

ORACLE®



## Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



# MySQL Optimizer: What's New in 8.0

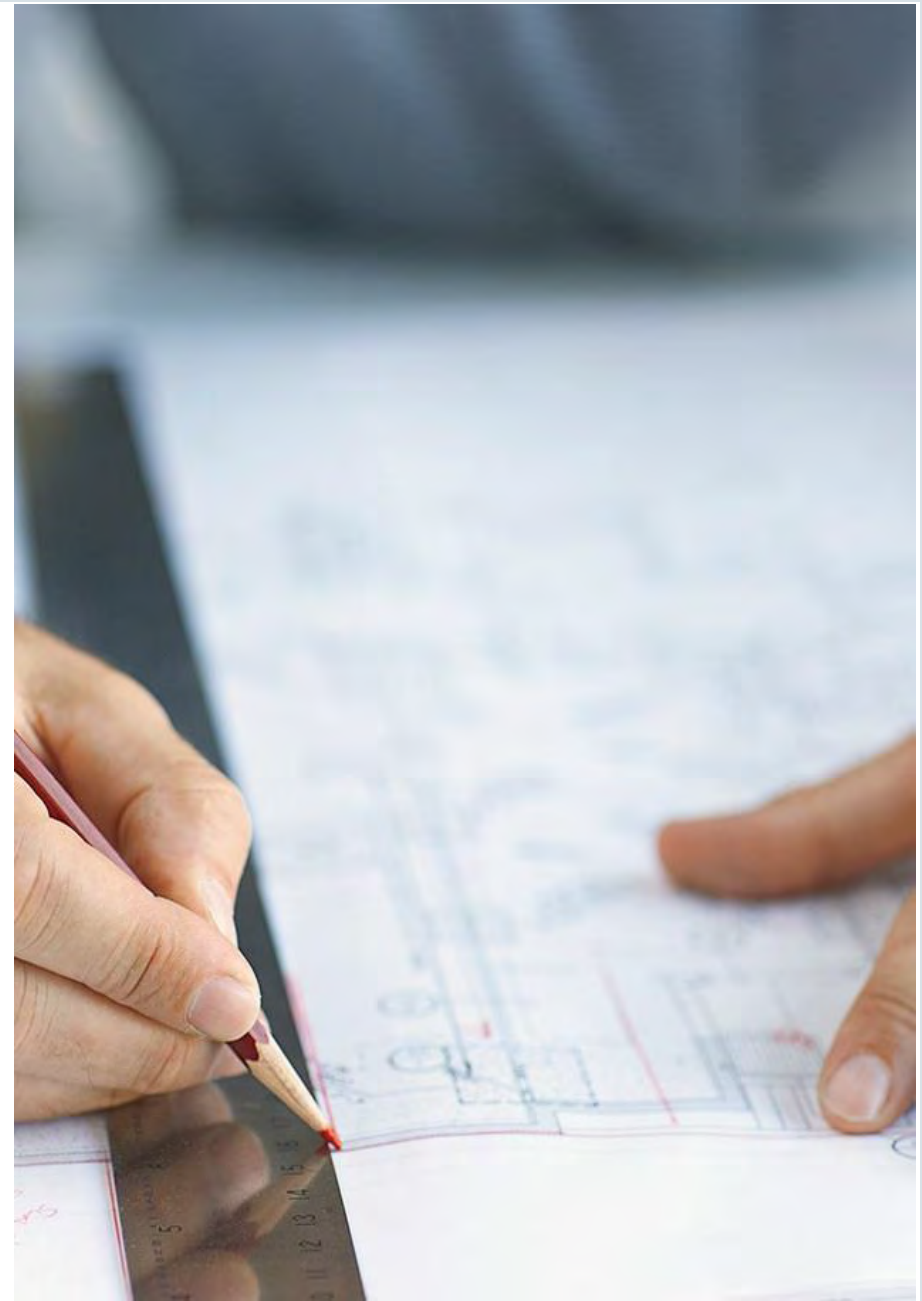
Manyi Lu  
Director

Øystein Grøvlen  
Senior Principal Software Engineer

MySQL Optimizer Team, Oracle  
October, 2017

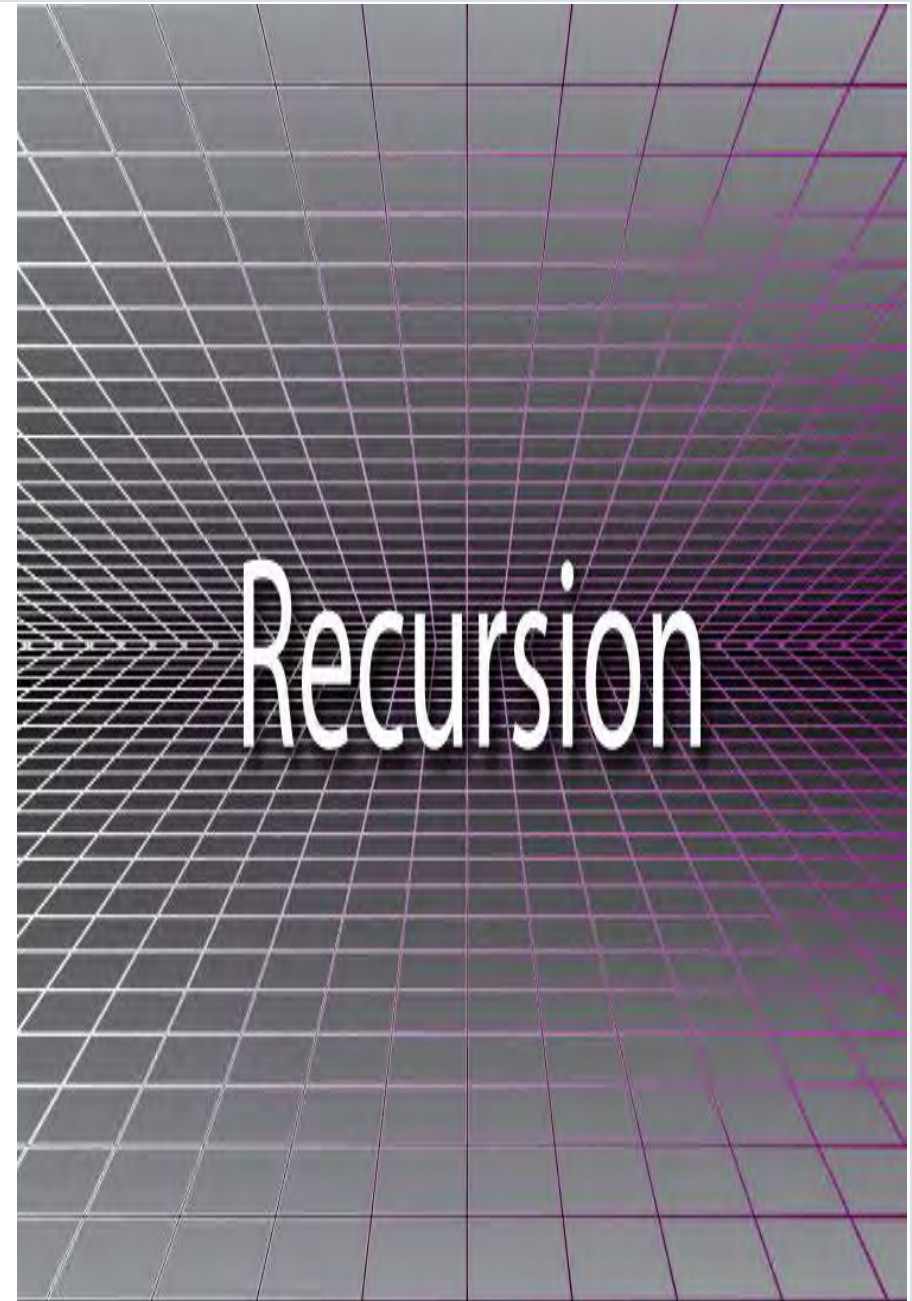
# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Common Table Expression

## Alternative to derived table

- A derived table is a subquery in the FROM clause

```
SELECT ... FROM (subquery) AS derived, t1 ...
```

- Common Table Expression (CTE) is just like a derived table, but its declaration is put before the query block instead of in FROM clause

```
WITH derived AS (subquery)  
SELECT ... FROM derived, t1 ...
```

- A CTE may precede SELECT/UPDATE/DELETE including sub-queries

```
WITH derived AS (subquery)  
DELETE FROM t1 WHERE t1.a IN (SELECT b FROM derived);
```

# Common Table Expression vs Derived Table

Better readability

Can be referenced multiple times

Can refer to other CTEs

Improved performance

# Better Readability

- Derived table:

```
SELECT ...  
FROM t1 LEFT JOIN ((SELECT ... FROM ...) AS dt JOIN t2 ON ...) ON ...
```

- CTE:

```
WITH dt AS (SELECT ... FROM ...)  
SELECT ...  
FROM t1 LEFT JOIN (dt JOIN t2 ON ...) ON ...
```



# Can Be Referenced Multiple Times

- Derived table can not be referenced twice:

```
SELECT ...  
FROM (SELECT a, b, SUM(c) s FROM t1 GROUP BY a, b) AS d1  
JOIN (SELECT a, b, SUM(c) s FROM t1 GROUP BY a, b) AS d2 ON d1.b = d2.a;
```

- CTE can:

```
WITH d AS (SELECT a, b, SUM(c) s FROM t1 GROUP BY a, b)  
SELECT ... FROM d AS d1 JOIN d AS d2 ON d1.b = d2.a;
```

- Better performance with materialization:
  - Multiple references only materialized once
  - Derived tables and views will be materialized once per reference.

# Can Refer to Other CTEs

- Derived tables can not refer to other derived tables:

```
SELECT ...  
FROM (SELECT ... FROM ...) AS d1, (SELECT ... FROM d1 ...) AS d2 ...
```

ERROR: 1146 (42S02): Table 'db.d1' doesn't exist

- CTEs can refer other CTEs:

```
WITH d1 AS (SELECT ... FROM ...),  
      d2 AS (SELECT ... FROM d1 ...)  
SELECT  
FROM d1, d2 ...
```

# Recursive CTE

```
WITH RECURSIVE cte AS
( SELECT ... FROM table_name      /* "seed" SELECT */
  UNION [DISTINCT|ALL]
  SELECT ... FROM cte, table_name ) /* "recursive" SELECT */
SELECT ... FROM cte;
```



Recursion

- A recursive CTE refers to itself in a subquery
- The “seed” SELECT is executed once to create the initial data subset, the recursive SELECT is repeatedly executed to return subsets of.
- Recursion stops when an iteration does not generate any new rows
  - To limit recursion, set `cte_max_recursion_depth`
- Useful to dig in hierarchies (parent/child, part/subpart)

# Recursive CTE

## A simple example

Print 1 to 10 :

```
WITH RECURSIVE qn AS  
  ( SELECT 1 AS a  
    UNION ALL  
    SELECT 1+a FROM qn WHERE a<10  
  )  
SELECT * FROM qn;
```

```
a  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

# Hierarchy Traversal

## Employee database

```
CREATE TABLE employees (  
  id INT PRIMARY KEY,  
  name VARCHAR(100),  
  manager_id INT,  
  FOREIGN KEY (manager_id)  
  REFERENCES employees(id) );
```

```
INSERT INTO employees VALUES  
(333, "Yasmina", NULL), # CEO  
(198, "John", 333), # John reports to 333  
(692, "Tarek", 333),  
(29, "Pedro", 198),  
(4610, "Sarah", 29),  
(72, "Pierre", 29),  
(123, "Adil", 692);
```

# Hierarchy Traversal

## List reporting chain

```

WITH RECURSIVE
emp_ext (id, name, path) AS (
  SELECT id, name, CAST(id AS CHAR(200))
  FROM employees
  WHERE manager_id IS NULL
UNION ALL
  SELECT s.id, s.name,
         CONCAT(m.path, ",", s.id)
  FROM emp_ext m JOIN employees s
  ON m.id=s.manager_id )
SELECT * FROM emp_ext ORDER BY path;

```

id	name	path
333	Yasmina	333
198	John	333,198
692	Tarek	333,692
29	Pedro	333,198,29
123	Adil	333,692,123
4610	Sarah	333,198,29,4610
72	Pierre	333,198,29,72

# Hierarchy Traversal

## List reporting chain

```

WITH RECURSIVE
emp_ext (id, name, path) AS (
  SELECT id, name, CAST(id AS CHAR(200))
  FROM employees
  WHERE manager_id IS NULL
UNION ALL
  SELECT s.id, s.name,
         CONCAT(m.path, ",", s.id)
  FROM emp_ext m JOIN employees s
         ON m.id=s.manager_id )
SELECT * FROM emp_ext ORDER BY path;

```

id	name	path
333	Yasmina	333
198	John	333,198
29	Pedro	333,198,29
4610	Sarah	333,198,29,4610
72	Pierre	333,198,29,72
692	Tarek	333,692
123	Adil	333,692,123

# Program Agenda

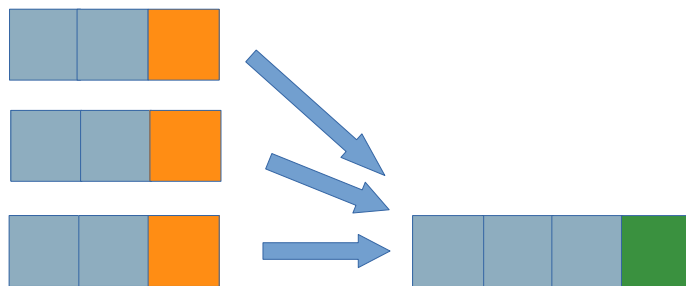
- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



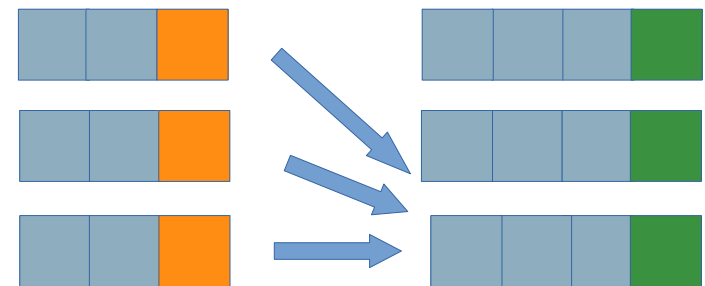


# Window Functions: What Are They?

- A window function performs a calculation across a set of rows that are related to the current row, similar to an aggregate function.
- But unlike aggregate functions, a window function does not cause rows to become grouped into a single output row.
- Window functions can access values of other rows “in the vicinity” of the current row



Aggregate function



Window function

# Window Function Example

Sum up total salary for each department:

```
SELECT name, dept_id, salary,
       SUM(salary) OVER (PARTITION BY
       dept_id) AS dept_total
FROM employee
ORDER BY dept_id, name;
```

The **OVER** keyword signals a window function

**PARTITION** == disjoint set of rows in result set

name	dept_id	salary	dept_total
Newt	NULL	75000	75000
Dag	10	NULL	370000
Ed	10	100000	370000
Fred	10	60000	370000
Jon	10	60000	370000
Michael	10	70000	370000
Newt	10	80000	370000
Lebedev	20	65000	130000
Pete	20	65000	130000
Jeff	30	300000	370000
Will	30	70000	370000

# Window Function Example: Frames

name	dept_id	salary	total
Newt	NULL	75000	75000
Dag	10	NULL	NULL
Ed	10	100000	100000
Fred	10	60000	160000
Jon	10	60000	220000
Michael	10	70000	190000
Newt	10	80000	210000
Lebedev	20	65000	65000
Pete	20	65000	130000
Jeff	30	300000	300000
Will	30	70000	370000

```

SELECT name, dept_id, salary,
       SUM(salary)
       OVER (PARTITION BY dept_id
            ORDER BY name
            ROWS 2 PRECEDING) total
FROM employee
ORDER BY dept_id, name;

```

*moving window frame:*  
SUM(salary) ...  
ROWS 2 PRECEDING

ORDER BY name  
within each partition

A frame is a subset of a  
partition

# Window Function Example: Frames

name	dept_id	salary	total
Newt	NULL	75000	75000
Dag	10	NULL	NULL
Ed	10	100000	100000
Fred	10	60000	160000
Jon	10	60000	220000
Michael	10	70000	190000
Newt	10	80000	210000
Lebedev	20	65000	65000
Pete	20	65000	130000
Jeff	30	300000	300000
Will	30	70000	370000

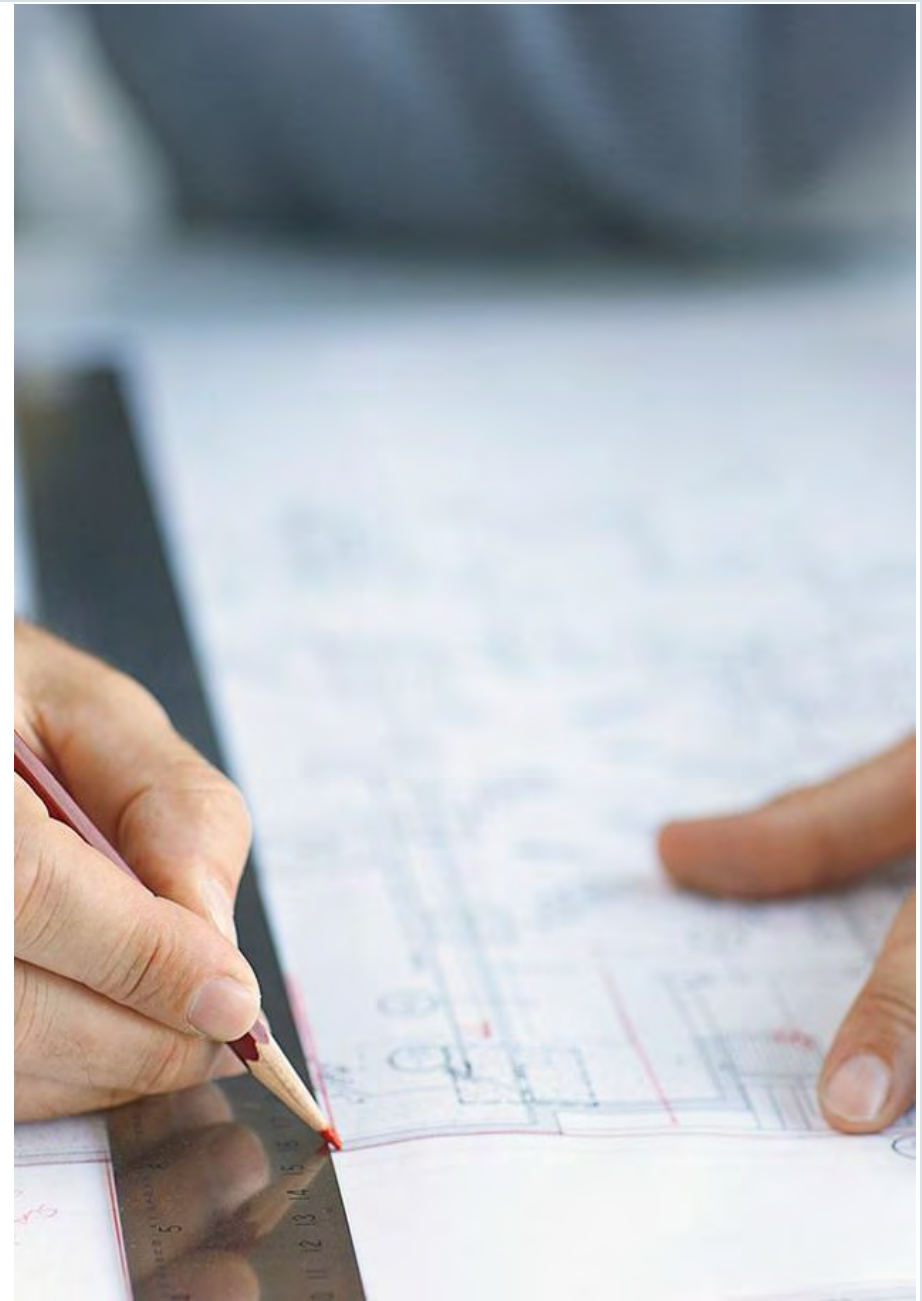
```

SELECT name, dept_id, salary,
       SUM(salary)
       OVER (PARTITION BY dept_id
            ORDER BY name
            ROWS 2 PRECEDING) total
FROM employee
ORDER BY dept_id, name;

```

# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Improved UTF-8 Support in MySQL 8.0

- Support for the latest Unicode 9.0
- utf8mb4 made default character set!
  - utf8mb4\_0900\_ai\_ci default collation
- Accent and case sensitive collations
  - Including 20+ language specific collations 👍
  - Now also Japanese and Russian
- Significantly improved performance





# What Is in MySQL 5.7 and Earlier Versions?

- Default charset is “latin1” and default collation is “latin1\_swedish\_ci”
- utf8 = utf8mb3: support BMP only
- utf8mb4 character set:
  - Only accent and case insensitive collations
  - Default collation is utf8mb4\_general\_ci, compares all characters beyond BMP, e.g. emojis, to be equal
  - 20+ language specific collations
  - Recommend to use: **utf8mb4\_unicode\_520\_ci**

# New Default Character Set

- No changes to existing tables
- Only has effect on new tables/schemas where character set is not explicitly defined.
- Separating character set/collation change from server upgrade
  - Upgrade first, change charset/collation afterwards
- Recommend users to not mixing collations
  - Error “Illegal mix of collations”
  - Slower query because index can no longer be used



# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# GIS

- Geography Support
  - longitude/latitude
- Spatial Reference Systems (SRS) Support
  - SRID 4326 = WGS 84 (“GPS coordinates”)
- Information Schema views
  - ST\_GEOMETRY\_COLUMNS
- Standard compliant axis ordering : longitude-latitude
- Example functions :
  - ST\_Distance( g1, g2)
  - ST\_SwapXY(g)
  - ST\_SRID(g [, new\_srid\_val])



# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# SELECT ... FOR UPDATE SKIP LOCKED

- Common problem:
  - Hot row contention, multiple worker threads accessing the same rows
- Solution 1:
  - Only read rows that are not locked
  - InnoDB skips a locked row, and the next one goes to the result set
- Example:
  - Booking system: Skip orders that are pending

```
START TRANSACTION;  
SELECT * FROM seats WHERE seat_no BETWEEN 2 AND 3 AND booked = 'NO'  
FOR UPDATE SKIP LOCKED;
```

# SELECT... FOR UPDATE NOWAIT

- Common problem:
  - Hot row contention, multiple worker threads accessing the same rows
- Solution 2:
  - If any of the rows are already locked, the statement should fail immediately
  - Without NOWAIT, have to wait for innodb lock wait timeout (default: 50 sec) while trying to acquire lock
- Usage:

```
START TRANSACTION;  
SELECT * FROM seats WHERE seat_no BETWEEN 2 AND 3 AND booked = 'NO'  
FOR UPDATE NOWAIT;  
ERROR 3572 (HY000): Statement aborted because lock(s) could not be acquired ...
```

# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# JSON Aggregation

Combine JSON documents in multiple rows into a JSON array

```
CREATE TABLE t1 (id INT, grp INT,  
jsoncol JSON);
```

```
INSERT INTO t1 VALUES (1, 1,  
'{"key1":"value1","key2":"value2"}');
```

```
INSERT INTO t1 VALUES (2, 1,  
'{"keyA":"valueA","keyB":"valueB"}');
```

```
INSERT INTO t1 VALUES (3, 2,  
'{"keyX":"valueX","keyY":"valueY"}');
```

```
SELECT JSON_ARRAYAGG(jsoncol)  
FROM t1;
```

```
[{"key1":"value1","key2":"value2"},  
{"keyA":"valueA","keyB":"valueB"},  
{"keyX":"valueX","keyY":"valueY"}]
```

# JSON Aggregation

Combine JSON documents in multiple rows into a JSON object

```
CREATE TABLE t1 (id INT, grp INT,  
jsoncol JSON);  
  
INSERT INTO t1 VALUES (1, 1,  
'{"key1":"value1","key2":"value2"}');  
  
INSERT INTO t1 VALUES (2, 1,  
'{"keyA":"valueA","keyB":"valueB"}');  
  
INSERT INTO t1 VALUES (3, 2,  
'{"keyX":"valueX","keyY":"valueY"}');
```

```
SELECT grp, JSON_OBJECTAGG(id, jsoncol)  
FROM t1  
GROUP BY grp;
```

```
1 | {"1":{"key1":"value1","key2":"value2"},  
  "2":{"keyA":"valueA","keyB":"valueB"}}  
2 | {"3":{"keyX":"valueX","keyY":"valueY"}}
```



# JSON\_TABLE

## Convert JSON documents to relational tables

- Table *t1* has a column *json\_col* with content like this:

```
{ "people": [
  { "name": "John Smith", "address": "780 Mission St, San Francisco, CA 94103"},
  { "name": "Sally Brown", "address": "75 37th Ave S, St Cloud, MN 94103"},
  { "name": "Paul Johnson", "address": "1262 Roosevelt Trail, Raymond, ME 04071"},
  ... ] }
```

- Convert JSON column into a table with 2 columns:

```
SELECT people.* FROM t1, JSON_TABLE(json_col, '$.people[*]' COLUMNS (
  name VARCHAR(40) PATH '$.name',
  address VARCHAR(100) PATH '$.address')) people
WHERE people.address LIKE '%San Francisco%';
```

# JSON\_TABLE – Nested Arrays

```
[
  { "father":"John", "mother":"Mary",
    "marriage_date":"2003-12-05",
    "children": [
      { "name":"Eric", "age":12 },
      { "name":"Beth", "age":10 } ] },
  { "father":"Paul", "mother":"Laura",
    "children": [
      { "name":"Sarah", "age":9},
      { "name":"Noah", "age":3},
      { "name":"Peter", "age":1} ] }
]
```

id	father	married	child_id	child	age
1	John	1	1	Eric	12
1	John	1	2	Beth	10
2	Paul	0	1	Sarah	9
2	Paul	0	2	Noah	3
2	Paul	0	3	Peter	1

# JSON\_TABLE – Nested Arrays

```
JSON_TABLE (families, '$[*]' COLUMNS (
  id FOR ORDINALITY,
  father VARCHAR(30) PATH '$.father',
  married INTEGER EXISTS PATH
    '$.marriage_date',
  NESTED PATH '$.children[*]' COLUMNS (
    child_id FOR ORDINALITY,
    child VARCHAR(30) PATH '$.name',
    age INTEGER PATH '$.age' ) )
```

id	father	married	child_id	child	age
1	John	1	1	Eric	12
1	John	1	2	Beth	10
2	Paul	0	1	Sarah	9
2	Paul	0	2	Noah	3
2	Paul	0	3	Peter	1

# JSON\_TABLE

SQL aggregation on JSON data

```
SELECT father, COUNT(*) "#children", AVG(age) "age average"
FROM t, JSON_TABLE (families, '$[*]' COLUMNS (
    id FOR ORDINALITY,
    father VARCHAR(30) PATH '$.father',
    NESTED PATH '$.children[*]' COLUMNS (age INTEGER PATH '$.age' ) ) AS fam
GROUP BY id, father;
```

father	#children	age average
John	2	11.0000
Paul	3	4.3333

# JSON Utility Functions

- `JSON_PRETTY(json_value)`
  - Pretty-print the JSON value
- `JSON_STORAGE_SIZE(json_value)`
  - The number of bytes used to store the binary representation of a JSON document
- `JSON_STORAGE_FREE(json_value)`
  - The number of bytes in its binary representation that is current not used.
  - The binary representation may have unused space after a JSON column was updated in place using `JSON_SET()`

# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Invisible Index

- Index is maintained by the SE, but ignored by the Optimizer
- Primary key cannot be INVISIBLE
- Use case: Check for performance drop BEFORE dropping an index

```
ALTER TABLE t1 ALTER INDEX idx INVISIBLE;
mysql> SHOW INDEXES FROM t1;
```

Table	Key_name	Column_name	Visible
t1	idx	a	NO

- To see an invisible index: **set optimizer\_switch='use\_invisible\_indexes=on';**

# Descending Index

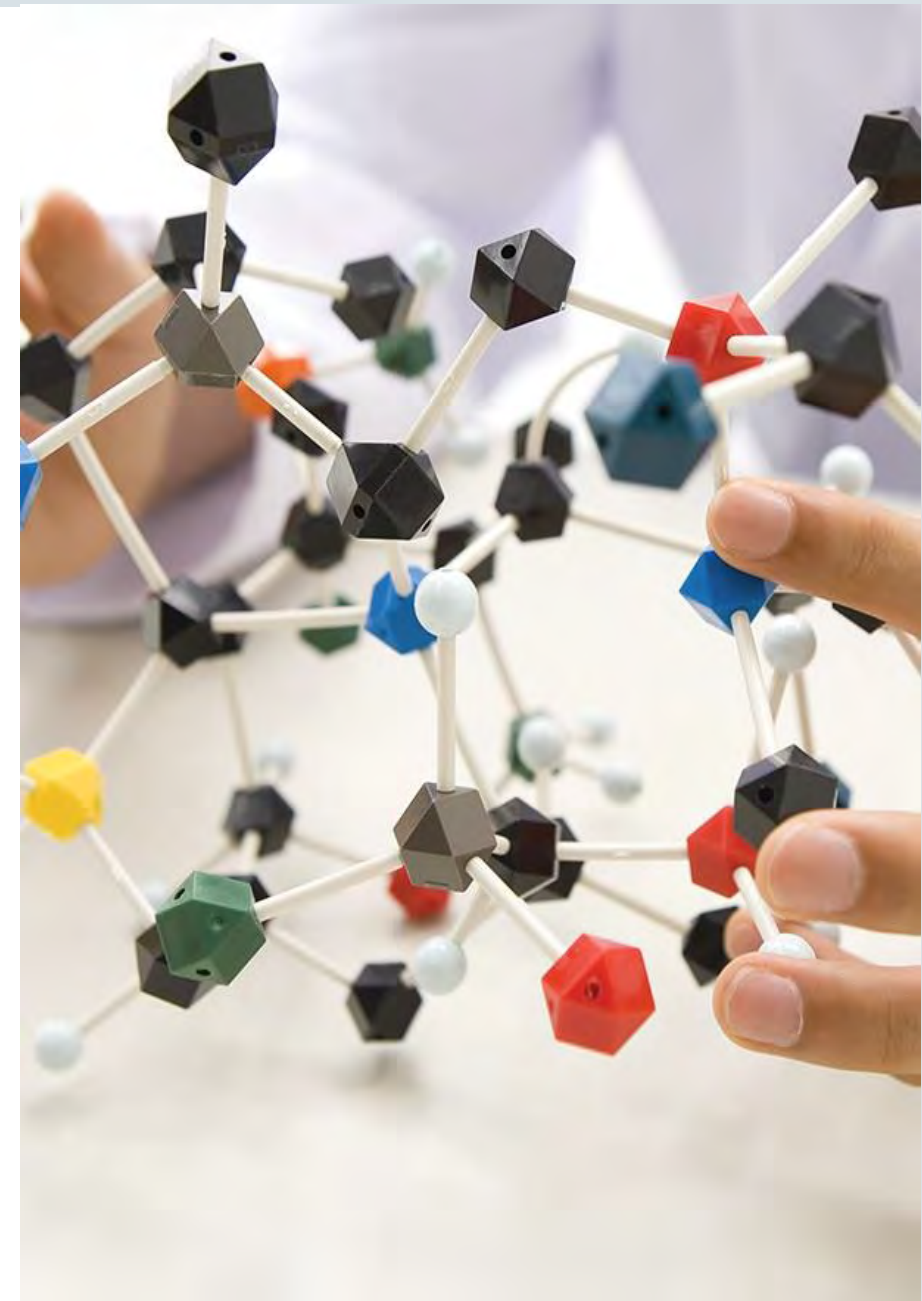
```
CREATE TABLE t1 (  
  a INT,  
  b INT,  
  INDEX a_b (a DESC, b ASC)  
);
```

- In 5.7: Index in ascending order is created, server scans it backwards
- In 8.0: Index in descending order is created, server scans it forwards
- Works on B-tree indexes only
- Benefits:
  - Use indexes instead of filesort for ORDER BY clause with ASC/DESC sort key
  - Forward index scan is slightly faster than backward index scan



# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Motivation for Improving the MySQL Cost Model

- Produce more correct cost estimates
  - Better decisions by the optimizer should improve performance
- Adapt to new hardware architectures
  - SSD, larger memories, caches
- More maintainable cost model implementation
  - Avoid hard coded “cost constants”
  - Refactoring of existing cost model code
- Configurable and tunable
- Make more of the optimizer cost-based

A diagram consisting of five red arrows pointing from the list items on the left to a red-bordered rounded rectangle on the right. The rectangle contains the text "Faster queries" in red. The arrows originate from the following list items: "Better decisions by the optimizer should improve performance", "SSD, larger memories, caches", "Avoid hard coded 'cost constants'", "Configurable and tunable", and "Make more of the optimizer cost-based".

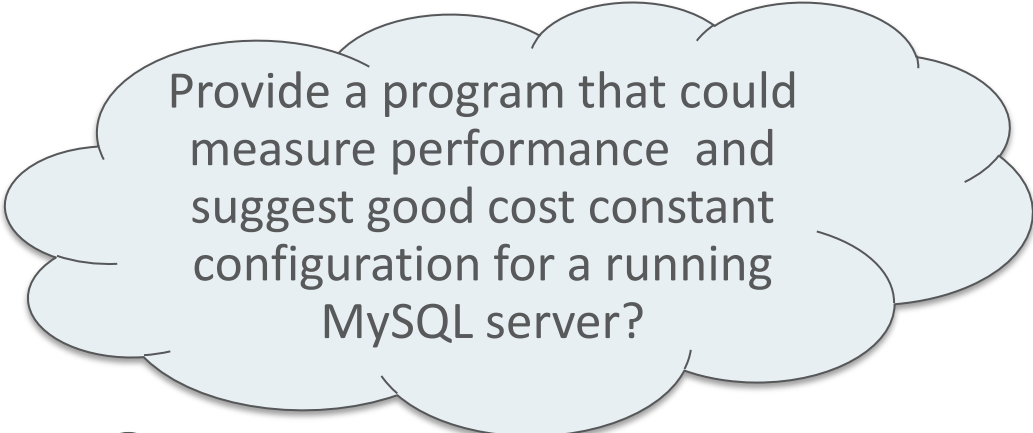
Faster queries

# New Storage Technologies

- Time to do a table scan of 10 million records:

Memory	5 s
SSD	20 - 146 s
Hard disk	32 - 1465 s

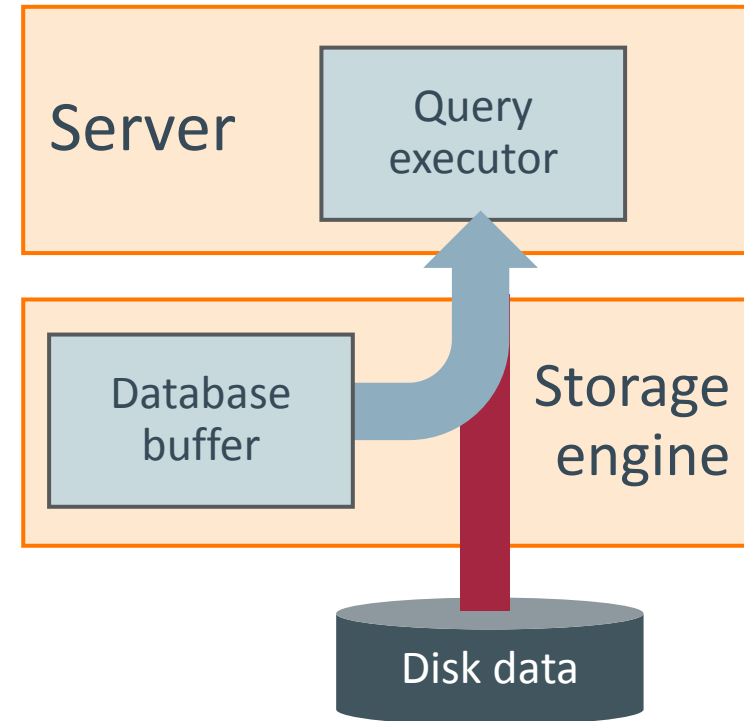
- Adjust cost model to support different storage technologies
- Provide configurable cost constants for different storage technologies



Provide a program that could measure performance and suggest good cost constant configuration for a running MySQL server?

# Memory Buffer Aware Cost Estimates

- Storage engines:
  - Estimate for how much of data and indexes are in a memory buffer
  - Estimate for hit rate for memory buffer
- Optimizer cost model:
  - Take into account whether data is already in memory or need to be read from disk



# MySQL 8.0: Disk vs Memory Access

- New defaults for const constants:

Cost	MySQL 5.7	MySQL 8.0
Read a random disk page	1.0	1.0
Read a data page from memory buffer	1.0	0.25
Evaluate query condition	0.2	0.1
Compare keys/rows	0.1	0.05

- InnoDB reports for each table/index percentage of data cached in buffer pool
- Note: Query plan may change between executions



# DBT-3 Query 8

## National Market Share Query

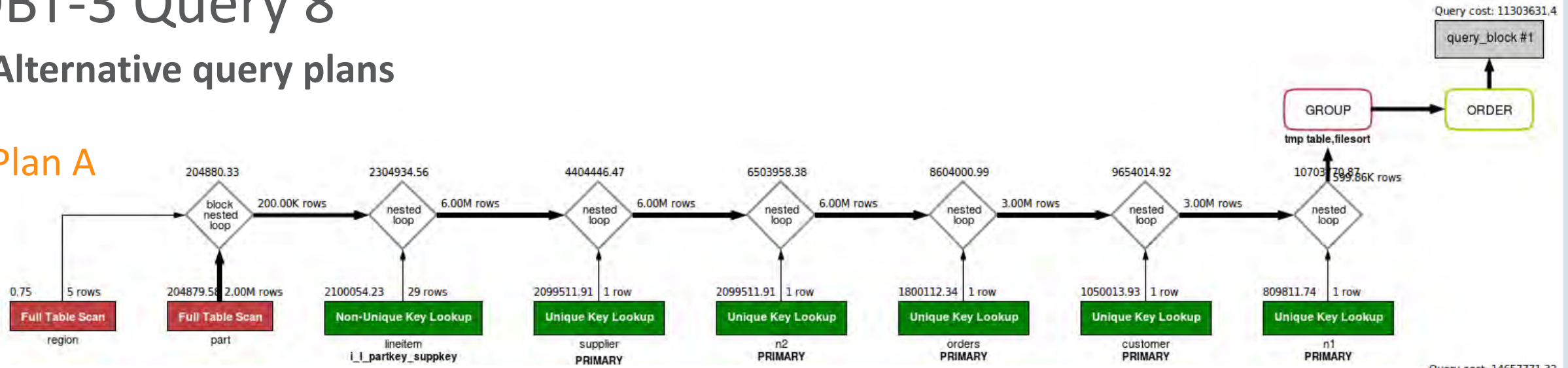
```
SELECT o_year, SUM(CASE WHEN nation = 'FRANCE' THEN volume ELSE 0 END) / SUM(volume) AS
    mkt_share
FROM (
    SELECT EXTRACT(YEAR FROM o_orderdate) AS o_year,
           l_extendedprice * (1 - l_discount) AS volume, n2.n_name AS nation
    FROM part
    JOIN lineitem ON p_partkey = l_partkey
    JOIN supplier ON s_suppkey = l_suppkey
    JOIN orders ON l_orderkey = o_orderkey
    JOIN customer ON o_custkey = c_custkey
    JOIN nation n1 ON c_nationkey = n1.n_nationkey
    JOIN region ON n1.n_regionkey = r_regionkey
    JOIN nation n2 ON s_nationkey = n2.n_nationkey
    WHERE r_name = 'EUROPE' AND o_orderdate BETWEEN '1995-01-01' AND '1996-12-31'
           AND p_type = 'PROMO BRUSHED STEEL'
) AS all_nations GROUP BY o_year ORDER BY o_year;
```

High selectivity

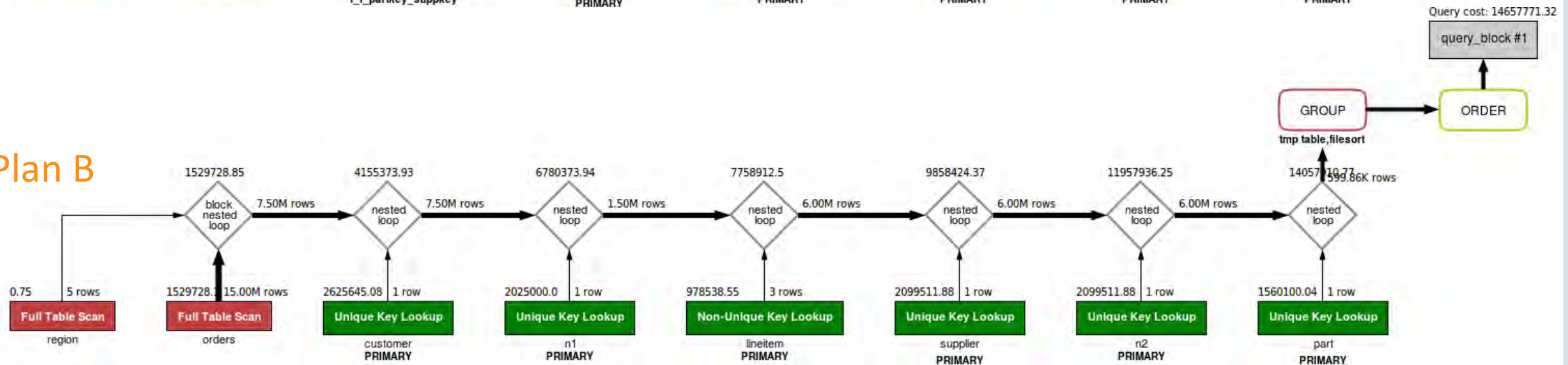
# DBT-3 Query 8

## Alternative query plans

### Plan A



### Plan B





# DBT-3 Query 8

## Execution time (MySQL 8.0.3)

	In-memory	Disk-bound
<b>Plan A</b>	<b>5.8 secs</b>	9 min 47 secs
<b>Plan B</b>	77.5 secs	<b>3 min 49 secs</b>

## Selected plan

	In-memory	Disk-bound
<b>MySQL 5.6</b>		Plan B
<b>MySQL 5.7</b>		Plan A
<b>MySQL 8.0</b>	Plan A	Plan B

DBT-3 Scale factor 10

In-Memory: innodb\_buffer\_pool\_size = 32 GB

Disk-bound: innodb\_buffer\_pool\_size = 1 GB



# Histograms

## Column statistics

- Provides the optimizer with information about column value distribution
- To create/recalculate histogram for a column:  
**`ANALYZE TABLE table UPDATE HISTOGRAM ON column WITH n BUCKETS;`**
- May use sampling
  - Sample size is based on available memory (`histogram_generation_max_mem_size`)
- Automatically chooses between two histogram types:
  - Singleton: One value per bucket
  - Equi-height: Multiple value per bucket

# Histograms

## Example query

```
EXPLAIN SELECT *
FROM customer JOIN orders ON c_custkey = o_custkey
WHERE c_acctbal < -1000 AND o_orderdate < '1993-01-01';
```

id	select type	table	type	possible keys	key	key len	ref	rows	filtered	extra
1	SIMPLE	orders	ALL	i_o_orderdate, i_o_custkey	NULL	NULL	NULL	15000000	31.19	Using where
1	SIMPLE	customer	eq_ref	PRIMARY	PRIMARY	4	dbt3.orders. o_custkey	1	33.33	Using where

# Histograms

Create histogram to get a better plan

**ANALYZE TABLE customer UPDATE HISTOGRAM ON c\_acctbal WITH 1024 buckets;**

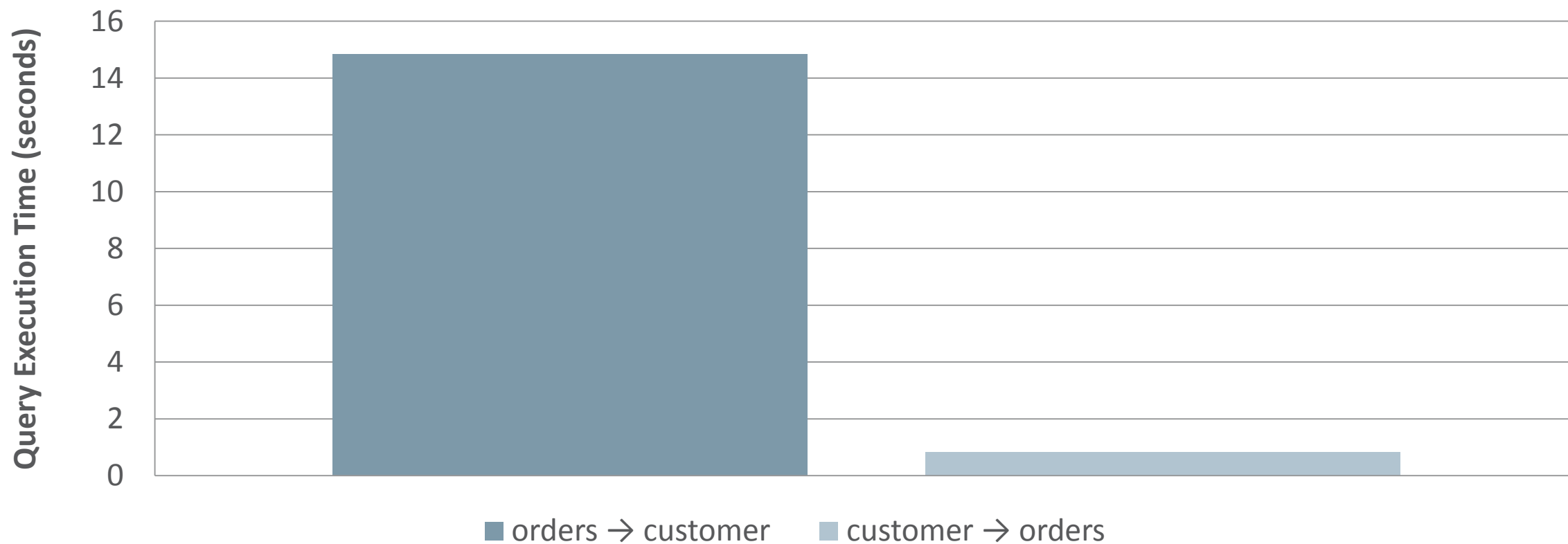
**EXPLAIN SELECT \***

**FROM customer JOIN orders ON c\_custkey = o\_custkey**

**WHERE c\_acctbal < -1000 AND o\_orderdate < '1993-01-01';**

id	select type	table	type	possible keys	key	key len	ref	rows	filtered	extra
1	SIMPLE	customer	ALL	PRIMARY	NULL	NULL	NULL	1500000	0.00	Using where
1	SIMPLE	orders	ref	i_o_orderdate, i_o_custkey	i_o_custkey	5	dbt3. customer. c_custkey	15	31.19	Using where

# Comparing Join Order Performance



# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Hints: Join Order

- Hints to control table order for join execution
- 5.7: `STRAIGHT_JOIN` to force the listed order in FROM clause
- 8.0:
  - `JOIN_FIXED_ORDER` `/* replacement for STRAIGHT_JOIN*/`
  - `JOIN_ORDER` `/* use specified order */`
  - `JOIN_PREFIX` `/* use specified order for first tables */`
  - `JOIN_SUFFIX` `/* use specified order for last tables */`
  - No need to reorganize the FROM clause to add join order hints like you will for `STRAIGHT_JOIN`

# Join Order Hints - Example

Change join order with hint

```
EXPLAIN SELECT /*+ JOIN_ORDER(customer, orders) */ *
FROM customer JOIN orders ON c_custkey = o_custkey
WHERE c_acctbal < -1000 AND o_orderdate < '1993-01-01';
```

id	select type	table	type	possible keys	key	key len	ref	rows	filtered	extra
1	SIMPLE	customer	ALL	PRIMARY	NULL	NULL	NULL	1500000	33.33	Using where
1	SIMPLE	orders	ref	i_o_orderdate, i_o_custkey	i_o_custkey	5	dbt3. customer. c_custkey	15	31.19	Using where

Alternatives with same effect for this query:

```
JOIN_PREFIX(customer) JOIN_SUFFIX(orders) JOIN_FIXED_ORDER()
```

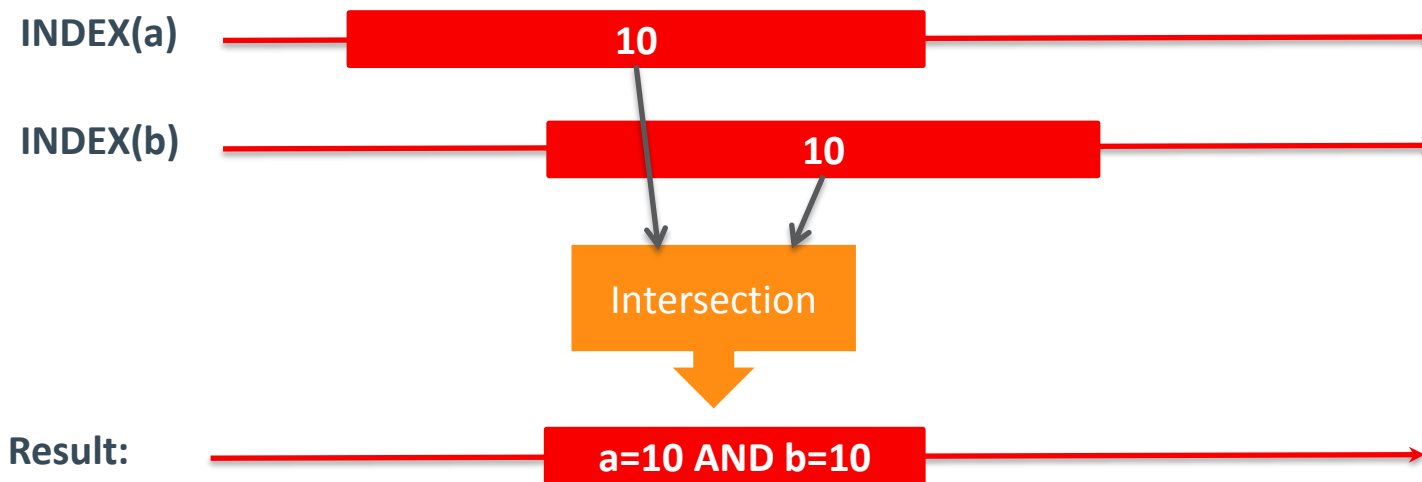
# Hints: Index Merge

- Index merge: Merge rows from multiple range scans on a single table
- Algorithms: union, intersection, sort union
- Users can specify which indexes to use for index merge

– /\*+ INDEX\_MERGE() \*/

– /\*+ NO\_INDEX\_MERGE() \*/

**SELECT \* FROM t1 WHERE a=10 AND b=10**





# Index Merge Hint - Example

**EXPLAIN SELECT count(\*) FROM users  
WHERE user\_type=2 AND status=0 AND parent\_id=0;**

Low selectivity



id	select type	table	type	possible keys	key	key len	ref	rows	Extra
1	SIMPLE	users	index_merge	parent_id, status, user_type	user_type, status, parent_id	1,1,4	NULL	2511	Using intersect (user_type, status, parent_id); Using where; Using index

```
mysql> SELECT count(*) FROM users WHERE user_type=2 AND status=0 AND parent_id=0;
```

...

```
1 row in set (1.37 sec)
```

```
mysql> SELECT /*+ INDEX_MERGE(users user_type, status) */ count(*)  
-> FROM users WHERE user_type=2 AND status=0 AND parent_id=0;
```

...

```
1 row in set (0.18 sec)
```

# Hints: Set session variables

- Set a session variable for the duration of a single statement
- Examples:

```
SELECT /* SET_VAR(sort_buffer_size = 16M) */ name FROM people ORDER BY name;
INSERT /* SET_VAR(foreign_key_checks = OFF) */ INTO t2 VALUES (1, 1), (2, 2), (3, 3);
SELECT /* SET_VAR(optimizer_switch = 'condition_fanout_filter = off') */ *
FROM customer JOIN orders ON c_custkey = o_custkey
WHERE c_acctbal < 0 AND o_orderdate < '1993-01-01';
```

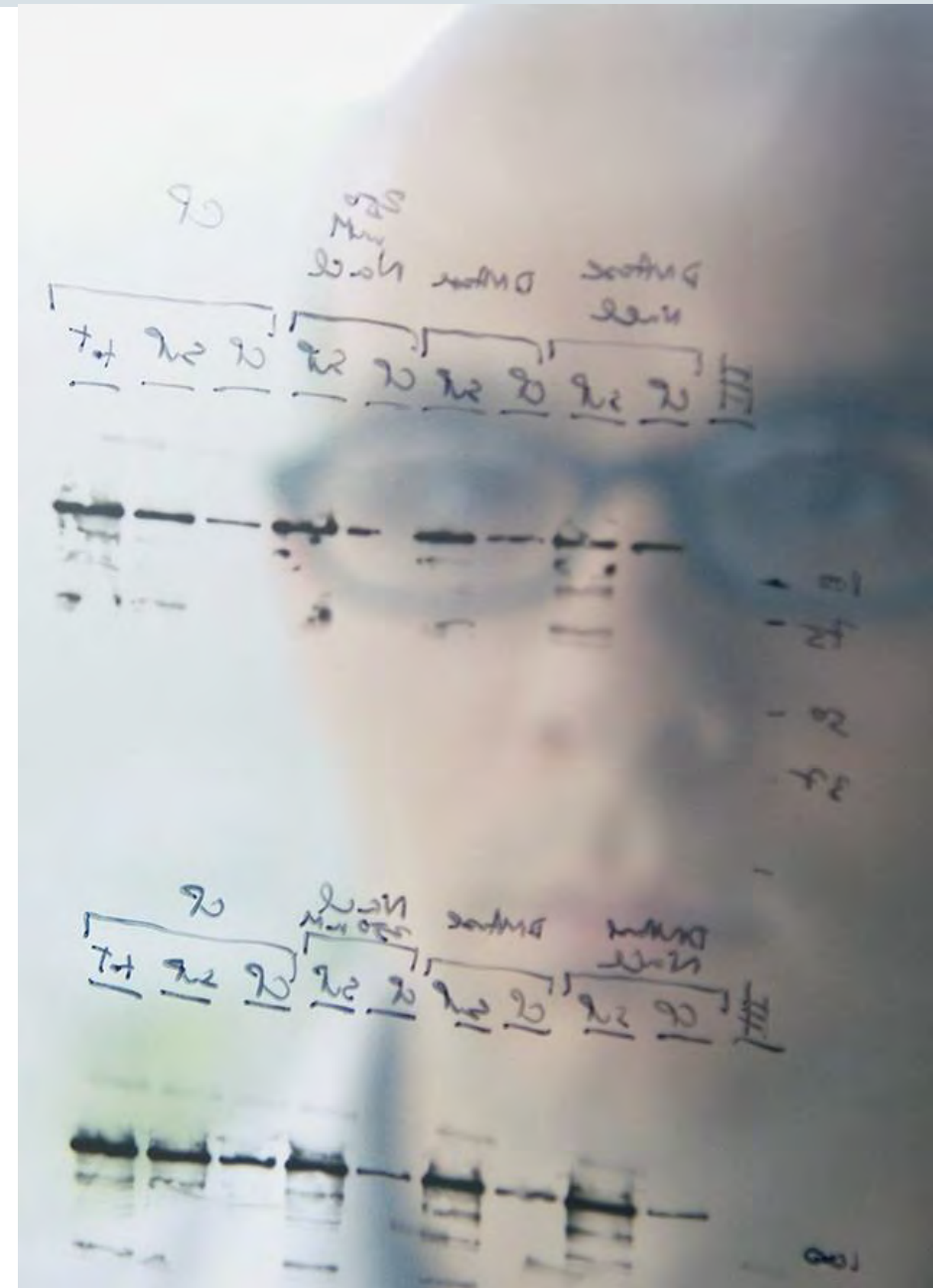
- NB! Not all session variables are settable through hints:

```
mysql> SELECT /*+ SET_VAR(max_allowed_packet=128M) */ * FROM t1;
Empty set, 1 warning (0,01 sec)

Warning (Code 4537): Variable 'max_allowed_packet' cannot be set using
SET_VAR hint.
```

# Program Agenda

- Common table expressions
- Window functions
- UTF8 support
- GIS
- SKIP LOCKED, NOWAIT
- JSON functions
- Index extensions
- Cost model
- Hints
- Better IPv6 and UUID support



# Improved Support for UUID

```
mysql> select uuid();
```

```
+-----+
| uuid() |
+-----+
| aab5d5fd-70c1-11e5-a4fb-b026b977eb28 |
+-----+
```

- Five “-” separated hexadecimal numbers
- MySQL uses version 1, the first three numbers are generated from the low, middle, and high parts of a timestamp.
- 36 characters, inefficient for storage
  - ➔ Convert to BINARY(16) datatype, only 16 bytes

# Improved Support for UUID

## Functions to convert UUID to and from binary

- `UUID_TO_BIN(string_uuid, swap_flag)`
- `BIN_TO_UUID(binary_uuid, swap_flag)`
- `IS_UUID(string_uuid)`


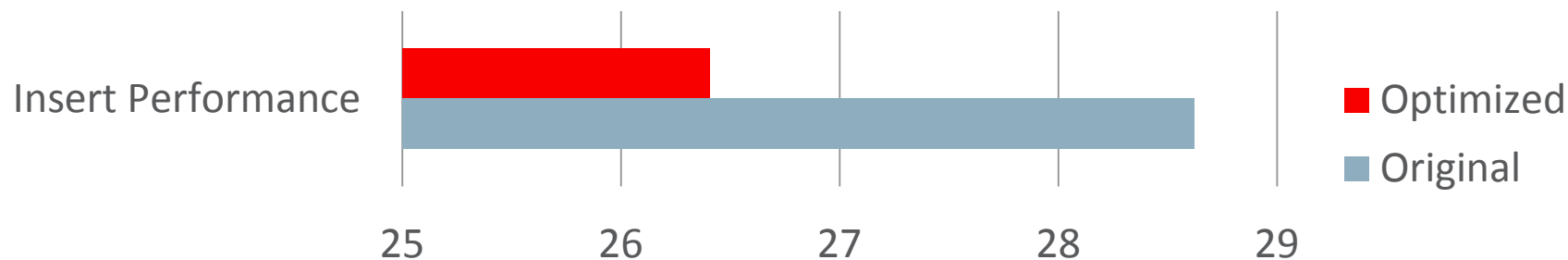


# UUID\_TO\_BIN Optimization

- Binary format is now smaller
- Shuffling low part with the high part improves index performance

From **VARCHAR(36)** 53303f87-78fe-11e6-a477-8c89a52c4f3b

To **BINARY(16)** 11e678fe53303f87a4778c89a52c4f3b

# IPv6

- **New!** Bit-wise operations on binary data types
  - Designed with IPv6 in mind:
  - INET6\_ATON(address) & INET6\_ATON(network)
  - No longer truncation beyond 64 bits



# All These Features and More...

- Improved Query Scan Performance
- GROUPING()
- Hint: Merge/materialize derived table/view
- JSON:
  - Partial update
  - Improved performance of sorting and grouping of JSON values
  - Path expressions: Array index ranges
  - JSON\_MERGE\_PATCH()
- Skip index dives for FORCE INDEX
- Parser Refactoring

**Try it out!**

**Give us feedback!**





## Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



# Integrated Cloud

## Applications & Platform Services

ORACLE®