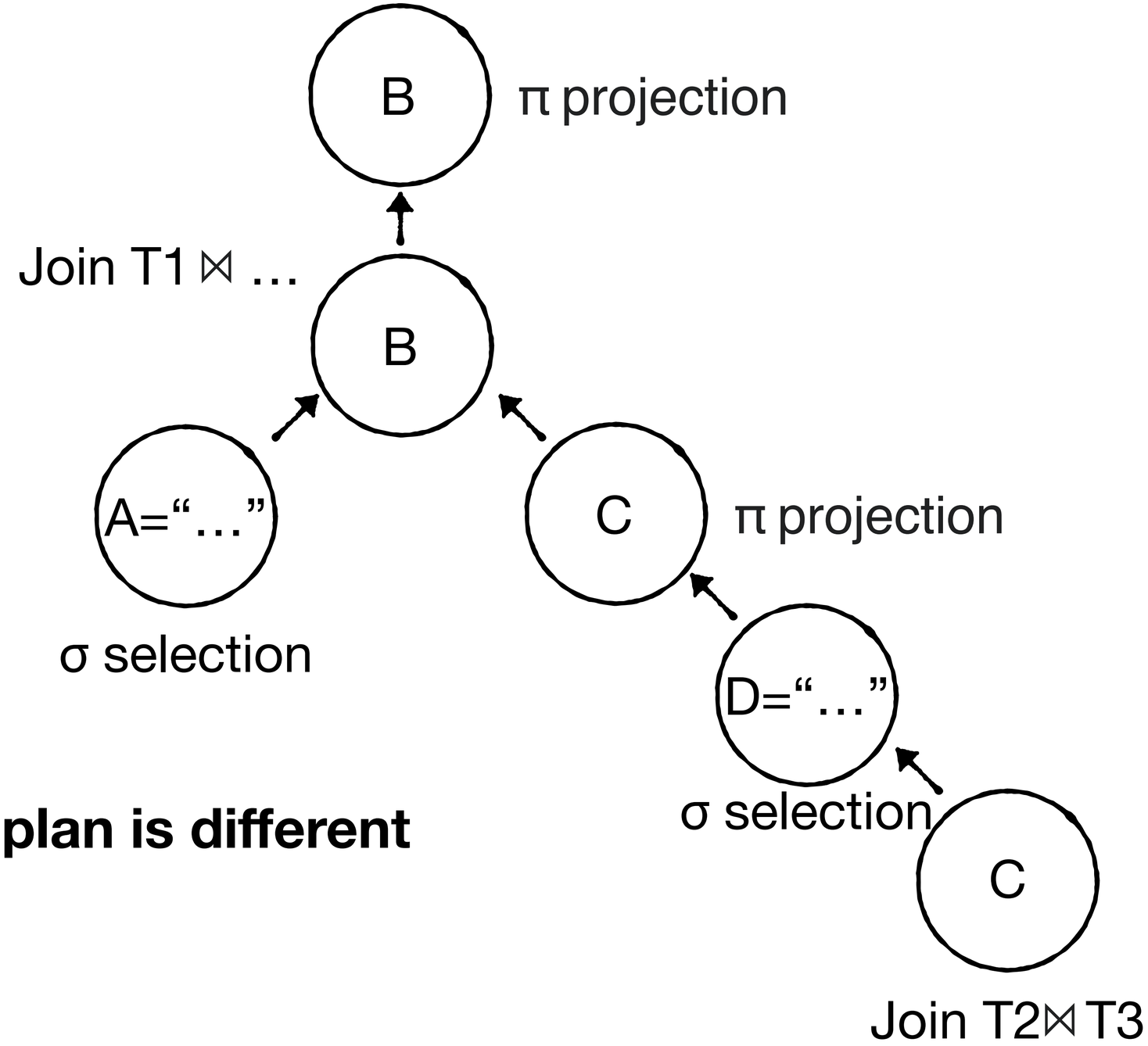
These query plans all produce the same output



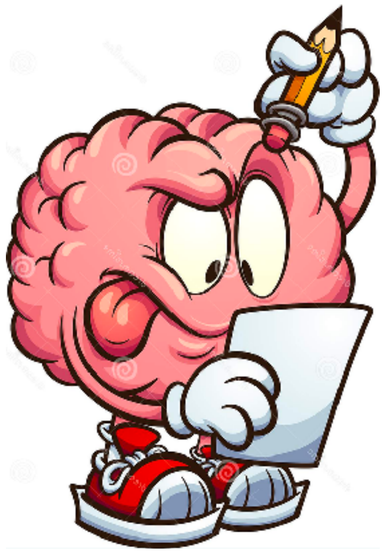
These query plans all produce the same output

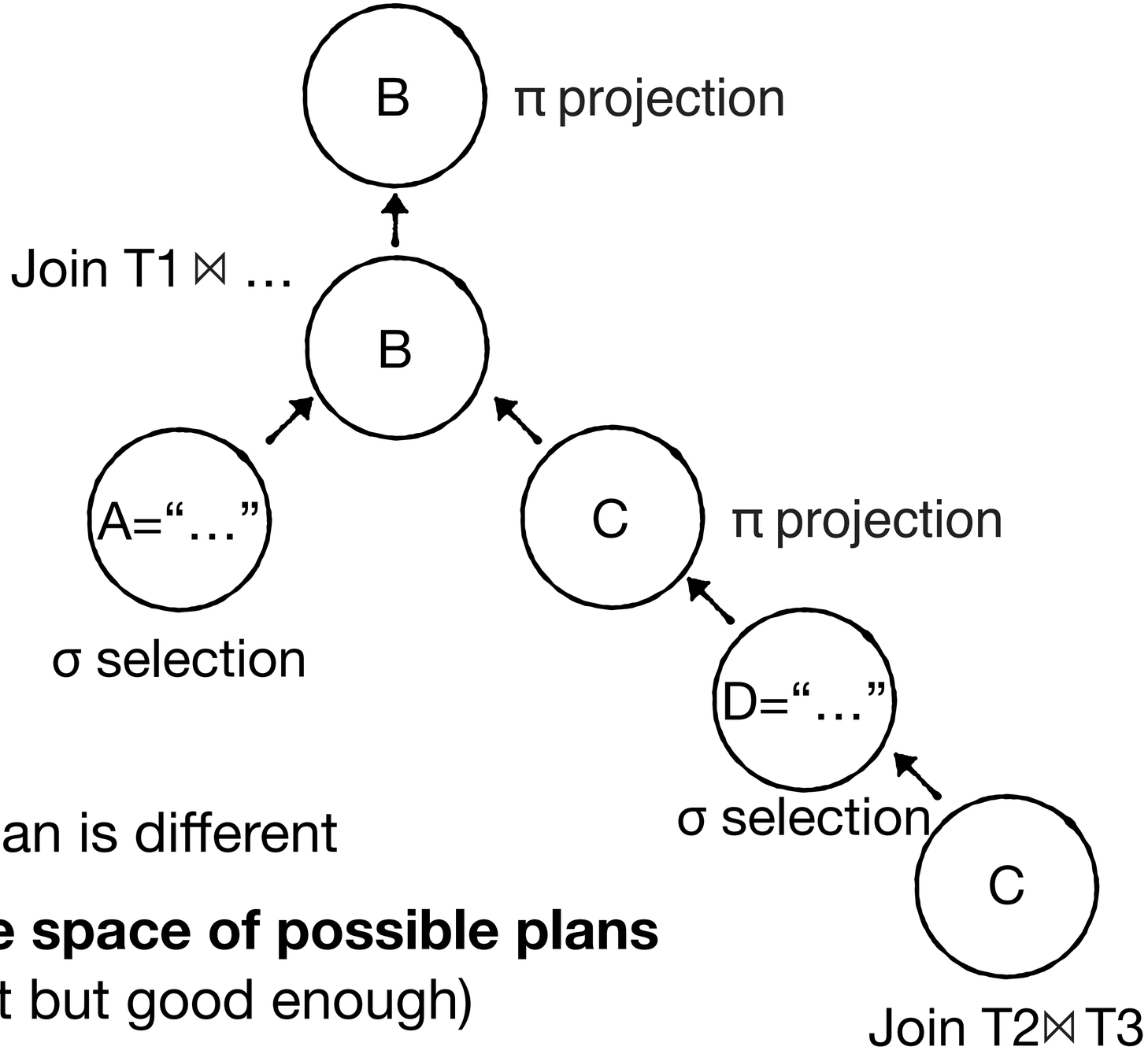
 # possible plans is exponential in # of selection predicates

(you can permute them)



However, the cost of each plan is different





However, the cost of each plan is different

The optimizer searches the space of possible plans to find a good one (not best but good enough)

Thanks!