IN3020/4020 – Database Systems
Spring 2021, Weeks 6.1-6.2

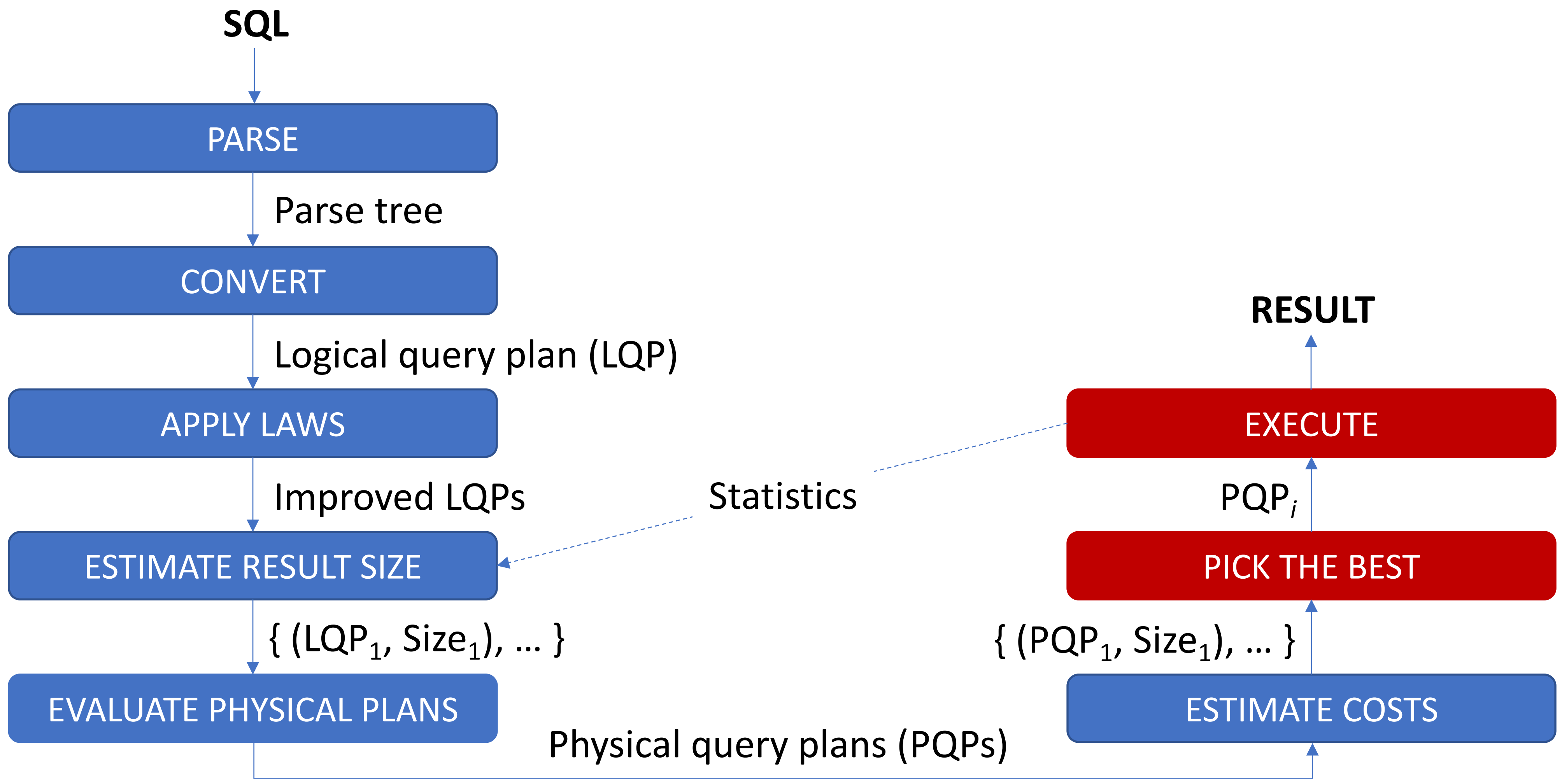
Efficient Query Execution – Parts 1-2

Egor V. Kostylev

Based upon slides by E. Thorstensen and M. Naci Akkøk

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Efficient execution of queries



Materials to Read

- Part 8, Chapter 18 of the Book (Elmasri & Navathe, «Fundementals of Database Systems»)
- NOTE: I do not follow any of them line by line

Efficient execution of queries

- Query management system key components:
  - optimisation
  - model for cost calculation
  - implementation algorithms

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 written to the main memory (quite simplification!)
    - other costs can be ignored

Cost parameters

- Which mechanism has the lowest cost depends on several factors, including
  - Number of blocks available in main memory, MBlocks
  - Whether there are indices, 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, Blocks_R
  - Number of tuples in R, Tup_R
  - Number of distinct values for an attribute A, $\text{Val}_R(A)$
    The average number of tuples with each value of A is Tup_R / $\text{Val}_R(A)$

Factors that also may <u>increase</u> cost

- Use of indexes that are not in main memory
- Ideally, the tuples of a relation R are **clustered** (tightly packed) and therefore occupy $Blocks_R$ 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 **<u>not</u>** 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 - Blocks_R disk IO
  - (Scattered relation - worst-case Tup_R disk IO, assuming a tuple is smaller that a block)
- Sorting
- Hash partitioning

- We will generally assume tightly packed relations!

Cost of basic operations: Relation read

- No indices: need to read all blocks in relation -- Blocks_R
- For data in memory (e.g., 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 anyway, whether we measure by blocks or tuples isn't as relevant, really
- Postgres distinguishes between
  - Sequential read
  - Random read

Ingestion and indices

- With indices: can reduce costs a lot, but depends what we need to retrieve
- In Postgres, indices are dense and 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 may not be 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 index values
- 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

Cost of basic operations: 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 Blocks_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 in the memory
  - Write the sorted result (a sublist) back to disk
  - Repeat until all blocks are read and all records are sorted into sublists
  - Costs $2Blocks_R$, 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 elements of these blocks
  - Place the smallest item to the final output list and remove the item from the sublist
  - Retrieve new blocks from each sublist as needed
  - Cost: $Blocks_R$
- Total cost (with phase 1): $3Blocks_R$
- Prerequisite: The number of sublists must be less than or equal to the number of available main memory blocks: $Blocks_R/MBlocks \leq MBlocks$, i.e., $MBlocks \geq \sqrt{Blocks_R}$

Cost of basic operations: Hash-partitioning

- The hash function partition tuples into buckets that should be processed together. With MBlocks available blocks: Use MBlocks-1 blocks for bucket buffers, 1 to read disk blocks
- Algorithm:

```
for each block b in relation R {
        read b into the read block;
        for each tuple t in b {
                if no space in buffer for bucket hash(t) {
                        output (e.g., write to disk) buffer hash(t);
                        initialize new block for bucket h(t)};
                copy t to buffer for bucket hash(t)}}
for every buffer bucket {write buffer bucket to disk}
```

- Costs Blocks_R: Reads all data
- Prerequisite: MBlocks > number of buckets (i.e., hash values)

Executing the Query – The algorithms

- 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

Groups of operators & their algorithms

- 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 (∪)
  - set and bag sections (∩)
  - set and bag difference ($-$)
  - join (⋈)
  - product (×)

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

Selection - Example

- $Tup_R = 20{,}000$, $Blocks_R = 1000$, $\sigma_{A=v}(R)$

- No index on A - retrieve all blocks
  - $\rightarrow$ 1000 disk IO
- Cluster index on A - retrieve $Blocks_R$ / $Val_R(A)$ blocks
  - $Val_R(A) = 100 \rightarrow 1000 / 100 = 10$ disk IO
  - $Val_R(A) = 10 \rightarrow 1000 / 10 = 100$ disk IO
  - $Val_R(A) = 20{,}000$, i.e., A is a candidate key $\rightarrow$ 1 disk IO

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

Algorithm	Memory Requirement	Disk IO
No index on A	MBlocks ≥ 1	$Blocks_R$
Cluster index on A	MBlocks ≥ 1	$Blocks_R$ / $Val_R(A)$

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

Memory Requirement	Disk IO
MBlocks $\geq$ 1	$Blocks_R$

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: $Blocks_R$
- Memory requirements: 1 block to R, $Blocks_{\delta(R)}$ blocks to keep a copy of each tuple where $\delta(R)$ is the number of different tuples in R, so $MBlocks \geq 1 + Blocks_{\delta(R)}$

Duplicate elimination (δ) – Two pass

- Sort-based algorithm:
  Similar to Two-Phase Multiway Merge Sort (TPMMS)
  - Read MBlocks blocks into memory at a time
  - Sort these MBlocks 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	$MBlocks \geq 1 + Blocks_{\delta(R)}$	$Blocks_R$
Two-pass sorting	$MBlocks \geq \sqrt{Blocks_R}$	$3Blocks_R$
Two-pass hashing	$MBlocks \geq \sqrt{Blocks_R}$	$3Blocks_R$

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

Algorithm	Memory Requirement	Disk IO
One-pass	MBlocks $\geq$ 1+Blocks$_{\gamma(R)}$	Blocks$_R$
Two-pass sorting	MBlocks $\geq \sqrt{\text{Blocks}_R}$	3Blocks$_R$
Two-pass hashing	MBlocks $\geq \sqrt{\text{Blocks}_R}$	3Blocks$_R$

Binary Operations

- A binary operation takes two relations as arguments:
  - union: $R \cup S$
  - intersection: $R \cap S$
  - difference: $R - S$
  - join: $R \bowtie S$
  - product: $R \times S$

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

- All operations that require comparison must use a search structure (such as binary trees or hashing), which also requires resources, but we will <u>not</u> 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
  - Read and copy each tuple in S to the output
- Total cost: $Blocks_R + Blocks_S$ disk IO
- Memory requirements: MBlocks >= 1 (read each block directly to output buffer)
- Set-union must remove duplicates
  - Read the smallest relation (assume S) into MBlocks-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: $Blocks_S$ < MBlocks

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 sublist
  - Output tuple
  - Remove duplicates from the front of all lists
- Total cost: $3Blocks_R + 3Blocks_S$ disk IO
- Memory requirements:
  - MBlocks buffers provide space for creating sublists of MBlocks blocks
  - $(Blocks_R + Blocks_S) / MBlocks \leq MBlocks$ gives $\sqrt{Blocks_R + Blocks_S} \leq MBlocks$
  - If $Blocks_R + Blocks_S > MBlocks^2$, we will get more than MBlocks sublists, and the algorithm will not work

Union (∪) – Two pass, hashing

- Set-union two-pass hash algorithm:
  - Partition both R and S in MBlocks-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: $3Blocks_R + 3Blocks_S$ disk IO
  - 2 for partitioning
  - 1 for union of bucket pairs
- Memory requirements:
  - MBlocks blocks allow room for creating MBlocks-1 buckets for each relation
  - For each bucket pair R_i and S_i: $Blocks_{Ri} \leq MBlocks-1$ or $Blocks_{Si} \leq MBlocks-1$
  - About $\min(Blocks_R, Blocks_S) \leq MBlocks^2$, i.e., $\sqrt{\min(Blocks_R, BlocksS)} \leq MBlocks$
  - Note: If the smallest of R_i and S_i cannot fit in MBlocks-1 blocks for some i, the algorithm will not work

Summary: Cost and Memory Requirements for R ∪ S (assuming $\text{Blocks}_S \leq \text{Blocks}_R$)

BAG version

Algorithm	Memory Requirement	Disk IO
One-pass	MBlocks ≥ 1	$\text{Blocks}_R + \text{Blocks}_S$

SET version

Algorithm	Memory Requirement	Disk IO
One-pass	MBlocks ≥ 1 + Blocks_S	$\text{Blocks}_R + \text{Blocks}_S$
Two-pass sorting	MBlocks ≥ $\sqrt{\text{Blocks}_R + \text{Blocks}_S}$	$3\text{Blocks}_R + 3\text{Blocks}_S$
Two-pass hashing	~ MBlocks ≥ $\sqrt{\text{Blocks}_S}$	$3\text{Blocks}_R + 3\text{Blocks}_S$

Summary: Cost and Memory Requirements for R ∩ S (assuming $\text{Blocks}_S \leq \text{Blocks}_R$)

SET version

Algorithm	Memory Requirement	Disk IO
One-pass	$\text{MBlocks} \geq 1 + \text{Blocks}_S$	$\text{Blocks}_R + \text{Blocks}_S$
Two-pass sorting	$\text{MBlocks} \geq \sqrt{\text{Blocks}_R + \text{Blocks}_S}$	$3\text{Blocks}_R + 3\text{Blocks}_S$
Two-pass hashing	$\sim \text{MBlocks} \geq \sqrt{\text{Blocks}_S}$	$3\text{Blocks}_R + 3\text{Blocks}_S$

BAG version is not as simple as for union

Summary: Cost and Memory Requirements for $R - S$ (assuming $\text{Blocks}_S \leq \text{Blocks}_R$)

SET version

Algorithm	Memory Requirement	Disk IO
One-pass	$\text{MBlocks} \geq 1 + \text{Blocks}_S$	$\text{Blocks}_R + \text{Blocks}_S$
Two-pass sorting	$\text{MBlocks} \geq \sqrt{\text{Blocks}_R + \text{Blocks}_S}$	$3\text{Blocks}_R + 3\text{Blocks}_S$
Two-pass hashing	$\sim \text{MBlocks} \geq \sqrt{\text{Blocks}_S}$	$3\text{Blocks}_R + 3\text{Blocks}_S$

BAG version is not as simple as for union

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 MBlocks-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: $Blocks_R + Blocks_S$ disk IO
- Memory requirements: $Blocks_S < MBlocks$

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

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

```
for each part p in partition of S of size Mblocks-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: $Blocks_S + Blocks_R \lceil Blocks_S / (MBlocks - 1) \rceil$ disk IO
  (Read S once, read R once for each part in S)
- 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

- 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 values for the join attributes
    (Note: Can also be found in subsequent blocks; in that case all must be kept in memory!)
  - If there are values in both R and S, join these and write to the output
  - Otherwise, drop all values
- Total cost: $5\text{Blocks}_R + 5\text{Blocks}_S$ disk IO
  - 4 for TPMMS (must write back to disk before final join, so 4 rather than 3)
  - 1 to join the sorted R and S
- Memory requirements:
  - TPMMS requires $\text{Blocks} \leq \text{MBlocks}^2$ for R and S, i.e., $\text{Blocks}_R \leq \text{MBlocks}^2$ and $\text{Blocks}_S \leq \text{MBlocks}^2$
  - If there is a value where all tuples cannot fit in MBlocks blocks, the algorithm does not work

Natural join ($\bowtie$): Two-pass hashing, hash-join algorithm

- Algorithm:
  - Partitions both R and S in MBlocks-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: $3Blocks_R + 3Blocks_S$ disk IO
  - 2 to partition relations
  - 1 to join
- Memory requirements:
  - MBlocks buffers can create MBlocks-1 buckets for each relation
  - For each pair of R_i and S_i, either $Blocks_{Ri} \leq MBlocks\text{-}1$ or $Blocks_{Si} \leq MBlocks\text{-}1$
  - About $\min(Blocks_R, Blocks_S) \leq MBlocks^2$, i.e., $\sqrt{\min(Blocks_R, Blocks_S)} \leq MBlocks$
  - If the smallest of R_i and S_i cannot fit in MBlocks-1 buffers, the algorithm does not work

Natural join ($\bowtie$): Two-pass hashing, hybrid hash-join algorithm

- Algorithm:
  - Partitions both R and S in MBlocks-1 buckets with the same hash function on the join attributes Y
    - Needs 1 block to keep consecutive blocks of R and S
    - Assuming $Tup_S < Tup_R$ partition S it first
    - Keep one bucket of S (the smaller table) in main memory, and join `on the fly'

  - Make natural join on each remaining bucket pair separately – $R_i \bowtie S_i$ - one-pass join
- Total cost: … disk IO
- Memory requirements: …

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 all tuples 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 (Estimate!):
  - $Blocks_R$ disk IO to read all R-tuples
  - 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): $Blocks_S / Val_S(Y)$ blocks with S-tuples for each R tuple, total $Tup_R * \lceil Blocks_S / Val_S(Y) \rceil$. Must round $Blocks_S / Val_S(Y)$ upwards if $Val_S (Y)$ is larger than $Blocks_S$, e.g., when Y is candidate key - then $\lceil Blocks_S / Val_S (Y) \rceil = 1$
    - If S is not sorted on Y: $Tup_S / Val_S(Y)$, total $Tup_R * Tup_S / Val_S(Y)$
  - The cost is completely dominated by Tup_R
- Memory requirements: 2 (one R block and one S block)

Natural join (⋈): Two Index based

- Algorithm R (X, Y) ⋈ S (Y, Z):
  - Assume there is an index for both R and S on the attributes Y
  - ….
- Total cost:
  - Blocks_R + Blocks_S disk IO (same as one pass!)
- Memory requirements: … (significantly better than MBlocks ≥ 1 + Blocks_S)

Summary: Cost and Memory Requirements for R ⋈ S

Algorithm	Memory Requirement	Disk IO
One-pass ($Blocks_S \leq Blocks_R$)	$MBlocks \geq 1 + Blocks_S$	$Blocks_R + Blocks_S$
Block-based nested loop	$MBlocks \geq 2$	$Blocks_S + Blocks_R \lceil Blocks_S / (MBlocks-1) \rceil$
Simple two-pass sorting	$MBlocks \geq \sqrt{Blocks_S}$	$5Blocks_R + 5Blocks_S$
Sort-join	$MBlocks \geq \sqrt{(Blocks_R + Blocks_S)}$	$3Blocks_R + 3Blocks_S$
Hash-join	$MBlocks \geq \sqrt{Blocks_S}$	$3Blocks_R + 3Blocks_S$
Hybrid hash-join	$MBlocks \geq \sqrt{Blocks_S}$	$(3-2MBlocks/Blocks_S)(Blocks_R + Blocks_S)$
Index-join, cluster index on S.Y	$MBlocks \geq 2$	$Tup_R * \lceil Blocks_S / Val_S(Y) \rceil$
Index-join, cluster index on S.Y S not sorted on Y	$MBlocks \geq 2$	$Tup_R * Tup_S / Val_S(Y)$
Zig-zag index-join on Y	$MBlocks \geq$ ...	$Blocks_R + Blocks_S$

Natural join (⋈) example

- Tup_R = 10,000, Tup_S = 5,000
- 4 KB blocks
- Records in R and S are 512 bytes (i.e., 8 records per block)
- $Blocks_R$ = 10,000 / 8 = 1,250, $Blocks_S$ = 5,000 / 8 = 625
- $Val_R(Y)$ = 100, $Val_S(Y)$ = 10 (for common attribute Y)
- Has cluster index on attribute Y for both R and S

Natural join R ⋈ S example (cont.)

Tup_R=10,000, Tup_S =5,000, $Blocks_R$ =1250, $Blocks_S$ =625, MBlocks=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 Y	2	625 000 (cluster index on S) 62 500 (cluster index on R)
Zig-zag index-join on Y	76	1875

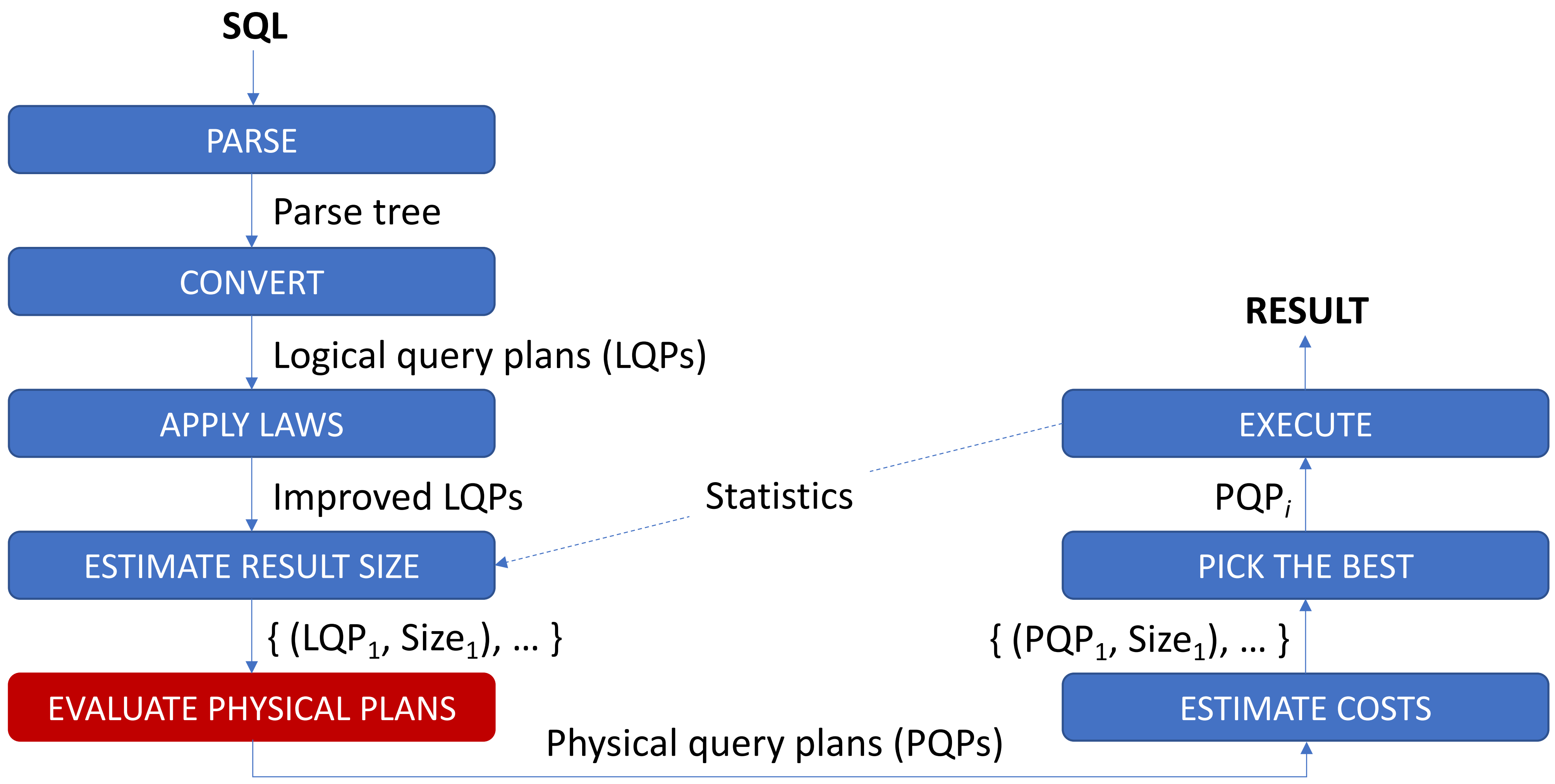
Choosing an algorithm

- 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 (Often used)
    - 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: Blocks_R = 1,000,000
  - TPMMS requires $\text{MBlocks} \geq \sqrt{\text{Blocks}_R} \rightarrow \text{MBlocks} \geq 1000$ 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 MBlocks groups and sort each Ri recursively
  - Merge the sublists
  - Total Cost: $(2k - 1)$ Blocks_R, where k is the number of passes required
  - We need $\sqrt[k]{\text{Blocks}_R}$ buffers, i.e., $\text{Blocks}_R \leq \text{MBlocks}^k$
- The same can be done for hash based algorithms

Evaluating Physical Plans



Picking a Query Plan (Logical of Physical)

- We looked at (simple) methods for cost estimation
  - Logical query plans: number of tuples
  - Physical query plans: number of IO blocks
- Even in such simple model there are tricky things:
  - e.g., in reality, the physical plan may not be possible (memory requirement not satisfied)
- Real DBMSs use much more complicated techniques
  - We will see examples
- How to chose a plan among many?

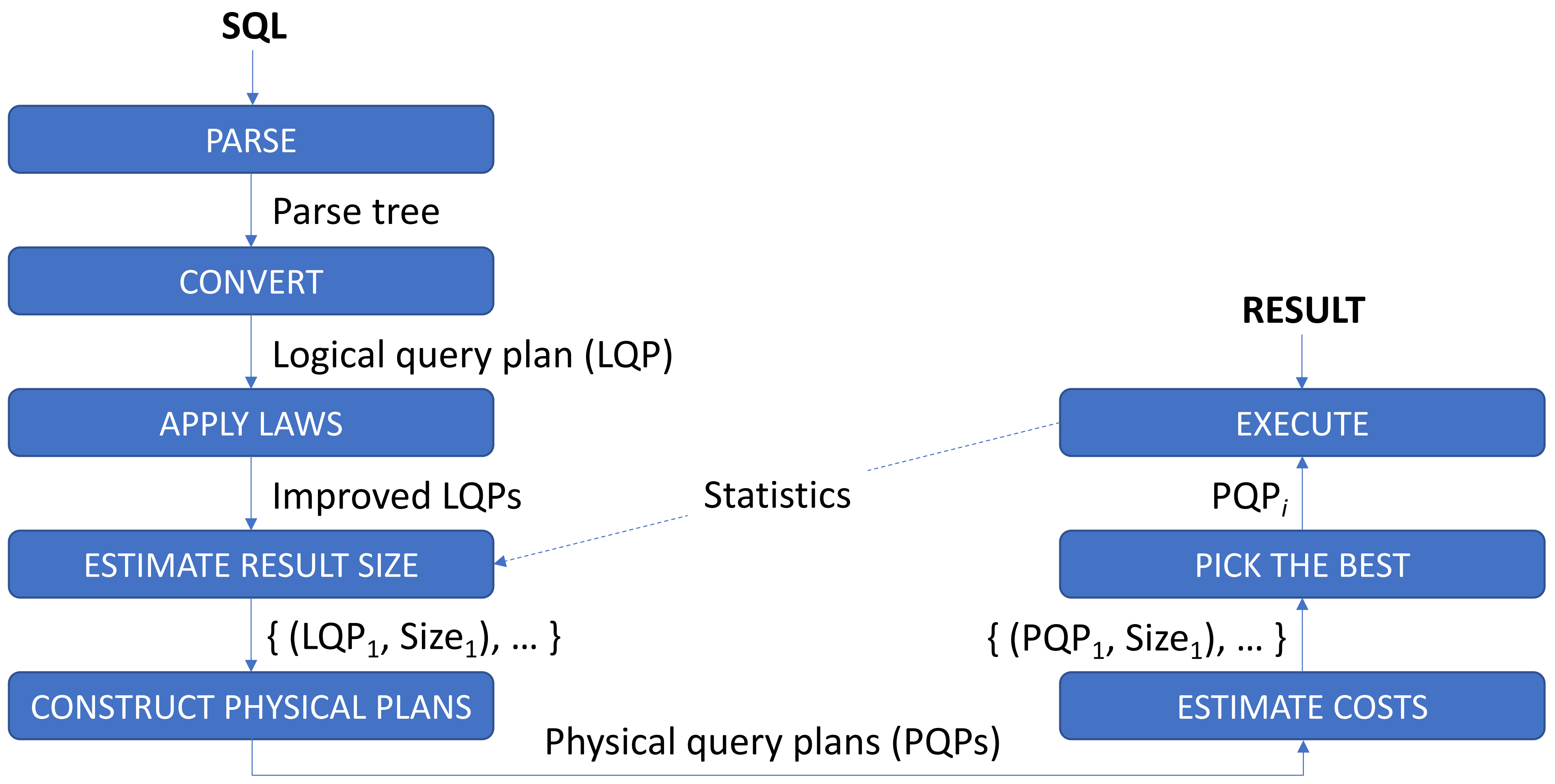
Picking a Query Plan (Logical or Physical)

- Options for choosing the “cheapest” query plan:
  - Exhaustive
  - Heuristic
  - branch-and-bound
  - Hill-climbing
  - dynamic programming
  - Sellinger-style optimizations
  - ....

Picking Query Plan (Logical or Physical)

- Exhaustive:
  - Look at all possible combinations of choices in the plan
  - Estimate the cost of each plan
  - Many plans, prohibitively expensive
- Heuristic (brute force):
  - Choose a plan according to heuristic rules based on past experience, such as:
    - Assume using index on operation $\sigma_{A = 10}(R)$
    - Assume using the smallest relation first when joining many relations
    - if arguments are sorted, user sort-based operator
    - ...
- ...

Overview: The (Typical) Journey of a Query



Query Plans in Real DBMSs

- Real DBMSs estimate costs based on their knowledge on the system (block size, main-memory operation costs, etc.)
- For example, in Postgres the defaults are something like
  - seq_page_cost = 1
  - random_page_cost = 4
  - cpu_tuple_cost = 0.01 (100x faster)
  - cpu_operator_cost = 0.0025 (400x faster)
  - cpu_index_tuple_cost = 0.005 (200x faster)
- Usually updated automatically, but sometimes can be manually changed

Query Plans in Real DBMSs

- Postgres has EXPLAIN command that provides information on query plans and their costs **(Play with it!)**
- https://wiki.postgresql.org/wiki/Using_EXPLAIN
- Keywords: Seq scan, Bitmap index scan, Loop, Hash, etc.

```
EXPLAIN SELECT * FROM POST ORDER BY body LIMIT 50;

Limit (cost = 23283.24..23283.37 rows = 50 width = 422)
-> Sort (cost = 23283.24..23859.27 rows = 230412 width = 422)
   Sort Key: body
-> Seq Scan on post (cost = 0.00..15629.12 rows = 230412 width = 422)
```

Postgres EXPLAIN example

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (actual time=0.080..0.526 rows=100
loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual time=0.049..0.049 rows=100
loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

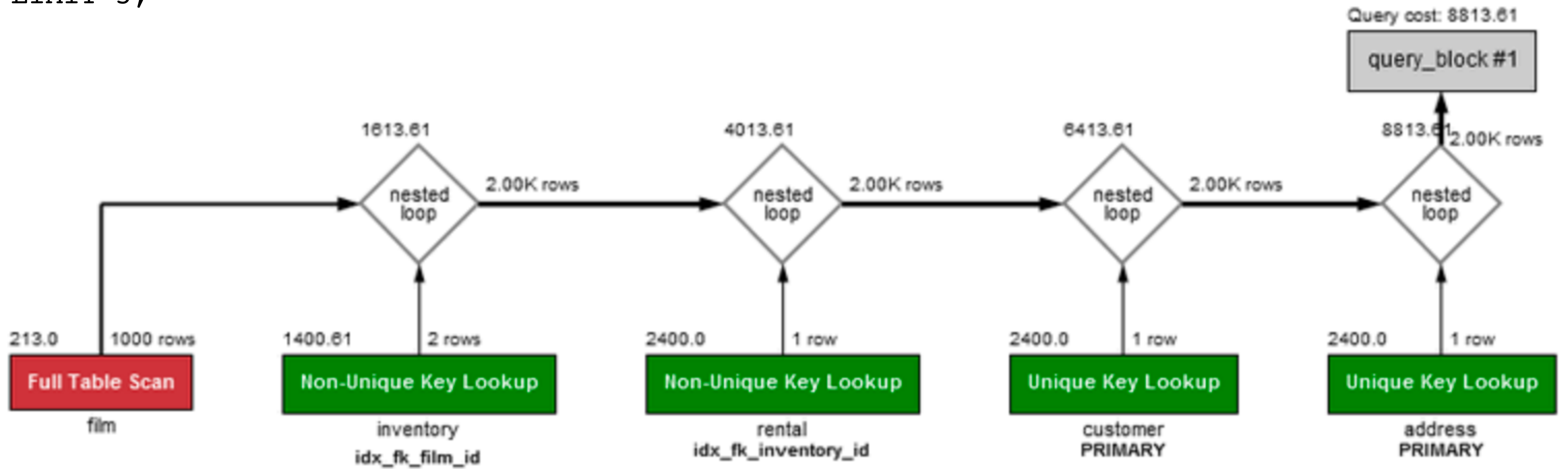
```
EXPLAIN (analyze, costs) SELECT filmid, title, count(*)
FROM film NATURAL JOIN filmgenre
GROUP BY filmid, title
ORDER BY count(*) DESC
LIMIT 25;

Limit  (cost=3118.16..3118.22 rows=25 width=32) (actual time=130.368..130.383 rows=25 loops=1)
  ->  Sort  (cost=3118.16..3198.41 rows=32100 width=32) (actual time=130.365..130.368 rows=25 loops=1)
        Sort Key: (count(*)) DESC
        Sort Method: top-N heapsort  Memory: 27kB
        ->  HashAggregate  (cost=1891.32..2212.32 rows=32100 width=32) (actual time=110.248..121.549 rows=24014 loops=1)
              Group Key: film.filmid, film.title
              ->  Hash Join  (cost=960.25..1617.76 rows=36475 width=24) (actual time=46.470..88.327 rows=33877 loops=1)
                    Hash Cond: (filmgenre.filmid = film.filmid)
                    ->  Seq Scan on filmgenre  (cost=0.00..561.75 rows=36475 width=4) (actual time=0.015..21.969 rows=36475 loops=1)
                    ->  Hash  (cost=559.00..559.00 rows=32100 width=24) (actual time=46.125..46.126 rows=32100 loops=1)
                          Buckets: 32768  Batches: 1  Memory Usage: 2033kB
                          ->  Seq Scan on film  (cost=0.00..559.00 rows=32100 width=24) (actual time=0.014..30.026 rows=32100 loops=1)
 Planning Time: 94.753 ms
 Execution Time: 132.680 ms
(14 rows)
```

Query Plans in Real DBMSs

- For Postgres plans: there are tools for visualizing, e.g., http://tatiyants.com/pev/
- Oracle has EXPLAIN PLAN and customizable plan table outputs https://docs.oracle.com/database/121/TGSQL/tgsql_genplan.htm#TGSQL94709
- MySQL has the visualization tools built in
  https://docs.oracle.com/cd/E17952_01/workbench-en/wb-performance-explain.html

MySQL visual explain plan example

```
SELECT CONCAT(customer.last_name, ', ', customer.first_name) AS customer, address.phone, film.title
FROM rental INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE()
LIMIT 5;
```


Query cost: 8813.61
query_block #1
1613.61
nested loop
2.00K rows
4013.61
nested loop
2.00K rows
6413.61
nested loop
2.00K rows
8813.61
2.00K rows
nested loop
213.0
1000 rows
Full Table Scan
film
1400.61
2 rows
Non-Unique Key Lookup
inventory
idx_fk_film_id
2400.0
1 row
Non-Unique Key Lookup
rental
idx_fk_inventory_id
2400.0
1 row
Unique Key Lookup
customer
PRIMARY
2400.0
1 row
Unique Key Lookup
address
PRIMARY