Query Optimization, part 2: query plans in practice
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 class 6)

SQL> exec dbms_stats.gather_table_stats(
‘SETQ_DB’, ‘BENCH’, cascade=>true);

• Here cascade means analyze its indexes too.
• To drop stats:
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, k100, k1k, k100k, k500k columns
• No index on k5, k25, k40, k40k, k250k columns
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;

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>NUM_DISTINCT</th>
<th>NUM_BUCKETS</th>
<th>HISTOGRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>K2</td>
<td>2</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K4</td>
<td>4</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K5</td>
<td>5</td>
<td>5</td>
<td>FREQUENCY</td>
</tr>
<tr>
<td>K10</td>
<td>10</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K25</td>
<td>25</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K100</td>
<td>100</td>
<td>100</td>
<td>FREQUENCY</td>
</tr>
<tr>
<td>K1K</td>
<td>1000</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K10K</td>
<td>10000</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K40K</td>
<td>40348</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K100K</td>
<td>100816</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K250K</td>
<td>248288</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>K500K</td>
<td>439200</td>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>KSEQ</td>
<td>1000000</td>
<td>1</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Oracle figures that the default RF of 1/num_distinct will be good enough for k10k and up
Simple plan example, indexed column k500

SQL> alter session set current_schema = setq_db;
SQL> explain plan for select max(s1) from bench where k500k=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>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("K500K"=2)

• K500K index has 2 rows for each key
• Table access by those two ROWIDs extracts 28 bytes (s1 value): 2 rows of 14 bytes each
• These are aggregated and one value returned
• Same plan for k100k, k10k, but not k100...
Simple plan, indexed k100 column:

```sql
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></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"=2)

• 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...
Simple plan, k100 case

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

- Cost of Oracle’s plan (with known histograms): read entire table, about 30,000 i/o.

- 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 index probes, so about 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.
Easier way to see plans:

set autotrace on explain statistics
• Or just set autotrace on exp stat
• Also set timing on
• 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 ..., set timing...

SQL> select max(s1) from bench where k500k=2;
MAX(S1)
----------
12345678
Elapsed: 00:00:00.03

<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>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("K500K"=2)

Statistics

1 recursive calls
0 db block gets
5 consistent gets
5 physical reads Unfortunately, physical writes are not reported here
...
Join Example
(with index on kseq)

```
SELECT max(b1.s2)
FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;
```
Join Example
(with indexes on k40k and kseq)

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

Left-deep tree (for NLJ)
SELECT max(b1.s2)
FROM bench b1, bench b2
WHERE b1.k40k=b2.kseq AND b2.k5=2;

Oracle uses Hash Join:

| Id  | Operation           | Name  | Rows  | Bytes | TempSp | Cost (%CPU) | Time 
|-----|---------------------|-------|-------|-------|--------|-------------|-------
| 0   | SELECT STATEMENT    |       | 1     | 34    |        | 17998 (1)| 00:00:01 |
| 1   | SORT AGGREGATE      |       | 1     | 34    |        |            |        |
|* 2  | HASH JOIN           |       | 1000K | 32M   | 3920K  | 17999 (1)| 00:00:01 |
|* 3  | TABLE ACCESS FULL   | BENCH | 200K  | 1567K |       | 8002 (1)| 00:00:01 |
| 4   | TABLE ACCESS FULL   | BENCH | 1000K | 24M   |       | 8003 (1)| 00:00:01 |

2 - access("B1"."K40K"="B2"."KSEQ")
3 - filter("B2"."K5"=2)

Line 3: 100000/5 = 200K rows, each with kseq, say 8 bytes, = 1600K = 1.6M bytes, OK
Line 4: all rows, drop all cols except k40k and s2, say 20 bytes = 20M bytes, OK

• This hash table is using the temp tablespace instead of dedicated memory, but its pages will be in memory. Here the smaller HT holds 1.6MB data, uses 3.9MB space.
• Recall the rule of thumb that a hash table should be at least twice the size of the data in it.
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\text{HJ}+N\text{HJ}), where M\text{HJ} and N\text{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).
Hash Join Optimization with Oracle (“hybrid hash join” of pg. 465)

- Oracle partitions all b2 (smaller side) data and builds one or more partition’s HT in memory in first pass, while writing other partitions to disk.

- While reading b1 side data, does join with in-memory b2 partition(s), writes out other b1 partitions for later processing (the ones with HTs not yet available)

- Works on processing written-out b1-partitions with next set of in-memory HTs of b2 data, etc.

- i/o Cost: read both tables, about 60,000 i/os. Write and read parts of smaller and larger tables.

- For small enough tables, no partition writing at all.

- This way, cost of HJ doesn’t jump up as join size crosses needs-partitioning boundary.
Nested Loops Cost Analysis, b2 outer

- Indexed NL Join? Not possible, no index on k40k.
- Could consider blocked NLJ.
Nested Loops Cost Analysis, b1 outer

- Indexed NL Join: 1M rows in b2 with index on kseq, 1M rows (20MB) in b1
- Cost: 1 match for each k40k value, 1M index probes, but to only to the first 4% of the table (b2.kseq < 40K), so 40K i/os assuming decent buffering, plus reading b1 table (about 30,000 i/os)
- Cost = 70,000, less if table fits in memory.
- Compare to HJ costs: 60,000 (in-mem HT), 66,400 (partitioning, less if hybrid)
- HJ also benefits from using sequential i/o:
  - NLJ: 30,000 seq + 40,000 random i/os
  - HJ: 60,000-66,400 seq (much faster for HDD)
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/os.
- 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.
In this case, we see mysql chooses the one better key
Mysql can merge indexes

mysql> explain select max(s1) from bench where k500k=2 and k10k=2;

explain select max(s1) from bench where k500k=2 and k10k=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>index_merge</td>
<td>k500kin,k10kin</td>
<td>k500kin,k10kin</td>
<td>4,4</td>
<td>NULL</td>
<td>1</td>
<td>Using intersect(k500kin,k10kin); 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 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 64KB in length)
  • Oracle: 6.7GB data (840K 8KB pgs) in Oracle
    • But 1.6 GB of this is in “LOB storage”, separate from main table
    • So main table has 5.1GB data (640K pgs) 1100 bytes/row (incl. review texts < 4KB)
  • Mysql: main table has 3.7GB data (230K 16KB pgs), 820 bytes/row
    • text 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, 22MB 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
Our Yelp queries: query 1

```
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';
```

- Oracle 12c on dbs3: (5.5s starting from empty buffer cache)

<table>
<thead>
<tr>
<th>First run</th>
<th>Second run</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(*)</td>
<td>-----------</td>
</tr>
<tr>
<td>723579</td>
<td>723579</td>
</tr>
</tbody>
</table>

  Elapsed: 00:00:03.71    Elapsed: 00:00:02.08

- Mysql 5.7 on pe07:

  +----------+  +----------+
  | COUNT(*) |  | COUNT(*) |
  | 725915   |  | 725915   |

  1 row in set (36.45 sec)  1 row in set (36.48 sec)

- These second runs are using cached data. How does Oracle win by factor of 17?
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 i/ops using 16KB pages, 2800 for 8KB pgs)

Cost = 30*30K + 1400/2800 = 900K i/ops, unless table fits in memory (and it does, in effect)

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.
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 i/os, 643K 8KB sequential i/os (Oracle tables).
- We will see this is Oracle’s choice
- Mysql can’t do hash join, MariaDB can

```
COUNT

\( b\.ld = r\.business_id \)

\( b\.state='NV' \)  \( r\.stars=5 \)

business \( \bowtie \) review
```
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';

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>53</td>
<td>229K (1)</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>HASH JOIN</td>
<td></td>
<td>390K</td>
<td>19M</td>
<td>229K (1)</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS FULL</td>
<td>BUSINESS</td>
<td>30571</td>
<td>806K</td>
<td>752 (1)</td>
</tr>
<tr>
<td>* 4</td>
<td>TABLE ACCESS FULL</td>
<td>REVIEW</td>
<td>1982K</td>
<td>49M</td>
<td>228K (1)</td>
</tr>
</tbody>
</table>

2 - access("B"."ID"="R"."BUSINESS_ID")
3 - filter("B"."STATE"='NV')
4 - filter("R"."STARS"=5)
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';

+--------------------------------------------------------------------------+
<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>partitions</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>B</td>
<td>NULL</td>
<td>ALL</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>R</td>
<td>NULL</td>
<td>ref</td>
<td>fk_reviews_business1_idx</td>
<td>fk_reviews_business1_idx</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yeIp_db.B.Id</td>
<td>-36</td>
<td>-10.00</td>
</tr>
</tbody>
</table>
+--------------------------------------------------------------------------+

• 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/ os for NLJ here, clear winner because seq. i/o is so much faster.
  • Recall we earlier estimated seq. i/o is up to 25x 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 (128GB vs 64GB)
Oracle chooses a NLJ for state='WI'
WI has only 4190 businesses, compared to 30K for NV

SQL> explain plan for SELECT COUNT(*) FROM yelp_db.business B, yelp_db.review R
WHERE B.id = R.business_id AND R.stars = 5 AND B.state = 'WI';

<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>53</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>NESTED LOOPS</td>
<td></td>
<td>53467</td>
<td>2767K</td>
</tr>
<tr>
<td>3</td>
<td>NESTED LOOPS</td>
<td></td>
<td>125K</td>
<td>2767K</td>
</tr>
<tr>
<td>* 4</td>
<td>TABLE ACCESS FULL</td>
<td>BUSINESS</td>
<td>4190</td>
<td>110K</td>
</tr>
<tr>
<td>* 5</td>
<td>INDEX RANGE SCAN</td>
<td>FK_REVIEWS_BUSINESS1_IDX</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>* 6</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>REVIEW</td>
<td>13</td>
<td>338</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

  4 - filter("B"."STATE"='WI')
  5 - access("B"."ID"="R"."BUSINESS_ID")
  6 - filter("R"."STARS"=5)
Nested Loops Cost Analysis, state='WI'
(Oracle uses NLJ, twice)

Indexed NL Join: 4.5M rows in review with index on business_id, 150K rows in business, 4K in 'WI' (RF = 0.06)

Cost: 4K index probes in first join (which find all 30 matches together) then follow r1.rids (4.1K*30=125K est., actually 100K) in second join, check stars=5, plus reading outer table (about 1400 i/ios using 16KB pages, 2800 for 8KB pgs)

Cost = 31*4K + 1400/2800 = 125K i/ios

Cost of HJ = 640K sequential i/ios, 5x this NLJ cost, but we expect seq. i/o to be much faster.

Note: Since v 11g, the rid access to review is using “vector i/o”, where multiple requests are sent at once to the disk system (after sort of rids), causing it to be much faster than normal random i/o.
Oracle NLJ vs mysql NLJ: state=‘WI’ query

Oracle: using two NLJs as shown on last slide, cost = 125K i/os
• First, after “alter system flush buffer_cache;” to clear buffer cache
  • Elapsed: 00:00:01.84 (only 0.015 ms/io, so not normal “random i/o”)
• 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/os)
• 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 Bitmap Indexes

create table emps ( 
    eid char(5) not null primary key, 
    ename varchar(16), 
    mgrid char(5) references emps, 
    gender char(1), salarycat smallint, dept char(5));

create bitmap index genderx on usemps(gender); (2 values, 'M' &'F')

create bitmap index salx on usemps(salarycat); (10 values, 1-10)

create bitmap index deptx on usemps(dept); (12 vals, 5 char: 'ACCNT')

• Best for low-cardinality columns
  • Bitmap for gender='M': 0010111...
  • Bitmap for gender='F': 1101000...
Bitmap indexes, cont.

• 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: overlay unsigned int array on bitmap, loop through two arrays ANDing array (& in C), and producing result of AND of predicates. Parallelism 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 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.