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.
SQL & Optimizer Features in 8.0

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

• 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 the result.
• 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)
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;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>333</td>
<td>Yasmina</td>
<td>333</td>
</tr>
<tr>
<td>198</td>
<td>John</td>
<td>333,198</td>
</tr>
<tr>
<td>692</td>
<td>Tarek</td>
<td>333,692</td>
</tr>
<tr>
<td>29</td>
<td>Pedro</td>
<td>333,198,29</td>
</tr>
<tr>
<td>123</td>
<td>Adil</td>
<td>333,692,123</td>
</tr>
<tr>
<td>4610</td>
<td>Sarah</td>
<td>333,198,29,4610</td>
</tr>
<tr>
<td>72</td>
<td>Pierre</td>
<td>333,198,29,72</td>
</tr>
</tbody>
</table>
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;
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.
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`
Window Function Example: Frames

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>160000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>220000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>190000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>210000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>300000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
</tr>
</tbody>
</table>

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;

A frame is a subset of a partition

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

ORDER BY name within each partition
JSON_TABLE

• JSON is not limited to CRUD, can also be used in complex queries
• JSON_TABLE creates a relational view of JSON data
  – Each object in a JSON array as a row
  – JSON values within an JSON object as column values
• Query the result of JSON_TABLE() as a relational table using SQL
• Leverage existing framework for aggregation
JSON_TABLE

Convert JSON documents to relational tables

• Table t1 has a column json_col with content like this:

```json
{ "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:

```sql
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
Convert JSON documents to relational tables

```
[  
  { "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} ] }  
]
```

<table>
<thead>
<tr>
<th>id</th>
<th>father</th>
<th>married</th>
<th>child_id</th>
<th>child</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>1</td>
<td>1</td>
<td>Eric</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>John</td>
<td>1</td>
<td>2</td>
<td>Beth</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>0</td>
<td>1</td>
<td>Sarah</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>0</td>
<td>2</td>
<td>Noah</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>0</td>
<td>3</td>
<td>Peter</td>
<td>1</td>
</tr>
</tbody>
</table>
JSON_TABLE – Nested Arrays

```sql
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' ) )
```
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;

<table>
<thead>
<tr>
<th>father</th>
<th>#children</th>
<th>age average</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2</td>
<td>11.0000</td>
</tr>
<tr>
<td>Paul</td>
<td>3</td>
<td>4.3333</td>
</tr>
</tbody>
</table>
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
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

Execution time (MySQL 8.0.3)

<table>
<thead>
<tr>
<th></th>
<th>In-memory</th>
<th>Disk-bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan A</td>
<td>5.8 secs</td>
<td>9 min 47 secs</td>
</tr>
<tr>
<td>Plan B</td>
<td>77.5 secs</td>
<td>3 min 49 secs</td>
</tr>
</tbody>
</table>

Selected plan

<table>
<thead>
<tr>
<th></th>
<th>In-memory</th>
<th>Disk-bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL 5.6</td>
<td>Plan B</td>
<td></td>
</tr>
<tr>
<td>MySQL 5.7</td>
<td>Plan A</td>
<td></td>
</tr>
<tr>
<td>MySQL 8.0</td>
<td>Plan A</td>
<td>Plan B</td>
</tr>
</tbody>
</table>

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
### Example query

**EXPLAIN**

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

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>filtered</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ALL</td>
<td>i_o_orderdate, i_o_custkey</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>150000000</td>
<td>31.19</td>
<td>Using where</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>eq__ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>dbt3.orders.o_custkey</td>
<td>1</td>
<td>33.33</td>
<td>Using where</td>
</tr>
</tbody>
</table>
Histograms

Create histogram to get a better plan

ANALYZE TABLE customer UPDATE HISTOGRAM ON c_acctbal;

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

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>filtered</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>ALL</td>
<td>PRIMARY</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>1500000</td>
<td>0.00</td>
<td>Using where</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ref</td>
<td>i_o_orderdate, i_o_custkey</td>
<td>i_o_custkey</td>
<td>5</td>
<td>dbt3.customer.c_custkey</td>
<td>15</td>
<td>31.19</td>
<td>Using where</td>
</tr>
</tbody>
</table>
Comparing Join Order

Performance

Query Execution Time (seconds)

- orders → customer
- customer → orders
Integrated Cloud
Applications & Platform Services
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
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 file sort for ORDER BY clause with ASC/DESC sort key
  – Forward index scan is slightly faster than backward index scan