SQL on iSeries:
Concepts and Implementations

Higher Productivity iSeries Programming Using SQL

By Thibault Dambrine
This Presentation

- SQL Data Definition Language: DDL
- Data Manipulation Techniques with SQL
- Implementing SQL
  - Interpreted SQL
  - Compiled SQL
    - SQL Stored Procedures
    - SQL Functions
- SQL Performance Considerations
## DDL: SQL Terminology

<table>
<thead>
<tr>
<th>iSeries</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>Collection or Schema</td>
</tr>
<tr>
<td>Physical File</td>
<td>Table</td>
</tr>
<tr>
<td>Record</td>
<td>Row</td>
</tr>
<tr>
<td>Field</td>
<td>Column</td>
</tr>
<tr>
<td>Logical File</td>
<td>View or Index</td>
</tr>
</tbody>
</table>
DDL: Data Definition Language
- Used to:
  - Define Tables
  - Alter Tables
  - Tables defined with DDL can be accessed with both SQL and Traditional languages like RPG/C/COBOL
DDL Limitations:

- Tables created with DDL can support ONE MEMBER ONLY

- Tables or Views using long names (up to 128 characters) will not be visible with iSeries commands DSPOBJD and DSPFD
A Word about NULLs

- The NULL value is effectively equivalent to "UNKNOWN"
- NULL is DIFFERENT BLANK

- Assigning a value of NULL:

  ```sql
  UPDATE TABLE_A SET USER_NAME = NULL
  ```

- Comparing a value with NULL:

  ```sql
  UPDATE TABLE_A SET COLUMN_A = 'NOT FILLED'
  WHERE LAST_NAME IS NULL
  ```
DDL Coding Example: A SIMPLE TABLE

CREATE TABLE ER100F
(
  BATCH_ID       FOR BTCHID  NUMERIC(10)  NOT NULL,
  SOURCE_FACILITY FOR SRCFAL CHAR(30)     NOT NULL,
  SOURCE_DESCRIPTION FOR SRCDSC VARCHAR(100) NOT NULL,
  LOAD_TIMESTAMP  FOR LDTMSP TIMESTAMP NOT NULL
) ;

LABEL ON ER100F (SOURCE_FACILITY TEXT IS 'Source Facility ');
LABEL ON ER100F (BATCH_ID TEXT IS 'Batch ID ');
LABEL ON ER100F (LOAD_TIMESTAMP TEXT IS 'Load Timestamp');

LABEL ON TABLE ER100F IS 'Test Data Fact Table' ;
DDL Coding Example: A UNIQUE Index

CREATE UNIQUE INDEX ER100FIDX ON ER100F
(
    BATCH_DATE,
    BATCH_ID
)

- Equivalent of a Logical File
- Visible with DSPDBR Command
Creating a VIEW: DDL Coding Example

```
CREATE VIEW MA_PROJ
    AS SELECT * FROM PROJECT
    WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

- Equivalent of a Logical File with a SELECT
- Visible with DSPDBR Command IF the VIEW name is 10 characters or less
DDL Coding Examples: A more complex view

```
CREATE VIEW RSLTS_ABOVE_AVG AS
SELECT MR.SOURCE_FACILITY, MR.BATCH_ID,
MR.MATERIAL_TYPE, MR.MATERIAL_NAME,
MR.COMPONENT_NAME, MR.ACTUAL_RESULTS
FROM MAT_RESULTS MR
WHERE MR.ACTUAL_RESULTS >
(SELECT AVG(AV.ACTUAL_RESULTS) FROM MAT_RESULTS AV)
```

Refining a data selection from a VIEW:

```
SELECT * FROM RSLTS_ABOVE_AVG ORDER BY SOURCE_FACILITY
```
Altering Existing Tables with DDL

- Adding a new column

```sql
ALTER TABLE EQP_TABLE ADD COLUMN EQUIPMENT_CATEGORY FOR EQPCAT CHAR(10)
```

- Removing a column

```sql
ALTER TABLE EQP_TABLE DROP COLUMN EQUIPMENT_CATEGORY
```
Setting up Constraints in SQL

- Setting up a Primary Key with existing tables

- Setting up a Primary Key and Parent/Child Constraint (when creating parent/child tables)

```
CREATE TABLE DEPT_TABLE
(
    DEPT_ID CHAR(2),
    DEPT_NAME VARCHAR(20),
    PRIMARY KEY (DEPT_ID)
)

CREATE TABLE EMPLOYEE_TABLE
(
    EMP_NUMBER INT,
    EMP_NAME VARCHAR(20),
    DEPT_ID CHAR(2),
    PRIMARY KEY (EMP_NUMBER),
    FOREIGN KEY (DEPT_ID) REFERENCES DEPT_TABLE (DEPT_ID)
)

ALTER TABLE EMPLOYEE_TABLE ADD CONSTRAINT CSTEMPDPT
FOREIGN KEY (DEPT_ID) REFERENCES DEPT_TABLE (DEPT_ID)
```
Dealing with SQL Object Names Longer than 10 Characters

- DDL allows for table names longer than 10 characters
- DSPFD CANNOT see these tables
- DSPOBJD CANNOT see these tables
- Keeping track of these tables can only be done through the SQL CATALOG
- SQL CATALOG Files are stored in
- QSYS2/SYS* system table objects
## Most Used Catalog Tables

<table>
<thead>
<tr>
<th>Catalog Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSCOLUMNS</td>
<td>Columns</td>
</tr>
<tr>
<td>SYSCST</td>
<td>Constraints</td>
</tr>
<tr>
<td>SYSFUNCS</td>
<td>Functions</td>
</tr>
<tr>
<td>SYSINDEXES</td>
<td>Indexes</td>
</tr>
<tr>
<td>SYSKEYS</td>
<td>Keys</td>
</tr>
<tr>
<td>SYSPROCS</td>
<td>Procedures</td>
</tr>
<tr>
<td>SYSTABLES</td>
<td>Tables</td>
</tr>
<tr>
<td>SYSTRIGGER</td>
<td>Triggers</td>
</tr>
<tr>
<td>SYSVIEWS</td>
<td>Views</td>
</tr>
</tbody>
</table>
Finding SQL Object Names Longer than 10 Characters

- To find a table with a long name:
  ```sql
  SELECT TABLE_NAME, TABLE_SCHEMA
  FROM QSYS2/SYSTABLES
  WHERE TABLE_NAME = 'MONTH_TO_DATE_SALES'
  ```

- To find the columns in a long file name:
  ```sql
  SELECT * FROM QSYS2/SYSCOLUMNS
  WHERE TABLE_NAME = 'MONTH_TO_DATE_SALES'
  ```
Real Life Use for Catalog Tables

- Where is this column (field) name used?

```sql
SELECT * FROM QSYS2/SYSCOLUMNS
WHERE COLUMN_NAME = 'GLMCU'
```

- Are the number of columns (fields) for this table the same in all schemas (libraries)?

```sql
SELECT TABLE_NAME, TABLE_SCHEMA, COUNT(*) FROM QSYS2/SYSCOLUMNS
WHERE TABLE_NAME = 'F0911' GROUP BY TABLE_NAME, TABLE_SCHEMA
```
DDIL Summary

- With DDL, you can create or alter tables
- DDL allows table and column names to be longer than 10 characters
- All DDL Objects can be found in the SQL Catalog Tables
- All SQL Catalog files start with SYS* and can be found in library QSYS2
Part 2
CODING in SQL: MAKE IT HAPPEN!

- SQL JOIN
- SQL Update
- Group BY
- Casting
- Date & Time Manipulation
SQL Joins

- Join or Inner Join
- Left/Right Join or Left/Right Outer Join
- Left/Right Exception Join
- Cross Join
JOIN or INNER JOIN

- Most commonly used join
- Returns as many rows as there are matches, no more, no less
- Returns values for all columns
INNER Join Example: Getting only the exact key matches

```sql
SELECT EM.EMPLOYEE_NBR, EM.EMPLOYEE_NAME, BM.EMPLOYEE_BENEFITS_DESC
FROM EMPLOYEE_MASTER EM
INNER JOIN BENEFITS_MASTER BM
ON EM.EMPLOYEE_NBR = BM.EMPLOYEE_NBR
```

<table>
<thead>
<tr>
<th>EM.EMPLOYEE_NBR</th>
<th>EM.EMPLOYEE_NAME</th>
<th>BM.EMPLOYEE_BENEFITS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>4567</td>
<td>Garth Johnson</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>7342</td>
<td>Gene Lockhart</td>
<td>FULL MEDICAL</td>
</tr>
<tr>
<td>121</td>
<td>Steve Carson</td>
<td>FULL MEDICAL</td>
</tr>
</tbody>
</table>
LEFT JOIN or LEFT OUTER JOIN (1 of 2)

- Returns values for ALL the rows on the left table and values from the joined table that match

- When a match is not found in the joined file (to the right), NULLs are returned

- NULL values can be overridden with the IFNULL operand
LOJ Example: Getting the matches, the data from the left table and defaults from the right table if no values found

```
SELECT
  EM.EMPLOYEE_NBR,
  EM.EMPLOYEE_NAME,
  IFNULL(BM.EMPLOYEE_BENEFITS_DESC, 'Benefits not yet allocated')
FROM EMPLOYEE_MASTER EM
  LEFT OUTER JOIN BENEFITS_MASTER BM ON EM.EMPLOYEE_NBR = BM.EMPLOYEE_NBR
```
LEFT JOIN or LEFT OUTER JOIN Results

LOJ Results **WITHOUT** IFNULL default override

<table>
<thead>
<tr>
<th>EM.EMPLOYEE_NBR</th>
<th>EM.EMPLOYEE_NAME</th>
<th>EM.EMPLOYEE_BENEFITS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>4567</td>
<td>Garth Johnson</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>852</td>
<td>Brian Evans</td>
<td>-</td>
</tr>
<tr>
<td>121</td>
<td>Steve McPhearson</td>
<td>-</td>
</tr>
</tbody>
</table>

LOJ Results **WITH** IFNULL default override

<table>
<thead>
<tr>
<th>EM.EMPLOYEE_NBR</th>
<th>EM.EMPLOYEE_NAME</th>
<th>EM.EMPLOYEE_BENEFITS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>4567</td>
<td>Garth Johnson</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>852</td>
<td>Brian Evans</td>
<td>BENEFITS NOT YET ALLOCATED</td>
</tr>
<tr>
<td>121</td>
<td>Steve McPhearson</td>
<td>BENEFITS NOT YET ALLOCATED</td>
</tr>
</tbody>
</table>
USING MORE THAN ONE LOJ Table

```sql
INSERT INTO EMPLOYEE_DATA
(
EMPLOYEE_NBR,
EMPLOYEE_NAME,
EMPLOYEE_BENEFITS_DESC,
EMPLOYEE_SALARY,
SALARY_CATEGORY
)

SELECT
EM.EMPLOYEE_NBR,
EM.EMPLOYEE_FIRST_NAME || ' ' || EM.EMPLOYEE_LAST_NAME,
IFNULL(BM.EMPLOYEE_BENEFITS_DESC, 'New Employee – Benefits not yet allocated'),
IFNULL(PM.YEARLY_SALARY, 0),
CASE
  WHEN PM.YEARLY_SALARY<100000 THEN 'REGULAR EMPLOYEE'
  WHEN PM.YEARLY_SALARY<=100000 THEN 'EXECUTIVE EMPLOYEE'
  WHEN PM.YEARLY_SALARY IS NULL THEN 'UNKNOWN - INVESTIGATE'
  ELSE 'DA BOSS'
END

FROM EMPLOYEE_MASTER EM
LEFT OUTER JOIN BENEFITS_MASTER BM ON EM.EMPLOYEE_NBR = BM.EMPLOYEE_NBR
LEFT OUTER JOIN PAYROLL_MASTER PM ON EM.EMPLOYEE_NBR = PM.EMPLOYEE_NBR;
```
LEFT EXCEPTION JOIN

- Returns only the rows from the left table that do not have a match in the right table

```sql
SELECT EM.EMPNO, EM.LASTNAME, E.M.PROJNO FROM EMPLOYEE EM
EXCEPTION JOIN PROJECT PJ
ON EM.PROJNO = PJ.PROJ#`
```
CROSS JOIN

- Also known as "CARTESIAN PRODUCT"
- Can be specified with the CROSS JOIN syntax or by listing two tables without a WHERE clause
- Returns a row in the result table for each combination of rows from the tables being joined

```
SELECT * FROM FILEA CROSS JOIN FILEB
```

```
SELECT * FROM FILEA, FILEB
```
### CROSS JOIN EXAMPLE

<table>
<thead>
<tr>
<th>EM.EMPNBR</th>
<th>EM.EMPNAME</th>
<th>BEN_NBR</th>
<th>EM.EMPLOYEE_BENEFITS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>1111</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>4567</td>
<td>Garth Johnson</td>
<td>2222</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>852</td>
<td>Brian Evans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>Steve McPhearson</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### CROSS JOIN Results

<table>
<thead>
<tr>
<th>EM.EMPNBR</th>
<th>EM.EMPNAME</th>
<th>BEN_NBR</th>
<th>EM.EMPLOYEE_BENEFITS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>Steve McPhearson</td>
<td>1111</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>121</td>
<td>Steve McPhearson</td>
<td>2222</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>852</td>
<td>Brian Evans</td>
<td>1111</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>852</td>
<td>Brian Evans</td>
<td>2222</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>1111</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>2222</td>
<td>BOTTOM DENTAL</td>
</tr>
<tr>
<td>4567</td>
<td>Garth Johnson</td>
<td>1111</td>
<td>TOP DENTAL</td>
</tr>
<tr>
<td>4567</td>
<td>Garth Johnson</td>
<td>2222</td>
<td>BOTTOM DENTAL</td>
</tr>
</tbody>
</table>
CASTING and Joining Tables With Incompatible Keys using CAST

```
SELECT CAST(ZIP_NUMBER AS CHAR(5)) FROM FILEB

SELECT INT(SUBSTRING(TELEPHONE, 1, 3) ) AREA_CODE FROM FILEA

- Tips & Techniques
  Joining with Cast Values:
  SELECT * FROM FILE_A, FILE_C
  WHERE FILEA.INT_KEY
       = CAST(SUBSTRING(TELEPHONE, 1, 3) as INT )
```
Join Summary

- Inner Join
- Left or Right Outer Join
- Left or Right Exception Join
- Cross Join
Update/Delete with SQL

- Use of SQL for UPDATE or DELETE
Updating Data in a Table Using a Correlated Query

```
UPDATE EMPLOYEE_TABLE EM
SET (EM.FIRST_NAME, EM.LAST_NAME) =
(SELECT UPDT.FIRST_NAME, UPDT.LAST_NAME
     FROM NEW_NAMES UPDT )
WHERE EXISTS
  (SELECT *
       FROM NEW_NAMES UPDT WHERE UPDT.ID = EM.ID )
```

- Note the use of TWO WHERE clauses
- **WARNING**: Will crash if the second select yields more than one row!
Updating Data in a Table Using MAX() value to avoid possible duplicates

```
UPDATE  EMPLOYEE_TABLE  EM
SET (EM.ID) =
(SELECT  MAX(UPD0.ID)  FROM  UPDATE_TABLE  UPD0)
    WHERE  EXISTS
(SELECT  *  FROM  UPDATE_TABLE  UPD1
WHERE
    UPD1.FIRST_NAME =  EM.FIRST_NAME
AND  UPD1.LAST_NAME =  EM.LAST_NAME
AND  UPD1.ADDRESS_1 =  EM.ADDRESS_1
AND  UPD1.ADDRESS_2 =  EM.ADDRESS_2
AND  UPD1.ADDRESS_3 =  EM.ADDRESS_3
);
```
UPDATE FGLDETOS FGL
SET
( FGL.ADDRESS_BOOK_NUMBER,
  FGL.DW_STS_ADDRESS_BOOK_NUMBER ) =
(SELECT A.Q1AN8R,'O'
FROM F590101A A
WHERE A.Q1AN8 = FGL.ADDRESS_BOOK_NUMBER
  AND A.Q1AN8 != A.Q1AN8R
  AND A.Q1AN8R > 0
  AND FGL.ROW_SOURCE='A'
)
WHERE EXISTS
 ( SELECT *
FROM F590101A A1
WHERE A1.Q1AN8 = FGL.ADDRESS_BOOK_NUMBER
  AND A1.Q1AN8 != Q1AN8R
  AND A1.Q1AN8R > 0
  AND FGL.ROW_SOURCE='A'
);
Deleting Data in a Table Using a Correlated Query

```
(DELETE FROM EMPLOYEE_TABLE EM
WHERE EXISTS
(SELECT * FROM UPDATE_TABLE UPDT WHERE
UPDT.ID = EM.ID));
```

- Note again the use of TWO WHERE clauses
Update/Delete Summary

- UPDATE or DELETE in SQL is done with correlated sub-queries

- Ensure you have unique values to update with in an update SQL statement
Value-Added Data using SQL: Using the GROUP BY function

- Using the keyword GROUP BY
  - HAVING vs. WHERE
- Using DISTINCT
- Dealing with Duplicate Values
- Date/Time Manipulations
Aggregating Data with GROUP BY

- Find distinct values, regardless of how many rows in a table – AND sum or count of values

```sql
SELECT CITY_NAME,
    COUNT(*) ORDERS_COUNT,
    SUM(ORDER_VALUE) ORDERS_VALUE,
    AVG(ORDER_VALUE) AVERAGE,
    MIN(ORDER_VALUE) MIN_ORDER,
    MAX(ORDER_VALUE) MAX_ORDER
FROM ORDERS
GROUP BY CITY_NAME
ORDER BY 4
```

<table>
<thead>
<tr>
<th>CITY_NAME</th>
<th>ORDERS_COUNT</th>
<th>ORDERS_VALUE</th>
<th>AVERAGE</th>
<th>MIN_ORDER</th>
<th>MAX_ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>2324.00</td>
<td>45646546.00</td>
<td>19641.37</td>
<td>123.00</td>
<td>852.00</td>
</tr>
<tr>
<td>Phoenix</td>
<td>3434.00</td>
<td>544696445.00</td>
<td>158618.65</td>
<td>1822.00</td>
<td>5236.00</td>
</tr>
<tr>
<td>Chicago</td>
<td>4553.00</td>
<td>834098534.00</td>
<td>183197.56</td>
<td>268.00</td>
<td>7411.00</td>
</tr>
<tr>
<td>Houston</td>
<td>2.00</td>
<td>554556.00</td>
<td>277278.00</td>
<td>965.00</td>
<td>1258.00</td>
</tr>
</tbody>
</table>
Aggregating Data – HAVING Clause

- For comparing individual rows, use WHERE
- For aggregated values, use HAVING

```
SELECT STORE_NAME, STORE_STATE, SUM(SALES) AS STORE_SALES
FROM STORE_INFORMATION
WHERE STORE_STATE = 'IL'
GROUP BY STORE_NAME, STORE_STATE
HAVING SUM(SALES) > 1500
```

<table>
<thead>
<tr>
<th>STORE_NAME</th>
<th>STORE_STATE</th>
<th>STORE_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontario Street</td>
<td>IL</td>
<td>3434</td>
</tr>
<tr>
<td>Michigan Avenue</td>
<td>IL</td>
<td>4553</td>
</tr>
</tbody>
</table>
Finding Distinct Values in a Table with SQL

- Find distinct values, regardless of how many rows in a table

```
SELECT DISTINCT CITY_NAME, ZIP_CODE FROM ORDERS WHERE CITY_NAME = 'CHICAGO'
ORDER BY ZIP_CODE
```

<table>
<thead>
<tr>
<th>CITY</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHICAGO</td>
<td>60606</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>60607</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>60608</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>60609</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>60610</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>60611</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>60612</td>
</tr>
</tbody>
</table>
Finding Duplicate Keys in a Table

( 
SELECT ADDRESS_1, ADDRESS_2, 
ADDRESS_3, COUNT(*) 
FROM CONTACT_TABLE 
HAVING COUNT(*) > 1 
GROUP BY ADDRESS_1, ADDRESS_2, 
ADDRESS_3
)

- Very Common SQL Example
- Note the use of the GROUP BY clause
- Unique Keys still best to keep duplicates out when possible!
- Useful to clean up raw data
Removing Duplicate Rows In A Table (Address Example)

```
(Destroy FROM CONTACT_TABLE AD1
WHERE AD1.ID_NUMBER <
(  
  SELECT MAX(AD2.ID_NUMBER)
  FROM CONTACT_TABLE AD2
  WHERE (AD1.ADDRESS_1 = AD2.ADDRESS_1 AND
         AD1.ADDRESS_2 = AD2.ADDRESS_2 AND
         AD1.ADDRESS_3 = AD2.ADDRESS_3 )
  )
)
```

- Note the use of the **MAX** clause
- Note the use of **Correlation Names** AD1 and AD2 - attacking the same table twice with two different correlated names
Extracting ONLY UNIQUE (no duplicate) Values USING DISTINCT with ALL the columns in the table

```sql
SELECT DISTINCT
PT1.CLERK,
PT1.TRANS_NUMBER,
PT1.ITEM,
PT1.SIZE,
PT1.COLOUR,
PT1.DOLLAR_AMT,
PT1.POLLING_TIME
FROM POLLING_TABLE PT1
```
Time & Date Values on iSeries: a Very useful Data Type

- The TIMESTAMP value on iSeries records time to ONE MILLIONTH of a SECOND

- Measure time values conveniently with SQL, from dates to seconds with very little effort
Date & Time Data Manipulations

- DATE and TIMESTAMMP data types allow easy date and time calculations

```sql
SELECT CURRENT_TIMESTAMP
  + 7 hours - 5 minutes - 10 seconds
FROM SYSIBM/SYSDUMMY1
2005-06-21-09.07.10.553453

SELECT CURRENT DATE + 30 DAYS FROM SYSIBM/SYSDUMMY1
  05/07/21

SELECT CHAR(DATE(TIMESTAMP('2005-06-21-09.07.10.553453')
  + 7 DAYS)) FROM SYSIBM/SYSDUMMY1
05/06/28

SELECT * from ORDER_TABLE WHERE
  CURRENT_TIMESTAMP - ORDER_DATE < 30 DAYS
```
SYSTEM Date & Time RETRIEVAL

- TIME Retrieval using CURTIME function
  SELECT `curtime()` FROM `sysibm/sysdummy1`

- DATE Retrieval using CURDATE function
  SELECT `curdate()` FROM `sysibm/sysdummy1`

- CURRENT TIMESTAMP Retrieval using NOW function
  SELECT `now()` FROM `sysibm/sysdummy1`

- GMT TIMESTAMP using NOW and TIMEZONE
  select `now()` - current timezone from `sysibm/sysdummy1`
Value Added DATA Recap

- Group BY
- Casting
- Date & Time Data Type
- Using the keyword GROUP BY
  - HAVING vs. WHERE
- Using DISTINCT
- Dealing with Duplicate Values
Part 3
SQL Implementation

- Interpreted SQL
- SQL Stored Procedures
Interpreted SQL

- Used with the RUNSQLSTM CL Command
- SQL commands are stored in a Source Member
- Format:

```
RUNSQLSTM
SOURCELIB/SOURCEFILE SOURCEMEMBR
```
Interpreted SQL Characteristics

- Must have an output if there is a select
- Can be used for Set Processing ONLY (as opposed to individual rows)
- Cannot receive parameters
- Cannot use loops
- Can use CASE Statements but not IF/Then/Else
Running Interpreted SQL

Can be run with the RUNSQLSTM CL command

RUNSQLSTM LIBRARY/FILE MEMBER

Sample Source:

```sql
INSERT INTO EXTRACT
SELECT INPUT.FIRST_NAME,
     INPUT.LAST_NAME, INPUT.SALARY
FROM PAYROLL INPUT
WHERE (INPUT.SALARY IS > 1000000);
```
SQL Stored Procedures Characteristics

- Compile into Executable CLE type *PGM objects
- Faster than interpreted code – MOST TIMES
- Can be debugged like any CLE program
- Debug to retrieve SQL Optimizer messages

- Can use Parameters, Variables
- Logic constructs (if/then/else, do/for loops)
- The ability to take advantage of compiled functions
CREATE PROCEDURE PROC_NAME
   LANGUAGE SQL

   -- START PROCEDURE
   -- This procedure will, for each row of table ER400SX, retrieve the current timestamp
   -- and update the column PUBLISH_TMS within ER400SX
   BEGIN

   -- DECLARE CURSOR VARIABLES
   DECLARE PUBLISH_TMS TIMESTAMP ;
   DECLARE WORK_TIMESTAMP TIMESTAMP ;
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000' ;
   DECLARE AT_END INT DEFAULT 0 ;
   DECLARE SQLCODE INT DEFAULT 0 ;
   DECLARE CURSOR_UPD CURSOR FOR
      SELECT PUBLISH_TMS FROM ER400SX MAIN;
   SET AT_END = 0 ;

   OPEN CURSOR_UPD ;
   WHILE (SQLCODE = 0 )
      FETCH CURSOR_UPD INTO WORK_TIMESTAMP ;
      UPDATE ER400SX
         SET PUBLISH_TMS  = CURRENT_TIMESTAMP,
             TIME_ELAPSED = DAY(CURRENT_TIME_STAMP - WORK_TIMESTAMP)
         WHERE CURRENT OF CURSOR_UPD ;
   END WHILE ;
   CLOSE CURSOR_UPD ;

   -- END PROCEDURE
END
SQL Stored Procedure Tips

- The code begins with
  `CREATE PROCEDURE PROC_NAME`
  where `PROC_NAME` will be the name of the procedure name – NOT the MEMBER NAME
- The procedure will be created in the Current Library
- The `CREATE PROCEDURE` statement will not replace an existing procedure
CREATE PROCEDURE DWCVGDOS01

LANGUAGE SQL
SET OPTION OUTPUT = *PRINT, DBGVIEW = *SOURCE

-- START PROCEDURE
BEGIN

-- DECLARE CURSOR VARIABLES
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
DECLARE SQLCODE INT DEFAULT 0;
DECLARE AT_END INT DEFAULT 0;
DECLARE CURRENT_ADDRESS_BOOK_VALUE INT;
DECLARE NEW_ADDRESS_BOOK_VALUE INT;
DECLARE CURRENT_SUR_KEY INT;

--------------------------------------------------------------
------------------
-- CURSOR 1 - FGLDET BEING UPDATED
--------------------------------------------------------------
------------------
DECLARE CURSOR_MAIN CURSOR FOR
SELECT GLAN8,
Q1AN8R,
DW_SURROGATE_KEY
FROM FGLDETOS AA
JOIN F590101A BB
ON BB.Q1AN8 = AA.GLAN8
AND BB.Q1AN8 <> BB.Q1AN8R
AND BB.Q1AN8R > 0
AND AA.ROW_SOURCE = 'A';

--------------------------------------------------------------
-------------
-- SET VARIABLES FOR PROCESSING
--------------------------------------------------------------
-------------
OPEN CURSOR_MAIN;
SET AT_END = 0;

END-- END OF PROCEDURE --

-- MAIN UPDATE LOOP. UPDATE THE MAIN FILE USING THE SECONDARY FILE.
BEGIN WHILE (SQLCODE = 0) DO
FETCH CURSOR_MAIN INTO CURRENT_ADDRESS_BOOK_VALUE,
NEW_ADDRESS_BOOK_VALUE,
CURRENT_SUR_KEY ;

UPDATE FGLDETOS FGL
SET
(FGL.ADDRESS_BOOK_NUMBER ,
FGL.DW_STS_ADDRESS_BOOK_NUMBER)
=
(NEW_ADDRESS_BOOK_VALUE ,
'O')
WHERE FGL.DW_SURROGATE_KEY = CURRENT_SUR_KEY ;
END WHILE;

CLOSE CURSOR_MAIN;
END
Steps to Create and Run a Stored Procedure

- Code the stored procedure in a source member
- Create the stored procedure in your current library (CURLIB) using RUNSQLSTM
  - This will result in the stored procedure to be created as an ILE C pgm, with your SQL code embedded within
- Syntax: CALL PROCEDURE_NAME
- **NOTE:** SQL procedure objects have to be called in an SQL environment
4 Ways to call an SQL Stored Procedure

- Interactively – from the STRSQL command prompt
- In Batch – using the RUNSQLSTM with an SQL source member containing the CALL to the SQL procedure
- Using the QMQRY (Query Manager Query) – The instruction is STRQMQRQRY and the QMQRY member should contain the call
- Using Dan Riehl's EXCSQL
Debugging an SQL Stored Procedure

- To be debuggable, the procedure has to be created in a debuggable mode
- RUNSQLSTM with DBGVIEW(*LIST) or DBGVIEW(*SOURCE)
- DBGVIEW(*LIST) provides a C view of the code
- DBGVIEW(*SOURCE) provides an SQL view of the code
- Once the procedure is compiled, use STRDBG PGM(PROC_NAME) UPDPROD(*YES)
SQL Stored Procedure File Operation Debugging – SQLCODE

- SQLCODE is a results indicator variable affected by each database operation
- Zero value in the SQLCODE indicates success
- To see the value of the SQLCODE variable, use EVAL SQLCODE
- SQLCODE is actually part of a larger system data structure. To see it, use EVAL sqlca
SQL Modular Programming with Functions – Recycle that code!

- SQL FUNCTIONS
  - Allow creation of your own functions in the same way that you can create your own commands

- Are Different from SQL Procedures:
  - procedures can receive and return many parameter values
  - functions can receive many but will only return a single parameter value.
The Mechanics of SQL Functions

- To compile a function, use the RUNSQLSTM command, just like creating a Stored procedure.

- SQL functions compile into objects of type *SRVPGM.

- This means the function cannot be called on its own.
CREATE FUNCTION HOW_OLD (INDATE DATE)
RETURNS CHAR(8)
LANGUAGE SQL
BEGIN
DECLARE HOW_OLD CHAR(8);
DECLARE RETVAL CHAR(8);
CASE
   WHEN INDATE < CURRENT_DATE - 60 DAYS THEN
      SET RETVAL = 'VERY OLD';
   WHEN INDATE < CURRENT_DATE - 30 DAYS THEN
      SET RETVAL = 'OLD';
   ELSE
      SET RETVAL = 'FRESH';
   END CASE;
RETURN(RETVAL);
END

SELECT HOW_OLD(CURRENT_DATE - 33 DAYS) FROM SYSIBM/SYSDUMMY1
SQL Functions – A simple Example
Translating a JDE Julian Date to MDY

SQL Function Code:

```sql
CREATE FUNCTION XJDETOMDY (IN_JDE_DATE INT)
RETURNS DATE
LANGUAGE SQL
BEGIN
    DECLARE OUT_YMD DATE;
    SET OUT_YMD = DATE(CHAR(1900000+IN_JDE_DATE));
    RETURN (OUT_YMD);
END
```

Execution:

```
CYYJJJ
SELECT XJDETOMDY(105144) FROM SYSIBM/SYSDUMMY1
05/24/05
```
SQL Functions – A simple Example
Translating a MDY Date to a JDE Julian Date

SQL Function Code:

```
CREATE FUNCTION XMDYTOJDE
(IN_YMD_DATE DATE)
RETURNS INT
LANGUAGE SQL
BEGIN
    DECLARE OUT_JDE_DATE INT ;
    DECLARE OUT_JDE_PART1 CHAR(1) ;
    DECLARE OUT_JDE_PART2 CHAR(2) ;
    DECLARE OUT_JDE_PART3I INT ;
    DECLARE OUT_JDE_PART3C CHAR(3) ;

    CASE
        WHEN IN_YMD_DATE < 
            DATE('01/01/2000')
            THEN SET OUT_JDE_PART1 = '0' ;
        ELSE
            SET OUT_JDE_PART1 = '1' ;
    END CASE ;

    SET OUT_JDE_PART2 = SUBSTR(CHAR(IN_YMD_DATE), 7,2) ;

    SET OUT_JDE_PART3I =  DAYS(IN_YMD_DATE) -
                          DAYS( DATE('01/01/' || OUT_JDE_PART2 ) ) ;

    CASE
        WHEN OUT_JDE_PART3I < 10
            THEN SET OUT_JDE_PART3C = '00' ||
                               CHAR(OUT_JDE_PART3I) ;
        WHEN OUT_JDE_PART3I < 100
            THEN SET OUT_JDE_PART3C = '0' ||
                               CHAR(OUT_JDE_PART3I) ;
        ELSE
            SET OUT_JDE_PART3C = CHAR(OUT_JDE_PART3I) ;
    END CASE ;

    SET OUT_JDE_DATE = INT(OUT_JDE_PART1 ||
                           OUT_JDE_PART2 || OUT_JDE_PART3C ) ;

    RETURN (OUT_JDE_DATE) ;
END
```

Execution:

```
SELECT XMDYTOJDE(DATE('05/24/05')) FROM SYSIBM/SYSDUMMY1
105144
```
Implementing SQL Recap

- Interpreted SQL
- SQL Stored Procedures
  - Debugging
- SQL Functions
Part 4
Performance & Security

- Performance
- Data Retrieval Tips
- Security
Real life SQL Rule Number 1: Indexes, Indexes, Indexes

- SQL performance can be fantastic, but it can also be terribly slow if not coded properly or if no index is recognized by the DB2 SQL Optimizer
- Code your SQL join statements with keys that match the order of the indexes
- Look for Optimizer Suggestions
Make the most out of your indexes: The Cardinality Rule

- Most efficient indexes for SQL processing are ones that are created in order of cardinality.
  - For example: in a table containing 10,000 rows with an index composed of 3 keys:
    - First key, Company Division has 4 possible values
    - Second key, Department has 48 possible values
    - Third key, Employee has 100,000 values
- Make your index unique if you can
Surrogate Keys – Beyond Indexes

- A Surrogate Key is an arbitrary, unique numeric key

- Unique Numeric keys are fastest for index access. If your key is too long or not unique, a surrogate key can improve your access performance, especially on updates
USE CAST ONLY IF THERE IS NO OTHER SOLUTION

- SQL allows joining data with different key types using CASTING

- Practical when no other solutions but precludes the use of indexes => SLOW PERFORMANCE
**Select for Insert: Be Explicit**

- Using `SELECT *` on an insert is an exposure if you make database changes

- Explicit column selects are safer

```sql
INSERT INTO EMPLOYEE_DATA (EMPLOYEE_NBR, EMPLOYEE_LAST_NAME, SALARY_CATEGORY) SELECT * FROM EMPLOYEE_MASTER;
```

```sql
INSERT INTO EMPLOYEE_DATA (EMPLOYEE_NBR, EMPLOYEE_LAST_NAME, SALARY_CATEGORY) SELECT EM.EMPLOYEE_NBR, EM.EMPLOYEE_LAST_NAME, EM.EMPLOYEE_CATEGORY FROM EMPLOYEE_MASTER EM;
```
SQL Testing Guidelines

- Test for Scale: What works on a small sample may be a dog with a large amount of data.

- Test for number of rows: SQL processing is primarily about processing SETS of data. Make sure you create test cases where you can predict the resulting number of records, especially on JOIN statements.
Get the Most out of your WHERE Clauses

- Order your WHERE Clauses by putting the comparisons in order of efficiency:
  - =
  - >, >=, <, <=
  - LIKE
  - <>
FETCH FIRST keyword

- Limit your results with FETCH FIRST

```
SELECT * FROM CUSTOMER AORDER BY 
  A.SALES DESC FETCH FIRST 5 ROWS ONLY
```
BATCH vs Interactive

- USE BATCH when possible
- Batch mode processing is MUCH FASTER than interactive mode
Method to find the SQL Optimizer suggestions to improve performance (1 of 2)

1) Go in debug and change the job to record all activities and second level text

- STRDBG UPDPROMD(*YES)
- CHGJOB LOG(4 4 *SECLVL)
  Note: with *SECLVL, both the message text and the message help (cause and recovery) of the error message are written to the job log
Method to find the SQL Optimizer suggestions to improve performance (2 of 2)

2) Call Stored Procedure from an SQL environment

3) Review the job log and look for the following messages:

"**** Starting optimizer debug message for query"

Or

"Access path suggestion for file"

The system will typically make precise index suggestions, or not suggest at all
COMPILED vs INTERPRETED SQL

- Main advantage of interpreted code is simplicity
  - Simplicity in coding (no compiling)
- Main advantage is that compiled code allows
  - Variable manipulation
  - Do-loops
  - If-then-else constructs
  - Record-by-record processing
  - Retrieval of SQL Optimizer Messages
Promotion & Implementation Considerations

- Large scale use of SQL in production requires some promotion control planning
- SQL can be implemented without compilation – Ensure your security is setup so that you control what can enter your production SQL source files
- SQL is different from other conventional languages. Ensure your promotion control software can handle SQL code/objects
Using SQL to Retrieve Data from a REMOTE Database

- Type in "CONNECT" then press F4 in an interactive STRSQL session
- CONNECT TO RMT_SYS USER USER_NAME USING 'PASSWORD'
- Allows data retrieval with SQL from a remote iSeries
- **NOTE:** The password is visible on the screen when called interactively
SQL access on iSeries: ODBC Accessibility

- You can reach your data using SQL on Microsoft Excel and an ODBC connection
- This may be a security exposure
- Verify your ODBC security
- PowerLock and other vendors have tools to shut these down or authorize only certain users
SARBOX Considerations

- Auditors may ask:
  Can SQL make "untraceable" changes in the database?
- AGAIN: Verify your ODBC security
- Journal critical tables if audit trails are absolutely necessary
- Create your own EXCSQL command and log the commands used
- Revoke STRSQL and allow SQL access only via EXCSQL
Performance & Security Recap

- Performance
  - Indexes
  - SQL Optimizer
- Data Retrieval Tips
  - Using the SQL Optimizer
  - ODBC Access, Keyword CONNECT
- Security
  - Audit Considerations
  - ODBC Access
What's Next?

- ODBC / JDBC / ADO / DRDA / XDA Clients

Network

- Host Server
- CLI / JDBC

Dynamic
- Prepare Every Time

Static
- Compiled embedded Statements

Extended Dynamic
- Prepare once and then Reference

Native (Record I/O)

SQL

SQL Optimizer

DB2 UDB for iSeries
Questions?

Email: dambrine@tylogix.com
Website: www.tylogix.com

Good Online SQL Tutorial Website:
http://www.w3schools.com/sql/default.asp

DB2 Personal Developer Website: