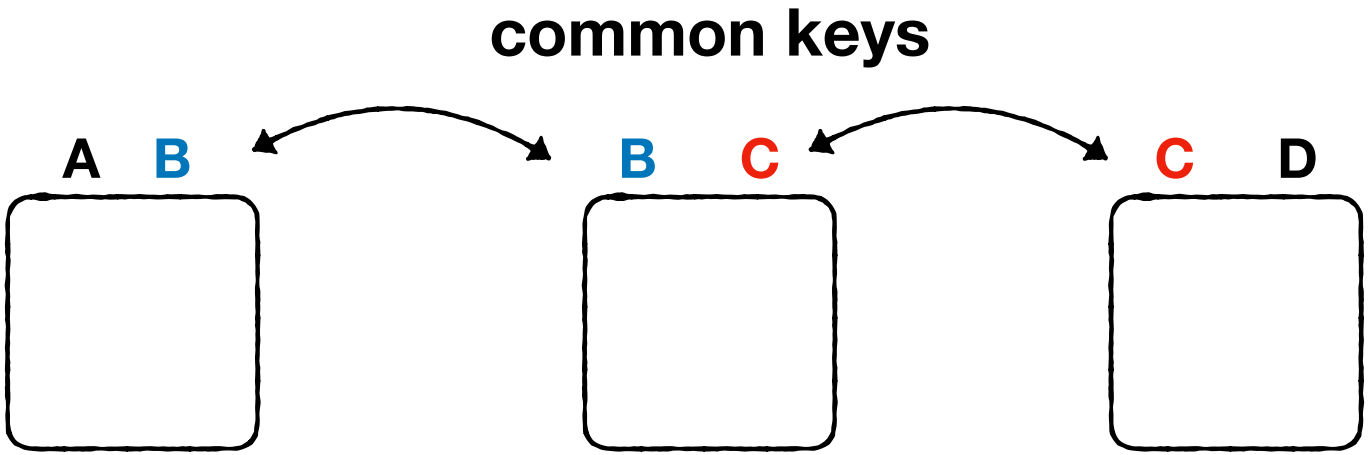
Query Evalaution

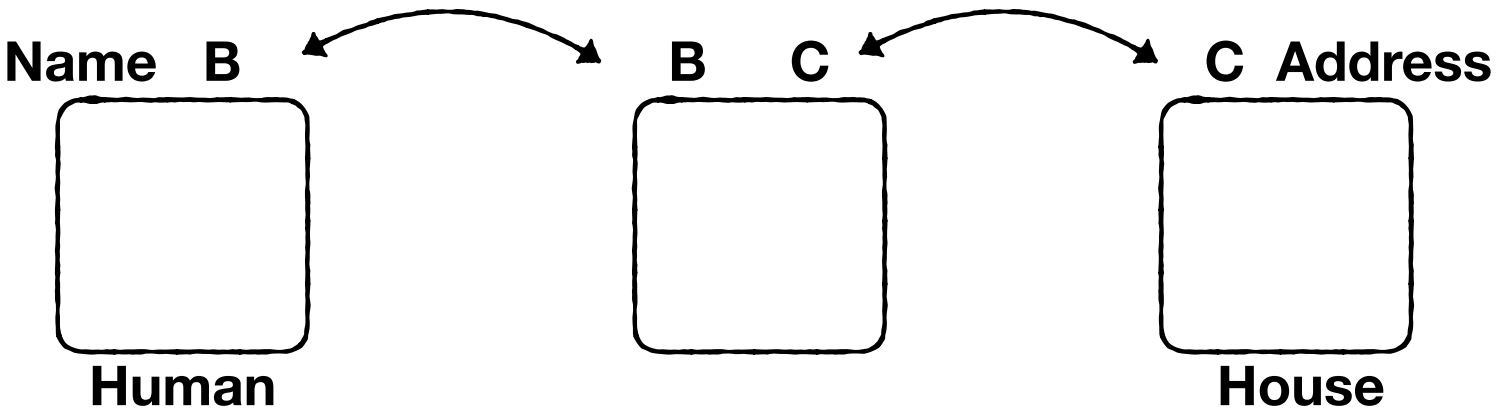
CSC443H1 Database System Technology

Niv Dayan - 14 March 2023

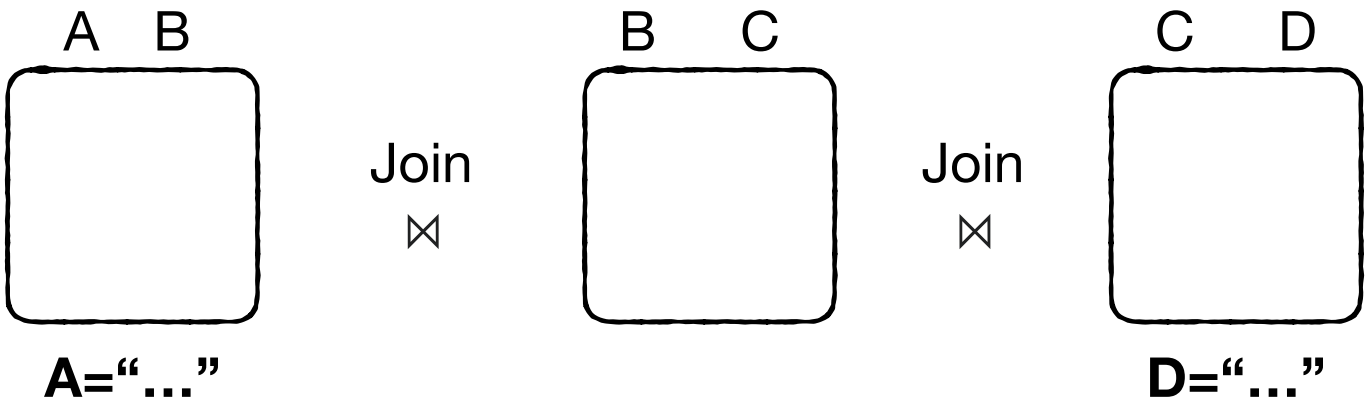
Consider three tables



Consider three tables

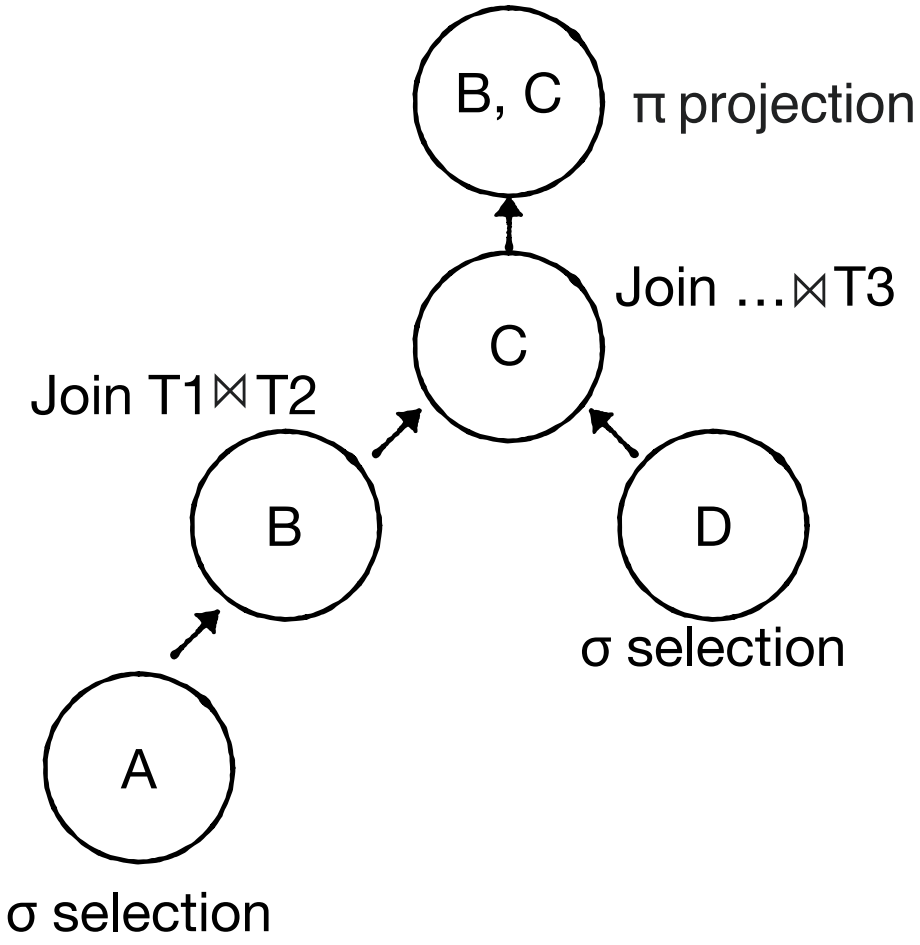


Consider three tables



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

All implement an iterator interface (the glue)



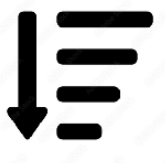
Each can be implemented using different algorithms

Selection

Projection

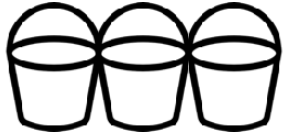
Join

Order by



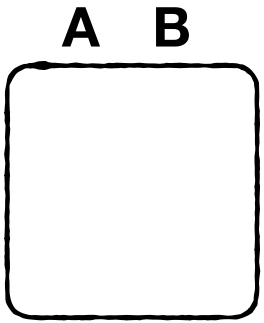
Distinct

Group by



different trade-offs for different contexts

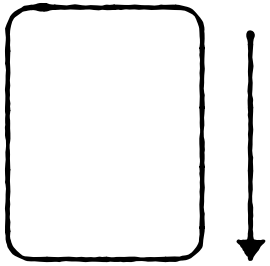
Selection



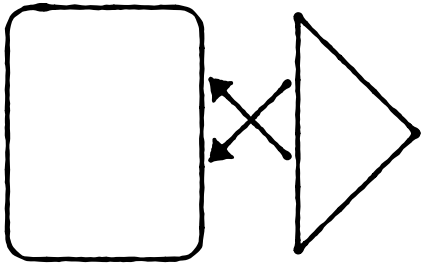
Select * from … where **A = “…”**

Select * from … where **A = “…”**

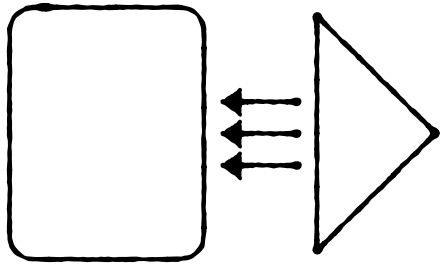
Scan



Unclustered Index



Clustered Index

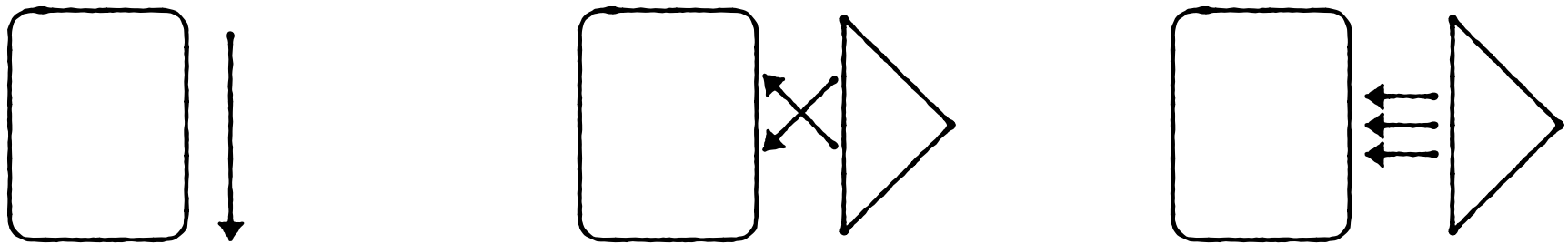


Select * from … where **A = "…"**

Scan

Unclustered Index

Clustered Index



O(N/B)

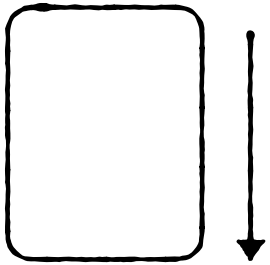
$O(\log_B(N)+\mathbf{S})$

Assuming B-tree

 # qualifying tuples

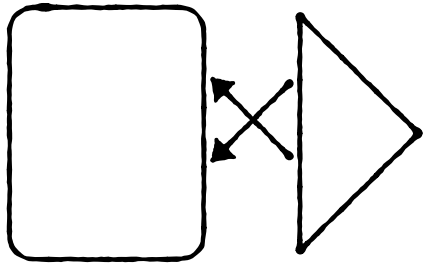
Select * from … where **A = “…”**

Scan



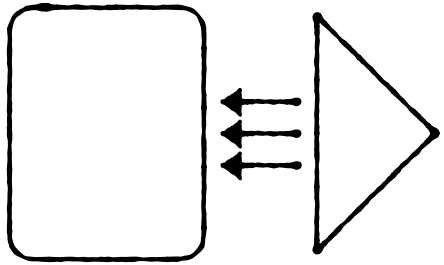
O(N/B)

Unclustered Index



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

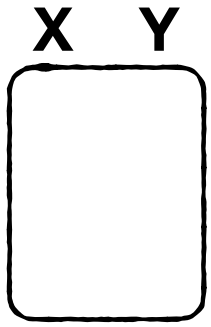
Clustered Index



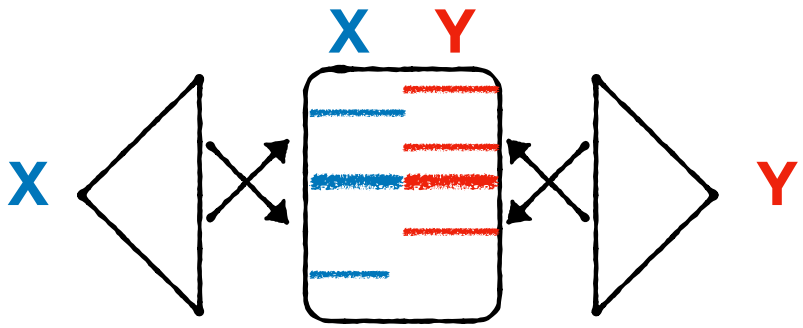
$O(\log_B(N)+S/B)$

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

How about when we have multiple selection conditions?



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

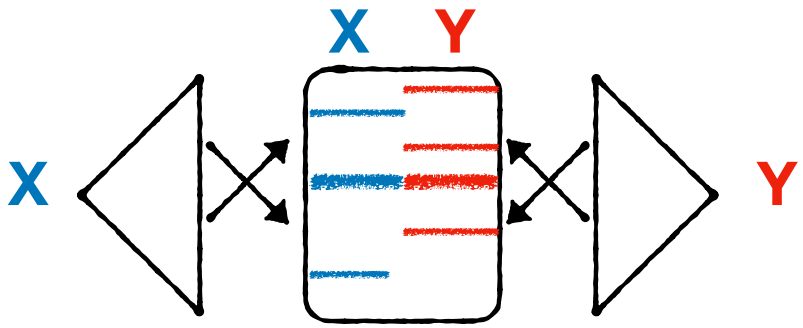


Let $|X_i|$ and $|Y_j|$ denote the number of matching rows to A and to B, resp.

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

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

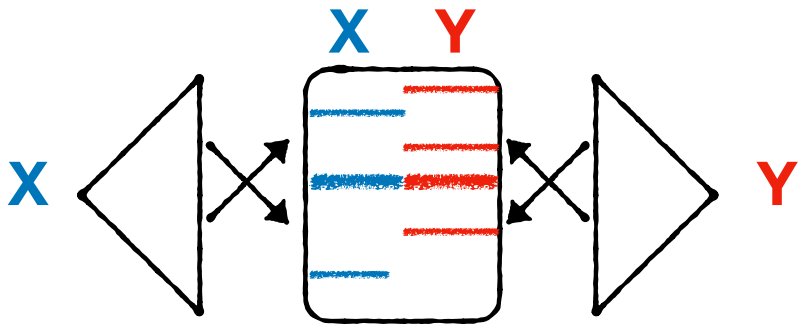
Select * from … where **X = “i” and Y=“j”**



Algorithm 2: **Search index X**

For each row in table, check if Y matches

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



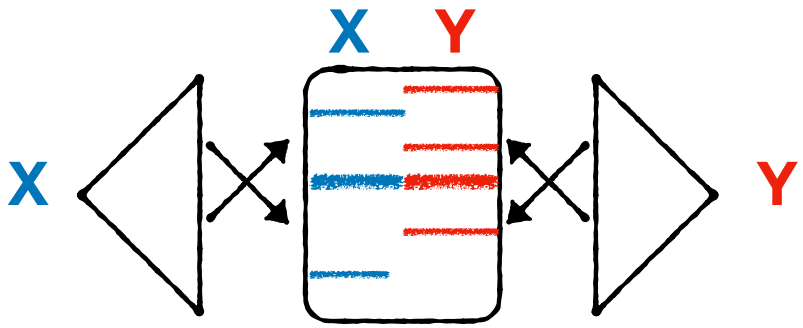
Algorithm 2: Search index X

For each row in table, check if Y matches

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

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

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

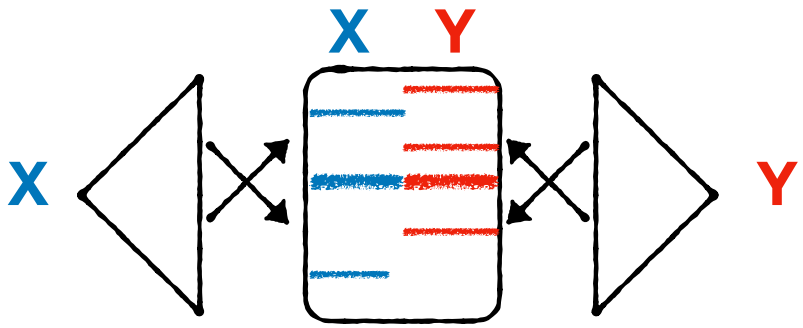


Principle: filter based on most selective predicate first

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

Is there yet another alternative?

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



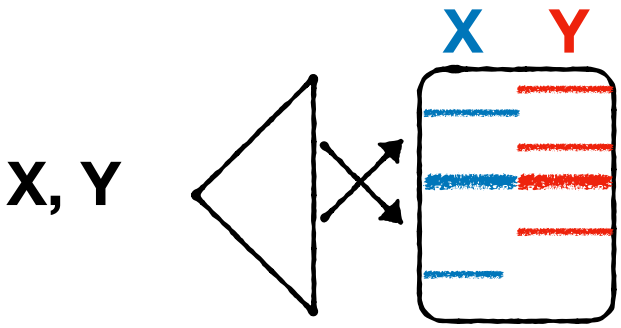
Algorithm 3: Search index A and B for matching row IDs

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 $|X_i|$, $|Y_j|$ and $\log_B(N)$

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



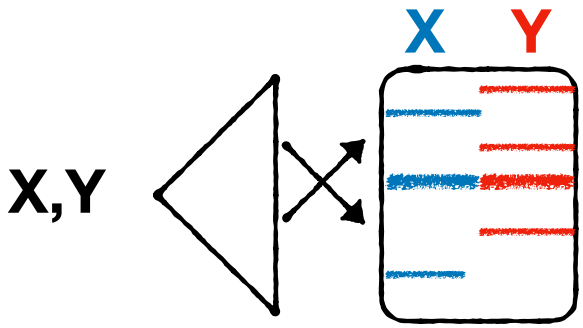
What if we combine these indexes into a composite index?

Select * from … where **surname = “Abbot” and firstname=“Bella”**

Surname, first name

Abbott, Adam
Abbott, Bella
Abby, Amelia

Select * from … where **Y=“…”**



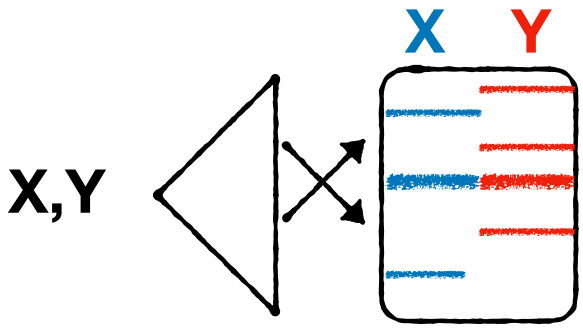
Algorithm 4: Search composite index A and B for matching row IDs

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

Cheapest option so far!

Downsides? Cannot handle queries just based on Y

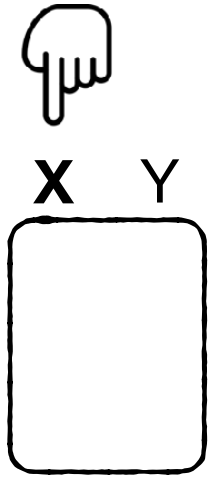
Select * from … where **Y=“…”**



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

Projection

Select X from table

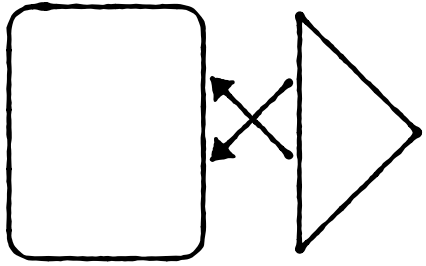


Trivial in row-stores. More interesting in column-stores (next week).

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

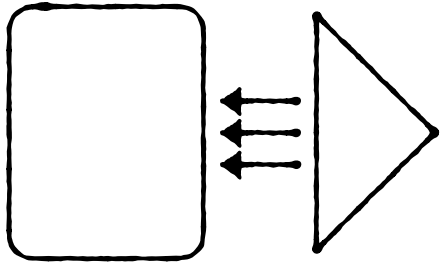
If we have an applicable index

Unclustered
index scan



(Good if input is small)

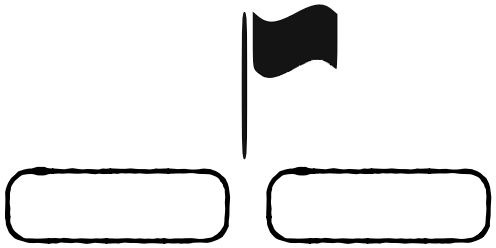
Clustered
index scan



(Good in all cases)

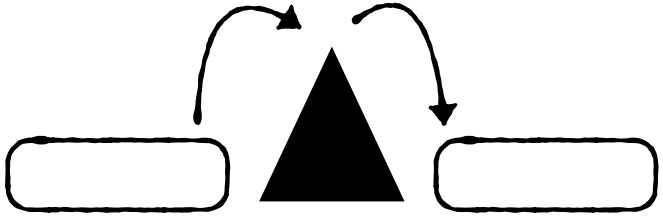
Select … From …**Order By** col1, col2, ... ASC | DESC;

Quick-Sort



If input fits in memory

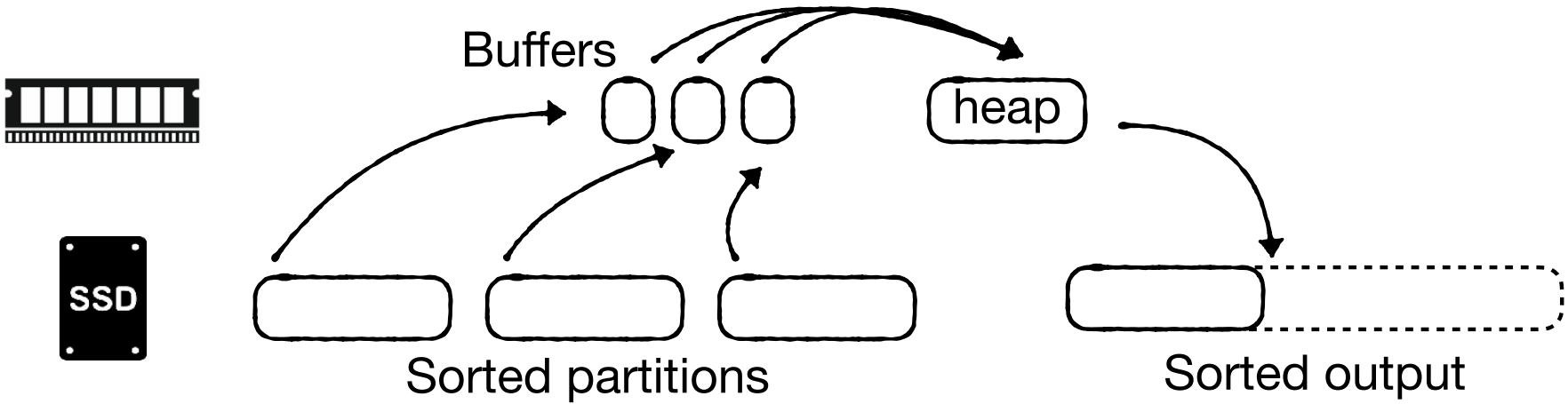
Heap-Sort



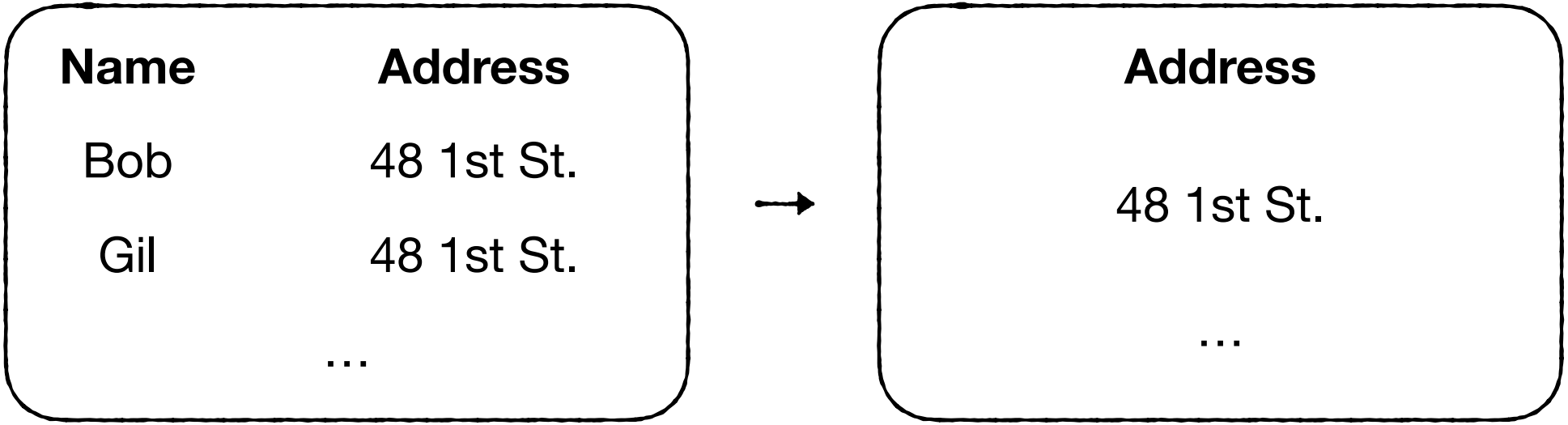
If quick-sort takes longer than expected

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

Or external sort of data does not fit in memory



Select **Distinct** address FROM people;



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

Sort & eliminate adjacent identical items

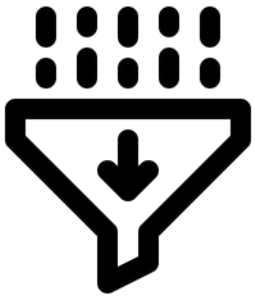
Quick-Sort
Heap-Sort
External Sort
Index scan
Etc.

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

Sort & eliminate adjacent identical items

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

Hash to identify identical items



Linear probing
Chaining
Extendible hashing
Cuckoo hashing
Etc.

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

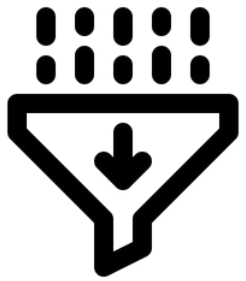
Sort & eliminate adjacent identical items

$O(N \log_2 N)$

Better if we later need to sort anyways

Robust to skew

Hash to identify identical items



$O(N)$

Good if user is fine with unordered output

Less robust to skew

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

(Income per household)

Name	Address	Income
Bob	48 1st St.	**100K**
Gil	48 1st St.	**100K**

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

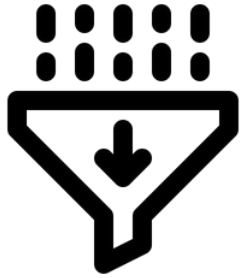
Select c1, c2, ... FROM … **Group By**

Index/Sort by Category

CPU: $O(N \log_2 N)$

Better if we later need to sort anyways

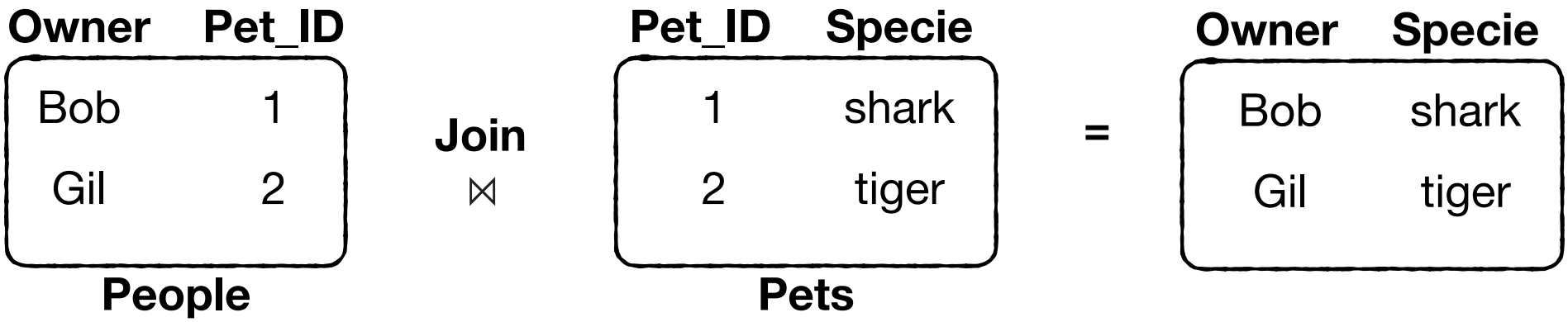
Hash by Category



O(N)

Good if user is fine with unordered output

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



Join Algorithms

Nested Loop

Block Nested Loop

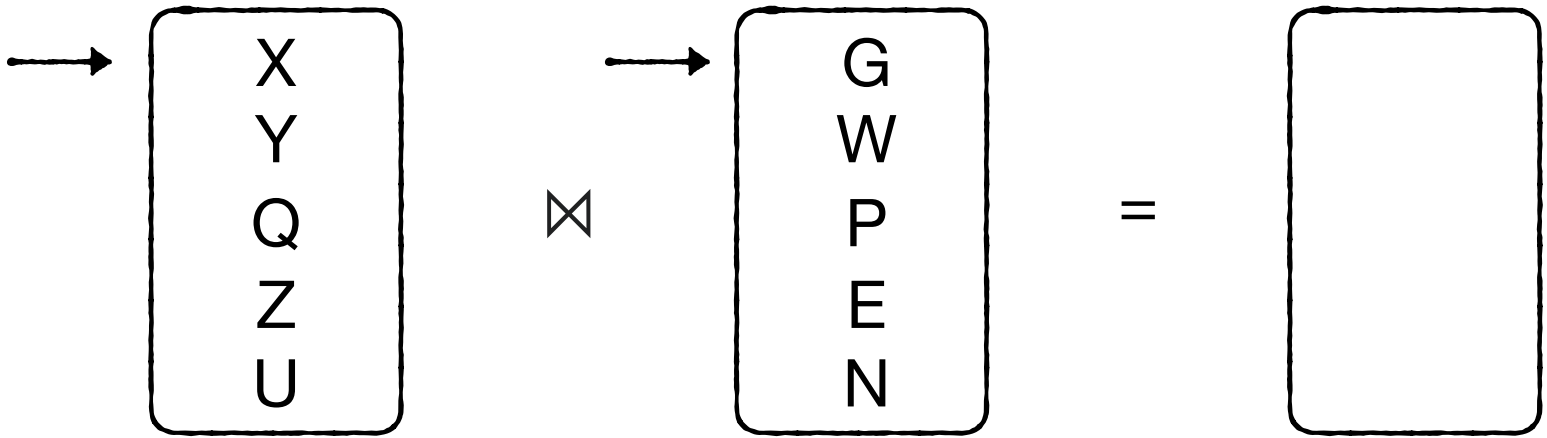
Index-Join

Sort-merge Join

Grace hash Join

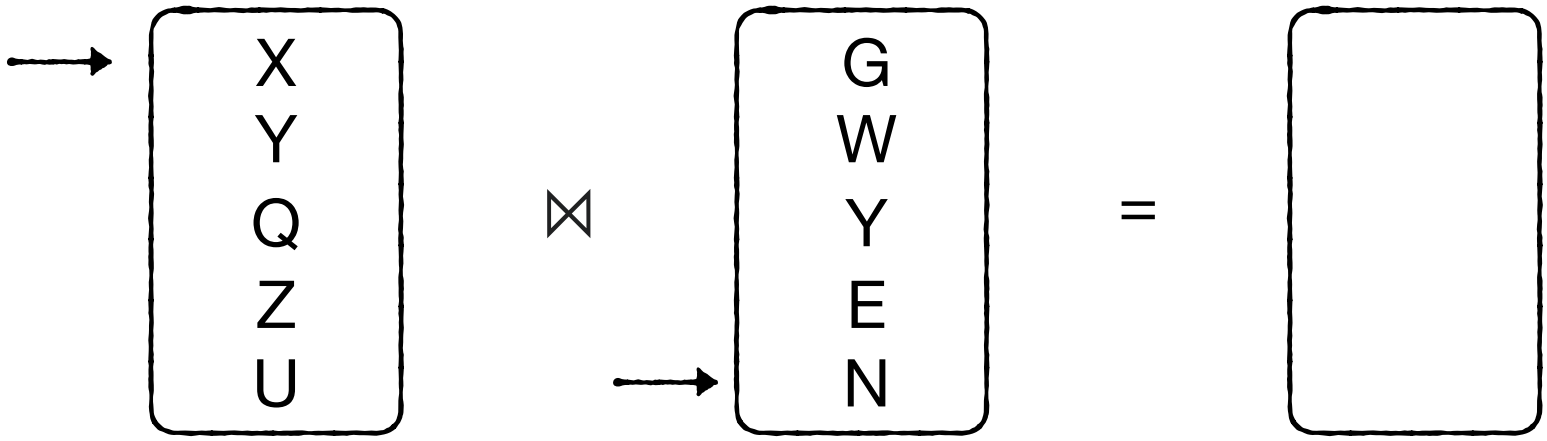
Nested Loop

For each entry in one relation, scan whole other relation



Nested Loop

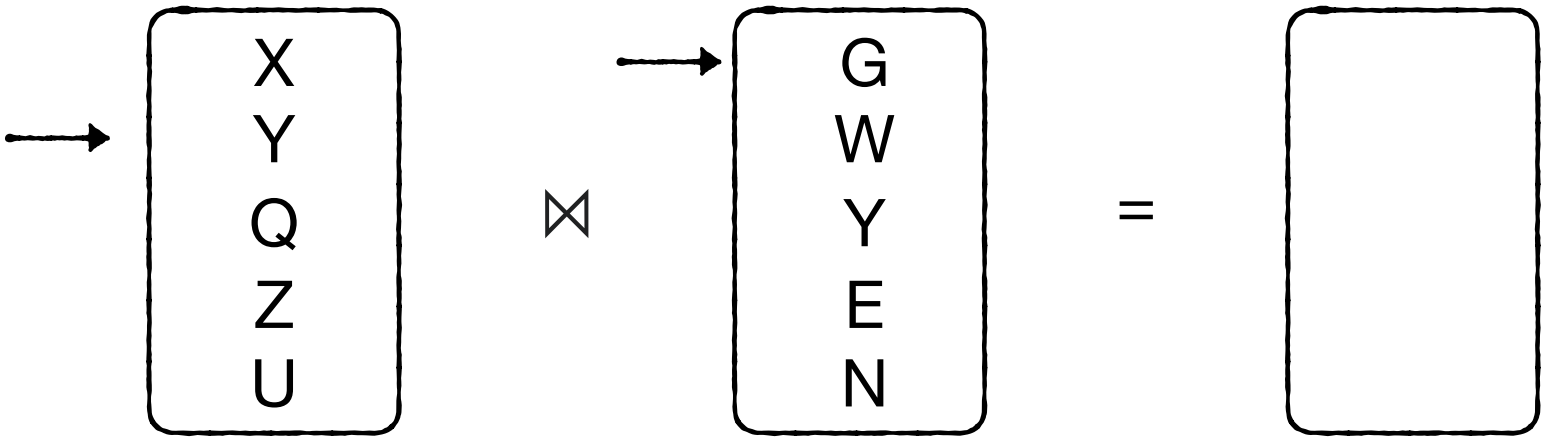
For each entry in one relation, scan whole other relation



No match for X

Nested Loop

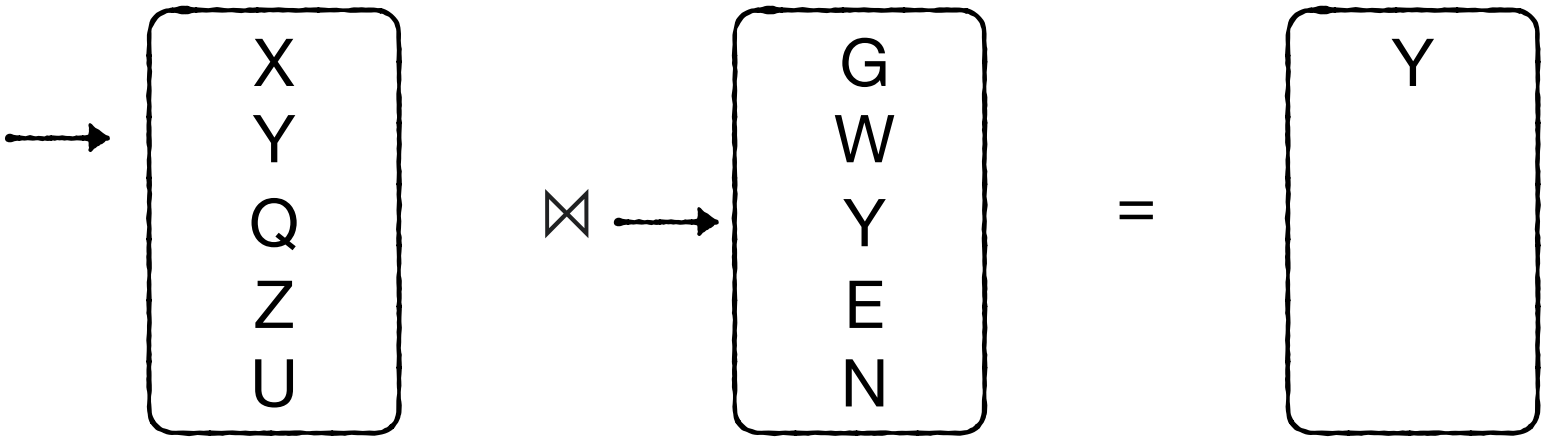
For each entry in one relation, scan whole other relation



No match for X

Nested Loop

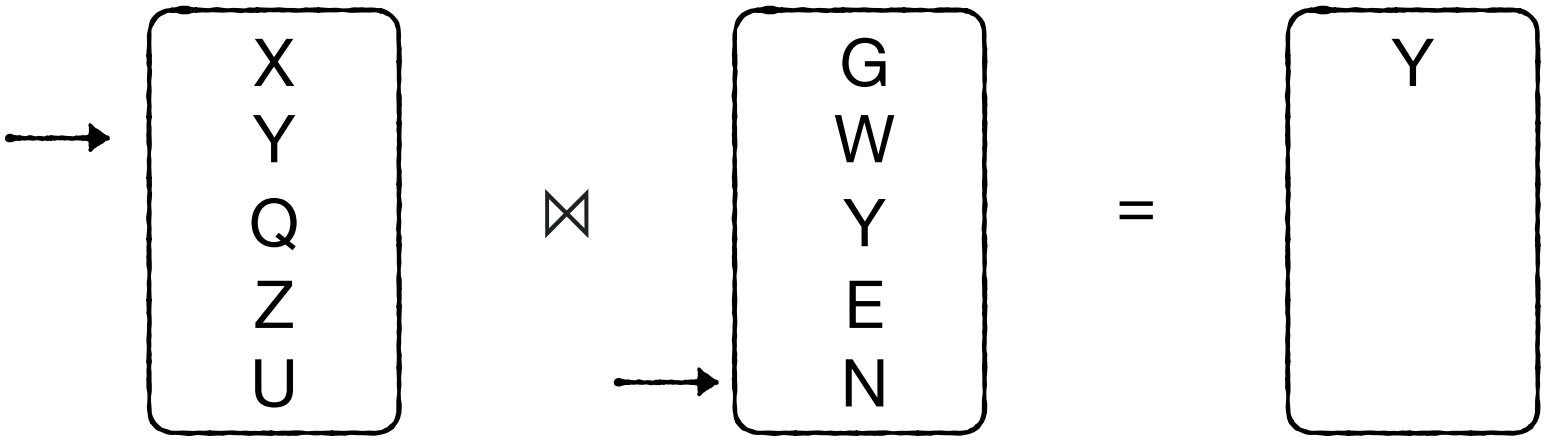
For each entry in one relation, scan whole other relation



Match for Y

Nested Loop

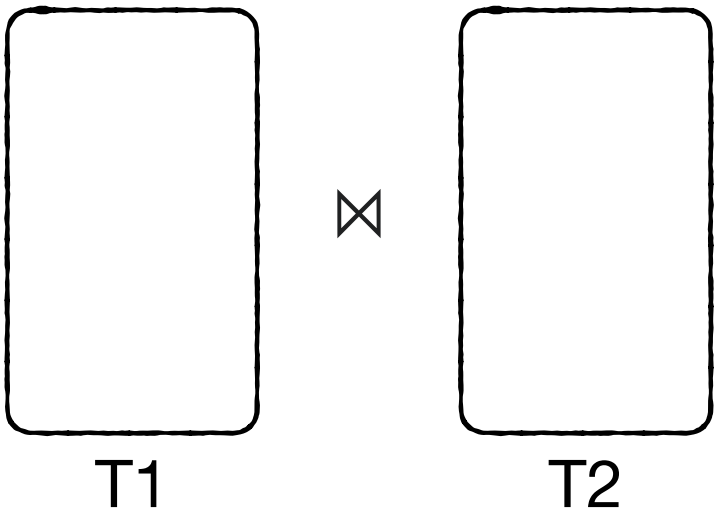
For each entry in one relation, scan whole other relation



No more matches for Y

Nested Loop

For each entry in one relation, scan whole other relation

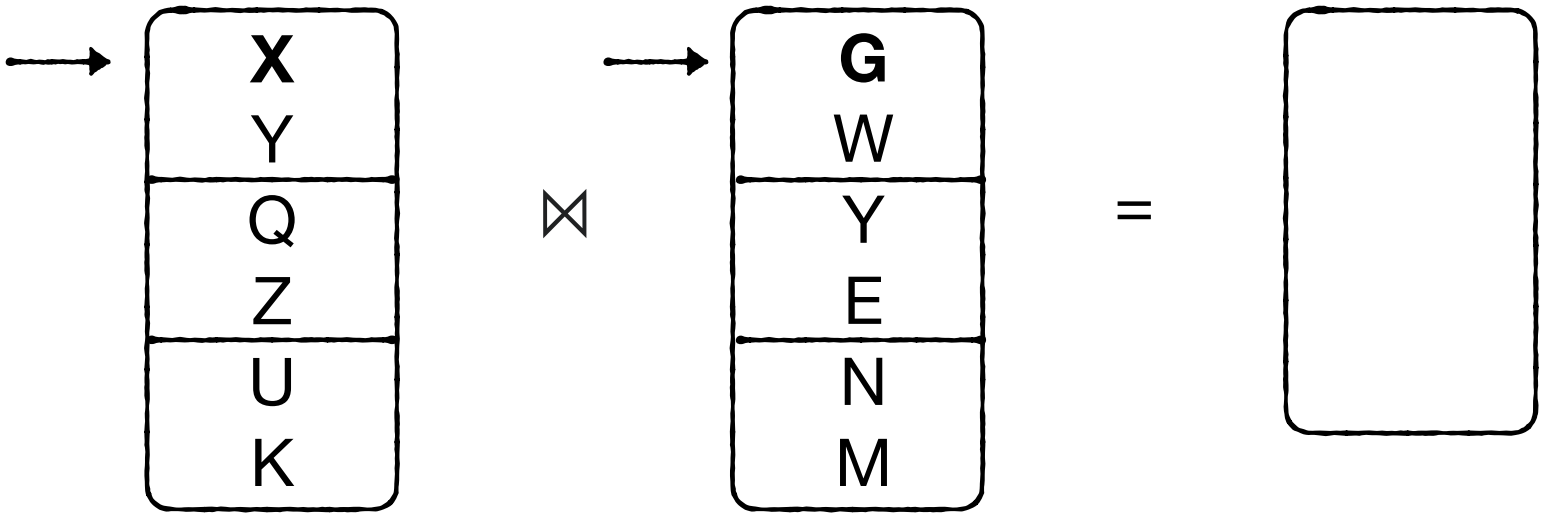


Cost: $O(|T1| \cdot |T2|/B)$ I/O

What's a simple improvement?

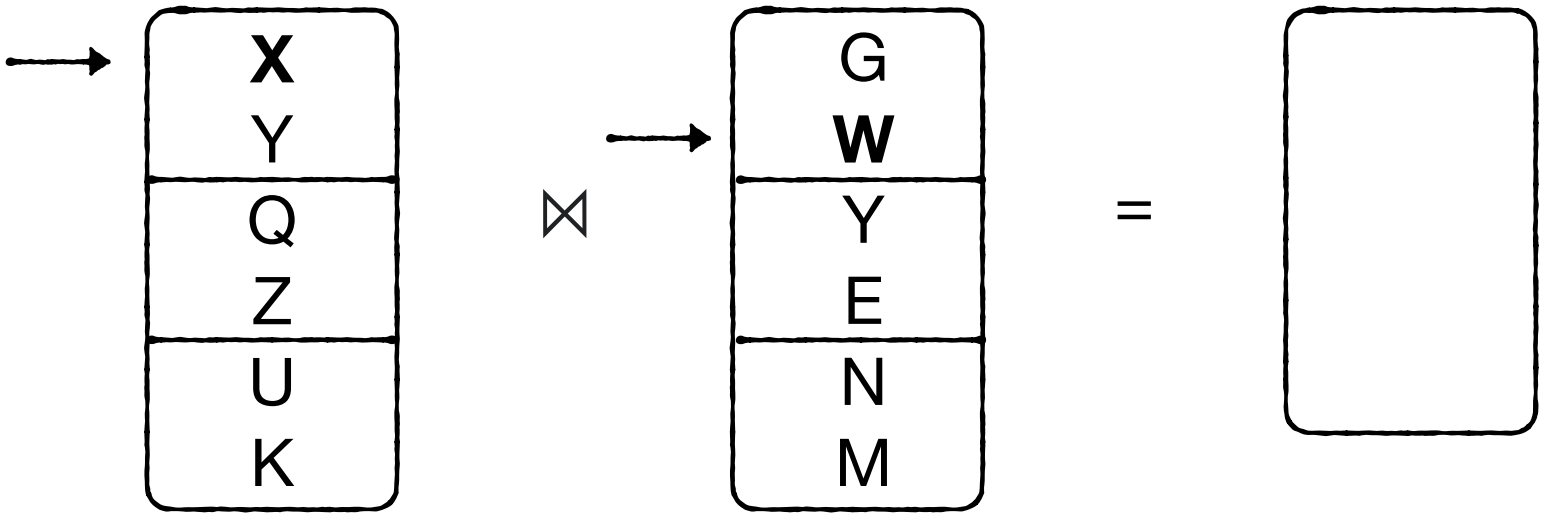
Block Nested Loop

For each block in one relation, scan whole other relation



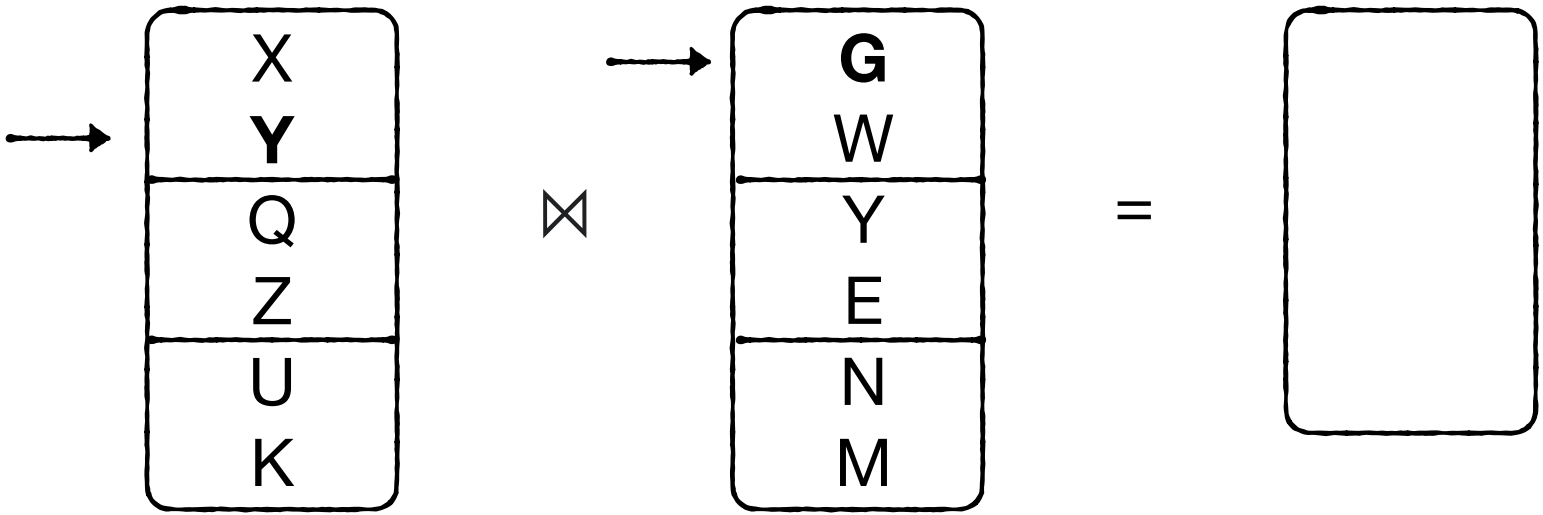
Block Nested Loop

For each block in one relation, scan whole other relation



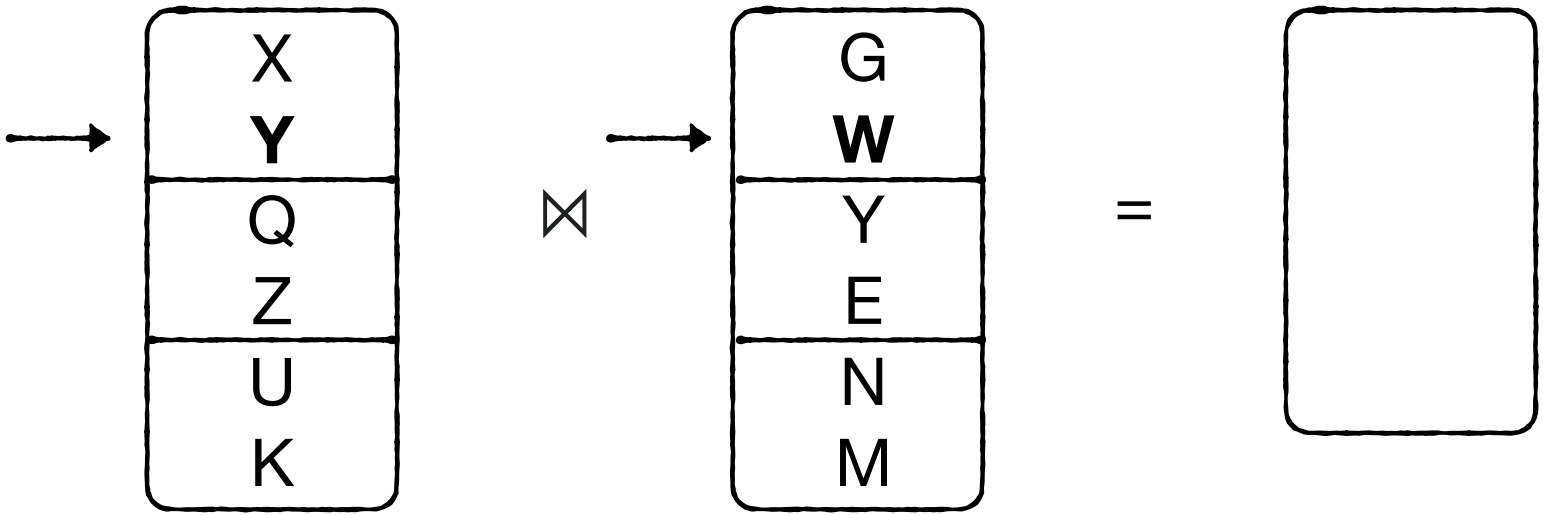
Block Nested Loop

For each block in one relation, scan whole other relation



Block Nested Loop

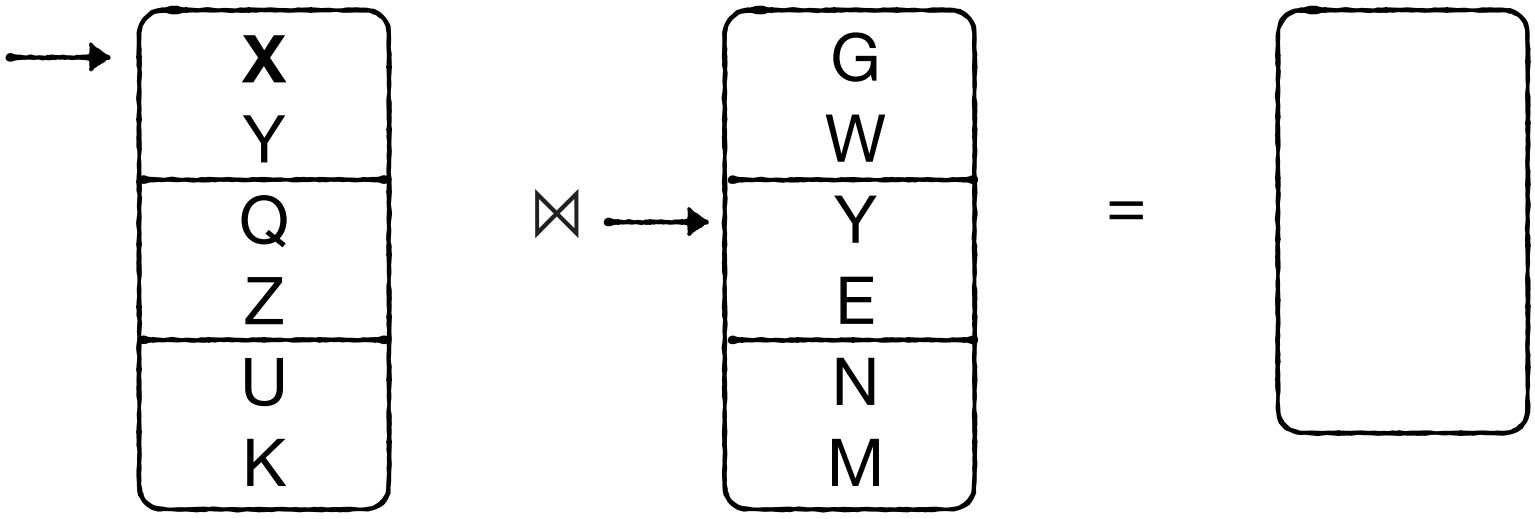
For each block in one relation, scan whole other relation



No match in first block

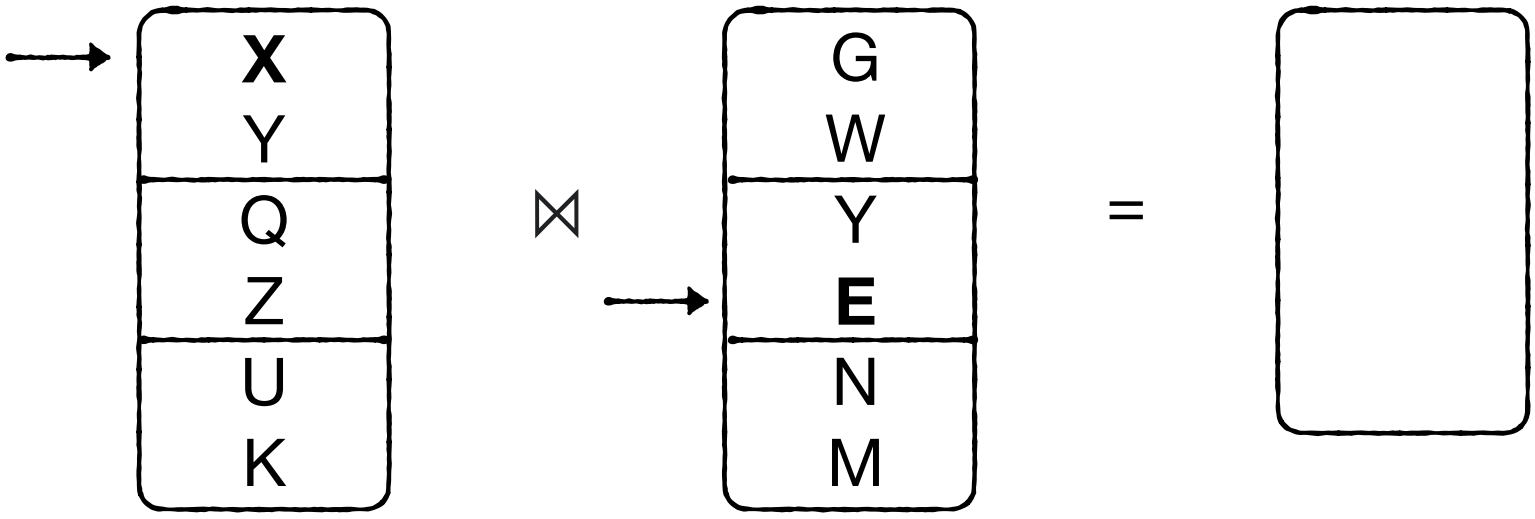
Block Nested Loop

For each block in one relation, scan whole other relation



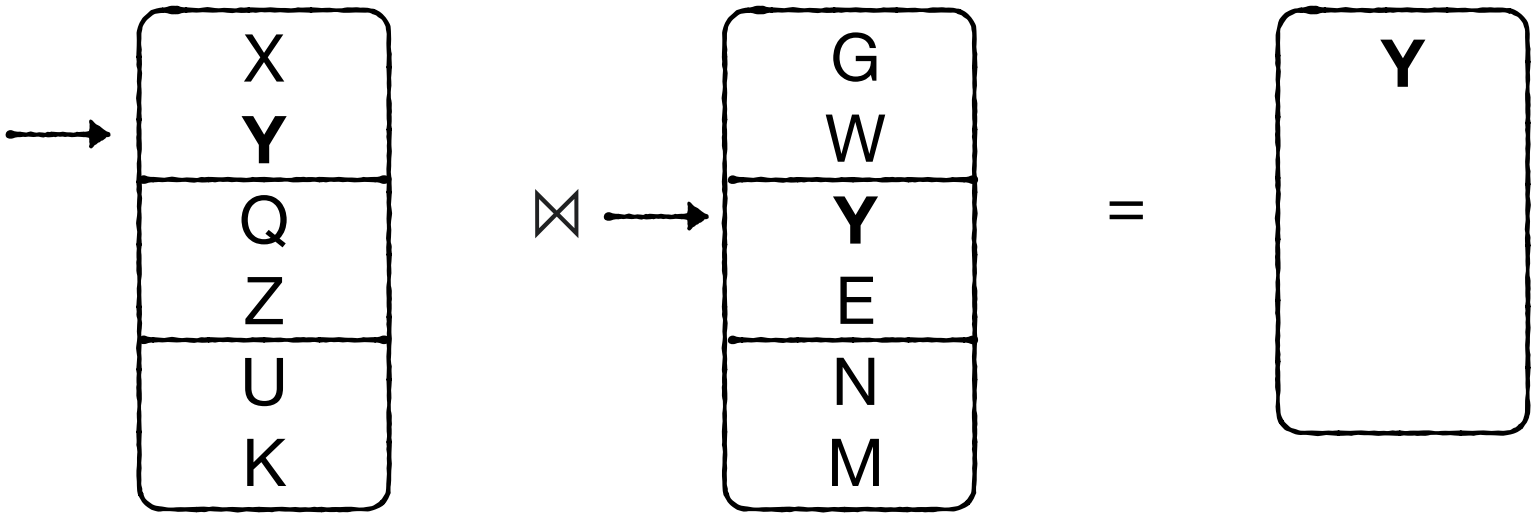
Block Nested Loop

For each block in one relation, scan whole other relation



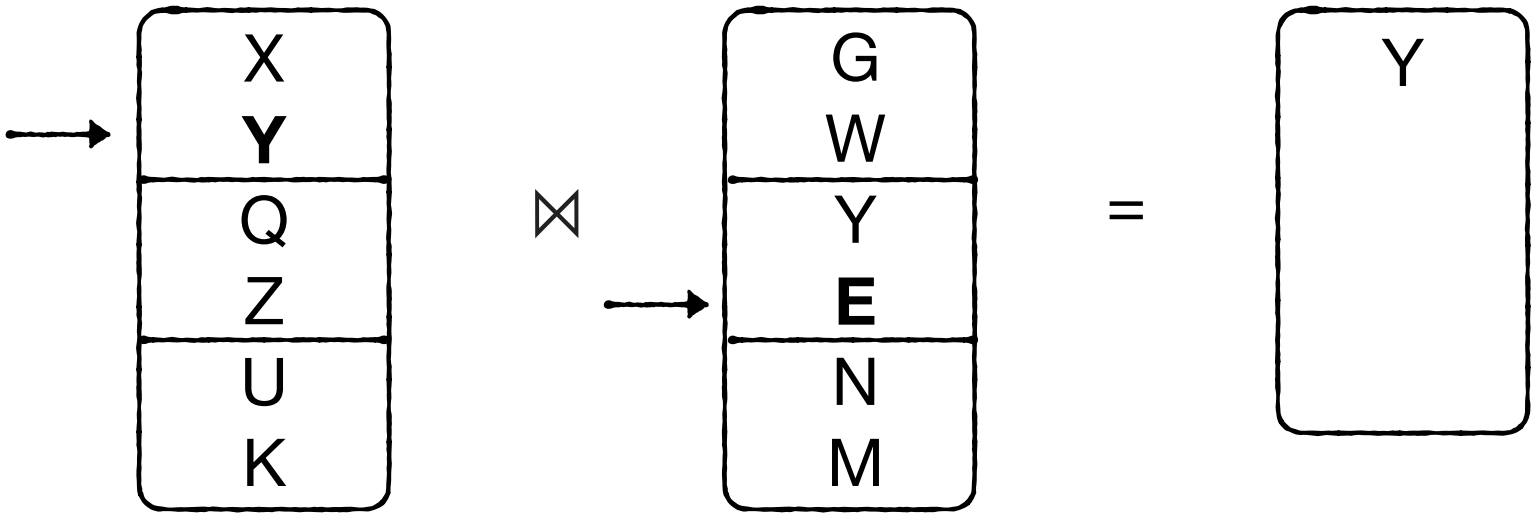
Block Nested Loop

For each block in one relation, scan whole other relation



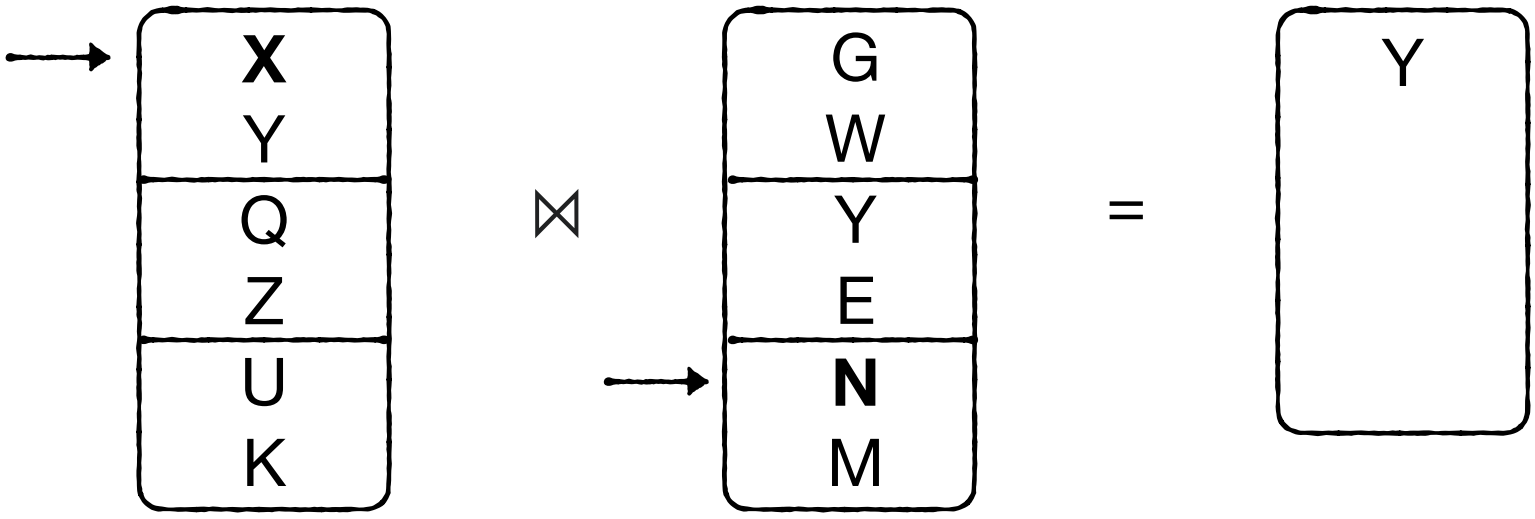
Block Nested Loop

For each block in one relation, scan whole other relation



Block Nested Loop

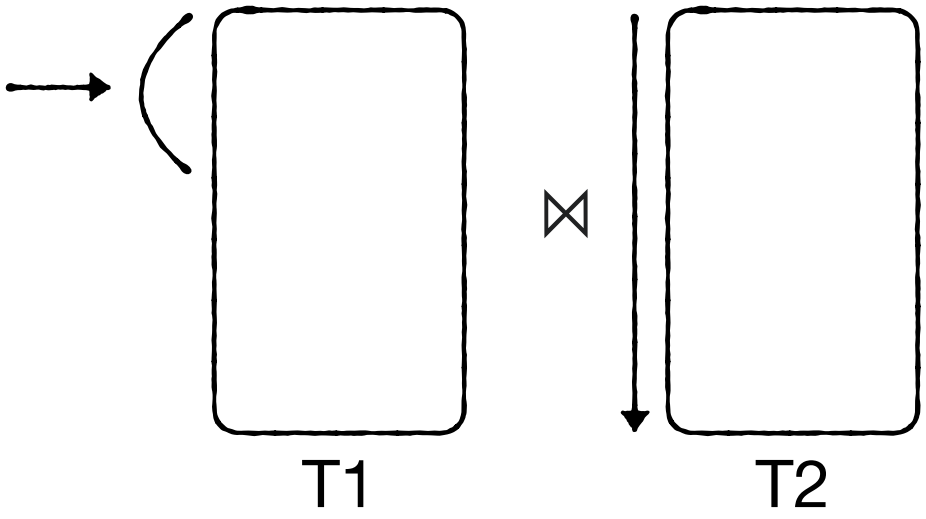
For each block in one relation, scan whole other relation



And so on…

Block Nested Loop

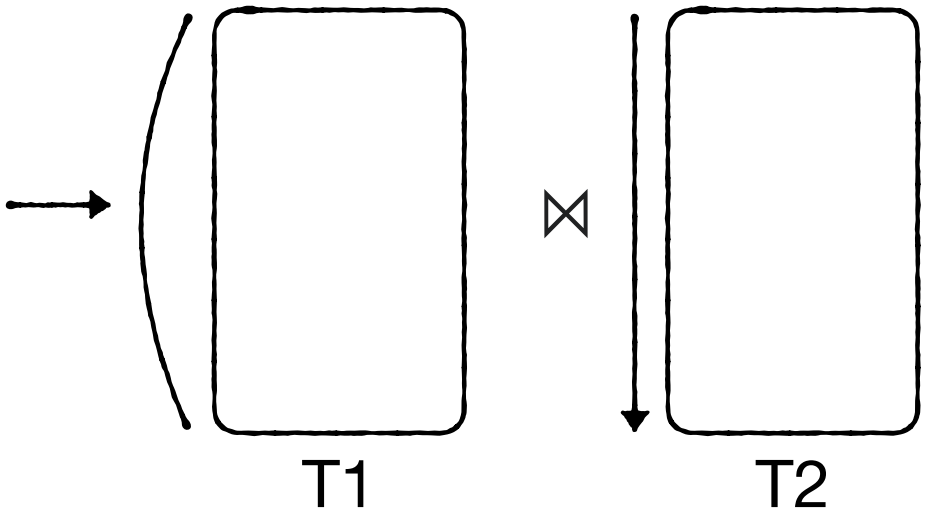
For each block in one relation, scan whole other relation



Cost: O(|T1|/B · |T2|/B) I/O

Block Nested Loop

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



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

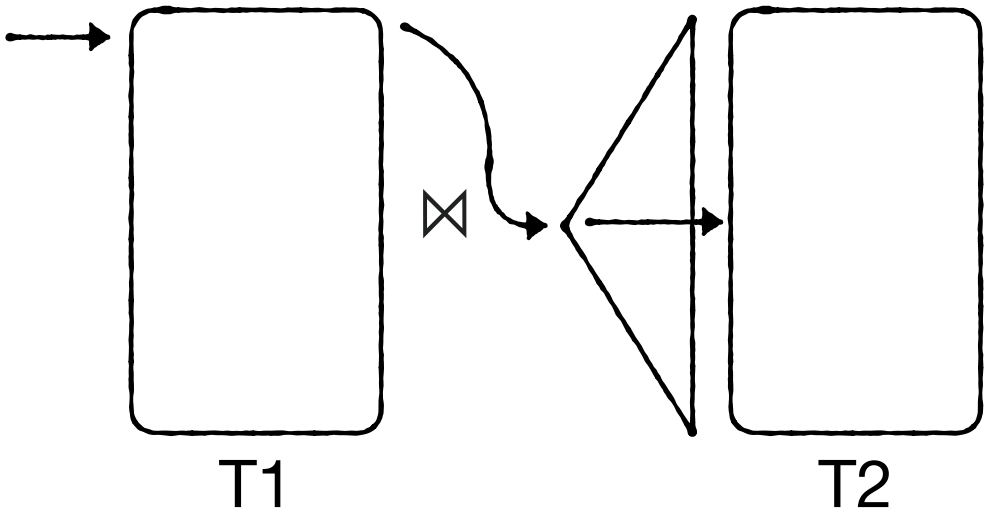
What if Q = min(|T1|/B, |T2|/B)?

(One table fits in memory)

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

Index-Join

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

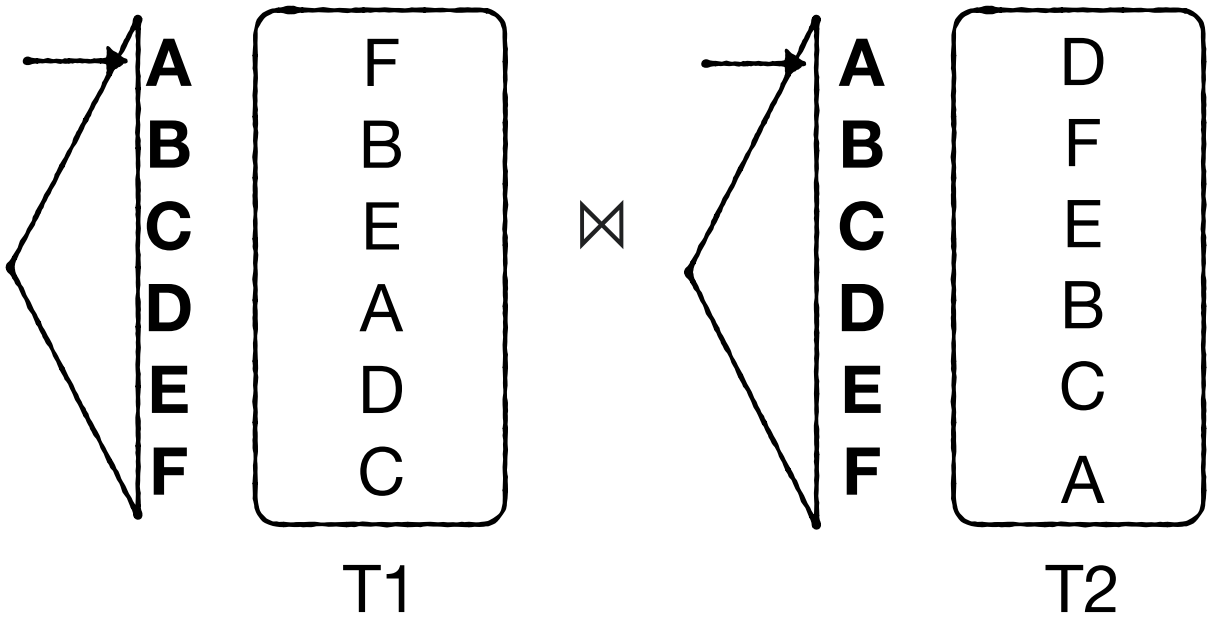


Cost: $O(|T1| \cdot \log_B |T2|)$ I/O

(Assuming B-tree)

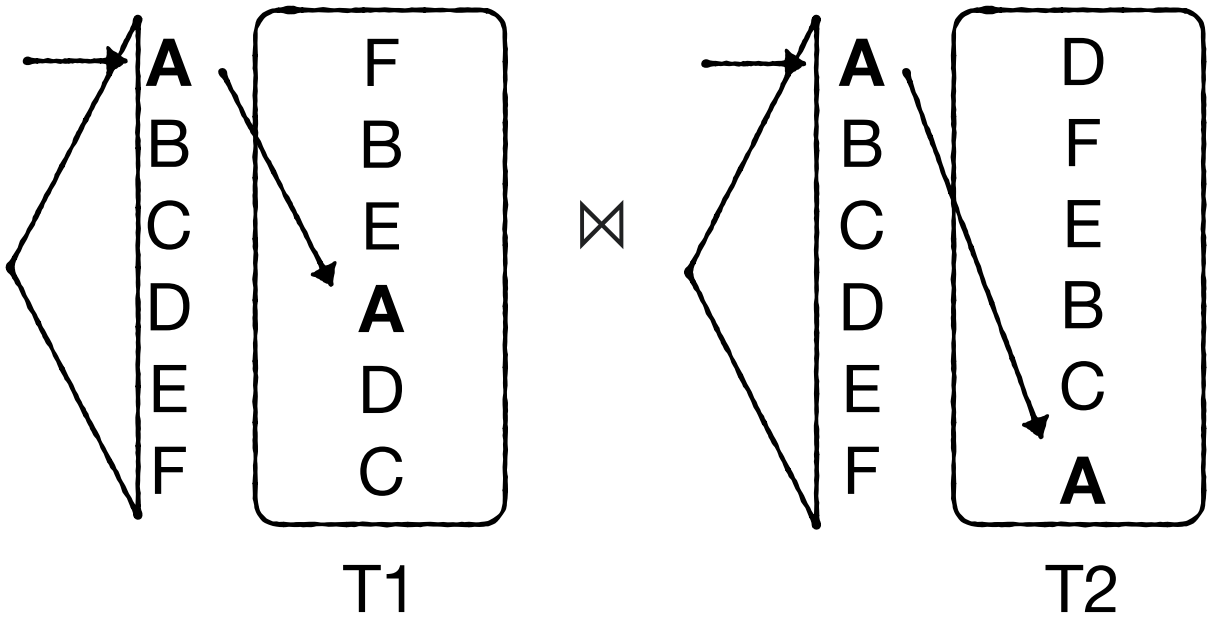
Index-Join

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



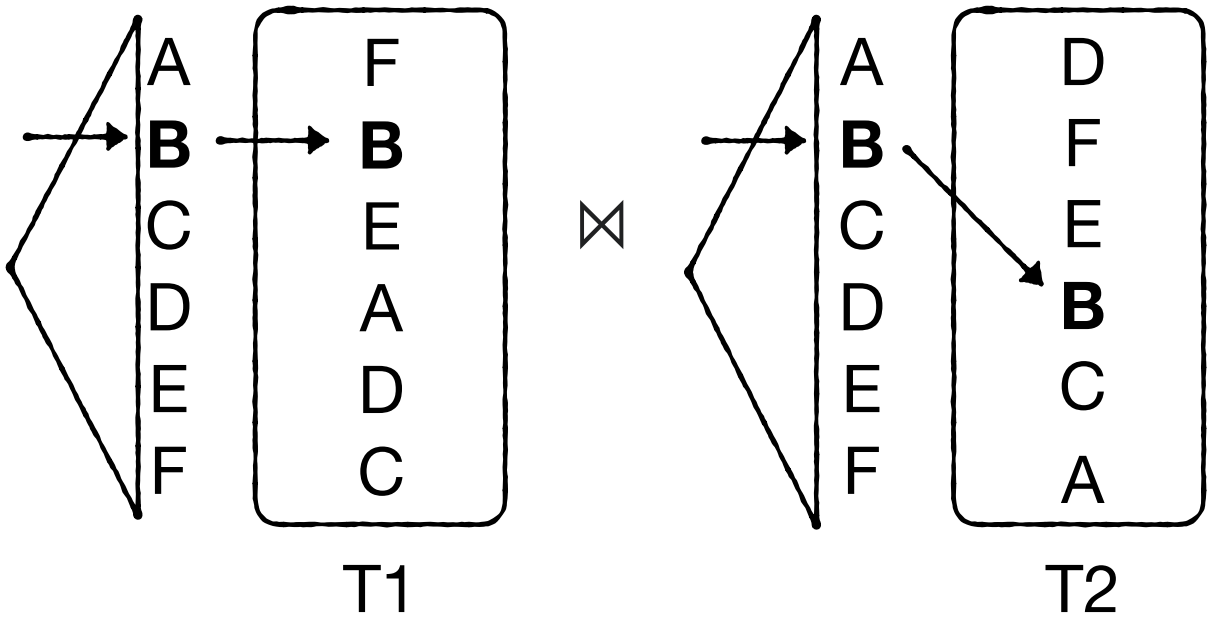
Index-Join

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



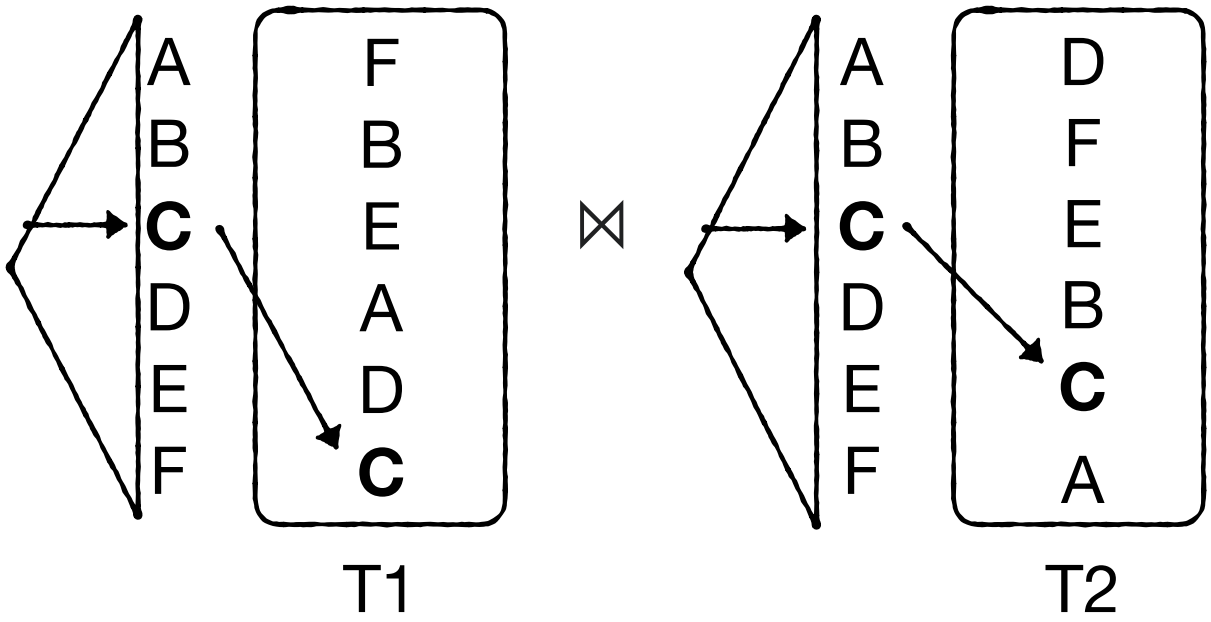
Index-Join

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



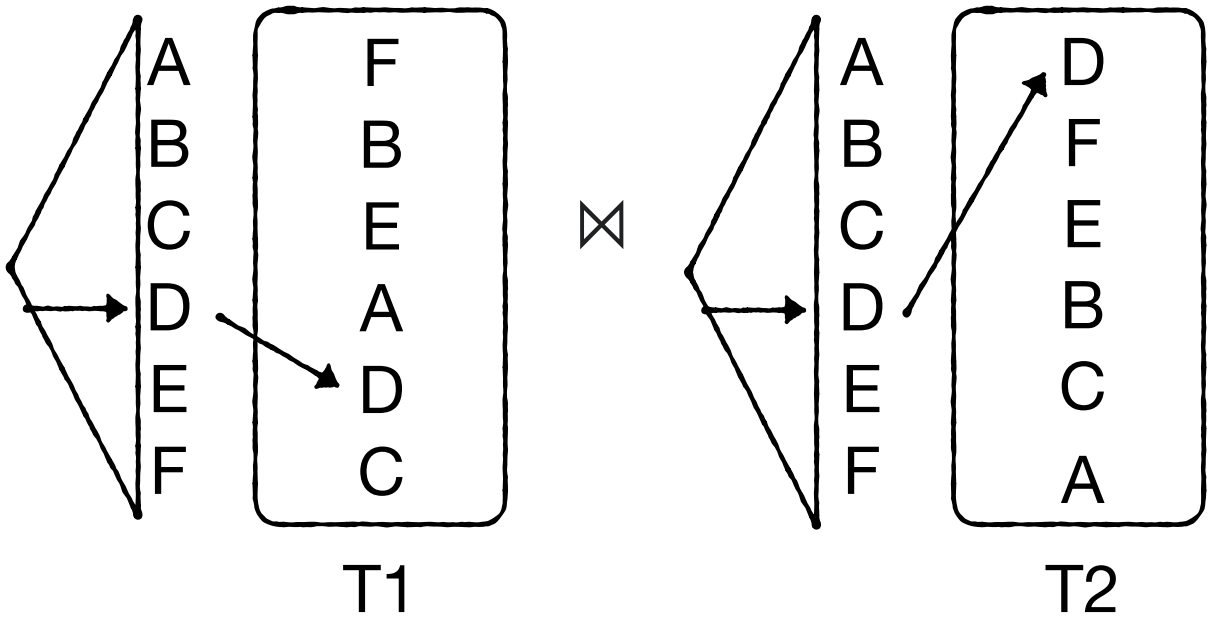
Index-Join

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



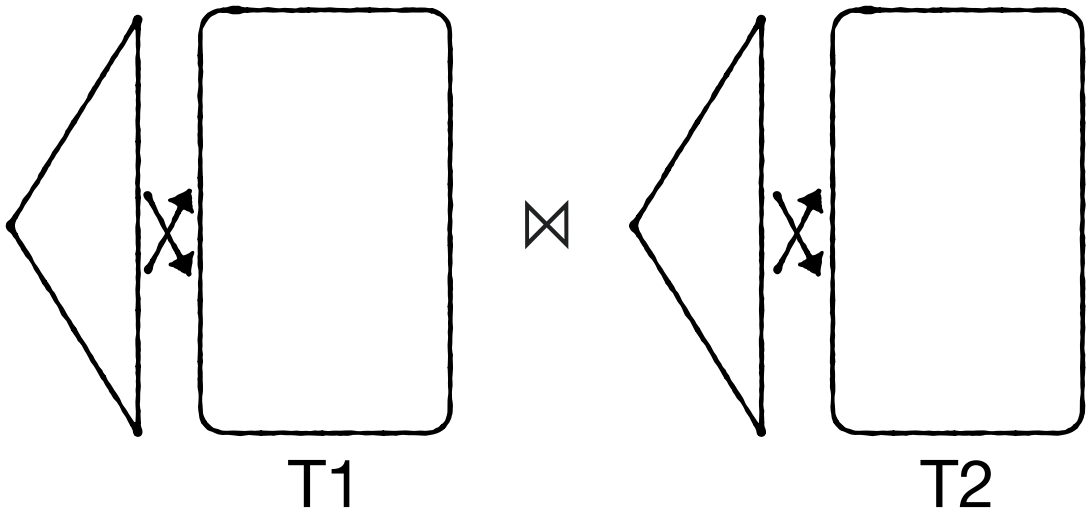
Index-Join

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



Index-Join

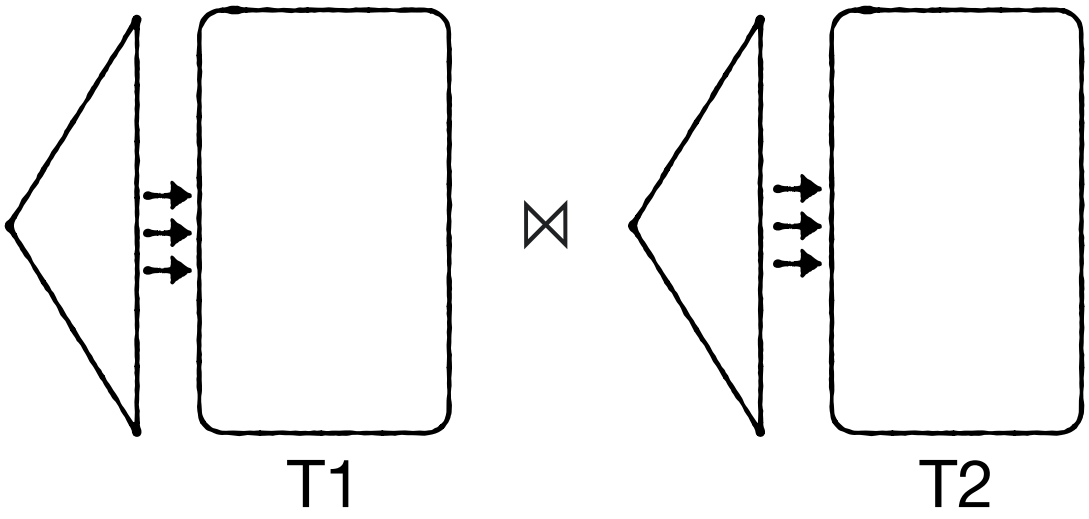
What if both relations have an unclustered index on the join key?



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

Index-Join

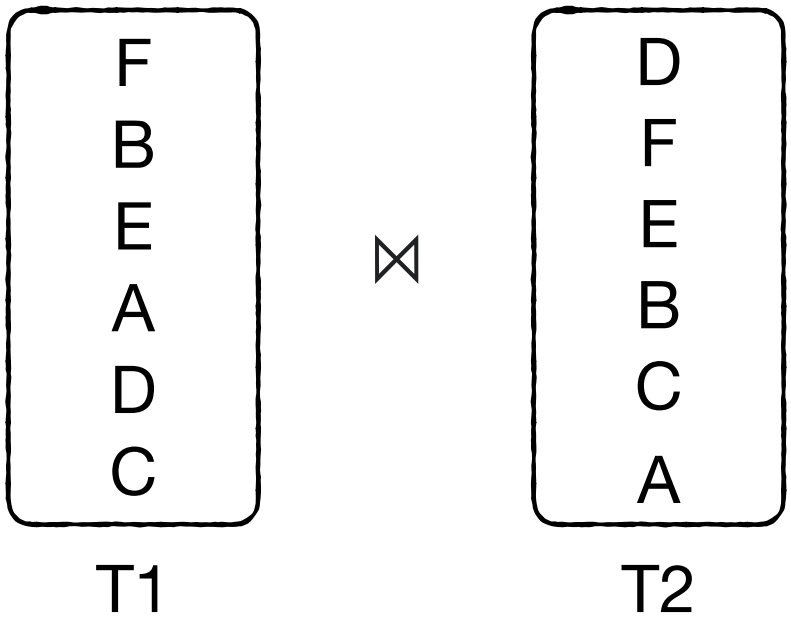
What if both indexes are clustered?



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

Sort-Merge Join

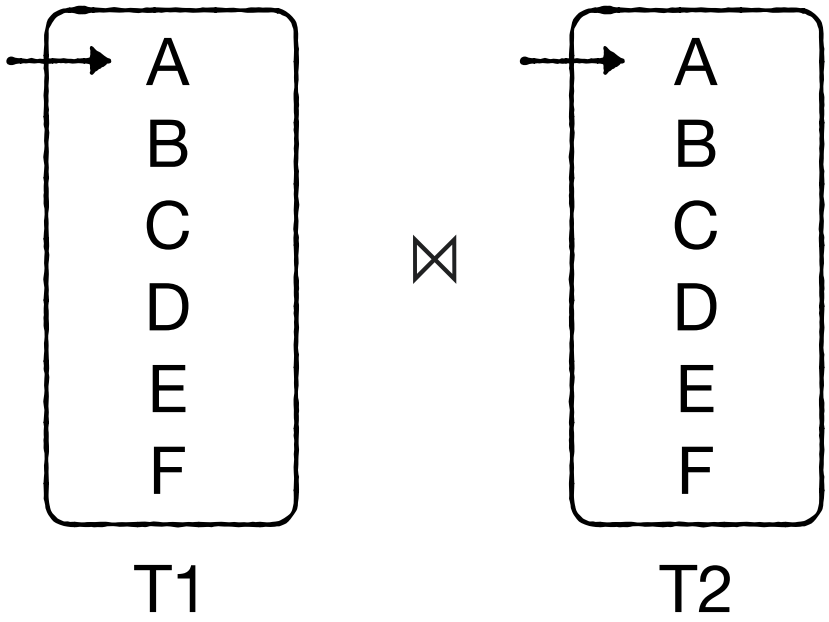
(A) Sort both relations based on join key



Sort-Merge Join

(A) Sort both relations based on join key

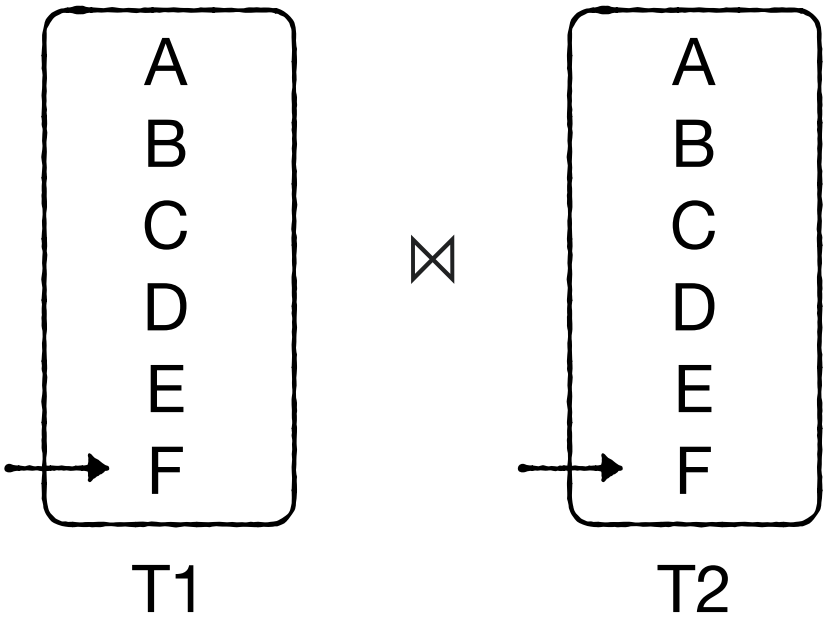
(B) Scan both relations linearly



Sort-Merge Join

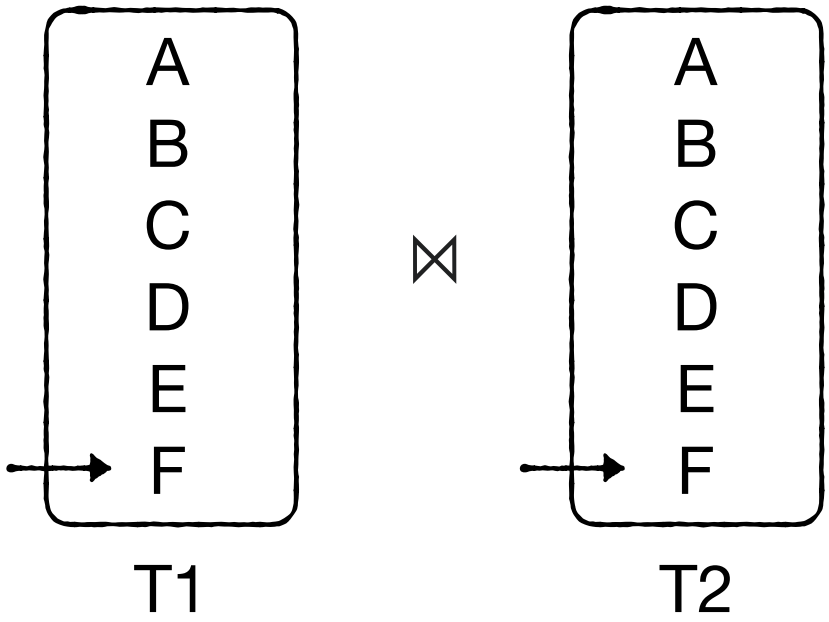
(A) Sort both relations based on join key

(B) Scan both relations linearly



Sort-Merge Join

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



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

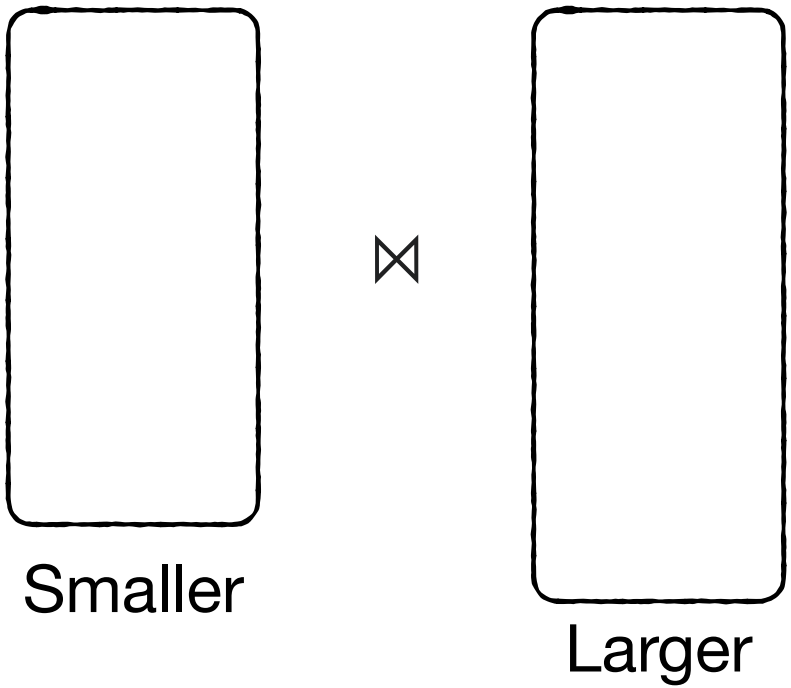
CPU cost? **O(|T1|·log$_2$|T1|+|T2|·log$_2$|T2|)**

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

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

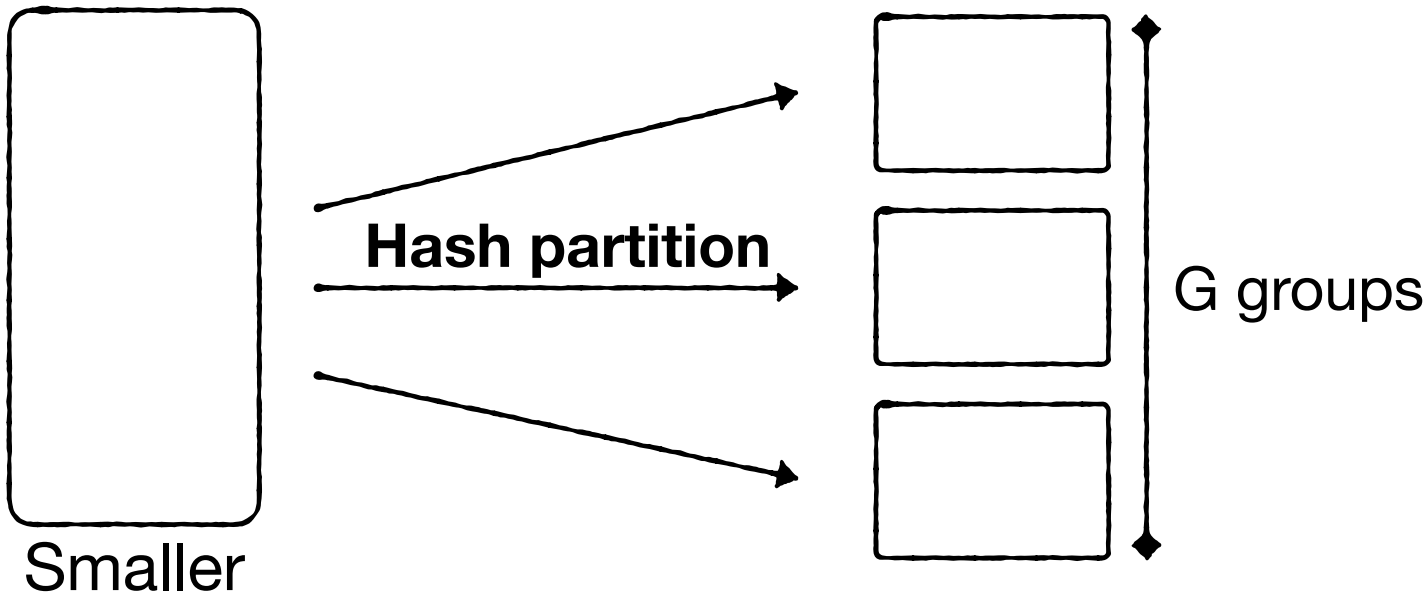
Grace Hash Join

Best for very large relations that do not fit in memory



Grace Hash Join

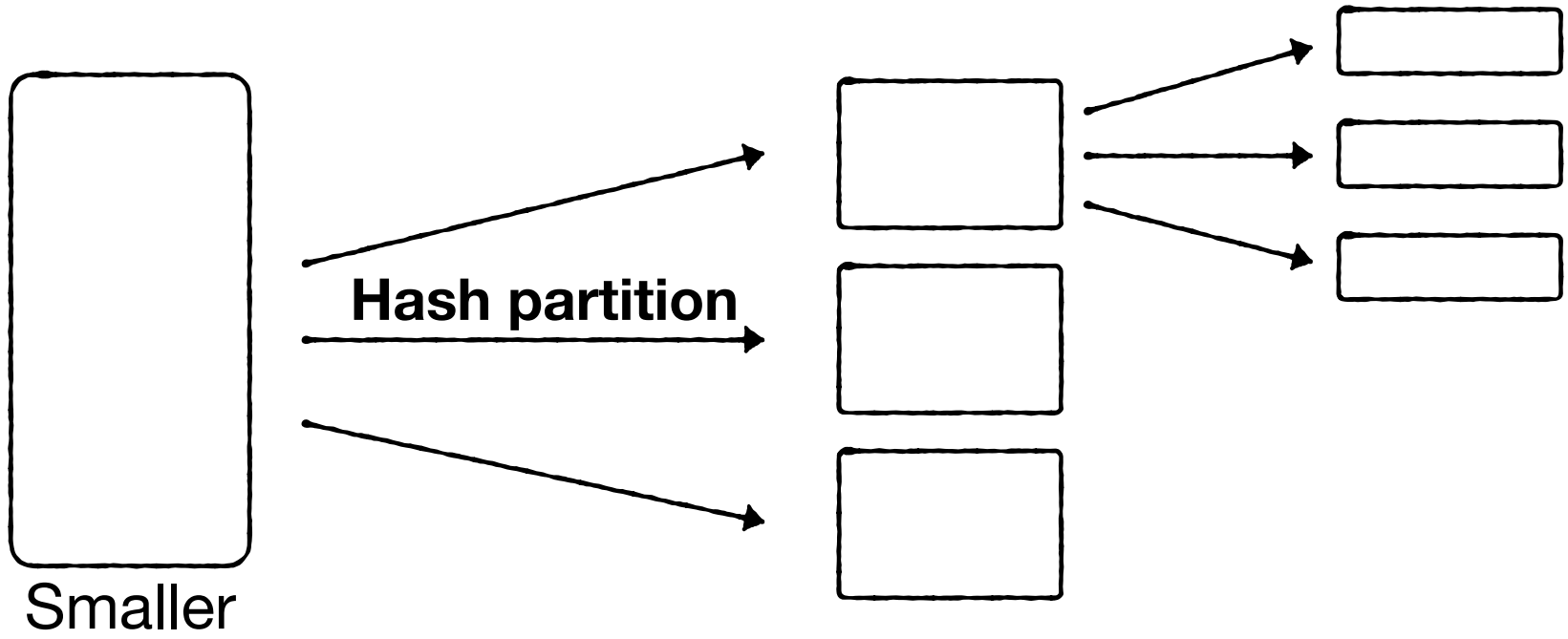
(A) Hash partition smaller table



Grace Hash Join

(A) Hash partition smaller table

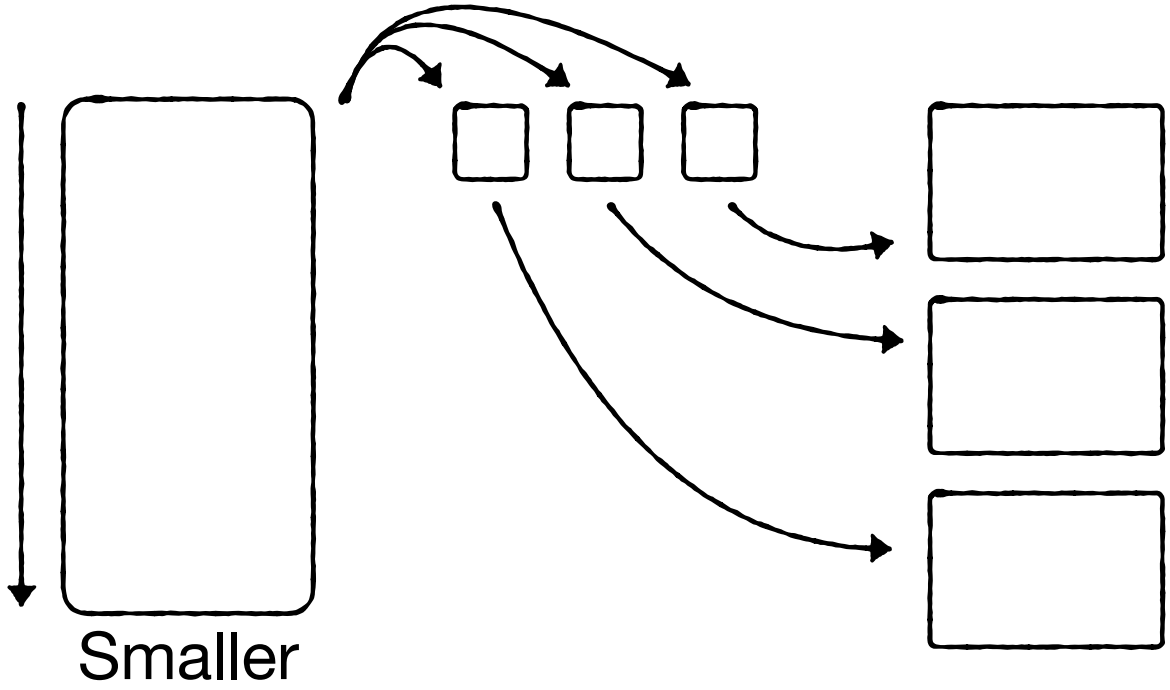
Repartition until each partition fits in memory



Grace Hash Join

(A) Hash partition smaller table

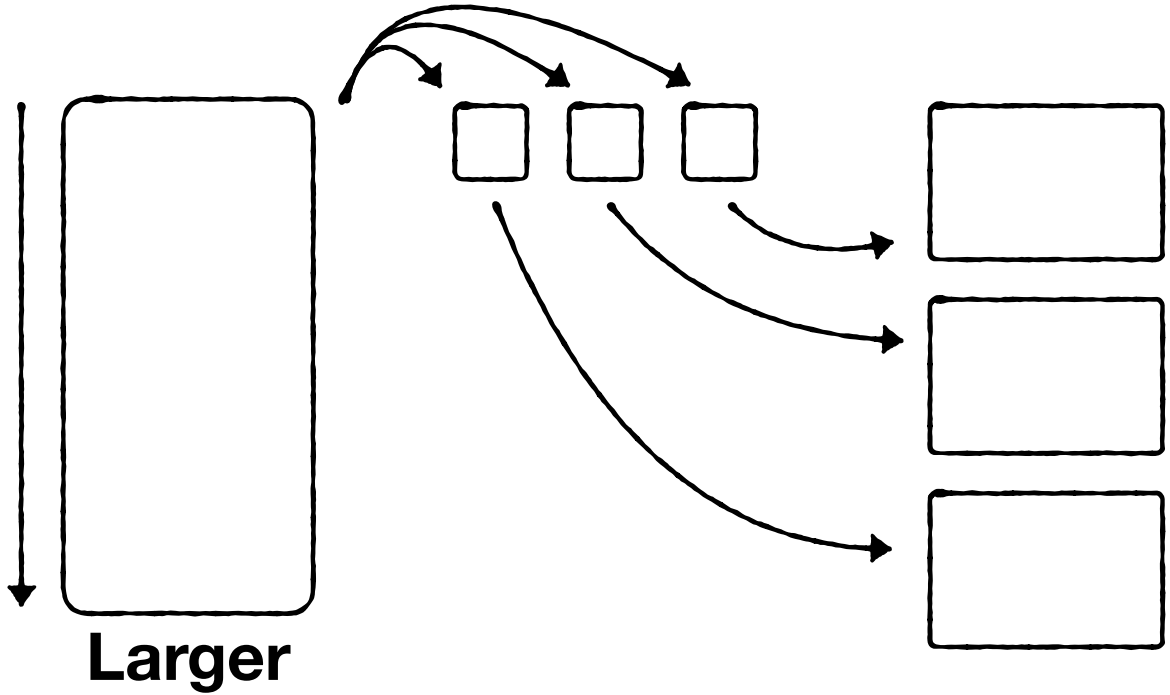
Requires one pass and multiple buffers



Grace Hash Join

(B) Hash partition larger table using same hash function

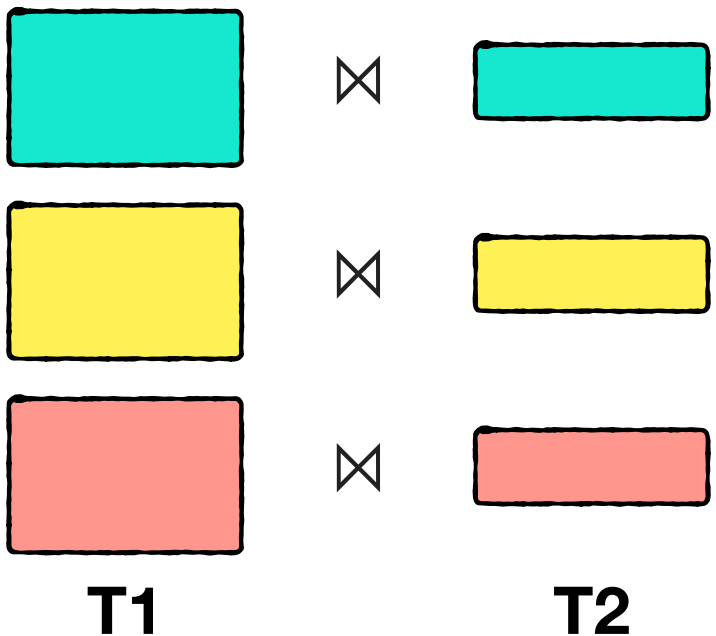
For larger table, each partition can be larger than memory.



Grace Hash Join

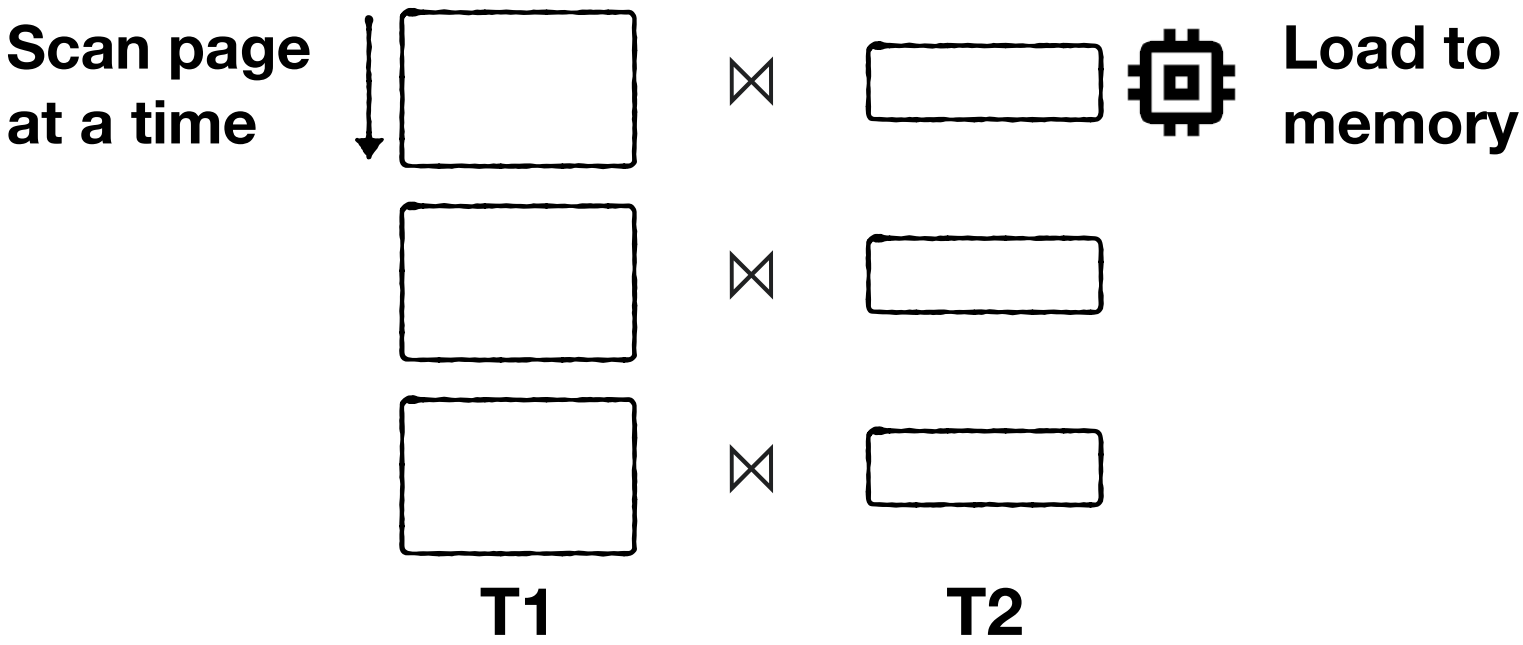
(C) Join each pair of matching partitions independently

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



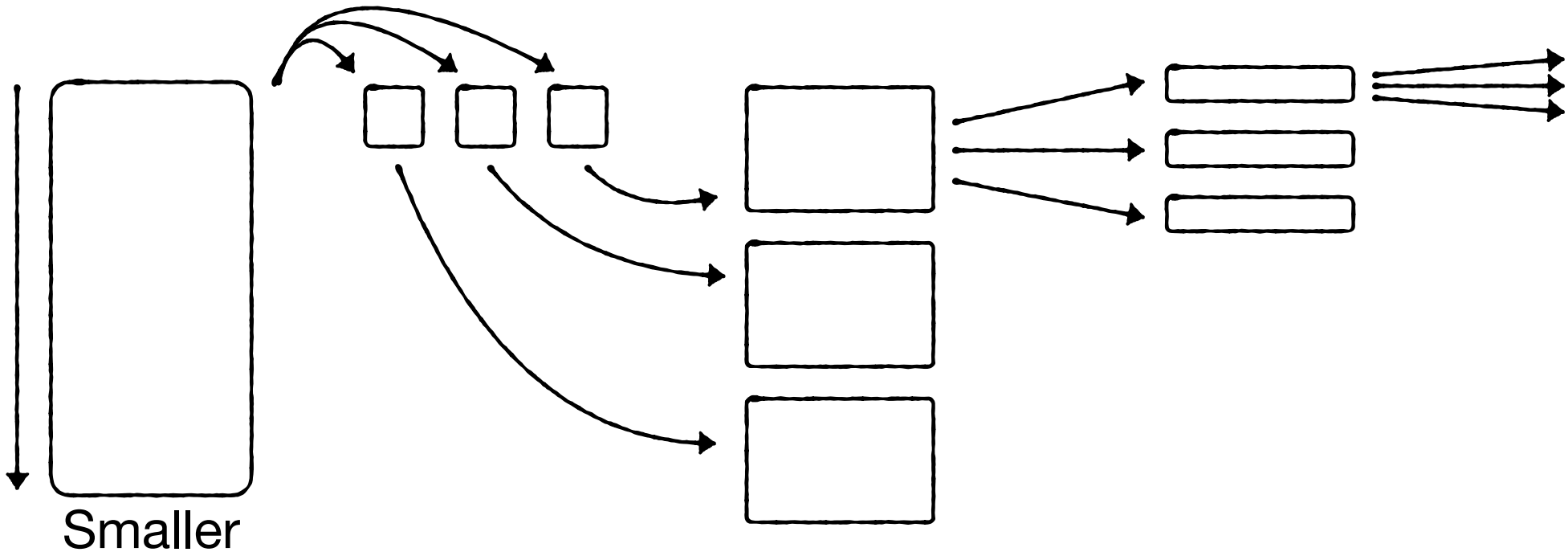
Grace Hash Join

(C) Join each pair of matching partitions independently



Grace Hash Join Analysis

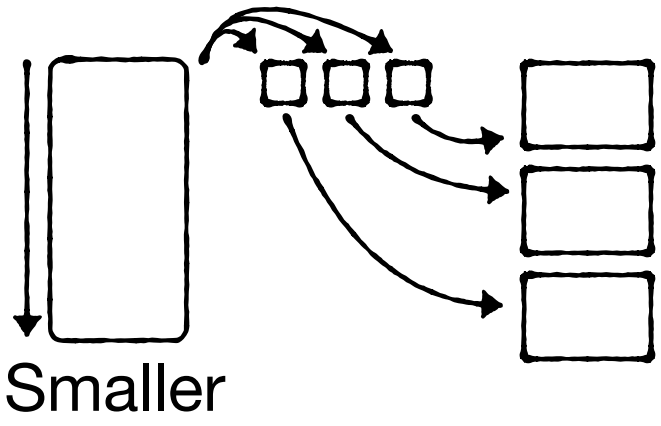
When partitioning smaller table, number of available buffers dictates how much iterations we need until all partitions it in memory



Grace Hash Join Analysis

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

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



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

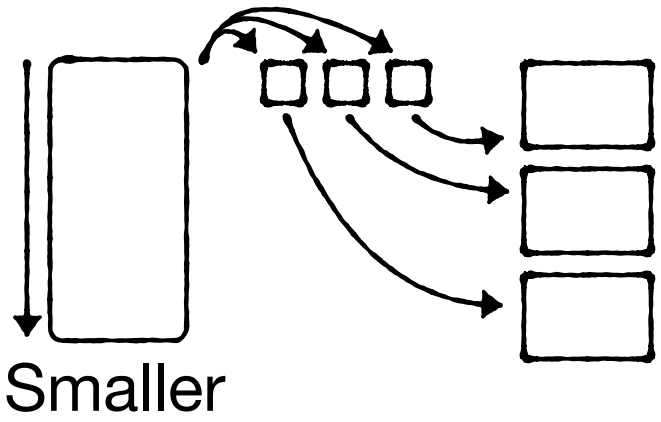
Partitioning fanout

 # 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 much iterations we need until all partitions it in memory

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



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

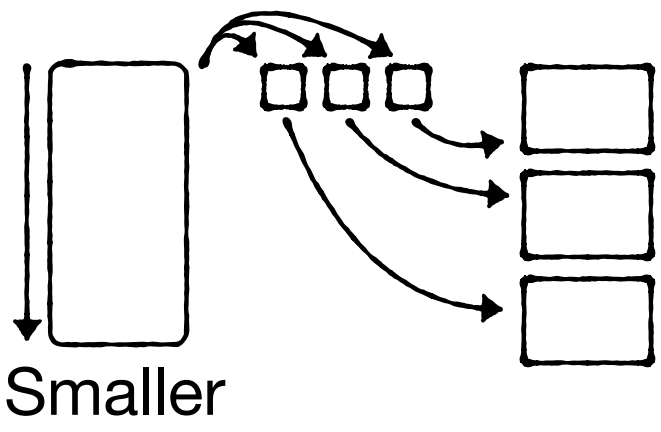
Partitioning Iterations

Joining Iteration

Grace Hash Join Analysis

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

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



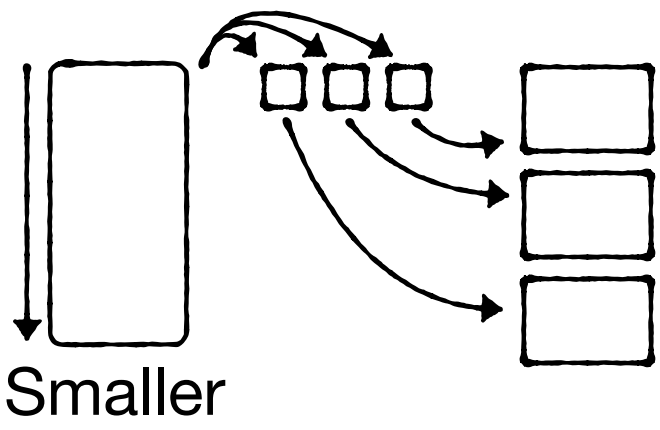
$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{M}\right)+1 \;=\; \log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{B}\right)$$

Same as external merge sort :)

Grace Hash Join Analysis

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

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

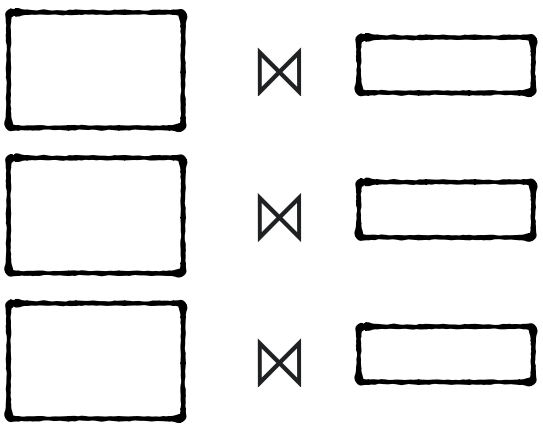


$$\log_{\frac{M}{B}}\left(\frac{|\text{Smaller}|}{B}\right) = 2 \longrightarrow M = \sqrt{|Smaller| \cdot B}$$

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



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 :)

Query Operators

Cardinality Estimation

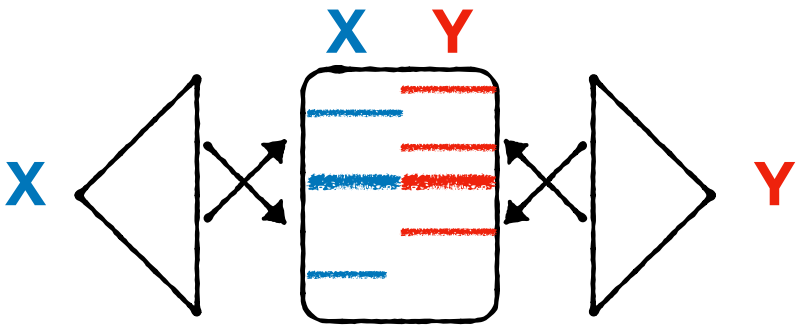
Query Optimization

Query
Operators

Cardinality
Estimation

Query
Optimization

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

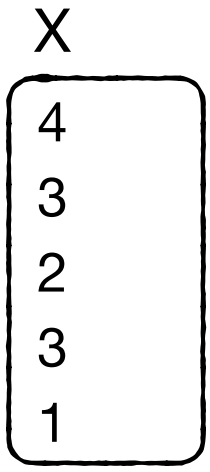
Determining best plan requires estimating $\lvert X_i \rvert$, $\lvert Y_j \rvert$ and $\lvert X_i \cap Y_j \rvert$

Cardinality Estimation

$|X_3|=2$

$|X|=4$

$N=5$



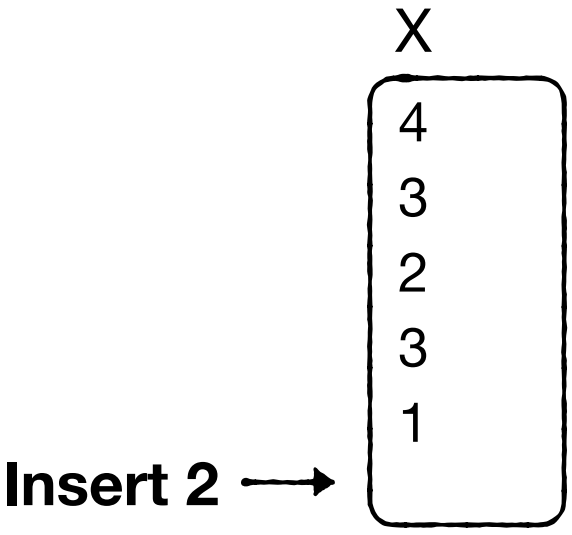
How many rows have a particular value X_i?

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

 # rows

 # unique X values

Cardinality Estimation



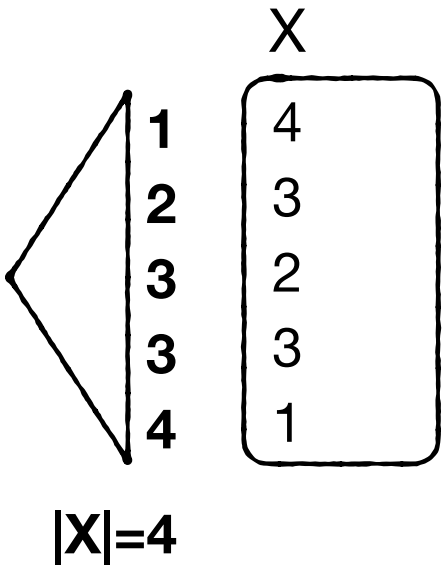
How many rows have a particular value X_i?

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



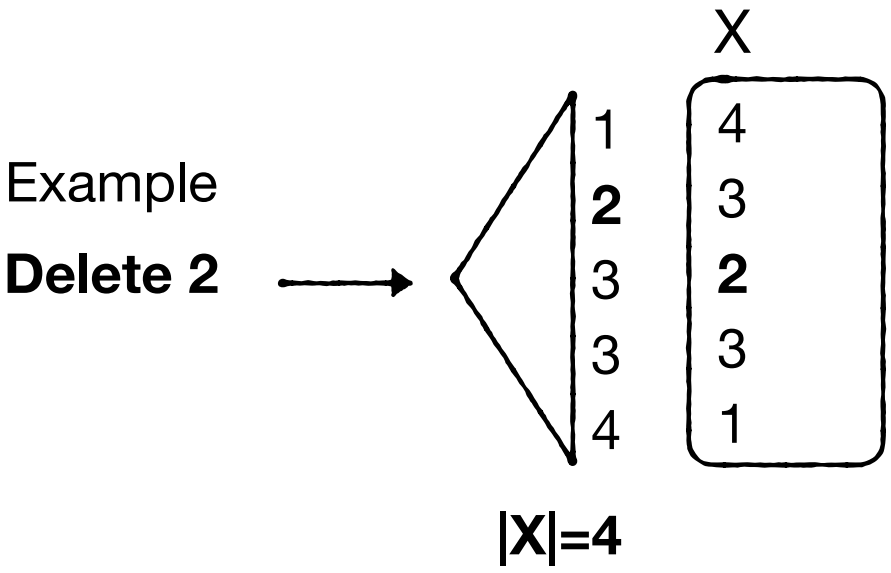
How many rows have a particular value X_i?

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

So we can maintain a cardinality counter for each index

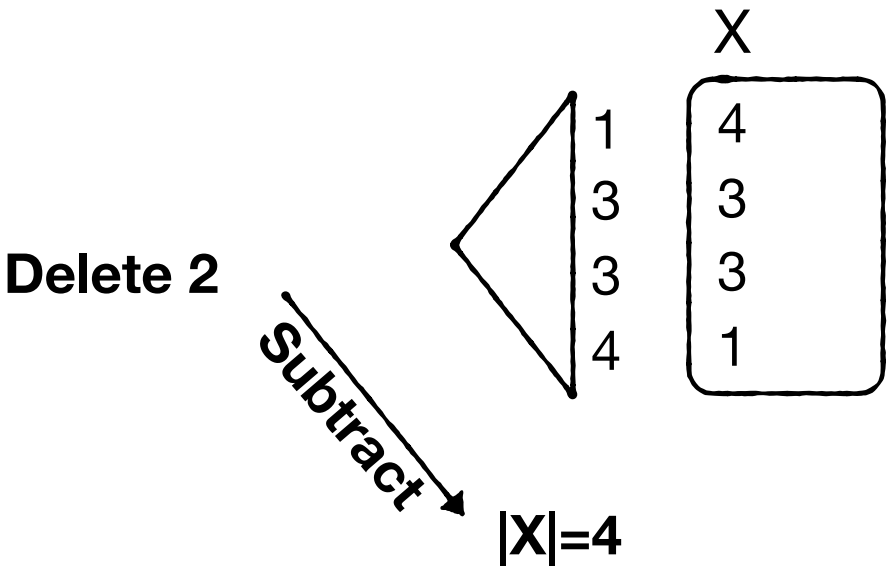
Cardinality Estimation



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

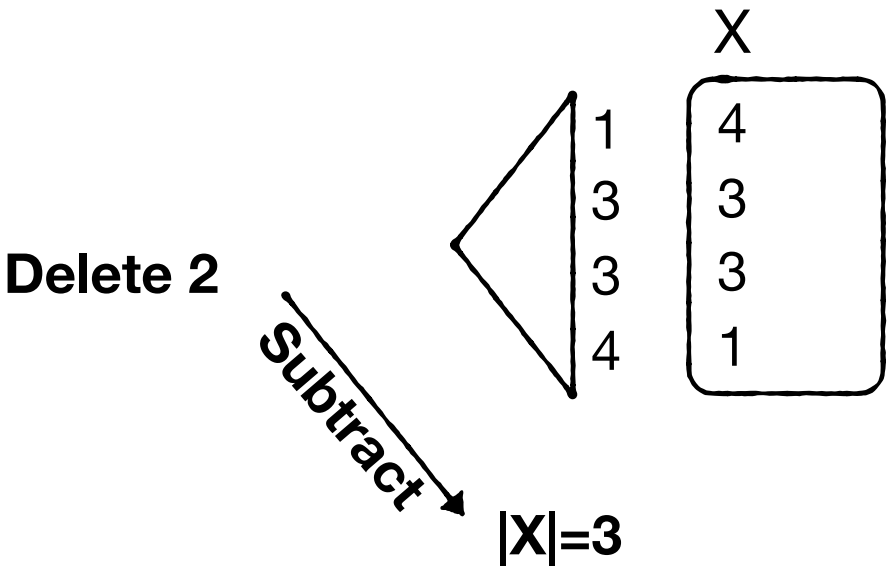
Cardinality Estimation



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

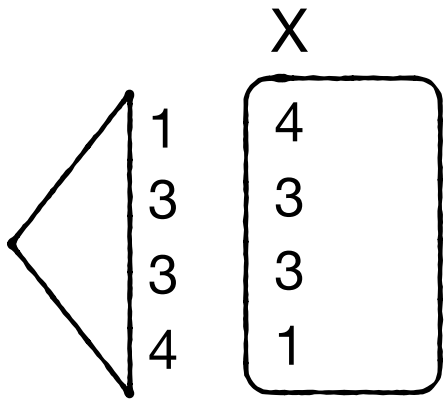
Cardinality Estimation



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

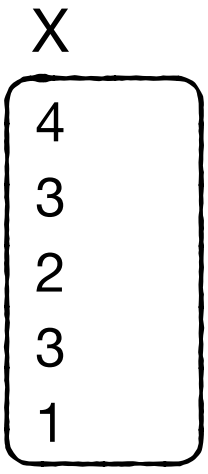
So we can maintain a cardinality counter for each index

Cardinality Estimation



But what if there is no index?

K Minimum Value (KMV) Sketch



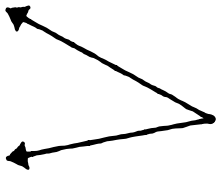
K Minimum Value (KMV) Sketch

X

4
3
2
3
1

Maintain K minimum hashes of all data entries

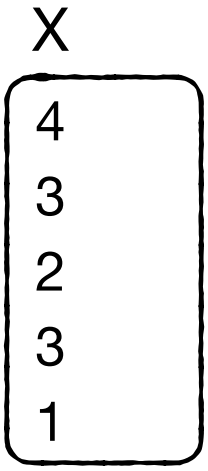
hash(1) = 0101
hash(2) = 1110
hash(3) = 0001
hash(4) = 1010



Assume K=2

2 min hashes

K Minimum Value (KMV) Sketch



Maintain K minimum hashes of all data entries

K=2

0101
0001

Insert 5 where hash(5) = 0011

Update sketch since hash is smaller than current maximum hash

K Minimum Value (KMV) Sketch

X

4
3
2
3
1
5

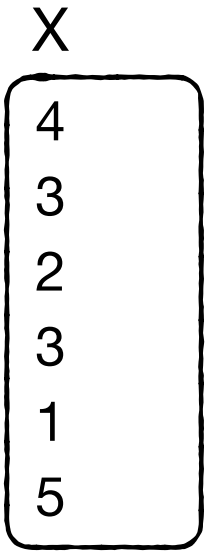
Maintain K minimum hashes of all data entries

K=2

0011
0001

Update sketch since hash is smaller than current maximum hash

K Minimum Value (KMV) Sketch



Maintain K minimum hashes of all data entries

K=2

0011
0001

Insight: adding the same key multiple times does not affect sketch.

Only adding new unique key might affect sketch

K Minimum Value (KMV) Sketch

X

4
3
2
3
1
5

Maintain K minimum hashes of all data entries

K=2

0011
0001

Estimate |X| = (K-1) * (hash key space size) / (max hash value)

= (2-1) * (16) / (3) = 5.3

K Minimum Value (KMV) Sketch

X

4
3
2
3
1
5

Maintain K minimum hashes of all data entries

K=2

0011
0001

Estimate |X| = (K-1) * (hash key space size) / (max hash value)

= (2-1) * (16) / (3) = 5.3

Standard error: $\frac{1}{\sqrt{K-2}}$ **(In practice, K should be > 2)**

K Minimum Value (KMV) Sketch

X

4
3
2
3
1
5

Maintain K minimum hashes of all data entries

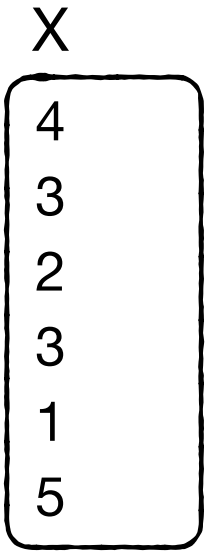
K=2

0011
0001

Estimate |X| = (K-1) * (hash key space size) / (max hash value)

= (2-1) * (16) / (3) = 5.3

We can now estimate any $|X_i|$ as N/|X| = 6/5.3 = 1.13

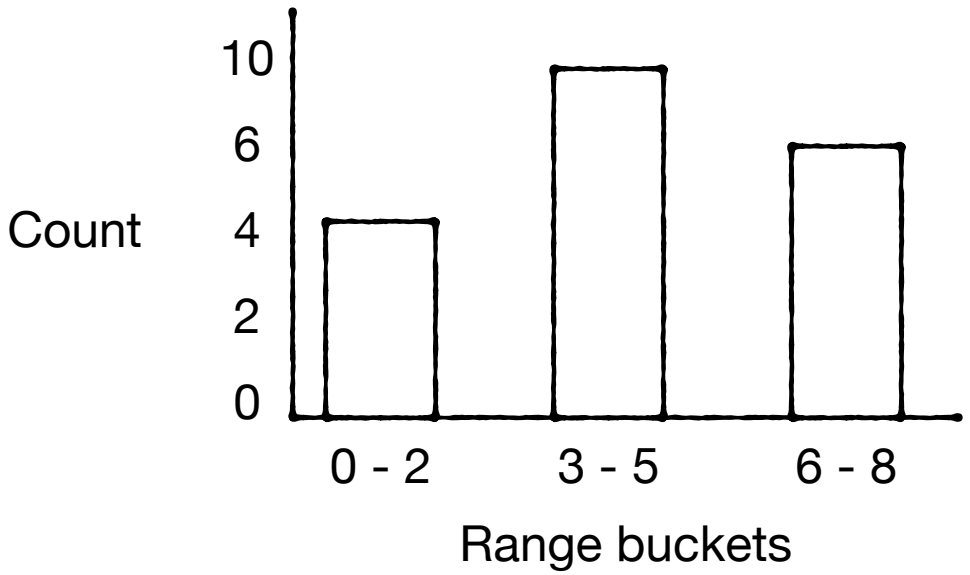


estimate any $|X_i|$ as $N/|X|$

Assumes counts of all values are uniform

Can we do better?

Histogram

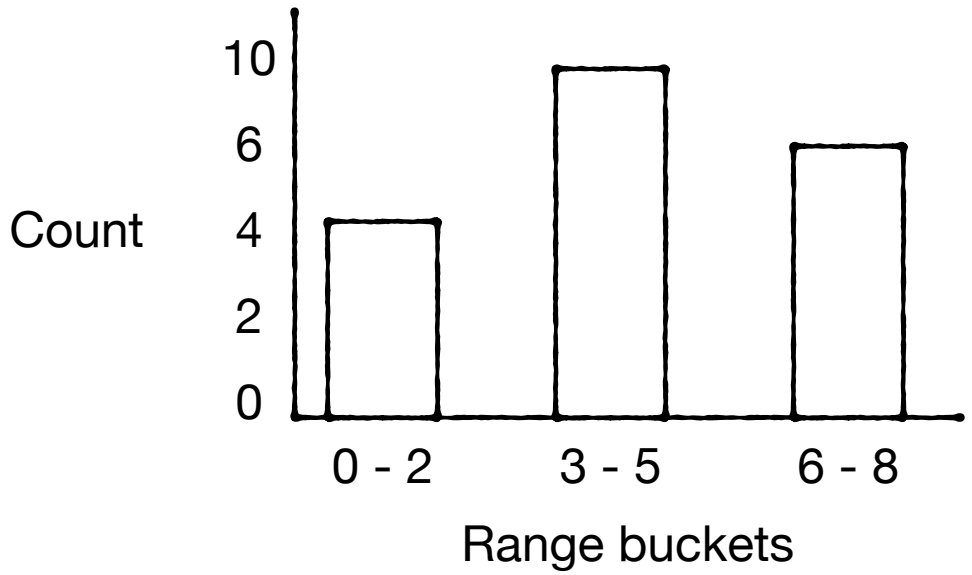


Equi-width buckets
(Same range size)

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

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

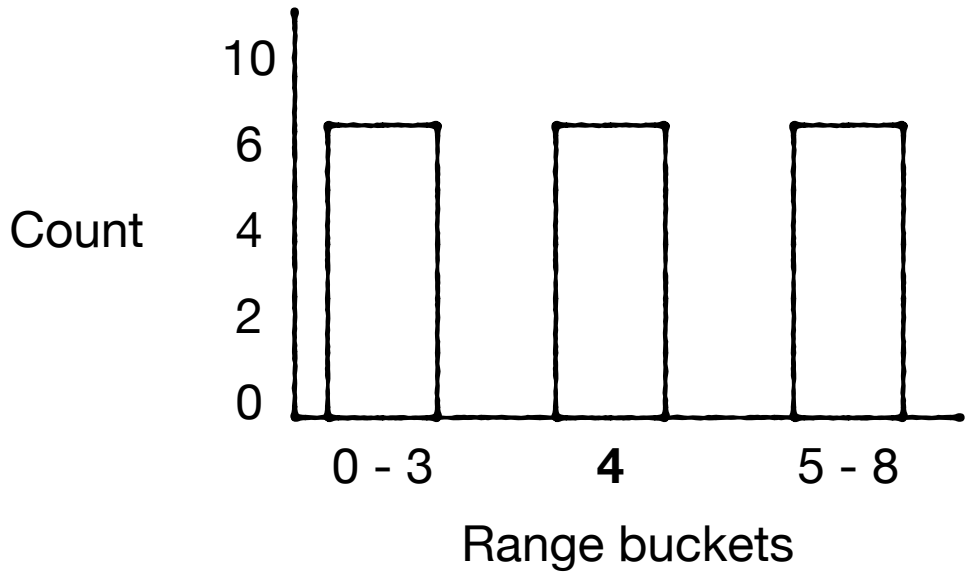
Histogram



Equi-width buckets
(Same range size)

We may underestimate very frequent value due to division by bucket range size, e.g., $|X_4|$ may much higher than estimate

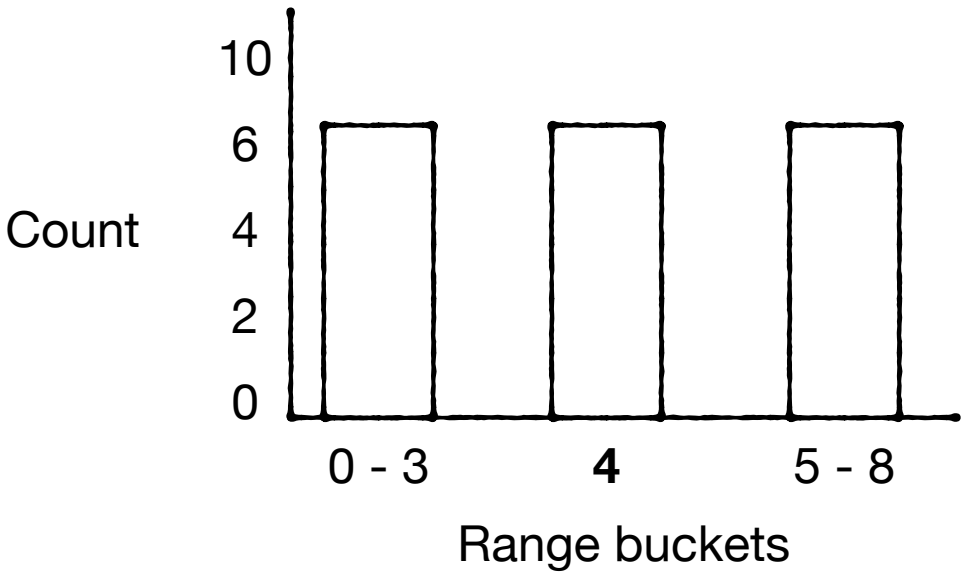
Histogram



Equi-depth buckets
(Same count size)

May over-estimate counts for infrequent values, but this is a better compromise for upper bounding our plan costs.

Histogram

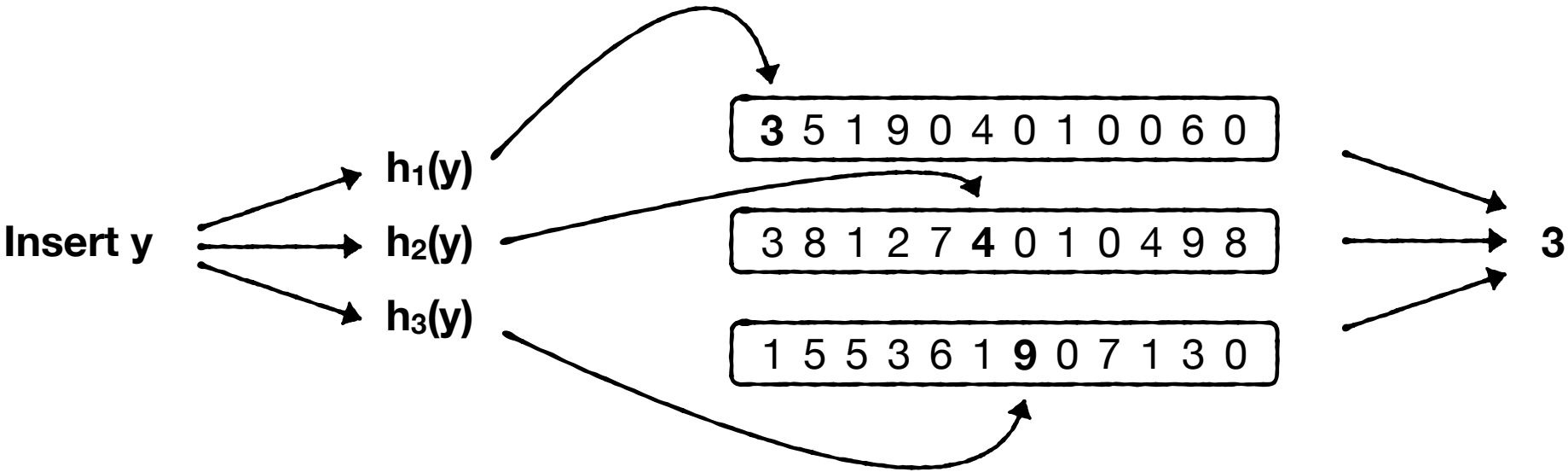


Equi-depth buckets
(Same count size)

Downside: $\log_2$(# buckets) CPU rather than O(1)

Count-min

Also applicable and likely more accurate than histograms, yet more expensive to query.

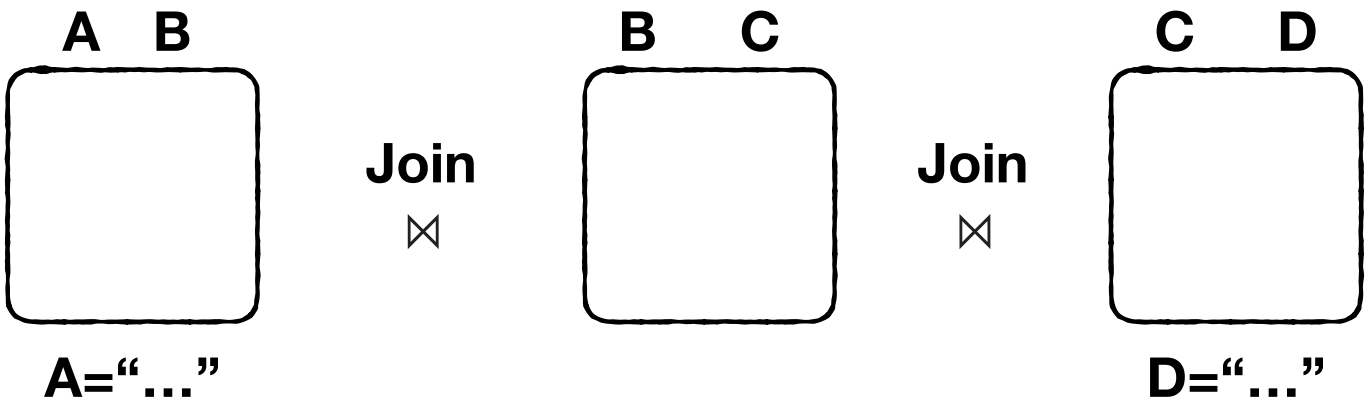


Query
Operators

Cardinality
Estimation

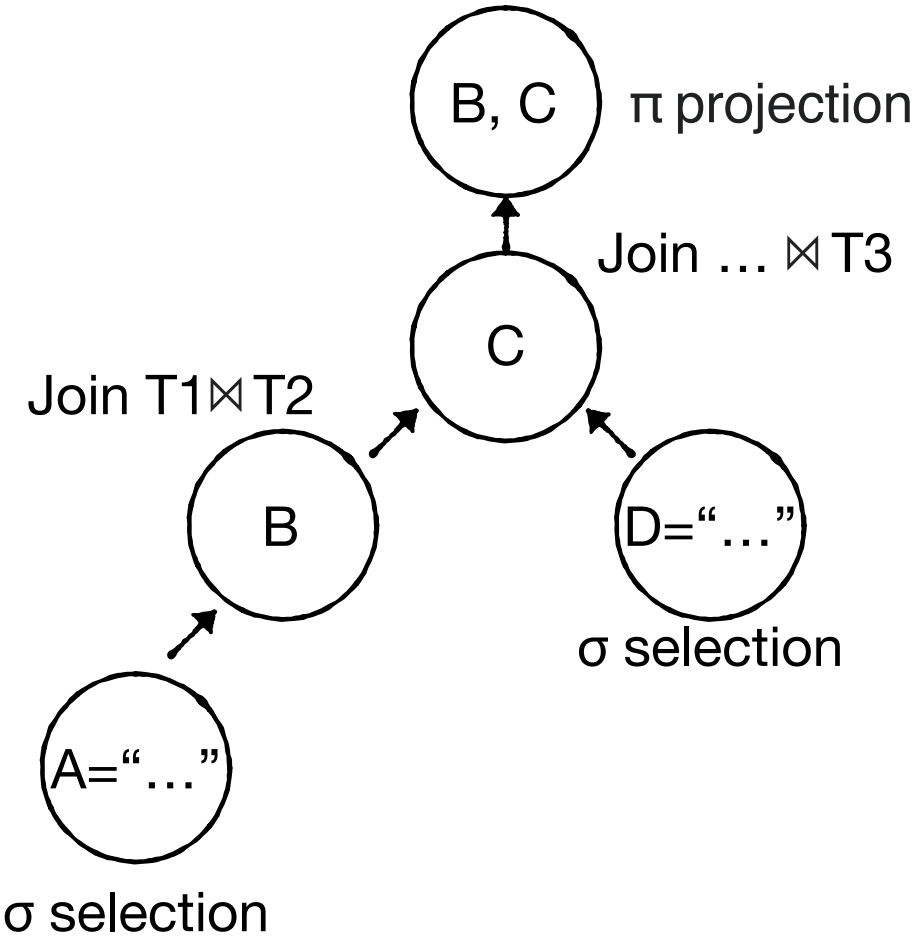
Query
Optimization

Query Optimization

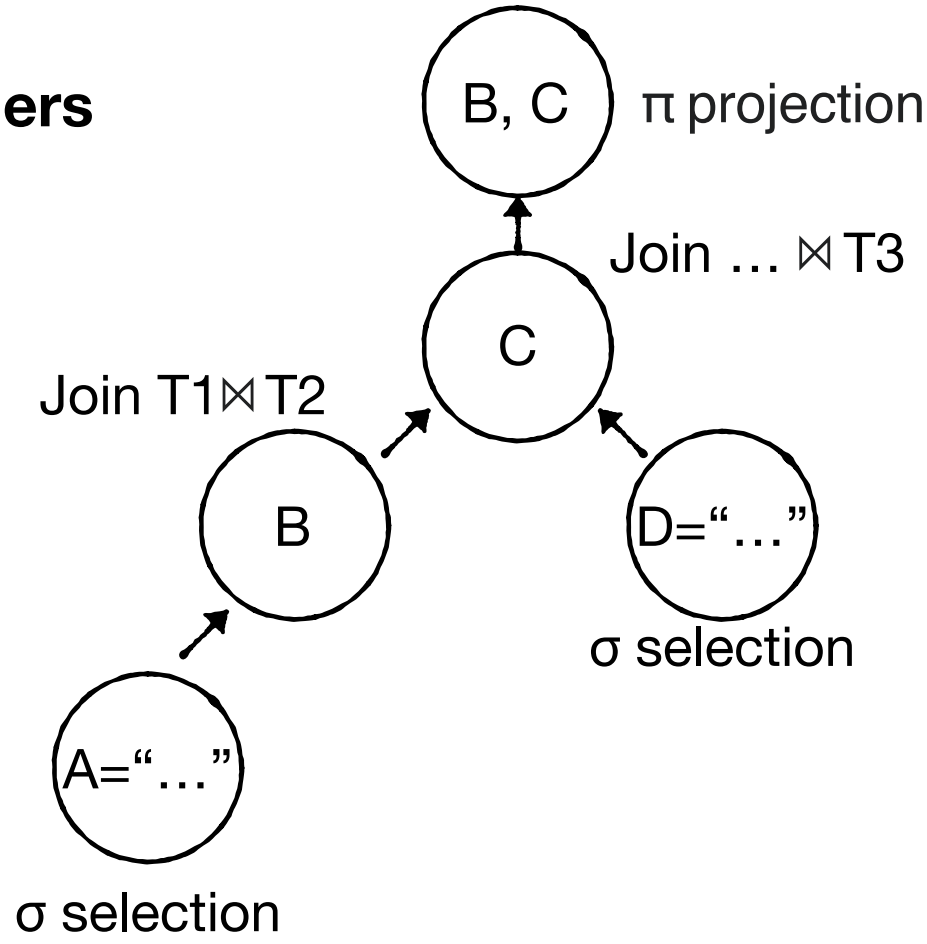


Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"

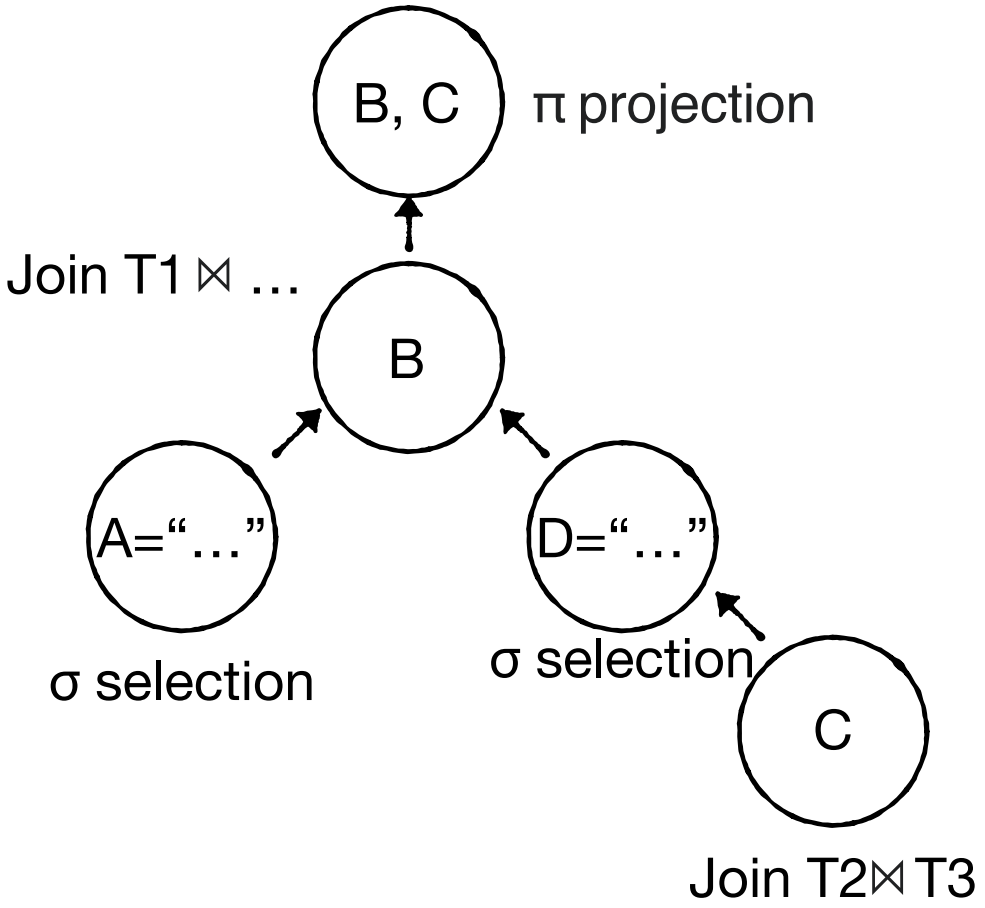
Select A, D from T1, T2, T3 where
T1.B = T2.B and T2.C = T3.C
and A="…" and D="…"



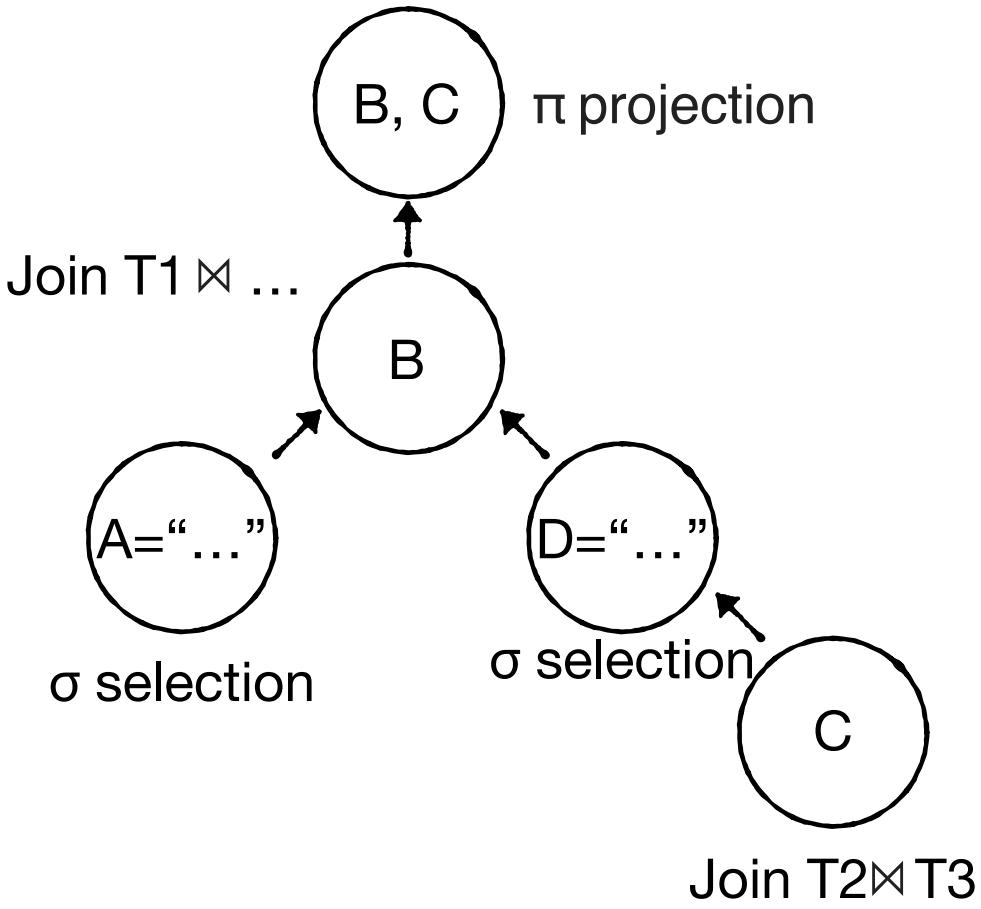
Can join tables in different orders

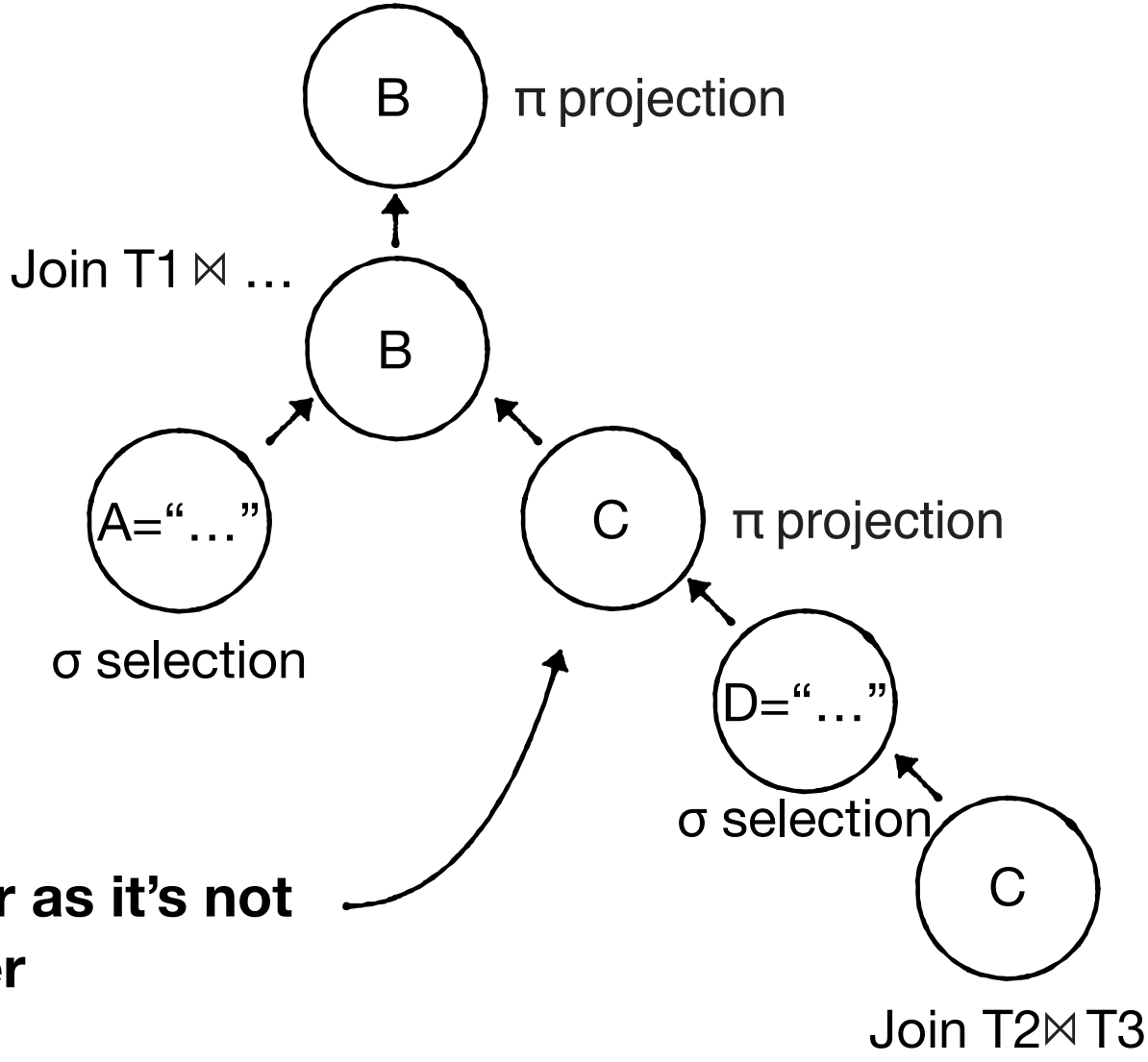


Can select in different orders



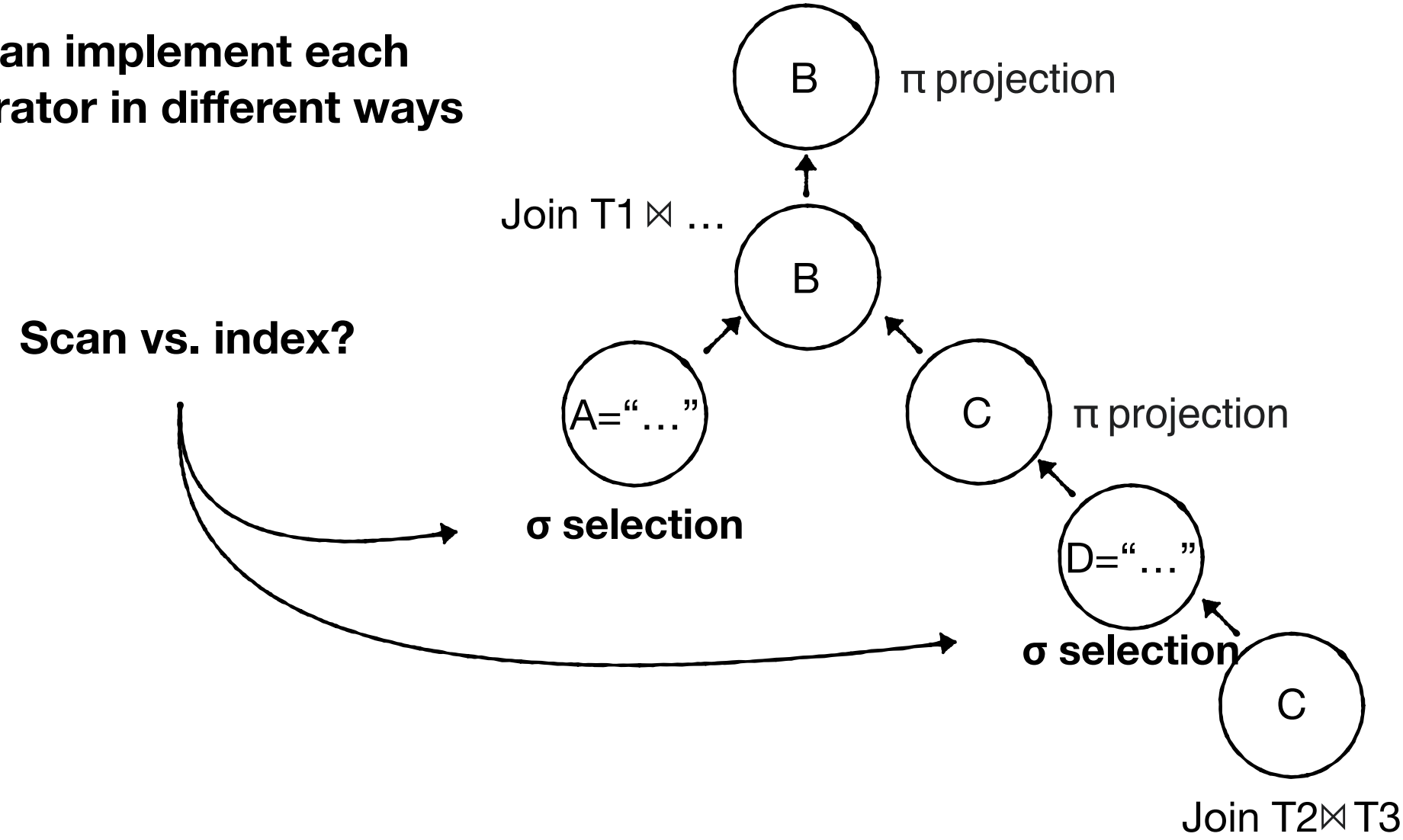
Can project columns in different orders as long as selections or joins do not rely on them



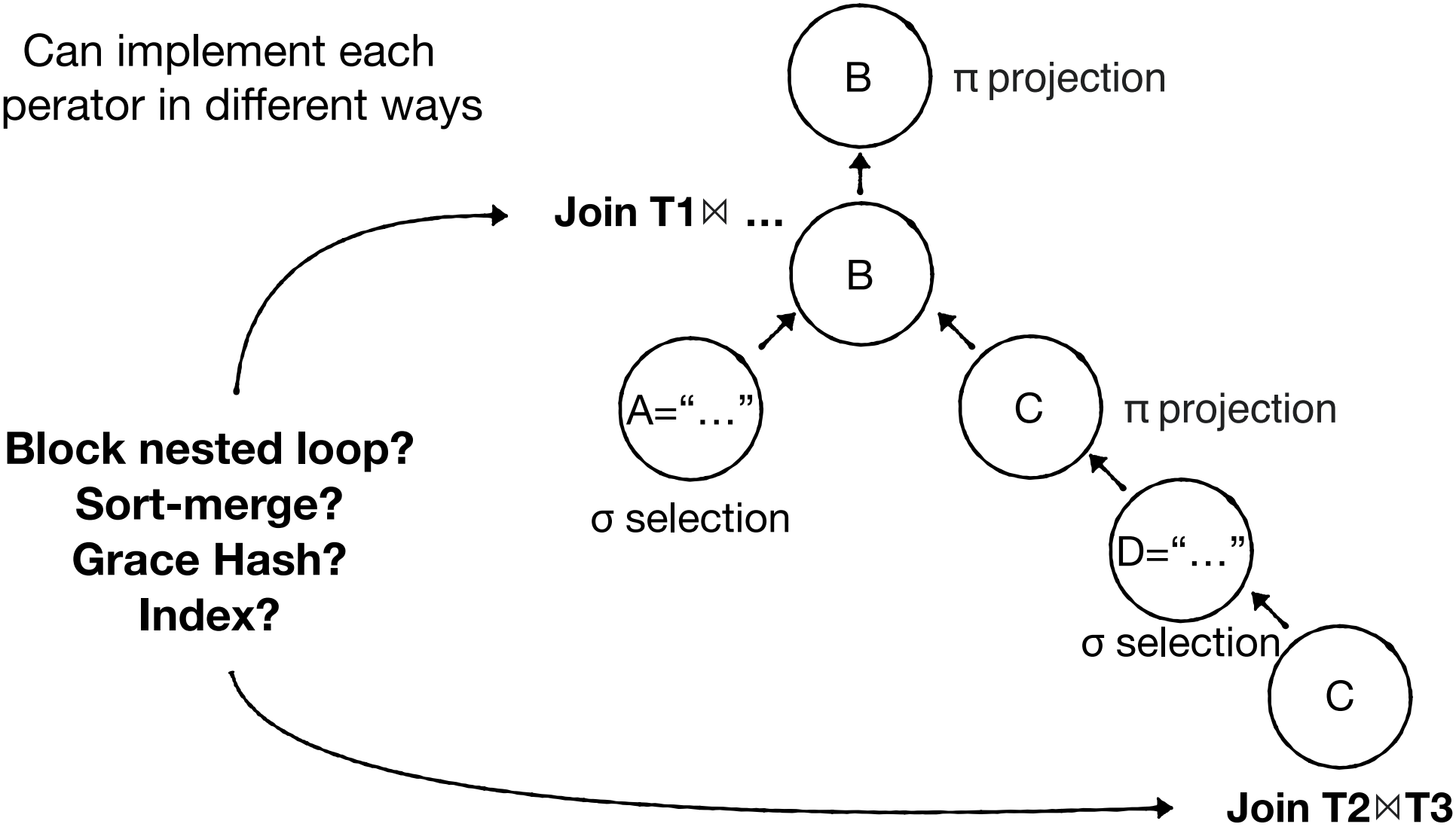


Drop column C earlier as it's not needed later

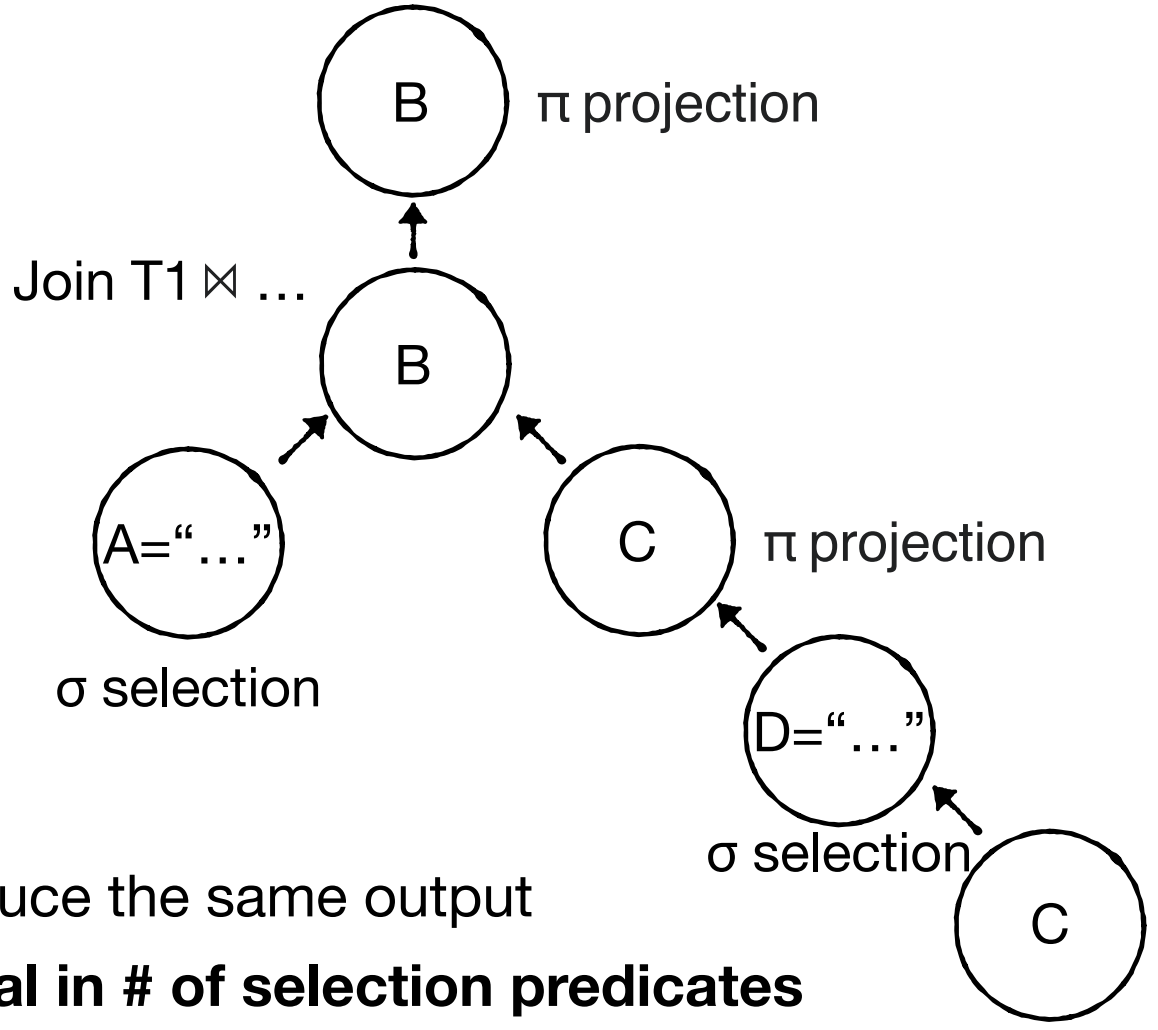
Can implement each operator in different ways



Can implement each operator in different ways



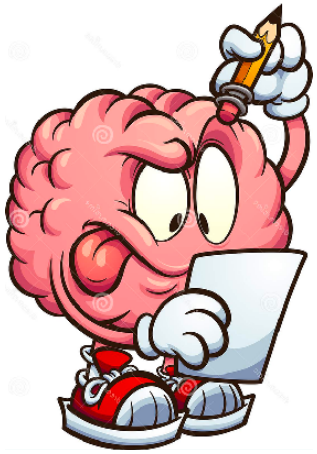
Block nested loop?
Sort-merge?
Grace Hash?
Index?

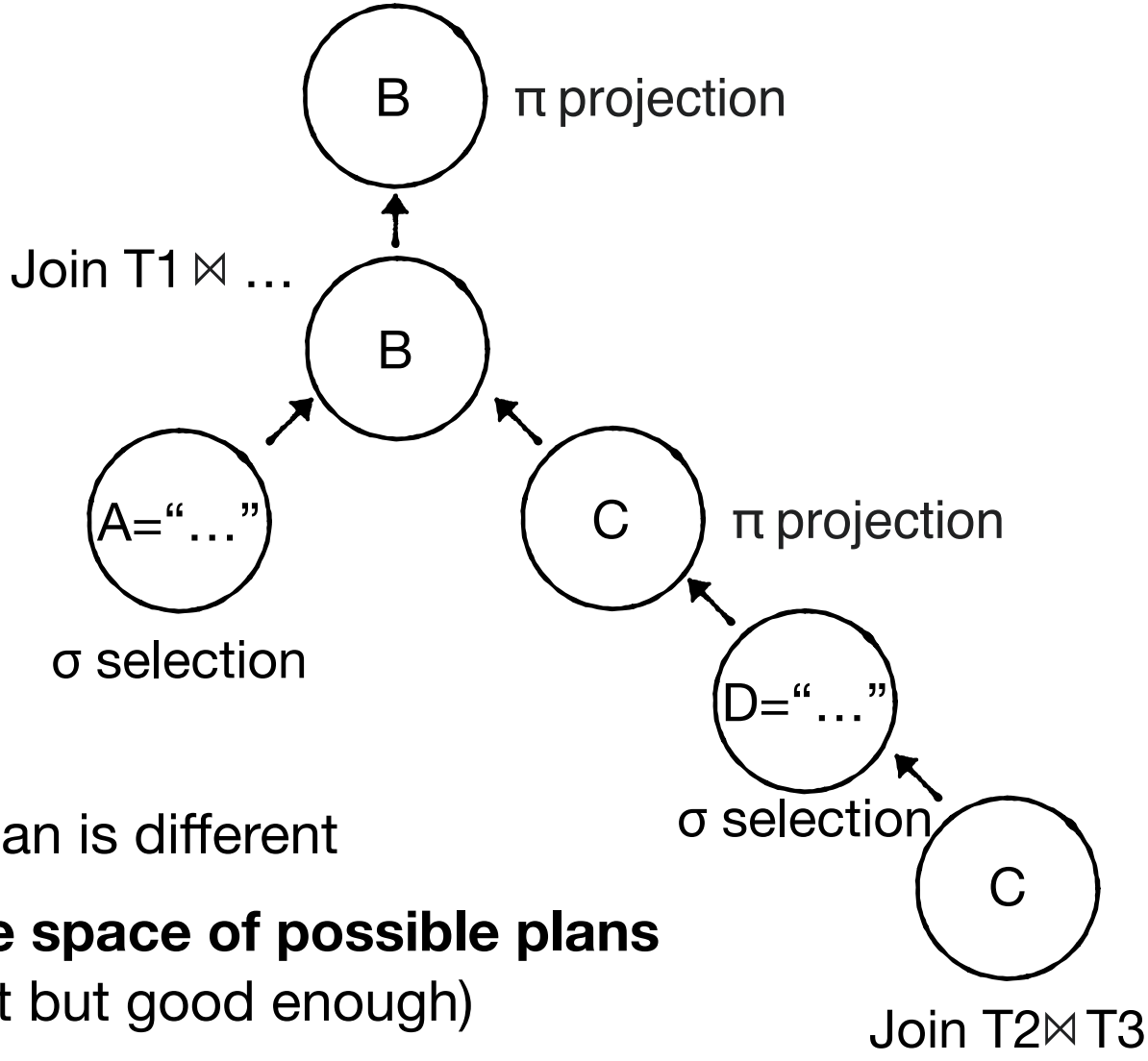


These query plans all produce the same output

 # possible plans is exponential in # of selection predicates

(you can permute them)

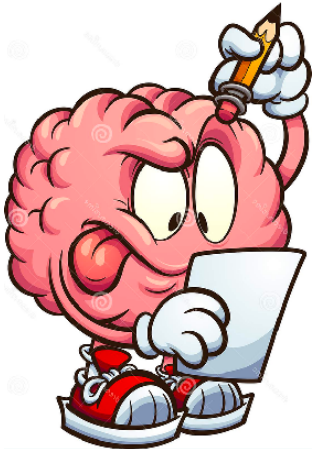




However, the cost of each plan is different

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

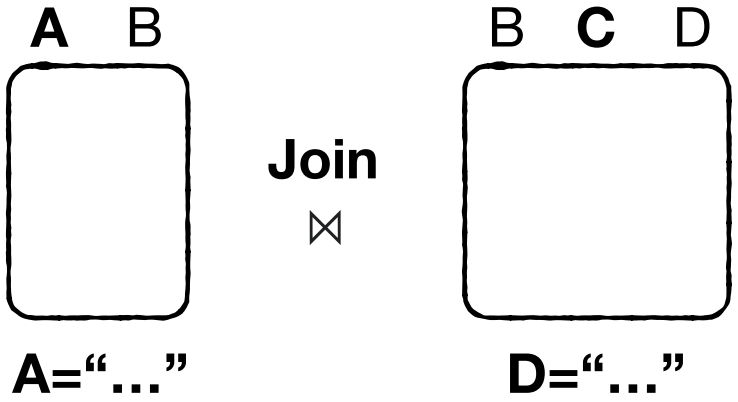
Query Optimization Principles



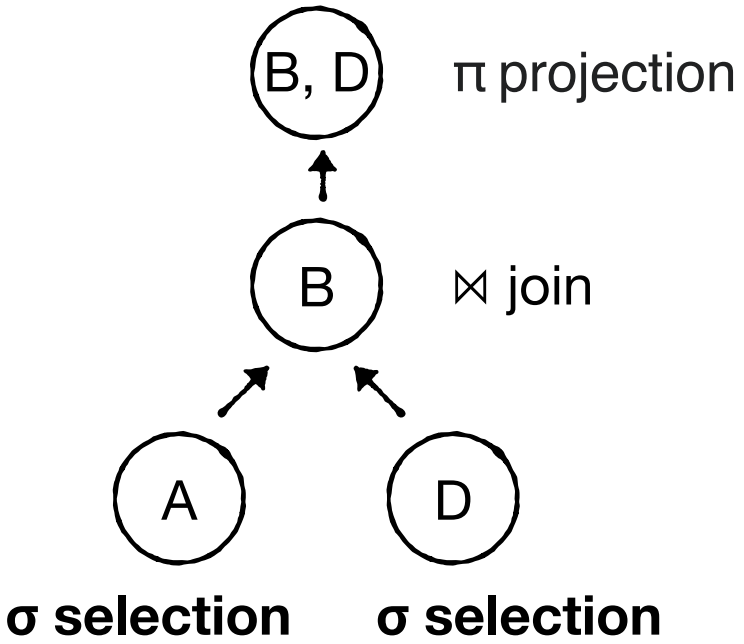
1. Pushing Selections & Projections
2. Operator Fusion
3. Consciousness of next operator
4. Restrict join space

Pushing Selections & Projections

Goal: reducing the data size flowing through subsequent operators

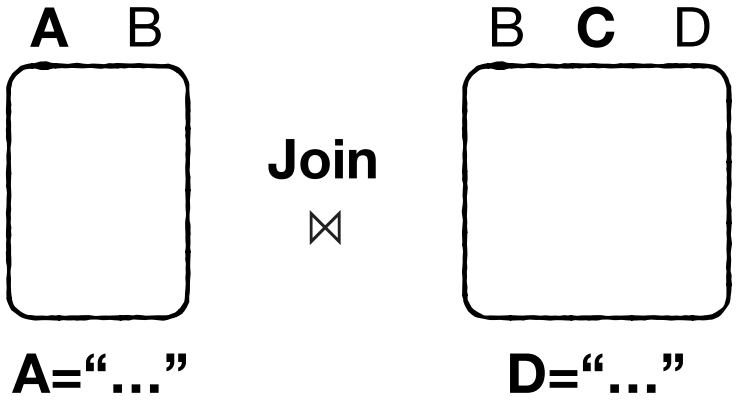


Select A, C from T1, T2 where T1.B = T2.B
and A="…" and D="…"

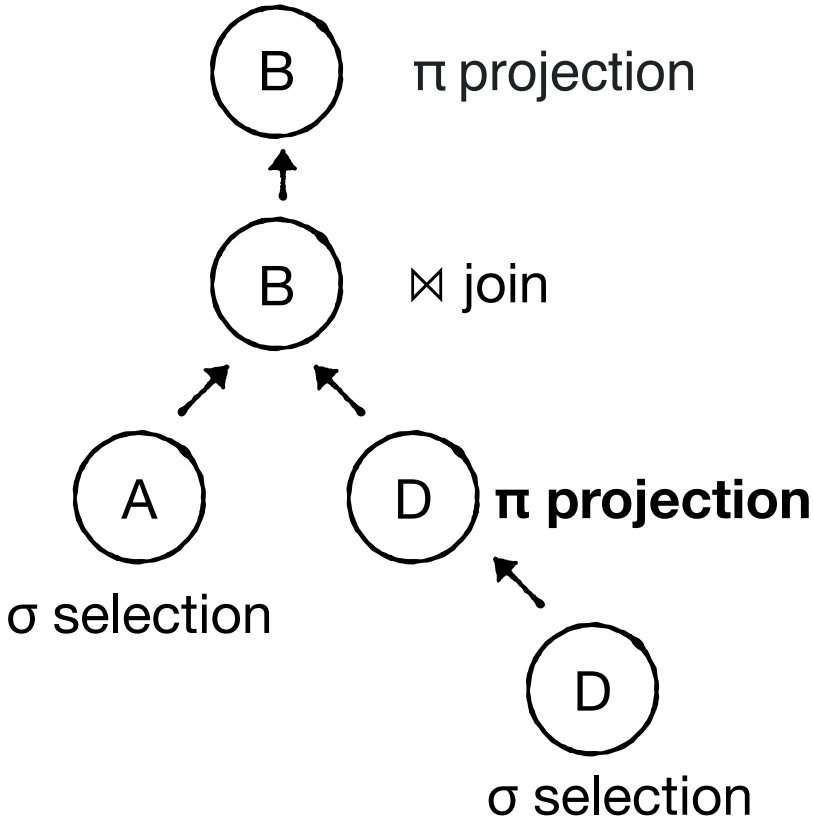


Pushing Selections & Projections

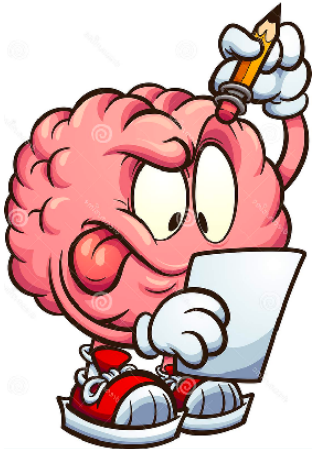
Goal: reducing the data size flowing through subsequent operators



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



Query Optimization Principles



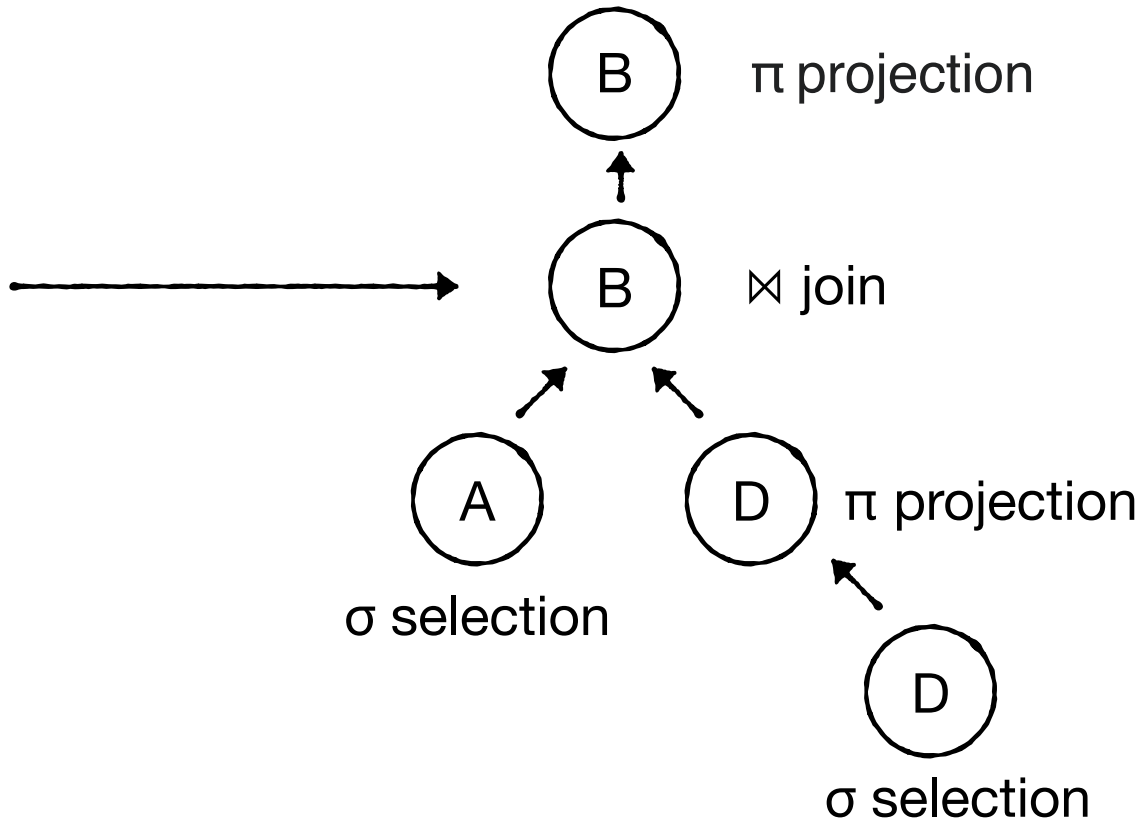
1. Pushing Selections & Projections
2. Operator Fusion
3. Consciousness of next operator
4. Restrict join space

Operator Fusion

Goal: combining work from across multiple logical operators

Example:

Suppose we employ two-pass Grace Hash Join



Operator Fusion

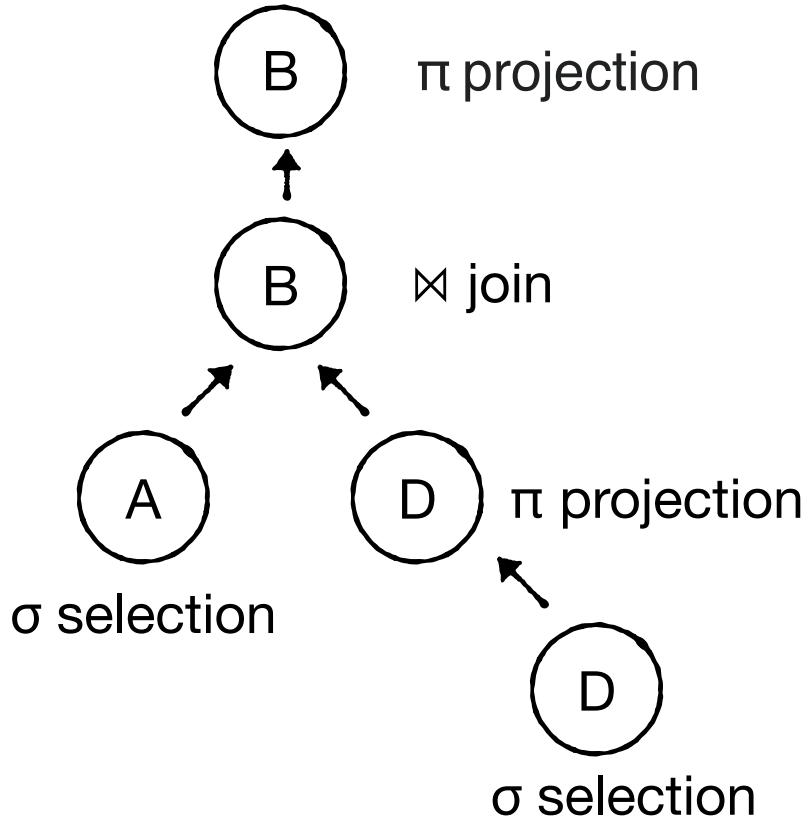
Goal: combining work from across multiple logical operators

Example:

Two Pass Grace Hash Join

(1) Partitioning pass

(2) Joining pass



Operator Fusion

Goal: combining work from across multiple logical operators

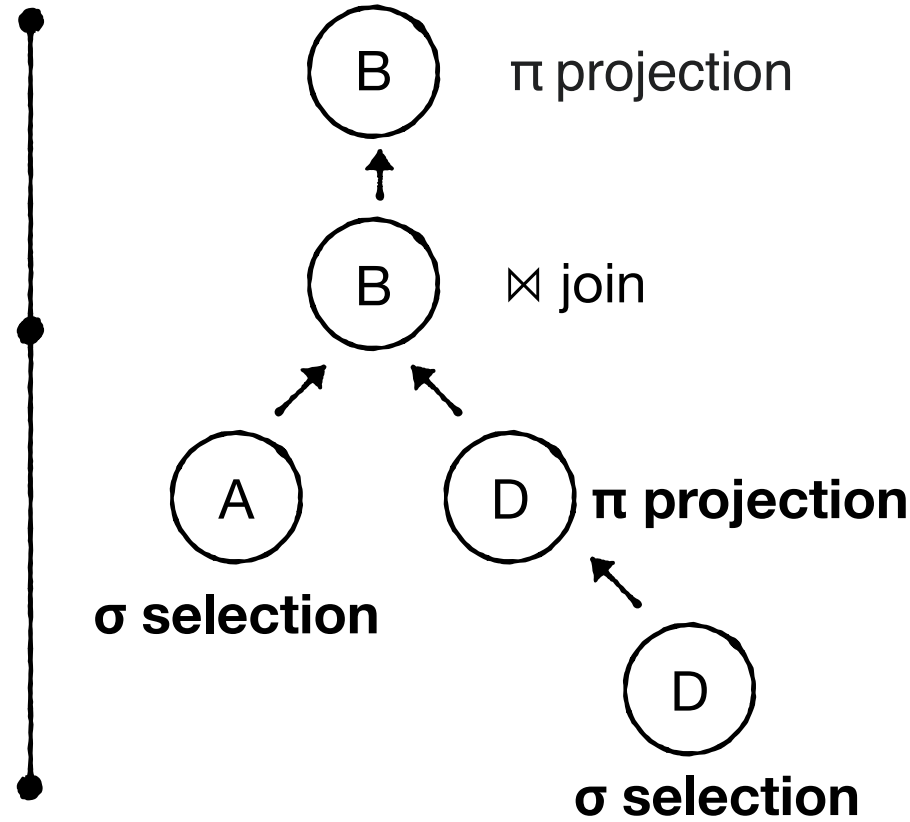
Example:

(2) Joining pass

Project out B for tuples as they come out of join

(1) Partitioning pass

Select over A and D and project out D as a part of this pass



Operator Fusion

Goal: combining work from across multiple logical operators

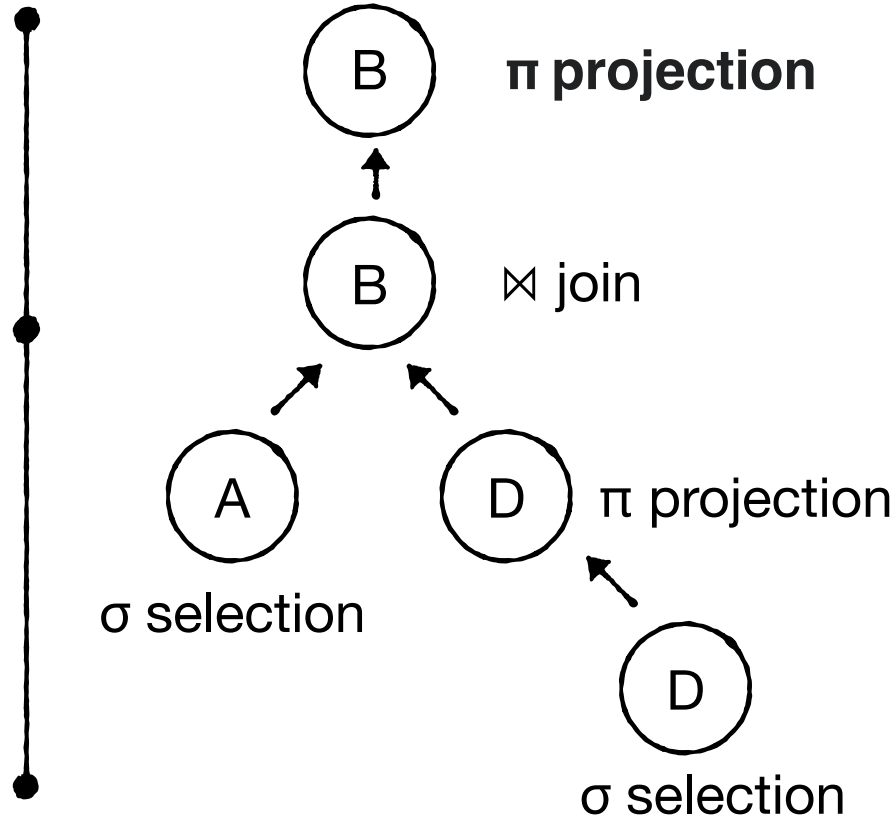
Example:

(2) Joining pass

Project out B for tuples as they come out of join

(1) Partitioning pass

Select over A and D and project out D as a part of this pass

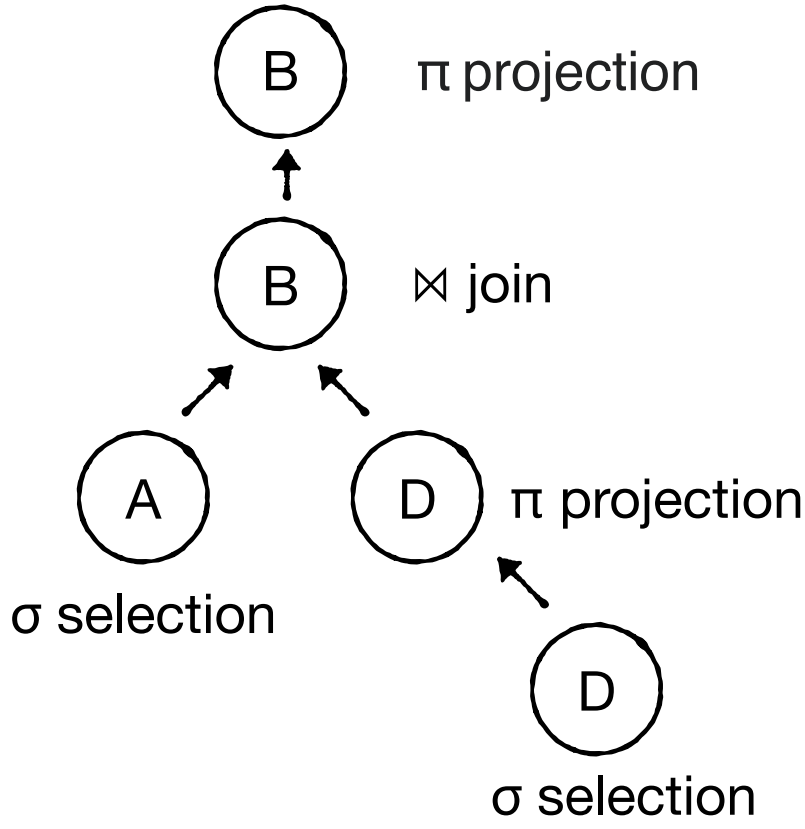


Operator Fusion

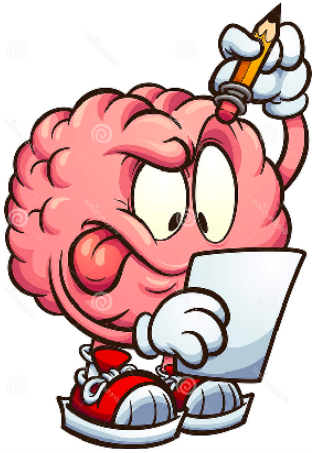
Goal: combining work from across multiple logical operators

Example:

We can process the whole query in just two passes :)



Query Optimization Principles

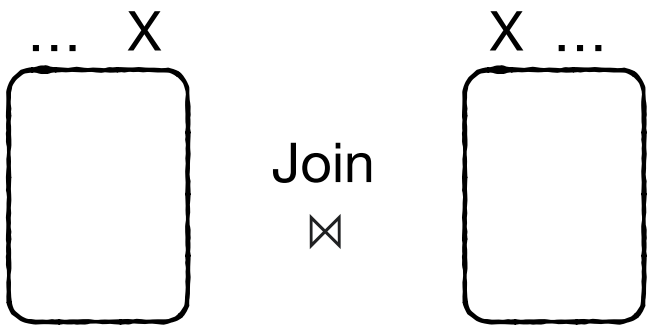


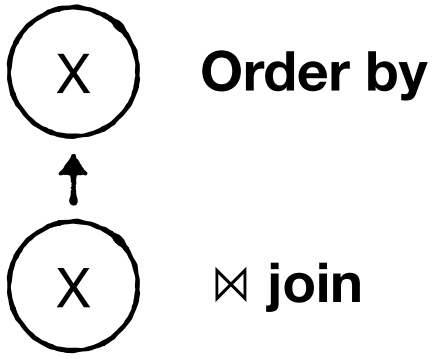
1. Pushing Selections & Projections
2. Operator Fusion
3. Consciousness of Next Operator
4. Restrict join space

Consciousness of Next Operator

Choosing a given operator now may cheapen subsequent ones

Example:





Select * from T1, T2 where T1.X = T2.X

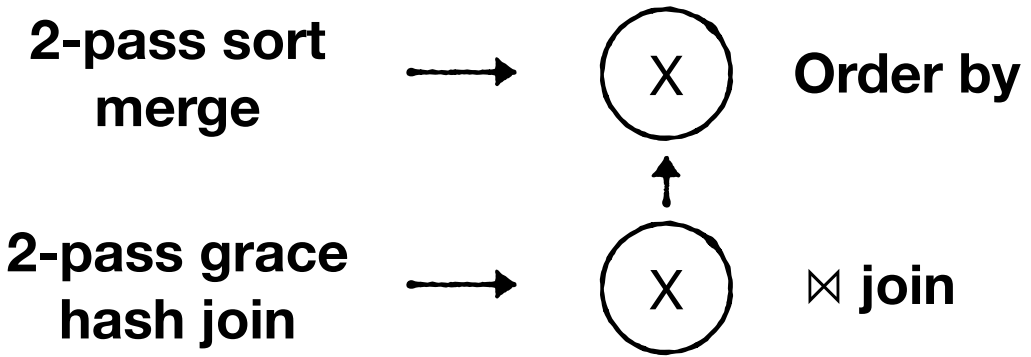
Order by X

Consciousness of Next Operator

Choosing a given operator now may cheapen subsequent ones

Example:

A plausible plan:



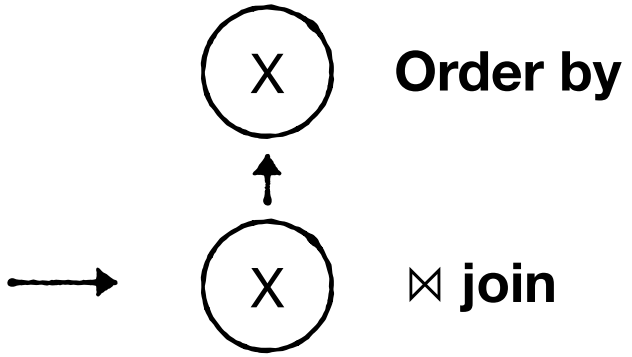
Consciousness of Next Operator

Choosing a given operator now may cheapen subsequent ones

Example:

A better plan:

2-pass sort merge join

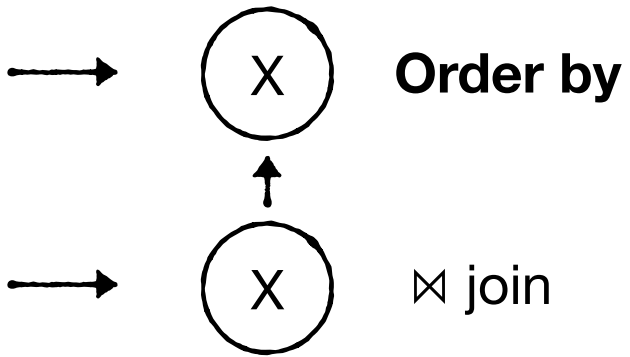


Consciousness of Next Operator

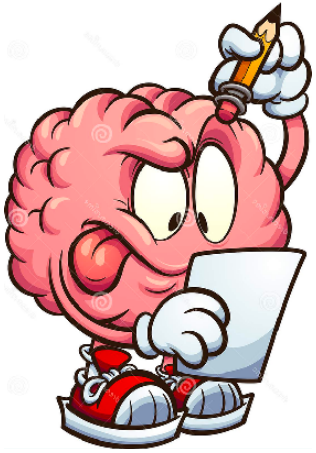
Choosing a given operator now may cheapen subsequent ones

Example:

A better plan:



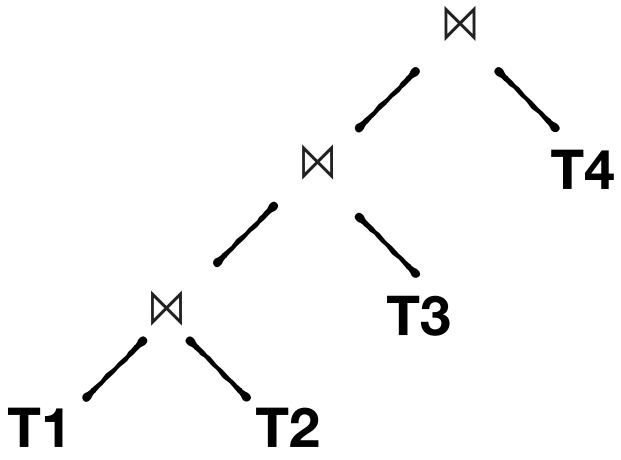
Query Optimization Principles



1. Pushing Selections & Projections
2. Operator Fusion
3. Consciousness of Next Operator
4. Restrict Join Space

Restrict Join Space

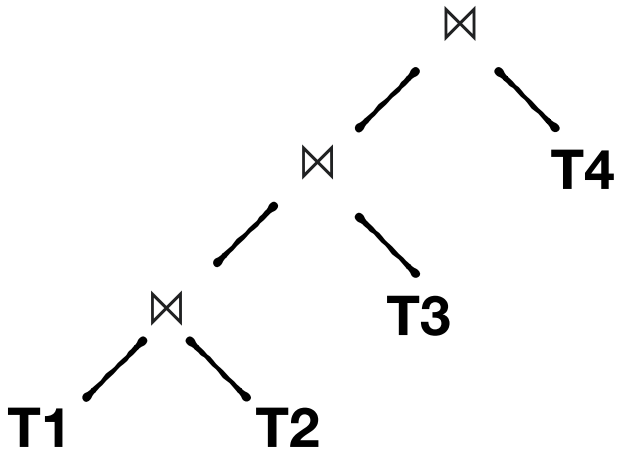
We can execute a complex join in many different orders



Optimizers only consider left-deep joins to restrict the search space

Restrict Join Space

We can execute a complex join in many different orders



Optimizers only consider left-deep joins to restrict the search space

Easy to pipeline join results from left relation