Query Evaluation & Optimization

Databases

Pierre Senellart

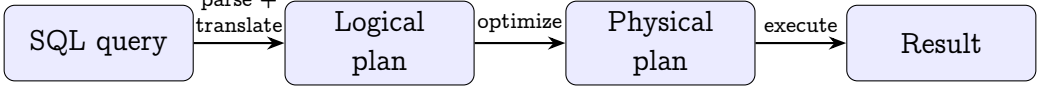
PSL UNIVERSITÉ PARIS

15 April 2026

Outline

How a DBMS processes a query

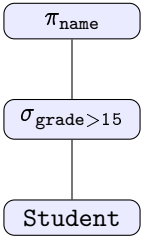
1. Parsing: SQL text → parse tree
2. Translation: parse tree → logical query plan (relational algebra expression)
3. Optimization: logical plan → optimized physical plan (choice of algorithms and access methods)
4. Execution: physical plan → result tuples



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. sort-merge join)
  - the access method (e.g., table 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 (recall session on storage)

Outline

The iterator model

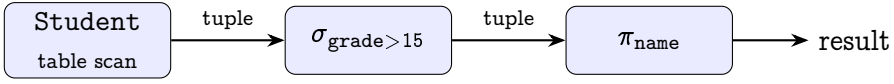
- Each physical operator implements three methods:

  `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

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

Outline

Implementing selection (σ)

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

- Cost: B page I/Os (where 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 = height of the tree, typically 2–4)
- Much cheaper when the predicate is selective (few tuples qualify)

Key insight: the optimizer chooses between scan and index based on selectivity estimates

Outline

Why sorting 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

The problem: data larger than memory

- Suppose we need to sort 12 pages of data, but we only have space for 4 pages in memory
- We cannot load everything at once and sort it
- Idea: sort small chunks, write them to disk, then merge the sorted chunks together

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 (each run is a sequence of pages in sorted order).

External merge sort: merging 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 the output
- When an input buffer is exhausted, load the next page from that run

	Run 1	Run 2	Run 3
In buffer	2, 5, 7, 9	1, 3, 8, 11	4, 6, 10, 12

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

With M buffer pages, we can merge up to $M-1$ runs at once (one buffer page per run, one page for output). This is called $(M-1)$-way merge.

What if there are too many runs?

- With M buffer pages, one merge pass can combine at most $M-1$ runs
- If we have more runs than that, we merge in multiple passes:
  - Merge the first $M-1$ runs into one, then the next $M-1$, etc.
  - This produces fewer, longer runs
  - Repeat until only one run remains

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

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

Compare with in-memory sorting: $\mathcal{O}(n \log_2 n)$ comparisons. External sort has an analogous log factor, but with base $M-1$ (the fan-in of each merge pass).

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: more examples

Using the formula: $2B \times (1 + \lceil \log_{M-1}(\lceil B/M \rceil) \rceil)$

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	$\lceil 1000/52 \rceil = 20$	$1 + \lceil \log_{51}(20) \rceil = 2$	$2 \times 1000 \times 2 = 4\,000$

- With enough buffer pages, 2 passes (sort + one merge) often suffice
- The last two examples are used in the join cost comparisons

Outline

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

Nested loop join (tuple-level)

```
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 R                          ▷ read M−2 pages of R
  for each block b_S of S                          ▷ read 1 page of S
    for each tuple r in b_R
      for each tuple s in b_S
        if r.a = s.b then
          output (r, s)
        end if
      end for
    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

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 covering 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: apply a hash function h to the join attribute to distribute tuples of R into $M-1$ partitions written to disk (1 input buffer page + $M-1$ output buffer pages); 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
- In MySQL (InnoDB), the most common join strategies are block nested loop and index nested loop

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 (see sort examples); 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: $B_R = 500$, $B_S = 1000$, $|R| = 5000$, $|S| = 40\,000$, $M = 52$.
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 needed: 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

Pipelining vs. materialization

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

Outline

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

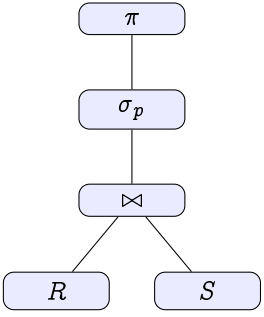
Equivalence rules for relational algebra

The optimizer rewrites the logical plan using equivalence rules:

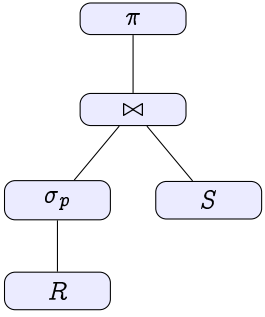
1. Selections cascade: $\sigma_{p_1 \wedge p_2}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(R))$
2. Selections commute: $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R))$
3. Projections cascade: only the outermost projection matters (if attributes are compatible)
4. Selection–join swap (push-down):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

The most important heuristic: push selections down

Before (naive):



After (push down σ):



- Applying selections early reduces the size of intermediate results
- This is almost always beneficial

Outline

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, written $V(R, a)$
  - Histograms on value distributions (in some systems)
- The optimizer uses these to estimate:
  - Selectivity of predicates: fraction of tuples that pass a filter
  - Output size of each operator
  - I/O cost of each physical operator

Selectivity estimation

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

- If values are uniformly distributed: selectivity $\approx 1/V(R, a)$
- Estimated output size: $|R|/V(R, a)$

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

- Selectivity $\approx (\max(a) - v)/(\max(a) - \min(a))$

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

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

Key point: these are estimates – real data is rarely uniform, so errors happen

Join ordering

- For a query joining n tables, the number of possible join orders grows very quickly:
  - 3 tables: 12 possible trees
  - 5 tables: 1680 possible trees
  - 10 tables: $>$ 17 billion possible trees
- The optimizer uses dynamic programming to find the best order:
  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 are used instead

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

Outline

Seeing the query plan: EXPLAIN

- In MySQL, `EXPLAIN` shows the execution plan chosen by the optimizer
- Syntax:

SQL

```sql
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 one row per table accessed in the query
- Shows the access method, join type, key used, estimated rows, etc.

EXPLAIN output: key columns

type: access method used for this table:

- `ALL` – full table scan (worst)
- `index` – full index scan
- `range` – index range scan
- `ref` – index lookup (non-unique)
- `eq_ref` – index lookup (unique / primary key)
- `const` – unique/PK lookup on a constant: row resolved at planning time

possible_keys: indexes the optimizer considered

key: the index actually chosen

rows: estimated number of rows to examine

Extra: additional info (`Using where`, `Using index`, `Using temporary`, `Using filesort`)

EXPLAIN example (1/2)

SQL

```
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),
  INDEX (course_id));
```

1000 students, 50 courses, ≈21 000 enrollments.

EXPLAIN example (2/2)

SQL

```sql
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';
```

MySQL output (no index on department):

table	type	key	Extra	rows
S	ALL	NULL	Using where	1000
E	ref	PRIMARY		21
C	eq_ref	PRIMARY		1

- S: full table scan, filtered by WHERE (no index on department)
- E: index lookup via primary key for each student
- C: primary key lookup for each enrollment

Improving a query with an index

SQL

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

Same query, after adding the index:

table	type	key	Extra	rows
S	ref	idx_dept		200
E	ref	PRIMARY		21
C	eq_ref	PRIMARY		1

- The index allows an index lookup instead of a full table scan
- Estimated rows for S drops from 1000 to 200
- `Using where` disappears: the index handles the filter directly

EXPLAIN ANALYZE

- `EXPLAIN ANALYZE` actually runs the query and reports real execution statistics
- Shows actual time and actual rows vs. estimates
- Useful to detect cases where the optimizer's estimates are wrong

Output (abbreviated):

Text

```
-> Nested loop inner join
     (cost=2052 rows=4392)
     (actual time=0.079..6.48 rows=4286)
   -> Index lookup on S using idx_dept
        (cost=23.8 rows=200)
        (actual time=0.050..0.288 rows=200)
   -> Index lookup on E using PRIMARY
        (cost=0.27 rows=22)
        (actual time=0.006..0.009 rows=21.4)
   -> Single-row index lookup on C
        (cost=0.25 rows=1)
        (actual time=0.001..0.001 rows=1)
```

Outline

Covering indexes

- A covering index contains all columns needed by the query
- The DBMS can answer the query from the index alone, without reading the data pages
- Indicated by `Using index` in the `Extra` column of `EXPLAIN`

SQL

```sql
-- This index covers the query below:
CREATE INDEX idx_cov
  ON Enrollment(student_id, grade);

EXPLAIN SELECT student_id, grade
  FROM Enrollment
  WHERE student_id = 42;
-- Extra: Using index
```

- Particularly effective for read-heavy workloads
- Trade-off: wider indexes cost more space and slow down writes

Helping the optimizer

- `ANALYZE TABLE R;` – update statistics so the optimizer has accurate data
- Create indexes on:
  - Columns used in `WHERE` clauses
  - Columns used in `JOIN` conditions
  - Columns used in `ORDER BY` / `GROUP BY`
- Avoid `SELECT *` – retrieving unnecessary columns prevents covering index optimizations
- Beware of functions on indexed columns:
  `WHERE YEAR(date_col) = 2026` cannot use an index on `date_col`; prefer
  `WHERE date_col BETWEEN '2026-01-01' AND '2026-12-31'`

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` is the essential tool to understand and improve query performance in MySQL

Licensing