Microsoft® Technology Courses

Introduction

Microsoft .NET Framework
Microsoft Visual Studio
C# Programming

Microsoft

Understanding Joins

- One of SQL's most powerful features is the capability to join tables on the fly within data-retrieval queries.
- Joins are one of the most important operations you can perform using SQL SELECT, and a good understanding of joins and join syntax is an extremely important part of learning SQL.
- Before you can effectively use joins, you must understand relational tables and the basics of relational database design.
- What follows is by no means a complete coverage of the subject, but it should be enough to get you up and running.

160

Understanding Relational Tables

- Before you can effectively use joins, you must understand relational tables and the basics of relational database design.
- Suppose you had a database table containing a product catalog, with each catalog item in its own row.
- The kind of information you would store with each item would include a product description and price, along with vendor information about the company that creates the product.
- Now suppose you had multiple catalog items created by the same vendor. Where would you store the vendor information (things such as vendor name, address, and contact information)?

161

Why Relational Tables?

- You wouldn't want to store that data along with the products for several reasons:
  - Because the vendor information is the same for each product that vendor produces, repeating the information for each product is a waste of time and storage space.
  - If vendor information changes (for example, if the vendor moves or the area code changes), you would need to update every occurrence of the vendor information.
  - When data is repeated (that is, the vendor information is used with each product), there is a high likelihood that the data will not be entered exactly the same way each time. Inconsistent data is extremely difficult to use in reporting.

162

Normalization in Act

- The key here is that having multiple occurrences of the same data is never a good thing, and that principle is the basis for relational database design.
- Relational tables are designed so information is split into multiple tables, one for each datatype. The tables are related to each other through common values (and thus the relational in relational design).
- In our example, you can create two tables, one for vendor information and one for product information. The vendors table contains all the vendor information, one table row per vendor, along with a unique identifier for each vendor. This value, called a primary key, can be a vendor ID or any other unique value.
- The products table stores only product information, with no vendor-specific information other than the vendor ID (the vendors table's primary key). This key, called a foreign key, relates the vendors table to the products table, and using this vendor ID enables you to use the vendors table to find the details about the appropriate vendor.

163

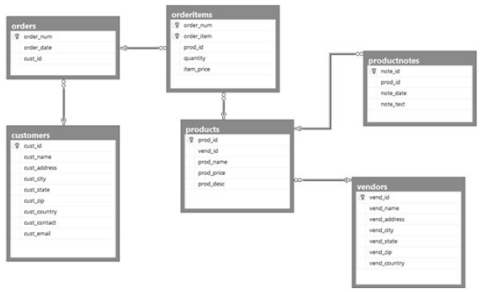
What we Gain?

- What does this do for you? Well, consider the following:
  - Vendor information is never repeated, so time and space are not wasted.
  - If vendor information changes, you can update a single record in the vendors table. Data in related tables does not change.
  - Because no data is repeated, the data used is obviously consistent, making data reporting and manipulation much simpler.
- The bottom line is that relational data can be stored efficiently and manipulated easily. Because of this, relational databases scale far better than nonrelational databases.

164

Presented by
Ranjan Bhatnagar

DistributorDB Database Relational Diagram



165

Why Use Joins?

- Breaking data into multiple tables enables more efficient storage, easier manipulation, and greater scalability.
- But these benefits come with a price.
- If data is stored in multiple tables, how can you retrieve that data with a single SELECT statement?
- The answer is to use a join. Simply put, a join is a mechanism used to associate tables within a SELECT statement (and thus the name join).
- Using a special syntax, you can join multiple tables so a single set of output is returned, and the join associates the correct rows in each table on the fly.

166

Types of JOINs

- Join conditions can be specified in either the FROM or WHERE clauses; specifying them in the FROM clause is recommended.
- WHERE and HAVING clauses can also contain search conditions to further filter the rows selected by the join conditions.
- Joins can be categorized as:
  - Inner Join
  - Outer Join
    - Left Join or Left Outer Join
    - Right Join or Right Outer Join
    - Full Outer Join
  - Cross Join

167

Defining Joins

- **Inner joins** (the typical join operation, which uses some comparison operator like = or <>). These include equi-joins and natural joins.
- Inner joins use a comparison operator to match rows from two tables based on the values in common columns from each table.
- For example, retrieving all rows where the student identification number is the same in both the students and courses tables.
- **Outer joins.** Outer joins can be a left, a right, or full outer join. Outer joins are specified with one of the following sets of keywords when they are specified in the FROM clause:

168

Definition Cont…

- **LEFT JOIN or LEFT OUTER JOIN** The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match. When a row in the left table has no matching rows in the right table, the associated result set row contains null values for all select list columns coming from the right table.
- **RIGHT JOIN or RIGHT OUTER JOIN** A right outer join is the reverse of a left outer join. All rows from the right table are returned. Null values are returned for the left table any time a right table row has no matching row in the left table.

169

Definition Cont…

- **FULL JOIN or FULL OUTER JOIN** A full outer join returns all rows in both the left and right tables. Any time a row has no match in the other table, the select list columns from the other table contain null values. When there is a match between the tables, the entire result set row contains data values from the base tables.
- **Cross joins** Cross joins return all rows from the left table. Each row from the left table is combined with all rows from the right table. Cross joins are also called Cartesian products.

170

Presented by
Ranjan Bhatnagar

Creating a Join

- Creating a join is very simple. You must specify all the tables to be included and how they are related to each other. Consider example:

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products
WHERE vendors.vend_id = products.vend_id
ORDER BY vend_name, prod_name;
```

- The **SELECT** statement starts by specifying the columns to be retrieved.
- The **FROM** clause has two tables listed in the FROM clause: vendors and products.
- The tables are correctly joined with a **WHERE** clause that instructs SQL Server to match vend_id in the vendors table with vend_id in the products table.

171

Fully Qualifying Column Names

- You must use the fully qualified column name (table and column separated by a period) whenever there is possible ambiguity about to which column you are referring.
- SQL Server returns an error message if you refer to an ambiguous column name without fully qualifying it with a table name.
- You'll notice that the columns are specified as vendors.vend_id and products.vend_id. This fully qualified column name is required here because if you just specify vend_id, SQL Server cannot tell which vend_id columns you are referring to (because there are two of them, one in each table).

172

Inner Join

- The join you have been using so far is called an equijoin, a join based on the testing of equality between two tables.
- This kind of join is also called an inner join. In fact, you may use a slightly different syntax for these joins, specifying the type of join explicitly.
- The following SELECT statement returns the exact same data as the preceding example:

```
SELECT vend_name, prod_name, prod_price
FROM vendors INNER JOIN products
ON vendors.vend_id = products.vend_id
ORDER BY vend_name, prod_name;
```

173

Analysis of Inner Join

- The SELECT in this statement is the same as the preceding SELECT statement, but the FROM clause is different.
- Here, the relationship between the two tables is part of the FROM clause specified as INNER JOIN.
- When using this syntax, the join condition is specified using the special ON clause instead of a WHERE clause.
- The actual condition passed to ON is the same as would be passed to WHERE.

174

Which Syntax To Use?

- Per the ANSI SQL specification, use of the INNER JOIN syntax is preferable.
- Although using the WHERE clause to define joins is indeed simpler, using explicit join syntax ensures that you will never forget the join condition, and in some cases it can impact performance, too.

175

Joining Multiple Tables

- SQL imposes no limit to the number of tables that may be joined in a SELECT statement.
- The basic rules for creating the join remain the same.
- First list all the tables and then define the relationship between each. Here is an example:

```
SELECT prod_name, vend_name, prod_price, quantity
FROM orderitems, products, vendors WHERE
products.vend_id = vendors.vend_id AND
orderitems.prod_id = products.prod_id AND
order_num = 20005;
```

176

Presented by
Ranjan Bhatnagar

Using Table Aliases

- Earlier we learned how to use aliases to refer to retrieved table columns.
- The syntax to alias a column looks like this:

```
SELECT RTrim(vend_name) + ' (' + RTrim(vend_country) + ')'
AS vend_title FROM vendors ORDER BY vend_name;
```

- In addition to using aliases for column names and calculated fields, SQL also enables you to alias table names.
- There are two primary reasons to do this:
  - To shorten the SQL syntax
  - To enable multiple uses of the same table within a single SELECT statement

177

Example using Aliases

```
SELECT cust_name, cust_contact FROM customers AS c, orders
AS o, orderitems AS oi WHERE c.cust_id = o.cust_id AND
oi.order_num = o.order_num AND prod_id = 'TNT2';
```

- You'll notice that the three tables in the FROM clauses all have aliases. customers AS c establishes c as an alias for customers, and so on.
- This enables you to use the abbreviated c instead of the full text customers.
- In this example, the table aliases were used only in the WHERE clause, but aliases are not limited to just WHERE. You can use aliases in the SELECT list,the ORDER BY clause, and in any other part of the statement as well.

178

Natural Joins

- Whenever tables are joined, at least one column appears in more than one table (the columns being joined).
- Standard joins (the inner joins you learned about) return all data, even multiple occurrences of the same column.
- A natural join simply eliminates those multiple occurrences so only one of each column is returned.
- A natural join is a join in which you select only columns that are unique. This is typically done using a wildcard (SELECT *) for one table and explicit subsets of the columns for all other tables.

```
SELECT c.*, o.order_num, o.order_date, oi.prod_id, oi.quantity,
OI.item_price FROM customers AS c, orders AS o, orderitems AS
oi WHERE c.cust_id = o.cust_id AND oi.order_num = o.order_num
AND prod_id = 'FB';
```

179

Outer Joins

- Most joins relate rows in one table with rows in another. But occasionally, you want to include rows that have no related rows.
- For example, you might use joins to accomplish the following tasks:
  - Count how many orders each customer placed, including customers who have yet to place an order.
  - List all products with order quantities, including products not ordered by anyone.
  - Calculate average sale sizes, taking into account customers who have not yet placed an order.
- In each of these examples, the join includes table rows that have no associated rows in the related table. This type of join is called an outer join.

180

Example Outer Join

- The following SELECT statement is a simple outer join.
- To retrieve a list of all customers, including those who have placed no orders, you can do the following:

```
SELECT customers.cust_id, orders.order_num FROM
customers LEFT OUTER JOIN orders
ON customers.cust_id = orders.cust_id;
```

- Like the inner join shown in the previous examples, this SELECT statement uses the keywords OUTER JOIN to specify the join type (instead of specifying it in the WHERE clause).

181

Outer Join with LEFT or RIGHT

- When using OUTER JOIN syntax, you must use the RIGHT or LEFT keyword to specify the table from which to include all rows (RIGHT for the one on the right of OUTER JOIN and LEFT for the one on the left).
- The previous example uses LEFT OUTER JOIN to select all the rows from the table on the left in the FROM clause (the customers table).
- To select all the rows from the table on the right, you use a RIGHT OUTER JOIN, as shown in this example:

```
SELECT customers.cust_id, orders.order_num
FROM customers RIGHT OUTER JOIN orders
ON orders.cust_id = customers.cust_id;
```

182

Presented by
Ranjan Bhatnagar

Outer Join Types

- There are two basic forms of outer joins:
  - The left outer join
  - The right outer join.
- The only difference between them is the order of the tables they are relating.
- In other words, a left outer join can be turned into a right outer join simply by reversing the order of the tables in the FROM or WHERE clause.
- As such, the two types of outer joins can be used interchangeably, and the decision about which one is used is based purely on convenience.

183

Full Outer Join

- There is one other form of outer join worth noting, although you will likely rarely find a use for it.
- The FULL OUTER JOIN is used to retrieve the related rows from both tables, as well as the unrelated rows from each.
- These will have NULL values for the unrelated columns in the other table.
- The syntax for a FULL OUTER JOIN is the same as the previously shown outer joins, obviously substituting RIGHT and LEFT for FULL.

```
SELECT customers.cust_id, orders.order_num
FROM customers full OUTER JOIN orders
ON orders.cust_id = customers.cust_id;
```

184

Using Self Join / Recursive Relationship

- A table can be joined to itself in a self-join.
- Use a self-join when you want to create a result set that joins records in a table with other records in the same table.
- To list a table two times in the same query, you must provide a table alias for at least one of instance of the table name.
- This table alias helps the query processor determine whether columns should present data from the right or left version of the table.

185

Understanding Self Join

- As mentioned earlier, one of the primary reasons to use table aliases is to be able to refer to the same table more than once in a single SELECT statement.
- Suppose that a problem is found with a product (item ID DTNTR), and you want to know all the products made by the same vendor so as to determine whether the problem applies to them too.
- This query requires that you first find out which vendor creates item DTNTR and then find which other products are made by that vendor. The following is one way to approach this problem:

```
SELECT prod_id, prod_name FROM products WHERE vend_id =
(SELECT vend_id FROM products WHERE prod_id = 'DTNTR');
```

186

Implementing Self Join

- This previous solution uses subqueries.
- The inner SELECT statement does a simple retrieval to return the vend_id of the vendor that makes item DTNTR.
- That ID is the one used in the WHERE clause of the outer query, so all items produced by that vendor are retrieved.
- Now look at the same query using a join:

```
SELECT p1.prod_id, p1.prod_name
FROM products AS p1, products AS p2
WHERE p1.vend_id = p2.vend_id
AND p2.prod_id = 'DTNTR';
```

187

Self Joins Instead of Subqueries

- Self joins are often used to replace statements using subqueries that retrieve data from the same table as the outer statement.
- Although the end result is the same, sometimes these joins execute far more quickly than do subqueries.
- It is usually worth experimenting with both to determine which performs better.

188

The Importance of the WHERE Clause

- Why should we use a WHERE clause to set the join relationship?
- Actually, there is a very good reason for this. Remember, when tables are joined in a SELECT statement, that relationship is constructed on the fly.
- Nothing in the database table definitions can instruct SQL Server how to join the tables. You have to do that yourself.
- When you join two tables, what you are actually doing is pairing every row in the first table with every row in the second table.

189

Cartesian Product

- The WHERE clause acts as a filter to only include rows that match the specified filter condition, the join condition, in this case. Without the WHERE clause, every row in the first table is paired with every row in the second table, regardless of whether they logically go together.
- The results returned by a table relationship without a join condition.
- The number of rows retrieved is the number of rows in the first table multiplied by the number of rows in the second table.

190

Cross Join

- To understand CROSS JOIN, look at the following SELECT statement:

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products ORDER BY vend_name, prod_name;
```

- As you can see in the output produced, the Cartesian product is seldom what you want.
- The data returned here has matched every product with every vendor, including products with the incorrect vendor (and even vendors with no products at all).
- The type of join that returns a Cartesian product referred to as a cross join.

191

Using Joins with Aggregate Functions

- Aggregate functions are used to summarize data. Although all the examples of aggregate functions thus far only summarized data from a single table, these functions can also be used with joins.
- You want to retrieve a list of all customers and the number of orders that each has placed.
- The following code uses the Count() function to achieve this:

```
SELECT customers.cust_name, customers.cust_id,
Count(orders.order_num) AS num_ord FROM customers INNER
JOIN orders ON customers.cust_id = orders.cust_id GROUP BY
customers.cust_name, customers.cust_id;
```

192

Working with Subqueries

- The Query term is used for any SQL statement. However, it is generally used to refer to SELECT statements.
- SQL also enables you to create subqueries: queries that are embedded into other queries. Why would you want to do this?
- Suppose you wanted a list of all the customers who ordered item TNT2. What would you have to do to retrieve this information? Here are the steps to accomplish this:
  - Retrieve the order numbers of all orders containing item TNT2.
  - Retrieve the customer ID of all the customers who have orders listed in the order numbers returned in the previous step.
  - Retrieve the customer information for all the customer IDs returned in the previous step.

193

3 Steps

```
SELECT order_num FROM orderitems
WHERE prod_id = 'TNT2';
```

- It retrieves the order_num column for all order items with a prod_id of TNT2. i.e. 20005 and 20007. So, in next step use these values as:

```
SELECT cust_id FROM orders
WHERE order_num IN (20005,20007);
```

- The above statement retrieves the customer IDs associated with orders 20005 and 20007 as Cust_Id 10001 and 10004.
- Now use this result in next query to get final required output as:

```
SELECT cust_name, cust_contact FROM customers
WHERE cust_id IN (10001,10004);
```

194

Write Subquery

- First combine two queries by turning the first (the one that returned the order numbers) into a subquery as:

  SELECT cust_id FROM orders WHERE order_num IN (SELECT order_num FROM orderitems WHERE prod_id = 'TNT2');
- Subqueries are always processed starting with the innermost SELECT statement.
- When the above SELECT statement is processed, SQL Server actually performs two operations.
- First, it runs the subquery, that query returns the two order numbers, 20005 and 20007.

195

Write Subquery Cont…

- Second, those two values are then passed to the WHERE clause of the outer query in the comma-delimited format required by the IN operator.
- Now to retrieve the customer information for each of those customer IDs, instead of hard-coding those customer IDs, you can turn this WHERE clause into yet another subquery:

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN (SELECT order_num
                                      FROM orderitems
                                      WHERE prod_id = 'TNT2'));
```

196

Types of Subqueries

- Single-Row subquery (subquery returns only one row)
  - The operators that can be used with single-row subqueires are =, >, >=, <, <=, and <>.
- Multiple-Row subquery (subquery returns multiple rows)
  - Operators that can be used with multiple-row subqueries are:
    - IN (equal to any member in the list)
    - ANY (compare values to each value returned by the subquery)

197

Single Row Subquery

- The following statement retrieves the details of the products holding the lowest price.

  SELECT * FROM products WHERE prod_price = (SELECT MIN(prod_price) FROM products)
- Having-clause can also be used with this type of subquery. For example, the following statement returns all vend_id and prices where the minimum price is more than the minimum price of the products for vend_id 1003.

  SELECT vend_id, MIN(prod_price) FROM products GROUP BY vend_id HAVING IN(prod_price) > (SELECT MIN(prod_price) FROM products WHERE vend_id= 1003)

198

Subqueries as Calculated Fields

- Another way to use subqueries is in creating calculated fields.
- Suppose you want to display the total number of orders placed by every customer in your customers table.
- Orders are stored in the orders table along with the appropriate customer ID.
- To perform this operation, follow these steps:
  - Retrieve the list of customers from the customers table.
  - For each customer retrieved, count the number of associated orders in the orders table.

199

Correlated Subquery

- To perform that Count(*) calculation for each customer, use Count(*) as a subquery. Look at the following code:

  SELECT cust_name, cust_state, (SELECT Count(*) FROM orders WHERE orders.cust_id = customers.cust_id) AS orders FROM customers ORDER BY cust_name;
- This SELECT statement returns three columns for every customer in the customers table: cust_name, cust_state, and orders. orders is a calculated field that is set by a subquery provided in parentheses.
- That subquery is executed once for every customer retrieved. In this example, the subquery is executed five times because five customers were retrieved.

200

Presented by Ranjan Bhatnagar

Correlated Subquery Cont…

- The WHERE clause in this subquery is a little different from the WHERE clauses used previously because it uses fully qualified column names. The following clause tells SQL to compare the cust_id in the orders table to the one currently being retrieved from the customers table:

  WHERE orders.cust_id = customers.cust_id
- This type of subquery is called a correlated subquery. This syntax, the table name and the column name separated by a period, must be used whenever there is possible ambiguity about column names.

201

Checking for Existence

- Another use for subqueries is in conjunction with the EXISTS predicate.
- EXISTS, when used in a WHERE clause, looks at the results returned by a subquery, not at specific columns of data, but rather at the number of rows.
- If the subquery returns rows, the EXISTS test is true and the WHERE clause matches.
- However, if no rows are returned, the EXISTS test is false, and the WHERE clause does not match.

202

Multiple Row Subquery

```
SELECT cust_id, cust_name FROM customers WHERE cust_id
IN (SELECT cust_id FROM orders WHERE DateDiff(month,
order_date, '2005-09-01') = 0 AND customers.cust_id =
orders.cust_id);
```

- This SELECT statement retrieves the customer name and ID for any customers who made orders in the month of September 2005.
- The WHERE clause uses IN and a subquery to first select the IDs of customers who made orders the specified month, and then uses the results of that subquery to select just the desired customers from the customers table.

203

Same Output

- Now let's look at another SELECT statement that returns the exact same output:

```
SELECT cust_id, cust_name FROM customers WHERE EXISTS
(SELECT * FROM orders WHERE DateDiff(month, order_date,
'2005-09-01') = 0 AND customers.cust_id = orders.cust_id);
```

- This WHERE clause uses EXISTS instead of IN. The subquery is much the same as the one used with IN, except that this one also matches the cust_id columns in both the customers table and the orders table so that just the customers with orders in September 2005 are retrieved.

204

Set Operators

- Most SQL queries contain a single SELECT statement that returns data from one or more tables.
- SQL Server provides the following set operators.
- Set operators combine results from two or more queries into a single result set.
- These combined queries are usually known as unions or compound queries.
- There are basically two scenarios in which you'd use combined queries:
  - To return similarly structured data from different tables in a single query
  - To perform multiple queries against a single table, returning the data as one query

205

Multiple WHERE Conditions

- For the most part, combining two queries to the same table accomplishes the same thing as a single query with multiple WHERE clause conditions.
- In other words, any SELECT statement with multiple WHERE clauses can also be specified as a combined query, as you'll see in the section that follows.
- However, the performance of each of the two techniques can vary based on the queries used. As such, it is always good to experiment to determine which is preferable for specific queries.

206

Using UNION

- While using UNION, specify each SELECT statement and place the keyword UNION between each.

```
SELECT vend_id, prod_id, prod_price FROM products
WHERE prod_price <= 5
UNION
SELECT vend_id, prod_id, prod_price FROM products
WHERE vend_id IN (1001,1002);
```

- The above statements is made up of two of the SELECT statements separated by the UNION keyword. UNION instructs SQL Server to execute both SELECT statements and combine the output into a single query result set.

207

UNION Rules

- As you can see, unions are very easy to use. But a few rules govern exactly which can be combined:
  - A UNION must be composed of two or more SELECT statements, each separated by the keyword UNION. Therefore, if you were combining four SELECT statements, you would use three UNION keywords.
  - Each query in a UNION must contain the same columns, expressions, or aggregate functions, and they must be listed in the same order.
  - Column datatypes must be compatible: They need not be the exact same type, but they must be of a type that SQL Server can implicitly convert.

208

Eliminating Duplicate Rows

- The UNION automatically removes any duplicate rows from the query result set.
- If you want you can change this default behavior of UNION.
- If you want all occurrences of all matches returned, you can use UNION ALL instead of UNION.

```
SELECT vend_id, prod_id, prod_price FROM products
WHERE prod_price <= 5
UNION ALL
SELECT vend_id, prod_id, prod_price FROM products
WHERE vend_id IN (1001,1002);
```

209

EXCEPT and INTERSECT

- Except and Intersect returns distinct rows by comparing the results of two queries.
  - EXCEPT returns distinct rows from the left input query that aren't output by the right input query.
  - INTERSECT returns distinct rows that are output by both the left and right input queries operator.
- The basic rules for combining the result sets of two queries that use EXCEPT or INTERSECT are the following:
  - The number and the order of the columns must be the same in all queries.
  - The data types must be compatible.

210

Example

- The following query returns any distinct values from the query to the left of the EXCEPT operator that are not also found on the right query.

```
SELECT vend_id, prod_id, prod_price FROM products
WHERE prod_price <= 5
except
SELECT vend_id, prod_id, prod_price FROM products
WHERE vend_id IN (1001,1002);
```

- The following query returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operator.

```
SELECT vend_id, prod_id, prod_price FROM products
WHERE prod_price <= 5
intersect
SELECT vend_id, prod_id, prod_price FROM products
WHERE vend_id IN (1001,1002);
```

211

Presented by
Ranjan Bhatnagar