Query Tuning Using Advanced Hints

NYOUUG Meeting
December 12, 2002

Claudio Fratarcangeli

Adept Technology Inc.
claudiof@computer.org
PUSH_PRED Hint

- Applicable when doing an outer join to a view
- Normally, outer join predicate is evaluated after view is materialized
- PUSH_PRED forces the outer join predicate between table and view to be pushed into the view
Outer Join to a View without PUSH_PRED Hint

Optimizer steps

- Execute view query independently

2. Materialize view results in an internal temporary table

- Outer Join to the view using sort merge or hash join
  
  Index is not available for joining to temporary table
  
  Full scan of temporary table required
Outer Join to a View with PUSH_PRED Hint

Optimizer steps

- Modify original statement by inserting outer join predicate inside of view statement

- Execute join
  
  If join column in view is indexed then an indexed nested loop join can be performed
PUSH_PRED Hint Sample Tables

CUSTOMERS

ORDERS

LINE_ITEMS

PRODUCTS
PUSH_PRED Hint Sample Tables

CREATE TABLE products (  
  product_id NUMBER(9),  
  product_name VARCHAR2(30),  
  product_descr VARCHAR2(4000),  
  instock VARCHAR2(1));

PK(Unique Index): product_id

CREATE TABLE orders (  
  order_id NUMBER(9),  
  customer_id NUMBER(6));

PK: order_id  
Non-unique index: order_id,customer_id

CREATE TABLE line_items (  
  order_id NUMBER(9),  
  product_id NUMBER(9),  
  comments VARCHAR2(80))

PK(Unique index): product_id, order_id

CREATE TABLE customers (  
  customer_id NUMBER(6),  
  customer_name VARCHAR2(30));

PK(Unique Index): customer_id
PUSH_PRED Hint Query

SELECT p.product_name, l.product_id
FROM
  products P,
  (SELECT li.product_id
   FROM line_items li, orders o
   WHERE li.order_id = o.order_id) l
WHERE p.product_id = l.product_id(+)

Query Plan without Hint

SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN (OUTER)
  TABLE ACCESS (FULL) OF 'PRODUCTS'
  VIEW
    HASH JOIN
      TABLE ACCESS (FULL) OF 'ORDERS'
      TABLE ACCESS (FULL) OF 'LINE_ITEMS'
Query with PUSH_PRED Hint

```
SELECT /*+ PUSH_PRED(l) */ p.product_name, l.product_id
FROM
  products P,
  (SELECT li.product_id
   FROM line_items li, orders o
   WHERE li.order_id = o.order_id) l
WHERE p.product_id = l.product_id(+)
```

Query Plan unchanged despite Hint

```
SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN (OUTER)
  TABLE ACCESS (FULL) OF 'PRODUCTS'
  VIEW
    HASH JOIN
      TABLE ACCESS (FULL) OF 'ORDERS'
      TABLE ACCESS (FULL) OF 'LINE_ITEMS'
```
Parameters that Control Pushing Predicate into View

- PUSH_PRED Hint doesn't work in 8i

- Fixed in 9i

- In 8i need to use undocumented initialization parameter
  `_PUSH_JOIN_PREDICATE`

- Set either in init.ora file or with ALTER SESSION statement

- `ALTER SESSION SET "_PUSH_JOIN_PREDICATE" = TRUE`
Query with PUSH_PRED Hint

ALTER SESSION SET "_PUSH_JOIN_PREDICATE" = TRUE;

SELECT /*+ PUSH_PRED(l) */ p.product_name, l.product_id
FROM
  products P,
  (SELECT li.product_id
   FROM line_items li, orders o
   WHERE li.order_id = o.order_id) l
WHERE p.product_id = l.product_id(+)

Query Plan still not changed

SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN (OUTER)
  TABLE ACCESS (FULL) OF 'PRODUCTS'
  HASH JOIN
  TABLE ACCESS (FULL) OF 'ORDERS'
  TABLE ACCESS (FULL) OF 'LINE_ITEMS'
Index Statistics Affect Pushing Predicate into View

- Optimizer assumes full table scan is cheaper than using index on LINEITEMS(PRODUCT_ID,ORDER_ID)

- Need to convince optimizer that cost of using index on LINEITEMS(PRODUCT_ID,ORDER_ID) is less than cost of full table scan

- Use initialization parameter:

  OPTIMIZER_INDEX_COST_ADJ
OPTIMIZER_INDEX_COST_ADJ
Initialization Parameter

- Parameter may be set either in init.ora file or using ALTER SESSION statement
- Range of values: 1 .. 10000
- Default value: 100
- Lower values tell optimizer that cost of using an index is lower
- Lower values cause optimizer to favor use of indexes
Query with PUSH_PRED Hint

```
ALTER SESSION SET "_PUSH_JOIN_PREDICATE" = TRUE;
ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 1;

SELECT /*+ PUSH_PRED(l) */ p.product_name, l.product_id
FROM
  products P,
  (SELECT li.product_id FROM line_items li, orders o
   WHERE li.order_id = o.order_id) l
WHERE p.product_id = l.product_id(+)
```

Query Plan

```
SELECT STATEMENT
  NESTED LOOPS (OUTER)
    TABLE ACCESS (FULL) OF 'PRODUCTS'
    VIEW PUSHED.Predicate
      NESTED LOOPS
        INDEX (RANGE SCAN) OF 'LINE_ITEMS_PK' (UNIQUE)
        INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
```
Practical Example of PUSH_JOIN Hint
Practical Example of PUSH JOIN Hint

CREATE TABLE products (  
  product_id NUMBER(9),  
  product_name VARCHAR2(30),  
  product_descr VARCHAR2(4000),  
  instock VARCHAR2(1));

PK(Unique Index): product_id

CREATE TABLE orders (  
  order_id NUMBER(9),  
  customer_id NUMBER(6));

PK: order_id
Non-unique index: order_id,customer_id

CREATE TABLE line_items (  
  order_id NUMBER(9),  
  product_id NUMBER(9),  
  comments VARCHAR2(80))

PK(Unique index): product_id, order_id

CREATE TABLE customers (  
  customer_id NUMBER(6),  
  customer_name VARCHAR2(30));

PK(Unique Index): customer_id
Practical Example of PUSH_JOIN Hint

- Query is executed by a currently logged in customer
- Find products where
  
  Product description matches some keyword search criteria

  AND

  Product is either in stock OR
  has been bought anytime in the past by the customer

- For each matching product:
  
  Show in stock status and whether or not it has been bought by current customer

- Sort matching products with most relevant first
- Show 150 most relevant matching products only
- Use Intermedia Text to do the text search matching
Practical Example of PUSH_JOIN Hint

Example query,

Find products with product description containing text, "SUN"

Sample Output

<table>
<thead>
<tr>
<th>PRODUCT NAME</th>
<th>SCORE</th>
<th>IN STOCK</th>
<th>BOUGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product 1</td>
<td>9</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Product 2</td>
<td>9</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Product 3</td>
<td>9</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Product 4</td>
<td>8</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Sample Tables

Number of Rows in Sample Tables

PRODUCTS – 20000
ORDERS – 58000
LINE_ITEMS – 58000
Sample Query: First Attempt

```
SELECT product_name, scor, instock, purchased
FROM
  (SELECT /*+ ORDERED USE_NL(li,o) */
   SCORE(10) scor, product_name, instock,
   DECODE(o.order_id,NULL,'N','Y') purchased
   FROM products p,
   line_items li,
   orders o
   WHERE
     CONTAINS(product_descr, 'SUN', 10) > 0
     AND p.product_id = li.product_id(+)
     AND li.order_id = o.order_id(+)
     AND o.customer_id(+) = 999
     AND (p.instock = 'Y' OR o.order_id IS NOT NULL)
   GROUP BY p.product_name, SCORE(10), li.product_id,
   instock
   ORDER BY SCORE(10) DESC
  )
WHERE ROWNUM < 151
```
## Sample Query: First Attempt Query Plan/Stats

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.09</td>
<td>0.09</td>
<td>0</td>
<td>94</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.02</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>11</td>
<td>3.00</td>
<td>3.03</td>
<td>0</td>
<td>28153</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>13</td>
<td>3.11</td>
<td>3.14</td>
<td>0</td>
<td>28247</td>
<td>0</td>
<td>150</td>
</tr>
</tbody>
</table>

```sql
0  SELECT STATEMENT  GOAL: CHOOSE
150  COUNT (STOPKEY)
150  VIEW
150  SORT (GROUP BY STOPKEY)
19702  FILTER
19702  NESTED LOOPS (OUTER)
19703  NESTED LOOPS (OUTER)
201  TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'
201  DOMAIN INDEX OF 'PRODUCT_TEXT'
19702  INDEX (RANGE SCAN) OF 'LINE_ITEMS_PK' (UNIQUE)
19500  INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
```
Sample Query: Second Attempt

- First find 150 most relevant PRODUCTS that will match criteria
- Then join to LINE_ITEMS
- Number of PRODUCTS and LINE_ITEMS rows joined is fewer
- Less rows accessed/ Better performance
SELECT /*+ ORDERED USE_NL(li,ord) */ scor, product_name, DECODE(ord.order_id, NULL, 'N', 'Y') purchased, instock FROM (SELECT scor, product_name, product_id, instock FROM (SELECT SCORE(10) scor, product_name, product_id, instock FROM products p WHERE CONTAINS(product_descr, 'SUN', 10) > 0 AND instock = 'Y' OR EXISTS (SELECT /*+ ORDERED USE_NL(ord) */ 1 FROM line_items li, orders ord WHERE ord.customer_id = 999 AND li.order_id = ord.order_id AND li.product_id = p.product_id AND ROWNUM = 1 ) ) ) ORDER BY SCORE(10) DESC WHERE ROWNUM < 151 p, line_items li, orders ord WHERE p.product_id = li.product_id(+) AND li.order_id = ord.order_id(+) AND ord.customer_id(+) = 999 GROUP BY scor, product_name, li.product_id, instock ORDER BY scor DESC
## Sample Query: Second Attempt Query Stats

<table>
<thead>
<tr>
<th>Call</th>
<th>Count</th>
<th>CPU</th>
<th>Elapsed</th>
<th>Disk</th>
<th>Query</th>
<th>Current</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.08</td>
<td>0.09</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>11</td>
<td>0.42</td>
<td>0.56</td>
<td>0</td>
<td>18140</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>0.51</td>
<td>0.66</td>
<td>0</td>
<td>18140</td>
<td>0</td>
<td>150</td>
</tr>
</tbody>
</table>
Sample Query: Second Attempt Query Plan

0 SELECT STATEMENT  GOAL: CHOOSE
150 SORT (GROUP BY)
14752 NESTED LOOPS (OUTER)
14753 NESTED LOOPS (OUTER)
151 VIEW
151 COUNT (STOPKEY)
150 VIEW
150 SORT (ORDER BY STOPKEY)
200 FILTER
201 TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'
201 DOMAIN INDEX OF 'PRODUCT_TEXT'
0 COUNT (STOPKEY)
0 NESTED LOOPS
0 INDEX (RANGE SCAN) OF 'LINE_ITEMS_PK' (UNIQUE)
0 INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
14752 INDEX (RANGE SCAN) OF 'LINE_ITEMS_PK' (UNIQUE)
14650 INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
Sample Query: Final Attempt

- Do not join to LINE_ITEMS to determine if customer has bought product
- Join to PRODUCTS table again and do correlated subquery to see if there exist any LINES_ITEMS for this product and customer
- Reduces the number of LINE_ITEMS rows that are accessed
- Better performance
SELECT /*+ PUSH_PRED(op) */ scor, product_name, 
    DECODE(op.product_id, NULL, 'N', 'Y') purchased, instock
FROM
  (SELECT scor, product_name, product_id, instock
   FROM
     (SELECT /*+ FIRST_ROWS */ score(10) scor, product_name, product_id, instock
     FROM products p
     WHERE CONTAINS(product_descr, 'SUN', 10) > 0
       AND (instock = 'Y' OR
             EXISTS
               ( SELECT /*+ ORDERED USE_NL(ord) */ 1
                   FROM line_items li, orders ord
                   WHERE
                     ord.customer_id = 999 AND li.order_id = ord.order_id
                     AND li.product_id = p.product_id AND rownum = 1
               )
     )
   )
  )
ORDER BY SCORE(10) DESC
WHERE ROWNUM < 151
) p,
  (SELECT product_id FROM products p
   WHERE EXISTS
     ( SELECT /*+ ORDERED USE_NL(ORD) */ 1
         FROM line_items li, orders ord
         WHERE ord.customer_id = 999 AND li.order_id = ord.order_id
         AND li.product_id = p.product_id AND rownum = 1
     )
  ) op
WHERE p.product_id = op.product_id(+)
ORDER BY scor DESC
### Final Attempt Query Plan/Stats / No Hint

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>1.63</td>
<td>1.67</td>
<td>0</td>
<td>94</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.22</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>11</td>
<td>1.46</td>
<td>1.48</td>
<td>0</td>
<td>132607</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>total</td>
<td>13</td>
<td>3.31</td>
<td>3.37</td>
<td>0</td>
<td>132701</td>
<td>0</td>
<td>150</td>
</tr>
</tbody>
</table>
Final Attempt Query Plan / No Hint

0  SELECT STATEMENT   GOAL: HINT: FIRST_ROWS
150  SORT (ORDER BY)
150  HASH JOIN (OUTER)
150  VIEW
150   COUNT (STOPKEY)
150  VIEW
150   FILTER
150    TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'
150    DOMAIN INDEX OF 'PRODUCT_TEXT'
0    COUNT (STOPKEY)
0    NESTED LOOPS
0     INDEX (RANGE SCAN) OF 'LINE_ITEMS_PROD' (NON-UNIQUE)
0     INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
200  VIEW
200   INDEX (FULL SCAN) OF 'PRODUCT_PK' (UNIQUE)
20000  COUNT (STOPKEY)
200   NESTED LOOPS
56002  INDEX (RANGE SCAN) OF 'LINE_ITEMS_PROD' (NON-UNIQUE)
200  INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
# Final Attempt Query Plan/Stats / With Hint

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>11</td>
<td>0.07</td>
<td>0.07</td>
<td>0</td>
<td>1392</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>13</td>
<td>0.09</td>
<td>0.09</td>
<td>0</td>
<td>1392</td>
<td>0</td>
<td>150</td>
</tr>
</tbody>
</table>
Final Attempt Query Plan / With Hint

0 SELECT STATEMENT  GOAL: HINT: FIRST_ROWS
150 SORT (ORDER BY)
150 NESTED LOOPS (OUTER)
151 VIEW
151 COUNT (STOPKEY)
150 VIEW
150 FILTER
150 TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'
150 DOMAIN INDEX OF 'PRODUCT_TEXT'
0 COUNT (STOPKEY)
0 NESTED LOOPS
0 INDEX (RANGE SCAN) OF 'LINE_ITEMS_PK' (UNIQUE)
0 INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
150 VIEW PUSHED PREDICATE
150 INDEX (UNIQUE SCAN) OF 'PRODUCT_PK' (UNIQUE)
150 COUNT (STOPKEY)
150 NESTED LOOPS
330 INDEX (RANGE SCAN) OF 'LINE_ITEMS_PK' (UNIQUE)
150 INDEX (RANGE SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
HASH_AJ Hint

- Use for NOT IN queries where there is no index on column in nested query
- Find rows in table A where there are no matching rows in table B
- Find CUSTOMERS who have no ORDERS
- There is no index on ORDERS.CUSTOMER_ID
SELECT count(*) FROM customers
WHERE customer_id NOT IN
  (SELECT customer_id FROM orders)
NOT IN / No Hint Query Stats/Plan

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>67.16</td>
<td>67.24</td>
<td>0</td>
<td>141975</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>67.16</td>
<td>67.24</td>
<td>0</td>
<td>141975</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

0 SELECT STATEMENT GOAL: CHOOSE
1 SORT (AGGREGATE)
998 FILTER
1000 INDEX (FAST FULL SCAN) OF 'CUSTOMER_PK' (UNIQUE)
999 INDEX (FULL SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
NOT IN Example / No Hint

- For each row in CUSTOMERS do a full scan of ORDERS to find a matching row

- 999 CUSTOMERS rows times 58000 ORDERS rows \( \approx 58,000,000 \)

- \( \approx 1000 \times 58000 = 58,000,000 \) row accesses required to execute this query

- This explains high number of block accesses

- Very expensive
NOT IN Example / With HASH_AJ Hint

```
SELECT count(*) FROM customers
WHERE customer_id NOT IN
  (SELECT /*+ HASH_AJ */ customer_id
   FROM orders)
```
### Query Plan/Stats

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>0.14</td>
<td>0.14</td>
<td>0</td>
<td>118</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>4</strong></td>
<td><strong>0.15</strong></td>
<td><strong>0.15</strong></td>
<td><strong>0</strong></td>
<td><strong>118</strong></td>
<td><strong>8</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

- 0 SELECT STATEMENT
- 1 SORT (AGGREGATE)
- 998 HASH JOIN (ANTI)
- 999 INDEX (FAST FULL SCAN) OF 'CUSTOMER_PK' (UNIQUE)
- 58000 VIEW OF 'VW_NSO_1'
- 58000 TABLE ACCESS (FULL) OF 'ORDERS'
NOT IN Example / With HASH_AJ Hint

- HASH_AJ Hint forces optimizer to perform a HASH ANTI-JOIN
- HASH ANTI-JOIN is similar to HASH JOIN
- Instead of finding matching rows, find rows in one table with no matching rows in other table
- Each table in HASH ANTI-JOIN is scanned only once
- 999 CUSTOMERS rows + 58000 ORDERS rows = 58999 row accesses
- 58,999 is far less than 58,000,000 row accesses without HASH_AJ hint
- This explains low number of block accesses in stats
HASH_AJ Prerequisites

- Join columns must be NOT NULL
- Assume ORDERS.CUSTOMER_ID or LINE_ITEMS.CUSTOMER_ID were NULLABLE
- Query must be rewritten as

```sql
SELECT count(*) FROM customers
WHERE customer_id IS NOT NULL AND
   customer_id NOT IN
   (SELECT /*+ HASH_AJ */ customer_id
    FROM orders
    WHERE customer_id IS NOT NULL)
```
NOT EXISTS

- MYTH:
  NOT EXISTS always better than NOT IN

- Assume ORDERS.CUSTOMER_ID is indexed

- 58000 rows in CUSTOMERS table

  
  ```sql
  SELECT count(*)
  FROM customers c
  WHERE NOT EXISTS
    (SELECT 1 FROM orders o
     WHERE o.customer_id = c.customer_id);
  ```
NOT EXISTS Versus NOT IN With HASH_AJ

NOT Exists with Index on ORDERS.CUSTOMER_ID

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>1.63</td>
<td>1.63</td>
<td>0</td>
<td>116110</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>1.64</td>
<td>1.64</td>
<td>0</td>
<td>116110</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

NOT IN with HASH_AJ Hint

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>0.35</td>
<td>0.36</td>
<td>0</td>
<td>225</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>0.35</td>
<td>0.36</td>
<td>0</td>
<td>225</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
NOT EXISTS Versus NOT IN with HASH_AJ

- Case 1: No Index on Inner Table
  - NOT IN with HASH_AJ is better

- Case 2: Index on Inner Table
  - Number of rows in Outer Table is Large
  - NOT IN with HASH_AJ is better
  - Example: 58000 rows in CUSTOMERS table

- Case 3: Index on Inner Table
  - Number of rows in Outer Table is Small
  - Number of rows in Inner Table is Large
  - NOT EXISTS is usually better

- Size of Inner Table may also affect result
  - NOT EXISTS may perform better for large indexed inner table
HASH_SJ Hint

- Use for CORRELATED EXISTS queries where there is no index on column in nested query

- Find rows in table A where there exist matching rows in table B

- Find CUSTOMERS who have one or more ORDERS

- There is no index on ORDERS.CUSTOMER_ID
HASH_SJ Example

```
SELECT count(*) FROM customers c
WHERE EXISTS
  (SELECT 1 /*+ HASH_SJ */ FROM orders o
   WHERE c.customer_id = o.customer_id)
```
### HASH_SJ Example Query Stats/Plan

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.02</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>0.12</td>
<td>0.12</td>
<td>0</td>
<td>118</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>0.14</td>
<td>0.17</td>
<td>0</td>
<td>118</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

0 SELECT STATEMENT GOAL: CHOOSE

1 SORT (AGGREGATE)

1 HASH JOIN (SEMI)

999 INDEX (FAST FULL SCAN) OF 'CUSTOMER_PK' (UNIQUE)

58000 TABLE ACCESS (FULL) OF 'ORDERS'
HASH_SJ Example

- HASH_SJ Hint forces optimizer to perform a HASH SEMI-JOIN
- HASH SEMI-JOIN is similar to HASH JOIN
- Unlike true HASH JOIN, only one row from outer table is returned even if there is more than one matching row from inner table
- Each table in HASH SEMI-JOIN is scanned only once
- 999 CUSTOMERS rows + 58000 ORDERS rows = 58999 row accesses
- This are far less than number of row accesses required without HASH_SJ hint
- This explains low number of block accesses in stats
Correlated EXISTS with Index on Inner Table

- Example:

  ```sql
  SELECT count(*) FROM customers c
  WHERE EXISTS
    (SELECT 1 FROM orders o
     WHERE c.customer_id = o.customer_id)
  ```

- Index on ORDERS.CUSTOMER_ID

- HASH_SJ Hint can yield better performance if number of rows in outer table (CUSTOMERS) is large
HASH_AREA_SIZE Init Parameter

- HASH_AREA_SIZE determines amount of memory allocated for HASH ANTI-JOIN, HASH SEMI-JOIN and HASH JOIN.

- Set HASH_AREA_SIZE appropriately when using HASH_AJ or HASH_SJ Hint on large tables

- Settable via
  - init.ora file
  - ALTER SYSTEM SET HASH_AREA_SIZE=
  - ALTER SESSION SET HASH_AREA_SIZE=
PUSH_SUBQ Hint

- Use to force evaluation of correlated subqueries as early as possible in a query involving joins
- Optimizer usually evaluates subqueries after evaluating joins
- PUSH_SUBQ hint causes optimizer to evaluate subqueries before evaluating all joins
PUSH_SUBQ Hint Example

- Table

  CREATE TABLE shipments (  
  shipment_id NUMBER(6),  
  order_id NUMBER(6),  
  shipment_date DATE);  

  PK(Unique Index): shipment_id  

  Index On: order_id

- Index

  Index on line_items(order_id,product_id)
PUSH_SUBQ Example / No Hint

Find all orders and order lines for orders that have shipped

```
SELECT /*+ ORDERED USE_NL(L,P) */ o.order_id,
    l.comments, p.product_name
FROM orders o, line_items l, products p
WHERE o.order_id = l.order_id
    AND l.product_id = p.product_id
    AND EXISTS
    (SELECT 1
     FROM shipments s
     WHERE s.order_id = o.order_id)
    AND o.order_id < 50000;
```
### PUSH_SUBQ Example / No Hint: Query Stats

<table>
<thead>
<tr>
<th>Call</th>
<th>Count</th>
<th>CPU</th>
<th>Elapsed</th>
<th>Disk</th>
<th>Query</th>
<th>Current</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>1000</td>
<td>5.03</td>
<td>5.06</td>
<td>0</td>
<td>417785</td>
<td>4</td>
<td>999</td>
</tr>
</tbody>
</table>

**Total**
- Count: 1002
- CPU: 5.03
- Elapsed: 5.06
- Disk: 0
- Query: 417785
- Current: 4
- Rows: 999
PUSH_SUBQ Example / No Hint: Query Plan

0 SELECT STATEMENT GOAL: CHOOSE
999 FILTER
44502 NESTED LOOPS
47000 NESTED LOOPS
47000 INDEX (FAST FULL SCAN) OF 'ORDERS_I1' (NON-UNIQUE)
93998 TABLE ACCESS (BY INDEX ROWID) OF 'LINE_ITEMS'
93998 INDEX (RANGE SCAN) OF 'LINE_ITEMS_ORD' (NON-UNIQUE)
91500 TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'
91500 INDEX (UNIQUE SCAN) OF 'PRODUCT_PK' (UNIQUE)
44501 INDEX (RANGE SCAN) OF 'SHIPMENTS_ORD' (NON-UNIQUE)
PUSH_SUBQ Example / No Hint

- Optimizer finds 47000 matching ORDERS rows
- It joins 47000 ORDERS rows with LINE_ITEMS and PRODUCTS
- After join is processed 44502 rows are found
- For each row in the join result the correlated subquery is executed
- Correlated subquery filters out all but 999 rows
- Very expensive
**PUSH_SUBQ** Example / With Hint

PUSH_SUBQ Hint placed in outer query block

```
SELECT /*+ ORDERED USE_NL(L,P) PUSH_SUBQ */ o.order_id, l.comments, p.product_name
FROM orders o, line_items l, products p
WHERE o.order_id = l.order_id
  AND l.product_id = p.product_id
  AND EXISTS
      (SELECT 1
       FROM shipments s
       WHERE s.order_id = o.order_id)
  AND o.order_id < 50000;
```
PUSH_SUBQ / With Hint: Query Stats

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>1000</td>
<td>1.39</td>
<td>1.52</td>
<td>0</td>
<td>103149</td>
<td>4</td>
<td>999</td>
</tr>
<tr>
<td>Total</td>
<td>1002</td>
<td>1.39</td>
<td>1.52</td>
<td>0</td>
<td>103149</td>
<td>4</td>
<td>999</td>
</tr>
<tr>
<td>Step</td>
<td>Operation</td>
<td>Table/Index Access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------</td>
<td>-------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td>GOAL: CHOOSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>999</td>
<td>NESTED LOOPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>NESTED LOOPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>INDEX (FAST FULL SCAN) OF 'ORDERS_I1' (NON-UNIQUE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46999</td>
<td>INDEX (RANGE SCAN) OF 'SHI PMENTS_ORD' (NON-UNIQUE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>TABLE ACCESS (BY INDEX ROWID) OF 'LINE_ITEMS'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>INDEX (RANGE SCAN) OF 'LINE_ITEMS_ORD' (NON-UNIQUE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>999</td>
<td>TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>INDEX (UNIQUE SCAN) OF 'PRODUCT_PK' (UNIQUE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PUSH_SUBQ Example / No Hint

- Optimizer first finds matching ORDERS rows
- For each matching ORDERS row it executes the correlated subquery against SHIPMENTS
- 999 ORDERS remain
- 999 ORDERS rows are joined with LINE_ITEMS and PRODUCTS
- Joining 999 ORDERS rows with LINE_ITEMS and PRODUCTS is much more efficient than joining 47000 ORDERS rows
- This explains lower number of block accesses
PUSH_SUBQ Hint

- Use When
  - There is a correlated subquery that correlates back to a table early in join order
  - The correlated subquery substantially reduces the number of rows in the outer table
- Benefit
  - Filtering tables early in join order prior to doing join substantially reduces the number of rows accessed in tables that occur later in join order