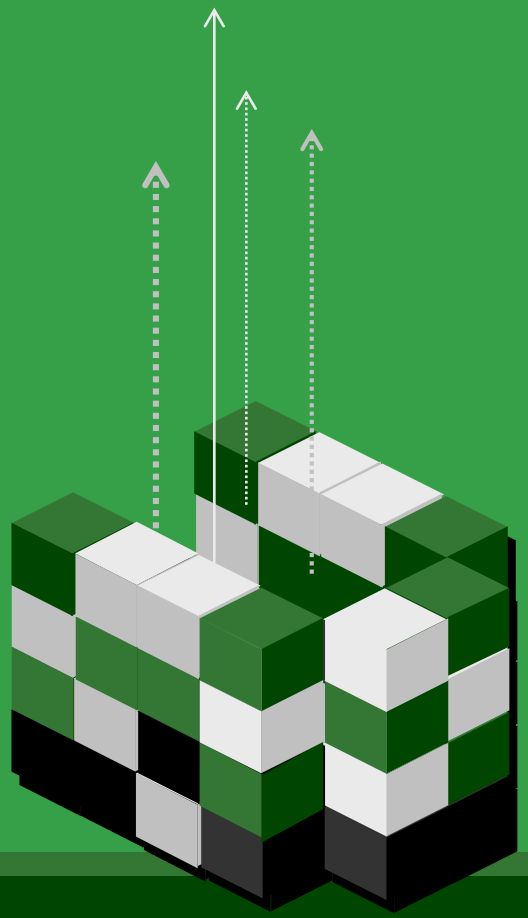
chaper 2

Introduction to Structured Query Language

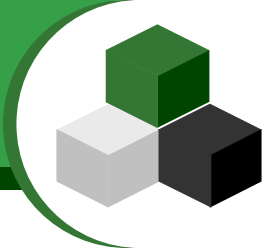


8、Looking for Patterns in NASDAQ Trading

Before we continue our discussion of SQL, consider an example problem that will illustrate the power of the SQL just described.

Suppose that a friend tells you that she suspects the stock market tends to go up on certain days of the week and down on others. She asks you to investigate past trading data to determine if this is true. Specifically, she wants to trade an index fund called the NASDAQ 100, which is a stock fund of the 100 top companies traded on the NASDAQ stock exchange. She gives you a dataset with 20 years (1985-2004) of NASDAQ 100 trading data for analysis. Assume she gives you the data in the form of a table named NDX containing 4611 rows of data for use with a relational database (this dataset is available on the text's Web site at www. pearsonhighered.com/kroenke).

8、Looking for Patterns in NASDAQ Trading



- **Investigating the Characteristics of the Data**

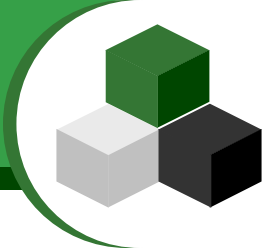
Suppose you first decide to investigate the general characteristics of the data. You begin by seeing what columns are present in the table by issuing the SQL query:

```
/* *** SQL-Query-NDX-CH02-01 ***    */
SELECT        *
FROM  NDX;
```

The first five rows of that query are as follows:

	TClose	PriorClose	ChangeClose	Volume	TMonth	TDayOfMonth	TYear	TDayOfWeek	TQuarter
1	1520.46	1530.65	-10.1900000000001	24827600	January	9	2004	Friday	1
2	1530.65	1514.26	16.3900000000001	26839500	January	8	2004	Thursday	1
3	1514.26	1501.26	13	22942800	January	7	2004	Wednesday	1
4	1501.26	1496.58	4.68000000000006	22732200	January	6	2004	Tuesday	1
5	1496.58	1463.57	33.01	23629100	January	5	2004	Monday	1

8、Looking for Patterns in NASDAQ Trading



- **Investigating the Characteristics of the Data**

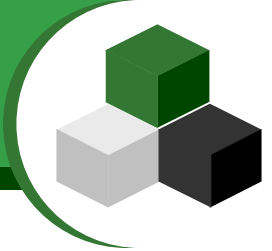
Assume that you learn that the first column has the value of the fund at the close of a trading day, the second column has the value of the fund at the close of the prior trading day, andthe third row has the difference between the current day ’s close and the prior day ’s close.Volume is the number of shares traded, and the rest of the data concerns the trading date.Next, you decide to investigate the change of the stock price by issuing the SQL query:

```
/* *** SQL-Query-NDX-CH02-02 *** */
SELECT AVG(ChangeClose) AS AverageChange,
MAX(ChangeClose) AS MaxGain,
MIN(ChangeClose) AS MaxLoss
FROM NDX;
```

The result of this query is:

	AverageChange	MaxGain	MaxLoss
1	0.281167028199584	399.6	-401.03

8、Looking for Patterns in NASDAQ Trading



By the way **DBMS products have many functions for formatting query results to reduce the number of decimal points displayed, to add currency characters such as $ or £, or to make other formatting changes. However, these functions are DBMSdependent. Search the documentation of your DBMS for the term formatting results to learn more about such functions.**

8、Looking for Patterns in NASDAQ Trading

- **Investigating the Characteristics of the Data**

Just out of curiosity, you decide to determine which days had the maximum and minimum change. To avoid having to key in the long string of decimal places that would be required to make an equal comparison, you use a greater than and less than comparison with values that are close:

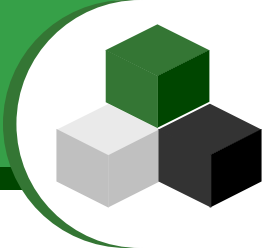
```
/* *** SQL-Query-NDX-CH02-03 ***    */
SELECT        ChangeClose, TMonth, TDayOfMonth, TYear
FROM  NDX
WHERE         ChangeClose  >  398
OR     ChangeClose  <  -400;
```

The result is:

	ChangeClose	TMonth	TDayOfMonth	TYear
1	-401.03	January	3	1994
2	399.6	January	3	2001

This result is surprising! Is there some reason that both the greatest loss and the greatest gain both occurred on January 3? You begin to wonder if your friend might have a promising idea.

8、Looking for Patterns in NASDAQ Trading



- **Searching for Patterns in Trading by Day of Week**

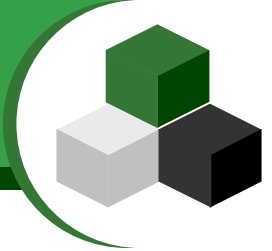
You want to determine if there is a difference in the average trade by day of week. Accordingly, you create the SQL query:

```
/* *** SQL-Query-NDX-CH02-04 *** */
SELECT TDayOfWeek, AVG(ChangeClose) AS AvgChange
FROM NDX
GROUP BY TDayOfWeek;
```

The result is:

	TDayOfWeek	AvgChange
1	Wednesday	0.777940552017005
2	Monday	-1.03577929465299
3	Friday	0.146021739130452
4	Thursday	2.17412972972975
5	Tuesday	-0.711440677966085

8、Looking for Patterns in NASDAQ Trading



- **Searching for Patterns in Trading by Day of Week**

Indeed, there does seem to be a difference according to the day of the week. The NASDAQ 100 appears to go down on Monday and Tuesday and then go up on the other three days of the week. Thursday, in particular, seems to be a good day to trade long.

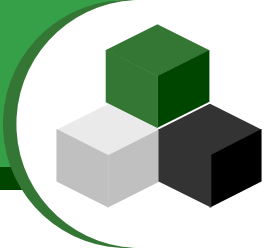
But, you begin to wonder, is this pattern true for each year? To answer that question, you use the query:

```
/* *** SQL-Query-NDX-CH02-05 ***    */
SELECT      TDayOfWeek, TYear, AVG(ChangeClose) AS
AvgChange
FROM NDX
GROUP BY    TD
ORDER BY    TD
```

Because there are 2(ı 100 rows, of which the first 12 are shown in the foll

	TDayOfWeek	TYear	AvgChange
1	Friday	2004	-7.2700000000001
2	Friday	2003	-2.48499999999996
3	Friday	2002	-2.19419999999997
4	Friday	2001	-19.5944
5	Friday	2000	8.8980392156863
6	Friday	1999	13.9656000000001
7	Friday	1998	5.2640816326531
8	Friday	1997	-0.194799999999989
9	Friday	1996	0.819019607843153
10	Friday	1995	0.691372549019617
11	Friday	1994	0.123725490196082
12	Friday	1993	-0.899399999999989

8、Looking for Patterns in NASDAQ Trading



- **Searching for Patterns in**

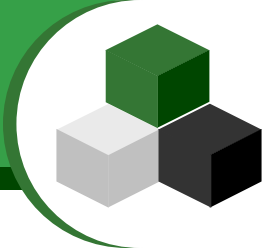
To simplify your analysis, y f rows to the most recent 5 years (2000-2004):

```
/* *** SQL-Query-ND
SELECT      TDayOfW                    geClose) AS
AvgChange
FROM  NDX
WHERE       TYear >
GROUP  BY   TDayOfW
ORDER  BY   TDayOfW
```

Partial results from this que

	TDayOfWeek	TYear	AvgChange
1	Friday	2004	-7.2700000000001
2	Friday	2003	-2.48499999999996
3	Friday	2002	-2.19419999999997
4	Friday	2001	-19.5944
5	Friday	2000	8.8980392156863
6	Monday	2004	33.01
7	Monday	2003	3.77416666666668
8	Monday	2002	-2.60229166666664
9	Monday	2001	-3.7527083333333
10	Monday	2000	-19.899574468085
11	Thursday	2004	16.390000000001
12	Thursday	2003	5.70700000000002
13	Thursday	2002	-3.77979999999998
14	Thursday	2001	9.31440000000003
15	Thursday	2000	24.766274509804
16	Tuesday	2004	4.68000000000006
17	Tuesday	2003	4.41307692307694
18	Tuesday	2002	-7.85882352941176
19	Tuesday	2001	-8.88459999999997
20	Tuesday	2000	-3.50627450980385
21	Wednesday	2004	13
22	Wednesday	2003	-1.69596153846152
23	Wednesday	2002	4.54372549019611
24	Wednesday	2001	7.47420000000005
25	Wednesday	2000	-37.8636538461538

8、Looking for Patterns in NASDAQ Trading



- **Searching for Patterns in Trading by Day of Week**

Alas, it does not appear that day of week is a very good predictor of gain or loss. At least, not for this fund over this period of time. We could continue this discussion to further analyze this data, but by now you should understand how useful SQL can be for analyzing and processing a table. Suggested additional NDX analysis exercises are included in the SQL problems at the end of this chapter.

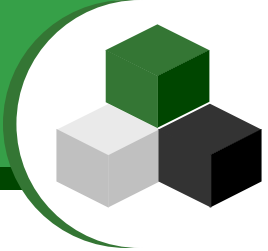
9、Querying Two or More Tables with SQL

So far in this chapter we've worked with only one table. Now we will conclude by describing SQL statements for querying two or more tables.

Suppose that you want to know the revenue generated by SKUs managed by the Water Sports department. We can compute revenue as the sum of ExtendedPrice, but we have a problem. ExtendedPrice is stored in the ORDER_ITEM table, and Department is stored in the SKU_DATA table. We need to process data in two tables, and all of the SQL presented so far operates on a single table at a time.

SQL provides two different techniques for querying data from multiple tables: subqueries and joins. Although both work with multiple tables, they are used for slightly different purposes, as you will learn.

9、Querying Two or More Tables with SQL



❖ **Querying Multiple Tables with Subqueries**

How can we obtain the sum of ExtendedPrice for items managed by the Water Sports department? If we somehow knew the SKU values for those items, we could use a WHERE clause with the IN keyword.

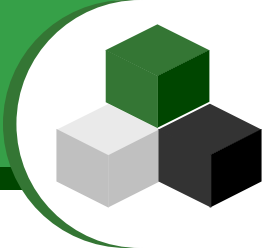
For the data in Figure 2-4, the SKU values for items in Water Sports are 100100, 100200,101100, and 101200. Knowing those values, we can obtain the sum of their ExtendedPrice withthe following SQL query:

```
/* *** SQL-Query-CH02-46 *** */
SELECT      SUM(ExtendedPrice) AS Revenue
FROM ORDER_ITEM
WHERE       SKU IN (100100, 100200, 101100, 101200);
```

The result is:

	Revenue
1	750.00

9、Querying Two or More Tables with SQL



❖ **Querying Multiple Tables with Subqueries**

But, in general, we do not know the necessary SKU values ahead of time. However, we do have a way to obtain them from an SQL query on the data in the SKU_DATA table. To obtain the SKU values for the Water Sports department, we use the SQL statement:

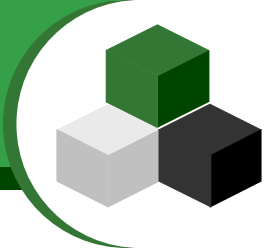
```
/* *** SQL-Query-CH02-47 *** */
SELECT      SKU
FROM  SKU_DATA
WHERE       Department='Water  Sports'
```

The result of this SQ[illegible]t is:

	SKU
1	100100
2	100200
3	101100
4	101200

which is, indeed, the desired list of SKU values.

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Subqueries**

  Now we need only combine the last two SQL statements to obtain the result we want. We replace the list of values in the WHERE clause of the first SQL query with the second SQL statement as follows:

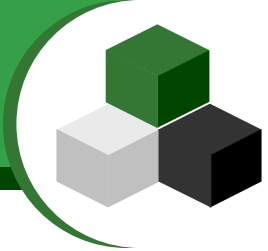
```
/* *** SQL-Query-CH02-48 *** */
SELECT      SUM(ExtendedPrice) AS Revenue
FROM ORDER_ITEM
WHERE       SKU IN
(SELECT     SKU
FROM SKU_DATA
WHERE       Department='Water Sports');
```

The result of the query is:

	Revenue
1	750.00

which is the same result we obtained before when we know the values of SKU to use.

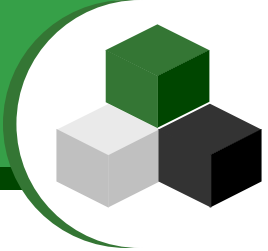
9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Subqueries**

  In the preceding SQL query, the second SELECT statement, the one enclosed in parentheses, is called a subquery. We can use multiple subqueries to process three or even more tables. For example, suppose we want to know the name of the buyers who manage any product purchased in January 2011. First, note that Buyer data is stored in the SKU_DATA table and OrderMonth and OrderYear data are stored in the RETAIL_ORDER table.

9、Querying Two or More Tables with SQL



❖ **Querying Multiple Tables with Subqueries**

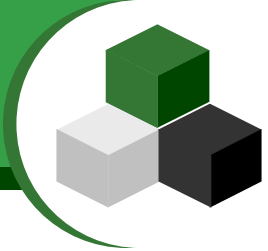
Now, we can use an SQL query with two subqueries to obtain the desired data as follows:

```
/* *** SQL-Query-CH02-49 *** */
SELECT        Buyer
FROM  SKU_DATA
WHERE         SKU  IN
(SELECT       SKU
FROM  ORDER_ITEM
WHERE         OrderNumber  IN
(SELECT       OrderNumber
FROM  RETAIL_ORDER
WHERE         OrderMonth='January'
AND   OrderYear=2011));
```

The result of this statement is:

	Buyer
1	Pete Hansen
2	Nancy Meyers
3	Nancy Meyers

9、Querying Two or More Tables with SQL

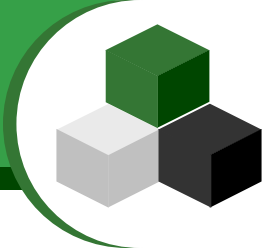


- **Querying Multiple Tables with Subqueries**

  To understand this statement, work from the bottom up. The bottom SELECT statement obtains the list of OrderNumbers of orders sold in January 2011. The middle SELECT statement obtains the SKU values for items sold in orders in January 2011. Finally, the top-level SELECT query obtains Buyer for all of the SKUs found in the middle SELECT statement.

  Any parts of the SQL language that you have learned earlier in this chapter can be applied to a table generated by a subquery, regardless of how complicated the SQL looks. For example, we can apply the DISTINCT keyword on the results to eliminate duplicate rows. Or, we can apply the GROUP BY and ORDER BY clauses as follows:

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Subqueries**

```
/* *** SQL-Query-CH02-50 *** */
SELECT      Buyer, COUNT(*) AS NumberSold
FROM SKU_DATA
WHERE       SKU IN
(SELECT     SKU
FROM ORDER_ITEM
WHERE       OrderNumber IN
(SELECT     OrderNumber
FROM RETAIL_ORDER
WHERE       OrderMonth='January'
AND   OrderYear=2011))
GROUP BY    Buyer
ORDER BY    NumberSold DESC;
```

The result is:

	Buyer	NumberSold
1	Nancy Meyers	2
2	Pete Hansen	1

9、Querying Two or More Tables with SQL

Does Not Work With MS Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason previously described on page 70. **Solution:** See the solution described in the "Does Not Work with Microsoft Access ANSI-89 SQL" box on page 70. The correct Microsoft Access ANSI-89 SQL statement for this query is:

```
/* *** SQL-Query-CH02-50-Access *** */
SELECT Buyer, Count(*) AS NumberSold
FROM SKU_DATA
WHERE SKU IN
(SELECT SKU
FROM ORDER_ITEM
WHERE OrderNumber IN
(SELECT OrderNumber
FROM RETAIL_ORDER
WHERE OrderMonth='January'
AND OrderYear=2011))
GROUP BY Buyer
ORDER BY Count(*) DESC;
```

9、Querying Two or More Tables with SQL

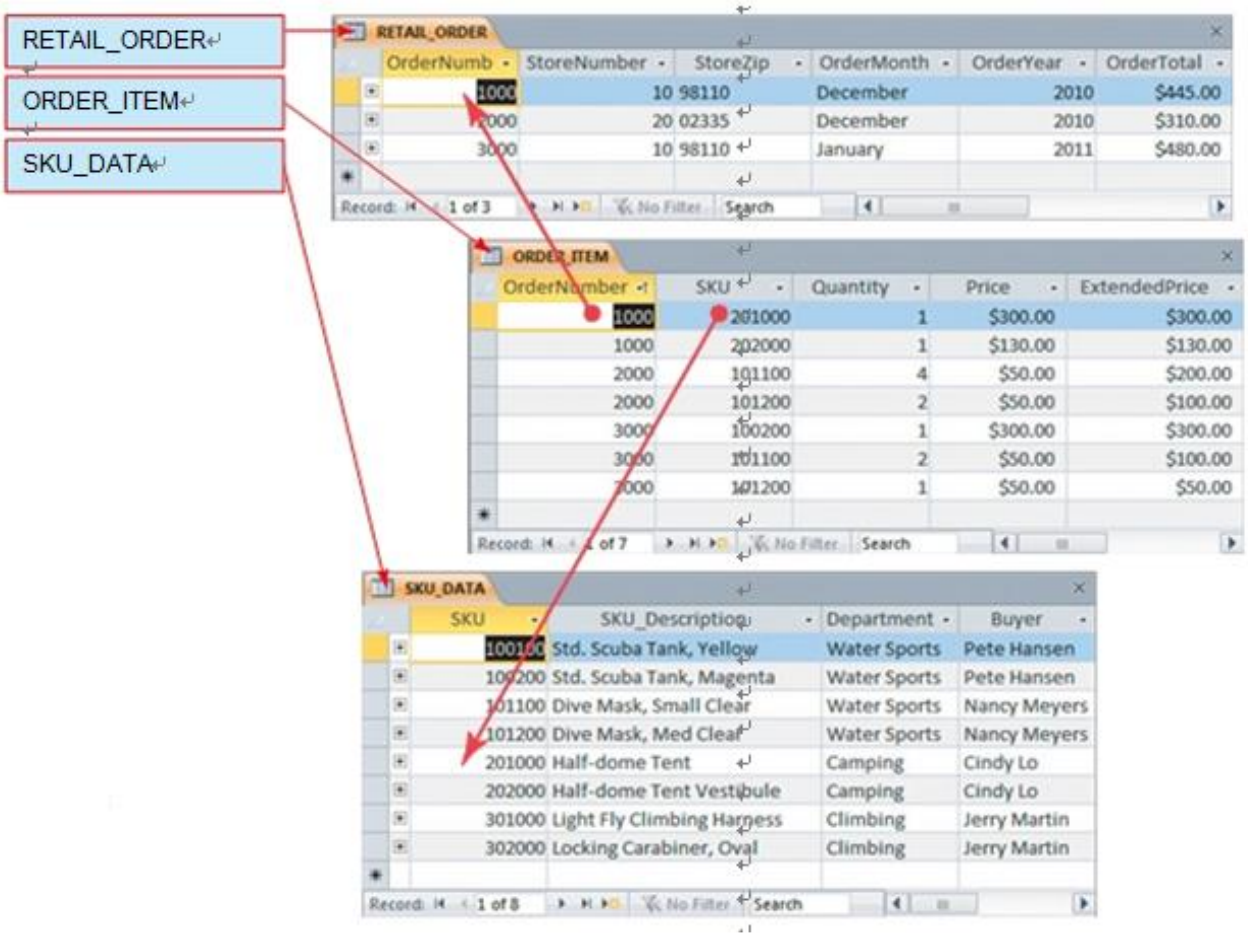
➢ Querying Multiple Tables with Joins

Subqueries a d data
can only com a that
arise from m

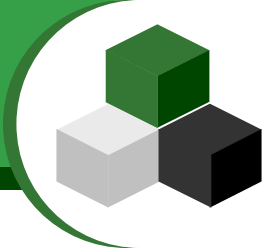
The SQ nating
(sticking toge r how
we might cor . We
can concaten
following SQ

```
/* *** S
SELECT
FROM RE
```

This statemer the
second table.



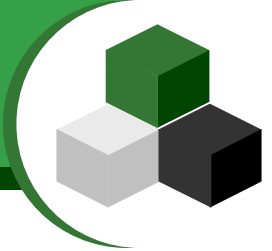
9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

	OrderNumber	StoreNumber	StoreZip	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2010	445.00	3000	100200	1	300.00	300.00
2	1000	10	98110	December	2010	445.00	2000	101100	4	50.00	200.00
3	1000	10	98110	December	2010	445.00	3000	101100	2	50.00	100.00
4	1000	10	98110	December	2010	445.00	2000	101200	2	50.00	100.00
5	1000	10	98110	December	2010	445.00	3000	101200	1	50.00	50.00
6	1000	10	98110	December	2010	445.00	1000	201000	1	300.00	300.00
7	1000	10	98110	December	2010	445.00	1000	202000	1	130.00	130.00
8	2000	20	02335	December	2010	310.00	3000	100200	1	300.00	300.00
9	2000	20	02335	December	2010	310.00	2000	101100	4	50.00	200.00
10	2000	20	02335	December	2010	310.00	3000	101100	2	50.00	100.00
11	2000	20	02335	December	2010	310.00	2000	101200	2	50.00	100.00
12	2000	20	02335	December	2010	310.00	3000	101200	1	50.00	50.00
13	2000	20	02335	December	2010	310.00	1000	201000	1	300.00	300.00
14	2000	20	02335	December	2010	310.00	1000	202000	1	130.00	130.00
15	3000	10	98110	January	2011	480.00	3000	100200	1	300.00	300.00
16	3000	10	98110	January	2011	480.00	2000	101100	4	50.00	200.00
17	3000	10	98110	January	2011	480.00	3000	101100	2	50.00	100.00
18	3000	10	98110	January	2011	480.00	2000	101200	2	50.00	100.00
19	3000	10	98110	January	2011	480.00	3000	101200	1	50.00	50.00
20	3000	10	98110	January	2011	480.00	1000	201000	1	300.00	300.00
21	3000	10	98110	January	2011	480.00	1000	202000	1	130.00	130.00

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

Because there are 3 rows of retail order and 7 rows of order items, there are 3 times 7, or 21, rows in this table. Notice that the retail order with OrderNumber 1000 has been combined with all seven of the rows in ORDER_ITEM, the retail order with OrderNumber 2000 has been combined with all seven of the same rows, and, finally, that the retail order with OrderNumber 3000 has again been combined with all seven rows.

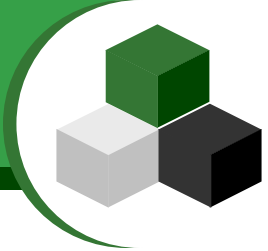
This is illogical—what we need to do is to select only those rows for which the OrderNumber of RETAIL_ORDER matches the OrderNumber in ORDER_ITEM. This is easy to do; we simply add an SQL WHERE clause to the query:

```
/* *** SQL-Query-CH02-52 *** */
SELE
FRO
WHE
mbe
```

	OrderNumber	StoreNumber	StoreZip	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	3000	10	98110	January	2011	480.00	3000	100200	1	300.00	300.00
2	2000	20	02335	December	2010	310.00	2000	101100	4	50.00	200.00
3	3000	10	98110	January	2011	480.00	3000	101100	2	50.00	100.00
4	2000	20	02335	December	2010	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2011	480.00	3000	101200	1	50.00	50.00
6	1000	10	98110	December	2010	445.00	1000	201000	1	300.00	300.00
7	1000	10	98110	December	2010	445.00	1000	202000	1	130.00	130.00

The result is:

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

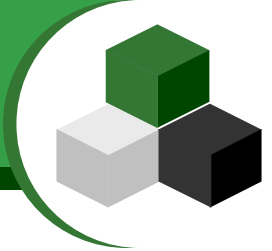
This is technically correct, but it will be easier to read if we sort the results using an ORDER BYclause:

```
/* *** SQL-Query-CH02-53 *** */
SELECT        *
FROM  RETAIL_ORDER,  ORDER_ITEM
WHERE
        RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNu
mber
ORDER  BY    RETAIL_ORDER.OrderNumber,
ORDER_ITEM.SKU;
```

The result is

	OrderNumber	StoreNumber	StoreZip	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2010	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2010	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2010	310.00	2000	101100	4	50.00	200.00
4	2000	20	02335	December	2010	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2011	480.00	3000	100200	1	300.00	300.00
6	3000	10	98110	January	2011	480.00	3000	101100	2	50.00	100.00
7	3000	10	98110	January	2011	480.00	3000	101200	1	50.00	50.00

9、Querying Two or More Tables with SQL

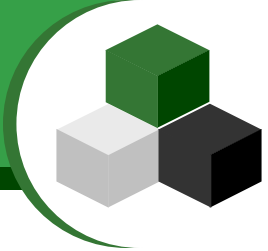


- **Querying Multiple Tables with Joins**

If you compare this result with the data in Figure 2-5, you will see that only the appropriate order items are associated with each retail order. You also can tell that this has been done by noticing that in each row the value of OrderNumber from RETAIL_ORDER (the first column) equals the value of OrderNumber from ORDER_ITEM (the seventh column). This was not true for our first result.

You may have noticed that we introduced a new variation in SQL statement syntax in the previous two queries, where the terms RETAIL_ORDER.OrderNumber, ORDER_ITEM.OrderNumber, and ORDER_ITEM.SKU were used. The new syntax is simply TableName.ColumnName, andit is used to specify exactly which table each column is linked to. RETAIL_ORDER.OrderNumber simply means the OrderNumber from the RETAIL_ORDER table. Similarly, ORDER_ITEM.Order-Number refers to the OrderNumber in the ORDER_ITEM table, and ORDER_ITEM.SKU refers to the SKU column in the ORDER_ITEM table. You can always qualify a column name with the name of its table like this. We have not done so previously because we were working with only one table, but the SQL statements shown previously would have worked just as well with syntax like SKU_DATA.Buyer rather than just Buyer or ORDER_ITEM.Price instead of Price.

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

The table that is formed by concatenating two tables is called a join. The process of creating such a table is called joining the two tables, and the associated operation is called a join operation. When the tables are joined using an equal condition (like the one on OrderNumber), this join is called an equijoin. When people say join, 99.99999 percent of the time they mean an equijoin. This type of join is also referred to as an inner join.

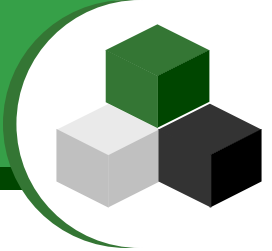
We can use a join to obtain data from two or more tables. For example, using the data in Figure 2-4, suppose we want to show the name of the Buyer and the ExtendedPrice of the sales of all items managed by that Buyer. The following SQL query will obtain that result:

```
/* *** SQL-Q                                    */
SELECT                                          e
FROM  SKU_DA
WHERE                                           ER_ITEM.SKU;
```

The result is:

	Buyer	ExtendedPrice
1	Pete Hansen	300.00
2	Nancy Meyers	200.00
3	Nancy Meyers	100.00
4	Nancy Meyers	100.00
5	Nancy Meyers	50.00
6	Cindy Lo	300.00
7	Cindy Lo	130.00

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

Again, the result of every SQL statement is just a single table, so we can apply any of the SQL syntax you learned for a single table to this result. For example, we can use the GROUP BY and ORDER BY clauses to obtain the total revenue associated with each buyer, as shown in the following SQL query:

```
/* *** SQL-Query-CH02-55 *** */
SELECT      Buyer, SUM(ExtendedPrice) AS BuyerRevenue
FROM SKU_DATA, ORDER_ITEM
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
GROUP BY    Buyer
ORDER BY    BuyerRevenue DESC;
```

The result is:

	Buyer	BuyerRevenue
1	Nancy Meyers	450.00
2	Cindy Lo	430.00
3	Pete Hansen	300.00

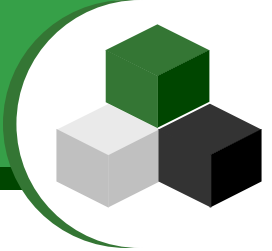
9、Querying Two or More Tables with SQL

Does Not Work With MS Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason previously described on page 70. Solution: See the solution described in the “Does Not Work with Microsoft Access ANSI-89 SQL” box on page 70. The correct Microsoft Access ANSI-89 SQL statement for this query is:

```
/* *** SQL-Query-CH02-55-Access *** */
SELECT Buyer, Sum(ORDER_ITEM.ExtendedPrice) AS
BuyerRevenue
FROM   SKU_DATA, ORDER_ITEM
WHERE  SKU_DATA.SKU=ORDER_ITEM.SKU
GROUP BY   Buyer
ORDER BY   Sum(ExtendedPrice) DESC;
```

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

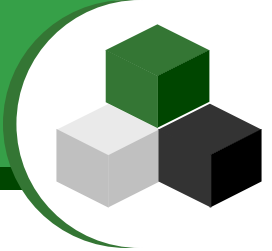
We can extend this syntax to join three or more tables. For example, suppose we want to obtain the Buyer and the ExtendedPrice and OrderMonth for all purchases of items managed by each buyer. To retrieve that data, we need to join all three tables together, as shown in this SQL query:

```
/* *** SQL-Query-CH02-56 *** */
SELECT      Buyer, ExtendedPrice, OrderMonth
FROM SKU_DATA, ORDER_ITEM, RETAIL_ORDER
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
AND
        ORDER_I                    ._ORDER.OrderNu
mber;
```

The result is:

	Buyer	ExtendedPrice	OrderMonth
1	Pete Hansen	300.00	January
2	Nancy Meyers	200.00	December
3	Nancy Meyers	100.00	January
4	Nancy Meyers	100.00	December
5	Nancy Meyers	50.00	January
6	Cindy Lo	300.00	December
7	Cindy Lo	130.00	December

9、Querying Two or More Tables with SQL



- **Querying Multiple Tables with Joins**

We can improve this result by sorting with the ORDER BY clause and grouping by Buyer with the GROUP BY clause:

```
/* *** SQL-Query-CH02-57 *** */
SELECT Buyer, OrderMonth, SUM(ExtendedPrice) AS BuyerRevenue
FROM SKU_DATA, ORDER_ITEM, RETAIL_ORDER
WHERE SKU_DATA.SKU=ORDER_ITEM.SKU
AND
ORDER_ITEM.OrderNumber=RETAIL_ORDER.OrderNumber
GROUP BY Buy
ORDER BY Buy
```

The result is:

	Buyer	OrderMonth	BuyerRevenue
1	Cindy Lo	December	430.00
2	Nancy Meyers	December	300.00
3	Nancy Meyers	January	150.00
4	Pete Hansen	January	300.00

9、Querying Two or More Tables with SQL

Joins also can be written using another syntax, the SQL JOIN . . . ON syntax, and there is a bit more for you to learn about joins when values are missing, but this chapter is long enough. We will finish the discussion of joins in Chapter 7. If you just cannot wait, turn to pages 272-277 for the rest of the join story.

- **Comparing Subqueries and Joins**

Subqueries and joins both process multiple tables, but they differ slightly. As mentioned earlier, a subquery can only be used to retrieve data from the top table. A join can be used to obtain data from any number of tables. Thus, a join can do everything a subquery can do, and more. So why learn subqueries? For one, if you just need data from a single table, you might use a subquery because it is easier to write and understand. This is especially true when processing multiple tables.

In Chapter 8, however, you will learn about a type of subquery called a correlated subquery. A correlated subquery can do work that is not possible with joins. Thus, it is important for you to learn about both joins and subqueries, even though right now it appears that joins are uniformly superior. If you’re curious, ambitious, and courageous, jump ahead and read the discussion of correlated subqueries on pages 315-320.

summary

Structured Query Language (SQL) was developed by IBM and has been endorsed by the ANSI SQL-92 and following standards. SQL is a data sublanguage that can be embedded into full programming languages or submitted directly to the DBMS. Knowing SQL is critical for knowledge workers, application programmers, and database administrators.

All DBMS products process SQL. Microsoft Access hides SQL, but SQL Server, Oracle Database, and MySQL require that you use it We are primarily interested in three categories of SQL statements: DML, DDL, and SQL/PSM statements. DML statements include statements for querying data and for inserting, updating, and deleting data. This chapter addresses only DML query statements. Additional DML statements, DDL and SQL/PSM are discussed in Chapter 7.

The examples in this chapter are based on three tables extracted from the operational database at Cape Codd Outdoor Sports. Such database extracts are common and important. Sample data for the three tables is shown in Figure 2-4.

The basic structure of an SQL query statement is SELECT/FROM/WHERE. The columns to be selected are listed after SELECT, the table(s) to process is listed after FROM, and any restrictions on data values are listed after WHERE. In a WHERE clause, character and date data valuesmust be enclosed in single quotes. Numeric data need not be enclosed in quotes. You can submit SQL statements directly to Microsoft Access, SQL Server, Oracle Database, and MySQL, as described in this chapter.

summary

This chapter explained the use of the following SQL clauses: SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING. This chapter explained the use of the following SQL keywords: DISTINCT, DESC, ASC, AND, OR, IN, NOT IN, BETWEEN, LIKE, % (* for Microsoft Access), _ (? for Microsoft Access), SUM, AVG, MIN, MAX, COUNT, AS. You should know how to mix and match these features to obtain the results you want. By default, the WHERE clause is applied before the HAVING clause.

You can query multiple tables using subqueries and joins. Subqueries are nested queries that use the SQL keywords IN and NOT IN. An SQL SELECT expression is placed inside parentheses. Using a subquery, you can display data from the top table only. A join is created by specifying multiple table names in the FROM clause. An SQL WHERE clause is used to obtain an equijoin. In most cases, equijoins are the most sensible option. Joins can display data from multiple tables. In Chapter 8, you will learn another type of subquery that can perform work that is not possible with joins.

LOGO

Thank You!