SQL and XQuery tutorial for IBM DB2, Part 3: SQL joins and unions

Complex queries that involve more than one relational table

Skill Level: Introductory

Pat Moffatt (pmoffatt@ca.ibm.com)
Information Management Program Manager, IBM Academic Initiative
IBM

Bruce Creighton (bcreight@ca.ibm.com)
Skills Segment Planner
IBM

Jessica Cao
Training Tools Developer
IBM

11 Aug 2006

This tutorial describes two ways to combine data from different tables in an IBM® DB2® database: by joining the tables with inner or outer joins and by using the UNION, EXCEPT, and INTERSECT operators, which take the intermediate result set from one query expression and combine it with the result set from another query expression. This tutorial is Part 3 of the SQL & XQuery tutorial for IBM DB2 series.

Section 1. Before you start

About this series

This tutorial series teaches basic to advanced SQL and basic XQuery topics and shows how to express commonly asked business questions as database queries by
using SQL queries or XQueries. Developers and database administrators can use this tutorial to enhance their database query skills. Academic Initiative members can use this tutorial series as a part of their database curriculum.

All the examples in this document are based on Aroma, a sample database that contains sales data for coffee and tea products sold in stores across the United States. Each example consists of three parts:

- A business question, expressed in everyday language
- One or more example queries, expressed in SQL or XQuery
- A table of results returned from the database

This guide is designed to allow participants to learn the SQL language and XQuery. As with any learning, it is important to supplement it with hands-on exercises. This is facilitated by the table definitions and data.

For students using this as part of an academic class, obtain from your instructor the instructions to connect to the Aroma database and learn about any differences between the guide and your local set up.

This tutorial was written for DB2 Express-C 9 for UNIX®, Linux® and Windows® (formerly known as Viper).

### About this tutorial

This tutorial describes two ways to combine data from different tables in an IBM DB2 database:

- By joining the tables
- By using the UNION, EXCEPT, and INTERSECT operators

The first part of this tutorial presents examples of inner and outer joins.

The second part illustrates how to combine data from different tables by using UNION, EXCEPT, and INTERSECT operators, which take the intermediate result set from one query expression and combine it with the result set from another query expression.

### Connecting to a database

You need to connect to a database before you can use SQL statements to query or manipulate data. The CONNECT statement associates a database connection with
Find out from your instructor the database name that you will need to be connected to. For this series, the database name is aromadb.

To connect to the aromadb database, type the following command in the DB2 command line processor:

```
CONNECT TO aromadb USER userid USING password
```

Replace the user ID and password with the user ID and password that you received from your instructor. If no user ID and password are required, simply use the following command:

```
CONNECT TO aromadb
```

The following message tells you that you have made a successful connection:

```
Database Connection Information
Database server = DB2/NT 9.0.0
SQL authorization ID = USERID
Local database alias = AROMADB
```

Once you are connected, you can start using the database.

---

**Section 2. Table names and schemas**

**What is a schema?**

A schema is used for grouping. In the simplest term, a schema is like a section in a public library: books are grouped into different sections. For history books, you go to the history section. Similarly, in a database (analogous to the library), tables are grouped into different schemas. When a user logs in to the database, that user identity becomes the default schema for all queries.

For example, if a user named DBUSER executes a query such as

```
SELECT ... FROM sales
```
the system interprets that as meaning

```
SELECT ... FROM dbuser.sales
```

If there is no schema `dbuser` in the database the system returns an error.

There are two methods to avoid this error. One is to use qualified table names for each table; the second is to set a different default schema identifier.

**Use qualified table names**

You can explicitly state the fully qualified table name in the FROM clause of each query. To do so, you prefix each table name with the schema name and a dot, as in

```
SELECT ... FROM aroma.sales ...
```

There are two clear advantages to using qualified table names:

1. Any user can examine the query and know precisely which schema the query was written against.
2. You can include tables from different schemas within the same query provided that the user has permission to access data from each schema, as in:

```
SELECT ... FROM aroma.sales, dbuser.customer ...
```

If you are going to be performing many queries from a single schema; however, you may not want to have to type all of the qualified table names. You can save keystrokes by setting a new default schema.

**Set a new default schema**

Setting a new schema requires just the simple command:

```
SET SCHEMA <schema_name>;
```

From this point forward, the system uses the new schema name as the default schema, so it is necessary only to use the unqualified table names. You can still include tables from other schemas by using qualified table names, as in:
A user must have the appropriate permissions to work in the new schema, however, or the queries and other actions will fail.

The setting remains in effect only for your current session. If you disconnect from the database, you must run the SET command again for your next series of queries.

Usage note

Queries in this series will use qualified table names.

Table aliases

You can also save keystrokes and make queries more readable by taking advantage of table aliases. Table aliases are generally a shortened name for a table, assigned in the FROM clause and used throughout the remainder of the query. These are most frequently used when a query joins multiple tables.

For example, a simple joined query might be:

```sql
SELECT date, dollars
FROM aroma.period, aroma.sales
WHERE aroma.period.perkey = aroma.sales.perkey
AND aroma.period.month = 'JAN'
AND aroma.period.year = 2006;
```

You can rewrite this query using table aliases. In our case, we are going to use the alias "a" for the period table and "b" for the sales table:

```sql
SELECT date, dollars
FROM aroma.period a, aroma.sales b
WHERE a.perkey = b.perkey
AND a.month = 'JAN'
AND a.year = 2006;
```

Each qualified table name in the FROM clause is followed by a space and the alias name. You can assign aliases to some, none, or all of the tables listed in the FROM clause.

However, once you assign a table alias in the FROM clause you must use that alias name each time you reference that table in the query. You will encounter error messages if you try to use the qualified table name elsewhere in the query.

When you assign an alias name you **must** also be careful not to use the name of an
existing table in the schema or your results may be incorrect.

In the above example, assume that user dbuser has signed on and has not changed the default schema. If there were already a table named "a" in the schema, the system would look for the column dbuser.a.perkey instead of replacing it with the value aroma.period.perkey.

Usage note

Most queries in this guide will take advantage of table aliases.

Section 3. Using simple joins

Question

What were the daily sales totals for Easter products sold on weekends on a type 900 promotion in 2005 and which stores registered those sales?

Example query

```
SELECT prod_name, store_name, day, dollars 
FROM aroma.promotion a, aroma.product b, 
aroma.period c, aroma.store d, aroma.sales e 
WHERE      a.promokey = e.promokey 
AND        b.prodkey = e.prodkey 
AND        b.classkey = e.classkey 
AND        c.perkey = e.perkey 
AND        d.storekey = e.storekey 
AND        prod_name LIKE 'Easter%' 
AND        day IN ('SA', 'SU') 
AND        promo_type = 900 
AND        year = 2005;
```

Result

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Day</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easter Sampler</td>
<td>Olympic Coffee</td>
<td>SA</td>
<td>150.00</td>
</tr>
<tr>
<td>Basket</td>
<td>Company</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

About the query
This business question requires a join of five tables in the Aroma retail schema: the Sales fact table and its Product, Period, Store, and Promotion tables. Refer back to Part 1 of this tutorial series to see the table layout for the Aroma database.

To join tables in a query, you must give the database server explicit instructions on how to perform the join. The joins are specified in the WHERE clause with five simple conditions that join the Sales table over its five primary key columns. The Product table has a two-part primary key, so it is joined to the Sales table over two columns: Prodkey and Classkey.

Usage notes

Any two tables can be joined over columns with comparable data types; joins are not dependent on the primary-key to foreign-key relationships used in this example.

Section 4. Using the ORDER BY clause

Question

What were the sales figures for Assam Gold Blend and Earl Grey at the Instant Coffee store during November 2005? Order the figures for each product from highest to lowest.

Example query

```
SELECT prod_name, store_name, dollars
FROM aroma.store a, aroma.sales b, aroma.product c,
aroma.period d
WHERE a.storekey = b.storekey
AND c.prodkey = b.prodkey
AND c.classkey = b.classkey
AND d.perkey = b.perkey
AND (prod_name like 'Assam Gold%' OR prod_name LIKE 'Earl%')
AND store_name LIKE 'Instant %'
AND month = 'NOV'
AND year = 2005
ORDER BY prod_name, dollars DESC;
```

Result

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>96.00</td>
</tr>
</tbody>
</table>
Ordering the result table: ORDER BY clause

You can use the ORDER BY clause to sort the result table of a query by the values in one or more specified columns. The default sort order is ascending (ASC); the DESC keyword changes the sort order to descending for the specified column, as follows:

ORDER BY prod_name, 3 DESC

Syntax of the ORDER BY clause

```
SELECT column name(s)
FROM table name(s)
[WHERE search_condition]
[ORDER BY order_list];
```

order_list  A list of columns by which data is ordered. Columns in the order_list need not occur in the select_list but must exist in tables referenced in the FROM clause.

About the query

The example query retrieves Assam Gold Blend and Earl Grey sales figures at the
Instant Coffee store during November 2005. The query sorts the results by product and total daily sales.

**Usage notes**

The ORDER BY clause must follow the other clauses in the SELECT statement and include a list of columns to be ordered. A column can be referenced by its name, column alias, or position (ordinal number) in the select list. For example, the ORDER BY clause on the facing page could be written as follows:

```
ORDER BY prod_name, 3 DESC
```

By specifying columns in order_list that are not in the column name(s), you can order data by columns that are not displayed in the result table.

---

**Section 5. Join of two tables**

Use the following sample table for the initial join query examples:

<table>
<thead>
<tr>
<th>State Table</th>
<th>Region Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td>State</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
</tr>
</tbody>
</table>

**Example query**

```
SELECT *
FROM aroma.state, aroma.region;
```

**Cartesian Product (join predicate not specified)**

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>City</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>
Note that your results may differ in the order presented. Without an "ORDER BY" clause, the system returns rows in whatever sequence they are found.

Example query

```sql
SELECT * 
FROM aroma.state, aroma.region 
WHERE state.city = region.city;
```

Subset of Cartesian Product (join predicate specified)

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
</tbody>
</table>

Inner joins

Most queries join information from different tables. Any two tables can be joined over columns with comparable data types; joins are not dependent on primary-key to foreign-key relationships.

Cartesian product

When two or more tables are referenced in the FROM clause of a query, the database server joins the tables. If neither the FROM clause nor the WHERE clause specifies a predicate for the join, the server computes a Cartesian product that contains \( m \times n \) rows, where \( m \) is the number of rows in the first table and \( n \) is the number of rows in the second table. This product is the set of all possible combinations formed by concatenating a row from the first table with a row from the second table.

Subset of the Cartesian product
If tables are explicitly joined over columns with comparable datatypes, the server computes a subset of the Cartesian product. This subset contains only those rows where the values in the joining columns match. For the duration of the query, the subset functions as a derived table that can be joined with other tables or the results of other query expressions.

About the query

The State and Region tables both contain City columns, which are specified as the joining columns in the WHERE clause. Consequently, only those rows of the Cartesian product that have matching City keys are displayed in the result. In the example query, the result table contains only two rows, whereas the full Cartesian product of these two tables contains nine rows.

Section 6. A different way to join tables

Question

Display a list of all product names that begin with an uppercase letter "A" and their associated class types. Order the list alphabetically by product.

Example query 1

```
SELECT prod_name, class_type
FROM aroma.product t, aroma.class c
WHERE t.classkey = c.classkey
    AND prod_name LIKE 'A%'
ORDER BY prod_name;
```

Example query 2

```
SELECT prod_name, class_type
FROM aroma.product t
JOIN aroma.class c ON t.classkey = c.classkey
WHERE prod_name LIKE 'A%'
ORDER BY prod_name;
```

Two queries; same result
Joins in the FROM clause

You can explicitly join tables in the FROM clause using the ON syntax as shown in the example above.

About the query

This query joins the Product and Class tables over columns with identical names. The resulting values in the ON clause therefore resemble the values shown in the query that constrains the join in the WHERE clause.

Usage notes

There is no performance difference between queries that constrain table joins in the FROM clause using the ON syntax and those that constrain in the WHERE clause. The decision of which constraint to use is up to the person creating the query.

Some people prefer the FROM clause constraint syntax because it clearly separates constraints being used for joining tables from those being used to limit the result sets. Others prefer listing all constraints in the WHERE clause.

The examples shown, however, do join tables across the primary key / foreign key relationship. This is generally the most efficient way to join tables. More details can be found in the SQL Reference Guide.

Section 7. Self-joins

The tables being joined in a query do not need to be distinct; you can join any table to itself as long as you give each table reference a different name. Self-joins are useful for discovering relationships between different columns of data in the same table.

Question
Which products in the Product table have the same names but different types of packaging?

**Example query**

```
SELECT a.prod_name AS products, a.pkg_type
FROM aroma.product a, aroma.product b
WHERE a.prod_name = b.prod_name
  AND a.pkg_type <> b.pkg_type
ORDER BY products, a.pkg_type;
```

**Result**

<table>
<thead>
<tr>
<th>Product</th>
<th>Pkg_Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Colombiano</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>No pkg</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>One-pound bag</td>
</tr>
</tbody>
</table>
About the query

This query joins the Product table to itself over the Prod_Name column, using the aliases a and b to distinguish the table references:

```
FROM aroma.product a, aroma.product b
```

The self-join compares Product table a to Product table b to find rows where the product names match but the package types differ:

```
WHERE a.prod_name = b.prod_name
AND a.pkg_type <> b.pkg_type
```

The result set consists of a list of each pair of identically named products and their individual package types.

---

Section 8. Outer join of two tables

<table>
<thead>
<tr>
<th>State Table</th>
<th>Region Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td>State</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
</tr>
</tbody>
</table>

The above two tables are used to demonstrate left outer join, right outer join, and full outer join. For simplicity, the schema aroma has not been included in the examples.

Example query (left outer join)
```sql
SELECT *
FROM state LEFT OUTER JOIN region
    ON state.city = region.city;
```

Result

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>[null]</td>
<td>[null]</td>
</tr>
</tbody>
</table>

Example query (right outer join)

```sql
SELECT *
FROM state RIGHT OUTER JOIN region
    ON state.city = region.city;
```

Result

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>[null]</td>
<td>[null]</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>

Example query (full outer join)

```sql
SELECT *
FROM state FULL OUTER JOIN region
    ON state.city = region.city;
```

Result

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>[null]</td>
<td>[null]</td>
</tr>
<tr>
<td>[null]</td>
<td>[null]</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>
Important: These examples use the tables introduced at the beginning of this tutorial.

Outer joins

In most cases, tables are joined according to search conditions that find only the rows with matching values; this type of join is known as an **inner join**. In some cases, however, decision-support analysis requires **outer joins**, which retrieve both matching and non-matching rows, which express, for example, a greater-than or less-than relationship.

An outer join operation returns all the rows returned by an inner join plus all the rows from one table that do not match any row from the other table. An outer join can be **left**, **right**, or **full**, depending on whether rows from the left, right, or both tables are retained. The first table listed in the FROM clause is referred to as the left table and the second as the right table. For all three types of outer join, NULLs are used to represent empty columns in rows that do not match.

**Syntax**

As shown in the preceding examples, an outer join between two tables can be specified in the FROM clause with the OUTER JOIN keywords followed by the ON subclause:

```
FROM table_1 LEFT|RIGHT|FULL OUTER JOIN table_2
ON table_1.column = table_2.column
```

For details about other ways to specify outer join predicates in the FROM clause, refer to the *SQL Reference Guide*.

**About the queries**

- The result of the left outer join contains every row from the **State** table and all matching rows in the **Region** table. Rows found only in the **Region** table are not displayed.
- The result of the right outer join contains every row from the **Region** table and all matching rows from the **State** table. Rows found only in the **State** table are not displayed.
- The result of the full outer join contains those rows that are unique to
each table, as well as those rows that are common to both tables.

Section 9. Using the SUM, AVG, MAX, MIN, COUNT set functions

Question
What were the total Lotta Latte sales figures in Los Angeles for 2005? What were the average, maximum, and minimum daily sales figures for that year, and how many daily totals were counted to produce these aggregate values?

Example query

```
SELECT SUM(dollars) as Dol_Sales, AVG(dollars) as Avg_Sales,
       MAX(dollars) as Max_Sales, MIN(dollars) as Min_Sales,
       COUNT(*) as Qty
FROM aroma.store a, aroma.sales b, aroma.product c, aroma.period d
WHERE  a.storekey = b.storekey
       AND c.prodkey = b.prodkey
       AND c.classkey = b.classkey
       AND d.perkey = b.perkey
       AND prod_name LIKE 'Lotta Latte%'
       AND year = 2005
       AND city LIKE 'Los Ang%';
```

Result

<table>
<thead>
<tr>
<th>Dol_Sales</th>
<th>Avg_Sales</th>
<th>Max_Sales</th>
<th>Min_Sales</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>13706.50</td>
<td>171.33125</td>
<td>39.00</td>
<td>39.00</td>
<td>80</td>
</tr>
</tbody>
</table>

Using set functions

Set functions operate on groups of values. For example, SUM(dollars) calculates the total dollars returned in a result table, and AVG(dollars) returns the average. The SQL set functions listed in the following table can occur one or more times in the select list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>

### SUM(expression)
Calculates the sum of all the values in expression.

### SUM(DISTINCT expression)
Calculates the sum of distinct values in expression.

### AVG(expression)
Calculates the average of all the values in expression.

### AVG(DISTINCT expression)
Calculates the average of distinct values in expression.

### MAX(expression)
Determines the maximum value in expression.

### MIN(expression)
Determines the minimum value in expression.

### COUNT(*)
Counts the number of rows returned.

### COUNT(expression)
Counts the number of non-null values in expression.

### COUNT(DISTINCT expression)
Counts the number of distinct non-null values in expression.

You can replace expression with any column name or numeric expression. Each function, except COUNT(*), ignores NULL values when calculating the returned aggregate value.

### About the query
The example query retrieves sales figures for Lotta Latte in Los Angeles during 2005. The result set also includes the average, maximum, and minimum sales during the year, and the number of daily totals on which those calculations are based.

### Usage notes
If the result set will contain individual values as well as aggregate values, the query must contain a GROUP BY clause. See the next section about the GROUP BY clause.

---

### Section 10. Using the GROUP BY clause to group rows

### Question
What were the annual totals for sales of coffee mugs in 2004 in each district? What were the average, maximum, and minimum sales during this time period? List the results by district.

**Example query**

```sql
SELECT district AS district_city, SUM(dollars) AS dol_sales,
       AVG(dollars) AS avg_sales, MAX(dollars) AS max_sales,
       MIN(dollars) AS min_sales
FROM aroma.store a, aroma.sales b, aroma.product c,
     aroma.period d, aroma.market e
WHERE  a.storekey = b.storekey
       AND c.prodkey = b.prodkey
       AND c.classkey = b.classkey
       AND d.perkey = b.perkey
       AND e.mktkey = a.mktkey
       AND prod_name LIKE '%Mug%'
       AND year = 2004
GROUP BY district
ORDER BY dol_sales DESC;
```

**Result**

<table>
<thead>
<tr>
<th>District_City</th>
<th>Dol_Sales</th>
<th>Avg_Sales</th>
<th>Max_Sales</th>
<th>Min_Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>1378.30</td>
<td>35.34102564</td>
<td>98.55</td>
<td>4.00</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>711.60</td>
<td>30.93913043</td>
<td>98.55</td>
<td>9.95</td>
</tr>
<tr>
<td>San Francisco</td>
<td>410.45</td>
<td>25.65312500</td>
<td>54.75</td>
<td>5.00</td>
</tr>
</tbody>
</table>

**Grouping rows: GROUP BY clause**

Set functions operate on all rows of a result table or on groups of rows defined by a GROUP BY clause. For example, you can group the sales for each market and calculate the respective sum, maximum, and minimum values.

**Syntax of the GROUP BY clause**

```sql
SELECT column name(s)
FROM table name(s)
[WHERE search_condition]
[GROUP BY group_list]
[ORDER BY order_list];
```

**group_list**

A list of column names either in *column name(s)* or in tables listed in the FROM clause. All nonaggregated columns in *column name(s)* must
About the query

The example query retrieves annual sales totals for coffee mugs in 2004 (they are sold in three districts only), ordering the figures from highest to lowest. Conceptually speaking, the server processes this query as follows:

1. Retrieves all rows of data from tables specified in the FROM clause, joins the rows from separate tables, and generates an intermediate result table.
2. Retains all rows from the intermediate result table that satisfy the search condition specified in the WHERE clause.
3. Divides the result table into groups specified in the GROUP BY clause.
4. Processes all set functions on specified groups for the entire result table.
5. Orders results according to the ORDER BY clause.
6. Returns only those columns specified in the select list.

Usage notes

A GROUP BY clause references items in the select list by their actual column names. You cannot use a column alias or positional integer in the GROUP BY clause.

Section 11. Using the GROUP BY clause to produce multiple groups

Question

What were the total sales in each city during 2004 and 2005? List city names by year within their region and district.

Example query
```sql
SELECT year, region, district, city, SUM(dollars) AS sales
FROM aroma.store a, aroma.sales b, aroma.product c,
aroma.period d, aroma.market e
WHERE a.storekey = b.storekey
AND c.prodkey = b.prodkey
AND c.classkey = b.classkey
AND d.perkey = b.perkey
AND e.mktkey = a.mktkey
AND year IN (2004, 2005)
GROUP BY year, region, district, city
ORDER BY year, region, district, city;
```

### Result

<table>
<thead>
<tr>
<th>Year</th>
<th>Region</th>
<th>District</th>
<th>City</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Central</td>
<td>Chicago</td>
<td>Chicago</td>
<td>133462.75</td>
</tr>
<tr>
<td>2004</td>
<td>Central</td>
<td>Chicago</td>
<td>Detroit</td>
<td>135023.50</td>
</tr>
<tr>
<td>2004</td>
<td>Central</td>
<td>Minneapolis</td>
<td>Milwaukee</td>
<td>172321.50</td>
</tr>
<tr>
<td>2004</td>
<td>North</td>
<td>Boston</td>
<td>Boston</td>
<td>184647.50</td>
</tr>
<tr>
<td>2004</td>
<td>North</td>
<td>Boston</td>
<td>Hartford</td>
<td>69196.25</td>
</tr>
<tr>
<td>2004</td>
<td>North</td>
<td>New York</td>
<td>New York</td>
<td>181735.00</td>
</tr>
<tr>
<td>2004</td>
<td>North</td>
<td>New York</td>
<td>Philadelphia</td>
<td>172395.75</td>
</tr>
<tr>
<td>2004</td>
<td>South</td>
<td>Atlanta</td>
<td>Atlanta</td>
<td>230346.45</td>
</tr>
<tr>
<td>2004</td>
<td>South</td>
<td>Atlanta</td>
<td>Miami</td>
<td>220519.75</td>
</tr>
<tr>
<td>2004</td>
<td>South</td>
<td>New Orleans</td>
<td>Houston</td>
<td>183853.75</td>
</tr>
<tr>
<td>2004</td>
<td>South</td>
<td>New Orleans</td>
<td>New Orleans</td>
<td>193052.25</td>
</tr>
<tr>
<td>2004</td>
<td>West</td>
<td>Los Angeles</td>
<td>Los Angeles</td>
<td>219397.20</td>
</tr>
<tr>
<td>2004</td>
<td>West</td>
<td>Los Angeles</td>
<td>Phoenix</td>
<td>192605.25</td>
</tr>
<tr>
<td>2004</td>
<td>West</td>
<td>San Francisco</td>
<td>Cupertino</td>
<td>180088.75</td>
</tr>
<tr>
<td>2004</td>
<td>West</td>
<td>San Francisco</td>
<td>Los Gatos</td>
<td>176992.75</td>
</tr>
<tr>
<td>2004</td>
<td>West</td>
<td>San Francisco</td>
<td>San Jose</td>
<td>395330.25</td>
</tr>
<tr>
<td>2005</td>
<td>Central</td>
<td>Chicago</td>
<td>Chicago</td>
<td>131263.00</td>
</tr>
<tr>
<td>2005</td>
<td>Central</td>
<td>Chicago</td>
<td>Detroit</td>
<td>136903.25</td>
</tr>
<tr>
<td>2005</td>
<td>Central</td>
<td>Minneapolis</td>
<td>Milwaukee</td>
<td>173844.25</td>
</tr>
<tr>
<td>2005</td>
<td>Central</td>
<td>Minneapolis</td>
<td>Minneapolis</td>
<td>132125.75</td>
</tr>
<tr>
<td>2005</td>
<td>North</td>
<td>Boston</td>
<td>Boston</td>
<td>189761.00</td>
</tr>
<tr>
<td>2005</td>
<td>North</td>
<td>Boston</td>
<td>Hartford</td>
<td>135879.50</td>
</tr>
<tr>
<td>2005</td>
<td>North</td>
<td>New York</td>
<td>New York</td>
<td>171749.75</td>
</tr>
<tr>
<td>2005</td>
<td>North</td>
<td>New York</td>
<td>Philadelphia</td>
<td>171759.50</td>
</tr>
</tbody>
</table>
Nesting grouped results: GROUP BY clause

When several column names occur in a GROUP BY clause, the result table is divided into groups within groups. For example, if you specify column names for year, region, and district in the GROUP BY clause, the returned figures are divided by year, each year is divided by region, and each region is divided by district.

In the above example, items were grouped by year, region, district, and city providing an even finer level of granularity. Had you omitted the city column, the result set would have been shorter and at a coarser level of granularity.

```
SELECT year, region, district, SUM(dollars) AS sales
FROM aroma.store a, aroma.sales b, aroma.product c,
     aroma.period d, aroma.market e
WHERE a.storekey = b.storekey
  AND c.prodkey = b.prodkey
  AND c.classkey = b.classkey
  AND d.perkey = b.perkey
  AND e.mktkey = a.mktkey
GROUP BY year, region, district
ORDER BY year, region, district;
```

Result

<table>
<thead>
<tr>
<th>Year</th>
<th>Region</th>
<th>District</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Central</td>
<td>Chicago</td>
<td>268486.25</td>
</tr>
<tr>
<td>2004</td>
<td>Central</td>
<td>Minneapolis</td>
<td>172321.50</td>
</tr>
<tr>
<td>2004</td>
<td>North</td>
<td>Boston</td>
<td>253843.75</td>
</tr>
<tr>
<td>2004</td>
<td>North</td>
<td>New York</td>
<td>354130.75</td>
</tr>
<tr>
<td>2004</td>
<td>South</td>
<td>Atlanta</td>
<td>450866.20</td>
</tr>
<tr>
<td>2004</td>
<td>South</td>
<td>New Orleans</td>
<td>376906.00</td>
</tr>
</tbody>
</table>
### Syntax of the GROUP BY clause

```
SELECT column name(s)
FROM table name(s)
[WHERE search_condition]
[GROUP BY group_list]
[ORDER BY order_list];
```

**group_list**

A list of column names either in `column name(s)` or in tables listed in the FROM clause. All non-aggregated columns in `column name(s)` must appear in `group_list`.

### About the query

The example query retrieves annual sales of all products for each city during 2004 and 2005. The sales figures are both grouped and ordered by year, region, district, and city.

<table>
<thead>
<tr>
<th>Year</th>
<th>Region</th>
<th>City</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>West</td>
<td>Los Angeles</td>
<td>412002.45</td>
</tr>
<tr>
<td>2004</td>
<td>West</td>
<td>San Francisco</td>
<td>752411.75</td>
</tr>
<tr>
<td>2005</td>
<td>Central</td>
<td>Chicago</td>
<td>268166.25</td>
</tr>
<tr>
<td>2005</td>
<td>Central</td>
<td>Minneapolis</td>
<td>305970.00</td>
</tr>
<tr>
<td>2005</td>
<td>North</td>
<td>Boston</td>
<td>325640.50</td>
</tr>
<tr>
<td>2005</td>
<td>North</td>
<td>New York</td>
<td>343509.25</td>
</tr>
<tr>
<td>2005</td>
<td>South</td>
<td>Atlanta</td>
<td>464073.95</td>
</tr>
<tr>
<td>2005</td>
<td>South</td>
<td>New Orleans</td>
<td>376836.00</td>
</tr>
<tr>
<td>2005</td>
<td>West</td>
<td>Los Angeles</td>
<td>425477.50</td>
</tr>
<tr>
<td>2005</td>
<td>West</td>
<td>San Francisco</td>
<td>770317.60</td>
</tr>
<tr>
<td>2006</td>
<td>Central</td>
<td>Chicago</td>
<td>64190.00</td>
</tr>
<tr>
<td>2006</td>
<td>Central</td>
<td>Minneapolis</td>
<td>76417.50</td>
</tr>
<tr>
<td>2006</td>
<td>North</td>
<td>Boston</td>
<td>78494.25</td>
</tr>
<tr>
<td>2006</td>
<td>North</td>
<td>New York</td>
<td>91840.25</td>
</tr>
<tr>
<td>2006</td>
<td>South</td>
<td>Atlanta</td>
<td>106912.20</td>
</tr>
<tr>
<td>2006</td>
<td>South</td>
<td>New Orleans</td>
<td>93156.75</td>
</tr>
<tr>
<td>2006</td>
<td>West</td>
<td>Los Angeles</td>
<td>103876.15</td>
</tr>
<tr>
<td>2006</td>
<td>West</td>
<td>San Francisco</td>
<td>192503.30</td>
</tr>
</tbody>
</table>

**Important**: The cities referred to in this query are...
Section 12. OR versus UNION

Question

What were the total sales in week 52 of 2005 for all Aroma stores classified as "Medium"? What were the totals during the same period for "Large" stores?

Example query with OR condition

```sql
SELECT store_name AS store, store_type AS size, state, SUM(dollars) AS sales
FROM aroma.period p
JOIN aroma.sales s ON p.perkey = s.perkey
JOIN aroma.store r ON r.storekey = s.storekey
WHERE (store_type = 'Medium'
OR store_type = 'Large')
AND year = 2005
AND week = 52
GROUP BY store_name, store_type, state
ORDER BY 2, 1;
```

Example UNION query

```sql
SELECT store_name AS store, store_type AS size, state, SUM(dollars) AS sales
FROM aroma.period p
JOIN aroma.sales s ON p.perkey = s.perkey
JOIN aroma.store r ON r.storekey = s.storekey
WHERE store_type = 'Medium'
AND year = 2005
AND week = 52
GROUP BY store_name, store_type, state

UNION

SELECT store_name AS store, store_type AS size, state, SUM(dollars) AS sales
FROM aroma.period p
JOIN aroma.sales s ON p.perkey = s.perkey
JOIN aroma.store r ON r.storekey = s.storekey
WHERE store_type = 'Large'
AND year = 2005
AND week = 52
```
Two queries; same result

<table>
<thead>
<tr>
<th>Store</th>
<th>Size</th>
<th>State</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beaches Brew</td>
<td>Large</td>
<td>CA</td>
<td>2908.80</td>
</tr>
<tr>
<td>Miami Espresso</td>
<td>Large</td>
<td>FL</td>
<td>4582.00</td>
</tr>
<tr>
<td>Olympic Coffee Company</td>
<td>Large</td>
<td>GA</td>
<td>3732.50</td>
</tr>
<tr>
<td>San Jose Roasting Company</td>
<td>Large</td>
<td>CA</td>
<td>3933.15</td>
</tr>
<tr>
<td>Beans of Boston</td>
<td>Medium</td>
<td>MA</td>
<td>3772.75</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Medium</td>
<td>CA</td>
<td>2893.00</td>
</tr>
<tr>
<td>Java Judy's</td>
<td>Medium</td>
<td>AZ</td>
<td>3011.25</td>
</tr>
<tr>
<td>Moulin Rouge Roasting</td>
<td>Medium</td>
<td>LA</td>
<td>3972.00</td>
</tr>
<tr>
<td>Texas Teahouse</td>
<td>Medium</td>
<td>TX</td>
<td>3382.75</td>
</tr>
</tbody>
</table>

Combining result sets: UNION

You can use the UNION, EXCEPT, and INTERSECT operators to combine the output of two or more query expressions into a single set of rows and columns. The server evaluates each query expression independently, then combines the output, displaying column headings from the first expression. The server eliminates duplicate result rows unless you specify the ALL keyword.

The union of two sets

![Venn diagram showing A Union B]
UNION, INTERSECT, EXCEPT

query_expression UNION | INTERSECT | EXCEPT [ALL]
query_expression [ORDER BY order_list];

Any join or nonjoin query expression, as defined in the SQL Reference Guide.

If the ORDER BY clause is used, the values must reference columns from the select list of the first query expression. Using column names from the second query expression results in an error.

About the query

The same business question can be answered by either specifying an OR condition in a single SELECT statement or combining two query expressions with a UNION operator.

Using the OR connective is easier in this simple example, but in some cases a UNION operation improves query performance. For example, suppose your query requires that you access data in two large fact tables. The outer join operation required by a single query might require more processing than using a UNION operation to combine the results of two query expressions.

The ORDER BY clause references the column positions, not the column names, defined in the select list of the first query expression:

ORDER BY 1, 2

Usage notes

UNION, INTERSECT, and EXCEPT queries must be symmetrical; that is, the number of columns and their order must be the same in the select lists on both sides of the UNION operator. Corresponding columns must have the same, or comparable, datatypes, although they may have different names.

Multiple UNION, INTERSECT, and EXCEPT operators can be used in a single statement; operations are evaluated from left to right unless you specify precedence with parentheses.

The UNION is useful when joining tables that have similar datatype, yet the column
names are different. In such a case, the default behavior is for the result set uses the column names from the first table.

Section 13. UNION versus UNION ALL

The UNION function includes an internal DISTINCT statement. As a result, duplicates are suppressed and only one instance of each value is displayed. Many business questions profit from this type of behavior.

Question

Provide a list of all cities in which we have facilities, meaning stores, headquarters, or both. List each city only once.

Example query (UNION)

```
SELECT city
FROM aroma.store
UNION
SELECT hq_city AS city
FROM aroma.market;
```

Result

<table>
<thead>
<tr>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
</tr>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Phoenix</td>
</tr>
<tr>
<td>San Francisco</td>
</tr>
<tr>
<td>San Jose</td>
</tr>
</tbody>
</table>

On other occasions, however, you may want to repeat values that occur more than once in the list of unioned objects. For example, you may want to count the total number of objects in each case.
Question

Provide a list of all cities in which we have facilities, meaning stores, headquarters, or both. List each city once for each instance in which a store or headquarters is located there.

Example query (UNION ALL)

```sql
SELECT city
FROM aroma.store
UNION ALL
SELECT hq_city AS city
FROM aroma.market
ORDER BY city;
```

Result

<table>
<thead>
<tr>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
</tr>
<tr>
<td>Atlanta</td>
</tr>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>Cupertino</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>San Jose</td>
</tr>
<tr>
<td>San Jose</td>
</tr>
<tr>
<td>San Jose</td>
</tr>
</tbody>
</table>

In this example, the cities of Atlanta, Boston, Chicago, and San Jose all have more than one facility, whereas Cupertino has only one.

Section 14. INTERSECT operation
Question

Which bulk tea products sold on promotion in San Jose in 2006 were also sold on promotion in New Orleans in 2005? What promotions were run on those products?

Example query

```sql
SELECT prod_name AS tea_name, promo_desc
FROM aroma.class c
JOIN aroma.product d ON c.classkey = d.classkey
JOIN aroma.sales s ON d.prodkey = s.prodkey
AND d.classkey = s.classkey
JOIN aroma.store t ON t.storekey = s.storekey
JOIN aroma.period p ON p.perkey = s.perkey
JOIN aroma.promotion n ON n.promokey = s.promokey
WHERE city = 'San Jose'
AND year = 2006
AND class_desc LIKE 'Bulk tea'
INTERSECT
SELECT prod_name AS tea_name, promo_desc
FROM aroma.class c
JOIN aroma.product d ON c.classkey = d.classkey
JOIN aroma.sales s ON d.prodkey = s.prodkey
AND d.classkey = s.classkey
JOIN aroma.store t ON t.storekey = s.storekey
JOIN aroma.period p ON p.perkey = s.perkey
JOIN aroma.promotion n ON n.promokey = s.promokey
WHERE city = 'New Orleans'
AND year = 2005
AND class_desc LIKE 'Bulk tea'
AND promo_desc NOT LIKE 'No promo'
ORDER BY promo_desc;
```

Result

<table>
<thead>
<tr>
<th>Tea_Name</th>
<th>Promo_Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irish Breakfast</td>
<td>Aroma catalog coupon</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Aroma catalog coupon</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Store display</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Temporary price reduction</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Temporary price reduction</td>
</tr>
</tbody>
</table>

Finding common rows: INTERSECT

You can use the INTERSECT operator to return only those rows that are common to the results returned by two or more query expressions.
The intersection of two sets

![Venn diagram showing intersection of sets A and B]

About the query

The example query finds the intersection of two query expressions, one that returns a list of bulk tea products sold on promotion in San Jose in 2006 and one that returns a similar list for New Orleans in 2005. The INTERSECT operator eliminates all rows that are not found in both preliminary result sets.

Section 15. EXCEPT operation

Question

What were the total 2005 revenues for stores in California cities that are not defined as HQ cities in the Market table?

Example query

```sql
SELECT x.city, store_name, SUM(dollars) AS sales_05
FROM
(SELECT city
     FROM aroma.store
     WHERE state='CA'
EXCEPT
SELECT hq_city
     FROM aroma.market
     WHERE hq_state = 'CA') AS x(city)
JOIN aroma.store t ON x.city = t.city
JOIN aroma.sales s ON s.storekey = t.storekey
JOIN aroma.period p ON p.perkey = s.perkey
```
WHERE year = 2005
GROUP BY x.city, store_name
ORDER BY 3 DESC;

Result

<table>
<thead>
<tr>
<th>City</th>
<th>Store_Name</th>
<th>Sales_05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cupertino</td>
<td>Cupertino Coffee Supply</td>
<td>196,439.75</td>
</tr>
<tr>
<td>Los Gatos</td>
<td>Roasters, Los Gatos</td>
<td>175,048.75</td>
</tr>
</tbody>
</table>

EXCEPT: Finding the exceptions in two result sets

The EXCEPT operator finds the exceptions in (or the difference between) the results of two query expressions. For example, an EXCEPT operation could compare lists of products sold at two stores, eliminate all the products sold at both, and retain only those products sold exclusively at the store specified in the first query expression.

The exceptions in two sets

About the query

In the example query, the function of the EXCEPT operator is to select those California cities that are defined in the City column of the Store table but not in the Hq_City column of the Market table.

This query uses a subquery in the FROM clause to produce a derived table of cities that can be joined with the Sales, Store, and Period tables. The table derived from the subquery is given a correlation name and one column name: x(city)

This derived table can be joined with the Store table using a join over the City column.
Usage notes

To test the outcome of the EXCEPT operation, you could run the subquery in this example as a query in its own right:

```
(SELECT city
 FROM aroma.store
 WHERE state = 'CA'
 EXCEPT
 SELECT hq_city as city
 FROM aroma.market
 WHERE hq_state = 'CA');
```

Result

<table>
<thead>
<tr>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cupertino</td>
</tr>
<tr>
<td>Los Gatos</td>
</tr>
</tbody>
</table>

For more examples of subqueries, refer to Part 5 of this series.

Section 16. Summary

Summary

This tutorial described:

- How to join tables
- How to combine the results of two independent query expressions by using the UNION, INTERSECT, and EXCEPT operators

It also covered the following topics:

- table names and schemas
- the ORDER BY clause
- the SUM, AVG, MAX, MIN, COUNT set functions, and
• the GROUP BY clause
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Database</td>
<td>Aroma_Data.zip</td>
<td>1MB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

Information about download methods
Resources

Learn

- View this article series' "Appendix A" (developerWorks, August 2006).
- Read "DB2 XML evaluation guide" (developerWorks, June 2006), a step-by-step tutorial introducing the reader to the DB2 Viper data server on Windows platforms using the XML storage and searching (SQL/XML, XQuery) capabilities available to support next-generation applications.
- Check out this article and "Get off to a fast start with DB2 Viper" (developerWorks, March 2006).
- Learn how to "Query DB2 XML data with XQuery" (developerWorks, April 2006).
- Learn how to "Query DB2 XML data with SQL" (developerWorks, March 2006).
- Read the IBM Systems Journal and celebrate 10 years of XML.
- Refer to the SQL Reference, Vol 1 for additional information.
- Refer to the SQL Reference, Vol 2 for additional information.
- Refer to the DB2 information Center for troubleshooting.
- Visit the DB2 XML technical enablement space for links to more than 25 papers on DB2 XML capabilities.

Get products and technologies

- Download DB2 Express-C, a no-charge data server for use in development and deployment of applications.

Discuss

- Participate in the discussion forum for this content.
- Visit the DB2 9 On-line Support Forum.

About the authors

Pat Moffatt
Pat Moffatt is the Information Management Program Manager for the IBM Academic Initiative. Through the Academic Initiative program, she ensures that appropriate Information Management resources are made available to help faculty integrate Information Management software into their curriculum. To learn more about this program, visit www.ibm.com/university/data.
Bruce Creighton
Bruce Creighton is a Skills Segment Planner in the Information Management Education Planning and Development department. In this role, he plans investment in educational content and balances the investment between areas where IBM can attain revenue and those where the requirement for skills development are important enough to provide free education.

Jessica Cao
Jessica Cao is an Arts and Science and Computer Science student at McMaster University. She expects to complete her combined honours degree in April 2009. Jessica is working in IBM Toronto lab's DB2 Information Management Skills Channel Planning and Enablement Program to take advantage of her interest in programming, editing, and writing.

Trademarks
IBM, DB2, Universal Database, OS/2, and pureXML are registered trademarks of IBM Corporation in the United States and other countries.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.