IN3020/4020 – Database Systems
Spring 2020, Week 7.1

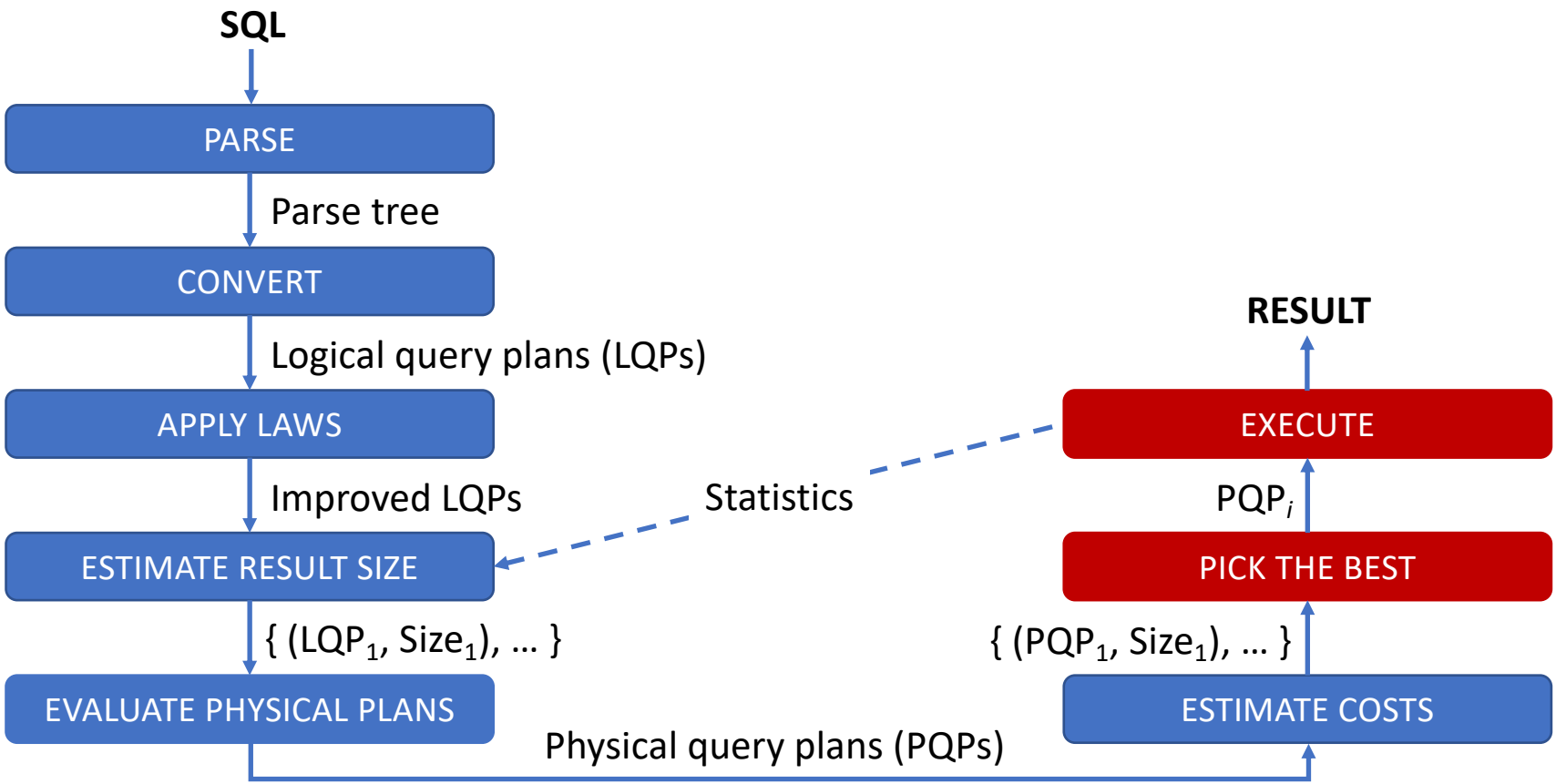
Efficient Query Execution – Part 1

Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

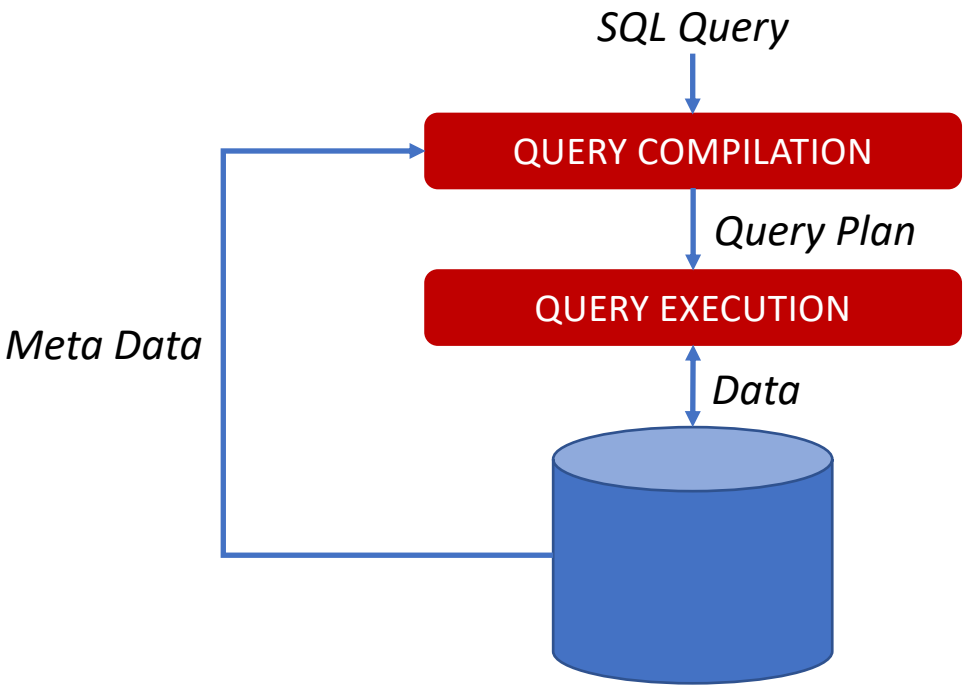
Efficient execution of queries



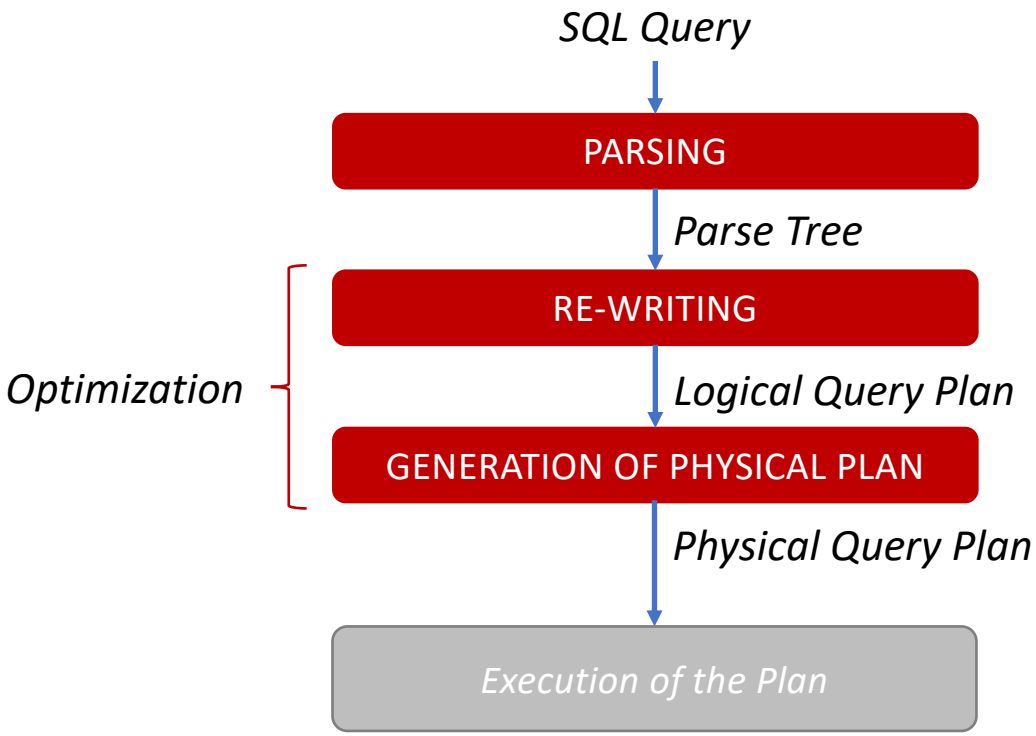
Efficient execution of queries

- Query management
- Model for cost calculation
- Cost of basic operations
- Implementation algorithm

Query management



Query management



Physical operators

- A query can be expressed in terms of a relational algebra expression
  - A physical query plan is implemented by a set of operators (algorithms) similar to the operators from the relational algebra
  - In addition, we need basic operators for reading (scanning) a relation, sorting a relation, etc.
- In order to choose a good plan, we must be able to estimate the cost of each operator
  - We will use the **number of disk IOs** and generally assume that
    - the operands must initially be retrieved from disk
    - the result is used directly from the primary memory
    - other costs can be ignored

Cost parameters

- Which mechanism has the lowest cost depends on several factors, including
  - Number of blocks available in memory, M
  - Whether there are indexes, and what kind
  - Layout on disk and special disk properties
  - ...
- For a relation R, we also need to know
  - Number of blocks required to store all the tuples, B_R
  - Number of tuples in R, T_R
  - Number of distinct values for an attribute A, $V_R(A)$
    The average number of tuples "like" A is T_R / $V_R(A)$

Factors that increase cost

- Use of indexes that are not in memory
- Ideally, the tuples of a relation R are **clustered** (tightly packed) and therefore occupy B_R blocks. But this is not always the case:
  - The blocks may contain free space for inserting new tuples.
  - The tuples are **scattered**, e.g. because they are stored entangled (mixed) with tuples from other relations.
  - ...
- BUT … we generally do **not** include these factors in our estimates!

Cost of basic operations

- The cost of reading a disk block is 1 disk IO
- The cost of writing a disk block is 1 disk IO
- Update costs 2 disk IOs

- Three fundamental operations:
- Unsorted reading of a relation R: The cost depends on storage
  - Clustered relation - B_R disk IO
  - Scattered relation - worst-case T_R disk IO
- Sorting
- Hash partitioning

- We will generally assume tightly packed relations!

Cost of basic operations

- Remember that Postgres distinguish between
  - Sequential read
  - Random read
- For data in memory (joins of joins), it is not the number of blocks but the number of tuples that is relevant
- Since things are tightly packed in the memory, blocks or tuples aren't as relevant

Ingestion and indices

- In Postgres, indices are dense: tables are generally not sorted
- That means index usage is per tuple, and reads are random reads. Reading an entire table is a sequential read.
- Indexes are therefore not used if we are to have many duplicates.

Bitmap heap scan

- In addition to the cost of random reads, the same block can be read multiple times.
- This is not good news if we have multiple indexes.
- Bitmap heap scan fixes this.

- Idea: Read index, find block number for all the tuples we should have.
- Read the correct blocks in their order.

Sorting

- Sort-Scan reads a relation R and returns it in sorted order with respect to a set of attributes X.
- If R has a B-tree index of X, then it can be used to traverse the tuples in R in sorted order
- If the whole R fits into the memory, we use an efficient sorting algorithm. Costs B_R disk-IO
- If R is too big to fit into memory, we use a sorting algorithm that manages data in multiple passes:
- Frequently used: Two-Phase Multiway Merge Sort (TPMMS)

Two-Phase Multiway Merge Sort (TPMMS)

- Phase 1: Sort parts of the relation (as much as there is room for in memory each time)
  - Fill all available memory blocks with blocks containing the relation
  - Sort this section into memory
  - Write the sorted result (a sublist) back to disk
  - Repeat until all blocks are read and all records are sorted into sublists.
  - Costs $2B_R$, i.e., all blocks are both read and written.

Two-Phase Multiway Merge Sort (TPMMS)

- Phase 2: Merge all the sorted sublists into one sorted list by repeating the following:
  - Read the first block from each sublist into memory and compare the first element of each block.
  - Place the smallest item in the final list and remove the item from the sublist.
  - Retrieve new blocks from each sublist as needed.
  - Cost: B_R
- Total cost (with phase 1): $3B_R$
- Prerequisite: The number of sublists must be less than or equal to the number of available memory blocks: $B_R/M \leq M$, i.e., $M \geq \sqrt{B_R}$

Hash-partitioning

- The hash function collects tuples that should be viewed in the same context. With M available buffers: Use M-1 buffers for buckets, 1 to read disk blocks
- Algorithm:

```
for each block b in relation R {
        read b into buffer M;
        for each tuple t in b {
                if not space in bucket h (t) {
                        copy bucket h(t) to disk;
                        initialize new block for bucket h(t)};
                copy t to bucket h(t)};
        }
for every non-bucket {write bucket to disk}
```

- Costs $2B_R$: Reads all data and writes it back partitioned. (Note: Includes writing back!)

Executing the Query

- Three main classes of algorithms:
  - sort based
  - hash based
  - index-based
- In addition, cost and complexity is divided into different levels
  - One-pass algorithms - data fits into memory, read only once from disk.
  - Two-pass algorithms - data too large for memory, read data, processes, write back, read again
  - n-pass algorithms - recursive generalizations of two-pass algorithms for methods that need multiple passes across the entire dataset

Various groups of operators

- Tuple-at-a-time, unary operations:
  - selection (σ)
  - projection (π)
- Full-relation, unary operations:
  - grouping (γ)
  - duplicate elimination (δ)
- Binary operations:
  - set and bag union ($\cup$)
  - set and bag sections ($\cap$)
  - set and bag difference ($-$)
  - join ($\bowtie$)
  - product ($\times$)

We will look at several ways to implement some of these operators using different algorithms and different number of passes.

Note that we will look at only some selected operators and summaries!

For more algorithms, see the textbook!

Selection and Projection

- Both selection (σ) and projection (π) have well-known algorithms, regardless of whether the relation fits in the memory or not:

  Retrieve one and one block into memory and process each tuple in the block!

- The only memory requirement is M $\geq$ 1 for the input buffer.
- The cost depends on where R is:
  - If in memory, cost = 0
  - If on disk, cost = at worst B_R disk-IO

Selection - Example

- T_R = 20,000, B_R = 1000, $\sigma_{A=v}(R)$
- No index on A - retrieve all blocks
  - → 1000 disk IO
- Cluster index on A - retrieve B_R / $V_R(A)$ blocks
  - VR (A) = 100 → 1000 / 100 = 10 disk IO
  - VR (A) = 10 → 1000 / 10 = 100 disk IO
  - VR (A) = 20,000, i.e., A is a candidate key → 1 disk IO

Summary: Cost and Memory Requirements for Selection $\sigma_{A=v}(R)$

Algorithm	Memory Requirement	Disk IO
No index on A	$M \geq 1$	B_R
Cluster index on A	$M \geq 1$	$B_R / V_R(A)$

Summary: Cost and Memory Requirements for Projection $\pi_L(R)$

Memory Requirement	Disk IO
$M \geq 1$	B_R

Need any more explanation?

Duplicate elimination (δ) – One pass

- Duplicate elimination (δ) can be performed by reading a block at a time, and for each tuple, …
  - copying to output buffer if first occurrence,
  - ignoring if we have seen a duplicate.
- Requires a copy of each tuple in the memory for comparison
- Total cost: B_R
- Memory requirements: 1 block to R, $B_{\delta(R)}$ blocks to keep a copy of each tuple where δ(R) is the number of different tuples in R, so $M \geq 1 + B_{\delta(R)}$

Duplicate elimination (δ) – Two pass

- Sort-based algorithm:
  Equivalent to Two-Phase Multiway Merge Black (TPMMS)
  - Read M blocks into memory at a time,
  - Sort these M blocks and write the sublist to disk,
  - Instead of merging the sublists, copy the first tuple and eliminate duplicates in front of the sublists.
- Hash-based algorithm:
  - Partition the relation,
  - Duplicate tuples will have the same hash value,
  - Read each bucket into memory and execute the one-pass algorithm that removes duplicates.

Summary: Cost and Memory Requirements for Duplicate Elimination δ(R)

Algorithm	Memory Requirement	Disk IO
One-pass	$M \geq 1 + B_{\delta(R)}$	B_R
Two-pass sorting	$M \geq \sqrt{B_R}$	$3B_R$
Two-pass hashing	$M \geq \sqrt{B_R}$	$3B_R$

Summary: Cost and Memory Requirements for Grouping γ(R)

Algorithm	Memory Requirement	Disk IO
One-pass	$M \geq 1 + B_{\gamma(R)}$	B_R
Two-pass sorting	$M \geq \sqrt{B_R}$	$3B_R$
Two-pass hashing	$M \geq \sqrt{B_R}$	$3B_R$

Binary Operations

- A binary operation takes two relations as arguments:
  - union: R ∪ S
  - intersection: R ∩ S
  - difference: R – S
  - join R ⋈ S
  - product: R × S

Note that there is a difference between the set and bag versions of these operators.

- We are going to look at algorithms for set and bag union, as well as natural join; remaining operators can be implemented in a similar manner
- All operations that require comparison must use a search structure (such as binary trees or hashing), which also requires resources, but we will not be including these in our estimates.

Union (∪) – One pass

- Bag-union can be calculated with a simple one-pass algorithm:
  - Read and copy each tuple in R to the output buffer,
  - Read and copy each tuple in S to the output buffer.
- Total cost: $B_R + B_S$ disk IO
- Memory requirements: 1 (read each block directly to output buffer)
- Set-union must remove duplicates
  - Read the smallest relation (assume S) into M-1 buffers and copy each tuple to output, but keep S in memory.
  - Read the blocks in R into the last buffer, and check for each tuple whether it exists in S; if not, copy it to output
- Memory requirements: $B_S < M$

What about the cost of set-union?

Set Union (∪) – Two pass, sorting

- Perform phase 1 of TPMMS for each of R and S (create sorted sublists),
- Use one buffer for each of the sublists of R and S,
- Repeat: Find the smallest remaining tuple in each sub-list
  - Output tuple,
  - Remove duplicates from the front of all lists
- Total cost: $3B_R + 3B_S$ disk IO
- Memory requirements:
  - M buffers provide space for creating sublists of M blocks
  - $(B_R + B_S) / M \leq M$ gives $\sqrt{(BR + BS)} \leq M$
  - If $B_R + B_S > M^2$, we will get more than M sublists, and the algorithm will not work.

Union (∪) – Two pass, hashing

- Set-union two-pass hash algorithm:
  - Partition both R and S in M-1 buckets with the same hash function.
  - Make union on each bucket pair $R_i \cup S_i$ separately (one-pass set union)
- Total cost: $3B_R + 3B_S$ disk IO
  - 2 for partitioning
  - 1 for union of bucket pairs
- Memory requirements:
  - M buffers allow room for creating M-1 buckets for each relation
  - For each bucket pair R_i and S_i: either $B_{Ri} \leq M-1$ or $B_{Si} \leq M-1$
  - Approximately $\min(B_R, B_S) \leq M^2$, i.e., $\sqrt{\min(BR, BS)} \leq M$
  - Note: If the smallest of R_i and S_i cannot fit in M-1 buffers for one i, the algorithm will not work

Summary: Cost and Memory Requirements for R ∪ S (given $B_S \leq B_R$)

BAG version

Algorithm	Memory Requirement	Disk IO
One-pass	$M \geq 1$	$(B_R + B_S)$

SET version

Algorithm	Memory Requirement	Disk IO
One-pass	$M \geq 1 + B_S$	$B_R + B_S$
Two-pass sorting	$M \geq \sqrt{(BR + BS)}$	$3B_R + 3B_S$
Two-pass hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

Summary: Cost and Memory Requirements for $R \cap S$ (given $B_S \leq B_R$)

SET version

Algorithm	Memory Requirement [1]	Disk IO
One-pass	$M \geq 1 + B_S$	$B_R + B_S$
Two-pass sorting	$M \geq \sqrt{(BR + BS)}$	$3B_R + 3B_S$
Two-pass hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

[1] **BAG version** will in addition need space for tuple-counters!

Summary: Cost and Memory Requirements for $R - S$ (given $B_S \leq B_R$)

SET version

Algorithm	Memory Requirement [1]	Disk IO
One-pass	$M \geq 1 + B_S$	$B_R + B_S$
Two-pass sorting	$M \geq \sqrt{(BR + BS)}$	$3B_R + 3B_S$
Two-pass hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

[1] **BAG version** will in addition need space for tuple-counters!

Natural Join (⋈) – One pass

- Natural join (⋈) concatenates tuples from R (X, Y) and S (Y, Z) such that R.Y = S.Y
- One-pass algorithm:
  - Read the smallest relation (assume S) into M-1 buffers.
  - Read relation R block by block. For each tuple t:
    - Join t with matching tuples in S
    - Move the result to the output
- Total cost: $B_R + B_S$ disk IO
- Memory requirements: $B_S < M$

Natural join (⋈): Nested loop join, block-based algorithm

- Keep as much as possible of the smallest relation in M-1 blocks
  - Algorithm: (assume S smallest)

```
for each partition p of S of size M-1 {
    read p into memory;
    for each block b for R {
        read b into memory;
        for each tuple t in b {
            find tuples in p that match t;
            join each of these with t to output}}}
```

- Total cost: $B_S + B_R \lceil B_S / (M - 1) \rceil$ disk IO.
  (Read S once, read R once for each S partition. There will be $\lceil B_S / (M - 1) \rceil$ partitions, round of BS / (M - 1) upwards.)
- Memory requirements: 2 (one R block and one S block; the size of the S partitions is adapted to M, so in the worst case, each S partition is one block)

Natural join (⋈): Two-pass sorting, simple algorithm

- Sort R and S separately using TPMMS on the join attributes.
- Join the sorted R and S by
  - If a buffer for R or S is empty, retrieve new block from disk
  - Find tuples that have the lowest v-value for the join attributes
    (Note: Can also be found in subsequent blocks; in that case all must be kept in memory!)
  - If there are v-values in both R and S, join these and write to the output
  - Otherwise, drop all v-values.
- Total cost: $5B_R + 5B_S$ disk IO
  - 4 for TPMMS (must write back to disk before final join, then 4 rather than 3)
  - 1 to join the sorted R and S
- Memory requirements:
  - TPMMS requires $B \leq M^2$ for each of the relations, i.e., $B_R \leq M^2$ and $B_S \leq M^2$
  - If there is a v-value where associated tuples cannot fit in M blocks, the algorithm does not work

Natural join (⋈): Two-pass hashing, «hash-join» algorithm

- Algorithm:
  - Partitions both R and S in M-1 buckets with the same hash function on the join attributes Y. (Needs 1 block to keep consecutive blocks of R and S.)
  - Make natural join on each bucket pair separately - $R_i \bowtie S_i$ - one-pass join.
- Total cost: $3B_R + 3B_S$ disk IO
  - 2 to partition relations
  - 1 to join
- Memory requirements:
  - M buffers can create M-1 buckets for each relation
  - For each pair of R_i and S_i, either $B_{Ri} \leq M-1$ or $B_{Si} \leq M-1$
  - About $\min(B_R, B_S) \leq M^2$, i.e., $\sqrt{\min(BR, BS)} \leq M$
  - If the smallest of Ri and Si cannot fit in M-1 buffers, the algorithm does not work

Natural join (⋈): Index based

- Algorithm R (X, Y) ⋈ S (Y, Z):
  - Assume there is an index for S on the attributes Y
  - Read each block for R. For each tuple, do the following:
    - Find duplicates in S with the same join attribute values using the index
    - Read the corresponding blocks, join the relevant tuples and output the result
- Total cost:
  - B_R disk IO to read all R-duplicates.
  - In addition, for each tuple in R we must read the corresponding S-tuples:
    - With cluster index on Y (so S is sorted on Y): B_S/V_S (Y) blocks with S-tuples for each R tuple, total $T_R \lceil B_S / V_S (Y) \rceil$. Must round B_S / V_S (Y) upwards if V_S (Y) is larger than B_S, e.g., when Y is candidate key - then $\lceil B_S / V_S (Y) \rceil = 1$.
    - If S is not sorted on Y: $T_S/ V_S(Y)$, total $T_R\ T_S / V_S (Y)$
  - The cost is completely dominated by T_R, so we ignore the costby reading R
- Memory requirements: 2 (one R block and one S block)

Summary: Cost and Memory Requirements for R ⋈ S

Algorithm	Memory Requirement	Disk IO
One-pass ($B_S \leq B_R$)	$M \geq 1 + B_S$	$B_R + B_S$
Block-based nested loop	$M \geq 2$	$B_S + B_R \lceil B_S / (M-1) \rceil$
Simple two-pass sorting	$M \geq \sqrt{B_S}$	$5B_R + 5B_S$
Sort-join	$M \geq \sqrt{(BR + BS)}$	$3B_R + 3B_S$
Hash-join	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$
Hybrid hash-join ($B_S \leq B_R$)	$M \geq \sqrt{B_S}$	$(3 - 2M/B_S)(B_R + B_S)$
Index-join, cluster index on S.Y	$M \geq 2$	$T_R \lceil B_S / V_S(Y) \rceil$
Index-join, cluster index on S.Y S not sorted on Y	$M \geq 2$	$T_R T_S / V_S(Y)$
Zig-zag index-join on Y	$M \geq B_{T_R}/VR(Y) + B_{T_s}/VS(Y)$	$B_R + B_S$

Natural join (⋈) example

- $T_R = 10.000$, $T_S = 5.000$
- $V_R(A) = 100$, $V_S(A) = 10$
- 4 KB blocks
- Records in R and S are 512 bytes (0,5 KB) i.e., 8 records per block of 4KB
- Has cluster index on attributte A for both R and S

$\Rightarrow$ $B_S = 5.000 / 8 = 625$
$B_R = 10.000 / 8 = 1250$

Natural join R ⋈ S example (cont.)

T_R=10.000, T_S =5.000, B_R =1250, B_S =625, M=101

Algorithm	Memory Requirement	Disk IO
One-pass	626	1875
Block-based nested loop	2	9375
Simple two-pass sorting	36	9375
Sort-join	44	5625
Hash-join	25	5625
Hybrid hash-join	25	5019
Index-join on A	2	625 000 (cluster index on S) 62 500 (cluster index on R)
Zig-zag index-join on A	76	1875

Choosing and algorithm

- One-pass algorithms are good if one of the arguments (relations) fits into the memory.
- Two-pass algorithms must be used if we have large relations.
  - Hash-based algorithms
    - Requires less memory compared to sorting approaches. Only dependent on the smallest relation. Used often.
    - Assumes approximately same size buckets regardless of hash value (good hash function) - in reality there will be some variations, must assume smaller bucket sizes.
  - Sort-based algorithms
    - Yield sorted results, which can be utilized by subsequent operators.
  - Index based algorithms
    - Excellent pre-selection
    - Excellent for join if both arguments have cluster indices.

N-pass algorithms

- For really large relations, two pass algorithms are usually not sufficient
- Example: B_R = 1,000,000
  - TPMMS requires $M \geq \sqrt{B_R} \rightarrow M \geq 1000$ blocks
  - If 1000 blocks are not available, TPMMS will not work.
  - ⇒ Need several passes through the data set.
- Sort-based algorithms:
  - If R fits into memory, sort.
  - If not, partition R into M groups and sort each Ri recursively.
  - Merge the sublists.
  - Total Cost: $(2k - 1)\ B_R$, where k is the number of passes required
  - We need $\sqrt[k]{B_R}$ buffers, i.e., $B_R \leq M^k$
- The same can be done for hash based algorithms.

Next…

EXPLAIN or EXPLAIN PLAN examples:

Read in advance and try to interpret!
Try several different DBMS!