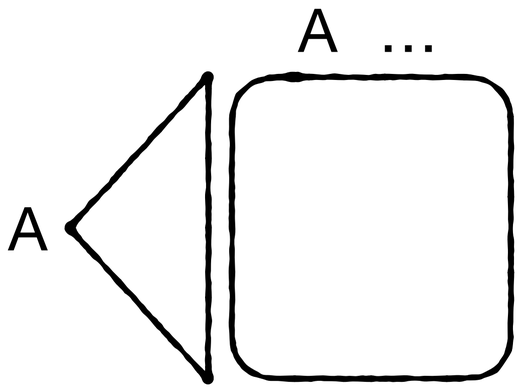
Tutorial on Query Evaluation

CSC443H1 Database System Technology

Niv Dayan - March 16, 2023

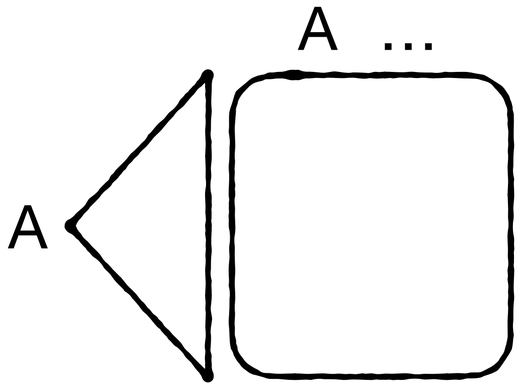
Question 1 - Index vs. Scan

Table T contains 2^{20} rows, and each page fits 2^7 rows. There is an unclustered index on column A with a unique value counter set to 2^{10}. Should we scan the relation or employ the index to answer the following query: select * from T where A="i"? What if the unique element count were 2^5 instead? We are using an SSD.



Question 1 - Index vs. Scan

Table T contains 2^{20} rows, and each page fits 2^7 rows. There is an unclustered index on column A with a unique value counter set to 2^{10}. Should we scan the relation or employ the index to answer the following query: select * from T where A="i"? What if the unique element count were 2^5 instead? We are using an SSD.



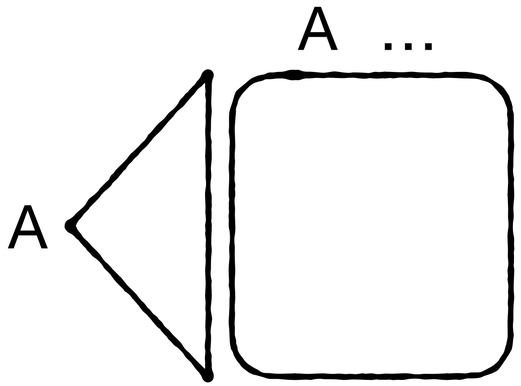
A scan would cost $2^{20}/2^7=2^{13}$ I/Os.

With 2^{10} unique values in col A, we can expect $2^{20}/2^{10}=2^{10}$ rows to match leading to 2^{10} random I/Os. This is better than scanning.

With 2^5 unique values in col A, we can expect $2^{20}/2^5=2^{15}$ rows to match leading to 2^{15} random I/Os. This is worse than scanning.

Question 1 - Index vs. Scan

Table T contains 2^{20} rows, and each page fits 2^7 rows. There is an unclustered index on column A with a unique value counter set to 2^{10}. Should we scan the relation or employ the index to answer the following query: select * from T where A="i"? What if the unique element count were 2^5 instead? We are using an SSD.



A scan would cost $2^{20}/2^7=2^{13}$ I/Os.

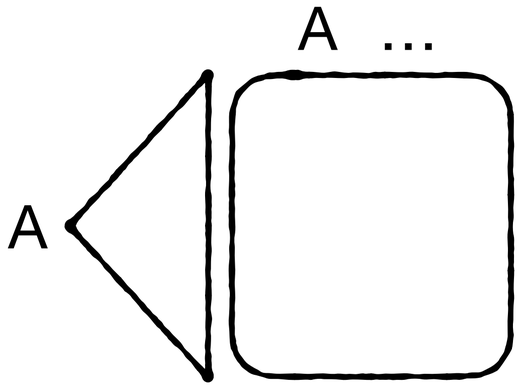
With 2^{10} unique values in col A, we can expect $2^{20}/2^{10}=2^{10}$ rows to match leading to 2^{10} random I/Os. This is better than scanning.

With 2^5 unique values in col A, we can expect $2^{20}/2^5=2^{15}$ rows to match leading to 2^{15} random I/Os. This is worse than scanning.

What if we used disk?

Question 1 - Index vs. Scan

Table T contains 2^{20} rows, and each page fits 2^7 rows. There is an unclustered index on column A with a unique value counter set to 2^{10}. Should we scan the relation or employ the index to answer the following query: select * from T where A="i"? What if the unique element count were 2^5 instead? We are using an SSD.



A scan would cost $2^{20}/2^7=2^{13}$ I/Os.

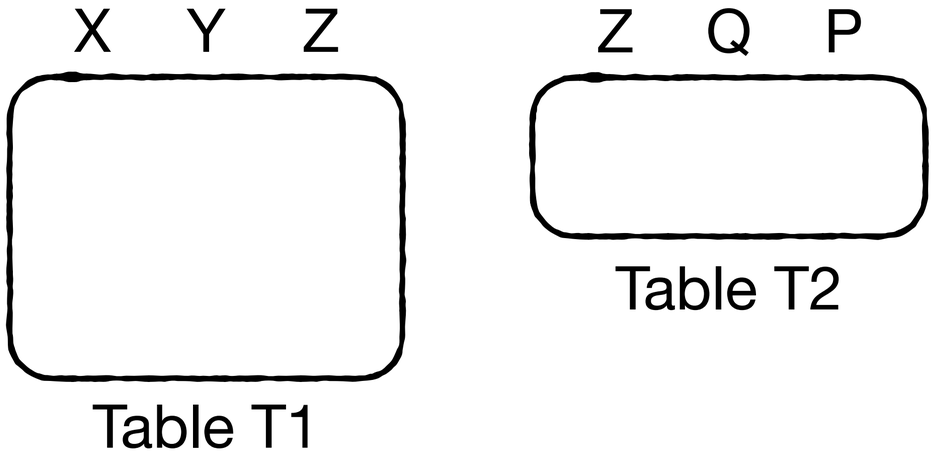
With 2^{10} unique values in col A, we can expect $2^{20}/2^{10}=2^{10}$ rows to match leading to 2^{10} random I/Os. This is better than scanning.

With 2^5 unique values in col A, we can expect $2^{20}/2^5=2^{15}$ rows to match leading to 2^{15} random I/Os. This is worse than scanning.

What if we used disk? A scan may still be best.

Question 2 - Index Design

Consider the following tables and query.



Select X, Y, Q from T1, T2 where X="i" and Y="j" and T1.Z = T2.Z

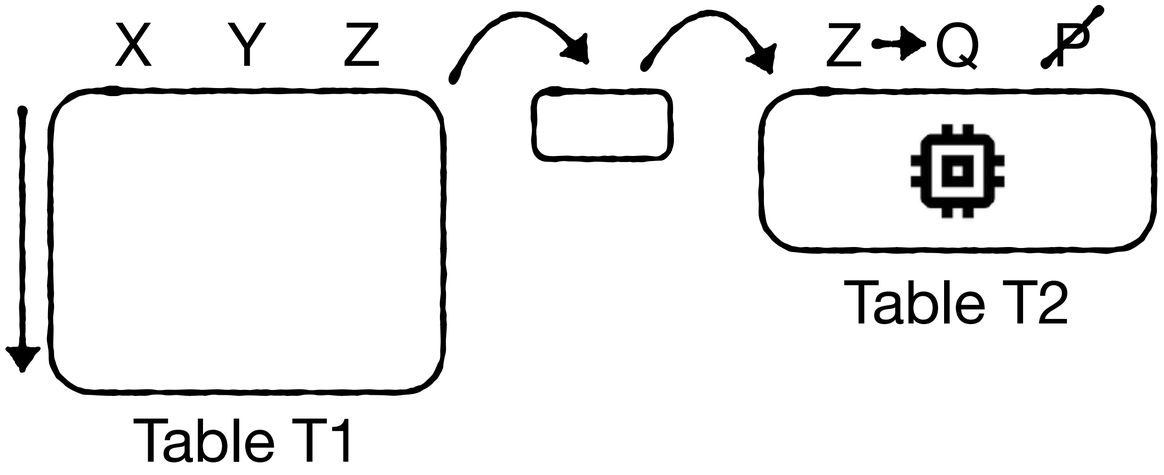
Assume: $|T1| > |T2| > |X_i| > |Y_j| > |X_i \cap Y_j|$

(1) Assume T2 fits in memory. How would we run the query with no indexes present? What's the I/O cost?

(2) Assuming tiny memory, which indexes would you construct in order to speed this query up maximally? What's the I/O cost?

Question 2 - Index Design

Consider the following tables and query.



Select X, Y, Q from T1, T2 where X="i" and Y="j" and T1.Z = T2.Z

(1) Assume T2 fits in memory. How would we run the query with no indexes present? What's the I/O cost?

Scan T2 and build an in-memory hash table mapping Z to Q (omitting P).

Scan T1 using one input buffer. For each row matching X and Y, join to Q using hash table based on Z.

An instance of block-nested loop:

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

Question 2 - Index Design

Consider the following tables and query.

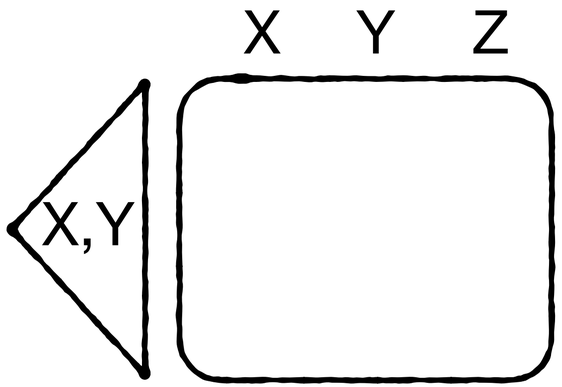


Table T1

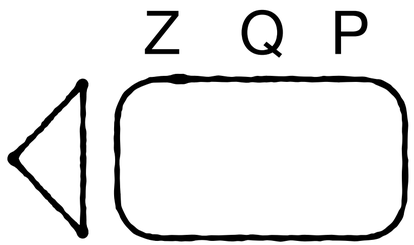


Table T2

Select X, Y, Q from T1, T2 where X="i" and Y="j" and T1.Z = T2.Z

(2) Assuming tiny memory, which indexes would you construct in order to speed this query up? What's the I/O cost?

Composite index on X and Y in T1.

Index on Z in T2.

Cost: $O(\log_B(|T1|) + |X_i \cap Y_j| \cdot \log_B(|T2|))$

(assuming B-trees internal nodes in storage)

Question 2 - Index Design

Consider the following tables and query.

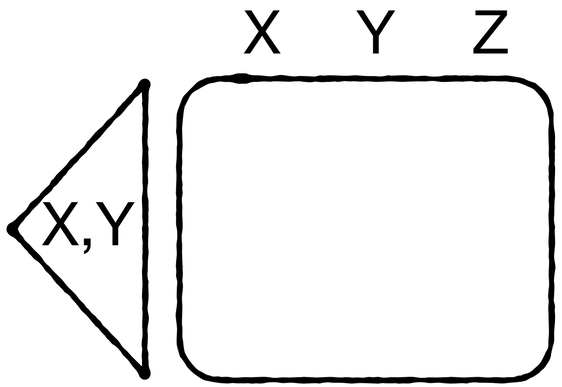


Table T1

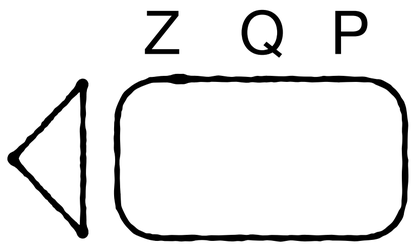


Table T2

Select X, Y, Q from T1, T2 where X="i" and Y="j" and T1.Z = T2.Z

(2) Assuming tiny memory, which indexes would you construct in order to speed this query up? What's the I/O cost?

Composite index on X and Y in T1.

Index on Z in T2.

What if we also the following queries?

Select * from T1 where X="i"

Select * from T1 where Y="j"

Question 2 - Index Design

Consider the following tables and query.

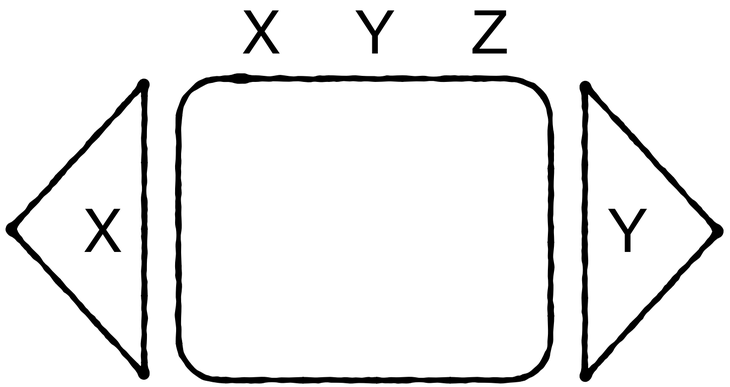


Table T1

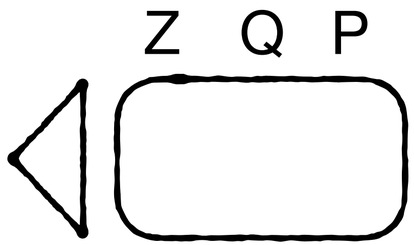


Table T2

Select X, Y, Q from T1, T2 where X="i" and Y="j" and T1.Z = T2.Z

(2) Assuming tiny memory, which indexes would you construct in order to speed this query up? What's the I/O cost?

What if we also the following queries?

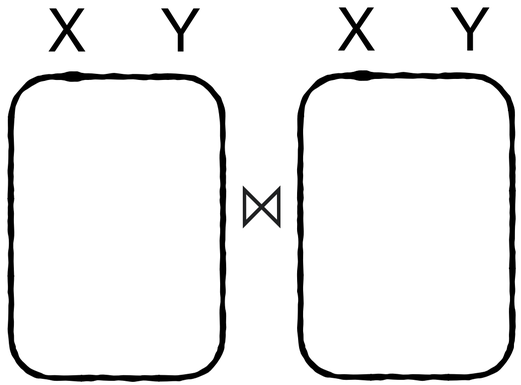
Select * from T1 where X="i"

Select * from T1 where Y="j"

Use separate indexes on X and Y. Composite index won't help one of these queries.

Question 3 - Joins

We would like to join two relations with 2^{30} and 2^{32} rows respectively. Each row has the same size of 64B. Both relations are much larger than the available memory, and the output does not need to be sorted.

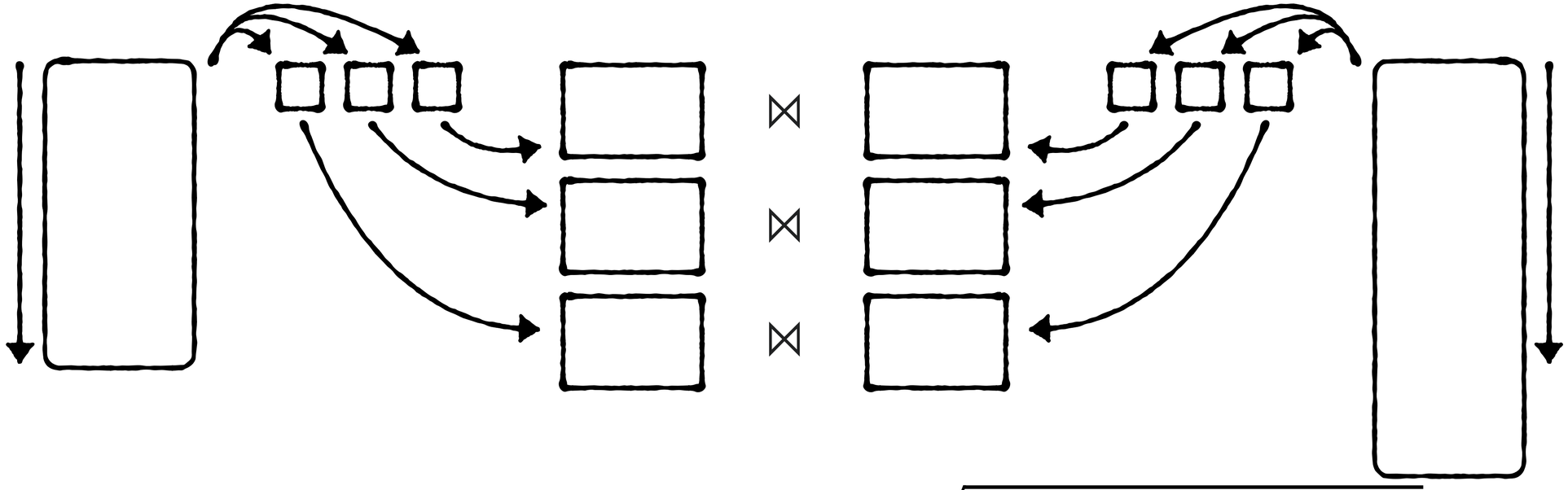


(1) Which join algorithm is best for this case?
(2) What's the minimum amount of memory needed to join the relations in 2 passes?
(3) How can we use additional memory beyond the minimum to further speed up the join?

Question 3 - Joins

We would like to join two relations with 2^{30} and 2^{32} rows respectively. Each row has the same size of 64B. Both relations are much larger than the available memory, and the output does not need to be sorted.

(1) Which join algorithm is best for this case? **Grace Hash Join**

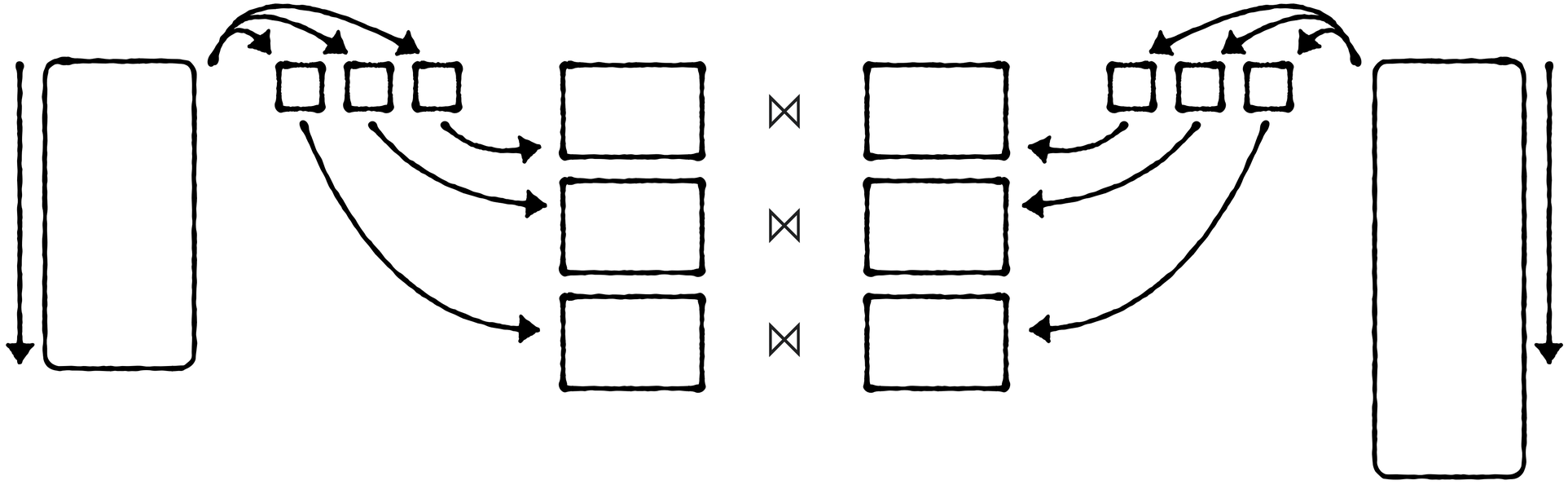


Question 3 - Joins

We would like to join two relations with 2^{30} and 2^{32} rows respectively. Each row has the same size of 64B. Both relations are much larger than the available memory, and the output does not need to be sorted.

(2) What's the minimum amount of memory needed to join the relations in 2 passes?

$$M = \sqrt{\min(|T1|, |T2|) \cdot B} = \sqrt{\min(2^{32}, 2^{30}) \cdot 64} = 2^{18}$$ entries fitting in memory

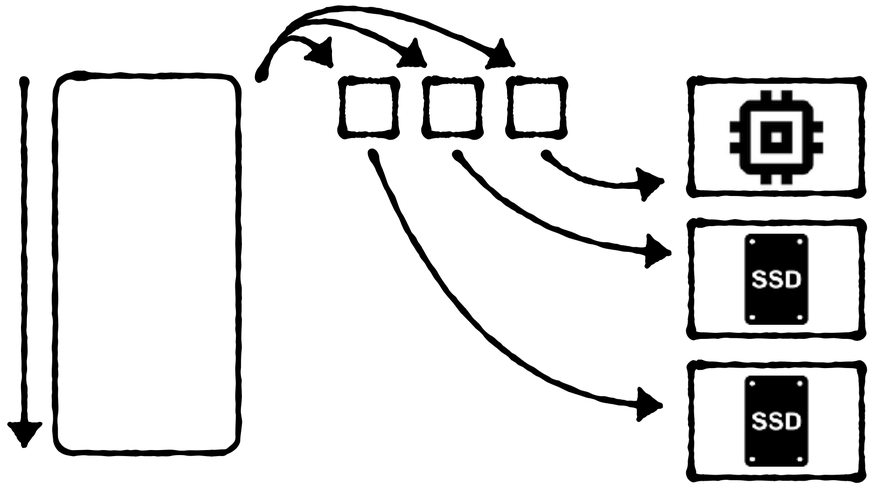


Question 3 - Joins

We would like to join two relations with 2^{30} and 2^{32} rows respectively. Each row has the same size of 64B. Both relations are much larger than the available memory, and the output does not need to be sorted.

(3) How can we use additional memory beyond the minimum to further speed up the join?

During partitioning phase, keep as many partitions in memory as we have space for.

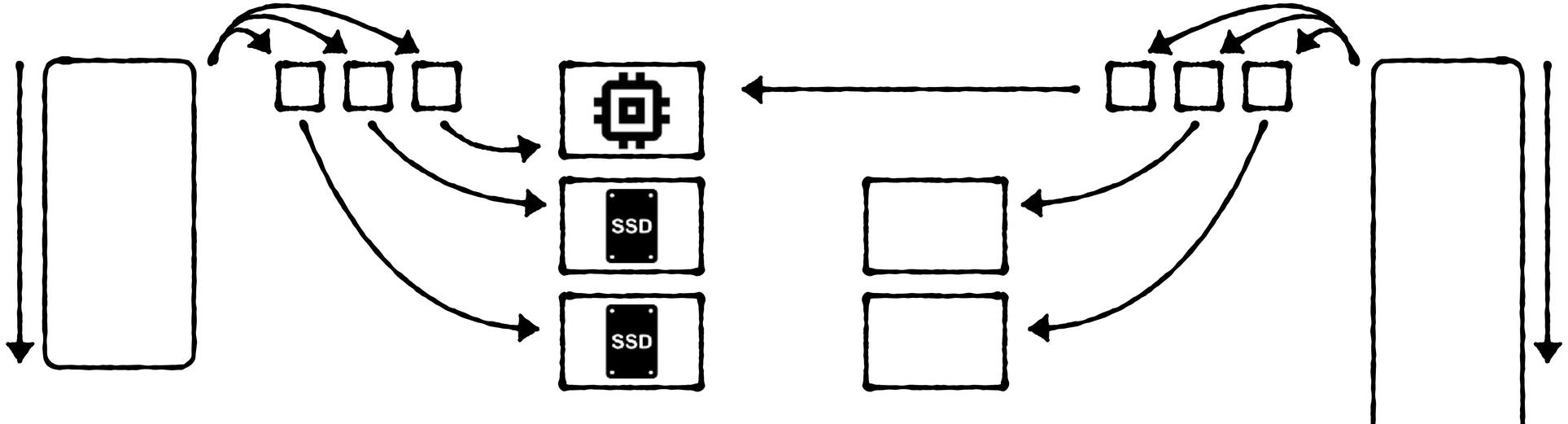


Question 3 - Joins

We would like to join two relations with 2^{30} and 2^{32} rows respectively. Each row has the same size of 64B. Both relations are much larger than the available memory, and the output does not need to be sorted.

(3) How can we use additional memory beyond the minimum to further speed up the join?

During joining phase, join to in-memory partitions immediately. This saves one pass over all partitions fitting in memory.

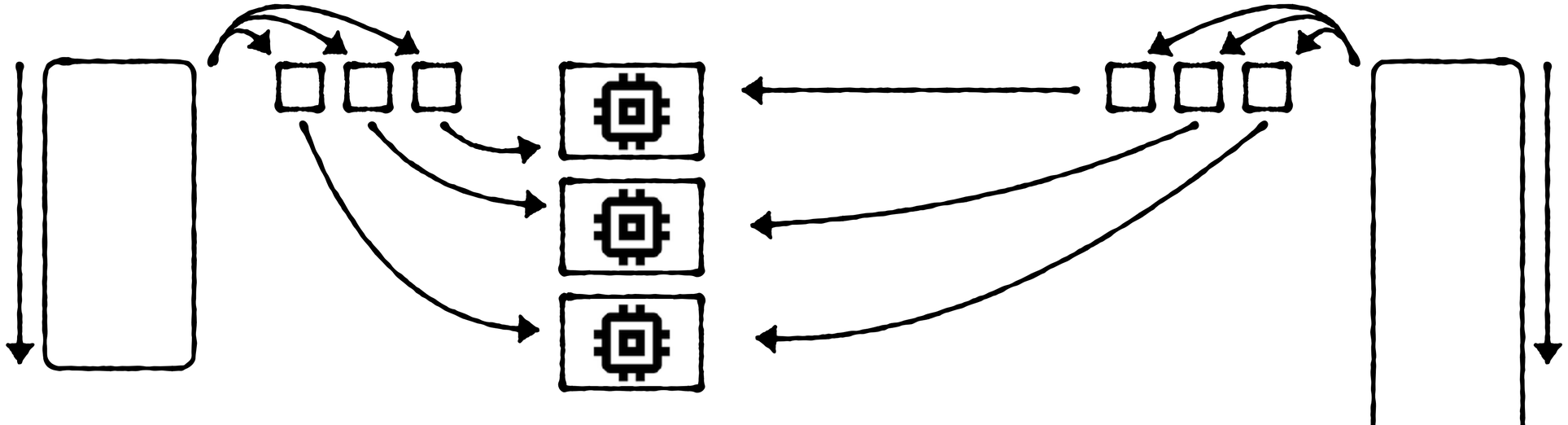


Question 3 - Joins

We would like to join two relations with 2^{30} and 2^{32} rows respectively. Each row has the same size of 64B. Both relations are much larger than the available memory, and the output does not need to be sorted.

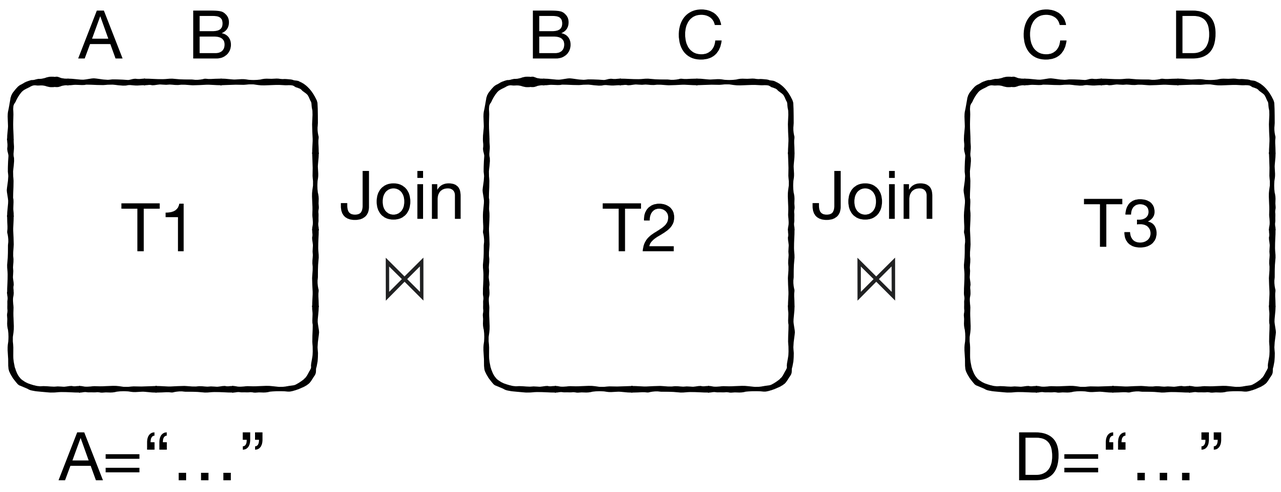
(3) How can we use additional memory beyond the minimum to further speed up the join?

As memory increases, this becomes identical to block nested join. This algorithm is called Hybrid Hash Join because it combines Grace Hash Join with block nested join.



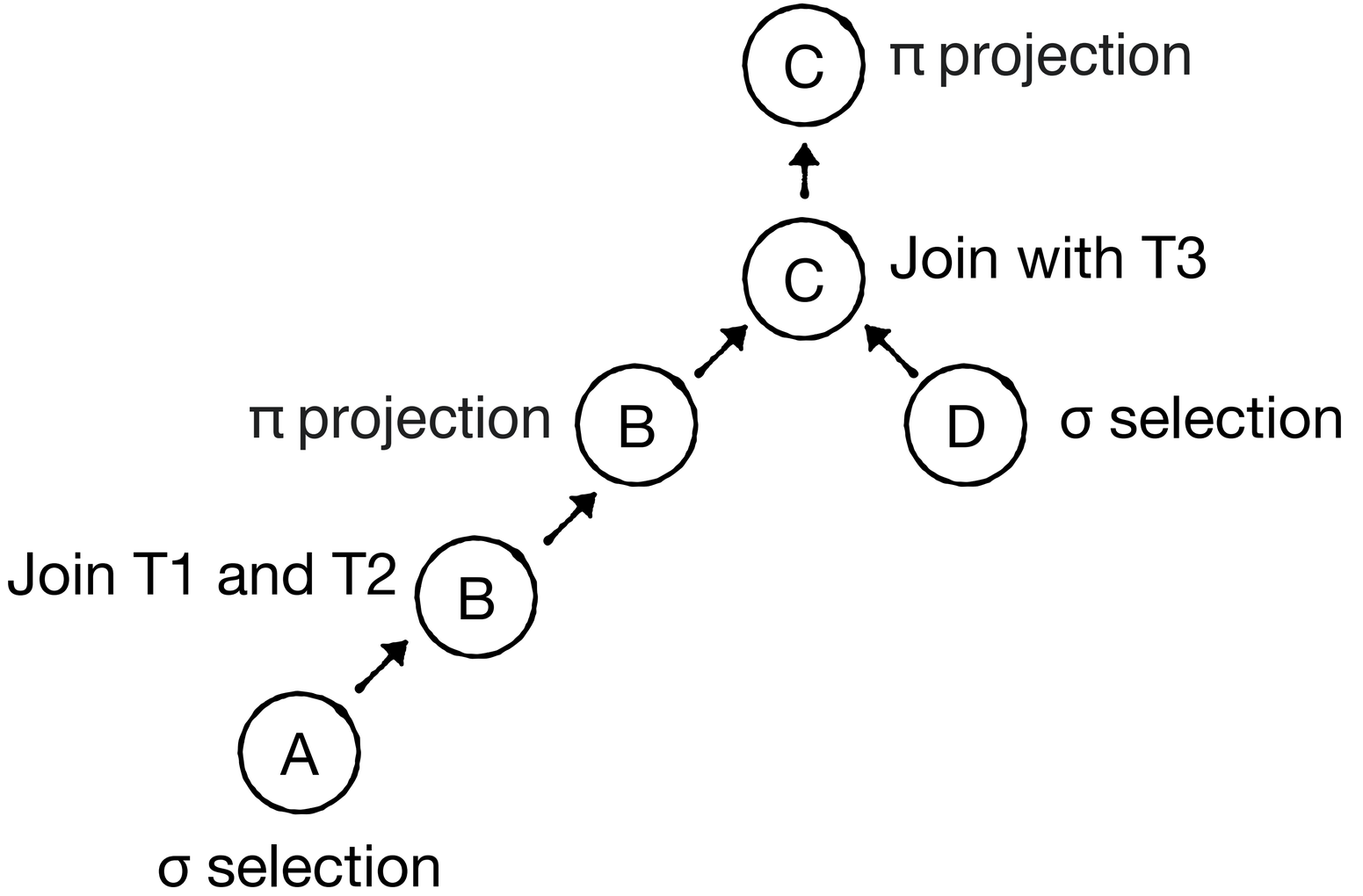
Question 4 - Query Plan

Construct a minimal cost logical query plan for the following query on three tables, all with equal sizes. Assume A is more selective than D. Use the query optimization principles we have seen in class.



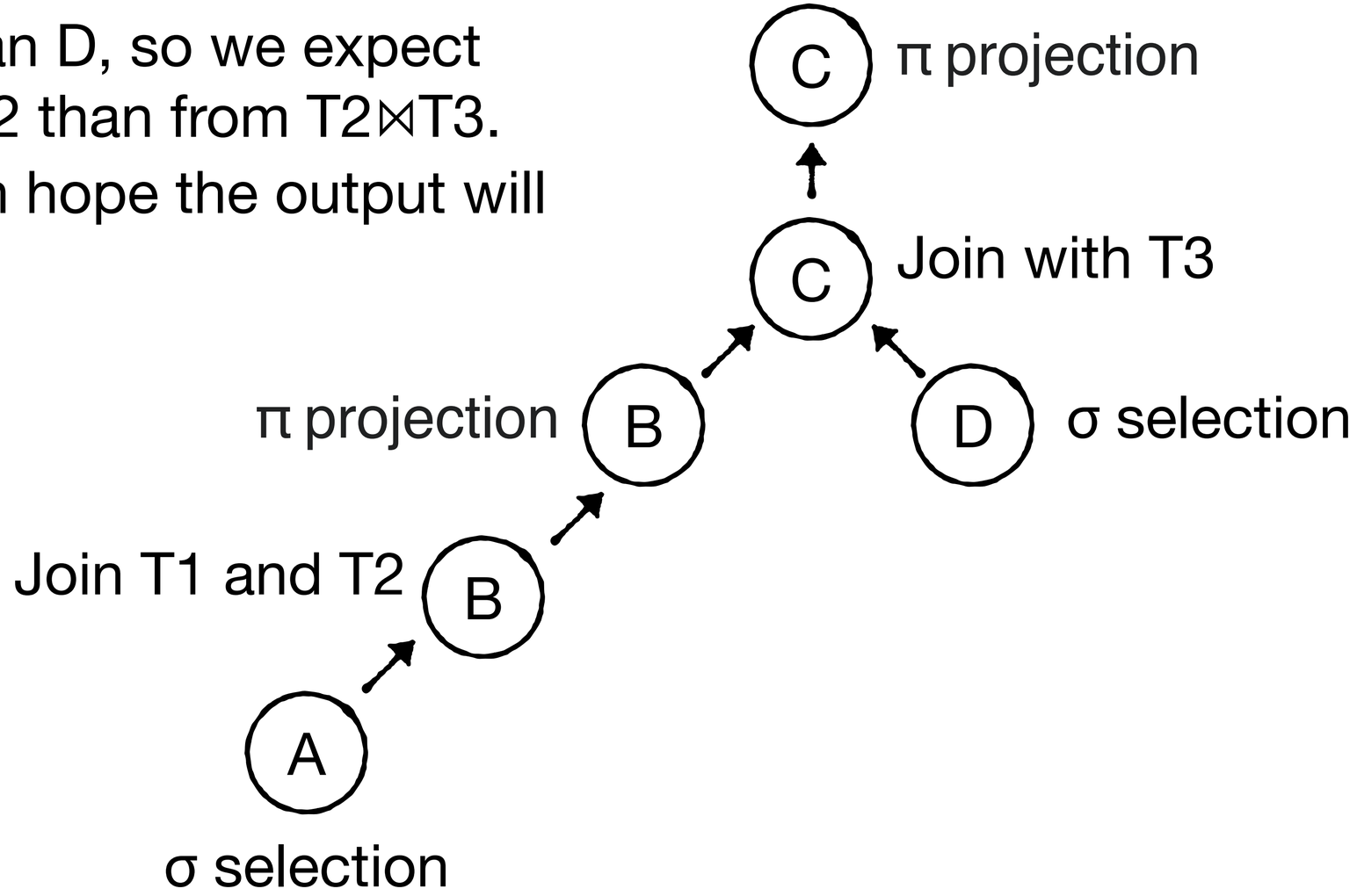
Select A, D from T1, T2, T3 where T1.B = T2.B and T2.C = T3.C
and A="i" and D="j"

Point 1: pushing selections and projections as much as possible



Point 1: pushing selections and projections as much as possible

Point 2: A is more selective than D, so we expect smaller output size from T1⋈T2 than from T2⋈T3. We therefore join T1⋈T2 first in hope the output will fit in memory



Point 1: pushing selections and projections as much as possible

Point 2: A is more selective than D, so we expect smaller output size from T1⋈T2 than from T2⋈T3. We therefore join T1⋈T2 first in hope the output will fit in memory

Point 3: We employ left-deep join structure