CS108: Advanced Database

- Database Programming

Lecture 04:

JOINs

Overview

- The process of combining tables into one result set

  by using the various forms of the JOIN clause:

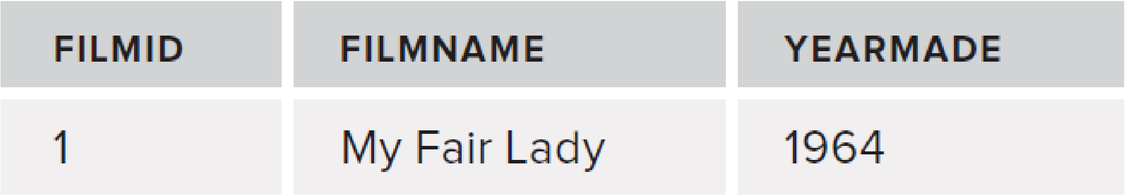
  - INNER JOIN
  - OUTER JOIN (both LEFT and RIGHT)
  - FULL JOIN
  - CROSS JOIN

Combining Table Data With Joins

- We’ll frequently run into situations in which not all of the information that we want is in one table.
- A JOIN, joins the information from two tables together into one result set.
- A result set can be thought as being a virtual table.
    - It has both columns and rows, and the columns have data types.
- All JOINs match one record up with one or more other records to make a record that is a superset created by the combined columns of both records.

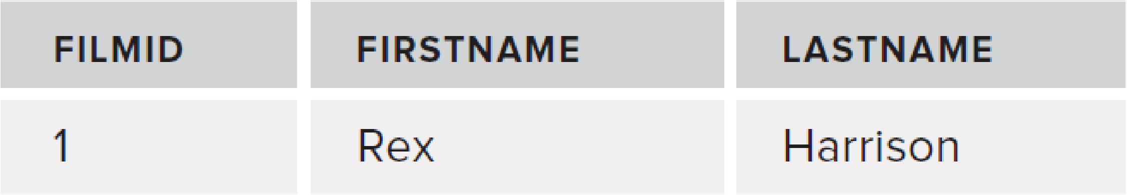
Combining Table Data With Joins

- For example, take a look at a record from a table called Films:



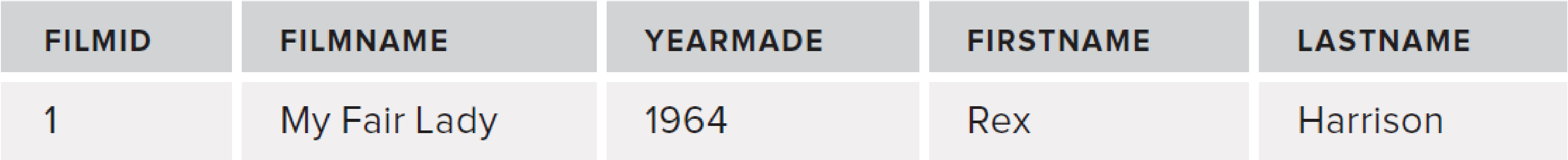
FILMID	FILMNAME	YEARMADE
1	My Fair Lady	1964

- Now follow that up with a record from a table called Actors:



FILMID	FIRSTNAME	LASTNAME
1	Rex	Harrison

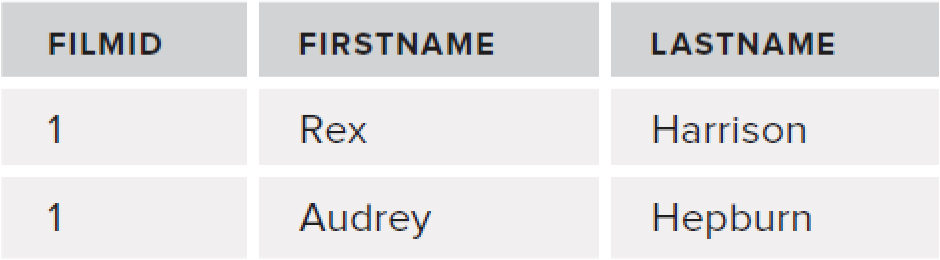
- With a JOIN, we can create one record from these two records found in totally separate tables:



FILMID	FILMNAME	YEARMADE	FIRSTNAME	LASTNAME
1	My Fair Lady	1964	Rex	Harrison

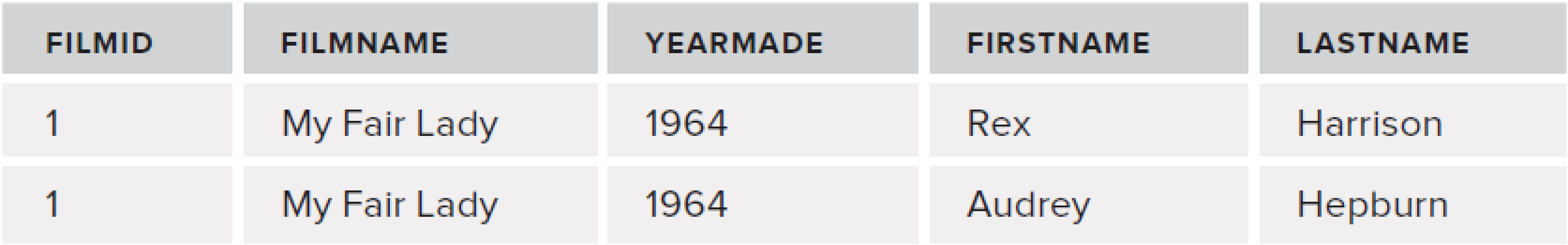
Combining Table Data With Joins

- JOINs can be one-to-many (Based on the JOIN types).
- For example, another record is added to the Actors table:



FILMID	FIRSTNAME	LASTNAME
1	Rex	Harrison
1	Audrey	Hepburn

- When we join that to the Films table:



FILMID	FILMNAME	YEARMADE	FIRSTNAME	LASTNAME
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn

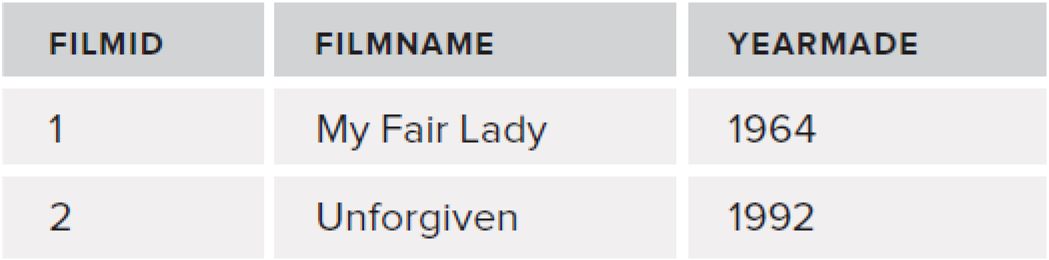
- JOIN by matching up the FilmID field from the two tables to create one record out of two.

Selecting Matching Rows with INNER JOIN

- INNER JOINs are far and away the most common kind of JOIN
- INNER JOINGs match records together based on one or more common fields, as do most JOINs
- INNER JOINs return only the records where there are matches for whatever field(s) you have said are to be used for the JOIN

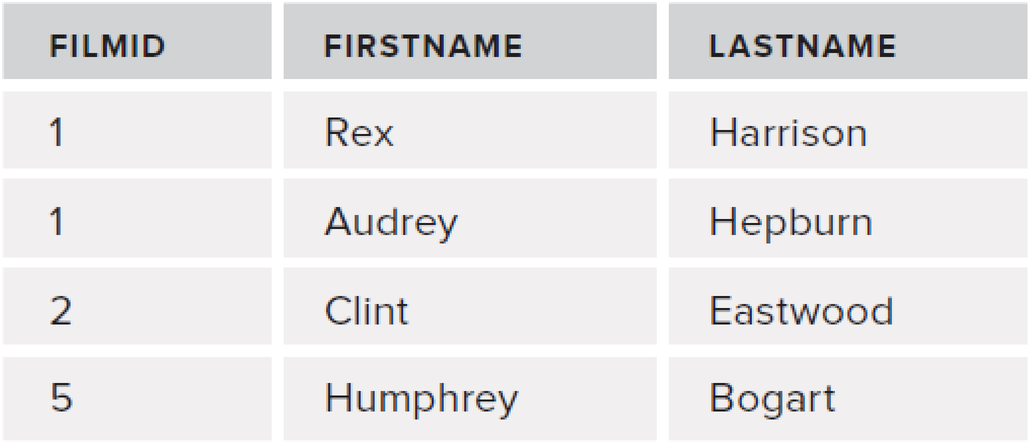
Example: INNER JOIN

- The Films table:



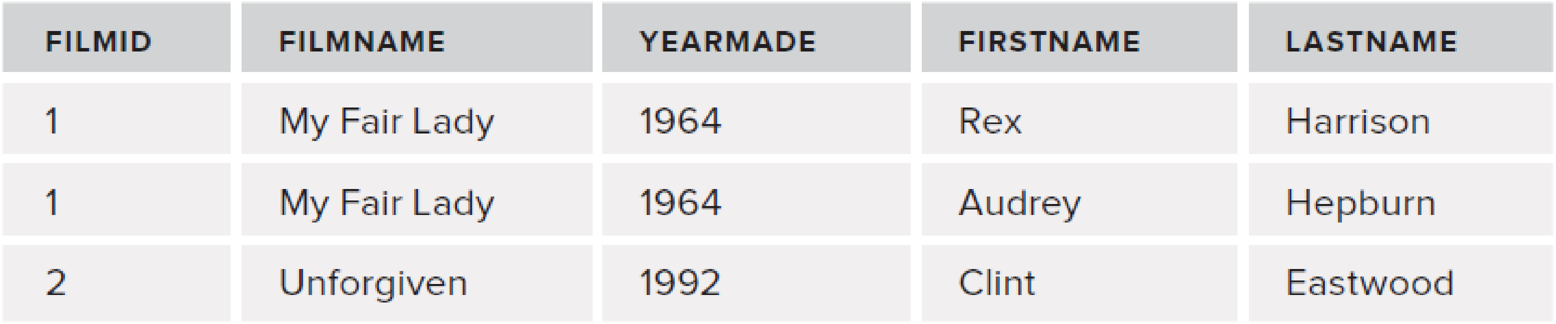
FILMID	FILMNAME	YEARMADE
1	My Fair Lady	1964
2	Unforgiven	1992

- The Actors table:



FILMID	FIRSTNAME	LASTNAME
1	Rex	Harrison
1	Audrey	Hepburn
2	Clint	Eastwood
5	Humphrey	Bogart

- Using an INNER JOIN, the result set would look like this:



FILMID	FILMNAME	YEARMADE	FIRSTNAME	LASTNAME
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood

INNER JOIN

- The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables
- The INNER JOIN syntax looks something like this:

```
SELECT <select list>
FROM   <first_table>
<join_type> <second_table>
   [ON <join_condition>]
```

- For example:

```
SELECT *
FROM       Person.Person
INNER JOIN HumanResources.Employee
        ON Person.Person.BusinessEntityID =
           HumanResources.Employee.BusinessEntityID
```

INNER JOIN

- When we want to refer to a column where the column name exists more than once in the JOIN result, we must fully qualify the column name.
- We can do this in one of two ways:
  - Provide the name of the table that the desired column is from, followed by a period and the column name (Table.ColumnName)
  - Alias the tables, and provide that alias, followed by a period and the column name (Alias.ColumnName)

INNER JOIN

- Be aware that using an alias is an all-or-nothing proposition. Once we decide to alias a table, we **MUST USE** that alias in every part of the query.
- For example,

```
SELECT     pbe.*, HumanResources.Employee.BusinessEntityID
FROM       Person.BusinessEntity   pbe
INNER JOIN HumanResources.Employee hre
        ON pbe.BusinessEntityID = hre.BusinessEntityID
```

- This may seem like it should run fine, but it will give you an error:

```
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier
“HumanResources.Employee.BusinessEntityID” could not be bound.
```

Example: A Simple INNER JOIN

- A small query:

```
SELECT pbe.BusinessEntityID, hre.JobTitle, pp.FirstName, pp.LastName
FROM       Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
        ON pbe.BusinessEntityID = hre.BusinessEntityID
INNER JOIN Person.Person pp
        ON pbe.BusinessEntityID = pp.BusinessEntityID
```

- This yields a pretty simple result set:

```
BusinessEntityID JobTitle                          FirstName      LastName
---------------- --------------------------------- -------------- ------------
1                Chief Executive Officer           Ken            Sánchez
2                Vice President of Engineering     Terri          Duffy
3                Engineering Manager               Roberto        Tamburello
...
(290 row(s) affected)
```

- Person.BusinessEntity: customer or vendor, or employee;

  HumanResources.Employee: employee; Person.Person: Name

An INNER JOIN is Like a WHERE Clause

- In the INNER JOINs will work like for any JOIN type, i.e., the column ordering and aliasing is exactly the same for any JOIN.
- The part that makes an INNER JOIN different from other JOINs is that it is an exclusive JOIN,
  - It excludes all records that don’t have a value in both tables (the first named, or left table, and the second named, or right table).

An INNER JOIN is Like a WHERE Clause

- Consider the following tables:

Sales.Customer
CustomerID
PersonID
StoreID
TerritoryID
AccountNumber
rowguid
ModifiedDate

Person.Person	
BusinessEntityID	Suffix
PersonType	EmailPromotion
NameStyle	AdditionalContactInfo
Title	Demographics
FirstName	rowguid
MiddleName	ModifiedDate
LastName	

- The Customer.PersonID column has a foreign key, Indeed, the PersonID ties back to the BusinessEntityID in the Person.Person table.

An INNER JOIN is Like a WHERE Clause

- For Example, the number of person is

```
SELECT COUNT(*)
FROM   Person.Person;
```

```
-----------
19972

(1 row(s) affected)
```

- We asked to produce a list of names (columns in Person.Person) associated with at least one ***customer*** (record in Sales.Customer) and the account number (column in Sales.Customer) of the customers they are associated with.

- A list of names associated with at least one customer and the account number of the customers they are associated with

```
SELECT CAST(LastName + ', ' + FirstName AS VARCHAR(35)) AS Name,
       AccountNumber
FROM   Person.Person  pp
JOIN   Sales.Customer sc
  ON   pp.BusinessEntityID = sc.PersonID
```

We did not use the INNER keyword in the query. That is because an INNER JOIN is the default JOIN type.

```
Name                                AccountNumber
----------------------------------- -------------
Abel, Catherine                     AW00029485
Abercrombie, Kim                    AW00029486
Acevedo, Humberto                   AW00029487
...
Zimmerman, Tiffany
Zukowski, Jake

(19119 row(s) affected)
```

Several contacts have been left out in the Person table, because they aren't customers.
Once again, the key to INNER JOINs is that they are exclusive.

More Join Example

- **Composite Joins** - A composite join is simply a join based on a predicate that involves **more than** one attribute from each side.

```
SELECT *
FROM Table1 AS T1
JOIN Table2 AS T2
     ON  T1.col1 = T2.col1
     AND T1.col2 = T2.col2
```

- **Non-Equi Joins** - When a join condition involves any operator besides equality, the join is said to be a non-equi join

```
SELECT P1.BusinessEntityID, P1.FirstName, P1.LastName,
       P2.BusinessEntityID, P2.FirstName, P2.LastName
FROM Person.Person AS P1
JOIN Person.Person AS P2
  ON P1.BusinessEntityID < P2.BusinessEntityID;
```

```
BID1   BID2
------ ------
1      2
1      3
2      3
1      4
...
```

Retrieving More Data with OUTER JOIN

- OUTER JOIN is something of the exception rather than the rule, because:
  - More often than not, we’ll want the kind of exclusiveness that an inner join provides
  - Many SQL writers learn inner joins and never go any further - they simply don’t understand the outer variety
  - There are often other ways to accomplish the same thing
  - They are often simply forgotten about as an option

Retrieving More Data with OUTER JOIN

- INNER JOINs are exclusive in nature, OUTER and FULL joins are inclusive.
- OUTER JOINs can also often speed performance when used instead of nested subqueries.
- A join having sides, a left and a right
  - The first named table is considered to be on the left and the second named table is considered to be on the right
  - INNER JOINs both sides are always treated equally

OUTER JOIN

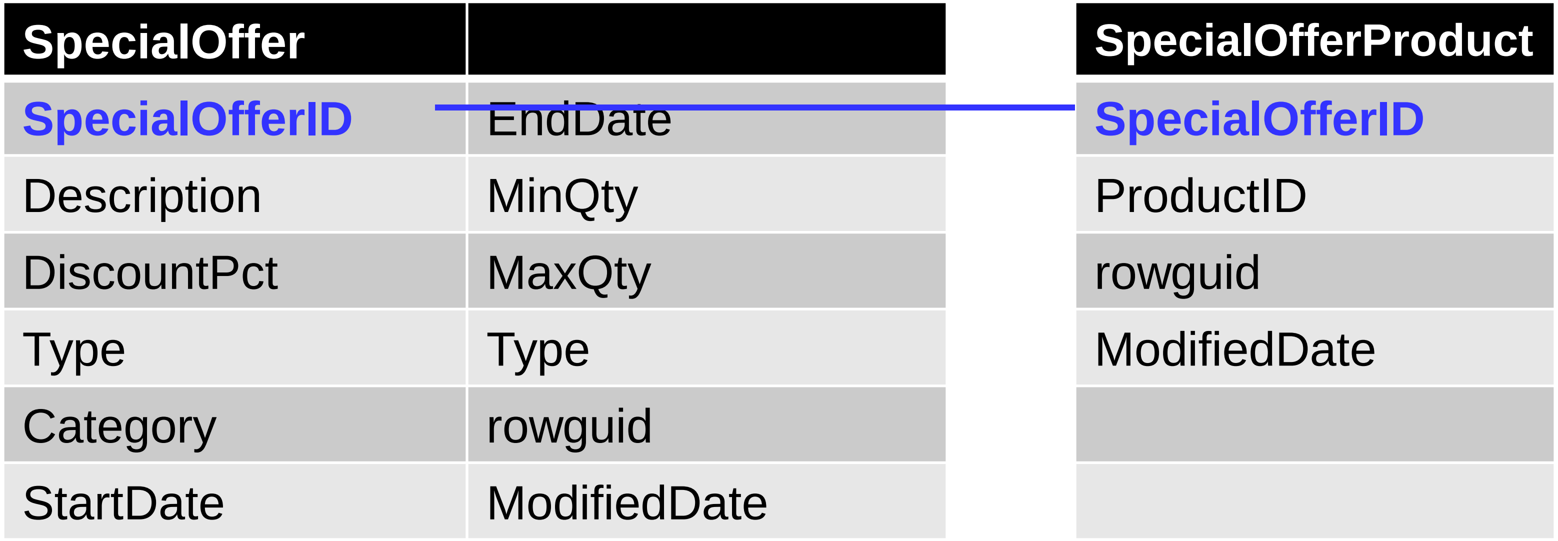
- OUTER JOINs are inclusive in nature.
- The syntax of OUTER JOIN is:

```
SELECT <SELECT list>
FROM <the table you want to be the “LEFT” table>
  <LEFT|RIGHT> [OUTER] JOIN <table you want to be the “RIGHT” table>
  ON <join condition>
```

- A LEFT OUTER JOIN includes all the information from the table on the left, and
- A RIGHT OUTER JOIN includes all the information from the table on the right.

Example: OUTER JOIN

- In AdventureWorks database
  - SpecialOffer in the Sales schema - sale discounts lookup table.
  - SpecialOfferProduct - special offers are associated with which products



SpecialOffer	
SpecialOfferID	EndDate
Description	MinQty
DiscountPct	MaxQty
Type	Type
Category	rowguid
StartDate	ModifiedDate

SpecialOfferProduct
SpecialOfferID
ProductID
rowguid
ModifiedDate

- Using INNER JOIN (eliminating the rows with no discount, that’s SpecialOfferID 1):

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM   Sales.SpecialOffer sso
JOIN   Sales.SpecialOfferProduct ssop
  ON   sso.SpecialOfferID  = ssop.SpecialOfferID
WHERE  sso.SpecialOfferID != 1
```

- This query yields 243 rows, each with an associated ProductID:

```
SpecialOfferID Description                     DiscountPct    ProductID
-------------- ------------------------------- -------------- -----------
2              Volume Discount 11 to 14        0.02           707
2              Volume Discount 11 to 14        0.02           708
……
16             Mountain-500 Silver Clearance  0.40           987
16             Mountain-500 Silver Clearance  0.40           988
(243 row(s) affected)
```

- If we wanted to see what **all the special offers** were — not just which ones were actually in use. This query only gives you special offers that have products utilizing the offer.

- Using OUTER JOIN (eliminating the rows with no discount, that’s SpecialOfferID 1):

```
SELECT     sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM       Sales.SpecialOffer sso
LEFT JOIN  Sales.SpecialOfferProduct ssop
        ON sso.SpecialOfferID  = ssop.SpecialOfferID
WHERE      sso.SpecialOfferID != 1
```

- This query yields 244 rows, each with an associated ProductID:

```
SpecialOfferID Description                    DiscountPct    ProductID
-------------- ------------------------------ -------------- -----------
2              Volume Discount 11 to 14       0.02           707
2              Volume Discount 11 to 14       0.02           708
……
6              Volume Discount over 60        0.20           NULL
……
16             Mountain-500 Silver Clearance  0.40           987
16             Mountain-500 Silver Clearance  0.40           988
(244 row(s) affected)
```

- We will find that we have included every row from that table except for SpecialOfferID 1.

OUTER JOIN

- SQL Server will fill in a NULL for any value that comes from the opposite side (LEFT JOIN: right side) of the join if there is no match with the inclusive side (LEFT JOIN: left side) of the JOIN
- Example: RIGHT OUTER JOIN

```
SELECT     sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM       Sales.SpecialOfferProduct ssop
RIGHT JOIN Sales.SpecialOffer sso
        ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE      sso.SpecialOfferID != 1          (244 row(s) affected)
```

```
SELECT     sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM       Sales.SpecialOffer sso
RIGHT JOIN Sales.SpecialOfferProduct ssop
        ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE      sso.SpecialOfferID != 1          (243 row(s) affected)
```

Finding Non-Matching Records

- One very common use for the inclusive nature of OUTER JOINs is *finding unmatched records* in the exclusive table.
- An OUTER JOIN returns a NULL value in the column wherever there is no match. Then we can use SELECT list and add an extra condition to the WHERE clause to find the non-matching records.
- For example, find out the discount that no product use

```
SELECT          Description
FROM            Sales.SpecialOffer sso
LEFT OUTER JOIN Sales.SpecialOfferProduct ssop
             ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE           sso.SpecialOfferID  != 1
  AND           ssop.SpecialOfferID IS NULL
```

Finding Non-Matching Records

- One very common use for the inclusive nature of OUTER JOINs is *finding unmatched records* in the exclusive table.
- An OUTER JOIN returns a NULL value in the column

```
Description
------------------------------
Volume Discount over 60
(1 row(s) affected)
```

- For example, find out the discount that no product use

```sql
SELECT          Description
FROM            Sales.SpecialOffer sso
LEFT OUTER JOIN Sales.SpecialOfferProduct ssop
             ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE           sso.SpecialOfferID  != 1
  AND           ssop.SpecialOfferID IS NULL
```

Finding Non-Matching Records

- *What if the record really has a NULL value? (Left Outer Join)*
- For example, if we are joining based on the SpecialOfferID columns in both tables, only three conditions can exist:
  1. If the SpecialOfferProduct.SpecialOfferID column has a *non-NULL* value, then, according to the ON operator of the JOIN clause, if a special offer *record exists*, SpecialOffer.SpecialOfferID must also have the same value as SpecialOfferProduct.SpecialOfferID (look at the ON ssop.SpecialOfferID = sso.SpecialOfferID).

Finding Non-Matching Records

- *What if the record really has a NULL value? (Left Outer Join)*
- For example, if we are joining based on the SpecialOfferID columns in both tables, only three conditions can exist:

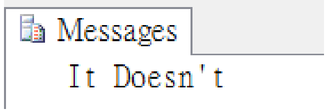
  2. If the SpecialOfferProduct.SpecialOfferID column has a *non-NULL* value, then, according to the ON operator of the JOIN clause, if a special offer *record does not exist*, SpecialOffer.SpecialOfferID will be *returned as NULL*.

Finding Non-Matching Records

- *What if the record really has a NULL value? (Left Outer Join)*
- For example, if we are joining based on the SpecialOfferID columns in both tables, only three conditions can exist:

   3. If the SpecialOfferProduct.SpecialOfferID happens to have a *NULL value*, and SpecialOffer.SpecialOfferID also has a *NULL value*, there will be no join (*NULL does not equal NULL*), and SpecialOffer.SpecialOfferID will *return NULL* because there is *no matching record*.

- NOTE: A value of NULL does not join to a value of NULL.



```
It Doesn't
```

```
IF (NULL = NULL) PRINT 'It Does'
ELSE             PRINT 'It Doesn''t'
```

```
IF (NULL != NULL) PRINT 'It Does'
ELSE              PRINT 'It Doesn''t'
```

Example: OUTER JOIN

- A small query (Finding Employees' Name):

```
SELECT      pp.BusinessEntityID, pp.FirstName, pp.LastName
FROM        Person.Person pp
INNER JOIN HumanResources.Employee hre
        ON hre.BusinessEntityID = pp.BusinessEntityID
                                            (290 row(s) affected)
```

- And from the query, there are 19972 records in Person

```
SELECT COUNT(*)
FROM   Person.Person                                      (19972)
```

- Question: Which persons are *not employees*?

```
SELECT      pp.BusinessEntityID, pp.FirstName, pp.LastName
FROM        Person.Person pp
LEFT JOIN   HumanResources.Employee hre
       ON   hre.BusinessEntityID = pp.BusinessEntityID
WHERE       hre.BusinessEntityID IS NULL;
                                          (19682 row(s) affected)
```

OUTER JOINs: Multi-Join Query

- It is when combining an OUTER JOIN with other JOINs that the concept of sides becomes even more critical.
- The important to understand here is that everything to the “left” - or before - the JOIN in question will be treated just as if it were a single table for the purposes of inclusion or exclusion from the query.
- The same is true for everything to the “right” - or after - the JOIN.

OUTER JOINs: Multi-Join Query

- Consider the following tables:

TABLE: Address

AddressID	Address
1	1234 Anywhere
3	567 Main St.
NULL	999 1st St.
NULL	1212 Smith Ave
NULL	364 Westin

TABLE: VendorAddress

VendorID	AddressID
1	1
2	3

TABLE: Vendors

VendorIName	VendorID
Don's Database Design Shop	1
Dave's Data	2
The SQL Sequel	3

OUTER JOINs: Multi-Join Query

- We want to find the **names** of every **vendor** as well as their **address**. But there are two issues here.
  - The query need to **return every vendor** no matter what - OUTER JOIN
  - A vendor can have more than one address and vice versa, which is an **associate table** - INNER JOIN
- In this example, we need an OUTER JOIN and an INNER JOIN

- Firstly, we try to find the names as well as their IDs

```
SELECT    v.VendorName, va.VendorID
FROM      Vendors v
LEFT JOIN VendorAddress va
       ON v.VendorID = va.VendorID
```

- The results resulted as we expected

```
VendorName                     VendorID
------------------------------ --------
Don's Database Design Shop     1
Dave's Data                    2
The SQL Sequel                 NULL
(3 row(s) affected)
```

- Then join the VendeorID and AddressID to obtain their Address

```
SELECT    v.VendorName, a.Address
FROM      Vendors v
LEFT JOIN VendorAddress va
       ON v.VendorID = va.VendorID
     JOIN Address a
       ON va.AddressID = a.AddressID
```

- Then join the VendeorID and AddressID to obtain their Address

```
SELECT    v.VendorName, a.Address
FROM      Vendors v
LEFT JOIN VendorAddress va
       ON v.VendorID = va.VendorID
     JOIN Address a
       ON va.AddressID = a.AddressID
```

```
VendorName                     Address
------------------------------ ---------------
Don's Database Design Shop     1234 Anywhere
Dave's Data                    567 Main St.
(2 row(s) affected)
```

- We've ***lost*** one of the vendors
  - An OUTER JOIN between Vendors and VendorAddress - it returns all vendors
  - But, an INNER JOIN is exclusive to both sides of the JOIN - only records where the result of the first JOIN has a match with the second JOIN will be included

- There are always multiple ways to solve a problem:
  - Use an OUTER JOIN
  - Change the order of the JOINs
  - Group the JOINs together

1. Use an OUTER JOIN:

```
SELECT     v.VendorName, a.Address
FROM       Vendors v
LEFT JOIN  VendorAddress va
       ON  v.VendorID = va.VendorID
LEFT JOIN  Address a
       ON  va.AddressID = a.AddressID
```

```
VendorName                   Address
---------------------------- -------------
Don's Database Design Shop   1234 Anywhere
Dave's Data                  567 Main St.
The SQL Sequel               NULL
(3 row(s) affected)
```

But, the logic is different from before:

- If there were rows in VendorAddress that didn’t have matching rows in Address, the earlier query used to exclude those (with its INNER JOIN syntax), and now they’re permitted.

2. Reorder of the JOINs:

```
SELECT     v.VendorName,
           a.Address
FROM       VendorAddress va
JOIN       Address a
  ON       va.AddressID = a.AddressID
RIGHT JOIN Vendors v
        ON v.VendorID = va.VendorID
```

```
VendorName                   Address
---------------------------- ------------
Don's Database Design Shop   1234 Anywhere
Dave's Data                  567 Main St.
The SQL Sequel               NULL
(3 row(s) affected)
```

This time without the subtle logic change

3. Group the JOINs:

```
SELECT    v.VendorName, a.Address
FROM      Vendors v
LEFT JOIN (
     VendorAddress va
     JOIN Address a
       ON va.AddressID = a.AddressID
)

       ON v.VendorID = va.VendorID
```

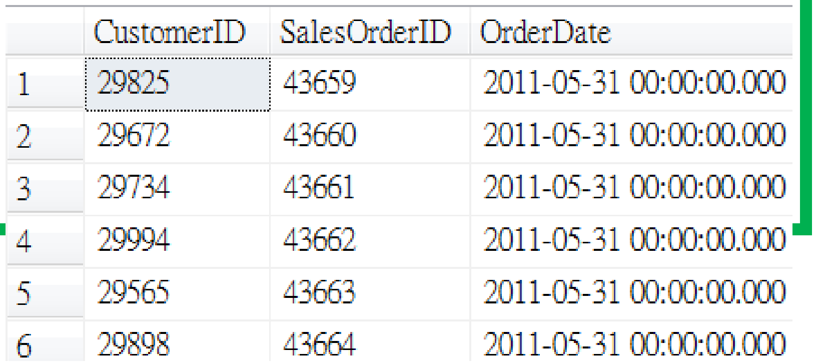
```
VendorName                   Address
---------------------------- ------------
Don's Database Design Shop   1234 Anywhere
Dave's Data                  567 Main St.
The SQL Sequel               NULL
(3 row(s) affected)
```

The parentheses are optional! The key to grouping joins is the order of the join conditions, not the parentheses.

Filtering Attributes from Nonpreserved Side

- Be careful the WHERE clause in OUTER JOIN
  - An expression in the form NULL <operator> <value> yields UNKNOWN
  - A WHERE clause filters UNKNOWN out.
  - Such a predicate in the WHERE clause causes all outer rows to be filtered out, effectively nullifying the outer join.

```
SELECT     C.CustomerID, O.SalesOrderID, O.OrderDate
FROM       Sales.Customer AS C
LEFT JOIN  Sales.SalesOrderHeader AS O
        ON C.CustomerID = O.CustomerID
WHERE      O.OrderDate >= '20000101'
```



	CustomerID	SalesOrderID	OrderDate
1	29825	43659	2011-05-31 00:00:00.000
2	29672	43660	2011-05-31 00:00:00.000
3	29734	43661	2011-05-31 00:00:00.000
4	29994	43662	2011-05-31 00:00:00.000
5	29565	43663	2011-05-31 00:00:00.000
6	29898	43664	2011-05-31 00:00:00.000

- O.OrderDate >= '20000101‘ will filtered out the UNKNOWN

Using the COUNT Aggregate

- Be careful the COUNT(*) in OUTER JOIN
  - The COUNT(*) aggregate takes into consideration both inner rows and outer rows
- For example, the following query is supposed to return the count of orders for each customer

```
SELECT     C.CustomerID,
           COUNT(*) AS [# of Orders]
FROM       Sales.Customer AS C
LEFT JOIN  Sales.SalesOrderHeader AS O
        ON C.CustomerID = O.CustomerID
GROUP BY   C.CustomerID
```

```
CustomerID  # of Order
----------- -----------
1           1
2           1
3           1
...
13147       1
13148       1
13149       2
13150       2
13151       1

(19820 row(s) affected)
```

- The COUNT(*) aggregate counts rows regardless of their meaning or contents

- For example, the following query is supposed to return the count of orders for each customer
  - The COUNT(*) aggregate function cannot detect whether a row really represents an order.
  - To fix the problem, we should use COUNT(<column>) instead of COUNT(*), and provide a column from the nonpreserved side of the join.

```
SELECT    C.CustomerID,
          COUNT(O.SalesOrderID)
                AS [# of Orders]
FROM      Sales.Customer AS C
LEFT JOIN Sales.SalesOrderHeader AS O
       ON C.CustomerID = O.CustomerID
GROUP BY  C.CustomerID
```

```
CustomerID  # of Order
----------- -----------
1           0
2           0
3           0
...
13147       1
13148       1
13149       2
13150       2
13151       1

(19820 row(s) affected)
```

FULL JOINS: Seeing both Sides

- A FULL JOIN (also known as a FULL OUTER JOIN) is a matching up of data on both sides of the JOIN with verything included, no matter what side of the JOIN it is on.
- Example:

```
SELECT    a.Address, va.AddressID
FROM      VendorAddress va
FULL JOIN Address a
       ON va.AddressID = a.AddressID
```

```
Address          AddressID
---------------- ---------
1234 Anywhere    1
567 Main St.     3
999 1st St.      NULL
1212 Smith Ave   NULL
364 Westin       NULL

(5 row(s) affected)
```

CROSS JOINS

- A CROSS JOIN will return all records where each row from the first table is combined with each row from the second table.
- A CROSS JOIN differs from other JOINs in that there is no ON operator.
- Example:

```
VendorName           Address
-------------------- ---------------
Don's Database ...   1234 Anywhere
Don's Database ...   567 Main St.
Don's Database ...   999 1st St.
Don's Database ...   1212 Smith Ave
Don's Database ...   364 Westin
Dave's Data          1234 Anywhere
Dave's Data          567 Main St.
Dave's Data          999 1st St.
Dave's Data          1212 Smith Ave
Dave's Data          364 Westin
The SQL Sequel       1234 Anywhere
The SQL Sequel       567 Main St.
The SQL Sequel       999 1st St.
The SQL Sequel       1212 Smith Ave
The SQL Sequel       364 Westin

(15 row(s) affected)
```

```sql
SELECT     v.VendorName, a.Address
FROM       Vendors v
CROSS JOIN Address a
```

Example: Producing Tables of Numbers

- One situation in which cross joins can be very handy is when they are used to produce a result set with a sequence of integers (1, 2, 3, and so on).
- Such a sequence of numbers is an extremely powerful tool that can be used for many purposes.
- By using cross joins, we can produce the sequence of integers in a very efficient manner.

- We can start by creating a table called Digits and populate the table with 10 rows with the digits 0 through 9.
- Suppose we need to write a query that produces a sequence of integers in the range 1 through 1,000. We can cross three instances of the Digits table, each representing a different power of 10 (1, 10, 100).

```sql
INSERT INTO Digits(digit)
VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9);

SELECT     D3.digit * 100 +
           D2.digit * 10  +
           D1.digit + 1 AS n
FROM       Digits AS D1
CROSS JOIN Digits AS D2
CROSS JOIN Digits AS D3
ORDER BY   n;
```

```
n
-----------
1
2
3
...
997
998
999
1000
```

Example: Calculating Running Totals

- In order to calculte the cumulative total we need to know the previous cumulative total.

```
Day    Sales   Total
-----  ------  -------
1      120     120
2      60      180
3      125     305
```

- One solution uses a CROSS JOIN and table aliases to join the table with itself.
- This causes each row in the left table (Sales a) to be joined with each row in the right table (Sales b) where the DayCount in b in less than the DayCount in a.
- The SUM(b.Sales) and the GROUP BY a.DayCount, a.Sales then allow the running total for each row to be calculated.

Example: Calculating Running Totals

```
CREATE TABLE #Sales ( Day INT, Sales INT );

INSERT INTO #Sales
VALUES (1, 120), (2, 60), (3, 125);

SELECT      s1.Day, s1.Sales, SUM(s2.Sales)
FROM        #Sales AS s1
CROSS JOIN  #Sales AS s2
WHERE       s2.Day <= s1.Day
GROUP BY    s1.Day, s1.Sales
ORDER BY    s1.Day, s1.Sales
```

```
Day    Sales   Total
-----  ------  ------
1      120     120
2      60      180
3      125     305
```

Summary: Table Join

Product	Category
A	1
B	2

Category	Name
1	Canned
2	Drink
3	Fresh

JOIN

Product	Name
A	Canned
A	Drink
A	Fresh
B	Canned
B	Drink
B	Fresh

CROSS JOIN: no row matching

Product	Name
A	Canned
B	Drink

INNER JOIN: row matching based on foreign key

Product	Category
A	1
B	2
C	

Category	Name
1	Canned
2	Drink
3	Fresh

JOIN

INNER JOIN

Product	Name
A	Canned
B	Drink

FULL JOIN

Product	Name
A	Canned
B	Drink
C	NULL
NULL	Fresh

LEFT JOIN

Product	Name
A	Canned
B	Drink
C	NULL

RIGHT JOIN

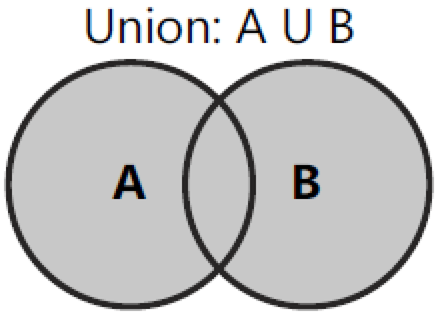
Product	Name
A	Canned
B	Drink
NULL	Fresh

Set Operators

Set Operations

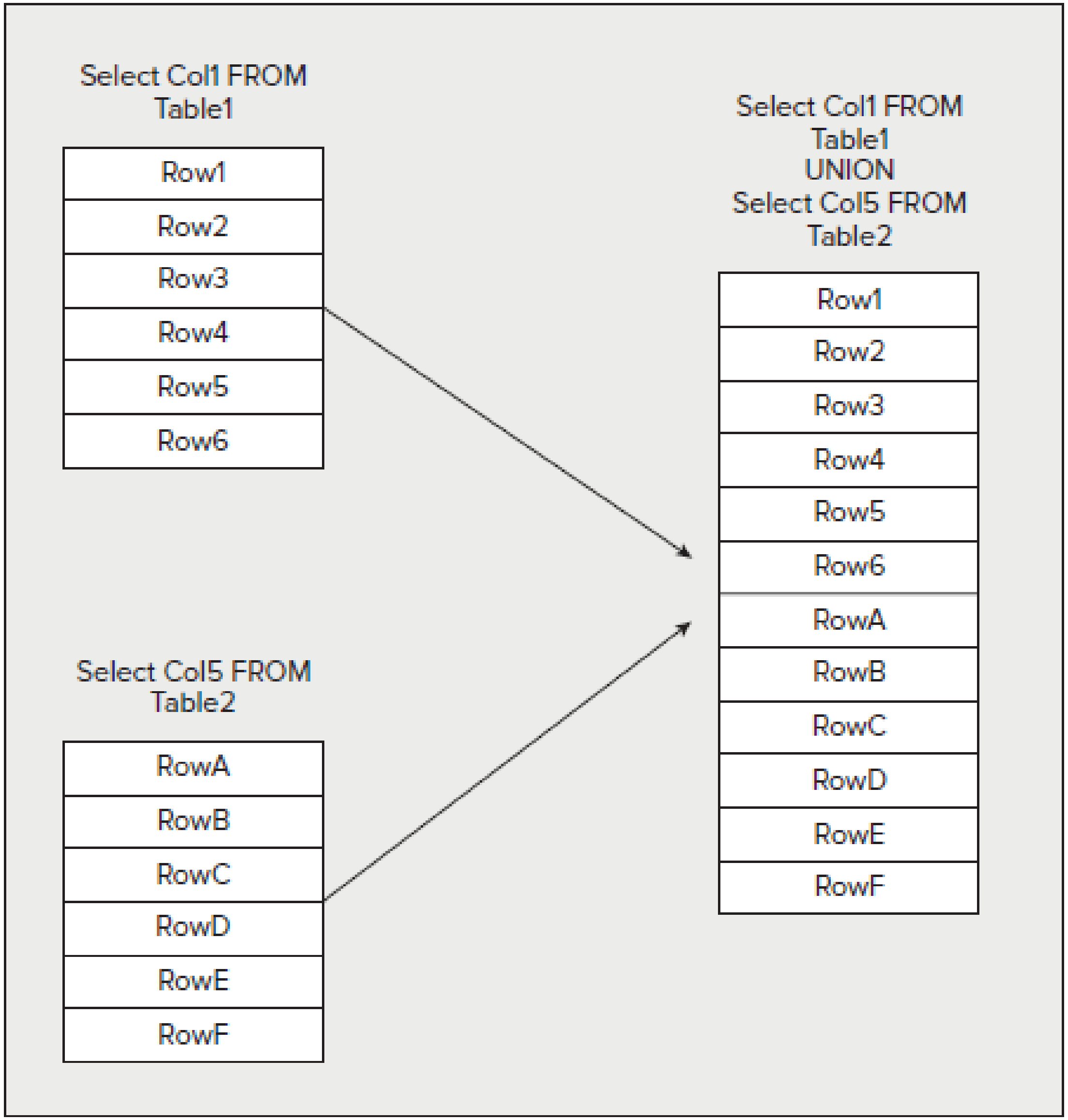
- Set operations combine results from two or more queries into a single result set
- Operands: UNION, EXCEPT, INETRSECT
- Common basic rules
  - The number and the order of the columns must be the same in all queries
  - The data types must be compatible
  - Ordering of the final result (ORDER BY) should be placed at the end of the whole statement
  - An interesting aspect of set operators is that when it is comparing rows, a set operator considers two NULLs as equal.

UNION



- UNION is an operator we can use to cause two or more queries to generate one result set.
- When dealing with queries that use a UNION, there are just a few key points:
  - All the UNIONed queries must have the same number of columns in the SELECT list.
  - The headings returned for the combined result set will be taken only from the first of the queries.
  - The data types of each column in a query must be implicitly compatible with the data type in the same relative column in the other queries.
  - Unlike non-UNION queries, the default return option for UNIONs is DISTINCT rather than ALL.

UNION



Select Col1 FROM Table1

Row1
Row2
Row3
Row4
Row5
Row6

Select Col5 FROM Table2

RowA
RowB
RowC
RowD
RowE
RowF

Select Col1 FROM Table1
UNION
Select Col5 FROM Table2

Row1
Row2
Row3
Row4
Row5
Row6
RowA
RowB
RowC
RowD
RowE
RowF

Example: UNION Distinct

```
SELECT FirstName + ' ' + LastName AS Name,
       pe.EmailAddress EmailAddress
FROM   Person.Person pp
JOIN   Person.EmailAddress pe
  ON   pp.BusinessEntityID = pe.BusinessEntityID
JOIN   Sales.Customer sc
  ON   pp.BusinessEntityID = sc.CustomerID

UNION

SELECT FirstName + ' ' + LastName AS VendorName,
       pe.EmailAddress VendorEmailAddress
FROM   Person.Person pp
JOIN   Person.EmailAddress pe
  ON   pp.BusinessEntityID = pe.BusinessEntityID
JOIN   Purchasing.Vendor pv
  ON   pp.BusinessEntityID = pv.BusinessEntityID
```

Example: UNION Distinct

```
SELECT FirstName + ' ' + LastName AS Name,
       pe.EmailAddress EmailAddress
FROM   Person.Person pp
JOIN   Person.EmailAddress pe
  ON   pp.BusinessEntityID = pe.BusinessEntityID
JOIN   Sales.Customer sc
  ON   pp.BusinessEntityID = sc.CustomerID
```

```
Name                 EmailAddress
-------------------- ----------------------------------------
A. Scott Wright      ascott0@adventure-works.com
Aaron Adams          aaron48@adventure-works.com
Aaron Allen          aaron55@adventure-works.com
…
…
Zachary Wilson       zachary36@adventure-works.com
Zainal Arifin        zainal0@adventure-works.com
Zheng Mu             zheng0@adventure-works.com
(10274 row(s) affected))
```

UNION ALL

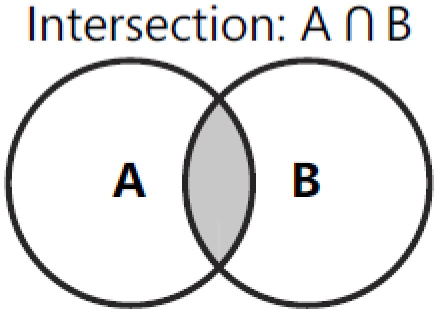
- A UNION deals with duplicate rows - throw out duplicates.
- We must explicitly to use UNION ALL to see duplicates.
- For example,

```
SELECT    a.Address, va.AddressID
FROM      VendorAddress va
FULL JOIN Address a
       ON va.AddressID = a.AddressID

UNION ALL

SELECT    a.Address, va.AddressID
FROM      VendorAddress va
FULL JOIN Address a
       ON va.AddressID = a.AddressID
```

```
Address          AddressID
---------------- ---------
1234 Anywhere    1
567 Main St.     3
999 1st St.      NULL
1212 Smith Ave   NULL
364 Westin       NULL
1234 Anywhere    1
567 Main St.     3
999 1st St.      NULL
1212 Smith Ave   NULL
364 Westin       NULL

(10 row(s) affected)
```

INTERSECT Distinct

- In T-SQL, the INTERSECT (implicit DISTINCT), operator returns the intersection of the result sets of two input queries, returning only rows that appear in both inputs.



- Example,

```
SELECT *
FROM (VALUES ('UK'   , NULL, 'London'  ),
             ('USA'  , 'WA', 'Kirkland'),
             ('NULL', NULL,  NULL      ),
             ('NULL', NULL,  NULL      )) AS T1(COUNTRY, REGION, CITY)

INTERSECT

SELECT *
FROM (VALUES ('UK'   , NULL, 'London'  ),
             ('USA'  , 'WA', 'Seattle' ),
             ('NULL', NULL,  NULL      ),
             ('NULL', NULL,  NULL      )) AS T2(COUNTRY, REGION, CITY)
```

```
COUNTRY REGION CITY
------- ------ --------
UK      NULL   London
NULL    NULL   NULL
```

INTERSECT ALL

- Standard SQL supports an ALL flavor of the INTERSECT operator, but this flavor has **NOT** yet been implemented as of SQL Server 2012.
- The keyword ALL in the INTERSECT ALL operator means that duplicate intersections will not be removed.
- INTERSECT ALL does not return all duplicates but only returns the number of duplicate rows, matching the lower of the counts in both multisets.
    - If there are x occurrences of a row R in the first input multiset and y occurrences of R in the second, R appears minimum(x, y) times in the result of the operator.

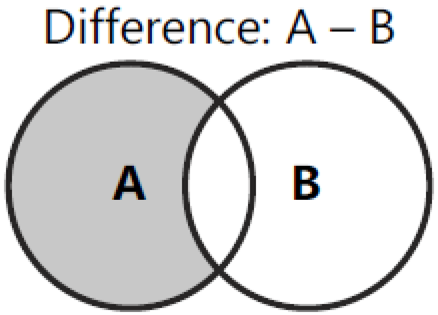
INTERSECT ALL

- We can use the ROW_NUMBER function to number the occurrences of each row in each input query to obtain the same result.
- INTERSECT ALL:

```
SELECT ROW_NUMBER()
       OVER(PARTITION BY COUNTRY, REGION, CITY
       ORDER BY (SELECT 0)) AS ROWNUM,
       COUNTRY, REGION, CITY
FROM   T1

INTERSECT

SELECT ROW_NUMBER()
       OVER(PARTITION BY COUNTRY, REGION, CITY
       ORDER BY (SELECT 0)) AS ROWNUM,
       COUNTRY, REGION, CITY
FROM   T2
```

```
ROWNUM     COUNTRY REGION CITY
---------- ------- ------ --------
1          NULL    NULL   NULL
2          NULL    NULL   NULL
1          UK      NULL   London
```

EXCEPT Distinct



- The EXCEPT operator logically first eliminates duplicate rows from the two input multisets – turning them to sets - and then returns only rows that appear in the first set but not the second. Example,

```
COUNTRY REGION CITY
------- ------ --------
USA     WA     Kirkland
```

```
SELECT *
FROM (VALUES ('UK'   , NULL, 'London'  ),
             ('USA'  , 'WA', 'Kirkland'),
             ('NULL', NULL,  NULL      ),
             ('NULL', NULL,  NULL      )) AS T1(COUNTRY, REGION, CITY)

EXCEPT

SELECT *
FROM (VALUES ('UK'   , NULL, 'London'  ),
             ('USA'  , 'WA', 'Seattle' ),
             ('NULL', NULL,  NULL      ),
             ('NULL', NULL,  NULL      )) AS T2(COUNTRY, REGION, CITY)
```

EXCEPT ALL

- The EXCEPT ALL operator has **NOT** yet been implemented as of SQL Server 2012.
- The EXCEPT ALL operator is very similar to the EXCEPT operator, but it also takes into account the number of occurrences of each row.
- EXCEPT ALL returns only occurrences of a row from the first multiset that do not have a corresponding occurrence in the second.
  - Provided that a row R appears x times in the first multiset and y times in the second, and x > y, R will appear x – y in the result of this operation.

EXCEPT ALL

- We can use the ROW_NUMBER function to number the occurrences of each row in each input query to obtain the same result.
- EXCEPT ALL:

```
SELECT ROW_NUMBER()
       OVER(PARTITION BY COUNTRY, REGION, CITY
       ORDER BY (SELECT 0)) AS ROWNUM,
       COUNTRY, REGION, CITY
FROM   T1

EXCEPT

SELECT ROW_NUMBER()
       OVER(PARTITION BY COUNTRY, REGION, CITY
       ORDER BY (SELECT 0)) AS ROWNUM,
       COUNTRY, REGION, CITY
FROM   T2
```

```
ROWNUM     COUNTRY REGION CITY
---------- ------- ------ --------
1          USA     WA     Kirkland
```

Summary

- Use an INNER JOIN when we want to exclude non-matching fields.
- Use an OUTER JOIN when we want to retrieve matches wherever possible, but also want a fully inclusive dataset on one side of the JOIN.
- Use a FULL JOIN when we want to retrieve matches wherever possible, but also want a fully inclusive dataset on both sides of the JOIN.
- Use a CROSS JOIN when we want a Cartesian product based on the records in two tables.
- Use a UNION when we want the combination of the result of a second query appended to the first query.

Exercises

1. Write a query against the AdventureWorks database that returns one column called Name and contains the last name of the employee with NationalIDNumber 112457891.
   - Tables: HumanResources.Employee, Person.Person
2. Write a query that return the number of customers who placed no orders.
   - Tables: Sales.Customer, Sales.SalesOrderHeader
   - Desired output: 701

Exercises

3. Write a query that generates five copies of each employee row

   - Tables: HumanResources.Employee, Digits

```
BusinessEntityID FirstName          LastName                 N
---------------- ------------------ ------------------------ --------
1                Ken                Sánchez                  0
1                Ken                Sánchez                  1
1                Ken                Sánchez                  2
1                Ken                Sánchez                  3
1                Ken                Sánchez                  4
2                Terri              Duffy                    0
2                Terri              Duffy                    1
2                Terri              Duffy                    2
2                Terri              Duffy                    3
...
```

Exercises

4. Write a query that returns all products (ID and Name) and including both all products that have no special offers, and all products that have the No Discount offer.
   - Tables: Production.Product, Sales.SpecialOfferProduct, Sales.SpecialOffer

```
ProductID    Name                        SpecialOfferID Description
680          HL Road Frame - Black, 58   1              No Discount
706          HL Road Frame - Red, 58     1              No Discount
707          Sport-100 Helmet, Red       1              No Discount
708          Sport-100 Helmet, Black     1              No Discount
988          Mountain-500 Silver, 52     16             Mountain...
...
355          Guide Pulley                NULL           NULL
378          Hex Nut 17                  NULL           NULL
...
```