Disclaimer/Trademarks

© Copyright IBM Corporation 2012. All rights reserved.
U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

THE INFORMATION CONTAINED IN THIS DOCUMENT HAS NOT BEEN SUBMITTED TO ANY FORMAL IBM TEST AND IS DISTRIBUTED AS IS. THE USE OF THIS INFORMATION OR THE IMPLEMENTATION OF ANY OF THESE TECHNIQUES IS A CUSTOMER RESPONSIBILITY AND DEPENDS ON THE CUSTOMER’S ABILITY TO EVALUATE AND INTEGRATE THEM INTO THE CUSTOMER’S OPERATIONAL ENVIRONMENT. WHILE IBM MAY HAVE REVIEWED EACH ITEM FOR ACCURACY IN A SPECIFIC SITUATION, THERE IS NO GUARANTEE THAT THE SAME OR SIMILAR RESULTS WILL BE OBTAINED ELSEWHERE. ANYONE ATTEMPTING TO ADAPT THESE TECHNIQUES TO THEIR OWN ENVIRONMENTS DO SO AT THEIR OWN RISK. ANY PERFORMANCE DATA CONTAINED IN THIS DOCUMENT WERE DETERMINED IN VARIOUS CONTROLLED LABORATORY ENVIRONMENTS AND ARE FOR REFERENCE PURPOSES ONLY. CUSTOMERS SHOULD NOT ADAPT THESE PERFORMANCE NUMBERS TO THEIR OWN ENVIRONMENTS AS SYSTEM PERFORMANCE STANDARDS. THE RESULTS THAT MAY BE OBTAINED IN OTHER OPERATING ENVIRONMENTS MAY VARY SIGNIFICANTLY. USERS OF THIS DOCUMENT SHOULD VERIFY THE APPLICABLE DATA FOR THEIR SPECIFIC ENVIRONMENT.

Trademarks
IBM, the IBM logo, ibm.com, and DB2 are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.
Outline

- Write efficient predicates
- Minimize SQL traffic
- Use multi-row operations
- Avoid sorting whenever possible
- Only touch columns and rows you need
- OPTIMIZE FOR n ROWS
- Literals vs. variables – know the difference
- Subqueries vs Joins
Traits of a well-performing SQL query

- Written in an efficient form
- Accurate statistics
- Optimal optimizer settings
- Adequate system resources
Query Optimization

**SQL QUERY**

```sql
SELECT N_NAME, COUNT(*)
FROM ORDER, CUSTOMER, NATION
WHERE C_NATIONKEY = N_NATIONKEY
AND C_CUSTKEY = O_CUSTKEY
AND N_REGIONKEY = 4
AND O_ORDERDATE BETWEEN ? AND ?
GROUP BY N_NAME;
```

**Database Objects:**
- Tables
- Indexes
- Views
- MQTs
...

**Statistics:**
- # of rows in tables
- # of distinct column values
...

**Configuration:**
- Buffer pools
- Sort pool
- RID pool
...

**Optimizer**

**Access Path**
Predicates, predicates, predicates
... A prime influence on access paths

- **Predicates**
  - Found inside WHERE, ON, HAVING clauses
  - Have a huge impact on query performance!

- **Can be:**
  - Extremely filtering (qualify very few rows) = good!
  - Poorly filtering (qualify a ton of rows)

```sql
SELECT ... FROM EMP E, DEPT D
WHERE
  E.GENDER = 'F'          equal
  AND E.AGE BETWEEN 25 AND 65  range
  AND E.DEPTID = D.DEPTID    equal
  AND E.SAL = (SELECT MAX(SAL)  subquery
                FROM EMP WHERE ...)
  AND E.EDU IN ('BA', 'BS', 'MA', 'MS')  In list
```
Predicates: Indexable or Not?

- **Indexable Predicates**
  - Can match to index entries
  - May or may not become index matching predicates depending on available indexes and access path selected
  - The best kind of predicates

- **Not Indexable Predicates**
  - Cannot match index entries

WHERE LASTNAME = 'SMITH' AND FIRSTNAME <> 'JOHN'
Predicate Processing
Index Matching

- Restrict the range of data that is retrieved
  - Index Matching defines START and STOP keys on the index
  - All other predicates will reject rows based upon this retrieved range of data

Index on EMPLOYEE(LASTNAME, FIRSTNAME, AGE)

```
SELECT COUNT(*)
FROM EMPLOYEES
WHERE LASTNAME = 'SPADE'
  AND FIRSTNAME = 'SAM'
  AND SALARY > ?
```
Predicate Processing

Index Screening

- Applied on the index after matching predicates, but before data access
- Column needs to exist in the chosen index
- Screening predicates do not limit the number of index entries read
- But can limit the number of data rows retrieved

Index on

EMPLOYEE(LASTNAME, FIRSTNAME, AGE)

SELECT COUNT(*)
FROM EMPLOYEES
WHERE LASTNAME = 'SPADE'
AND SALARY > ?
AND AGE > ?
SELECT ... FROM EMP
WHERE TYPE = 'ENGINEER'
AND SAL > 125000
AND GRADE <> 'X'
AND UPPER(PROJ) LIKE '%DEV%'

Index on
(TYPE, DEPT, SAL)
Predicate Processing (contd.)

- **Stage 1**
  - Evaluated by the Data/Index Manager with relatively little expense
  - Some Stage 1 predicates are “Indexable” (i.e. use indexes)

- **Stage 2**
  - Much more expensive for DB2 to resolve due to additional processing and code path.
  - Cannot make effective use of indexes.

- **What determines stage 1 vs stage 2?**
  - Predicate syntax
  - Type and length of constants or columns in the predicate
  - Whether the predicate is applied before or after a join
  - Table join sequence
  - Read the official books for your particular release

- **Well written queries**
  - Filter as much as needed/possible within the query itself
  - Favor Stage 1 Indexable -> Stage 1 Others -> Stage 2
## Promote predicates to earlier stage

- Watch out for functions or arithmetic against columns

<table>
<thead>
<tr>
<th>Stage 2</th>
<th>Stage 1</th>
<th>Indexable</th>
</tr>
</thead>
<tbody>
<tr>
<td>QTY * 2 = :hv</td>
<td>QTY = :hv / 2</td>
<td></td>
</tr>
<tr>
<td>YEAR(DCOL) = 2008</td>
<td>DCOL</td>
<td>DCOL BETWEEN '2008-01-01' AND '2008-12-31'</td>
</tr>
<tr>
<td>:hv BETWEEN C1 AND C2</td>
<td>:hv &gt;= C1 AND :hv &lt;= C2</td>
<td></td>
</tr>
<tr>
<td>DCOL + 10 YEARS &lt; CURRENT DATE</td>
<td>DCOL</td>
<td>DCOL &lt; CURRENT DATE - 10 YEARS</td>
</tr>
<tr>
<td>DCOL &lt;&gt; '9999-12-31'</td>
<td>DCOL &lt; '9999-12-31'</td>
<td></td>
</tr>
<tr>
<td>GENDER &lt;&gt; 'F'</td>
<td>GENDER = 'M'</td>
<td></td>
</tr>
</tbody>
</table>
Minimize SQL traffic

- Don’t issue SQL if you can avoid it
  - E.g., Consider caching read-only constants on client

- Avoid generic “I/O boxes”
  - E.g., Consider customizing your SQL to suit your true need

- Avoid joins in applications
  - Let DB2 do what it does best
Avoid touching unnecessary data

- Only touch the columns you really need

- Extra columns can be a drag on performance
  - Access path may not be the best
    - E.g., INDEXONLY not available
  - Data is carried all the way from disk to the client
  - Increased CPU costs

- Avoid “SELECT *” unless really needed
Don’t return unnecessary rows

- Don’t filter rows in the application that DB2 can filter
  - Use predicates

- Consider FETCH FIRST n ROWS only
  - When the client will only see a limited # of rows
  - DB2 optimizes the access path accordingly
  - Can be used in subselects

```sql
SELECT PNAME, PCOST, SALARY, PID
FROM PRODUCTS
WHERE (PNAME = ? AND PID > ?)
  OR (PNAME > ?)
ORDER BY PNAME, PID
FETCH FIRST 20 ROWS ONLY
```
Don’t use features that aren’t needed

- Specify “FOR READ ONLY” on cursors that aren’t going to be used for positioned update / delete
  - Default is to assume cursor might be used for positioned delete
  - Some query optimizations can only be done on read-only cursors

- Specify “NO SCROLL” for cursors that don’t need scrolling (or take the default)
  - Scrolling adds overhead, even if only fetching in a forward direction
  - Some query optimization can only be done on non-scroll cursors

- Multi-row Fetch… great if fetching lot’s of data, but if fetching only a few rows:
  - Adds overhead to fetch rows that might not be needed
  - Some query optimizations can only be done on single-row fetch cursors
Minimize SQL traffic
Use Multi-row FETCH

- Returns up to 32,767 rows in a single API call
- Significant CPU performance improvements
- Works for static or dynamic SQL
- Works for scrollable or non-scrollable cursors
- Support for positioned UPDATEs and DELETEs
- Sample program DSNTEP4 = DSNTEP2 with multi-row fetch)
Minimize SQL traffic

Use Multi-row FETCH

- Coding multi-row fetch
  - “WITH ROWSET POSITIONING” on cursor declaration
  - “NEXT ROWSET” and “FOR n ROWS” on the FETCH
  - Define host variable arrays
  - Fetch loop to process the rows

- When using multi-row fetch
  - Avoid GET DIAGNOSTICS due to high CPU overhead
  - Use the SQLCODE field of the SQLCA
    - Fetch was successful (SQLCODE 000)
    - Fetch failed (negative SQLCODE)
    - End of file (SQLCODE 100)
Combine UPDATE and INSERT into a single statement via the SQL MERGE statement

```
MERGE INTO PRODUCT AS OLDPROD
USING (VALUES (:PID, :COST, :DISCOUNT)
    FOR :ROWCNT ROWS)
    AS NEWPROD(PID, COST, DISCOUNT)
ON OLDPROD.PID = NEWPROD.PID
WHEN MATCHED THEN
    UPDATE SET COST = NEWPROD.COST,
        DISCOUNT = NEWPROD.DISCOUNT
WHEN NOT MATCHED THEN
    INSERT (PID, COST, DISCOUNT)
VALUES (NEWPROD.PID,
        NEWPROD.COST,
        NEWPROD.DISCOUNT)
```
Minimize SQL traffic

Select from Insert / Update / Delete

Benefits
- Select what was just changed
- Save multiple calls to DB2

Common Use Cases
- Identity columns or sequence values that get automatically assigned by DB2
- User-defined defaults and expressions that are not known to the developer
- Columns modified by triggers that can vary from insert to insert depending on values
- ROWIDs, CURRENT TIMESTAMP that are assigned automatically

Example:

```sql
/* Generate a unique id for the next customer */
SELECT CUSTID
FROM FINAL TABLE
(INsert INTO CUSTOMERS (CUSTID, CUSTNAME)
VALUES
  (NEXT VALUE FOR CUSTSEQ, 'John Roberts'))
```
Avoid Unnecessary Sorts

- DB2 may perform a sort to support
  - ORDER BY
  - GROUP BY
  - Duplicate removal (DISTINCT, UNION, ...)
  - Join processing
  - Subquery processing

- But ...
  - Sorts can be expensive
  - An SQL statement may have multiple sorts

- Action items:
  - Examine DB2 explain information to check for sorts
  - Try to take advantage of ways in which DB2 can avoid a sort
  - If you must sort, only sort what’s needed
Avoid Unnecessary Sorts (contd.)

ORDER BY

Index on (PTYPE, PNAME, PCOST)

- Matches all index columns
  SELECT ...
  FROM PROD
  ORDER BY PTYPE, PNAME, PCOST

- Matching leading index column(s)
  SELECT ...
  FROM PROD
  ORDER BY PTYPE

- Matching some index column(s), but others column(s) constrained
  SELECT ...
  FROM PROD
  WHERE PTYPE = ‘X05’
  ORDER BY PNAME
Avoid Unnecessary Sorts (contd.)

GROUP BY

Index on (PTYPE, PNAME, PCOST)

- Matches leading index columns
  
  SELECT PTYPE, PNAME, COUNT(*)
  FROM PROD
  GROUP BY PTYPE, PNAME;

- Matching leading index column(s) but in different order
  
  SELECT PNAME, PTYPE, AVG(SALARY)
  FROM PROD
  GROUP BY PNAME, PTYPE
  
  // Watch out: results will not be in “GROUP BY order”

- Matching some index column(s), but others column(s) constrained
  
  SELECT TYPE, COUNT(*)
  FROM PROD
  WHERE PTYPE = ‘X05’
  GROUP BY PNAME;
Avoid Unnecessary Sorts (contd.)

DISTINCT

- DB2’s DISTINCT processing has evolved
  - Prior to V9, DISTINCT usually involved a sort unless a unique index was available
    - GROUP BY could be used as a workaround
  - With DB2 9, DB2 may take better advantage of indexes

- Use DISTINCT only when needed
  - DISTINCT may involve expensive sorting
  - DISTINCTs inside subselects may involve materializations
  - Don’t use DISTINCT just to be safe
    - Make sure duplicate rows are actually possible
Avoid Unnecessary Sorts (contd.)

DISTINCT

- If duplicates are to be removed:
  - Try rewriting the query using an IN or EXISTS subquery.
  - EXISTS is a faster alternative because DB2 can do “early out”

- Example
  
  ```sql
  SELECT DISTINCT d.deptno, d.dname deptname
  FROM dept d, emp e
  WHERE d.deptno = e.deptno;
  ```

- Rewritten query
  
  ```sql
  SELECT d.deptno, d.dname deptname
  FROM dept d
  WHERE EXISTS (SELECT 1 FROM emp e
    WHERE e.deptno = d.deptno);
  ```
OPTIMIZE FOR and FETCH FIRST

- When # of rows needed is significantly < # of rows returned, but exact number is not known:
  - Tell the optimizer!
  - DB2 will try to eliminate sorts such as “RID List Prefetch sort”

```sql
SELECT EMPNO, PNAME, DEPTNO, SALARY
FROM EMPLOYEE
WHERE DEPTNO > ?
OPTIMIZE FOR 1 ROW
```

- In cases where max number of fetches is known use FETCH FIRST n ROWS ONLY
  - Discourages sorting and may reduce internal processing when sort can’t be avoided (i.e. sort can be more efficient if it knows only “top n” values are going to be returned)
Parameterize Dynamic SQL, unless, ...

SELECT ... FROM ORDERS WHERE CUSTID = 1331
SELECT ... FROM ORDERS WHERE CUSTID = 78
SELECT ... FROM ORDERS WHERE CUSTID = 3633
SELECT ... FROM ORDERS WHERE CUSTID = 26631
SELECT ... FROM ORDERS WHERE CUSTID = 12
...

VS.

SELECT ... FROM ORDERS WHERE CUSTID = ?
Parameterize Dynamic SQL, unless, ...

- **Embedded Literals**
  - Optimizer can produce best access path with a specific value
  - Useful when you want to beat skew
    - But you need the right frequency/histogram stats
    - Dynamic SQL cache may not be effectively used
      - V10 Statement Concentration can help

- **Markers or Host Variables**
  - For dynamic SQL, full dynamic SQL cache exploitation
    - Suboptimal access paths for skewed data
      - What if ‘M’ = 1%, ‘F’ = 99%?
  - REOPT(ONCE / AUTO / ALWAYS) can help
Think joins before subqueries

- **Joins**
  - Allow DB2 to pick the best table access sequence
  - Can outperform subqueries

- **Subqueries**
  - Force a specific sequence onto DB2

- Think of joining as a first resort, and subquerying as a last resort.

- DB2 can rewrite some subqueries -> joins
Think joins before subqueries (contd.)

Unique index on (DIVISION, DEPTNO)

Original query:

```
SELECT ... FROM EMP
WHERE DEPTNO IN
  (SELECT DEPTNO FROM DEPT
   WHERE LOCATION IN ('SAN JOSE', 'SAN FRANCISCO')
   AND DIVISION = 'MARKETING');
```

Rewritten query:

```
SELECT ... FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
  AND DEPT.LOCATION IN ('SAN JOSE', 'SAN FRANCISCO')
  AND DEPT.DIVISION = 'MARKETING';
```
Subqueries
Correlated vs Non-Correlated

SELECT * FROM EMP X
WHERE JOB = 'DESIGNER'
AND EXISTS (SELECT 1
FROM PROJ
WHERE DEPTNO = X.WORKDEPT
AND MAJPROJ = 'MA2100');

SELECT * FROM EMP
WHERE JOB = 'DESIGNER'
AND WORKDEPT IN (SELECT DEPTNO
FROM PROJ
WHERE MAJPROJ = 'MA2100');
Subqueries: To correlate or not?
Answer: It depends!

SELECT EMPID, EDLEVEL
FROM EMP E
WHERE
    JOBTYPE = ?
AND EDLEVEL >=
    (SELECT AVG(EDLEVEL)
     FROM EMP
     WHERE DEPTID = E.DEPTID)

Average computed for each employee's department, over and over again
Works best for few employees selected.

SELECT EMPID, NAME, EDLEVEL
FROM EMP E,
    (SELECT DEPTID,
     AVG(EDLEVEL) AVGED
     FROM EMP
     GROUP BY DEPTID) A
WHERE
    JOBTYPE = ?
AND E.DEPTID = A.DEPTID
AND EDLEVEL >= AVGED

Average-per-department, computed once for all departments
Works best when many employees selected.
Subquery evaluation order

- Non-correlated subqueries are executed before correlated
  - Multiple non-correlated subqueries are executed in the sequence they are coded

- Next are correlated subqueries
  - Multiple correlated subqueries are executed in the sequence they are coded
  - Correlated subqueries cannot be executed however until all correlation predicates are available

- Code subqueries in order of restrictiveness
WHERE NOT EXISTS (SELECT 1 FROM DSN8710.PROJ P1 WHERE P1.RESPEMP = E.EMPNO)
AND NOT EXISTS (SELECT 1 FROM DSN8710.PROJ P2 WHERE P2.DEPTNO = E.WORKDEPT)

Reverse the subqueries

WHERE NOT EXISTS (SELECT 1 FROM DSN8710.PROJ P2 WHERE P2.DEPTNO = E.WORKDEPT)
AND NOT EXISTS (SELECT 1 FROM DSN8710.PROJ P1 WHERE P1.RESPEMP = E.EMPNO)
What did we discuss?

- Write efficient predicates
- Minimize SQL traffic
- Use multi-row operations
- Avoid sorting whenever possible
- Only touch columns and rows you need
- OPTIMIZE FOR n ROWS
- Literals vs. variables – know the difference
- Subqueries vs Joins