Working with the Oracle query optimizer

- First need to make sure stats are in place in the catalog (if tables were just loaded, or indexes added)
- Analyze table on Oracle 12c is supported, but doesn't collect statistics for partitioned tables, run in parallel, etc.
- See notice in analyze doc
- To get better stats, we need to execute the following code (from slide 6)

  ```sql
  SQL> exec dbms_stats.gather_table_stats('SETQ_DB', 'BENCH', cascade=>'true');
  ```

  Here cascade means analyze its indexes too.
- To drop stats:

  ```sql
  SQL> exec dbms_stats.delete_table_stats('SETQ_DB', 'bench');
  ```

Bench table

- 1M rows, 240 bytes each, so 240MB table data (30K pages)
- Data is in a heap table
  - Recall that Oracle only offers clustered index via "IOT" index-organized table.
- Column names show their cardinality:
  - k4 means 4 different values, 1,2,3,4
  - k100K means 100k different values 1, 2, …, 99999, 100000
- B-tree indexes on kseq, k4, k10, k1k, k100, k500 columns
- No index on k5, k25, k40, k40k, k250k columns

Simple plan example, indexed column k500

SQL> alter session set current_schema = setq_db;
SQL> explain plan for select max(s1) from bench where k500=2;
SQL> select * from table(dbms_xplan.display());

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>BENCH</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>* 3</td>
<td>INDEX RANGE SCAN</td>
<td>K500KIN</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

Predicate Information (identified by operation id):

```
 3 - access("K500KIN")
```

- K500K index has 2 rows for each key
- Table access by these two ROWIDs extracts 28 bytes (1 value): 2 rows of 14 bytes each
- These are aggregated and one value returned
- Same plan for k100, k10k, but not k100k...

Simple plan, indexed k100 column:

SQL> explain plan for select max(s1) from bench where k100=2;
SQL> select * from table(dbms_xplan.display());

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>BENCH</td>
<td>10000</td>
<td>117K</td>
</tr>
</tbody>
</table>
```

Predicate Information (identified by operation id):

```
 2 - filter("K100"=")
```

- Here RF=1/100, so about 10,000 rows are produced by the filtered table scan, and each needs the s1 value, 12 bytes, so 120KB of data.
- Oracle has ignored the K100 index, preferring to do a table scan (30,000 pages) rather than do 10,000 index probes and rid lookups. Let's see why...

Seeing the results of gathering stats

SQL> SELECT column_name, num_distinct, num_buckets, histogram FROM ALL_TAB_COL_STATISTICS where table_name='BENCH' order by num_distinct;

```
COLUMN_NAME NUM_DISTINCT NUM_BUCKETS HISTOGRAM
---------- ------------ ------------ ------------
K2           2           1 NONE         
K4           4
K5           5 FREQUENCY
K10          10          10 FREQUENCY
K25          25          NONE          
K100         1000000     1000000     NONE          
K100k        100816      1000000     NONE          
K250k        248288      248288      NONE          
K500k        439200      439200      NONE          
KSEQ         1000000     1000000     NONE          
```

Oracle figures that the default RF of 1/num_distinct will be good enough for k10k and up
Simple plan, k100 case

```sql
select max(s1) from bench where k100=2;
```

- Cost of Oracle's plan (with known histograms): read entire table, about 30,000 I/Os.
- Cost of index-driven plan:
  - Here RF=1/100, so about 10,000 rows are found in the index. Maybe 100 I/Os to index.
  - Each needs the s1 value, so the ROWID is used to access the table. This takes 10,000 I/Os (assuming buffering of upper levels of the index.)
- The difference here: sequential vs. random I/O
  - Plan 1: table scan, 30,000 sequential I/Os
  - Plan 2: use index, 10,000 random accesses
- But sequential I/O uses multi-block I/O, can be 10-25x faster.
- That's assuming HDD. On SSD, use the index.

Join Example
(with index on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>INDEX REBUILD INDEX</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Example with set autotrace ...

```sql
SQL> select max(s1) from bench where k100=2;
```

```
MAX(S1)  --------
--------
12345678
Elapsed: 00:00:00.03
```

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes (also Cost, Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>*</td>
<td>INDEX RANGE SCAN</td>
<td>K500KIN</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

Predicate Information (identified by operation id):

```sql
2 - access("K500K"=2)
```

Statistics

```sql
1  recursive calls
0  db block gets
5  consistent gets
5  physical reads
```

Join Example
(with indexes on k40k and kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>INDEX REBUILD INDEX</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>EXPLAIN FILTER</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Easier way to see plans:

- set autotrace on explain statistics
- Also set line 130 to avoid wrapping
- Then just select …
- After this returns, you see the explain plan, plus actual statistics on the query
- The explain plan is not guaranteed to be the exact plan used
- Note: to set string column output format: SQL> column column_name format a10

Example with set autotrace ...

```sql
SQL> select max(s1) from bench where k500k=2;
```

```
MAX(S1)  --------
--------
```

Join Example
(with index on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>INDEX REBUILD INDEX</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on k40k and kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>EXPLAIN FILTER</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>EXPLAIN FILTER</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>EXPLAIN FILTER</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>EXPLAIN FILTER</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>EXPLAIN FILTER</td>
<td></td>
<td>21380</td>
<td></td>
</tr>
</tbody>
</table>
```

Join Example
(with indexes on kseq)

```sql
SELECT max(b1.s2) FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```

Oracle uses Hash Join:

```sql
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>1000K</td>
<td>32M</td>
</tr>
</tbody>
</table>
```
Hash Join Cost Analysis, case of in-memory HT, no partitioning (for small tables)

- Hash Join: 1M rows (24MB) from b1, 200K rows (1.5MB) from selection on b2
- So build hash table from b2, should fit in memory (apparently 3.9MB). If not, use partitioning.
- Hash the 1M rows of b1 and output to pipeline
- I/O Cost: read bench twice (once as b1, once as b2), about 60,000 I/Os. Less if table fits in memory (our case, so only read it once)
- MySQL can’t do hash join, MariaDB can

Hash Join Cost Analysis, by textbook algorithm

- Hash Join: 1M rows (24MB) from b1, 200K rows (1.5MB) from selection on b2 (see explain plan, slide before last)
- Book assumes partitioning needed first. Suppose only 1MB of memory available.
- So read and write all data of both tables into partitions, say 100 partitions (using 800KB of buffers, about 1MB)
- For each partition, build hash table from b2, should fit in memory (about .01(3.8M) = 38KB)
- Hash the 10K rows of b1 part., output to pipeline
- I/O Cost: read both tables, about 60,000 I/Os. Write and read incoming tables to HJ: 24MB=3K blocks, 1.5MB = .2K blocks, total 3200 writes, 3200 reads, 6400 I/Os.
- Cost = 66,400 I/Os.
- Cost = M + N + 2(M_HJ + N_HJ), where M_HJ and N_HJ are the #pages coming into the HJ operator after selections are made and unused columns are dropped. The book ignores this effect, simplifying to 3(M+N).

nested loops cost analysis, b2 outer

- Indexed NL Join? Not possible, no index on k40k.
- Could consider blocked NLJ.

Paged Nested Loops Cost Analysis

- Paged NL Join: 200K rows (1.5 MB = 190 pages) from selection
- Then read one page of left-side input, read all of b1, then another page, read all of b1.
- Cost: read b1 190 times, b2 once, = 191*30,000 I/Os. No good.
**Blocked Nested Loops Cost Analysis**

- **Blocked NL Join:** 200K rows (1.5MB) from selection, 1M rows (240MB) in b1.
- **Cost:** assume 1MB memory available, so block = 1MB.
- Then read .75MB (half) of left-side input, read all of b1, then another .75MB, read all of b1.
- **Cost:** read b1 twice, b2 once, = 90,000 seq i/o.
- Only 60,000 if can use 2MB memory, and that's the same as hash join.
- Mysql v 5.6 can use this approach.

**Hash Join Optimization**

Since hash joins are common plans used by Oracle, how can we help make them fast?

- Raise **pga_aggregate_target** to maybe 10% of server memory (exact commands depend on Oracle version)
- Since the hash join speed depends on the size of the tables, be sparing with your select list: avoid select * from ...
- Don't worry about indexes on the join condition columns: they won't be used!
- Of course, if you think NLJ is possible, do use these indexes.
- Add indexes to help with selective single-table predicates: they will be used (on either or both sides) and greatly reduce the size of the join.
- Be more selective with the single-table predicates if possible.
  * Ex: instead of looking at all employees, look at one department's.

**What about mysql?**

- Mysql v 5.7 (our case on pe07) only joins using nested loops join, including blocked nested loops.
- MariaDB 10.1 (our case on cloud sites) uses hash join too
- Mysql has “explain”, but it is not as complete or easy to understand as Oracle’s.
- Mysql v 5.7 has new JSON-format plans for explain.

**Mysql EXPLAIN**

```sql
mysql> explain select max(s1) from bench where k500k=2 and k4=2;
```

```
<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>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>bench</td>
<td>ref</td>
<td>k500kin,k4in</td>
<td>k500kin</td>
<td>4</td>
<td></td>
<td>1</td>
<td>Using intersect(k500kin,k4in); Using where</td>
</tr>
</tbody>
</table>
```

- Shows index merge of two indexes.
- Though not really worth it: only 2 rows satisfy k500k=2
- This was mysql v5.6. Mysql v5.7 uses only the one index, a better plan.

**Mysql can merge indexes**

```sql
mysql> explain select max(s1) from bench where k500k=2 and k4=2;
```

```
<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>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>bench</td>
<td>ref</td>
<td>k500kin,k4in</td>
<td>k500kin</td>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

- In this case, we see mysql chooses the one better key.

**Mysql and joins**

- Mysql only uses Nested Loop Joins, and left-deep plans.
- Thus it is sufficient to know the order of the joins and we know the plan tree.
- The explain output lists one line per table, leftmost table first.
Yelp_db core tables

- Review table: the big table in the middle
  - 4.5M rows in both DBs, but different storage of review text/clob (the actual texts of the submitted reviews, up to 54K bytes larg)
  - Oracle: 6.7GB data (840K 8KB pgs) in Oracle
  - MySQL: main table has 3.7GB data (230K 16KB pgs), 820 bytes/row
  - test column data separately stored (all of it, not just bigger review texts)
  - Index on PK = id, clustered only in mysql.
  - Indexes on 2 FK cols: business_id and user_id
- Business table
  - 150K rows, 225MB data (1400 16KB pgs), so 140 bytes/row
  - Index on PK = id, clustered only in mysql.
  - Each business has 30 reviews by simple division: 4.5M/150K = 30
- Yuser table
  - 1M rows, 150MB data (9400 pgs), so 150 bytes/row/on ave.
  - Index on PK = id, clustered only in mysql.
  - All indexes are B-tree indexes

Nested Loops Cost Analysis, business outer

- Indexed NL Join: 4.5M rows in review with index on business_id, 150K rows in business, 30K in "NV" (RF = 0.2)
- Cost: 30K index probes, each matching 30 reviews, then follow rid, plus reading outer table (about 1400 files using 16KB pages, 2800 for 8KB pgs)
- Cost = 30*30K + 1400/2800 = 900K pages
- We will see this is mysql's choice.
- Note: 900K row accesses hits almost all pages in the mysql table, since it has only 230K pages.

Oracle plan: Hash Join

```
SELECT COUNT(*) FROM yelp_db.business B, yelp_db.review R
WHERE B.id = R.business_id AND R.stars = 5 AND B.state = 'NV';
```

```
+----+----------+--------+--------+--------+----------+--------+----------+--------+
| Id | Operation | Name   | Type   | Possible Keys | Key Conditions | Extra |
+----+----------+--------+--------+--------+----------+--------+----------+--------+
|  1 | SELECT STATEMENT | | | | | |
|  2 | SELECT STATEMENT | | | | | |
|  3 | TABLE ACCESS FULL | BUSINESS | 820 | 820 | (1 row(s) filtered) | |
|  4 | TABLE ACCESS FULL | REVIEW | 498 | 498 | (1 row(s) filtered) | |
+----+----------+--------+--------+--------+----------+--------+----------+--------+

Cost = 30*30K + 1400/2800 = 900K pages
```

Hash Join Cost Analysis, case of in-memory HT, no partitioning (for small smaller table)

- Hash Join: 0.8MB from selection on business, 50MB from selection on review
- So build hash table from business, should fit in memory. (If not, use partitioning.)
- Hash the rows of review and output to pipeline
- i/o Cost: read both tables, about 230K 16KB (io), 643K 8KB sequential (i/o Oracle tables).

Mysql plan: Indexed NLJ, business outer

```
mysql> explain SELECT COUNT(*) FROM yelp_db.business B, yelp_db.review R
WHERE B.id = R.business_id AND R.stars = 5 AND B.state = 'NV';
```

```
+----+----------+--------+--------+--------+----------+--------+----------+--------+
| Id | Operation | Name   | Type   | Possible Keys | Key Conditions | Extra |
+----+----------+--------+--------+--------+----------+--------+----------+--------+
|  1 | SIMPLE   | R      | NULL   | ALL     | NULL     | NULL |
|  2 | PRIMARY  | NULL   | NULL   | NULL    | NULL     | NULL |
|  3 | filter("B"."BUSINESS_ID") | | | | | | |
|  4 | filter("R"."STATE"='NV') | | | | | | |
+----+----------+--------+--------+--------+----------+--------+----------+--------+

This report shows mysql does an indexed NLJ with business (B) the outer table (the first listed table here).
Oracle Hash Join wins over Mysql NLJ

- Oracle chooses 840K seq. i/o for HJ over 900K random i/o for NLJ here, clear winner because seq. i/o is so much faster.
- Recall we earlier estimated seq. i/o is up to 3x faster on HDD even if data is trapped in a tablespace for random i/o.
- Of course dbs3 has much faster disk system than pe07.
- Similar story for the other two queries: HJ vs. NLJ, HJ wins.
- We don’t really know why Oracle is 17x faster using cached data, since no disk i/o is happening in that case.
- Streaming data in memory is faster than random access in memory, one effect (more CPU cache traffic with random access)
- The systems have similar CPUs, though pe07 has 2 processors vs. 1 for dbs3.
- pe07 has twice as much memory as dbs3 (128G vs 64GB).

Nested Loops Cost Analysis, state=’WI’
(Oracle uses NLJ, twice)

Oracle Bitmap Indexes

```
cREATE TABLE emps

  id int not null primary key,
  ems int,
  gender char(1),
  salarycat smallint,
  dept char(5)
);

CREATE BITMAP INDEX genderx ON emps(gender); (2 values, 'M' & 'F')
CREATE BITMAP INDEX salesx ON emps(salarycat); (10 values, 1-10)
CREATE BITMAP INDEX deptx ON emps(dept); (12 vals, 5 char: 'ACCOUNT')
```

Oracle choose a NLJ for state=’WI’

- WI has only 4190 businesses, compared to 30K for NV.
- Oracle uses two NLJs as shown on last slide, cost = 125K i/o.
- First, after "alter system flush buffer_cache," to clear buffer cache
  - Elapsed: 00:00:01.84 (only 0.015 ms/i/o), so not normal “random (io)"
- Second: table data should be in buffer cache
  - Elapsed: 00:00:00.35

Mysql time: using single NLJ as shown earlier [cost = 4K*30 = 120K i/o]  

- First time (but some data in OS buffers)
  - 1 row in set (2.70 sec)
- Second time: table data should be in buffer cache
  - 1 row in set (2.69 sec)

Oracle NLJ vs mysql NLJ: state=’WI’ query

- Even with a null-value bitmap, only 3 bits/row for gender
- ORACLE uses compression for low-density bitmaps, so they don’t waste space.
- Note: Call a bitmap “verbatim” if not compressed.
- Fast AND and OR of verbatim bitmaps speeds queries. Idea is: everlay unsigned int array on bitmap, loop through two arrays ANDing array (6 in C), and producing result of AND of predicates.
- Parallellism speeds things (64 bits at a time).
- But for updates, bitmaps can cause a slowdown when the bitmaps are compressed (need to be decompressed, may recompress differently). Don’t use bitmap indexes if have frequent updates (OLTP situation).
Query plan with bitmap indexes

EXPLAIN PLAN FOR SELECT * FROM t WHERE c1 = 2 AND c2 <> 6 OR c3 BETWEEN 10 AND 20;

SELECT STATEMENT
   TABLE ACCESS T BY INDEX ROWID
   BITMAP CONVERSION TO ROWID
   BITMAP OR
   BITMAP MINUS
   BITMAP MINUS
   BITMAP INDEX C1_IND SINGLE VALUE
   BITMAP INDEX C2_IND SINGLE VALUE
   BITMAP INDEX C2_IND SINGLE VALUE
   BITMAP MERGE
   BITMAP INDEX C3_IND RANGE SCAN

Bitmap plan discussion

• In this example, the predicate c1 = 2 yields a bitmap from which a subtraction can take place.
• From this bitmap, the bits in the bitmap for c2 = 6 are subtracted.
• Also, the bits in the bitmap for c2 IS NULL are subtracted, explaining why there are two MINUS row sources in the plan.
• The NULL subtraction is necessary for semantic correctness unless the column has a NOT NULL constraint.
• The TO ROWIDS operation is used to generate the ROWIDs that are necessary for the table access.

Scaling up

• Our experiments are using a single disk, so parallelism is not important.
• Serious databases use RAID, so multiple disks are working together, more or less like one faster disk.
• Huge databases use partitioning and query plans where work on different partitions proceeds in parallel.
• Will return to this when studying data warehousing.