Mydbops
Scaling Databases

Exploring MySQL 9.7

JSON Duality Views & Hypergraph Optimizer

Mydbops MyWebinar 52

Presenter: Shahbaz Gavasekar

Organization: Mydbops

Mydbops by the Numbers

Unmatched database engineering scale

10+ years	10 B +	800 +	3000 +
Of Expertise	DB Transactions Handled per Day	Happy Clients	Tickets Handled per Day

> Your Trusted Database Management Partner

Statistical profiles

Our Services

Consulting models and technical expertise

Consulting Services

Targeted Engagement designing solutions, analyzing layout patterns, and removing barriers.

Managed Services

24×7 DBA Team providing production assurance, monitoring, security, and continuous updates.

Database Technologies

MySQL MongoDB PostgreSQL

MariaDB TiDB Cassandra

Service coverage maps

JSON Duality Views in MySQL 9.7

Agenda Outline

- Relational vs. Document Models
- What are JSON Duality Views?
- JSON Duality Views – Concurrency Control with ETAG
- The Challenge Before MySQL 9.7
- How JSON Duality Views Solve the Problem
- Test Cases
- Inspecting JSON Duality Views
- Limitations and Consideration

Relational VS Document Model

Comparing structural paradigms

Relational Model

```
club table:
  id  INT PK
  name  VARCHAR(60)

player table:
  id  INT PK
  club_id  INT FK → club
  name  VARCHAR(60)
  position  VARCHAR(40)
```

- Normalized Structure
- Referential Integrity enforced
- Querying nested data needs multi-table joins

Document Model

```
{
  "_id": 1,
  "name": "Gremio FC",
  "squad": [
    { "id": 9, "name": "Carlos Vinicius",
      "position": "Forward" },
    { "id": 5, "name": "Arthur",
      "position": "Midfielder" }
  ]
}
```

- Hierarchical Structure
- Read in one query — no joins needed
- No built-in referential checks

What are JSON Duality Views?

Unifying physical schemas and document trees

The same data exists as a normalized relational table AND as a hierarchical JSON document, at the same time. Read it as a document. Write it as a document. MySQL maps everything to the underlying rows automatically.

One Copy of Data

No duplication. No sync. One source of truth.

Full Read & Write

INSERT, UPDATE, DELETE through the JSON document.

Constraints Enforced

Foreign keys, data types and integrity are kept.

The challenge before MySQL 9.7

Traditional pain points in handling JSON data

1

JSON Assembled in App Code

The app fetched the data, then assembled the JSON itself, field by field.

2

JSON Views Were Read-Only

You could read data as JSON, but writing meant going back to separate updates on every table.

3

Same Data, Two Copies

A JSON copy was stored next to the real columns — now two copies to keep in sync by hand.

How JSON Duality Views Solve the Problem

Eliminating data-handling plumbing

1

No More App-Layer Assembly

MySQL builds and returns the JSON document directly from the view — joins and mapping logic live in one place, defined once.

2

Fully Writable Views

INSERT, UPDATE and DELETE work directly on the JSON document — MySQL translates each operation into the correct row-level changes.

3

Single Source of Truth

Data is stored once, in normalized tables. The JSON view is generated on demand — no duplication, no sync, no drift.

JSON Duality Views – Concurrency & Transaction Control with ETAG

Asserting race conditions in real time

"_metadata.etag" — a hash on every document read from a Duality View, marking its current row state.

Client A

Read Etag: a1b2c3

Writes succeeds → new d4e5f6

← Client A Updates →

MySQL DB Engine

Active Row State Etag

`d4e5f6`

← Client B Updates →

Client B

Read Etag: a1b2c3

Write Rejected × (Etag Mismatch)

1. Both Clients Read

Client A and Client B both read the same row. Both walk away holding etag a1b2c3.

2. Client A Updates

A sends the etag it read — a1b2c3 — and it still matches. The write succeeds and commits, new etag: d4e5f6.

3. Client B Updates

B sends the etag it read — still a1b2c3 — but A's change is already committed as d4e5f6. Mismatch — write rejected.

Optimistic locking pathways

Test Case 1 - Read Only JSON Duality View - Underlying Tables

Defining parent structures

```
mysql> show create table department\G
*************************** 1. row ***************************
       Table: department
Create Table: CREATE TABLE `department` (
  `id` int NOT NULL,
  `name` varchar(100) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.001 sec)
```

Test Case 1 - Read Only JSON Duality View - Underlying Tables

Defining child reference parameters

```
mysql> show create table employee\G
*************************** 1. row ***************************
       Table: employee
Create Table: CREATE TABLE `employee` (
  `id` int NOT NULL,
  `department_id` int NOT NULL,
  `name` varchar(100) NOT NULL,
  `role` varchar(60) NOT NULL,
  `salary` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `department_id` (`department_id`),
  CONSTRAINT `employee_ibfk_1` FOREIGN KEY (`department_id`)
    REFERENCES `department` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.000 sec)
```

Test Case 1 - Read Only JSON Duality View

Creating basic relational trees

department_ro exposes department + employee data as a single JSON document, read-only.

```
mysql> CREATE JSON RELATIONAL DUALITY VIEW `department_ro` AS select
  json_duality_object('_id':`department`.`id`,'department':`department`.`name`,
  'employees':(select json_arrayagg(
    json_duality_object('id':`employee`.`id`,'name':`employee`.`name`,
    'role':`employee`.`role`))
  from `employee` where (`employee`.`department_id` = `department`.`id`)))
  AS `Name_exp_1` from `department`
```

JSON RELATIONAL DUALITY VIEW

The keyword that creates the view. It exposes tables as one JSON document, still linked to the original data.

json_duality_object()

Takes key-value pairs — each key gets its value from a column, or from related rows like 'employees'.

Test Case 2 - Writable JSON Duality View

Authorizing mutation routes

department_dv adds WITH (INSERT, UPDATE, DELETE) at each level to allow writes through the JSON document.

```
mysql> CREATE JSON RELATIONAL DUALITY VIEW `department_dv` AS select
  json_duality_object( WITH (INSERT,UPDATE,DELETE)
  '_id':`department`.`id`,'department':`department`.`name`,
  'employees': (select json_arrayagg(
    json_duality_object( WITH (INSERT,UPDATE,DELETE)
    'id':`employee`.`id`,'name':`employee`.`name`,
    'role':`employee`.`role`,'salary':`employee`.`salary`))
  from `employee` where (`employee`.`department_id` = `department`.`id`)))
  AS `Name_exp_1` from `department`
```

Keyword: WITH (INSERT, UPDATE, DELETE)
Placed inside json_duality_object(...) at a given nesting level, this clause grants write access for that level. It must be specified separately for the department object and again for each employee object — without it, that part of the document is read-only.

SQL definitions

Test Case 3 - Insert via JSON duality view

Loading structural parameters

INSERT — Add a department with employees

```
mysql> INSERT INTO department_dv VALUES ('{
  "_id": 10,
  "department": "Engineering",
  "employees": [
    { "id": 101, "name": "Alice Menon", "role": "Backend Engineer", "salary": 95000 },
    { "id": 102, "name": "Ravi Sharma", "role": "Frontend Engineer", "salary": 88000 },
    { "id": 103, "name": "Priya Nair", "role": "DevOps Engineer", "salary": 91000 }
  ]
}');
Query OK, 4 rows affected (0.016 sec)
Rows affected: 4  Warnings: 0.
```

Test Case 3 - Insert via JSON duality view

Asserting physical updates

MySQL parsed the JSON document and inserted matching rows into BOTH tables in one statement.

```
mysql> select * from department;
+----+-------------+
| id | name        |
+----+-------------+
| 10 | Engineering |
+----+-------------+
1 row in set (0.000 sec)

mysql> select * from employee;
+-----+---------------+--------------+------------------+----------+
| id  | department_id | name         | role             | salary   |
+-----+---------------+--------------+------------------+----------+
| 101 |            10 | Alice Menon  | Backend Engineer | 95000.00 |
| 102 |            10 | Ravi Sharma  | Frontend Engineer | 88000.00 |
| 103 |            10 | Priya Nair   | DevOps Engineer  | 91000.00 |
+-----+---------------+--------------+------------------+----------+
3 rows in set (0.001 sec)
```

Validation outcomes

Test Case 3 - Read back from the JSON duality view

Evaluating output document trees

Querying the read-only duality view returns one JSON document combining both tables, plus _metadata.etag.

```
mysql> select * from department_ro;
+-----------------------------------------------------------------------------------------------------------+
| data                                                                                                      |
+-----------------------------------------------------------------------------------------------------------+
| {"_id": 10, "_metadata": {"etag": "ca920f01c1d7f6005ef7d0e977b55598"},                                     |
|  "employees": [{"id": 101, "name": "Alice Menon", "role": "Backend Engineer"},                            |
|                {"id": 102, "name": "Ravi Sharma", "role": "Frontend Engineer"},                            |
|                {"id": 103, "name": "Priya Nair",  "role": "DevOps Engineer"}],                             |
|  "department": "Engineering"}                                                                             |
+-----------------------------------------------------------------------------------------------------------+
1 row in set (0.003 sec)
```

What is _metadata.etag? A hash that fingerprints the current state of this document — returned on every read. (Reference - Use case 4)

Test Case 4 - Update via JSON duality view

Mutations with absent locking tokens

Sending an update WITHOUT _metadata.etag renames the department and adds a 4th employee — MySQL allows it, but warns about the missing ETAG.

```
mysql> UPDATE department_dv
    -> SET data = '{
    ->   "_id": 10,
    ->   "department": "Engineering-new",
    ->   "employees": [
    ->     { "id": 101, "name": "Alice Menon", "role": "Backend Engineer", "salary": 95000 },
    ->     { "id": 102, "name": "Ravi Sharma", "role": "Frontend Engineer", "salary": 88000 },
    ->     { "id": 103, "name": "Priya Nair", "role": "DevOps Engineer", "salary": 91000 },
    ->     { "id": 104, "name": "Karan Mehta", "role": "QA Engineer", "salary": 78000 }
    ->   ]
    -> }'
    -> WHERE data->>'$._id' = '10';
Query OK, 2 rows affected, 1 warning (0.006 sec)
Rows affected: 2  Warnings: 1.
```

Test Case 4 - Update via JSON duality view

Confirming database state shifts

Verifying the change: the missing-ETAG warning, the renamed department, and the new 4th employee row.

```
mysql> show warnings;
+---------+------+--------------------------------+
| Level   | Code | Message                        |
+---------+------+--------------------------------+
| Warning | 6493 | ETAG missing for updated value |
+---------+------+--------------------------------+
1 row in set (0.000 sec)

mysql> select * from department;
+----+-----------------+
| id | name            |
+----+-----------------+
| 10 | Engineering-new |
+----+-----------------+

mysql> select * from employee;
+-----+---------------+--------------+-------------------+----------+
| 101 |            10 | Alice Menon  | Backend Engineer  | 95000.00 |
| 102 |            10 | Ravi Sharma  | Frontend Engineer | 88000.00 |
| 103 |            10 | Priya Nair   | DevOps Engineer   | 91000.00 |
| 104 |            10 | Karan Mehta  | QA Engineer       | 78000.00 |
+-----+---------------+--------------+-------------------+----------+
4 rows in set (0.000 sec)
```

Test Case 4 - Update via JSON duality view using etag

Enforcing concurrency tokens

Now we fetch and pass _metadata.etag with update and it succeeds with 0 warnings and renames the department.

```
mysql> SELECT data->>'$._metadata.etag' FROM department_dv WHERE data->>'$._id' = '10';
+----------------------------------+
| 9b89c243c7376be11ac6a04c76cfda9   |
+----------------------------------+
1 row in set (0.001 sec)

mysql> UPDATE department_dv SET data = '{
  "_id": 10, "department": "Engineering - Platform",
  "_metadata": { "etag": "9b89c243c7376be11ac6a04c76cfda9" },
  "employees": [
    { "id": 101, "name": "Alice Menon",  "role": "Backend Engineer",  "salary": 95000 },
    { "id": 102, "name": "Ravi Sharma",  "role": "Frontend Engineer", "salary": 88000 },
    { "id": 103, "name": "Priya Nair",   "role": "DevOps Engineer",   "salary": 91000 },
    { "id": 104, "name": "Karan Mehta",  "role": "QA Engineer",       "salary": 78000 }
  ]
}' WHERE data->>'$._id' = '10';
Query OK, 1 row affected (0.008 sec)
Rows affected: 1  Warnings: 0.
```

Test Case 4 - Update via JSON duality view using etag

Tracing state tracking identifiers

The department row is renamed. After every successful write, the document gets a NEW etag — fetch it for the next update.

```
mysql> select * from department;
+----+------------------------+
| id | name                   |
+----+------------------------+
| 10 | Engineering - Platform |
+----+------------------------+
1 row in set (0.001 sec)

mysql> SELECT data->'$._metadata.etag' FROM department_dv WHERE data->>'$._id' = '10';
+--------------------------------------+
| "16b832b69352e5b859541a3567b5062f" |
+--------------------------------------+
1 row in set (0.002 sec)
```

Test Case 4 - Update via JSON duality view - Stale etag rejected

Resolving logical merge conflicts

Now we try to update again, but pass the OLD etag (9b89c2...) instead of the latest one (16b832...).

```
mysql> UPDATE department_dv SET data = '{
  "_id": 10,
  "department": "Engineering - Stale",
  "_metadata": { "etag": "9b89c243c7376be11ac6a04c76cfda9" },
  "employees": [
    { "id": 101, "name": "Alice Menon",  "role": "Backend Engineer",  "salary": 95000 },
    { "id": 102, "name": "Ravi Sharma",  "role": "Frontend Engineer", "salary": 88000 },
    { "id": 103, "name": "Priya Nair",   "role": "DevOps Engineer",   "salary": 91000 },
    { "id": 104, "name": "Karan Mehta",  "role": "QA Engineer",       "salary": 78000 }
  ]
}' WHERE data->>'$._id' = '10';
```

ERROR 6494 (HY000): Cannot update JSON duality view. The ETAG of the document in the database did not match the ETAG "9b89c243c7376be11ac6a04c76cfda9" passed in.

Test Case 4 - Update via JSON duality view - Latest etag

Securing database states

Same update, but now passing the LATEST etag (16b832...) returned by the previous read. The update succeeds.

```
mysql> UPDATE department_dv SET data = '{
  "_id": 10,
  "department": "Engineering - Stale",
  "_metadata": { "etag": "16b832b69352e5b859541a3567b5062f" },
  "employees": [
    { "id": 101, "name": "Alice Menon",  "role": "Backend Engineer",  "salary": 95000 },
    { "id": 102, "name": "Ravi Sharma",  "role": "Frontend Engineer", "salary": 88000 },
    { "id": 103, "name": "Priya Nair",   "role": "DevOps Engineer",   "salary": 91000 },
    { "id": 104, "name": "Karan Mehta",  "role": "QA Engineer",       "salary": 78000 }
  ]
}' WHERE data->>'$._id' = '10';
Query OK, 1 row affected (0.007 sec)
Rows affected: 1  Warnings: 0.
```

Test Case 4 - The ETAG Optimistic control flow

Validating physical record modifications

The department table now reflects the update — the name was changed only after the correct ETAG was supplied.

```
mysql> select * from department;
+----+---------------------+
| id | name                |
+----+---------------------+
| 10 | Engineering - Stale |
+----+---------------------+
1 row in set (0.001 sec)
```

Summary of the ETAG flow:
(1) update without etag → succeeds but Warning 6493
(2) update with current etag → succeeds, 0 warnings, returns a NEW etag
(3) update with a STALE etag → rejected with ERROR 6494
(4) update with the LATEST etag → succeeds.

This is MySQL's optimistic concurrency control.

Test Case 5 - Deletes via JSON Duality View

Tracking state structures before deletions

Before deleting, let's confirm the current rows in both tables and the full JSON document

```
mysql> select * from department;
+----+----------------------+
| id | name                 |
+----+----------------------+
| 10 | Engineering - Stale |
+----+----------------------+

mysql> select * from employee;
+-----+---------------+--------------+-------------------+----------+
| id  | department_id | name         | role              | salary   |
+-----+---------------+--------------+-------------------+----------+
| 101 |            10 | Alice Menon  | Backend Engineer  | 95000.00 |
| 102 |            10 | Ravi Sharma  | Frontend Engineer | 88000.00 |
| 103 |            10 | Priya Nair   | DevOps Engineer   | 91000.00 |
| 104 |            10 | Karan Mehta  | QA Engineer       | 78000.00 |
+-----+---------------+--------------+-------------------+----------+
4 rows in set (0.001 sec)
```

Deletion testing preparation

Test Case 5 - Deletes via JSON Duality View

Asserting transactional states

Reading view department_dv returns the full nested JSON document, including the latest _metadata.etag.

```
mysql> select * from department_dv;
+-----------------------------------------------------------------------------------------+
| data                                                                                    |
+-----------------------------------------------------------------------------------------+
| {"_id": 10, "_metadata": {"etag": "2f28bf14cf2d019eceef9c0ed506213c"},             |
|  "employees": [                                                                         |
|    {"id": 101, "name": "Alice Menon",  "role": "Backend Engineer",  "salary": 95000}, |
|    {"id": 102, "name": "Ravi Sharma",  "role": "Frontend Engineer", "salary": 88000}, |
|    {"id": 103, "name": "Priya Nair",   "role": "DevOps Engineer",   "salary": 91000}, |
|    {"id": 104, "name": "Karan Mehta",  "role": "QA Engineer",       "salary": 78000}  |
|  ], "department": "Engineering - Stale"}                                                |
+-----------------------------------------------------------------------------------------+
1 row in set (0.001 sec)
```

Pre-deletion checks

Test Case 5 - Deletes via JSON Duality View

Asserting cascade behaviors

A single DELETE on the duality view removes the matching rows from BOTH department and employee tables.

```
mysql> DELETE FROM department_dv WHERE data->>'$._id' = '10';
Query OK, 5 rows affected (0.009 sec)

mysql> select * from department;
Empty set (0.000 sec)

mysql> select * from employee;
Empty set (0.000 sec)
```

5 rows affected = 1 department row + 4 employee rows.

Deleting the document deletes data from both tables that matched the condition, in one statement — relational integrity is maintained automatically.

Inspecting Duality Views — Querying INFORMATION_SCHEMA

Reviewing metadata parameters

View Permissions — Insert, Update, or Delete — JSON_DUALITY_VIEWS

```
mysql> SELECT TABLE_NAME, ROOT_TABLE_NAME, ALLOW_INSERT, ALLOW_UPDATE, ALLOW_DELETE
    -> FROM INFORMATION_SCHEMA.JSON_DUALITY_VIEWS WHERE TABLE_SCHEMA = 'json';
+---------------+-----------------+--------------+--------------+--------------+
| TABLE_NAME    | ROOT_TABLE_NAME | ALLOW_INSERT | ALLOW_UPDATE | ALLOW_DELETE |
+---------------+-----------------+--------------+--------------+--------------+
| department_dv | department      |            1 |            1 |            1 |
| department_ro | department      |            0 |            0 |            0 |
+---------------+-----------------+--------------+--------------+--------------+
2 rows in set (0.002 sec)
```

Inspecting Duality Views — Querying INFORMATION_SCHEMA

Asserting column mappings and child relationships

```
-- Mapping JSON Keys to Table Columns — JSON_DUALITY_VIEW_COLUMNS
mysql> SELECT REFERENCED_TABLE_NAME AS tbl, REFERENCED_COLUMN_NAME AS col, JSON_KEY_NAME AS json_key
    -> FROM INFORMATION_SCHEMA.JSON_DUALITY_VIEW_COLUMNS WHERE TABLE_NAME = 'department_dv';
+------------+--------+------------+
| tbl        | col    | json_key   |
+------------+--------+------------+
| department | id     | _id        |
| department | name   | department |
| employee   | id     | id         |
| employee   | name   | name       |
| employee   | role   | role       |
| employee   | salary | salary     |
+------------+--------+------------+

-- How Tables Are Linked — JSON_DUALITY_VIEW_LINKS
mysql> SELECT PARENT_TABLE_NAME, CHILD_TABLE_NAME, PARENT_COLUMN_NAME, CHILD_COLUMN_NAME
    -> FROM INFORMATION_SCHEMA.JSON_DUALITY_VIEW_LINKS WHERE TABLE_NAME = 'department_dv';
+-------------------+------------------+--------------------+-------------------+
| PARENT_TABLE_NAME | CHILD_TABLE_NAME | PARENT_COLUMN_NAME | CHILD_COLUMN_NAME |
+-------------------+------------------+--------------------+-------------------+
| department        | employee         | id                 | department_id     |
+-------------------+------------------+--------------------+-------------------+
1 row in set (0.002 sec)
```

Limitations

Restrictions in using JSON duality views

Permissions Are Per Nesting Level

Each level of the JSON document needs its own write access — skip one, and that level becomes read-only.

Locking Isn't Automatic

Leave out the etag, and the last write simply wins — there's no concurrency protection unless you send it back explicitly.

Every Key Needs a Real Column

No computed or virtual fields — each key has to come from an actual column.

No Caching for the View

Every read of the duality view re-runs its underlying query, live — test it at scale before relying on it.

Limitations metrics

Mydbops
Scaling Databases

Exploring MySQL 9.7

Hypergraph Optimizer

Part 2: Smarter Query Planner

Evaluating query plan paths at scale

MySQL

Hypergraph Optimizer - A Smarter Query Planner

Agenda Outline

- Query Processing - A Quick Primer
- Why a New Optimizer?
- What is the Hypergraph Optimizer?
- Classic VS Hypergraph
- Benchmarks
- How to Enable It?
- When to Use It - and When to Skip It

Query Processing — A Quick Primer

The stages of execution

1

Parse

Is the SQL valid? Syntax parse trees.

2

Resolve

Check tables, columns and accesses.

3

Rewrite

Simplify structures and subqueries.

4

Optimize

Select cheapest plan order.

Hypergraph lives here

5

Execute

Run the plan and stream rows.

Why a New Optimizer? — Shortcomings of Classic

Mydbops
Scaling Databases

Analyzing limits in legacy algorithms

Left-deep chains only

Always builds a single chain: t1→t2→t3→t4. Cannot run two independent sub-joins and merge results.

```
t1 → t2 → t3 → t4  (one long chain)
```

Nested-loop bias

Defaults to nested loops whenever an index exists — even when a hash table + single scan would be far faster.

```
for each row in A: lookup B[index]
    → millions of times
```

ORDER BY is an afterthought

Plans join first, sorts at the end — even when index pre-sorts rows and lets LIMIT short-circuit.

```
JOIN all rows → SORT → LIMIT 10
```

Greedy search on 8+ tables

Exhaustive search is abandoned; classic falls back to heuristics and may miss the best plan.

```
Too many combos → give up → guess
```

No cost caching

Recalculates same sub-join cost multiple times during plan selection — wasted CPU during planning.

```
cost(A+B) computed → then again → then again
```

What is the Hypergraph Optimizer?

A query with 4 tables — as a graph

Tables = Nodes

Every table in your SQL query becomes a node in the graph. It could be orders, customers, products — each is a node.

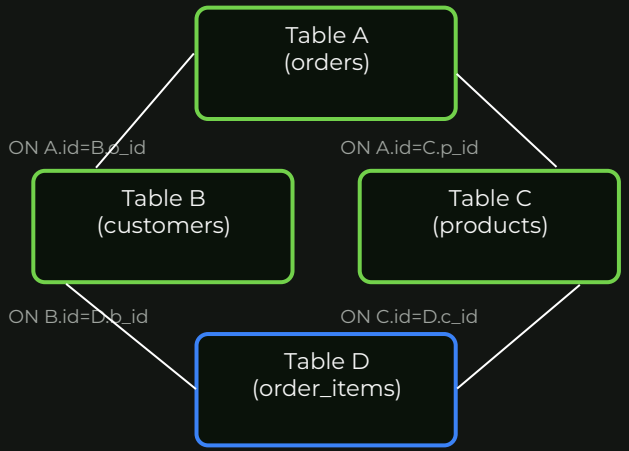
JOIN Conditions = Edges

Every ON clause connecting two tables becomes an edge. The edge carries the join predicate that links them.

Hyperedges

When a predicate spans 3+ tables, it becomes a hyperedge — this is what makes it a hypergraph, not just a graph.

A query with 4 tables — as a graph



Classic vs Hypergraph — A Real-World Analogy

Contrasting path-finding behaviors

Classic Optimizer

"Visits stops in a fixed order, without checking the full map"

Total: 45 km ← Not the shortest!

Hypergraph Optimizer

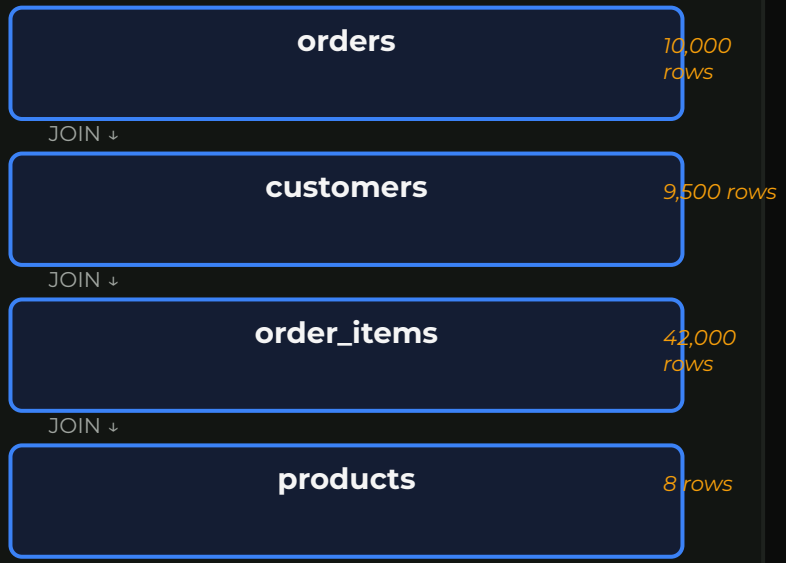
"Sees ALL stops on the map, picks the shortest route"

Via Stop B: 16 km — Shortest!

Classic vs Hypergraph — Plan Shape

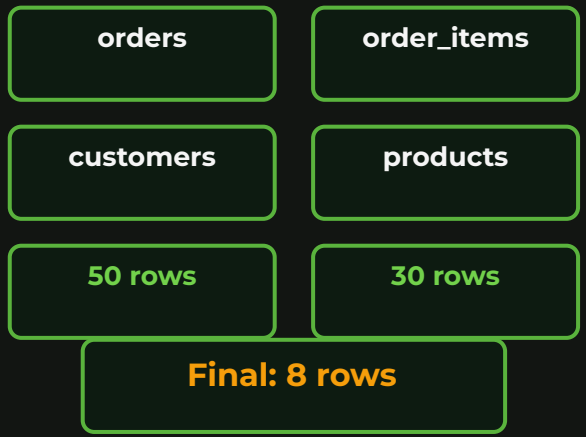
Asserting left-deep chains and bushy configurations

Classic — Left-Deep Chain



Large intermediate results are carried through every join.

Hypergraph — Bushy Plan



Reduces data early before combining results.

Plan Shape: Classic = always left-deep | Hypergraph = bushy plans allowed

Execution plan topologies

Classic vs Hypergraph — Join Algorithm

Evaluating default biases and metric evaluations

Classic — Nested-Loop Join

- Choose nested loop join
- Execution cost grows with more rows
- Classic always prefer nested loop join, even if hash join would be faster.

Join Algorithm: Classic = nested-loop bias

Hypergraph — Hash Join

- Evaluates multiple join strategy
- Prefers hash join instead of nested loop when it reduces cost
- Hypergraph compares both plans and picks the cheaper one.

Join Algorithm: Hypergraph = Real cost comparison

Classic vs Hypergraph — ORDER BY Handling

Filesorts vs early-sort indices

Classic — Process First, Order Later

1. Read the data
2. Process more rows
3. **Sort a target result set**
4. Return required rows

More data sorted than necessary.

ORDER BY: Classic = Sort last

Hypergraph — Order Aware Planning

1. **Consider ordering early**
2. Choose more efficient plan
3. Process fewer rows
4. Return required rows

Hypergraph chooses the plan with ORDER BY in mind from the start.

ORDER BY: Hypergraph = Consider ordering early

Classic vs Hypergraph — Cost Caching

Resolving repetitive planning overheads

Classic — No Cost Caching

3 candidate plans for A x B x C x D — all start with A x B

- ((A x B) x C) x D → cost(A x B) computed
- (A x B) x (D x C) → cost(A x B) computed again
- (A x B) x (C x D) → cost(A x B) computed again

cost(A x B) computed 3x: same answer, worked out 3 separate times. Every plan that starts with A x B re-resolves it from scratch.

Cost Caching: Classic = recalculates repeatedly

Hypergraph — Caches Sub-Plan Costs

Same 3 candidate plans for A x B x C x D

- ((A x B) x C) x D → cost(A x B) computed and cached
- (A x B) x (D x C) → cost(A x B) from cache
- (A x B) x (C x D) → cost(A x B) from cache

cost(A x B) computed 1x: worked out once, looked up twice. The first plan computes A x B once; the rest just look it up.

Cost Caching: Hypergraph = computes once, reuses everywhere

Execution plan costs

How to Enable the Hypergraph Optimizer

Applying session, global, and query-level controls

Not enabled by default in MySQL 9.7 — must be explicitly activated at session, global or per query level.

1. Enable at Session Level

```
SET optimizer_switch='hypergraph_optimizer=on';
```

2. Enable at Global Level

```
SET global optimizer_switch='hypergraph_optimizer=on';
```

3. Enable at Statement / Query Level (Safest Approach)

```
SELECT /*+ SET_VAR(optimizer_switch='hypergraph_optimizer=on') */
column1, column2 FROM table1 JOIN table2 ON table1.id = table2.id
WHERE condition = 'value';
```

Activation commands

Putting Theory to the Test

Now let's look at the actual benchmark numbers.

> For every benchmark, we used the exact same dataset and the exact same query — the only thing that changed was the optimizer switch.

We tested three scenarios, each stressing a specific weakness of the classic optimizer:

1	**Benchmark 1**	Stresses join algorithm choice
2	**Benchmark 2**	Stresses ORDER BY handling
3	**Benchmark 3**	Stresses plan shape — a larger six-table join, left-deep chains vs bushy plans

Benchmark 1 — Nested Loop vs Hash Join (4 tables)

customers (100,000 rows) → orders (1,000,000 rows) → order_items (5,000,000 rows) → products (100,000 rows)

```
SELECT c.city, p.product_name, COUNT(*) AS cnt
FROM customers c JOIN orders o ON c.customer_id=o.customer_id
JOIN order_items oi ON o.order_id=oi.order_id JOIN products p ON oi.product_id=p.product_id
GROUP BY c.city, p.product_name;
```

CLASSIC PLAN • 38.13s

```
-> Nested loop inner join (cost=3.73e+6 rows=5.14e+6)
  (actual time=0.165..29885 rows=5e+6 loops=1)
  -> Covering index scan on c using idx_city
    (actual time=0.0776..81 rows=100000 loops=1)
  -> Covering index lookup on o using idx_customer
    (actual time=0.00726..0.0101 rows=10 loops=100000)
  -> Index lookup on oi using idx_order
    (actual time=0.0221..0.0241 rows=5 loops=1e+6)
  -> Single-row index lookup on p using PRIMARY
    (actual time=313e-6..354e-6 rows=1 loops=5e+6)
-> Aggregate using temporary table
  (actual time=38005..38005 rows=100000 loops=1)
```

- Uses Nested Loop join as its strategy
- Re-searches the other table, row by row
- Cost grows with every row — repetition is expensive

HYPERGRAPH PLAN • 22.16s

```
-> Inner hash join (oi.product_id = p.product_id)
  (actual time=724..14734 rows=5e+6 loops=1)
  -> Inner hash join (c.customer_id = o.customer_id)
    (actual time=654..10436 rows=5e+6 loops=1)
    -> Inner hash join (o.order_id = oi.order_id)
      (actual time=594..6782 rows=5e+6 loops=1)
      -> Index scan on oi using PRIMARY
        (actual time=1.49..2221 rows=5e+6 loops=1)
      -> Hash -> idx_customer scan on o (1M rows)
      -> Hash -> idx_city scan on c (100,000 rows)
      -> Hash -> PRIMARY scan on p (100,000 rows)
-> Aggregate using temporary table
  (actual time=21946..21946)
```

- Uses Hash Join instead of Nested Loop
- Scans each table once
- No repeated loops — matched in memory

SUMMARY: Hash Join scans each table once and matches in memory, instead of repeating lookups row by row like Nested Loop.

42% faster

Performance stats

Benchmark 2 — Late Sort vs Early Sort (Sort + Limit)

customers (100,000 rows) → orders (1,000,000 rows) → order_items (5,000,000 rows)

```
SELECT c.customer_name, o.order_id, o.order_date, SUM(oi.quantity) qty
FROM customers c JOIN orders o ON c.customer_id=o.customer_id
JOIN order_items oi ON oi.order_id=o.order_id
GROUP BY c.customer_name, o.order_id, o.order_date ORDER BY o.order_date DESC LIMIT 50;
```

CLASSIC PLAN • 144.4s (2 min 24s)

```
-> Limit: 50 row(s)
  (actual time=144380..144380 rows=50 loops=1)
  -> Sort: o.order_date DESC, limit input to 50 row(s)
    (actual time=144380..144380 rows=50 loops=1)
    -> Table scan on <temporary> (rows=1e+6)
      -> Aggregate using temporary table
        (actual time=140706..140706 rows=1e+6 loops=1)
        -> Nested loop inner join
          -> Table scan on o (1,000,000 rows)
            (actual time=2.4..775 rows=1e+6 loops=1)
          -> Single-row PK lookup on c (1M loops)
          -> idx_order lookup on oi (rows=5 loops=1e+6)
```

- Finishes the full join first
- Sorts and limits last
- Wastes work on rows it later discards

HYPERGRAPH PLAN • 3.01s

```
-> Limit: 50 row(s)
  (actual time=2833..2997 rows=50 loops=1)
  -> Group aggregate: sum(oi.quantity)
    (actual time=2833..2997 rows=50 loops=1)
    -> Nested loop inner join (rows=251)
      -> Sort: o.order_date DESC, ... (rows=51)
        (actual time=2821..2821 rows=51 loops=1)
        -> Inner hash join (c.customer_id=o.customer_id)
          (actual time=160..1435 rows=1e+6 loops=1)
          -> Index scan on o using PRIMARY (1M rows)
            (actual time=18.1..654 rows=1e+6 loops=1)
          -> idx_order lookup on oi (rows=4.92 loops=51)
```

- Sorts and limits early
- Trims rows before the costly join
- Less work reaches the expensive part

SUMMARY: Hypergraph pushes Sort and Limit before the expensive join, so only a handful of rows reach the costly part of the plan.

48x faster

Benchmark queries

Benchmark 3 — Left-Deep Chain vs Bushy Plan (6 tables)

customers(100K) → orders(1M) → order_items(5M) → products(100K) → suppliers(1K), and orders → shipments(1M)

```
SELECT c.city, p.product_name, s.supplier_name, sh.shipment_status, COUNT(*) AS cnt
FROM customers c JOIN orders o ON c.customer_id=o.customer_id JOIN order_items oi ON o.order_id=oi.order_id
JOIN products p ON oi.product_id=p.product_id JOIN suppliers s ON p.supplier_id=s.supplier_id
JOIN shipments sh ON sh.order_id=o.order_id GROUP BY c.city, p.product_name, s.supplier_name, sh.shipment_status;
```

CLASSIC PLAN • 139.5s (2m 19s)

```
-> Table scan on <temporary> (rows=300000 loops=1)
  -> Aggregate using temporary table
    (rows=299999 loops=1)
    -> Nested loop inner join
      (cost=5.89e+6, rows=5e+6 loops=1)
      -> Nested loop inner join
        (cost=4.09e+6, rows=5e+6 loops=1)
        -> Nested loop inner join
          (cost=2.29e+6, rows=5e+6 loops=1)
          -> Nested loop inner join
            (cost=492466, rows=1e+6 loops=1)
            -> Nested loop inner join
              -> idx_city(c)+idx_customer(o)  (10 loops=1e5)
              -> idx_order lookup on sh (rows=1 loops=1e6)
            -> Filter -> idx_order lookup on oi (5 loops=1e6)
          -> Filter -> PK lookup on p (rows=1 loops=5e6)
        -> PK lookup on s (rows=1 loops=5e6)
```

- Chains all tables into one long path
- Same tables get re-probed repeatedly
- No way to split the work

HYPERGRAPH PLAN • 54.7s

```
-> Group aggregate: count(0)  (rows=300000 loops=1)
  -> Sort: city, product, supplier, status
    (rows=5e+6 loops=1)
    -> Inner hash join (oi.product_id=p.product_id)
      (rows=5e+6 loops=1)
      -> Inner hash join (o.order_id=oi.order_id)
        (rows=5e+6 loops=1)
        -> Inner hash join (c.customer_id=o.customer_id)
          (rows=1e+6 loops=1)
          -> Inner hash join (o.order_id=sh.order_id)
            (rows=1e+6 loops=1)
            -> Index scan on sh using PRIMARY (1e+6)
            -> Hash -> idx_customer scan on o (1e+6)
            -> Hash -> idx_city scan on c (100000)
          -> Hash -> Index scan on oi PRIMARY (5e+6)
        -> Hash -> hash join (p.supplier_id=s.supplier_id)
```

- Splits tables into independent branches
- Solves each branch separately
- Merges branches once at the end

SUMMARY: Hypergraph solves two smaller branches independently and merges them once, instead of forcing every table through one long chain like Classic.

61% faster

Performance stats

When to Use It — and When to Skip It

Suitability assessments

Use Hypergraph

Joining 4+ Tables: More tables means more possible join orders — it's built to search through all of them for the fastest one.

ORDER BY With LIMIT: It can read rows already in index order, skipping a full sort entirely.

Analytics & Reporting Queries: It builds smarter join plans that produce far fewer rows along the way.

Classic May Be Fine

Simple Lookups by Primary Key: Fetching one row is so fast that planning it can take longer than running it.

Queries on a Single Table: There's no join to optimize, so it has nothing extra to offer.

Switching Without Testing First: Some queries may get slower under the new optimizer — always confirm with EXPLAIN ANALYZE before you switch.

Key Takeaways

Summary review

JSON Duality Views

- Works as both relational rows and JSON documents
- Read & write JSON directly — no app-side assembly
- ETag provides concurrency control to prevent lost updates

Hypergraph Optimizer

- Smarter join ordering for queries with 4+ table joins
- Great fit for analytical and reporting queries
- Speeds up queries with heavy sorting (ORDER BY + LIMIT)

Your single takeaway — paste this, run your slowest query:

```
SET SESSION optimizer_switch='hypergraph_optimizer=on';
EXPLAIN ANALYZE <your query here>;
```

Session-scoped — only affects your current connection, and resets when you disconnect.

Mydbops
Scaling Databases

Thank You

Mydbops MyWebinar 52

Scaling Databases, Securing Workflows.

MySQL