SQL on IBM i:
Joins and Aggregate Functions

By Thibault Dambrine

Agenda: All about SQL!

- SQL Standard Joins
- SQL Aggregate Functions
- SQL Cross Join
- SQL Correlated Updates
- SQL Self Joins
- SQL Union Statements
- SQL Data Transformation
- SQL Performance Considerations

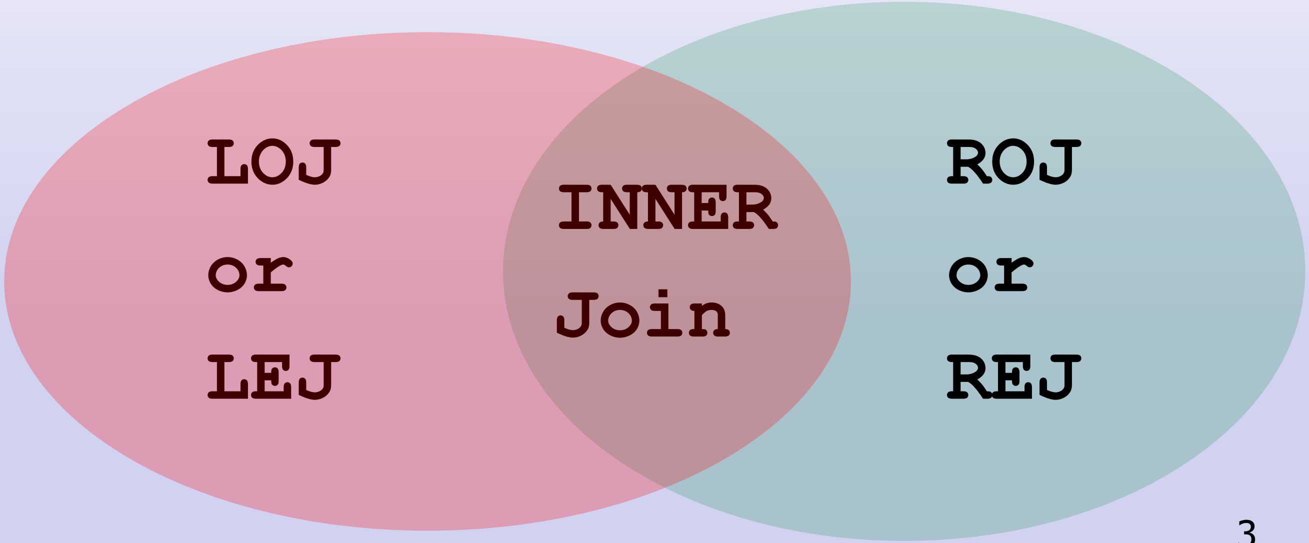
SQL "Standard" Joins

- Join or Inner Join
- Left or Right Join or Left/Right Outer Join
- Left or Right Exception Join

Key Point:

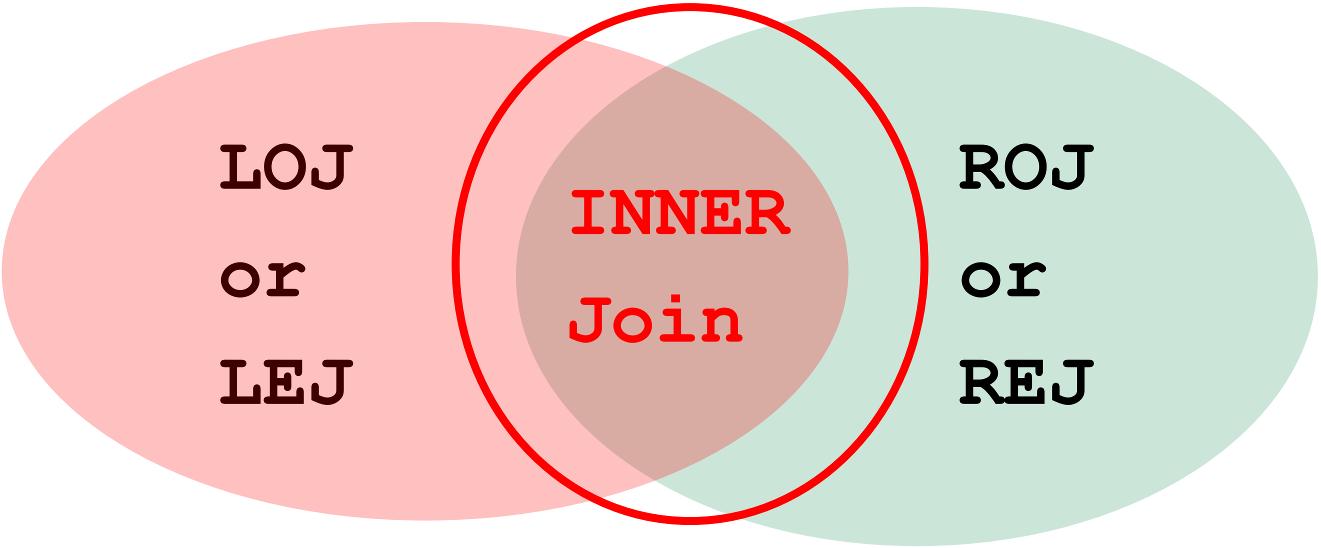
* The LEFT or RIGHT statement indicates the "root" or "main" table.

* The table on the other side of the JOIN will be the "joined" table



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



Two Base Tables for this Presentation

- Employee Table:

EMP_NBR	EMP_NAME	BEN_NBR
121	Steven Lee	111
852	Brian Evans	111
1234	John Smith	222
4567	Garth Robson	0

- Benefits Table:

BEN_NBR	EMP_BEN_DESC
111	TOP DENTAL
222	BOTTOM DENTAL
333	NEW DENTAL

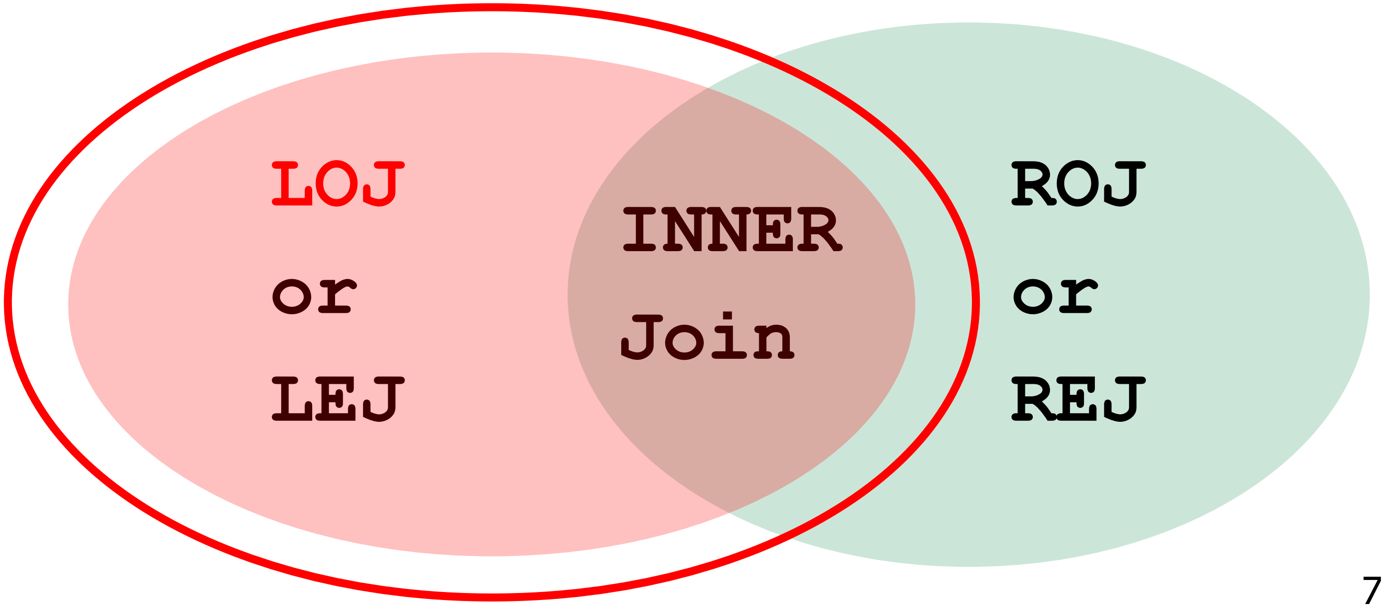
INNER JOIN Example: Getting only the exact key matches

```
SELECT
EM.EMP_NBR,
EM.EMP_NAME,
EM.BEN_NBR
BM.EMP_BEN_DESC
FROM
EMPLOYEE_MASTER EM INNER JOIN BENEFITS_MASTER BM
ON EM.BEN_NBR = BM.BEN_NBR
```

EM.EMP_NBR	EM.EMP_NAME	EM.BEN_NBR	BM.EMP_BEN_DESC
121	Steven Lee	111	TOP DENTAL
852	Brian Evans	111	TOP DENTAL
1234	John Smith	222	BOTTOM DENTAL

LEFT JOIN or LEFT OUTER JOIN

- Second Most commonly used join
- Useful when you need to see ALL from the LEFT table and what ever can be found on the right side
- The "Not Found" data on the right is padded with NULL or DEFAULT Values



LOJ Example: Getting the matches, the data from the left table and defaults from the right table if no values found

Note the use of IFNULL, which can replace un-found values with a pre-determined default (as opposed to a NULL)

```
SELECT
EM.EMP_NBR,
EM.EMP_NAME,
EM.BEN_NBR,
IFNULL(BM.EMP_BEN_DESC,'Benefits not yet allocated')
FROM
EMPLOYEE_MASTER EM LEFT OUTER JOIN BENEFITS_MASTER BM
ON EM.BEN_NBR = BM.BEN_NBR
```

LEFT JOIN or LEFT OUTER JOIN

LOJ Results **WITH** IFNULL default override

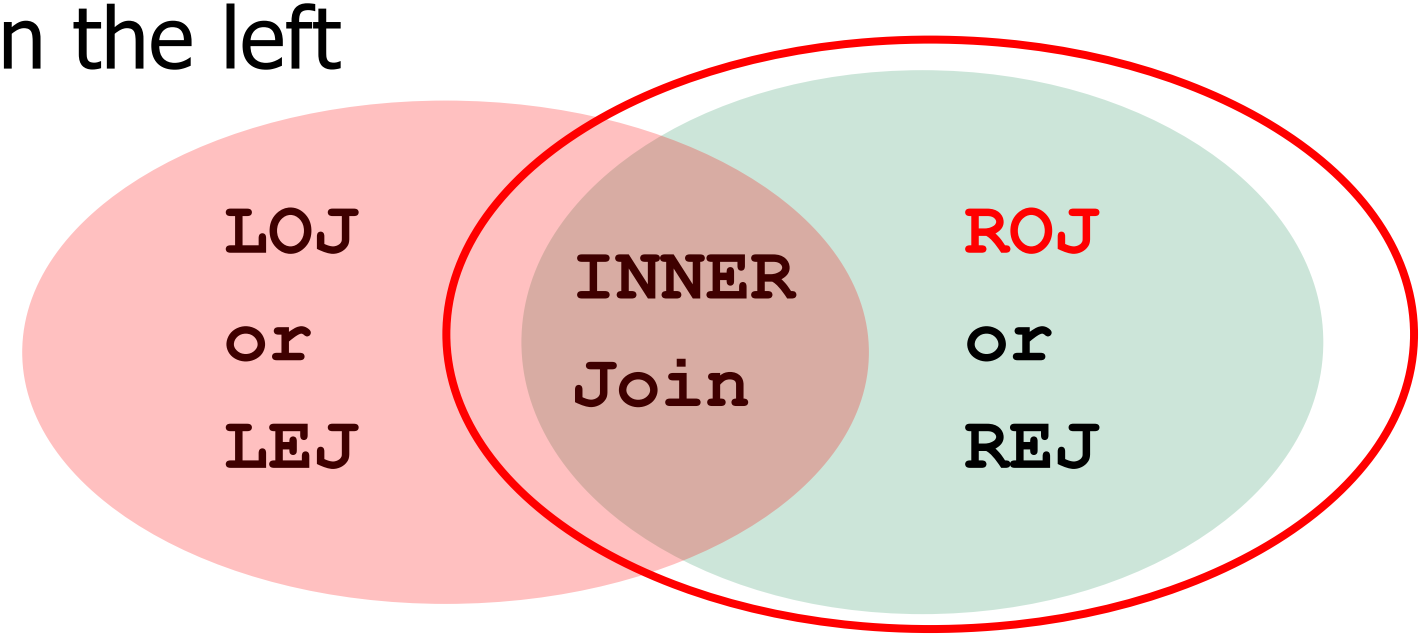
EMP_NBR	**EMP_NAME**	**BEN_NBR**	**EM.EMP_BEN_DESC**
121	Steven Lee	111	TOP DENTAL
852	Brian Evans	111	TOP DENTAL
1234	John Smith	222	BOTTOM DENTAL
4567	Garth Robson	0	**Benefits Not Yet Allocated**

LOJ Results **WITHOUT** IFNULL default override

EMP_NBR	**EMP_NAME**	**BEN_NBR**	**EM.EMP_BEN_DESC**
121	Steven Lee	111	TOP DENTAL
852	Brian Evans	111	TOP DENTAL
1234	John Smith	222	BOTTOM DENTAL
4567	Garth Robson	0	-

RIGHT JOIN or RIGHT OUTER JOIN

- Seldom used join
- Mirror image of LOJ, same rules: Bring ALL data from the right table, whatever can be found on the left



ROJ Example: Getting the matches, the data from the right table and defaults from the left table if no values found

Note:
Right Outer Join is the only change from previous example

```
SELECT
EM.EMP_NBR,
EM.EMP_NAME,
EM.BEN_NBR,
IFNULL(BM.EMP_BEN_DESC,'Benefits not yet allocated')
FROM
EMPLOYEE_MASTER EM RIGHT OUTER JOIN BENEFITS_MASTER BM
ON EM.BEN_NBR = BM.BEN_NBR
```

RIGHT JOIN or RIGHT OUTER JOIN Result

ROJ Results has **NO** IFNULL default overrides on the EMPLOYEE table, only on the BENEFITS TABLE
The NULLS WILL SHOW.

EMP_NBR	**EMP_NAME**	**BEN_NBR**	**EM.EMP_BEN_DESC**
121	Steven Lee	111	TOP DENTAL
852	Brian Evans	111	TOP DENTAL
1234	John Smith	222	BOTTOM DENTAL
-	-	-	NEW DENTAL

Multiple LEFT OUTER JOIN Method

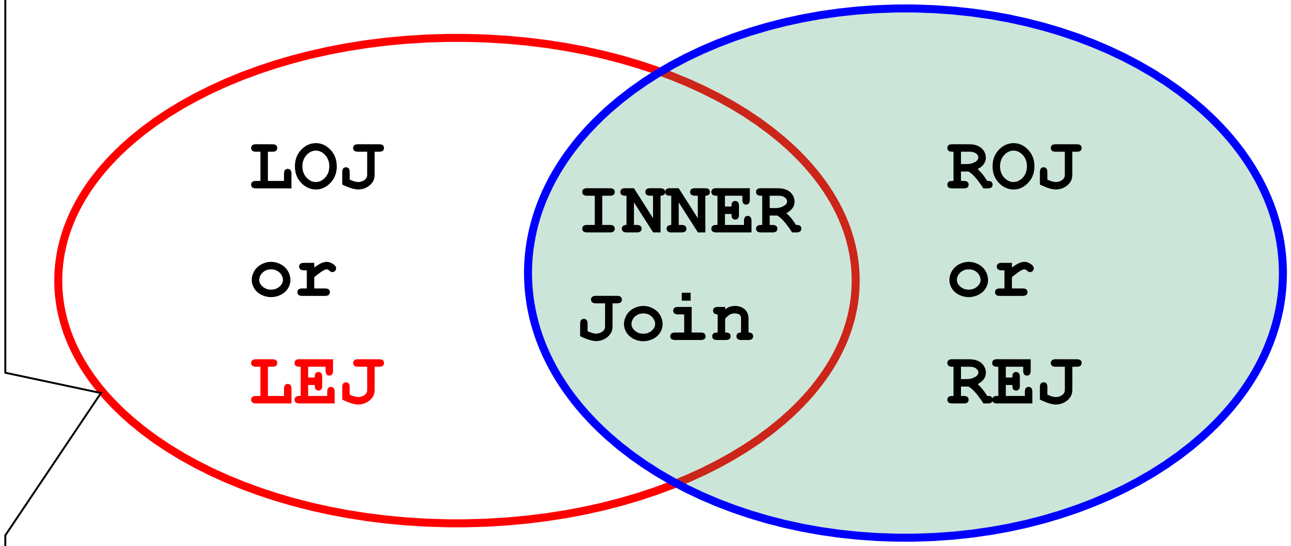
- One-to-many Left Outer Join can be a strong performer – AS LONG AS ALL TABLES ARE INDEXED!

```
INSERT INTO PRODUCT_BIG_PICTURE
SELECT PRD.*, INV.*, SLS.*, LDT.*
FROM PRODUCT_MASTER PRD
                  LEFT OUTER JOIN INVENTORY_LVL INV
                  ON PRD.PRD# = INV.PRD#
                  LEFT OUTER JOIN SALES SLS
                  ON PRD.PRD# = SLS.PRD#
                  LEFT OUTER JOIN LEAD_TIME LDT
                  ON PRD.PRD# = LDT.PRD#
```

LEFT EXCEPTION JOIN

LEJ

Returns data from the left table, minus any keys connecting to the right



- Returns only the rows from the left table that <u>do not</u> have a match in the right table
- Much more powerful than using "NOT IN" or "NOT EXISTS"

Two Base Tables for this Presentation

- An Employee Table:

EMP_NBR	EMP_NAME	BEN_NBR
121	Steven Lee	111
852	Brian Evans	111
1234	John Smith	222
4567	Garth Robson	0

- A Benefits Table:

BEN_NBR	EMP_BEN_DESC
111	TOP DENTAL
222	BOTTOM DENTAL
333	NEW DENTAL

LEFT EXCEPTION JOIN

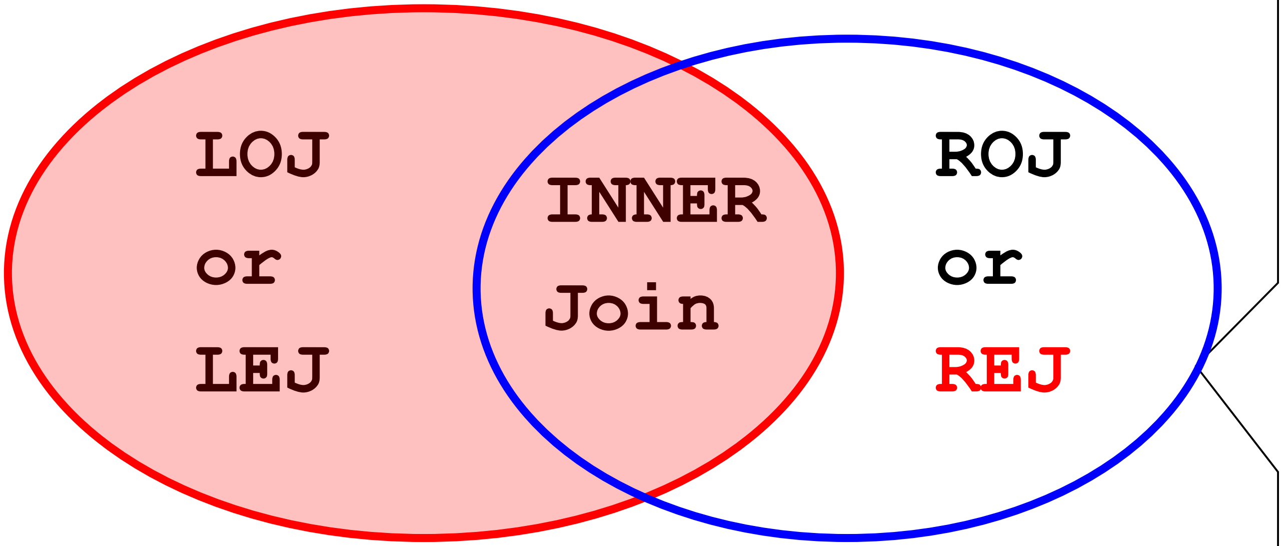
Returns only the rows from the left table that do not have a match in the right table

Example: What Employees DO NOT have Benefit Plans?

```
SELECT
EM.EMP_NBR,
EM.EMP_NAME,
EM.BEN_NBR
FROM EMPLOYEE EM LEFT EXCEPTION JOIN BENEFITS BN
ON EM.BEN_NBR = BN.BEN_NBR
```

EMP_NBR	EMP_NAME	BEN_NBR
4567	Garth Robson	0

RIGHT EXCEPTION JOIN



REJ
Returns only data from the right side, minus any keys that match on the left table

- Returns only the rows from the RIGHT table that do not have a match in the left table
- Much more powerful than using "NOT IN" or "NOT EXISTS"

Two Base Tables for this Presentation

- An Employee Table:

EMP_NBR	EMP_NAME	BEN_NBR
121	Steven Lee	111
852	Brian Evans	111
1234	John Smith	222
4567	Garth Robson	0

- A Benefits Table:

BEN_NBR	EMP_BEN_DESC
111	TOP DENTAL
222	BOTTOM DENTAL
333	NEW DENTAL

RIGHT EXCEPTION JOIN

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

Example: What Benefits plans ARE NOT USED by employees?

```
SELECT
BN.BEN_NBR,
BN.EMP_BEN_DESC
FROM EMPLOYEE EM RIGHT EXCEPTION JOIN BENEFITS BN
ON EM.BEN_NBR = BN.BEN_NBR
```

BEN_NBR	EMP_BEN_DESC
333	NEW DENTAL

Exception Join Practical Use: Spotting KEY Differences

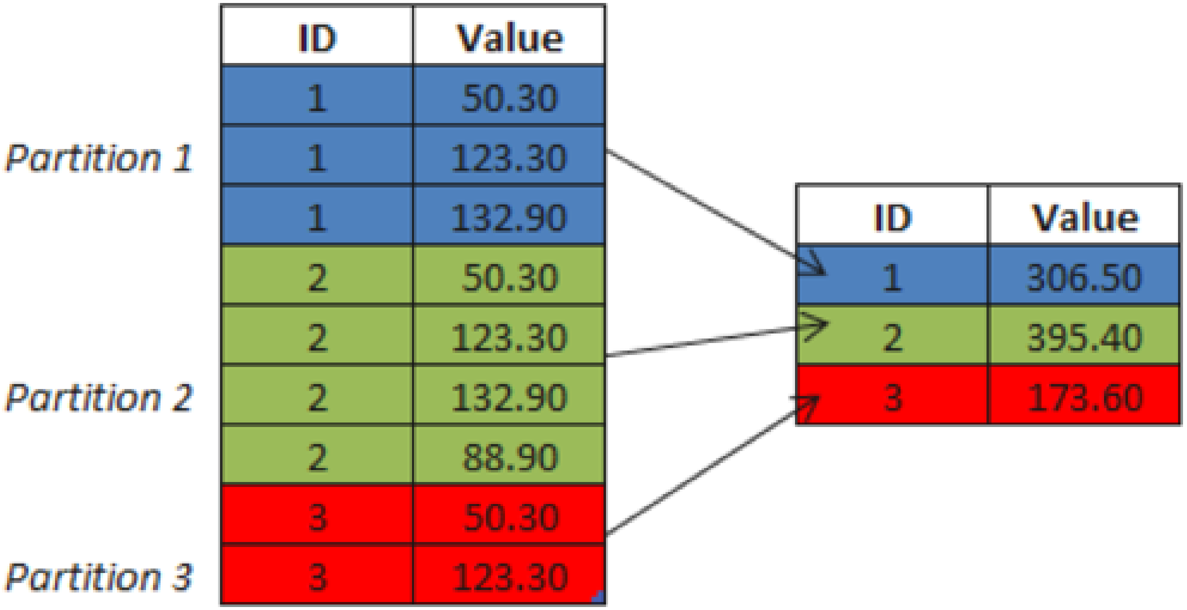
Useful for audits

Spot key differences between data sets:

- Useful for integrities
- Will pick up exactly "what the differences are
- Can be used both ways, for example:
  - “Vendor without PO's” (need to identify)
    - VENDOR_FILE Left exception Join PO_FILE
  - “PO's without Vendor” (problem!)
    - PO_FILE Left exception Join VENDOR_FILE

SQL Aggregate Functions

- GROUP BY Construct
- Distinction Between WHERE and HAVING



Partition 1

Partition 2

Partition 3

ID	Value
1	50.30
1	123.30
1	132.90
2	50.30
2	123.30
2	132.90
2	88.90
3	50.30
3	123.30

ID	Value
1	306.50
2	395.40
3	173.60

Aggregating Data with GROUP BY

- Get aggregated values, for a specified group
- Note the “Select” and the “Group by” parameters are identical

```
SELECT CITY,
       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
ORDER BY AVERAGE
```

CITY_NAME	ORDERS_COUNT	ORDERS_VALUE	AVERAGE	MIN_ORDER	MAX_ORDER
Edmonton	2324.00	45646546.00	19641.37	123.00	852.00
Red Deer	3434.00	544696445.00	158618.65	1822.00	5236.00
Calgary	4553.00	834098534.00	183197.56	268.00	7411.00
Banff	2.00	554556.00	277278.00	965.00	1258.00

WHERE and HAVING Clauses

- Use **WHERE** to compare individual row values
- Use **HAVING** to compare aggregated values

```
SELECT STORE_NAME, STORE_PROV,
       SUM(SALES) STORE_SALES
FROM STORE_INFORMATION
WHERE STORE_PROV = 'AB'
GROUP BY STORE_NAME, STORE_PROV
HAVING SUM(SALES) > 1500
```

STORE_NAME	STORE_PROV	STORE_SALES
Calgary Store	AB	3434
Red Deer Store	AB	4553
Edmonton Store	AB	8522

Finding Duplicate Data in a Table

```
SELECT NPAY.ID, COUNT(*)
    FROM NEW_PAY NPAY
    GROUP BY NPAY.ID
    HAVING COUNT(*) > 1
```

- Will find any duplicated employee ID in **NEW_PAY**, and the number of duplicates

- In effect, if the count per ID is more than 1, there is a duplicate

SQL Cross Join

- Cross Join or Cartesian Product (same as “no join criteria”)
- Uses
- Caveats

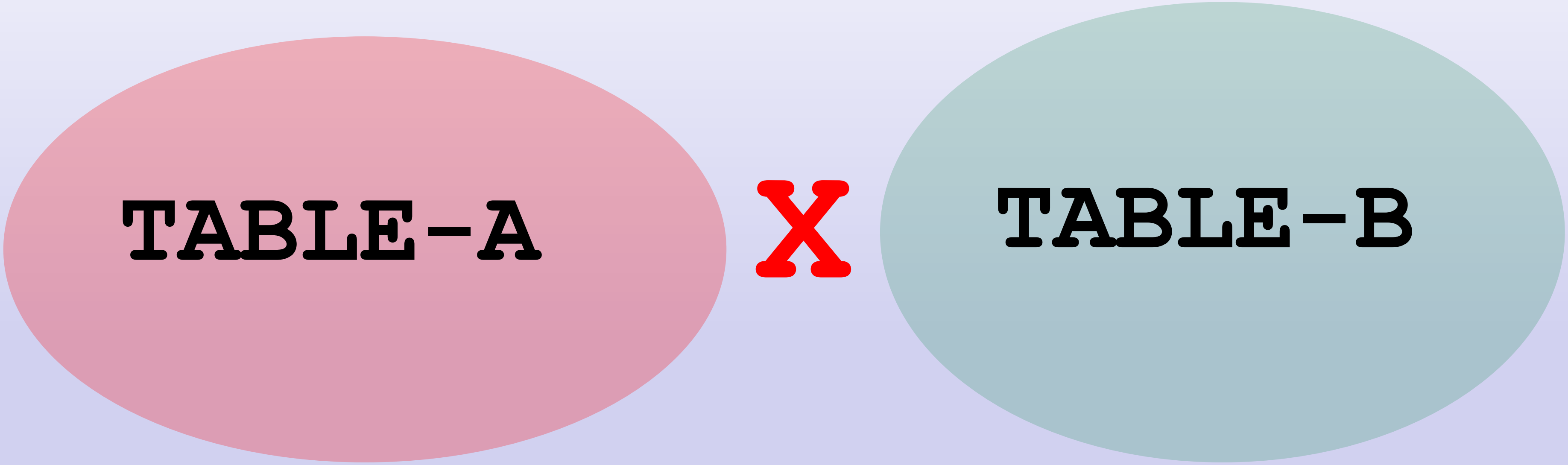


TABLE-A X TABLE-B

CROSS JOIN SYNTAX

- Also known as "CARTESIAN PRODUCT"
- Can be specified with the CROSS JOIN syntax or by listing two tables without a WHERE clause
- Returns every possible combination of the two row sets

Syntax:

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

same as:

```
SELECT * FROM FILEA, FILEB
```

CROSS JOIN or "CARTESIAN PRODUCT"

- Happens when there are no Join Criteria

- Returns every possible combination of two tables's contents combined

For TABLE_X with X Rows and
TABLE_Y with Y Rows

- The Cross Join will return X * Y Rows

CROSS JOIN EXAMPLE

EM.EMP_NBR	EM.EMP_NAME
121	Steve McPhearson
852	Brian Evans
1234	John Smith
4567	Garth Robson

BEN_NBR	EM.EMP_BEN_DESC
111	TOP DENTAL
222	BOTTOM DENTAL

CROSS JOIN Results

EM.EMP_NBR	EM.EMP_NAME	BEN_NBR	EM.EMP_BEN_DESC
121	Steve McPhearson	111	TOP DENTAL
121	Steve McPhearson	222	BOTTOM DENTAL
852	Brian Evans	111	TOP DENTAL
852	Brian Evans	222	BOTTOM DENTAL
1234	John Smith	111	TOP DENTAL
1234	John Smith	222	BOTTOM DENTAL
4567	Garth Robson	111	TOP DENTAL
4567	Garth Robson	222	BOTTOM DENTAL

CROSS JOIN or "CARTESIAN PRODUCT"

-> MOST OFTEN CONSIDERED BAD

Is there use for a CROSS-JOIN?

Exploring Sales Data with CROSS-JOIN

```
SELECT SLS.STORE_NBR, SLS.PRODUCT_NBR,
SUM(SLS.QTY_SOLD) FROM SALES_TR SLS
GROUP BY SLS.STORE_NBR, SLS.PRODUCT_NBR
```

Query Above Will return nothing for zero $ products
The query below will show product with zero sales

```
SELECT STR.STORE_NBR, PRD.PRODUCT_NBR,
SUM(IFNULL(SLS.QTY_SOLD,0)) AS TOTALSALES
FROM STORES STR CROSS JOIN PRODUCTS PRD
```

Get
All Product
and
All Store
Combinations

This cross-join will be the root side of the Left Outer Join

```
LEFT OUTER  JOIN SALES_TR SLS
ON    SLS.STORE_NBR = STR.STORE_NBR
AND   SLS.PRODUCT_NBR = PRD.PRODUCT_NBR
GROUP BY STR.STORE_NBR,  PRD.PRODUCT_NBR
```

Outer Join to
Sales
Transactions

Account ID	Year	Month 01 Amount	Month 02 Amount	Month 03 Amount	Month 04 Amount	Month 05 Amount	Month 06 Amount	Month 07 Amount	Month 08 Amount	Month 09 Amount	Month 10 Amount	Month 11 Amount	Month 12 Amount
000001	2005	16.66	27.22	38.33	49.44	60.55	71.66	82.77	93.88	104.99	15.1	16.11	17.12

Account ID	Year	Month	Month Amount
000001	2005	1	16.66
000001	2005	2	27.22
000001	2005	3	38.33
000001	2005	4	49.44
000001	2005	5	60.55
000001	2005	6	71.66
000001	2005	7	82.77
000001	2005	8	93.88
000001	2005	9	104.99
000001	2005	10	5.1
000001	2005	11	16.11
000001	2005	12	17.12

Pivoting a Table From Horizontal to Vertical Using SQL **CROSS JOIN**

H to V Table Pivot using **Cross-Join**

Pivot a 12-month table From HORIZONTAL To VERTICAL by Using a CROSS JOIN To a 12-ROW table containing numbers 1 to 12 [here named]

' MONTH_NUMERIC '

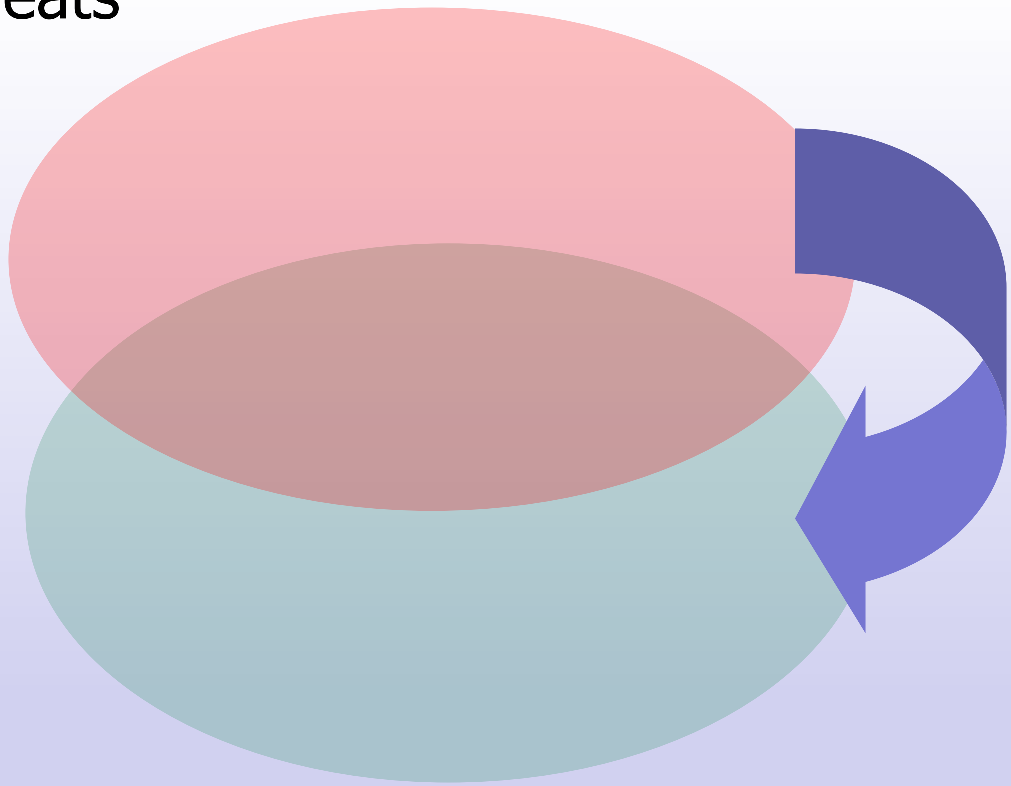
Use the CASE statement to pick the right value depending on the month processed

```
INSERT  INTO  VERTICAL
(       YEAR                ,
        ACCOUNT_ID          ,
           MONTH                ,
        NET_POSTING    )
```

```
SELECT
   YEAR                   ,
   MONTH                  ,
   ACCOUNT_ID             ,
  CASE  MONTH_VALUE
        WHEN  1   THEN  NET_01
        WHEN  2   THEN  NET_02
        WHEN  3   THEN  NET_03
        WHEN  4   THEN  NET_04
        WHEN  5   THEN  NET_05
        WHEN  6   THEN  NET_06
        WHEN  7   THEN  NET_07
        WHEN  8   THEN  NET_08
        WHEN  9   THEN  NET_09
        WHEN  10  THEN  NET_10
        WHEN  11  THEN  NET_11
        WHEN  12  THEN  NET_12
     END
 FROM    HORIZONTAL
   CROSS  JOIN   MONTH_NUMERIC  ;
```

SQL Correlated Updates & Deletes

- Updates and Deletes Based on Another Table
- The Double “WHERE” Clause
- Caveats



Updating Data in a Table Using a Correlated Query (update with join not possible for now)

```
UPDATE EMPLOYEE EM
 SET ( EM.PAY_SCALE, EM.SALARY) =
       (
         SELECT NPAY.PAY_SCALE, NPAY.SALARY
         FROM   NEW_PAY NPAY
        )
WHERE EXISTS
    (SELECT '*'
     FROM NEW_NEWPAY NPAY WHERE NPAY.ID = EM.ID )
```

- Note the use of two WHERE clauses
- **WARNING:** Will crash if either files contain duplicate keys

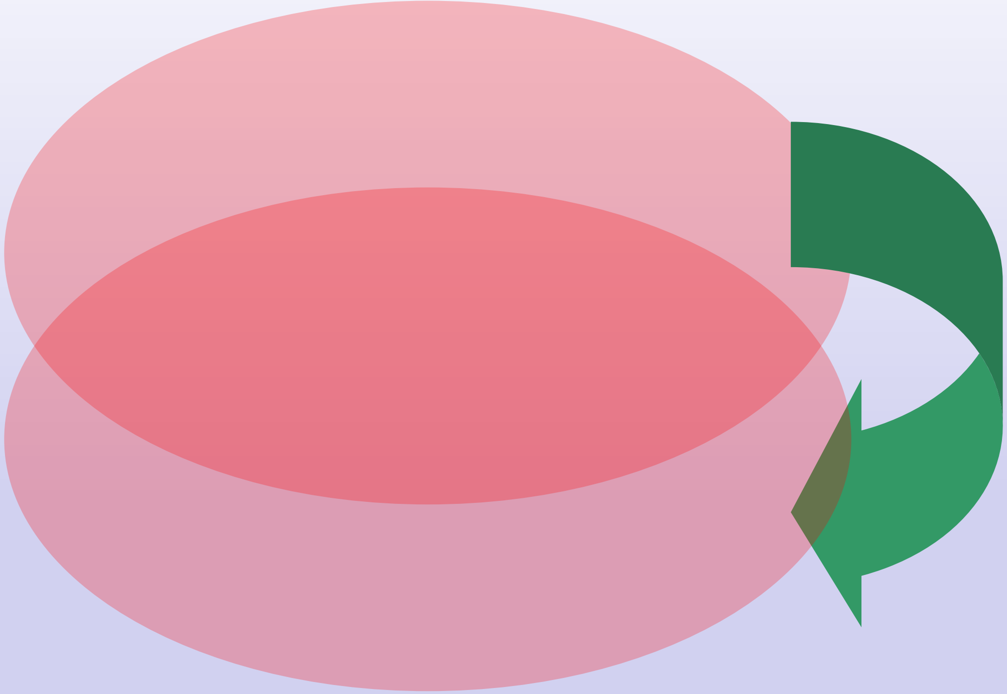
Deleting Data in a Table Using a Correlated Sub-Select
(only method currently available on DB2 for i)

```
DELETE FROM EMPLOYEE_TABLE EM
WHERE EXISTS
(SELECT '*' FROM RETIREE_TABLE RET
            WHERE RET.ID = EM.ID);
```

Note the use of TWO WHERE clauses

SQL Self Joins

- Pivoting data from Vertical to Horizontal
- Finding variations for specific values in a journal
- Finding and deleting duplicate values



Account ID	Year	Month 01 Amount	Month 02 Amount	Month 03 Amount	Month 04 Amount	Month 05 Amount	Month 06 Amount	Month 07 Amount	Month 08 Amount	Month 09 Amount	Month 10 Amount	Month 11 Amount	Month 12 Amount
000001	2005	16.66	27.22	38.33	49.44	60.55	71.66	82.77	93.88	104.99	15.1	16.11	17.12

Account ID	Year	Month	Month Amount
000001	2005	1	16.66
000001	2005	2	27.22
000001	2005	3	38.33
000001	2005	4	49.44
000001	2005	5	60.55
000001	2005	6	71.66
000001	2005	7	82.77
000001	2005	8	93.88
000001	2005	9	104.99
000001	2005	10	5.1
000001	2005	11	16.11
000001	2005	12	17.12

Pivoting a Table From Vertical to Horizontal Using SQL by **JOINING A FILE TO ITSELF**

V to H Table Pivot using **Self-Join** (1 of 2)

- Principle: Join the table to itself 12 times to spread the data sideways for 12 months

```
INSERT INTO HORIZONTAL
(
   YEAR      ,
      ACCOUNT_ID,
      NET_01    ,
      NET_02    ,
      NET_03    ,
      NET_04    ,
      NET_05    ,
      NET_06    ,
      NET_07    ,
      NET_08    ,
      NET_09    ,
      NET_10    ,
      NET_11    ,
      NET_12    ,
)
```

```
SELECT
 V01.YEAR                         ,
 V01.ACCOUNT_ID                   ,
 V01.NET_POSTING                  ,
 IFNULL(V02.NET_POSTING, 0)   ,
 IFNULL(V03.NET_POSTING, 0)  ,
 IFNULL(V04.NET_POSTING, 0)  ,
 IFNULL(V05.NET_POSTING, 0)  ,
 IFNULL(V06.NET_POSTING, 0)  ,
 IFNULL(V07.NET_POSTING, 0)  ,
 IFNULL(V08.NET_POSTING, 0)  ,
 IFNULL(V09.NET_POSTING, 0)  ,
 IFNULL(V10.NET_POSTING, 0)  ,
 IFNULL(V11.NET_POSTING, 0)  ,
 IFNULL(V12.NET_POSTING, 0)
```

V to H Table Pivot using **Self-Join** (2 of 2)

```
FROM VERTICAL V01

LEFT OUTER JOIN VERTICAL V02
on    V01. YEAR         = V02. YEAR
and   V01. ACCOUNT_ID   = V02. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V03
on    V01. YEAR         = V03. YEAR
and   V01. ACCOUNT_ID   = V03. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V04
on    V01. YEAR         = V04. YEAR
and   V01. ACCOUNT_ID   = V04. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V05
on    V01. YEAR         = V05. YEAR
and   V01. ACCOUNT_ID   = V05. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V06
on    V01. YEAR         = V06. YEAR
and   V01. ACCOUNT_ID   = V06. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V07
on    V01. YEAR         = V07. YEAR
and   V01. ACCOUNT_ID   = V07. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V08
on    V01. YEAR         = V08. YEAR
and   V01. ACCOUNT_ID   = V08. ACCOUNT_ID
```

```
LEFT OUTER JOIN VERTICAL V09
on    V01. YEAR         = V09.YEAR
and   V01. ACCOUNT_ID   = V09.ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V10
on    V01. YEAR         = V10.YEAR
and   V01. ACCOUNT_ID   = V10.ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V11
on    V01. YEAR         = V11. YEAR
and   V01. ACCOUNT_ID   = V11. ACCOUNT_ID

LEFT OUTER JOIN VERTICAL V12
on    V01. YEAR         = V12. YEAR
and   V01. ACCOUNT_ID   = V12. ACCOUNT_ID


WHERE     V01.MONTH_VALUE = 01
      and V02.MONTH_VALUE = 02
      and V03.MONTH_VALUE = 03
      and V04.MONTH_VALUE = 04
      and V05.MONTH_VALUE = 05
      and V06.MONTH_VALUE = 06
      and V02.MONTH_VALUE = 07
      and V03.MONTH_VALUE = 08
      and V04.MONTH_VALUE = 09
      and V05.MONTH_VALUE = 10
      and V06.MONTH_VALUE = 11
      and V06.MONTH_VALUE = 12   ;
```

Find Value Changes in Journal Rows using **Self-Join**

This technique is useful sniff out variations within a specific field in a file journal. In this case, a margin change.

- Generate OUTFILE FILE TRJRNDLY01, which is simply ensuring the data is ordered by key and timestamp.
- Order is critical. It will ensure we can use the RRN for the next query to get the logical previous row.

Generate OUTFILE FILE TRJRNDLY01 – Ensure data is in <Value to Monitor> and Timestamp Order:

```
INSERT INTO TRJRNDLY01
SELECT * FROM TR_JOURNAL
ORDER  BY TRCOMP, TRDIVN, TRDPTN, TRCUSN, TRITEM, TRTIMSTP
```

Generate OUTFILE FILE TRJRNDLY02: - Kick out a new row for every <Value to Monitor> change

```
INSERT INTO TRJRNDLY02
SELECT
     AA.TRCOMP, AA.TRDIVN, AA.TRDPTN, AA.TRCUSN, AA.TRITEM,
     BB.TRMARGIN MARGIN_BEFORE, AA.TRMARGIN MARGIN_AFTER,
     AA.TRMARGIN - BB.TRMARGIN MARGINDIFFERENCE,
     char(BB.TRTIMSTP  ) BEFORE_TIMESTAMP,
     char(AA.TRTIMSTP  ) AFTER_TIMESTAMP,
     BB.MRPGMNAM, BB.JOBNAME
     CURRENT TIMESTAMP CURRENT_TIMESTAMP
FROM TRJRNDLY01 AA INNER JOIN TRJRNDLY01 BB
ON   RRN(AA)-1 = RRN(BB)
     AND AA.TRCOMP   =  BB.TRCOMP
     AND AA.TRDIVN   =  BB.TRDIVN
     AND AA.TRDPTN   =  BB.TRDPTN
     AND AA.TRCUSN   =  BB.TRCUSN
     AND AA.TRMARGIN <> BB.TRMARGIN
```

Values extracted:

- Keys,
- Margins before and after + difference
- Program that did the change
- Data Time Stamps
- Current Time Stamp
- Program Name

File joined to itself,
current record to previous record

Extract Criteria:
Where a change was made on margin

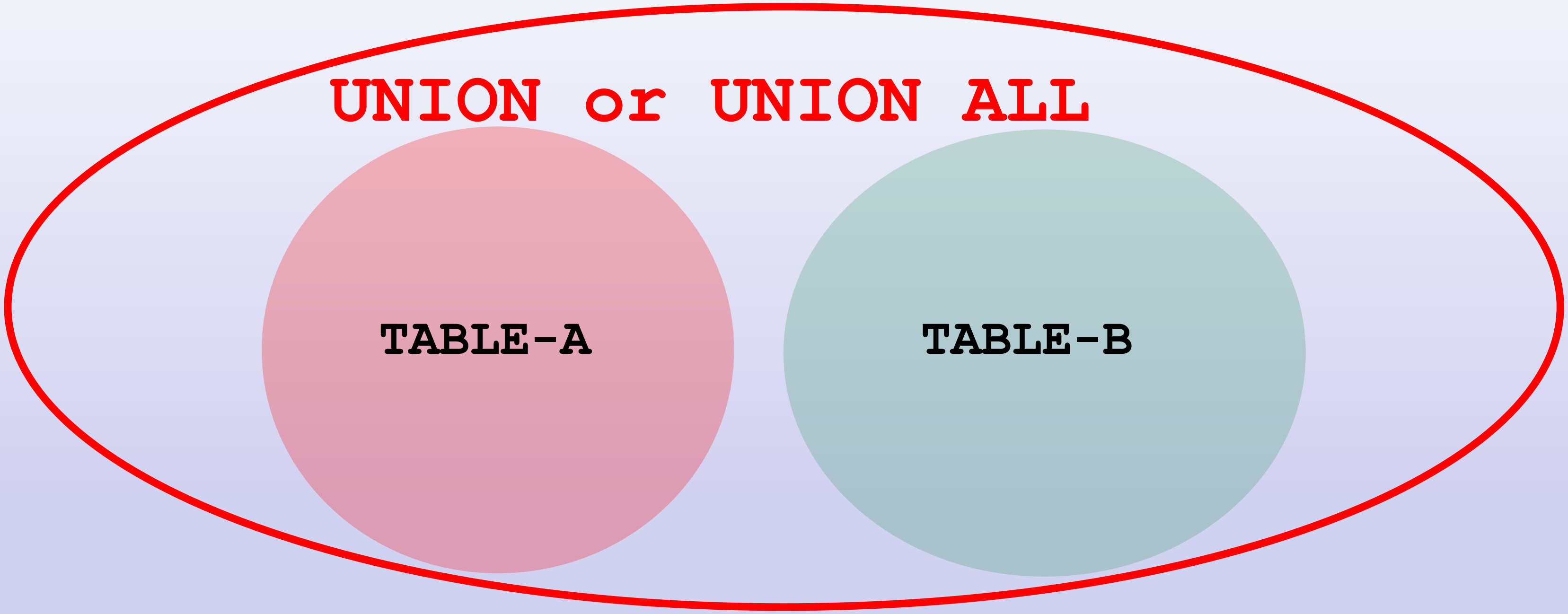
Removing Duplicate Rows In A Table using a self Correlated Sub-Select

```
DELETE FROM NEW_PAY NPAY1
WHERE RRN(NPAY1) <
 (
    SELECT MAX( RRN(NPAY2) )
    FROM NEW_PAY NPAY2
    WHERE
      NPAY1.ID = NPAY2.ID)
 )
```

- Note the use of the MAX clause
- Note the use of Correlation Names **NPAY1** and **NPAY2** - attacking the same table twice with two different correlated names

SQL Union Statements

- UNION (distinct)
- UNION ALL (all data)
- Targeting a DB2 Database File Member in SQL



UNION

- Returns data from two sets of data
- The data from both SELECTS must be of the same format
- NOTE:

  UNION **RETURNS DISTINCT VALUES ONLY**

```
SELECT EMPLOYEE_NUMBER, FIRST_NAME, LAST_NAME FROM
    ALBERTA/EMPLOYEE_TABLE
 UNION
SELECT EMPLOYEE_NUMBER, FIRST_NAME, LAST_NAME FROM
    NOVASCOTIA/EMPLOYEE_TABLE
```

UNION ALL

- Returns data from two sets of data
- The data from both SELECTS must be of the same format
- - NOTE: UNION ALL **RETURNS ALL VALUES, REGARDLESS OF DUPLICATES**

```
SELECT EMPLOYEE_NUMBER, FIRST_NAME, LAST_NAME FROM
    ALBERTA/EMPLOYEE_TABLE
 UNION ALL
SELECT EMPLOYEE_NUMBER, FIRST_NAME, LAST_NAME FROM
    NOVASCOTIA/EMPLOYEE_TABLE
```

Using UNION with Multi-Member (conventional iSeries) FILES with SQL

SQL allows the targeting of individual members with the use of an ALIAS

```
CREATE ALIAS LIBRARY1/SLSHST1999
    FOR LIBRARY1/SALESHIST(HST_1999)

CREATE ALIAS LIBRARY1/SLSHST2000
    FOR LIBRARY1/SALESHIST(HST_2000)
```

Using UNION to retrieve all members data

```
SELECT * FROM LIBRARY1/SLSHST1999
UNION ALL
SELECT * FROM LIBRARY1/SLSHST2000
ORDER BY SALES_DATE
```

SQL Transformation – Data & Types

- Using Case
- Casting Syntax
- Joining with Cast Keys
- Casting to int using unreliable Character Data
- Casting numeric data: Digits vs. Char

Data Transformation: Using CASE

- Evaluated in the order listed
- Note: Will yield a NULL if no ELSE default is specified

```
SELECT ET.EMPLOYEE_NO, ET.FIRST_NAME, ET.LASTNAME,
CASE
      WHEN ET.YEARS_OF_SERVICE > 30
            THEN 'ELIGIBLE FOR RETIREMENT'
      WHEN ET.YEARS_OF_SERVICE > 15
            THEN '15 YEARS OR LESS TO GO!'
      ELSE 'TAKE A DEEP BREATH!'
END
FROM EMPLOYEE_TABLE ET
```

Type Transformation: Using CAST (Two different syntaxes)

INT to CHAR using the "CAST" operand:

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

CHAR to INT using the "CAST" operand:

```
SELECT
INT(SUBSTRING(TELEPHONE, 1,3)
    || SUBSTRING(TELEPHONE, 5,4) ) INT_TEL_NO
FROM FILEA
```

Joining Tables With Incompatible Keys using CAST

Joining with Cast Values

```
SELECT
LT.FIRST_NAME,
LT.LAST_NAME,
LT.TELEPHONE
FROM   LOCAL_NAMES_TABLE
LT INNER JOIN COMPARE_TABLE CT
ON INT(SUBSTRING(LT.TELEPHONE, 1,3)
       || SUBSTRING(LT.TELEPHONE, 5,4))
     = CT.TELEPHONE#
```

Caveat!
Beware of
performance
hit with
JOINS using CAST

Dealing with **Unreliable Numeric Data** Stored in a Character Column

- Storing Numeric Data in an Character Column is makes for UNRELIABLE JOINS
- Sometimes, you just have no choice

```
SELECT * FROM FILEAA AA
     LEFT OUTER JOIN FILEBB BB
     ON
                 -- the case statement will determine if the
     CASE WHEN   -- values within the column are purely numeric
               ( -- or not
             LOCATE(SUBSTR(AA.CHAR_PO_NUMBER,1, 1),'0123456789') = 0
          OR LOCATE(SUBSTR(AA.CHAR_PO_NUMBER,2, 1),'0123456789') = 0
          OR LOCATE(SUBSTR(AA.CHAR_PO_NUMBER,3, 1),'0123456789') = 0
          OR LOCATE(SUBSTR(AA.CHAR_PO_NUMBER,4, 1),'0123456789') = 0
          OR LOCATE(SUBSTR(AA.CHAR_PO_NUMBER,5, 1),'0123456789') = 0
               )
     THEN 0     -- default value (there is non-numeric data)
     ELSE INT(AA.CHAR_PO_NUMBER)   -- valid numeric value
     END

     = BB.NUMERIC_PO_NUMBER
```

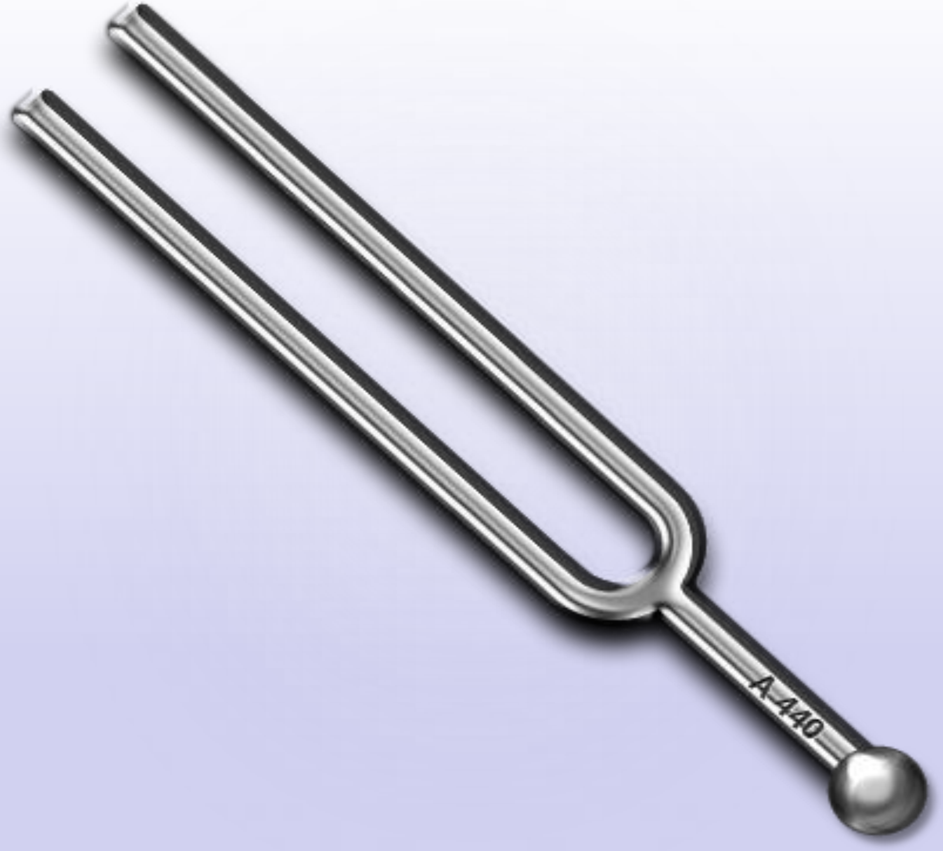
Casting Numeric Data into Characters

- CHAR, DIGITS does not

- With a NUMERIC(5,0) value equal to 00888:
  - **CHAR** (NUMVAR1) will yield   '888'
    - Strips leading zeros
  - **DIGITS**(NUMVAR1) will yield '00888'
    - Includes leading zeros

SQL Performance Considerations

- Index Awareness
- Correlated Sub-Selects vs Joins
- Cascaded Joins
- Performance Checklist



Index Awareness

Indexes = Join Performance

- Be aware of the instances where DB2 will not use an index
  - Data type conversions, casts
  - Formulas in a join
  - Use of like (or not like) patterns
  - Use of in (or not in) patterns

Correlated Sub-Selects vs. Joins

This correlated sub-query will look for product ID's that DID have sales

```
SELECT PRD.ID FROM PRODUCT_TBL PRD
WHERE EXISTS
( SELECT SLS.ID FROM SALES_TBL SLS
  WHERE SLS.ID = PRD.ID )
```

WHERE EXISTS

Can be re-written as a INNER JOIN:

```
SELECT PRD.ID FROM PRODUCT_TBL PRD
INNER JOIN SALES_TBL SLS
ON SLS.ID = PRD.ID
```

Correlated Sub-Selects vs. Joins

This correlated sub-query will look for product ID's that did NOT have sales

```
SELECT PRD.ID FROM PRODUCT_TBL PRD
WHERE NOT EXISTS
( SELECT SLS.ID FROM SALES_TBL SLS
  WHERE SLS.ID = PRD.ID )
```

WHERE NOT EXISTS

Can be re-written as a LEFT EXECPTION JOIN:

```
SELECT PRD.ID FROM PRODUCT_TBL PRD
LEFT EXCEPTION JOIN SALES_TBL SLS
ON SLS.ID = PRD.ID
```

Joins vs. Sub-Queries

Rule of thumb:

- Joins are more efficient than Correlated sub-queries

Exception:

- When the sub-query contains one or more aggregates and it is not correlated

Joins vs. Sub-Selects – Aggregated

A non-correlated Sub-Select can be the best way to get the desired results

Example: Find above average sales performance:

```
SELECT TS1.SALESMAN, TS1.SALES,
FROM TOTAL_SALES TS1
WHERE TS1.SALES >
(SELECT AVG(TS2.SALES)FROM TOTAL_SALES TS2)
```

The AVERAGE aggregate function is performed ONLY ONCE for the entire query

Beware of Cascaded Joins: Break it up! (**Slow Join** Problem!)

- Proverbial "Forever Processing" Join:
- Files BB, CC, DD are all intertwined!

```
INSERT INTO FILEA
SELECT BB.* FROM FILEB BB
INNER  JOIN FILEC CC
   ON BB.KEYB = CC.KEYC
LEFT EXCEPTION JOIN FILED DD
   ON  CC.KEY2 = DD.WORK_KEY

 WHERE DD.WORK_KEY NOT LIKE '%DW%'
   AND DD.DW_ROW_TYPE NOT IN ('R','P')
```

Join from FILEB to FILEC
from FILEC to FILED
VERY EXPENSIVE!

LIKE and
NOT IN
Operations
EXPENSIVE!

Performance Considerations: Break it up!   (Solution Part 1)

- Use an INDEXED WORKFILE to split the load into manageable chunks
- First, minimize the negative effect of "LIKE" and "NOT IN"

```
INSERT INTO DDWORKFILE
SELECT DD.* FROM FILED DD
WHERE DD.WORK_KEY NOT LIKE '%DW%'
AND DD.DW_ROW_TYPE NOT IN ('R','P')
```

Performance Considerations: Break it up!   (Solution Part 2)

- Again, Use an INDEXED WORKFILE to split the load into manageable chunks
- Second, Create a new intermediate work file for the other join

```
INSERT INTO CCWORKFILE
SELECT CC.* FROM FILEC CC
LEFT EXCEPTION JOIN DDWORKFILE DD
   ON  CC.KEY2 = DD.WORK_KEY
```

Performance Considerations: Break it up!   (Solution Part 3)

- The new join will use only keys, NO OTHER SELECTION CRITERIA

```
INSERT INTO FILEA
SELECT BB.* FROM FILEB BB
INNER  JOIN CCWORKFILE CC
   ON BB.KEYB = CC.KEYC
```

- 3 Simple joins are
  - - more efficient
  - - quicker to execute
- Than one complicated SQL statement

SQL Performance Checklist

- Are CAST operations used in joins?
- Are LIKE or NOT IN operations used in joins?
- Are there Formulas in WHERE clauses?
- Is the technique optimum?
  - Is Join vs. Correlated sub-query used?
  - Are there cascaded joins?
- Could the process be broken up in smaller pieces?
- Are there proper INDEXES?
- Did you test with life-size samples?

Recap For This Presentation:

- Joins
  - Inner, Outer, Exception, Union, Union All
  - Updates and Deletes – note the "double WHERE"
- Casting & Case
  - Casts and Case can be used in joins (beware of performance!)
- Performance
  - Joins vs correlated sub-selects
  - Pay attention to keys indexes, formulas in joins
  - Pay attention to cascaded joins

Questions

Email: dambrine@tylogix.com

See the SQL Section in www.tylogix.com