EXPLAIN ANALYZE: Reading the Database's Mind

Why EXPLAIN Matters

You've written a query. It works. But is it **fast**?

Without EXPLAIN, you're flying blind. You don't know:

- Which indexes are used
- What join strategies the optimizer chose
- How many rows are scanned
- Where the bottleneck is

EXPLAIN shows the query plan. EXPLAIN ANALYZE shows the plan **and** executes the query, giving you actual timing.

Basic EXPLAIN Syntax

EXPLAIN (Plan Only)

```
EXPLAIN
SELECT * FROM users WHERE email = 'alice@example.com';
```

Output (Postgres):

```
Index Scan using idx_users_email on users  (cost=0.29..8.30 rows=1 width=128)
  Index Cond: ((email)::text = 'alice@example.com'::text)
```

What it tells you:

- Using index `idx_users_email`
- Estimated cost: 0.29 (startup) to 8.30 (total)
- Estimated rows: 1
- Row width: 128 bytes

Note: This doesn't execute the query. It's the optimizer's **estimate**.

EXPLAIN ANALYZE (Actual Execution)

```
EXPLAIN ANALYZE
SELECT * FROM users WHERE email = 'alice@example.com';
```

Output:

```
Index Scan using idx_users_email on users  (cost=0.29..8.30 rows=1 width=128) (actual
time=0.025..0.026 rows=1 loops=1)
  Index Cond: ((email)::text = 'alice@example.com'::text)
Planning Time: 0.123 ms
Execution Time: 0.051 ms
```

What it adds:

- **actual time:** Real execution time (ms)
- **rows:** Actual row count (1)
- **loops:** How many times this step ran (1)
- **Planning Time:** Time to generate the plan
- **Execution Time:** Time to run the query

Warning: EXPLAIN ANALYZE **executes the query**. Don't run it on UPDATE/DELETE in production without a transaction:

```
BEGIN;
EXPLAIN ANALYZE DELETE FROM users WHERE ...;
ROLLBACK;
```

Reading EXPLAIN Output

The Structure: Execution Nodes

EXPLAIN output is a **tree** of execution nodes.

Example:

```
Nested Loop  (cost=0.58..16.61 rows=1 width=256)
  ->  Index Scan using idx_orders_user_id on orders  (cost=0.29..8.30 rows=1
width=128)
        Index Cond: (user_id = 123)
  ->  Index Scan using users_pkey on users  (cost=0.29..8.30 rows=1 width=128)
        Index Cond: (id = orders.user_id)
```

How to read it:

- **Indentation** shows nesting
- **Top node** is the final step
- **Child nodes** feed into parent

Execution order: Bottom-up (leaves to root).

1. Scan `idx_orders_user_id` → find orders for user 123
2. For each order, scan `users_pkey` → fetch user data
3. Nested loop joins them

Cost Notation: `cost=X..Y`

Format: `cost=startup..total`

- **Startup cost:** Time to produce the first row
- **Total cost:** Time to produce all rows

Units: Arbitrary (not seconds). Used for comparison.

Example:

```
Seq Scan on users  (cost=0.00..1843.50 rows=100000 width=128)
Index Scan on users  (cost=0.29..8.30 rows=1 width=128)
```

Interpretation: Index scan is **much** cheaper (8.30 vs 1843.50).

Note: Cost includes CPU + I/O. Configured by planner settings (`seq_page_cost` , `random_page_cost` , etc.).

Rows Estimate vs Actual

Estimate: `rows=1000` — What the optimizer thinks

Actual: `actual rows=50000` — What actually happened

Large mismatch = bad. The optimizer made wrong assumptions (stale statistics, bad estimate).

Fix: Run `ANALYZE` to update statistics.

Width: Row Size

Width: Average row size in bytes.

Example: `width=128` means each row is ~128 bytes.

Why it matters: Wider rows = more I/O, slower scans.

Actual Time: Wall-Clock Timing

Format: `actual time=startup..total`

Example: `actual time=0.025..1.520`

- **0.025 ms:** Time to first row
- **1.520 ms:** Time to last row

Note: This is **cumulative** for child nodes (includes children's time).

Loops: How Many Times a Node Ran

Format: `loops=N`

Example:

```
Nested Loop  (actual rows=100 loops=1)
  ->  Seq Scan on users  (actual rows=10 loops=1)
  ->  Index Scan on orders  (actual rows=10 loops=10)
```

Interpretation:

- Users scanned once (10 rows)
- For each user, orders scanned (10 rows each)
- Orders node ran 10 times (loops=10)

Total rows from orders: 10 rows/loop × 10 loops = 100 rows.

High loops = expensive. Nested loops with many iterations are slow.

Common Execution Nodes

Sequential Scan

```
Seq Scan on users  (cost=0.00..1843.50 rows=100000 width=128)
  Filter: (active = true)
```

What it does: Reads every row in the table.

When it's used:

- No suitable index
- Table is small
- Query matches most rows

Filter: Applied after reading rows (not efficient).

Optimization: Add an index on `active`, or use a partial index.

Index Scan

```
Index Scan using idx_users_email on users  (cost=0.29..8.30 rows=1 width=128)
  Index Cond: (email = 'alice@example.com')
```

What it does: Use the index to find rows, then fetch from table.

Index Cond: Condition applied using the index (efficient).

When it's fast: Low row count (high selectivity).

Index Only Scan

```
Index Only Scan using idx_users_email_name on users  (cost=0.29..8.30 rows=1 width=64)
  Index Cond: (email = 'alice@example.com')
  Heap Fetches: 0
```

What it does: Fetch all data from the index (no table access).

Heap Fetches: How many times it had to check the table (for visibility). 0 = fully satisfied by index.

When it's possible: Covering index (index includes all needed columns).

Fastest scan type.

Bitmap Index Scan + Bitmap Heap Scan

```
Bitmap Heap Scan on users  (cost=12.34..567.89 rows=500 width=128)
  Recheck Cond: (age > 30)
  ->  Bitmap Index Scan on idx_users_age  (cost=0.00..12.20 rows=500 width=0)
        Index Cond: (age > 30)
```

What it does:

1. Scan index, build bitmap of matching row IDs
2. Scan table using the bitmap

When it's used: Moderate selectivity (more rows than index scan, fewer than seq scan).

Recheck Cond: Sometimes the bitmap is lossy (approximate), so it rechecks.

Nested Loop Join

```
Nested Loop  (cost=0.58..16.61 rows=1 width=256)
  ->  Index Scan on orders  (cost=0.29..8.30 rows=1 width=128)
  ->  Index Scan on users  (cost=0.29..8.30 rows=1 width=128)
```

What it does:

```
for each row in orders:
    for each row in users:
        if join condition matches:
            emit row
```

When it's used:

- Small outer table
- Inner table has an index on join key

Performance: O(rows_outer × log(rows_inner)) with index.

Worst case: O(rows_outer × rows_inner) without index.

Hash Join

```
Hash Join  (cost=45.00..789.00 rows=1000 width=256)
  Hash Cond: (orders.user_id = users.id)
  ->  Seq Scan on orders  (cost=0.00..567.00 rows=10000 width=128)
  ->  Hash  (cost=25.00..25.00 rows=1000 width=128)
        ->  Seq Scan on users  (cost=0.00..25.00 rows=1000 width=128)
```

What it does:

1. Build a hash table from `users`
2. Scan `orders`, probe hash table

When it's used:

- Large tables
- Equijoin (=)
- No suitable index

Performance: O(rows_users + rows_orders)

Memory: Hash table must fit in `work_mem`. If not, spills to disk (slow).

Merge Join

```
Merge Join  (cost=123.45..678.90 rows=1000 width=256)
  Merge Cond: (orders.user_id = users.id)
  ->  Sort  (cost=80.00..85.00 rows=2000 width=128)
        Sort Key: orders.user_id
        ->  Seq Scan on orders  (cost=0.00..567.00 rows=10000 width=128)
  ->  Sort  (cost=40.00..42.50 rows=1000 width=128)
        Sort Key: users.id
        ->  Seq Scan on users  (cost=0.00..25.00 rows=1000 width=128)
```

What it does:

1. Sort both tables by join key
2. Merge-scan through both (like mergesort)

When it's used:

- Both tables are already sorted (or have indexes)
- Equijoin

Performance: O(rows_users + rows_orders) if pre-sorted, else O(N log N) for sorting.

Aggregate

```
Aggregate  (cost=1843.50..1843.51 rows=1 width=8)
  ->  Seq Scan on users  (cost=0.00..1843.50 rows=100000 width=0)
```

What it does: Compute aggregate (COUNT, SUM, AVG, etc.).

No GROUP BY: Single result row.

HashAggregate

```
HashAggregate  (cost=1843.50..1853.50 rows=1000 width=12)
  Group Key: user_id
  ->  Seq Scan on orders  (cost=0.00..1343.50 rows=100000 width=12)
```

What it does: GROUP BY using a hash table.

When it's used: Default for GROUP BY (if result fits in memory).

Disk Usage: If hash table exceeds `work_mem`, it spills to disk (look for "Disk Usage" in ANALYZE output).

GroupAggregate

```
GroupAggregate  (cost=0.42..123.45 rows=1000 width=12)
  Group Key: user_id
  ->  Index Scan using idx_orders_user_id on orders  (cost=0.42..100.00 rows=10000
width=12)
```

What it does: GROUP BY on pre-sorted data.

When it's used: Input is already sorted (index scan or explicit sort).

Performance: More efficient than HashAggregate if data is sorted.

Sort

```
Sort  (cost=123.45..128.45 rows=2000 width=128)
  Sort Key: created_at DESC
  Sort Method: quicksort  Memory: 245kB
  ->  Seq Scan on orders  (cost=0.00..567.00 rows=10000 width=128)
```

What it does: Sort rows (for ORDER BY or Merge Join).

Sort Method:

- **quicksort Memory:** In-memory sort (fast)
- **external merge Disk:** Spilled to disk (slow)

Optimization: Add an index on the sort column to avoid sorting.

Limit

```
Limit  (cost=0.42..10.42 rows=10 width=128)
  ->  Index Scan using idx_orders_created_at on orders  (cost=0.42..5000.00 rows=5000
width=128)
```

What it does: Stop after N rows.

Performance: If the input is sorted (via index), Limit is cheap. Otherwise, the database computes all rows, sorts, then truncates.

Spotting Problems

Problem 1: Seq Scan When You Expected Index Scan

```
Seq Scan on users  (cost=0.00..1843.50 rows=100000 width=128)
  Filter: (email = 'alice@example.com')
```

Why it happens:

- No index on `email`
- Index exists but isn't used (function on column, type mismatch)
- Statistics are stale

Fix:

- Create index: `CREATE INDEX idx_users_email ON users(email);`
- Check for functions: `WHERE LOWER(email) = ...` → functional index
- Update stats: `ANALYZE users;`

Problem 2: High Loop Count in Nested Loop

```
Nested Loop  (actual rows=1000000 loops=1)
  ->  Seq Scan on users  (actual rows=1000 loops=1)
  ->  Seq Scan on orders  (actual rows=1000 loops=1000)
```

Why it's bad: Orders scanned 1000 times (once per user). Total scans: 1000 × all_orders.

Fix:

- Add index on `orders.user_id`
- Consider hash join (if optimizer isn't already)

Problem 3: Rows Estimate Way Off

```
Hash Join  (cost=45.00..789.00 rows=10 width=256) (actual rows=1000000 loops=1)
```

Estimate: 10 rows
Actual: 1,000,000 rows

Why it's bad: Optimizer chose hash join expecting 10 rows, but got a million. It might have chosen a different plan.

Fix:

- Update statistics: `ANALYZE users; ANALYZE orders;`
- Increase statistics target: `ALTER TABLE users ALTER COLUMN email SET STATISTICS 1000;`

Problem 4: Sort Spilling to Disk

```
Sort  (actual rows=1000000 loops=1)
  Sort Key: created_at
  Sort Method: external merge  Disk: 123456kB
```

Why it's bad: Disk I/O is 100x slower than memory.

Fix:

- Increase `work_mem` : `SET work_mem = '256MB';`
- Add an index on `created_at` to avoid sorting

Problem 5: Hash Table Spilling to Disk

```
Hash Join  (actual rows=1000000 loops=1)
  Hash Buckets: 524288  Batches: 16  Memory Usage: 123456kB
```

"Batches: 16" means it spilled to disk 16 times.

Fix:

- Increase `work_mem`
- Filter earlier (reduce rows before joining)

Practical Debugging Workflow

Step 1: Run EXPLAIN ANALYZE

```sql
EXPLAIN ANALYZE
SELECT ...;
```

Step 2: Identify the Expensive Node

Look for:

- **Highest cost**
- **Longest actual time**
- **Most rows scanned**

Example:

```
Nested Loop  (actual time=0.05..5234.67 rows=1000 loops=1)
  ->  Seq Scan on users  (actual time=0.01..0.50 rows=10 loops=1)
  ->  Seq Scan on orders  (actual time=0.01..523.40 rows=100 loops=10)
```

Bottleneck: `Seq Scan on orders` (runs 10 times, 523 ms each).

Fix: Add index on `orders.user_id`.

Step 3: Check Index Usage

Look for:

- `Seq Scan` when you expect `Index Scan`
- `Filter:` conditions (applied after scan, not via index)

Fix: Create appropriate indexes.

Step 4: Check Row Estimates

Compare: `rows=X` (estimate) vs `actual rows=Y`

If way off: Run `ANALYZE`.

Step 5: Check for Sorts and Spills

Look for:

- `Sort Method: external merge` (disk sort)
- `Batches: >1` in hash joins (disk spill)

Fix: Increase `work_mem` or reduce rows.

Step 6: Optimize and Re-Run

After making changes (indexes, query rewrite), run EXPLAIN ANALYZE again.

Compare:

- Before: `Execution Time: 5234 ms`
- After: `Execution Time: 23 ms`

Success!

EXPLAIN Options (Postgres)

BUFFERS: See I/O Stats

```sql
EXPLAIN (ANALYZE, BUFFERS)
SELECT ...;
```

Output:

```
Buffers: shared hit=1234 read=567 written=0
```

- **hit:** Pages found in cache (fast)

- **read:** Pages read from disk (slow)
- **written:** Pages written (for sorts/hashes)

High `read` = I/O bottleneck.

VERBOSE: Show More Detail

```
EXPLAIN (ANALYZE, VERBOSE)
SELECT ...;
```

Shows: Column lists, output columns, etc.

Use when: Debugging complex queries.

FORMAT: JSON, YAML, XML

```
EXPLAIN (ANALYZE, FORMAT JSON)
SELECT ...;
```

Use when: Parsing output programmatically.

MySQL EXPLAIN Differences

Basic Syntax

```
EXPLAIN
SELECT * FROM users WHERE email = 'alice@example.com';
```

Output (tabular):

```
+----+-------------+-------+-------+------------------+------------------+---------+--
-----+------+-------+
| id | select_type | table | type  | possible_keys    | key              | key_len |
ref   | rows | Extra |
+----+-------------+-------+-------+------------------+------------------+---------+--
-----+------+-------+
|  1 | SIMPLE      | users | ref   | idx_users_email  | idx_users_email  | 767     |
const |    1 | NULL  |
+----+-------------+-------+-------+------------------+------------------+---------+--
-----+------+-------+
```

Key fields:

- **type:** Join type (`const` , `ref` , `range` , `index` , `ALL`)
  - `const` : Single row (primary key lookup)
  - `ref` : Index lookup (non-unique)
  - `range` : Index range scan
  - `index` : Full index scan
  - `ALL` : Full table scan (bad)
- **possible_keys:** Indexes MySQL considered

- **key:** Index actually used
- **rows:** Estimated rows

EXPLAIN ANALYZE (MySQL 8.0.18+)

```
EXPLAIN ANALYZE
SELECT ...;
```

Output: Tree format with actual times (similar to Postgres).

EXPLAIN FORMAT=JSON

```
EXPLAIN FORMAT=JSON
SELECT ...;
```

Output: JSON (more detail, easier to parse).

Key Takeaways

1. **EXPLAIN shows the query plan; EXPLAIN ANALYZE executes and shows actual results.**
2. **Read output bottom-up.** Child nodes feed into parents.
3. **Cost is relative**, not absolute. Use it to compare plans.
4. **Rows estimate vs actual:** Large mismatch = bad statistics.
5. **Seq Scan isn't always bad.** For small tables or high selectivity, it's fine.
6. **Index Scan is good; Index Only Scan is better.**
7. **Nested loops with high loop counts are expensive.** Add indexes.
8. **Hash/Merge joins are for large tables.** Hash spilling to disk is bad.
9. **Sorts and aggregates spilling to disk = increase work_mem.**
10. **Always run ANALYZE after bulk changes.** Optimizer needs fresh stats.
11. **Use BUFFERS to see I/O.** High disk reads = performance problem.

Next up: Transactions, isolation levels, and locking—because concurrent writes are hard.