Databases

Query Evaluation & Optimization

Pierre Senellart

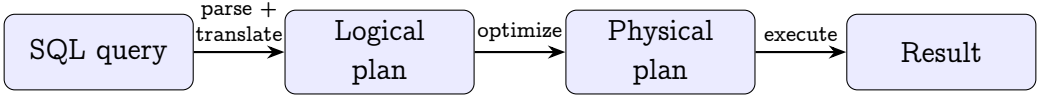
ENS ÉCOLE NORMALE SUPÉRIEURE | PSL

15 April 2026

Plan

How a DBMS processes a query

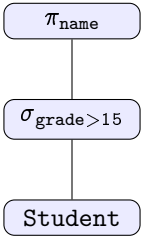
1. Parsing: SQL text → parse tree
2. Translation: parse tree → logical query plan (relational algebra expression)
3. Optimization: logical plan → optimized physical plan (choice of algorithms and access methods)
4. Execution: physical plan → result tuples



Logical plan = relational algebra tree

- A logical plan is a tree of relational algebra operators
- Leaves are base tables; internal nodes are operators (σ, π, $\bowtie$, $\cup$, $\cap$, $-$, $\rho \ldots$)
- Multiple equivalent trees may exist for the same query
- In practice, the DBMS works in multiset (bag) semantics: duplicates are kept unless explicitly removed (e.g., **DISTINCT**)

Example: **SELECT** `name` **FROM** `Student` **WHERE** `grade` > 15



Physical plan

- A physical plan annotates each node of the logical plan with:
  - the algorithm to use (e.g., nested loop join vs. hash join)
  - the access method (e.g., sequential scan vs. index scan)
  - how intermediate results flow between operators
- The query optimizer explores many physical plans and picks the one with lowest estimated cost
- Cost is usually measured in number of page I/Os (see earlier lecture on storage and indexing)

Plan

The iterator (Volcano) model [Graefe, 1994]

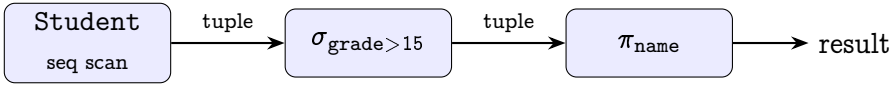
- Each physical operator implements three methods:

  `open()`: initialize the operator (allocate buffers, open files or child iterators)
  `next()`: return the next output tuple, or signal end-of-data
  `close()`: release resources

- The root operator calls `next()` repeatedly; each call pulls data up through the tree
- Advantages:
  - Pipelining: tuples flow one at a time without materializing full intermediate results
  - Uniform interface: any operator can sit above any other
- This is a pull-based (demand-driven) model: the consumer asks for the next tuple

Pipelining: example

Consider: **SELECT** `name` **FROM** `Student` **WHERE** `grade > 15`



- The scan reads one tuple at a time from disk
- Each tuple is immediately passed to σ, then to π, then to the output
- No need to store all of `Student` in memory first
- Especially powerful with **LIMIT**: adding **LIMIT** `5` lets the pipeline stop after producing 5 results, without scanning the rest of the table

Pull-based vs. push-based execution

Pull-based: the consumer calls `next()` on its child (Volcano model). Simple, composable; one virtual function call per tuple per operator.

Push-based: the producer drives a loop and pushes tuples into the next operator. Within a pipeline segment, this is a single loop body where all non-blocking operators are inlined – no per-operator function calls.

	Pull (Volcano)	Push
Control flow	consumer-driven	producer-driven
Granularity	one tuple at a time	one tuple, fused operators
Overhead	virtual call / tuple / op	one loop, operators inlined
Example	PostgreSQL	HyPer

- Orthogonal optimization: vectorized execution processes batches of ∼1000 tuples, amortizing interpretation overhead and enabling SIMD (MonetDB, DuckDB)

Example: pull vs. push

SELECT `name` **FROM** `Student` **WHERE** `grade > 15` (pipeline: Scan $\rightarrow \sigma \rightarrow \pi$)

Pull-based: three `next()` methods calling each other:

```
proj.next():                filter.next():
  t = filter.next()           loop:
  return t.name                 t = scan.next()
                                if t.grade > 15: return t
```

Each output tuple requires 3 virtual function dispatches.

Push-based (compiled): the scan drives one loop; σ and π are inlined:

```
for each tuple t in Student:
    if t.grade > 15:        -- filter inlined
        emit(t.name)        -- projection inlined
```

One tight loop, no function calls at operator boundaries.

Plan

Implementing selection (σ)

Sequential scan: read every page of the table, test each tuple against the predicate

- Cost: B page I/Os (B = number of pages in the table)
- Always works, but expensive for selective predicates

Index scan: use a B^+-tree or hash index on the selection attribute

- For equality on a hash index: ≈ 1 I/O
- For equality or range on a B^+-tree: $\approx h +$ matching pages (h = tree height, typically 2–4)
- Much cheaper when the predicate is selective (few tuples qualify)

Bitmap scan: for moderately selective predicates: scan the index, build a bitmap of matching page addresses, then fetch those pages in physical order

- Avoids the random I/O of a plain index scan when many rows match, but still skips irrelevant pages
- Can combine bitmaps from multiple indexes (AND/OR)

Key insight: the optimizer chooses between these access methods based on selectivity estimates

Plan

Sorting: why it matters

- Many operations require sorted data:
  - **ORDER BY**
  - Sort-merge join
  - Duplicate elimination (**DISTINCT**)
  - Grouped aggregation (**GROUP BY**)
- Data may be too large to sort in memory
- We need an external sorting algorithm
- Idea: sort small chunks in memory, write them to disk, then merge the sorted chunks together

External merge sort

1. Sort phase: read M pages at a time, sort in memory, write back to disk as a sorted run. Produces $\lceil B/M \rceil$ sorted runs.
2. Merge phase: merge up to $M-1$ runs at once (one buffer page per run, one for output). If more than $M-1$ runs, repeat in multiple passes.

I/O cost: each pass reads and writes all B pages: $2B$ I/Os per pass.
Number of passes $= 1 + \lceil \log_{M-1}(\lceil B/M \rceil) \rceil$

$$\text{Total cost} = 2B \times (1 + \lceil \log_{M-1}(\lceil B/M \rceil) \rceil)$$

With enough buffer pages, 2 passes (sort + one merge) often suffice.

External merge sort: a small example

12 pages of data, 4 pages of buffer memory.

Step 1 – Create sorted runs: read 4 pages at a time, sort them in memory, write back to disk.

	Pages 1–4	Pages 5–8	Pages 9–12
On disk (unsorted)	7, 2, 5, 9	1, 8, 3, 11	4, 10, 6, 12
After in-memory sort	2, 5, 7, 9	1, 3, 8, 11	4, 6, 10, 12

We now have 3 sorted runs.

Step 2 – Merge the sorted runs: use 3 buffer pages for inputs (one per run) and 1 for output. Repeatedly pick the smallest value among the input buffers, write it to output. When an input buffer is exhausted, load the next page from that run.

Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

External merge sort: large example

$B = 1000$ pages, $M = 10$ buffer pages.

1. Sort phase: produce $\lceil 1000/10 \rceil = 100$ sorted runs.
   Cost: $2 \times 1000 = 2000$ I/Os (read all, write all).
2. Merge pass 1: merge 100 runs, 9 at a time $\rightarrow \lceil 100/9 \rceil = 12$ runs.
   Cost: $2 \times 1000 = 2000$ I/Os.
3. Merge pass 2: merge 12 runs $\rightarrow \lceil 12/9 \rceil = 2$ runs.
   Cost: $2 \times 1000 = 2000$ I/Os.
4. Merge pass 3: merge 2 runs $\rightarrow 1$ sorted file.
   Cost: $2 \times 1000 = 2000$ I/Os.

Total: $4 \times 2000 = 8000$ I/Os $\quad (= 2B \times 4$ passes$)$

External merge sort: examples

B	M	Initial runs	Passes	I/O cost
1 000	10	$\lceil 1000/10 \rceil = 100$	$1 + \lceil \log_9(100) \rceil = 4$	$2 \times 1000 \times 4 = 8\,000$
500	52	$\lceil 500/52 \rceil = 10$	$1 + \lceil \log_{51}(10) \rceil = 2$	$2 \times 500 \times 2 = 2\,000$
1 000	52	$\lceil 1000/52 \rceil = 20$	$1 + \lceil \log_{51}(20) \rceil = 2$	$2 \times 1000 \times 2 = 4\,000$

Plan

The join: the most important operator

- Joins combine tuples from two (or more) relations
- Joins are very common in SQL queries
- Join performance is often the bottleneck
- Multiple algorithms exist, each suited to different situations
- We analyze equi-joins: $R \bowtie_{R.a=S.b} S$

Notation:

- B_R, B_S: number of pages in R and S
- $|R|$, $|S|$: number of tuples in R and S
- M: number of available buffer pages

Note: these are binary join algorithms; for multi-way joins, the optimizer builds a tree of binary joins (see lecture on optimal joins for worst-case optimal alternatives)

Nested loop join

```
for each tuple r in R
  for each tuple s in S
    if r.a = s.b then
      output (r, s)
    end if
  end for
end for
```

- For each tuple in R, we scan all of S
- I/O cost: $B_R + |R| \cdot B_S$
- Very expensive: for each *tuple* (not page!) of R, we read all of S
- Tip: make the smaller relation the outer one (minimizes $|R|$)
- Only practical when the inner relation fits in the buffer pool

Block nested loop join

for each block b_R of $M-2$ pages of R
  for each page p_S of S
    for each tuple r in b_R, each tuple s in p_S
      if $r.a = s.b$ **then**
        output (r, s)
      end if
    end for
  end for
end for

- Use $M-2$ buffer pages for R, 1 for S, 1 for output
- I/O cost: $B_R + \lceil B_R/(M-2) \rceil \cdot B_S$
- Much better: we scan S once per *block* of R, not per tuple
- Tip: make the smaller relation the outer one
- Not all systems implement this: PostgreSQL uses only tuple-at-a-time nested loop, relying on an index scan on the inner side instead

Index nested loop join

for each tuple r in R
  use index on $S.b$ to find matching tuples
  for each matching tuple s
    output (r, s)
  end for
end for

- Requires an index on the join attribute of the inner relation S
- I/O cost: $B_R + |R| \cdot$ (index lookup cost)
- With a B$^+$-tree: lookup $\approx$ 2–4 I/Os per tuple
- If the index is not covering (does not contain all needed columns), add 1 extra I/O per match to fetch the actual data page
- Very efficient when R is small and a good index on S is available

Sort-merge join

1. Sort R on attribute a (external merge sort)
2. Sort S on attribute b (external merge sort)
3. Merge: advance cursors through both sorted files, matching equal values

- Merge phase I/O cost: $B_R + B_S$ (one scan of each)
- Total I/O cost: sort cost for R + sort cost for S + $(B_R + B_S)$
- Very efficient when one or both inputs are already sorted (e.g., clustered index)
- Produces output in sorted order, useful for subsequent operations

Hash join

1. Build phase: hash tuples of R into $M-1$ partitions written to disk; do the same for S
2. Probe phase: for each partition i, load the smaller of R_i/S_i into an in-memory hash table; scan the other and probe for matches

- I/O cost: $3(B_R + B_S)$
  - Partition phase: read + write both relations: $2(B_R + B_S)$
  - Probe phase: read both relations: $B_R + B_S$
- Requires that each partition of the smaller relation fits in memory: $\lceil B_R/(M-1) \rceil \leqslant M-2$, i.e., works when $B_R \lesssim M^2$
- Does not produce sorted output
- Often the fastest algorithm for large equi-joins when no useful index exists

Hash join: example

$R = \{1, 2, 3, 5, 8, 11\}$, $S = \{2, 3, 7, 8, 11, 13\}$ (join attribute values shown)
Hash function: $h(x) = x \bmod 3$ $(M-1 = 3$ partitions$)$

Build phase: partition both relations by h:

Partition	R_i	S_i
$h = 0$	$\{3\}$	$\{3\}$
$h = 1$	$\{1\}$	$\{7, 13\}$
$h = 2$	$\{2, 5, 8, 11\}$	$\{2, 8, 11\}$

Probe phase: for each partition i, build hash table on R_i, scan S_i:

- Partition 0: $R_0 = \{3\}$ vs. $S_0 = \{3\}$ $\Rightarrow$ match 3
- Partition 1: $R_1 = \{1\}$ vs. $S_1 = \{7, 13\}$ $\Rightarrow$ no match
- Partition 2: $R_2 = \{2, 5, 8, 11\}$ vs. $S_2 = \{2, 8, 11\}$ $\Rightarrow$ matches 2, 8, 11

Join algorithms: summary

Algorithm	I/O cost	Best when
Nested loop	$B_R + \lvert R\rvert \cdot B_S$	never (naive)
Block nested loop	$B_R + \lceil \frac{B_R}{M-2} \rceil \cdot B_S$	no index, small R
Index nested loop	$B_R + \lvert R\rvert \cdot C_{\text{lookup}}$	index on inner
Sort-merge	sort $+ B_R + B_S$	pre-sorted input
Hash join	$3(B_R + B_S)$	large, unsorted

- The optimizer picks the algorithm based on table sizes, available indexes, and buffer space
- Most modern DBMS support all five; the three most commonly used in practice are hash join, merge join, and nested loop (with index)

Worked example: comparing join costs

Setup: $B_R = 500$, $B_S = 1000$, $|R| = 5000$, $|S| = 40\,000$, $M = 52$ buffer pages.

Nested loop: $500 + 5000 \times 1000 = 5\,000\,500$ I/Os

Block nested loop: $500 + \lceil 500/50 \rceil \times 1000 = 500 + 10 \times 1000 = 10\,500$ I/Os

Sort-merge: Sort R: 2000 I/Os; sort S: 4000 I/Os; merge: $500 + 1000 = 1500$. Total: 7500 I/Os

Hash join: $3 \times (500 + 1000) = 4500$ I/Os

Takeaway: choice of algorithm matters enormously — from 4 500 to 5 000 500 I/Os for the same result

Worked example: with an index

Same setup. Now assume a B$^+$-tree index on $S.b$ (3 levels).

Index nested loop: $500 + 5000 \times (3 + 1) = 20\,500$ I/Os
(3 I/Os for tree traversal + 1 to fetch the data page, per outer tuple)

Index nested loop (covering): $500 + 5000 \times 3 = 15\,500$ I/Os
(no data page fetch: all columns are in the index)

- Better than naive nested loop (5 000 500), but worse than block nested loop (10 500) and hash join (4 500) here
- Index nested loop becomes the winner when the outer relation is very small (e.g., after a selective filter)
- If a selection on R keeps only 100 tuples: $B_R + 100 \times 4 = 500 + 400 = 900$ I/Os — now the fastest option by far

Lesson: the best algorithm depends on the actual data sizes, not just the table definition

Pipelining vs. materialization

- Pipelining: an operator produces output tuples as soon as it receives input (e.g., selection, projection)
- Materialization: an operator must consume *all* its input before producing any output (e.g., external sort, hash join build phase)
- Such blocking operators break the pipeline: the next operator must wait
- A good optimizer tries to minimize materializations

Pipelining	Blocking
Selection (σ)	External sort
Projection (π)	Hash join (build phase)
Nested loop (per outer tuple)	Aggregate (`GROUP BY`)

Plan

Why query optimization matters

- The same SQL query can be executed in many different ways
- The difference between a good and a bad plan can be orders of magnitude
- Example: joining 3 tables R, S, T with 1000 pages each
  - Plan A: $(R \bowtie S) \bowtie T$ – intermediate result may have 1 000 000 pages
  - Plan B: $(R \bowtie T) \bowtie S$ – intermediate result may have only 100 pages
- The optimizer must find a good plan automatically

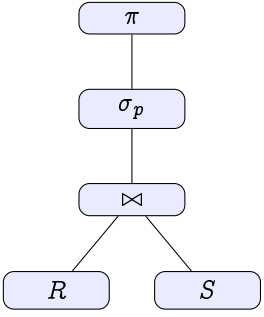
Equivalence rules for relational algebra

The optimizer rewrites the logical plan using equivalence rules:

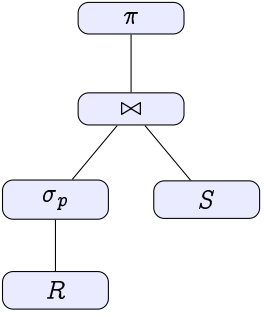
1. Selections cascade: $\sigma_{p_1 \wedge p_2}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(R))$
2. Selections commute: $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R))$
3. Projections cascade: only the outermost projection matters (if attributes are compatible)
4. Selection–join swap (push-down): if p only involves attributes of R, $\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$
5. Join commutativity: $R \bowtie S \equiv S \bowtie R$
6. Join associativity: $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$

Push selections down

Before (naive):



After (push down σ):



- Applying selections early reduces the size of intermediate results
- This is almost always beneficial (the most important heuristic)
- Similarly: push projections down to reduce tuple width earlier

Plan

Cost-based optimization

- Heuristic rules alone are not enough: the optimizer must estimate the cost of each candidate plan
- To estimate cost, the DBMS maintains statistics:
  - Number of tuples in each table ($|R|$)
  - Number of pages (B_R)
  - Number of distinct values of attribute a in R: $V(R, a)$
  - Histograms on value distributions
- The optimizer uses these to estimate:
  - Selectivity of predicates: fraction of tuples that pass a filter
  - Output size (cardinality) of each operator
  - I/O cost of each physical operator

Statistics maintained by the DBMS

- The DBMS periodically collects and stores table-level statistics:
  - Number of tuples $|R|$
  - Number of pages B_R
- And column-level statistics:
  - Number of distinct values $V(R, a)$
  - Minimum and maximum values
  - Most Common Values (MCV): list of the k most frequent values and their frequencies
  - Histogram: distribution of remaining values (after excluding MCVs)
- Statistics must be kept up to date: after bulk inserts/deletes, outdated statistics cause bad plans
- Most systems collect statistics automatically in the background

Selectivity estimation

Given a selection $\sigma_{a=v}(R)$:

- If values are uniformly distributed: selectivity $\approx 1/V(R,a)$
- Estimated output size: $|R|/V(R,a)$
- With Most Common Values (MCV) lists: use exact frequency if v is common, uniform assumption otherwise

Given a range predicate $\sigma_{a>v}(R)$:

- Selectivity $\approx (\max(a) - v)/(\max(a) - \min(a))$
- With histograms: use bucket boundaries for better estimates

Given a join $R \bowtie_{R.a=S.b} S$:

- Estimated output size $\approx |R| \cdot |S| / \max(V(R,a),\, V(S,b))$

Histograms

- A histogram partitions the domain of an attribute into buckets and records the frequency of values in each bucket
- Two main types:
  - Equi-width: all buckets cover the same value range; simple but poor for skewed data
  - Equi-depth (equi-height): each bucket contains approximately the same number of tuples; adapts better to skew
- To estimate selectivity of $\sigma_{a>v}$: find which bucket v falls in, interpolate within that bucket
- MCV values are typically stored separately and excluded from the histogram, improving accuracy for skewed distributions
- Limitation: assumes independence between attributes; correlations can cause large estimation errors

Cardinality estimation errors

- The independence assumption: for a conjunction $\sigma_{p_1 \wedge p_2}(R)$, estimate $|R| \cdot \text{sel}(p_1) \cdot \text{sel}(p_2)$
- This can be wildly wrong when attributes are correlated
- Example: `city = 'Paris'` and `country = 'France'` are strongly correlated; the product of their individual selectivities vastly underestimates the true selectivity
- Cardinality estimation errors propagate up the plan tree and can cause the optimizer to choose a catastrophically bad plan
- Some systems support multi-column statistics to track correlations explicitly
- An active area of research: learned cardinality estimators, sampling-based approaches [Leis et al., 2015]

Plan

Join ordering

- For a query joining n tables, the number of possible join orders grows very quickly:
  - 3 tables: 12 possible binary trees
  - 5 tables: 1680 possible trees
  - 10 tables: $>$ 17 billion possible trees
- The optimizer uses dynamic programming to find the best order [Selinger et al., 1979] (recall the DP approach from the lecture on optimal joins):
  1. Find the best plan for each single table
  2. Find the best plan for each pair of tables
  3. Find the best plan for each triple, reusing results from step 2
  4. Continue until the full join is planned
- For very large queries ($n > 10$): heuristics and greedy approaches
- These are all binary join plans; the lecture on optimal joins showed that worst-case optimal multi-way algorithms can avoid this enumeration entirely

The System R optimizer

- The System R optimizer [Selinger et al., 1979] is the foundational design, still used (with extensions) in PostgreSQL, Oracle, SQL Server, etc.
- Key ideas:
  1. Enumerate plans bottom-up using dynamic programming
  2. For each subplan, keep only the cheapest plan per interesting order (sorted output that avoids a later sort)
  3. Use cost formulas based on statistics to compare plans
- Interesting orders: if a join produces output sorted on a key needed by a later `ORDER BY` or merge join, a slightly more expensive sorted plan may be globally optimal
- Complexity: $O(3^n)$ for n tables; practical for $n \leqslant 10$–12
- Why 3^n? The DP considers every subset $S \subseteq \{T_1, \ldots, T_n\}$ and every way to split S into two non-empty parts $S_1 \cup S_2 = S$. Total work: $\sum_S 2^{|S|} = \sum_{k=0}^{n} \binom{n}{k} 2^k = 3^n$ (binomial theorem)

Role of indexes in optimization

- Indexes dramatically affect the optimizer's choices:
  - An index on a selection attribute makes index scan possible
  - An index on a join attribute enables index nested loop join
  - A clustered index provides sorted access (useful for sort-merge join, **ORDER BY**)
- Creating the right indexes is one of the most important performance tuning tasks
- The optimizer considers all available indexes when estimating the cost of each access path
- Trade-off: each index speeds up reads but slows down writes (every **INSERT**/**UPDATE**/**DELETE** must also update the index)

Plan

Seeing the query plan: EXPLAIN

- **EXPLAIN** shows the execution plan chosen by the optimizer without running the query

```
EXPLAIN SELECT S.name, C.title, E.grade
  FROM Student S
  JOIN Enrollment E ON S.student_id = E.student_id
  JOIN Course C ON E.course_id = C.course_id
  WHERE S.department = 'CS';
```

- Returns a tree of plan nodes (indentation = nesting)
- Each node shows the operator, its estimated cost (startup..total), and estimated rows
- Cost is in arbitrary units (roughly: sequential page reads)

EXPLAIN output: reading the plan

```
Hash Join  (cost=19.82..411.47 rows=4205 width=26)
  Hash Cond: (e.course_id = c.course_id)
  ->  Hash Join  (cost=17.70..397.37 rows=4205 width=21)
        Hash Cond: (e.student_id = s.student_id)
        ->  Seq Scan on enrollment e
              (cost=0.00..324.25 rows=21025 width=14)
        ->  Hash  (cost=15.20..15.20 rows=200 width=15)
              ->  Bitmap Heap Scan on student s
                    (cost=5.70..15.20 rows=200 width=15)
                    Recheck Cond: (department = 'CS')
                    ->  Bitmap Index Scan on idx_dept
                          (cost=0.00..5.65 rows=200 width=0)
                          Index Cond: (department = 'CS')
  ->  Hash  (cost=1.50..1.50 rows=50 width=13)
        ->  Seq Scan on course c
              (cost=0.00..1.50 rows=50 width=13)
```

PostgreSQL plan node types

Seq Scan:	sequential scan of all pages (full table scan)
Index Scan:	traverse B^+-tree, fetch matching tuples from heap
Index Only Scan:	answer from index alone (covering index)
Bitmap Index Scan:	build a bitmap of matching pages from the index
Bitmap Heap Scan:	fetch pages indicated by the bitmap; useful when many rows match (avoids random I/O)
Nested Loop:	for each outer tuple, scan inner
Hash Join:	build hash table on one input, probe with the other
Merge Join:	merge two sorted inputs
Sort:	external merge sort (blocking)
Aggregate:	`GROUP BY` / aggregate functions
Memoize:	cache results of parameterized inner scans (PostgreSQL 14+)

Example schema

```
CREATE TABLE Student (
  student_id INT PRIMARY KEY,
  name VARCHAR(100),
  department VARCHAR(50));
CREATE TABLE Course (
  course_id INT PRIMARY KEY,
  title VARCHAR(100),
  department VARCHAR(50));
CREATE TABLE Enrollment (
  student_id INT REFERENCES Student(student_id),
  course_id INT REFERENCES Course(course_id),
  grade DECIMAL(4,2),
  PRIMARY KEY (student_id, course_id));
CREATE INDEX ON Enrollment(course_id);
```

1000 students (5 departments), 50 courses, ≈21 000 enrollments.

EXPLAIN: without index on department

```
EXPLAIN SELECT name FROM Student
  WHERE department = 'CS';
```

```
Seq Scan on student
    (cost=0.00..19.50 rows=200 width=11)
  Filter: (department = 'CS')
```

- Full sequential scan: reads all 7 pages of `student`
- Applies the filter after reading each tuple
- Estimates 200 matching rows out of 1000 (selectivity $= 1/5$, since $V(\text{Student}, \text{department}) = 5$)

EXPLAIN: with index on department

```
CREATE INDEX idx_dept ON Student(department);
ANALYZE Student;

EXPLAIN SELECT name FROM Student
  WHERE department = 'CS';
```

```
Bitmap Heap Scan on student
    (cost=5.70..15.20 rows=200 width=11)
  Recheck Cond: (department = 'CS')
  ->  Bitmap Index Scan on idx_dept
        (cost=0.00..5.65 rows=200 width=0)
        Index Cond: (department = 'CS')
```

- PostgreSQL chooses a bitmap scan: first scan the index to collect matching page addresses, then fetch those pages from the heap
- Lower total cost (15.20 vs. 19.50)
- For very selective predicates ($\ll 20\%$), PostgreSQL would use a plain Index Scan instead

EXPLAIN ANALYZE: actual execution

EXPLAIN ANALYZE runs the query and reports actual vs. estimated statistics:

```
Hash Join  (cost=19.82..411.47 rows=4205 width=26)
           (actual time=0.12..1.99 rows=4174 loops=1)
  Hash Cond: (e.course_id = c.course_id)
  ->  Hash Join  (cost=17.70..397.37 rows=4205 width=21)
                 (actual time=0.10..1.69 rows=4174 loops=1)
        Hash Cond: (e.student_id = s.student_id)
        ->  Seq Scan on enrollment e
              (cost=0.00..324.25 rows=21025 width=14)
              (actual time=0.00..0.74 rows=21025 loops=1)
        ->  Hash  (cost=15.20..15.20 rows=200 width=15)
              (actual time=0.08..0.08 rows=200 loops=1)
              ->  Bitmap Heap Scan on student s  ...
  Planning Time: 0.58 ms
  Execution Time: 2.11 ms
```

- Compare `rows=4205` (estimated) vs. `rows=4174` (actual): good estimate here
- Large discrepancies signal stale statistics or correlation issues

Covering indexes

- A covering index contains all columns needed by the query
- The DBMS can answer from the index alone, without reading the heap
- Indicated by Index Only Scan in the plan

```
CREATE INDEX idx_cov
  ON Enrollment(student_id, grade);
EXPLAIN SELECT student_id, grade
  FROM Enrollment WHERE student_id = 42;
```

```
Index Only Scan using idx_cov on enrollment
    (cost=0.29..48.99 rows=26 width=10)
  Index Cond: (student_id = 42)
```

- Trade-off: wider indexes use more space and slow down writes

Plan

Statistics in PostgreSQL

- **ANALYZE** collects statistics; stored in `pg_statistic` (exposed via `pg_stats`)
- The autovacuum daemon runs **ANALYZE** automatically

```
-- Table-level statistics:
SELECT relname, reltuples::bigint, relpages
FROM pg_class WHERE relname = 'student';
--  relname | reltuples | relpages
--  student |      1000 |        7

-- Per-column statistics:
SELECT attname, n_distinct, most_common_vals
FROM pg_stats
WHERE tablename = 'student'
  AND attname = 'department';
-- n_distinct = 5
-- most_common_vals = {Biology,Chemistry,CS,Math,Physics}
```

Helping the optimizer

- **ANALYZE** (or let autovacuum do it): keep statistics fresh
- Create indexes on:
  - Columns in **WHERE** clauses
  - Columns in **JOIN** conditions
  - Columns in **ORDER BY** / **GROUP BY**
- Avoid **SELECT** $*$: prevents covering index optimizations
- Beware of functions on indexed columns:
  WHERE EXTRACT(**YEAR FROM** `d`) = `2026` cannot use an index on `d`; prefer
  WHERE `d >= '2026-01-01'` **AND** `d < '2027-01-01'`
- Use **CREATE STATISTICS** for correlated columns

Plan

Conclusion

- A DBMS translates SQL into a logical plan (relational algebra), then into an optimized physical plan
- The iterator model lets operators pipeline tuples efficiently
- Several algorithms exist for each operator; the choice depends on data size, indexes, and buffer space
- The join is the most expensive and most important operator to optimize
- The optimizer uses equivalence rules (push selections down) and cost estimation (statistics, selectivity) to pick the best plan
- **EXPLAIN** and **EXPLAIN ANALYZE** are essential tools to understand and improve query performance

References

- Classic reference on query optimization (System R optimizer): [Selinger et al., 1979]
- Chapters on query evaluation and optimization in [Garcia-Molina et al., 2008]
- Cardinality estimation in practice: [Leis et al., 2015]
- Query processing in PostgreSQL: `https://www.postgresql.org/docs/current/planner-optimizer.html`

Bibliography I

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2nd edition, 2008. ISBN 978-0-13-187325-4.

Goetz Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994. doi: 10.1109/69.273032. URL https://doi.org/10.1109/69.273032.

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015. doi: 10.14778/2850583.2850594. URL http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proc. SIGMOD*, pages 23–34. ACM, 1979. doi: 10.1145/582095.582099. URL https://doi.org/10.1145/582095.582099.