# Query Optimization, part 3: query plans in practice

CS634
Lecture 13, Mar 21, 2016

Slides by E. O'Neil based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

## Using Explain Plan on Oracle

- Analyze table on Oracle 10g does not create histograms
- See notice in analyze doc
- To get better plans, we need to execute the following code:

```
SQL> begin
DBMS_STATS.GATHER_table_STATS (OWNNAME =>'EONEIL',
TABNAME=>'BENCH');
end;
/
```

## Seeing the results of gathering stats

```
SQL>  SELECT column_name, num_distinct, num_buckets, histogram
FROM USER_TAB_COL_STATISTICS where table_name='BENCH';
COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
KSEQ                                 990565           1 NONE
K500K                                432419           1 NONE
K250K                                245497           1 NONE
K100K                                 92975           1 NONE
K40K                                  39125           1 NONE
K10K                                   9997           1 NONE
K1K                                     997           1 NONE
K100                                    101         100 FREQUENCY
K25                                      25          25 FREQUENCY
K10                                      10          10 FREQUENCY
K5                                        5           5 FREQUENCY
…
```

**Oracle figures that the default RF of 1/num_distinct will be good enough for k1k and up**

## Simple plan example

```
SQL> explain plan for select max(s1) from bench where k500k=2;
SQL> select plan_table_output from table(dbms_xplan.display(NULL,null,'TYPICAL'));

| Id  | Operation                     | Name    | Rows  | Bytes |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     1 |    14 |
|   1 |  SORT AGGREGATE               |         |     1 |    14 |
|   2 |   TABLE ACCESS BY INDEX ROWID| BENCH   |     2 |    28 |
|*  3 |    INDEX RANGE SCAN           | K500KIN |     2 |
 -------------------------------------------------------------------

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, k100 case

```
SQL> explain plan for select max(s1) from bench where k100=2;
SQL> select plan_table_output from
table(dbms_xplan.display(NULL,null,'TYPICAL'));
| Id  | Operation           | Name  | Rows  | Bytes |
---------------------------------------------------------
|   0 | SELECT STATEMENT    |       |     1 |    12 |
|   1 |  SORT AGGREGATE     |       |     1 |    12 |
|*  2 |   TABLE ACCESS FULL| BENCH | 10320 |   120K|
---------------------------------------------------------
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.
- If no histograms are generated, Oracle shows a plan using the index, not a good idea.

## 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/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 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 10x faster.
- That's assuming disk. On SSD, use the index.

## Easier way: `set autotrace on explain statistics`

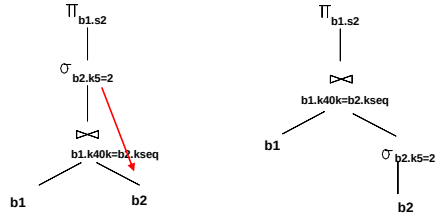
- Or just **`set autotrace on exp stat`**
- Also **`set timing on`**
- Also **`set line 130`** to avoid wrapping
- Then just select …
- After this returns, you see an abbreviated explain plan, plus actual statistics on the query
- The explain plan is not guaranteed to be the exact plan used

## Example with set autotrace …

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

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=1 Bytes=14)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'BENCH' (TABLE) (Cost=6 Card=2 Bytes=28)
   3    2       INDEX (RANGE SCAN) OF 'K500KIN' (INDEX) (Cost=3 Card=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 indexes on k40k and kseq)

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




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

Oracle uses Hash Join:

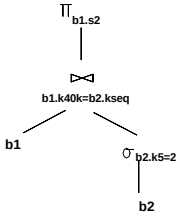
```
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |     1 |    34 |       | 16908   (7)| 00:03:23 |
|   1 |  SORT AGGREGATE     |       |     1 |    34 |       |            |          |
|*  2 |   HASH JOIN         |       |  1000K|    32M|  3792K| 16908   (7)| 00:03:23 |
|*  3 |    TABLE ACCESS FULL| BENCH |   193K|  1514K|       |  7002   (9)| 00:01:25 |
|   4 |    TABLE ACCESS FULL| BENCH |  1000K|    24M|       |  6751   (5)| 00:01:22 |
-----------------------------------------------------------------------------------------------

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

Line 3: 100000/5 = 200K rows, each with kseq, say 10 bytes, = 2000K = 2M bytes, OK

Line 4: all rows, drop all cols except k40k and s2, say 20 bytes = 20M bytes, OK

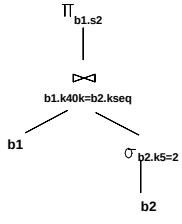
- This hash table is using the temp tablespace instead of memory. Apparently, (by Burleson), Oracle only uses 5% of pga_aggregate_target for memory hash tables, and that's only .05*24M = 1.2MB for our server. Here the HT holds 1.5MB data, apparently needs 3.8MB space.
- This query does about 2400 writes while doing the hash join. Not great. Note the cost being more than 2 table scans.
- The cost figures here are obsolete. They changed downwards after I ran the procedure to gather system stats. However the plans didn't change.

## Hash Join Cost Analysis, case of in-memory HT



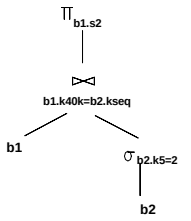

- Hash Join: 1M rows (24MB) from b1, 200K rows (1.5MB) from selection,
- So build hash table from b2, should fit in memory (apparently 3.8MB) if raise pga_aggregate_target to > 4*20 = 80MB.
- Hash the 1M rows and output to pipeline
- i/o Cost: read table twice, about 60,000 i/os. More if hash table can't fit in memory. Less if table fits in memory.
- Mysql can't do hash join, MariaDB can

## Hash Join Cost Analysis, by textbook



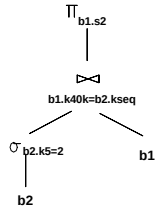

- Hash Join: 1M rows (24MB) from b1, 200K rows (1.5MB) from selection,
- Book assumes partitioning needed first. Suppose only 1MB of memory available.
- So read and write all data of both tables into partitions, say 100 partitions (using 800KB of buffers)
  - For each partition, build hash table from b2, should fit in memory (about .01(3.8M) = 38KB)
  - Hash the 10K rows and 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).

## Hash Join Cost Analysis, with Oracle using temp space ("hybrid hash join" of pg. 465)



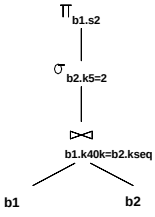

- Hash Join: 1M rows (24MB) from b1, 200K rows (1.5MB) from selection,
- Textbook method uses 3200 writes for HJ, but Oracle only used 2400—how is that possible?
- Oracle partitions all b1 data and builds one partition's HT in memory in first pass (or maybe several HTs)
- While reading b2 side data, does join with in-memory partition(s), writes out other partitions
- i/o Cost: read both tables, about 60,000 i/os. Write and read *part* of incoming tables to HJ: 2400 writes and reads instead of 3200.
- Cost = 64,800 i/os.
- This way, cost of HJ doesn't jump up as join size crosses need-temp-space boundary.

## Nested Loops Cost Analysis, b2 outer



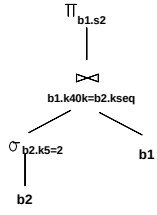

- Indexed NL Join: 200K rows (1.5 MB) from selection, 1M rows (24 MB) in b1 using index on k40k
- Cost: about 25 matches for kseq value, 200K*25 index probes (not really), 5M i/os plus reading table b2. No good.
- Here we are using one i/o for each B-tree probe, assuming its upper-level pages are in memory.

## Nested Loops Cost Analysis, b1 outer



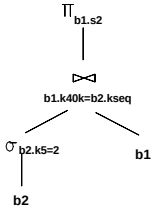

- Indexed NL Join: 1M rows in b2 with index on kseq, 1M rows (20MB) in b1
- Cost: 1 match for each k40k value, 1M index probes, but to only 40K different spots, so 40K i/os assuming decent buffering, plus reading b1 table (about 30,000 i/os)
- Best indexed NLJ plan, cost = 70,000, unless table fits in memory.
- Compare to HJ costs: 60,000 (in-mem HT), 66,400 (partitioning) 64,800 (hybrid)
- HJ also benefits from using sequential i/o:
  - NLJ: 30,000 seq + 40,000 random i/os
  - HJ: 60,000-63,800 seq (much faster)

## Paged Nested Loops Cost Analysis




- Paged NL Join: 200K rows (1.5 MB = 190 pages) from selection
- Then read on page of left-side input, read all of b1, then another page, read all of b1.
- Cost: read b1 256 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 (20MB) 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.

## Oracle loves hash joins

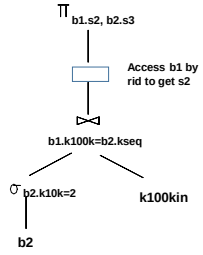
Most joins I tried with the bench table and itself (self-joins) used hash joins.

Here is one that used nested loops:

## Oracle chooses a NLJ

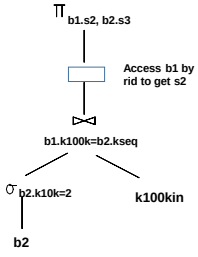
```
SQL>  explain plan for select max(b1.s2), max(b2.s3) from
bench b1, bench b2 where b1.k100k=b2.kseq and b2.k10k=2;

| Id  | Operation                      | Name    | Rows  | Bytes |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT               |         |     1 |    56 |
|   1 |  SORT AGGREGATE                |         |     1 |    56 |
|   2 |   TABLE ACCESS BY INDEX ROWID  | BENCH   |    11 |   286 |
|   3 |    NESTED LOOPS                |         |  1076 | 60256 |
|   4 |     TABLE ACCESS BY INDEX ROWID| BENCH   |   100 |  3000 |
|*  5 |      INDEX RANGE SCAN          | K10KIN  |   100 |       |
|*  6 |     INDEX RANGE SCAN           | K100KIN |    11 |       |
-------------------------------------------------------------------
Predicate Information (identified by operation id):
   5 - access("B2"."K10K"=2)
   6 - access("B1"."K100K"="B2"."KSEQ")
```

## NLJ Plan




- Outer table is accessed by k10k index to find k10k=2, about 100 rows. Get b2.kseq, b2.s3, using 100 i/os plus one or two for index pages.
- For each resulting kseq, find about 10 rows of 100kin matching this kseq, join. This is 1000 index probes, so about 1000 i/os.
- Now have 1000 rows: (b2.kseq, b2.s3, b1.rid)
- Use b1.rid to get b1.s2. 1000 i/os.
- Total: 2100 i/os.
- Note: this trick of going back to the table *after* the join to get more column values is not covered in the book. It is only useful if the join produces a moderate number of rows.

## Try Hash Join Plan for this query




- B2 table is accessed by k10k index to find k10k=2, about 100 rows. Get b2.kseq, b2.s3, using 100 i/os plus one or two for index pages
- Hash smaller table from b2 using kseq.
- Read k100kin for right table. 10% (30,000) = 3000 i/os (may be over-estimate*)
- Check k100k values by hashing on k100k, see if match, saving b1.rid
- Use b1.rid to get b1.s2. 1000 i/os.
- Total: 4100 i/os.
- Compare to 2100 for NLJ.

*Oracle would know actual index size:

Select leaf_blocks from user_indexes where index_name = 'K100KIN'; returns 2096

## Why NLJ here?

- We have the needed indexes on join columns for indexed NLJ
- We see the outer table going into the join is not huge
- Specifically, the number of rows from the outer table stream (100), multiplied by the #duplicates in the index (10), isn't high (like 10,000), so all those index probes don't add up too high.
- 10,000 index probes cost at least 10,000 i/os, 100 seconds at 100 i/o/s.

## 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)

Good article: http://use-the-index-luke.com/sql/join/hash-join-partial-objects (ignore the last part about objects)

- 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 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 employess, look at one department's.

## What about mysql?

- Mysql v 5.6 (our case) only joins using nested loops join, including blocked nested loops.
- MariaDB 5.5 (current) uses hash join too
- Mysql/Innodb only uses one index (per table) for a query, according to https://dev.mysql.com/doc/refman/5.6/en/optimizing-innodb-queries.html (I added "per table" based on examples). But this isn't strictly true, as we will see.
- Mysql has "explain", but it is not as complete or easy to understand as Oracle's.

## Mysql EXPLAIN

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

+----+-------------+-------+------+---------------+---------+---------+-|
  id | select_type | table | type | possible_keys | key     | key_len | …|
+----+-------------+-------+------+---------------+---------+---------+-|
   1 | SIMPLE      | bench | ref  | k500kin,k4in  | k500kin | 4       | |
+----+-------------+-------+------+---------------+---------+---------+--+
1 row in set (0.00 sec)
```

- 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
--------------

+----+-------------+-------+-------------+---------------+---------------+---------+-----+
| id | select_type | table | type        | possible_keys | key           | key_len | ref |
rows | Extra                                                |
+----+-------------+-------+-------------+---------------+---------------+---------+-----+-
|  1 | SIMPLE      | bench | index_merge | k500kin,k10kin | k500kin,k10kin | 4,4     | NULL |
1 | Using intersect(k500kin,k10kin); Using where |
+----+-------------+-------+-------------+---------------+---------------+---------+-----+
```

- Shows index merge of two indexes.
- Though not really worth it: only 2 rows satisfy k500k=2

## 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.

## Try a simple join to get started.

```
mysql> explain select max(b1.s2), max(b2.s2) from bench b1, bench b2
where b1.k10k=b2.kseq;
--------------

+----+-------------+-------+--------+---------------+---------+---------+
| id | select_type | table | type   | possible_keys | key     | key_len |
                                      |ref            | rows    | Extra |
+----+-------------+-------+--------+---------------+---------+---------+
|  1 | SIMPLE      | b1    | ALL    | k10kin        | NULL    | NULL    |
                                     |NULL           | 985592  | NULL  |

|  1 | SIMPLE      | b2    | eq_ref | PRIMARY       | PRIMARY | 4       |
                                     |eoneilldb.b1.K10K |      1 | NULL  |
+----+-------------+-------+--------+---------------+---------+---------+
```

- Here b1 is the left (outer) table and b2 is the right (inner) table, with no index used for b1, and the PRIMARY index used for access to b2, matched against b1.K10K.
- This involves 1M index probes into the PRIMARY index, though actually only 10K different values, so 10K i/os needed if buffering can hold 10K index rows.

## Add a predicate on one table: see it's made outer, with index use for it

```
mysql> explain select max(b1.s2), max(b2.s3) from bench b1,
bench b2 where b1.k100k=b2.kseq and b2.k10k=2;
+----+-------------+-------+------+---------------+---------+
| id | select_type | table | type | possible_keys  | key
                  | key_len | ref              | rows | Extra |
+----+-------------+-------+------+---------------+---------+-
|  1 | SIMPLE      | b2    | ref  | PRIMARY,k10kin | k10kin  |
                  4       | const            |   98 | NULL  |

|  1 | SIMPLE      | b1    | ref  | k100kin        | k100kin |
                  4       | eoneilldb.b2.KSEQ |    4 | NULL  |
+----+-------------+-------+------+---------------+---------+-
```

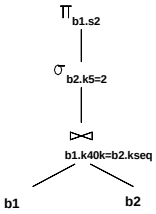
- This is the join that Oracle used NLJ for (slide 16)
- Same join order and use of k10kin on left as Oracle
- Oracle used index scan instead of table scan on right, then rid lookup to get b1.s2, saving about 24,000 i/os.

## Try the join that Oracle wanted hash join for

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

+----+-------------+-------+--------+---------------+---------+---------
| id | select_type | table | type   | possible_keys | key     | key_len
                                    | ref           | rows    | Extra     |
+----+-------------+-------+--------+---------------+---------+---------
|  1 | SIMPLE      | b1    | ALL    | NULL          | NULL    | NULL
                                   | NULL          | 985592  | NULL      |

|  1 | SIMPLE      | b2    | eq_ref | PRIMARY       | PRIMARY | 4
                                   | eoneilldb.b1.K40K |      1 | Using where |
+----+-------------+-------+--------+---------------+---------+---------
```

- "Using where" just means the found rows are checked for the b2.k5 condition before being sent along.
- mysql uses 16KB pages, so the table is only 15,000 pages
- Cost = 15,000 + 40K index probes = 55K i/os (see next slide)
- Compare to Oracle's hash join (slide 11): 60K seq i/os.
- No sign of blocked NLJ here (which does have an "Extra" notation)

## Nested Loops Cost Analysis, b1 outer (copy of slide 15)




- 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 40K different spots, so 40K i/os assuming decent buffering, plus reading outer table (about 30,000 i/os using 8KB pages)
- Best indexed NLJ plan, cost = 70,000, unless table fits in memory.
- Actually, mysql uses 16KB pages, so the table is only 15,000 pages, so cost = 15+ 40K = 55K

## Execution time for this query

Mysql with no OS buffering (`innodb_flush_method = O_DIRECT` in my.cnf):

- Query took 5.4 s on mysql with 500M buffer pool. Table scan takes 4.1 secs, and remaining 1.3 s represents processing.
- Query took 20 s with 24 M buffer pool, 5x the table scan time of 4 s. 240MB/4s = 60 MB/s sequential i/o rate (compare to 100MB/s rule)
- Query took 426 s (7+ min) with 10M buffer pool
  - 40K/422s = 95 op/s, close to 100 op/s expected for random i/o and reported for this disk.

Oracle

- Query took 33 s with 24MB database buffers. Table scan takes 5.1 s, so this is 6x cost of table scan. Buffer cleared by scan of another big table.

Appears to be a tie: although Oracle appears to be smarter, it doesn't make much difference to this. Should be much faster with larger pga_aggregate_target.

## Fixing a query plan: Oracle query hints

- Example from Oracle 10g doc: index to use, or don't use this index:

```
SELECT /*+ INDEX (employees emp_department_ix)*/
    employee_id, department_id FROM employees
  WHERE department_id > 50;

SELECT /*+ NO_INDEX(employees emp_empid) */
    employee_id FROM employees
  WHERE employee_id > 200;
```

- Also there are hints for join order, and specify/avoid use of NL, HJ, sort-merge join, and many other things.
- Mysql has index hints, and "optimizer switches"

## 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(salrycat);  (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 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.