```
CREATE TABLE state
(
  state_code char (2),
  state_name char (15),
  CONSTRAINT PK_state PRIMARY KEY (state_code),
  CONSTRAINT RelationstateState_tax_rates1 FOREIGN KEY ()
      REFERENCES state_tax_rates
);

CREATE TABLE warehouse
(
  warehouse_id integer,
  warehouse_street char (50),
  warehouse_city char (3),
  wharehouse_state_code char (2),
  warehouse_zip char (10),
  warehouse_phone char (12),
  warehouse_manager_id integer,
  CONSTRAINT PK_warehouse PRIMARY KEY (warehouse_id)
);

CREATE TABLE state_tax_rates
(
  state_code char (2),
  sales_tax_rate number (5,20),
  CONSTRAINT PK_state_tax_rates PRIMARY KEY (state_code),
  CONSTRAINT RelationstateState_tax_rates1 FOREIGN KEY ()
      REFERENCES state
);

CREATE TABLE product
(
  UPC char (15),
  product_name varchar (256),
  product_unit char (10),
  shipping_weight integer,
  product_image blob,
  web_orderable boolean,
  CONSTRAINT PK_product PRIMARY KEY (UPC)
);

CREATE TABLE product_taxed
(
  UPC integer,
  state_code integer,
  CONSTRAINT PK_product_taxed PRIMARY KEY (UPC,state_code),
  CONSTRAINT RelationProductProduct_taxed1 FOREIGN KEY ()
      REFERENCES product,
  CONSTRAINT RelationState_tax_ratesProduct_taxed1 FOREIGN KEY ()
```

```
      REFERENCES state_tax_rates
);
```

```
CREATE TABLE In_store_payment_type
(
  in_store_payment_type_code integer,
  in_store_payment_type_text char (10),
  CONSTRAINT PK_In_store_payment_type PRIMARY KEY
      (in_store_payment_type_code)
);

CREATE TABLE web_customer
(
  web_customer_numb integer,
  web_customer_first_name char (15),
  web_customer_last_name INTEGER,
  web_customer_phone char (12),
  web_customer_password char (15),
  web_customer_user_id char (15),
  CONSTRAINT PK_web_customer PRIMARY KEY (web_customer_number)
);

CREATE TABLE web_sale_shipping_address
(
  web_shipping_address_numb integer,
  web_customer_numb integer,
  web_shipping_street char (50),
  web_customer_shipping_city char (50),
  web_customer_shipping_state_code char (2),
  eb_cutomer_shipping_zip char (10),
  web_customer_shipping_first_name char (15),
  web_customer_shipping_last_name char (15),
  CONSTRAINT PK_web_sale_shipping_address
      PRIMARY KEY (web_shipping_address_numb),
  CONSTRAINT RelationWeb_customerWeb_sale_shipping_address1
      FOREIGN KEY () REFERENCES web_customer,
  CONSTRAINT RelationstateWeb_sale_shipping_address1
      FOREIGN KEY () REFERENCES state
);

CREATE TABLE web_customer_credit_card
(
  web_customer_numb integer,
  web_credit_card_numb char (16),
  web_credit_card_exp_date date,
  web_sale_name_on_credit_card varchar (50),
  CONSTRAINT PK_web_customer_credit_card
      PRIMARY KEY (web_customer_numb,web_credit_card_numb),
```

```
  CONSTRAINT RelationWeb_customerweb_customer_credit_card1
      FOREIGN KEY () REFERENCES web_customer
);
```

```
CREATE TABLE web_sale
(
  web_sale_numb integer,
  web_sale_date date,
  web_customer_numb integer,
  web_sale_total number (7,2),
  web_sale_merchandise_total number (6,2),
  web_sale_shipping number (6,2),
  web_sale_tax number (6,2),
  CONSTRAINT PK_web_sale PRIMARY KEY (web_sale_numb),
  CONSTRAINT RelationWeb_customerWeb_sale1 FOREIGN KEY ()
      REFERENCES web_customer,
  CONSTRAINT Relationweb_customer_credit_cardWeb_sale1
      FOREIGN KEY () REFERENCES web_customer_credit_card
);

CREATE TABLE web_sale_shipment
(
  web_sale_numb inter,
  web_sale_date_shipped date,
  web_shipment_merchandise_total number(7,2),
  web_shipment_tax number (6,2),
  web_shipment_shipping number (6,2),
  web_shipment_total_charged number (6,2),
  CONSTRAINT PK_web_sale_shipment PRIMARY KEY (web_sale_numb),
  CONSTRAINT RelationWeb_saleWeb_sale_shipment1
      FOREIGN KEY () REFERENCES web_sale
);

CREATE TABLE pay_type
(
  pay_type_code integer,
  pay_type_description varhar (10),
  CONSTRAINT PK_pay_type PRIMARY KEY (pay_type_code)
);
```

```
CREATE TABLE employee
(
  employee_id integer,
  employee_first_name varchar (15),
  employee_last_name integer,
  employee_street varchar (50),
  employee_city varchar (50),
  employee_state_code char (2),
  employee_zip char (10),
  employee_phone char (12)
```

```
  employee_phone char (12),
  employee_ssn char (11),
  employee_birth_date date,
  dept_id integer,
  store_numb integer,
  supervisor_id integer,
  pay_type_code integer,
  pay_rate number (10,2),
  CONSTRAINT PK_employee PRIMARY KEY (employee_id),
  CONSTRAINT RelationEmployeeDepartment1
      FOREIGN KEY () REFERENCES department,
  CONSTRAINT Relationpay_typeEmployee1
      FOREIGN KEY () REFERENCES pay_type,
  CONSTRAINT RelationEmployeestate1
      FOREIGN KEY () REFERENCES state
);

CREATE TABLE department
(
  dept_numb integer,
  store_numnb integer,
  dept_manager_id integer,
  CONSTRAINT PK_department PRIMARY KEY (dept_numb,store_numnb),
  CONSTRAINT RelationStoreDepartment1 FOREIGN KEY () REFERENCES Store
);

CREATE TABLE store
(
  store_numb integer,
  store_street char (50),
  store_city char (50),
  store_state_code char (2),
  store_zip char (10),
  store_main_phone char (12),
  store_manager_id integer,
  CONSTRAINT PK_Store PRIMARY KEY (store_numb)
);
```

```
CREATE TABLE work_schedule
(
  employee_numb integer,
  work_date date,
  work_start_time time,
  work_hours_scheduled integer,
  CONSTRAINT PK_work_schedule PRIMARY KEY (employee_numb,work_date),
  CONSTRAINT RelationEmployeeWork_schedule2
      FOREIGN KEY () REFERENCES employee
);

CREATE TABLE credit_sale_details
(
  in_store_transaction_numb integer,
  in_store_sale_credit_card_numb char (16),
  in_store_exp_date date
```

```
  in_store_exp_date date,
  in_store_sale_name_on_credit_card varchar (50),
  CONSTRAINT PK_credit_sale_details
      PRIMARY KEY (in_store_transaction_numb)
);

CREATE TABLE promotion_type
(
  promotion_type_code integer,
  promotion_type_name INTEGER,
  CONSTRAINT PK_Promotion_type PRIMARY KEY (promotion_type_code)
);

CREATE TABLE keyword_list
(
  keyword__code integer,
  keyword_text varchar (50),
  CONSTRAINT PK_keyword_list PRIMARY KEY (keyword__code)
);

CREATE TABLE product_keyword
(
  UPC char (15),
  keyword_code integer,
  CONSTRAINT PK_product_keyword PRIMARY KEY (UPC),
  CONSTRAINT RelationKeyword_listProduct_keyword1
      FOREIGN KEY () REFERENCES keyword_list
);
```

```
CREATE TABLE sales_promotion
(
  UPC char (15),
  promotion_start_date date,
  promotion_end_date date,
  promotion_type_code integer,
  promotion_details char (50),
  CONSTRAINT PK_sales_promotion PRIMARY KEY (UPC,promotion_start_date),
  CONSTRAINT RelationProductSales_promotion1
      FOREIGN KEY () REFERENCES product
);

CREATE TABLE In_store_sale
(
  in_store_transaction_numb integer,
  in_store_transaction_date date,
  in_store_sale_product_total number (8,2),
  in_store_sales_tax number (6,2),
  in_store_sale_total number (8,2),
  in_store_sale_payment_type_code integer,
  CONSTRAINT PK_In_store_sale PRIMARY KEY (in_store_transaction_numb),
  CONSTRAINT RelationIn_store_salecredit_sale_details1
      FOREIGN KEY () REFERENCES credit_sale_details,
  CONSTRAINT RelationIn_store_payment_typeIn_store_sale1
      FOREIGN KEY () REFERENCES In_store_payment_type
);
```

```
);

CREATE TABLE product_stocked
(
  SKU char (15),
  UPC char (15),
  in_store boolean,
  store_ numb integer,
  dept_numb integer,
  warehouse_numb integer,
  size char (10),
  color char (15),
  number_on_hand integer,
  retail_price number (7,2),
  CONSTRAINT PK_product_stocked PRIMARY KEY (SKU),
  CONSTRAINT RelationProductProduct_stocked1
      FOREIGN KEY () REFERENCES product,
  CONSTRAINT RelationDepartmentProduct_stocked1
      FOREIGN KEY () REFERENCES department
);
```

```
CREATE TABLE In_store_sale_item
(
  in_store_transaction_numb integer,
  in_store_SKU integer,
  in_store_quantity integer,
  in_store_price number (7,2)0,
  CONSTRAINT PK_In_store_sale_item
      PRIMARY KEY (in_store_transaction_numb,in_store_SKU),
  CONSTRAINT RelationProduct_stockedIn_store_sale_item1
      FOREIGN KEY () REFERENCES product_stocked,
  CONSTRAINT RelationIn_store_saleIn_store_sale_item1
      FOREIGN KEY () REFERENCES In_store_sale
);

CREATE TABLE web_sale_item
(
  web_sale_numb integer,
  web_sale_SKU char (15),
  web_sale_quantity integer,
  web_sale_shipped boolean,
  web_sale_price number (6,2),
  web_shipping_address_numb integer,
  CONSTRAINT PK_web_sale_item PRIMARY KEY (web_sale_numb,web_sale_SKU),
  CONSTRAINT RelationProduct_stockedWeb_sale_item1
      FOREIGN KEY () REFERENCES product_stocked,
  CONSTRAINT RelationWeb_saleWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale,
  CONSTRAINT RelationWeb_sale_shipmentWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale_shipment,
  CONSTRAINT RelationWeb_sale_shipping_addressWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale_shipping_address
);

CREATE TABLE web_security_question
(
  web_security_question_numb integer,
  web_security_question_text varchar (256)
```

```
  web_security_question_text varchar (256),
  CONSTRAINT PK_web_security_question PRIMARY KEY
      (web_security_question_numb)
);

CREATE TABLE web_security_question_answer
(
  web_security_question_numb integer,
  web_customer_numb integer,
  web_security_question_answer_text varchar (256),
  CONSTRAINT PK_web_security_question_answer PRIMARY KEY
      (web_security_question_numb,web_customer_numb)
  CONSTRAINT RelationWeb_security_questionWeb_security_question_answer1
      FOREIGN KEY () REFERENCES web_security_question
  CONSTRAINT RelationWeb_customerWeb_security_question_answer1
      FOREIGN KEY () REFERENCES web_customer
);
```

FIGURE 15.4 **SQL CREATE statements for the SmartMart database.**

```
CREATE TABLE state
(
  state_code char (2),
  state_name char (15),
  CONSTRAINT PK_state PRIMARY KEY (state_code),
  CONSTRAINT RelationstateState_tax_rates1 FOREIGN KEY ()
      REFERENCES state_tax_rates
);

CREATE TABLE warehouse
(
  warehouse_id integer,
  warehouse_street char (50),
  warehouse_city char (3),
  wharehouse_state_code char (2),
  warehouse_zip char (10),
  warehouse_phone char (12),
  warehouse_manager_id integer,
  CONSTRAINT PK_warehouse PRIMARY KEY (warehouse_id)
);

CREATE TABLE state_tax_rates
(
  state_code char (2),
  sales_tax_rate number (5,20),
  CONSTRAINT PK_state_tax_rates PRIMARY KEY (state_code),
  CONSTRAINT RelationstateState_tax_rates1 FOREIGN KEY ()
      REFERENCES state
);

CREATE TABLE product
(
  UPC char (15),
  product_name varchar (256),
  product_unit char (10),
  shipping_weight integer,
  product_image blob,
  web_orderable boolean,
  CONSTRAINT PK_product PRIMARY KEY (UPC)
);

CREATE TABLE product_taxed
(
  UPC integer,
  state_code integer,
  CONSTRAINT PK_product_taxed PRIMARY KEY (UPC,state_code),
  CONSTRAINT RelationProductProduct_taxed1 FOREIGN KEY ()
      REFERENCES product,
  CONSTRAINT RelationState_tax_ratesProduct_taxed1 FOREIGN KEY ()
```

```
      REFERENCES state_tax_rates
);
```

```
CREATE TABLE In_store_payment_type
(
  in_store_payment_type_code integer,
  in_store_payment_type_text char (10),
  CONSTRAINT PK_In_store_payment_type PRIMARY KEY
      (in_store_payment_type_code)
);

CREATE TABLE web_customer
(
  web_customer_numb integer,
  web_customer_first_name char (15),
  web_customer_last_name INTEGER,
  web_customer_phone char (12),
  web_customer_password char (15),
  web_customer_user_id char (15),
  CONSTRAINT PK_web_customer PRIMARY KEY (web_customer_number)
);

CREATE TABLE web_sale_shipping_address
(
  web_shipping_address_numb integer,
  web_customer_numb integer,
  web_shipping_street char (50),
  web_customer_shipping_city char (50),
  web_customer_shipping_state_code char (2),
  eb_cutomer_shipping_zip char (10),
  web_customer_shipping_first_name char (15),
  web_customer_shipping_last_name char (15),
  CONSTRAINT PK_web_sale_shipping_address
      PRIMARY KEY (web_shipping_address_numb),
  CONSTRAINT RelationWeb_customerWeb_sale_shipping_address1
      FOREIGN KEY () REFERENCES web_customer,
  CONSTRAINT RelationstateWeb_sale_shipping_address1
      FOREIGN KEY () REFERENCES state
);

CREATE TABLE web_customer_credit_card
(
  web_customer_numb integer,
  web_credit_card_numb char (16),
  web_credit_card_exp_date date,
  web_sale_name_on_credit_card varchar (50),
  CONSTRAINT PK_web_customer_credit_card
      PRIMARY KEY (web_customer_numb,web_credit_card_numb),
```

```
  CONSTRAINT RelationWeb_customerweb_customer_credit_card1
      FOREIGN KEY () REFERENCES web_customer
);
```

```
CREATE TABLE web_sale
(
  web_sale_numb integer,
  web_sale_date date,
  web_customer_numb integer,
  web_sale_total number (7,2),
  web_sale_merchandise_total number (6,2),
  web_sale_shipping number (6,2),
  web_sale_tax number (6,2),
  CONSTRAINT PK_web_sale PRIMARY KEY (web_sale_numb),
  CONSTRAINT RelationWeb_customerWeb_sale1 FOREIGN KEY ()
      REFERENCES web_customer,
  CONSTRAINT Relationweb_customer_credit_cardWeb_sale1
      FOREIGN KEY () REFERENCES web_customer_credit_card
);

CREATE TABLE web_sale_shipment
(
  web_sale_numb inter,
  web_sale_date_shipped date,
  web_shipment_merchandise_total number(7,2),
  web_shipment_tax number (6,2),
  web_shipment_shipping number (6,2),
  web_shipment_total_charged number (6,2),
  CONSTRAINT PK_web_sale_shipment PRIMARY KEY (web_sale_numb),
  CONSTRAINT RelationWeb_saleWeb_sale_shipment1
      FOREIGN KEY () REFERENCES web_sale
);

CREATE TABLE pay_type
(
  pay_type_code integer,
  pay_type_description varhar (10),
  CONSTRAINT PK_pay_type PRIMARY KEY (pay_type_code)
);
```

```
CREATE TABLE employee
(
  employee_id integer,
  employee_first_name varchar (15),
  employee_last_name integer,
  employee_street varchar (50),
  employee_city varchar (50),
  employee_state_code char (2),
  employee_zip char (10),
  employee_phone char (12)
```

```
  employee_phone char (12),
  employee_ssn char (11),
  employee_birth_date date,
  dept_id integer,
  store_numb integer,
  supervisor_id integer,
  pay_type_code integer,
  pay_rate number (10,2),
  CONSTRAINT PK_employee PRIMARY KEY (employee_id),
  CONSTRAINT RelationEmployeeDepartment1
      FOREIGN KEY () REFERENCES department,
  CONSTRAINT Relationpay_typeEmployee1
      FOREIGN KEY () REFERENCES pay_type,
  CONSTRAINT RelationEmployeestate1
      FOREIGN KEY () REFERENCES state
);

CREATE TABLE department
(
  dept_numb integer,
  store_numnb integer,
  dept_manager_id integer,
  CONSTRAINT PK_department PRIMARY KEY (dept_numb,store_numnb),
  CONSTRAINT RelationStoreDepartment1 FOREIGN KEY () REFERENCES Store
);

CREATE TABLE store
(
  store_numb integer,
  store_street char (50),
  store_city char (50),
  store_state_code char (2),
  store_zip char (10),
  store_main_phone char (12),
  store_manager_id integer,
  CONSTRAINT PK_Store PRIMARY KEY (store_numb)
);
```

```
CREATE TABLE work_schedule
(
  employee_numb integer,
  work_date date,
  work_start_time time,
  work_hours_scheduled integer,
  CONSTRAINT PK_work_schedule PRIMARY KEY (employee_numb,work_date),
  CONSTRAINT RelationEmployeeWork_schedule2
      FOREIGN KEY () REFERENCES employee
);

CREATE TABLE credit_sale_details
(
  in_store_transaction_numb integer,
  in_store_sale_credit_card_numb char (16),
  in_store_exp_date date
```

```
  in_store_exp_date date,
  in_store_sale_name_on_credit_card varchar (50),
  CONSTRAINT PK_credit_sale_details
      PRIMARY KEY (in_store_transaction_numb)
);

CREATE TABLE promotion_type
(
  promotion_type_code integer,
  promotion_type_name INTEGER,
  CONSTRAINT PK_Promotion_type PRIMARY KEY (promotion_type_code)
);

CREATE TABLE keyword_list
(
  keyword__code integer,
  keyword_text varchar (50),
  CONSTRAINT PK_keyword_list PRIMARY KEY (keyword__code)
);

CREATE TABLE product_keyword
(
  UPC char (15),
  keyword_code integer,
  CONSTRAINT PK_product_keyword PRIMARY KEY (UPC),
  CONSTRAINT RelationKeyword_listProduct_keyword1
      FOREIGN KEY () REFERENCES keyword_list
);
```

```
CREATE TABLE sales_promotion
(
  UPC char (15),
  promotion_start_date date,
  promotion_end_date date,
  promotion_type_code integer,
  promotion_details char (50),
  CONSTRAINT PK_sales_promotion PRIMARY KEY (UPC,promotion_start_date),
  CONSTRAINT RelationProductSales_promotion1
      FOREIGN KEY () REFERENCES product
);

CREATE TABLE In_store_sale
(
  in_store_transaction_numb integer,
  in_store_transaction_date date,
  in_store_sale_product_total number (8,2),
  in_store_sales_tax number (6,2),
  in_store_sale_total number (8,2),
  in_store_sale_payment_type_code integer,
  CONSTRAINT PK_In_store_sale PRIMARY KEY (in_store_transaction_numb),
  CONSTRAINT RelationIn_store_salecredit_sale_details1
      FOREIGN KEY () REFERENCES credit_sale_details,
  CONSTRAINT RelationIn_store_payment_typeIn_store_sale1
      FOREIGN KEY () REFERENCES In_store_payment_type
);
```

```
);

CREATE TABLE product_stocked
(
  SKU char (15),
  UPC char (15),
  in_store boolean,
  store_ numb integer,
  dept_numb integer,
  warehouse_numb integer,
  size char (10),
  color char (15),
  number_on_hand integer,
  retail_price number (7,2),
  CONSTRAINT PK_product_stocked PRIMARY KEY (SKU),
  CONSTRAINT RelationProductProduct_stocked1
      FOREIGN KEY () REFERENCES product,
  CONSTRAINT RelationDepartmentProduct_stocked1
      FOREIGN KEY () REFERENCES department
);
```

```
CREATE TABLE In_store_sale_item
(
  in_store_transaction_numb integer,
  in_store_SKU integer,
  in_store_quantity integer,
  in_store_price number (7,2)0,
  CONSTRAINT PK_In_store_sale_item
      PRIMARY KEY (in_store_transaction_numb,in_store_SKU),
  CONSTRAINT RelationProduct_stockedIn_store_sale_item1
      FOREIGN KEY () REFERENCES product_stocked,
  CONSTRAINT RelationIn_store_saleIn_store_sale_item1
      FOREIGN KEY () REFERENCES In_store_sale
);

CREATE TABLE web_sale_item
(
  web_sale_numb integer,
  web_sale_SKU char (15),
  web_sale_quantity integer,
  web_sale_shipped boolean,
  web_sale_price number (6,2),
  web_shipping_address_numb integer,
  CONSTRAINT PK_web_sale_item PRIMARY KEY (web_sale_numb,web_sale_SKU),
  CONSTRAINT RelationProduct_stockedWeb_sale_item1
      FOREIGN KEY () REFERENCES product_stocked,
  CONSTRAINT RelationWeb_saleWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale,
  CONSTRAINT RelationWeb_sale_shipmentWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale_shipment,
  CONSTRAINT RelationWeb_sale_shipping_addressWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale_shipping_address
);

CREATE TABLE web_security_question
(
  web_security_question_numb integer,
  web_security_question_text varchar (256)
```

```
  web_security_question_text varchar (256),
  CONSTRAINT PK_web_security_question PRIMARY KEY
      (web_security_question_numb)
);

CREATE TABLE web_security_question_answer
(
  web_security_question_numb integer,
  web_customer_numb integer,
  web_security_question_answer_text varchar (256),
  CONSTRAINT PK_web_security_question_answer PRIMARY KEY
      (web_security_question_numb,web_customer_numb)
  CONSTRAINT RelationWeb_security_questionWeb_security_question_answer1
      FOREIGN KEY () REFERENCES web_security_question
  CONSTRAINT RelationWeb_customerWeb_security_question_answer1
      FOREIGN KEY () REFERENCES web_customer
);
```

FIGURE 15.4 **SQL CREATE statements for the SmartMart database.**

Introduction

This part of the book goes in depth into the uses and syntax of interactive SQL. You will learn to create a wide variety of queries, as well as perform data modification. In this part, you will read about the impact that equivalent syntaxes may have on the performance of your queries. Finally, you will learn about creating additional database structural elements that require querying in their syntax.

CHAPTER 16

Simple SQL Retrieval

Abstract

This chapter begins a sequence of chapters that focus on SQL retrieval statements. This chapter covers specifying columns for display, and choosing rows from one table at a time. It introduces the construction of SQL predicates using relationship operators, logical operators, and SQL's special operators.

Keywords

SQL
SQL SELECT
SQL WHERE
SQL predicates
SQL operators

As we've said several times earlier, retrieval is what databases are all about: We put the data in so we can get them out in some meaningful way. There are many ways to formulate a request for data retrieval, but when it comes to relational databases, SQL is the international standard. It may not seem very "modern" because it's a text-based language, but it's the best we have when it comes to something that can be used in many environments.

Note: Yes, SQL is a standard and, in theory, implementations should be the same. However, although the basic syntax is the same, the "devil is in the details," as the old saying goes. For example, some implementations require a semicolon to terminate a statement, but others do not. You will therefore find notes throughout the SQL chapters warning you about areas where implementations may differ. The only solution is to use product documentation to verify the exact syntax used by your DBMS.

SQL has one command for retrieving data: SELECT. This is nowhere as restrictive as it might seem. SELECT contains syntax for choosing columns, choosing rows, combining tables, grouping data, and performing some simple calculations. In fact, a single SELECT statement can result in a DBMS performing any or all of the relational algebra operations.

The basic syntax of the SELECT statement has the following general structure:

```
SELECT column1, column2 …
FROM table1, table2 …
WHERE predicate
```

The SELECT clause specifies the columns you want to see. You specify the tables used in the query in the FROM clause. The optional WHERE clause can contain a wide variety of criteria that identify which rows you want to retrieve.

Note: Most SQL command processors are not case sensitive when it comes to parts of a SQL statement. SQL keywords, table names, column names, and so on can be in any case you choose. However, most DBMSs are case sensitive when it comes to matching data values. Therefore, whenever you place a value in quotes for SQL to match, you must match the case of the stored data. In this book, SQL keywords will appear in uppercase letters; database components such as column and table names will appear in lowercase letters.

In addition to these basic clauses, SELECT has many other syntax options. Rather than attempt to summarize them all in a single general statement, you will learn to build the parts of a SELECT gradually throughout this and the next few chapters of this book.

Note: The SQL queries you see throughout the book are terminated by a semicolon (;). This is not part of the SQL standard, but is used by many DBMSs so that you can type a command on multiple lines. The SQL command processor doesn't execute the query until it encounters the semicolon.

Note: The examples in this chapter and the remaining SQL retrieval chapters use the rare book store database introduced in Chapter 6.

Revisiting the Sample Data

In Chapter 6 you were introduced to the rare book store database and the sample data with which the tables have been populated. The sample data

In Chapter 6 you were introduced to the rare book store database and the sample data with which the tables have been populated. The sample data are repeated here for easier reference when reading Chapters 16–19.

Table 6.1
Publisher

publisher_id	publisher_name
1	Wiley
2	Simon & Schuster
3	Macmillan
4	Tor
5	DAW

Table 6.2
Author

author_numb	author_last_first
1	Bronte, Charlotte
2	Doyle, Sir Arthur Conan
3	Twain, Mark
4	Stevenson, Robert Louis
5	Rand, Ayn
6	Barrie, James
7	Ludlum, Robert
8	Barth, John
9	Herbert, Frank
10	Asimov, Isaac
11	Funke, Cornelia
12	Stephenson, Neal

Table 6.3
Condition codes

condition_code	condition_description
1	New
2	Excellent
3	Fine
4	Good
5	Poor

Table 6.4

Work

work_numb	author_numb	title
1	1	Jane Eyre
2	1	Villette
3	2	Hound of the Baskervilles
4	2	Lost World, The
5	2	Complete Sherlock Holmes
7	3	Prince and the Pauper
8	3	Tom Sawyer
9	3	Adventures of Huckleberry Finn, The
6	3	Connecticut Yankee in King Arthur's Court, A
13	5	Fountainhead, The
14	5	Atlas Shrugged
15	6	Peter Pan
10	7	Bourne Identity, The
11	7	Matarese Circle, The
12	7	Bourne Supremacy, The
16	4	Kidnapped
17	4	Treasure Island
18	8	Sot Weed Factor, The
19	8	Lost in the Funhouse
20	8	Giles Goat Boy
21	9	Dune
22	9	Dune Messiah
23	10	Foundation
24	10	Last Foundation
25	10	I, Robot
26	11	Inkheart
27	11	Inkdeath
28	12	Anathem
29	12	Snow Crash
30	5	Anthem
31	12	Cryptonomicon

Table 6.5

Books

isbn	work_numb	publisher_id	edition	binding	copyright_year
978-1-11111-111-1	1	1	2	Board	1857
978-1-11111-112-1	1	1	1	Board	1847
978-1-11111-113-1	2	4	1	Board	1842
978-1-11111-114-1	3	4	1	Board	1801
978-1-11111-115-1	3	4	10	Leather	1925
978-1-11111-116-1	4	3	1	Board	1805
978-1-11111-117-1	5	5	1	Board	1808
978-1-11111-118-1	5	2	19	Leather	1956
978-1-11111-120-1	8	4	5	Board	1906

978-1-11111-119-1	6	2	3	Board	1956
978-1-11111-121-1	8	1	12	Leath	1982
978-1-11111-122-1	9	1	12	Leather	1982
978-1-11111-123-1	11	2	1	Board	1998
978-1-11111-124-1	12	2	1	Board	1989
978-1-11111-125-1	13	2	3	Board	1965
978-1-11111-126-1	13	2	9	Leath	2001
978-1-11111-127-1	14	2	1	Board	1960
978-1-11111-128-1	16	2	12	Board	1960
978-1-11111-129-1	16	2	14	Leather	2002
978-1-11111-130-1	17	3	6	Leather	1905
978-1-11111-131-1	18	4	6	Board	1957
978-1-11111-132-1	19	4	1	Board	1962
978-1-11111-133-1	20	4	1	Board	1964
978-1-11111-134-1	21	5	1	Board	1964
978-1-11111-135-1	23	5	1	Board	1962
978-1-11111-136-1	23	5	4	Leather	2001
978-1-11111-137-1	24	5	4	Leather	2001
978-1-11111-138-1	23	5	4	Leather	2001
978-1-11111-139-1	25	5	4	Leather	2001
978-1-11111-140-1	26	5	1	Board	2001
978-1-11111-141-1	27	5	1	Board	2005
978-1-11111-142-1	28	5	1	Board	2008
978-1-11111-143-1	29	5	1	Board	1992
978-1-11111-144-1	30	1	1	Board	1952
978-1-11111-145-1	30	5	1	Board	2001
978-1-11111-146-1	31	5	1	Board	1999

Table 6.6

Volume

inventory_id	isbn	condition_ code	date_acquired	asking_price	selling_price	sale_id
1	978-1-11111-111-1	3	12-JUN-19 00:00:00	175.00	175.00	1
2	978-1-11111-131-1	4	23-JAN-20 00:00:00	50.00	50.00	1
7	978-1-11111-137-1	2	20-JUN-19 00:00:00	80.00		
3	978-1-11111-133-1	2	05-APR-18 00:00:00	300.00	285.00	1
4	978-1-11111-142-1	1	05-APR-18 00:00:00	25.95	25.95	2
5	978-1-11111-146-1	1	05-APR-18 00:00:00	22.95	22.95	2
6	978-1-11111-144-1	2	15-MAY-19 00:00:00	80.00	76.10	2
8	978-1-11111-137-1	3	20-JUN-20 00:00:00	50.00		
9	978-1-11111-136-1	1	20-DEC-19 00:00:00	75.00		
10	978-1-11111-136-1	2	15-DEC-19 00:00:00	50.00		
11	978-1-11111-143-1	1	05-APR-20 00:00:00	25.00	25.00	3
12	978-1-11111-132-1	1	12-JUN-20 00:00:00	15.00	15.00	3
13	978-1-11111-133-1	3	20-APR-20 00:00:00	18.00	18.00	3
15	978-1-11111-121-1	2	20-APR-20 00:00:00	110.00	110.00	5
14	978-1-11111-121-1	2	20-APR-20 00:00:00	110.00	110.00	4
16	978-1-11111-121-1	2	20-APR-20 00:00:00	110.00		
17	978-1-11111-124-1	2	12-JAN-21 00:00:00	75.00		
18	978-1-11111-146-1	1	11-MAY-20 00:00:00	30.00	30.00	6
19	978-1-11111-122-1	2	06-MAY-20 00:00:00	75.00	75.00	6
20	978-1-11111-130-1	2	20-APR-20 00:00:00	150.00	120.00	6
21	978-1-11111-126-1	2	20-APR-20 00:00:00	110.00	110.00	6
22	978-1-11111-139-1	2	16-MAY-20 00:00:00	200.00	170.00	6
23	978-1-11111-125-1	2	16-MAY-20 00:00:00	45.00	45.00	7
24	978-1-11111-131-1	3	20-APR-20 00:00:00	35.00	35.00	7
25	978-1-11111-126-1	2	16-NOV-20 00:00:00	75.00	75.00	8
26	978-1-11111-133-1	3	16-NOV-20 00:00:00	35.00	55.00	8
27	978-1-11111-141-1	1	06-NOV-20 00:00:00	24.95		
28	978-1-11111-141-1	1	06-NOV-20 00:00:00	24.95		
29	978-1-11111-141-1	1	06-NOV-20 00:00:00	24.95		
30	978-1-11111-145-1	1	06-NOV-20 00:00:00	27.95		
31	978-1-11111-145-1	1	06-NOV-20 00:00:00	27.95		
32	978-1-11111-145-1	1	06-NOV-20 00:00:00	27.95		
33	978-1-11111-139-1	2	06-OCT-20 00:00:00	75.00	50.00	9
34	978-1-11111-133-1	1	16-NOV-20 00:00:00	125.00	125.00	10
35	978-1-11111-126-1	1	06-OCT-20 00:00:00	75.00	75.00	11
36	978-1-11111-130-1	3	06-DEC-19 00:00:00	50.00	50.00	11
37	978-1-11111-136-1	3	06-DEC-19 00:00:00	75.00	75.00	11

38	978-1-11111-130-1	2	06-APR-20 00:00:00	200.00	150.00	12
39	978-1-11111-132-1	3	06-APR-20 00:00:00	75.00	75.00	12
40	978-1-11111-129-1	1	06-APR-20 00:00:00	25.95	25.95	13
41	978-1-11111-141-1	1	16-MAY-20 00:00:00	40.00	40.00	14
42	978-1-11111-141-1	1	16-MAY-20 00:00:00	40.00	40.00	14
43	978-1-11111-132-1	1	12-NOV-20 00:00:00	17.95		
44	978-1-11111-138-1	1	12-NOV-20 00:00:00	75.95		
45	978-1-11111-138-1	1	12-NOV-20 00:00:00	75.95		
46	978-1-11111-131-1	3	12-NOV-20 00:00:00	15.95		
47	978-1-11111-140-1	3	12-NOV-20 00:00:00	25.95		
48	978-1-11111-123-1	2	16-AUG-20 00:00:00	24.95		
49	978-1-11111-127-1	2	16-AUG-20 00:00:00	27.95		
50	978-1-11111-127-1	2	06-JAN-21 00:00:00	50.00	50.00	15
51	978-1-11111-141-1	2	06-JAN-21 00:00:00	50.00	50.00	15
52	978-1-11111-141-1	2	06-JAN-21 00:00:00	50.00	50.00	16

inventory_id	isbn	condition_ code	date_acquired	asking_price	selling_price	sale_id
53	978-1-11111-123-1	2	06-JAN-21 00:00:00	40.00	40.00	16
54	978-1-11111-127-1	2	06-JAN-21 00:00:00	40.00	40.00	16
55	978-1-11111-133-1	2	06-FEB-21 00:00:00	60.00	60.00	17
56	978-1-11111-127-1	2	16-FEB-21 00:00:00	40.00	40.00	17
57	978-1-11111-135-1	2	16-FEB-21 00:00:00	40.00	40.00	18
59	978-1-11111-127-1	2	25-FEB-21 00:00:00	35.00	35.00	18
58	978-1-11111-131-1	2	16-FEB-21 00:00:00	25.00	25.00	18
60	978-1-11111-128-1	2	16-DEC-20 00:00:00	50.00	45.00	
61	978-1-11111-136-1	3	22-OCT-20 00:00:00	50.00	50.00	19
62	978-1-11111-115-1	2	22-OCT-20 00:00:00	75.00	75.00	20
63	978-1-11111-130-1	2	16-JUL-20 00:00:00	500.00		
64	978-1-11111-136-1	2	06-MAR-20 00:00:00	125.00		
65	978-1-11111-136-1	2	06-MAR-20 00:00:00	125.00		
66	978-1-11111-137-1	2	06-MAR-20 00:00:00	125.00		
67	978-1-11111-137-1	2	06-MAR-20 00:00:00	125.00		
68	978-1-11111-138-1	2	06-MAR-20 00:00:00	125.00		
69	978-1-11111-138-1	2	06-MAR-20 00:00:00	125.00		
70	978-1-11111-139-1	2	06-MAR-20 00:00:00	125.00		
71	978-1-11111-139-1	2	06-MAR-20 00:00:00	125.00		

Table 6.7

Customer

customer_ numb	first_ name	last_ name	street	city	state_ province	zip_ postcode	contact_ phone
1	Janice	Jones	125 Center Road	Anytown	NY	11111	518-555-1111
2	Jon	Jones	25 Elm Road	Next Town	NJ	18888	209-555-2222
3	John	Doe	821 Elm Street	Next Town	NJ	18888	209-555-3333
4	Jane	Doe	852 Main Street	Anytown	NY	11111	518-555-4444
5	Jane	Smith	1919 Main Street	New Village	NY	13333	518-555-5555
6	Janice	Smith	800 Center Road	Anytown	NY	11111	518-555-6666
7	Helen	Brown	25 Front Street	Anytown	NY	11111	518-555-7777
8	Helen	Jerry	16 Main Street	Newtown	NJ	18886	518-555-8888
9	Mary	Collins	301 Pine Road, Apt. 12	Newtown	NJ	18886	518-555-9999
10	Peter	Collins	18 Main Street	Newtown	NJ	18886	518-555-1010
11	Edna	Hayes	209 Circle Road	Anytown	NY	11111	518-555-1110
12	Franklin	Hayes	615 Circle Road	Anytown	NY	11111	518-555-1212
13	Peter	Johnson	22 Rose Court	Next Town	NJ	18888	209-555-1212
14	Peter	Johnson	881 Front Street	Next Town	NJ	18888	209-555-1414
15	John	Smith	881 Manor Lane	Next Town	NJ	18888	209-555-1515

Table 6.8

Sale

sale_id	customer_ numb	sale_date	sale_total_ amt	credit_card_ numb	exp_month	exp_year
3	1	15-JUN-21 00:00:00	58.00	1234 5678 9101 1121	10	18
4	4	30-JUN-21 00:00:00	110.00	1234 5678 9101 5555	7	17
5	6	30-JUN-21 00:00:00	110.00	1234 5678 9101 6666	12	17
6	12	05-JUL-21 00:00:00	505.00	1234 5678 9101 7777	7	16
7	8	05-JUL-21 00:00:00	80.00	1234 5678 9101 8888	8	16
8	5	07-JUL-21 00:00:00	90.00	1234 5678 9101 9999	9	15
9	8	07-JUL-21 00:00:00	50.00	1234 5678 9101 8888	8	16
10	11	10-JUL-21 00:00:00	125.00	1234 5678 9101 1010	11	16
11	9	10-JUL-21 00:00:00	200.00	1234 5678 9101 0909	11	15
12	10	10-JUL-21 00:00:00	200.00	1234 5678 9101 0101	10	15
13	2	10-JUL-21 00:00:00	25.95	1234 5678 9101 2222	2	15
14	6	10-JUL-21 00:00:00	80.00	1234 5678 9101 6666	12	17
15	11	12-JUL-21 00:00:00	75.00	1234 5678 9101 1231	11	17
16	2	25-JUL-21 00:00:00	130.00	1234 5678 9101 2222	2	15
17	1	25-JUL-21 00:00:00	100.00	1234 5678 9101 1121	10	18
18	5	22-AUG-21 00:00:00	100.00	1234 5678 9101 9999	9	15
2	1	05-JUN-21 00:00:00	125.00	1234 5678 9101 1121	10	18
1	1	29-MAY-21 00:00:00	510.00	1234 5678 9101 1121	10	18
19	6	01-SEP-21 00:00:00	95.00	1234 5678 9101 7777	7	16
20	2	01-SEP-21 00:00:00	75.00	1234 5678 9101 2222	2	15

Choosing Columns

One of the characteristics of a relation is that you can view any of the columns in any order you choose. SQL therefore lets you specify the columns you want to see, and the order in which you want to see them, using the relational algebra project operation to produce the final result table.

Retrieving All Columns

To retrieve all the columns in a table, viewing the columns in the order in which they were defined when the table was created, you can use an asterisk (*) rather than listing each column. For example, to see all the works that the rare book store has handled, you would use

```
SELECT *
FROM work;
```

Because this query is requesting all rows in the table, there is no WHERE clause. As you can see in Figure 16.1, the result table labels each column with its name.

work_numb	author_numb	title
1	1	Jane Eyre
2	1	Villette
3	2	Hound of the Baskervilles
4	2	Lost World, The
5	2	Complete Sherlock Holmes
7	3	Prince and the Pauper
8	3	Tom Sawyer
9	3	Adventures of Huckleberry Finn, The
6	3	Connecticut Yankee in King Arthur's Court, A
13	5	Fountainhead, The
14	5	Atlas Shrugged
15	6	Peter Pan

```
 15 |           6 | Peter Pan
 10 |           7 | Bourne Identity, The
 11 |           7 | Matarese Circle, The
 12 |           7 | Bourne Supremacy, The
 16 |           4 | Kidnapped
 17 |           4 | Treasure Island
 18 |           8 | Sot Weed Factor, The
 19 |           8 | Lost in the Funhouse
 20 |           8 | Giles Goat Boy
 21 |           9 | Dune
 22 |           9 | Dune Messiah
 23 |          10 | Foundation
 24 |          10 | Last Foundation
 25 |          10 | I, Robot
 26 |          11 | Inkheart
 27 |          11 | Inkdeath
 28 |          12 | Anathem
 29 |          12 | Snow Crash
 30 |           5 | Anthem
 31 |          12 | Cryptonomicon
```

FIGURE 16.1 **Viewing all columns in a table**

Note: The layouts of the printed output of many SQL queries in this book have been adjusted so that it will fit across the width of the pages. When you actually view listings on the screen, each row will be in a single horizontal line. If a listing is too wide to fit on the screen or a terminal program's window, either each line will wrap as it reaches the right edge of the display, or you will need to scroll.

Using the * operator to view all columns is a convenient shorthand for interactive SQL when you want a quick overview of data. However, it can be troublesome when used in embedded SQL (SQL inside a program of some sort) if the columns in the table are changed. In particular, if a column is added to the table and the application is not modified to handle the new column, then the application may not work properly. The * operator also displays the columns in the order in which they appeared in the CREATE statement that set up the table. This may simply not be the order you want.

Retrieving Specific Columns

In most SQL queries, you will want to specify exactly which column or columns you want retrieved and perhaps the order in which you want them to appear. To specify columns, you list them following SELECT in the order in which you want to see them. For example, a query to view the names and phone numbers of all of our store's customers is written

```
SELECT first_name,last_name,contact_phone
FROM customer;
```

The result (see Figure 16.2) shows all rows in the table for just the three columns specified in the query. The order of the columns in the result table matches the order in which the columns appeared after the SELECT keyword.

```
 first_name | last_name | contact_phone
------------+-----------+--------------
 Janice     | Jones     | 518-555-1111
 Jon        | Jones     | 209-555-2222
 John       | Doe       | 209-555-3333
 Jane       | Doe       | 518-555-4444
 Jane       | Smith     | 518-555-5555
 Janice     | Smith     | 518-555-6666
 Helen      | Brown     | 518-555-7777
 Helen      | Jerry     | 518-555-8888
 Mary       | Collins   | 518-555-9999
 Peter      | Collins   | 518-555-1010
```

Edna	Hayes	518-555-1110
Franklin	Hayes	518-555-1212
Peter	Johnson	209-555-1212
Peter	Johnson	209-555-1414
John	Smith	209-555-1515

FIGURE 16.2 **Choosing specific columns.**

Removing Duplicates

Unique primary keys ensure that relations have no duplicate rows. However, when you view only a portion of the columns in a table, you may end up with duplicates. For example, executing the following query produced the result in Figure 16.3:

```
SELET customer_numb, credit_card_numb
FROM sale;
```

customer_numb	credit_card_numb
1	1234 5678 9101 1121
1	1234 5678 9101 1121
1	1234 5678 9101 1121
1	1234 5678 9101 1121
2	1234 5678 9101 2222
2	1234 5678 9101 2222
2	1234 5678 9101 2222
4	1234 5678 9101 5555
5	1234 5678 9101 9999
5	1234 5678 9101 9999
6	1234 5678 9101 6666
6	1234 5678 9101 7777
6	1234 5678 9101 6666
8	1234 5678 9101 8888
8	1234 5678 9101 8888
9	1234 5678 9101 0909
10	1234 5678 9101 0101
11	1234 5678 9101 1231
11	1234 5678 9101 1010
12	1234 5678 9101 7777

FIGURE 16.3 **A result table with duplicate rows.**

Duplicates appear because the same customer used the same credit card number for more than one purchase. Keep in mind that, although this table with duplicate rows is not a legal relation, that doesn't present a problem for the database because it is not stored in the base tables.

To remove duplicates from a result table, you insert the keyword DISTINCT following SELECT:

```
SELECT DISTINCT customer_numb, credit_card_numb
FROM sale;
```

The result is a table without the duplicate rows (see Figure 16.4). Although a legal relation has no duplicate rows, most DBMS vendors have implemented SQL retrieval so that it leaves the duplicates. As you read earlier the primary reason is performance. To remove duplicates, a DBMS must sort the result table by every column in the table. It must then scan the table from top to bottom, looking at every "next" row to identify duplicate rows that are next to one another. If a result table is large, the sorting and scanning can significantly slow down the query. It is therefore

duplicate rows that are next to one another. If a result table is large, the sorting and scanning can significantly slow down the query. It is therefore up to the user to decide whether to request unique rows.

customer_numb	credit_card_numb
1	1234 5678 9101 1121
2	1234 5678 9101 2222
4	1234 5678 9101 5555
5	1234 5678 9101 9999
6	1234 5678 9101 6666
6	1234 5678 9101 7777
8	1234 5678 9101 8888
9	1234 5678 9101 0909
10	1234 5678 9101 0101
11	1234 5678 9101 1010
11	1234 5678 9101 1231
12	1234 5678 9101 7777

FIGURE 16.4 **The result table in Figure 16.3 with the duplicates removed**

Ordering the Result Table

The order in which rows appear in the result table may not be what you expect. In some cases, rows will appear in the order in which they are physically stored. However, if the query optimizer uses an index to process the query, then the rows will appear in index key order. If you want row ordering to be consistent and predictable, you will need to specify how you want the rows to appear.

When you want to control the order of rows in a result table, you add an ORDER BY clause to your SELECT statement. For example, if you issue the query

```
SELECT *
FROM author;
```

to get an alphabetical list of authors, you will see the listing in Figure 16.5. The author numbers are in numeric order because that was the order in which the rows were added, but the author names aren't alphabetical. Adding the ORDER BY clause sorts the result in alphabetical order by the *author_first_last* column (see Figure 16.6):

author_numb	author_last_first
1	Bronte, Charlotte
2	Doyle, Sir Arthur Conan
3	Twain, Mark
4	Stevenson, Robert Louis
5	Rand, Ayn
6	Barrie, James
7	Ludlum, Robert
8	Barth, John
9	Herbert, Frank
10	Asimov, Isaac
11	Funke, Cornelia
12	Stephenson, Neal

FIGURE 16.5 **The unordered result table.**

author_numb	author_last_first
10	Asimov, Isaac
6	Barrie, James
8	Barth, John
1	Bronte, Charlotte
2	Doyle, Sir Arthur Conan
11	Funke, Cornelia
9	Herbert, Frank
7	Ludlum, Robert
5	Rand, Ayn
12	Stephenson, Neal
4	Stevenson, Robert Louis
3	Twain, Mark

FIGURE 16.6 **The result table from Figure 16.5 sorted in alphabetical order by author name.**

```
SELECT *
```

```
FROM author
ORDER BY author_last_first;
```

The keywords ORDER BY are followed by the column or columns on which you want to sort the result table. When you include more than one column, the first column represents the outer sort, the next column a sort within it. For example, assume that you issue the query

```
SELECT zip_postcode, last_name, first_name
FROM customer
ORDER BY zip_postcode, last_name;
```

The result (see Figure 16.7) first orders by the zip code, and then sorts by the customer's last name within each zip code. If we reverse the order of the columns on which the output is to be sorted, as in

```
SELECT zip_postcode, last_name, first_name
FROM customer
ORDER BY last_name, zip_postcode;
```

zip_postcode	last_name	first_name
11111	Brown	Helen
11111	Doe	Jane
11111	Hayes	Edna
11111	Hayes	Franklin
11111	Jones	Janice
11111	Smith	Janice
13333	Smith	Jane
18886	Collins	Mary
18886	Collins	Peter
18886	Jerry	Helen
18888	Doe	John
18888	Johnson	Peter
18888	Johnson	Peter
18888	Jones	Jon
18888	Smith	John

FIGURE 16.7 **Sorting output by two columns.**

the output (see Figure 16.8) then sorts first by last name, and then by zip code within each last name.

zip_postcode	last_name	first_name
11111	Brown	Helen
18886	Collins	Peter
18886	Collins	Mary

```
11111     | Doe       | Jane
18888     | Doe       | John
11111     | Hayes     | Franklin
11111     | Hayes     | Edna
18886     | Jerry     | Helen
18888     | Johnson   | Peter
18888     | Johnson   | Peter
11111     | Jones     | Janice
18888     | Jones     | Jon
11111     | Smith     | Janice
13333     | Smith     | Jane
18888     | Smith     | John
```

FIGURE 16.8 **Reversing the sort order of the query in Figure 16.7.**

Choosing Rows

As well as viewing any columns from a relation, you can also view any rows you want. We specify row selection criteria in a SELECT statement's WHERE clause.

In its simplest form, a WHERE clause contains a logical expression against which each row in a table is evaluated. If a row meets the criteria in the expression, then it becomes a part of the result table. If the row does not meet the criteria, then it is omitted. The trick to writing row selection criteria is, therefore, knowing how to create logical expressions against which data can be evaluated.

Predicates

As you read earlier, a logical expression that follows WHERE is known as a predicate. It uses a variety of operators to represent row selection criteria. If a row meets the criteria in a predicate (in other words, the logical expression evaluates as true), then the row is included in the result table. If the row doesn't meet the criteria (the logical expression evaluates as false), then the row is excluded.

Relationship Operators

In Table 16.1 you can see the six operators used to express data relationships. To write an expression using one of the operators, you surround it with two values. In database queries, such expressions have either a column name on one side and a literal value on the other, as in

```
cost > 1.95
```

Table 16.1

The Relationship Operators

Operator	Meaning	Examples
=	Equal to	cost = 1.95 numb_in_stock = reorder_point
<	Less than	cost < 1.95 numb_in_stock < reorder_point
<=	Less than or equal to	cost <= 1.95 numb_in_stock <= reorder_point
>	Greater than	cost > 1.95 numb_in_stock >= reorder_point
>=	Greater than or equal to	cost >= 1.95 numb_in_stock >= reorder_point
!= or < > [1]	Not equal to	cost != 1.95 numb_in_stock != reorder_point

[1] Check the documentation that accompanies your DBMS to determine whether the "not equal to" operator is != or < >.

or column names on both sides:

```
numb_on_hand <= reorder_point
```

The first expression asks the question "Is the cost of the item greater than 1.95?" The second asks "Is the number of items in stock less than or equal to the reorder point?"

The way in which you enter literal values into a logical expression depends on the data type of the column to which you are comparing the value:

- Numbers: type numbers without any formatting. In other words, leave out dollar signs, commas, and so on. You should, however, put decimal points in the appropriate place in a number with a factional portion.
- Characters: type characters surrounded by quotation marks. Most DBMSs accept pairs of either single or double quotes. If your characters include an apostrophe (a single quote), then you should use double quotes. Otherwise, use single quotes.
- Dates: type dates in the format used to store them in the database. This will vary from one DBMS to another.
- Times: type times in the format used to store them in the database. This will vary from one DBMS to another.

When you are using two column names, keep in mind that the predicate is applied to each row in the table individually. The DBMS substitutes the values stored in the columns in the same row, when making its evaluation of the criteria. You can, therefore, use two column names when you need to examine data that are stored in the same row but in different columns. However, you cannot use a simple logical expression to compare the same column in two or more rows.

The DBMS also bases the way it evaluates data on the type of data:

- Comparisons involving numeric data are based on numerical order.
- Comparisons involving character data are based on alphabetical order.
- Comparisons involving dates and times are based on chronological order.

Logical Operators

Sometimes a simple logical expression is not enough to identify the rows you want to retrieve; you need more than one criterion. In that case, you can chain criteria together with logical operators. For example, assume that you want to retrieve volumes that you have in stock that cost more than $75, and that are in excellent condition. The conditions are coded (2 = excellent). The predicate you need is, therefore, made up of two simple expressions:

```
condition_code = 2
asking_price > 75
```

A row must meet both of these criteria to be included in the result table. You therefore connect the two expressions with the logical operator AND into a single complex expression:

```
condition_code = 2 AND asking_price > 75
```

Whenever you connect two simple expressions with AND, a row must meet *both* of the conditions to be included in the result.

You can use the AND operators to create a predicate that includes a range of dates. For example, if you want to find all sales that were made in Aug. and Sep. of 2019, the predicate would be written:[1]

```
sale_date >= '01-Aug-2019' AND sale_date <= '30-Sep-2019'
```

The same result would be generated by

```
Sale_date > '31-Jul-2019' AND sale_date < '1-Oct-2019'
```

To be within the interval, a sale date must meet both individual criteria.

You will find a summary of the action of the AND operators in Table 16.2. The labels in the columns and rows represent the result of evaluating the single expressions on either side of the AND. As you can see, the only way to get a true result for the entire expression is for both simple expressions to be true.

Table 16.2

AND Truth Table

AND	True	False
True	True	False
False	False	False

If you want to create an expression from which a row needs to meet only one condition from several conditions, then you connect simple expressions with the logical operator OR. For example, if you want to retrieve volumes that cost more than $125 or less than $50, you would use the predicate

```
asking_price > 125 OR asking_price < 50
```

Whenever you connect two simple expressions with OR, a row needs to meet only one of the conditions to be included in the result of the query. When you want to create a predicate that looks for dates outside an interval, you use the OR operator. For example, to see sales that occurred prior to Mar. 1, 2019 or after Dec. 31, 2019, the predicate is written

```
sale_date < '01-Mar-2019' OR sale_date > '31-Dec-2019'
```

You can find a summary of the OR operation in Table 16.3. Notice that the only way to get a false result is for both simple expressions surrounding OR to be false.

Table 16.3

OR Truth Table

OR	True	False
True	True	True
False	True	False

There is no limit to the number of simple expression you can connect with AND and OR. For example, the following expression is legal:

```
condition_code >= 3 AND selling_price < asking_price
    AND selling_price > 75
```

Negation

The logical operator NOT (or !) inverts the result of logical expression. If a row meets the criteria in a predicate, then placing NOT in front of the criteria *excludes* the row from the result. By the same token, if a row does not meet the criteria in a predicate, then placing NOT in front of the expression *includes* the row in the result. For example,

```
NOT (asking_price <= 50)
```

retrieves all rows where the cost is not less than or equal to $50 (in other words, greater than $50[2]). First the DBMS evaluates the value in the *asking_price* column against the expression *asking_price* <= 50. If the row meets the criteria, then the DBMS does nothing. If the row does not meet the criteria, it includes the row in the result.

The parentheses in the preceding example group the expression to which NOT is to be applied. In the following example, the NOT operator applies only to the expression *asking_price* <= 50.

```
NOT (asking_price <= 50) AND selling_price < asking_price
```

NOT can be a bit tricky when it is applied to complex expressions. As an example, consider this expression:

```
NOT (asking_price <= 50 AND selling_price < asking_price)
```

Rows that have both an asking price of less than or equal to $50, and a selling price that was less than the asking price will meet the criteria within parentheses. However, the NOT operator excludes them from the result. Those rows that have either an asking price of more than $50 or a selling price greater than or equal to the asking price will fail the criteria within the parentheses, but will be included in the result by the NOT. This means that the expression is actually the same as

```
asking_price > 50 OR selling_price >= asking_price
```

or

```
NOT (asking_price <= 50) OR NOT (selling_price < asking_price)
```

Precedence ad Parentheses

When you create an expression with more than one logical operation, the DBMS must decide on the order in which it will process the simple expressions. Unless you tell it otherwise, a DBMS uses a set of default *rules of precedence*. In general, a DBMS evaluates simple expressions first, followed by any logical expressions. When there is more than one operator of the same type, evaluation proceeds from left to right.

As a first example, consider this expression:

```
asking_price < 50 OR condition_code = 2 AND selling_price >
    asking_price
```

If the asking price of a book is $25, its condition code is 3, and the selling price was $20, the DBMS will exclude the row from the result. The first simple expression is true; the second is false. An OR between the first two produces a true result, because at least one of the criteria is true. Then the DBMS performs an AND between the true result of the first portion and the result of the third simple expression (false). Because we are combining a true result and a false result with AND, the overall result is false. The row is therefore excluded from the result.

We can change the order in which the DBMS evaluates the logical operators and, coincidentally, the result of the expression, by using parentheses to group the expressions that are to have higher precedence:

```
asking_price < 50 OR (condition_code = 2 AND selling_price >
    asking_price)
```

A DBMS gives highest precedence to the parts of the expression within parentheses. Using the same sample data from the preceding paragraph, the expression within parentheses is false (both simple expressions are false). However, the OR with the first simple expression produces true, because the first simple expression is true. Therefore, the row is included in the result.

Special Operators

SQL predicates can include a number of special operators that make writing logical criteria easier. These include BETWEEN, LIKE, IN, and IS NULL.

Note: There are additional operators that are used primarily with subqueries, SELECT statements in which you embed one complete SELECT within another. You will be introduced to them in Chapter 17.

Between

The BETWEEN operator simplifies writing predicates that look for values that lie within an interval. Remember the example you saw earlier in this chapter using AND to generate a date interval? Using the BETWEEN operator, you could rewrite that predicate as

```
sale_date BETWEEN '01-Aug-2021' AND '30-Sep-2021'
```

Any row with a sale date of Aug. 1, 2021 through Sep. 30, 2021 will be included in the result.

If you negate the BETWEEN operator, the DBMS returns all rows that are outside the interval. For example,

```
sale_date NOT BETWEEN '01-Aug-2021' AND '30-Sep-2021'
```

retrieves all rows with dates *prior* to Aug. 1, 2019 and *after* Sep. 31, 2021. It does not include 01-Aug-2021 or 30-Sep-2021. NOT BETWEEN is therefore a shorthand for the two simple expressions linked by OR that you saw earlier in this chapter.

Like

The LIKE operator provides a measure of character string pattern matching by allowing you to use placeholders (wildcards) for one or more characters. Although you may find that the wildcards are different in your particular DBMS, in most case, % stands for zero, one, or more characters and _ stands for zero or one character.

The way in which the LIKE operator works is summarized in Table 16.4. As you can see, you can combine the two wildcards to produce a variety of begins with, ends with, and contains expressions.

Table 16.4

Using the LIKE Operator

Expression	Meaning
`LIKE 'Sm%'`	Begins with *Sm*
`LIKE '%ith'`	Ends with *ith*
`LIKE '%ith%'`	Contains *ith*
`LIKE 'Sm_'`	Begins with *Sm* and is followed by at most one character
`LIKE '_ith'`	Ends with *ith* and is preceded by at most one character
`LIKE '_ith_'`	Ontains *ith* and begins and ends with at most one additional character
`LIKE '%ith_'`	Contains *ith*, begins with any number of characters, and ends with at most one additional character
`LIKE '_ith%'`	Contains *ith*, begins with at most one additional character, and ends with any number of characters

As with BETWEEN you can negate the LIKE operator:

```
last_name NOT LIKE 'Sm%'
```

Rows that are like the pattern are therefore excluded from the result.

One of the problems you may run into when using LIKE is that you need to include the wildcard characters as part of your data. For example, what can you do if you want rows that contain 'nd_by'? The expression you want is

```
column_name LIKE '%nd_by%'
```

The problem is that the DBMS will see the _ as a wildcard, rather than as characters in your search string. The solution was introduced in SQL-92, providing you with the ability to designate an *escape character*.

An escape character removes the special meaning of the character that follows. Because many programming languages use \ as the escape character, it is a logical choice for pattern matching, although it can be any character that is not part of your data. To establish the escape character, you add the keyword ESCAPE, followed by the escape character, to your expression:

```
column_name LIKE '%nd\_by%' ESCAPE '\'
```

In

The IN operator compares the value in a column against a set of values. IN returns true if the value is within the set. For example, assume that a store employee is checking the selling price of a book and wants to know if it is $25, $50, or $60. Using the IN operator, the expression would be written:

```
selling_price IN (25,50,60)
```

This is shorthand for

```
selling_price = 25 OR selling_price = 50 OR selling_price = 60
```

Therefore, any row whose price is one of those three values will be included in the result. Conversely, if you write the predicate

```
selling_price NOT IN (25,50,60)
```

the DBMS will return the rows with prices other than those in the set of values. The preceding expression is therefore the same as

```
selling_price != 25 AND selling_price != 50 AND selling_price !=60
```

or

```
NOT (selling_price = 25 OR selling_price = 50 OR selling_proce = 60)
```

Note: The most common use of IN and NOT IN is with a subquery, where the set of values to which data are compared are generated by an embedded SELECT. You will learn about this is Chapter 17.

Is Null

As you know, null is a specific indicator in a database. Although columns that contain nulls appear empty when you view them, the database actually stores a value that represents null, so that an unknown value can be distinguished from, for example, a string value containing a blank. As a user, however, you will rarely know exactly what a DBMS is using internally for null. This means that you need some special way to identify null in a predicate, so you can retrieve rows containing nulls. That is where the IS NULL operator comes in.

For example, an expression to identify all rows for volumes that have not been sold is written as

```
sale_date IS NULL
```

Conversely, to find all volumes that have been sold, you could use

```
sale_date IS NOT NULL
```

Performing Row Selection Queries

To perform SQL queries that select specific rows, you place a predicate after the SQL keyword WHERE. Depending on the nature of the predicate, the intention of the query may be to retrieve one or more rows. In this section, you therefore will see some SELECT examples that combine a variety of row selection criteria. You will also see how those criteria are combined in queries with column selection, and with sorting of the output.

Using a Primary Key Expression to Retrieve One Row

A common type of SQL retrieval query uses a primary key expression in its predicate to retrieve exactly one row. For example, if someone at the rare book store wants to see the name and telephone number of customer number 6, then the query is written

```
SELECT first_name, last_name, contact_phone
FROM customer
WHERE customer_numb = 6;
```

The result is the single row requested by the predicate.

```
first_name | last_name | contact_phone
-----------+-----------+--------------
Janice     | Smith     | 518-555-6666
```

If a table has a concatenated primary key, such as the employee number and child name for the *dependents* table you saw earlier in the book, then a primary key expression needs to include a complex predicate in which each column of the primary key appears in its own simple logical expression. For example, if you wanted to find the birthdate of employee number 0002's son John, you would use following query:

```
SELECT child_birth_date
FROM dependents
WHERE employee_number = '0002' AND child_name = 'John';
```

In this case, the result is simply

```
child_birth_date
----------------
```

2-Dec-1999

Retrieving Multiple Rows

Although queries with primary key expressions are written with the intention of retrieving only one row, more commonly SQL queries are designed to retrieve multiple rows.

Using Simple Predicates

When you want to retrieve data based on a value in a single column, you construct a predicate that includes just a simple logical expression. For example, to see all the books ordered on sale number 6, an application program being used by a store employee would use

```
SELECT isbn
FROM volume
WHERE sale_id = 6;
```

The output (see Figure 16.9) displays a single column for rows where the *sale_id* is 6.

```
isbn
-------------------
978-1-11111-146-1
978-1-11111-122-1
978-1-11111-130-1
978-1-11111-126-1
978-1-11111-139-1
```

FIGURE 16.9 **Displaying a single column from multiple rows using a simple predicate.**

Using Complex Predicates

When you want to see rows that meet two or more simple conditions, you use a complex predicate in which the simple conditions are connected by AND or OR. For example, if someone wanted to see the books on order number 6 that sold for less than the asking price, the query would be written

```
SELECT isbn
FROM volume
WHERE sale_id = 6 AND selling_price < asking_price;
```

Only two rows meet the criteria:

isbn

978-1-11111-130-1
978-1-11111-139-1

By the same token, if you wanted to see all sales that took place prior to Aug. 1, 2021, and for which the total amount of the sale was less than $100, the query would be written[3]

```
SELECT sale_id, sale_total_amt
FROM sale
WHERE sale_date < '1-Aug-2021' AND sale_total_amt < 100;
```

It produces the result in Figure 16.10.

sale_id	sale_total_amt
3	58.00
7	80.00
8	90.00
9	50.00
13	25.95
14	80.00
15	75.00

FIGURE 16.10 **Retrieving rows using a complex predicate including a date.**

Note: Don't forget that the date format required by your DMBS may be different from the one used in examples in this book.

Alternatively, if you needed information about all sales that occurred prior to or on Aug. 1, 2021 that totaled more than 100, along with sales that occurred after Aug. 1, 2019 that totaled less than 100, you would write the query

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE (sale_date <= '1-Aug-2021' AND sale_total_amt > 100) OR
(sale_date > '1-Aug-2021' AND sale_total_amt < 100);
```

Notice that although the AND operator has precedence over OR and, therefore, the parentheses are not strictly necessary, the predicate in this query includes parentheses for clarity. Extra parentheses are never a problem—as long as you balance every opening parenthesis with a closing parenthesis—and you should feel free to use them whenever they help make it easier to understand the meaning of a complex predicate. The result of this query can be seen in Figure 16.11.

sale_id	sale_date	sale_total_amt
4	30-JUN-21 00:00:00	110.00
5	30-JUN-21 00:00:00	110.00
6	05-JUL-21 00:00:00	505.00
10	10-JUL-21 00:00:00	125.00
11	10-JUL-21 00:00:00	200.00

12	10-JUL-21 00:00:00	200.00
16	25-JUL-21 00:00:00	130.00
2	05-JUN-21 00:00:00	125.00
1	29-MAY-21 00:00:00	510.00
19	01-SEP-21 00:00:00	95.00
20	01-SEP-21 00:00:00	75.00

FIGURE 16.11 **Using a complex predicate that includes multiple logical operators.**

Note: The default output format for a column of type date in PostgreSQL (the DBMS used to generate the sample queries) includes the time. Because no time was entered (just a date), the time displays as all 0s.

Using Between and Not Between

As an example of using one of the special predicate operators, consider a query where someone wants to see all sales that occurred between Jul. 1, 2019 and Aug. 31, 2019. The query would be written

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE sale_date BETWEEN '1-Jul-2021' AND '31-Aug-2021';
```

It produces the output in Figure 16.12.

sale_id	sale_date	sale_total_amt
6	05-JUL-21 00:00:00	505.00
7	05-JUL-21 00:00:00	80.00
8	07-JUL-21 00:00:00	90.00
9	07-JUL-21 00:00:00	50.00
10	10-JUL-21 00:00:00	125.00
11	10-JUL-21 00:00:00	200.00
12	10-JUL-21 00:00:00	200.00
13	10-JUL-21 00:00:00	25.95
14	10-JUL-21 00:00:00	80.00
15	12-JUL-21 00:00:00	75.00
16	25-JUL-21 00:00:00	130.00
17	25-JUL-21 00:00:00	100.00
18	22-AUG-21 00:00:00	100.00

FIGURE 16.12 **Using BETWEEN to retrieve rows in a date range.**

The inverse query retrieves all orders not placed between Jul. 1, 2019 and Aug. 31, 2019 is written

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE sale_date NOT BETWEEN '1-Jul-2019' AND '31-Aug-2019';
```

and produces the output in Figure 16.13.

sale_id	sale_date	sale_total_amt
3	15-JUN-21 00:00:00	58.00
4	30-JUN-21 00:00:00	110.00
5	30-JUN-21 00:00:00	110.00

```
 2 | 05-JUN-21 00:00:00 |          125.00
 1 | 29-MAY-21 00:00:00 |          510.00
19 | 01-SEP-21 00:00:00 |           95.00
20 | 01-SEP-21 00:00:00 |           75.00
```

FIGURE 16.13 **Using NOT BETWEEN to retrieve rows outside a date range.**

If we want output that is easier to read, we might ask the DBMS to sort the result by sale date:

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE sale_date NOT BETWEEN '1-Jul-2021' AND '31-Aug-2021'
ORDER BY sale_date;
```

producing the result in Figure 16.14.

```
 sale_id |     sale_date       | sale_total_amt
---------+---------------------+---------------
       1 | 29-MAY-21 00:00:00 |          510.00
       2 | 05-JUN-21 00:00:00 |          125.00
       3 | 15-JUN-21 00:00:00 |           58.00
       5 | 30-JUN-21 00:00:00 |          110.00
       4 | 30-JUN-21 00:00:00 |          110.00
      19 | 01-SEP-21 00:00:00 |           95.00
      20 | 01-SEP-21 00:00:00 |           75.00
```

FIGURE 16.14 **Output sorted by date.**

Nulls and Retrieval: Three-Valued Logic

The predicates you have seen to this point omit one important thing: the presence of nulls. What should a DBMS do when it encounters a row that contains null, rather than a known value? As you know, the relational data model doesn't have a specific rule as to what a DBMS should do, but it does require that the DBMS act consistently when it encounters nulls.

Consider the following query as an example:

```
SELECT inventory_id, selling_price
FROM volume
WHERE selling_price < 100;
```

The result can be found in Figure 16.15. Notice that every row in the result table has a value of selling price, which means that rows for unsold items—those with null in the selling price column—are omitted. The DBMS can't ascertain what the selling price for unsold items will be: maybe it will be less than $100, or maybe it will be greater than or equal to $100.

```
 inventory_id | selling_price
--------------+---------------
            2 |         50.00
            4 |         25.95
            5 |         22.95
            6 |         76.10
           11 |         25.00
```

```
11 |          25.00
12 |          15.00
13 |          18.00
18 |          30.00
19 |          75.00
23 |          45.00
24 |          35.00
25 |          75.00
26 |          55.00
33 |          50.00
35 |          75.00
36 |          50.00
37 |          75.00
39 |          75.00
40 |          25.95
41 |          40.00
42 |          40.00
50 |          50.00
51 |          50.00
52 |          50.00
53 |          40.00
54 |          40.00
55 |          60.00
56 |          40.00
57 |          40.00
59 |          35.00
58 |          25.00
60 |          45.00
61 |          50.00
62 |          75.00
```

FIGURE 16.15 **Retrieval based on a column that includes rows with nulls.**

The policy of most DBMSs is to exclude rows with nulls from the result. For rows with null in the selling price column, the *maybe* answer to "Is selling price less than 100" becomes *false*. This seems pretty straightforward, but what happens when you have a complex logical expression of which one portion returns *maybe*? The operation of AND, OR, and NOT must be expanded to take into account that they may be operating on a *maybe*.

The three-valued logic table for AND can be found in Table 16.5. Notice that something important hasn't changed: the only way to get a true result is for both simple expressions linked by AND to be true. Given that most DBMSs exclude rows where the predicate evaluates to *maybe*, the presence of nulls in the data will not change what an end user sees.

Table 16.5

Three-Valued AND Truth Table

AND	True	False	Maybe
True	True	False	Maybe
False	False	False	False
Maybe	Maybe	False	Maybe

The same is true when you look at the three-valued truth table for OR (see Table 16.6). As long as one simple expression is true, it does not matter whether the second returns true, false, or maybe. The result will always be true.

Table 16.6

Three-Valued OR Truth Table

OR	True	False	Maybe
True	True	True	True
False	True	False	Maybe
Maybe	True	Maybe	Maybe

Maybe	True	Maybe	Maybe

If you negate an expression that returns *maybe*, the NOT operator has no effect. In other words, NOT (MAYBE) is still *maybe*.

To see the rows that return *maybe*, you need to add an expression to your query that uses the IS NULL operator. For example, the easiest way to see which volumes have not been sold is to write a query like:

```
SELECT inventory_id, isbn, selling_price
FROM volume
WHERE selling_price IS NULL;
```

The result can be found in Figure 16.16. Note that the selling price column is empty in each row. (Remember that you typically can't see any special value for null.) Notice also that the rows in this result table are all those excluded from the query in Figure 16.15.

inventory_id	isbn	selling_price
7	978-1-11111-137-1	
8	978-1-11111-137-1	
9	978-1-11111-136-1	
10	978-1-11111-136-1	
16	978-1-11111-121-1	
17	978-1-11111-121-1	
27	978-1-11111-141-1	
28	978-1-11111-141-1	
29	978-1-11111-141-1	
30	978-1-11111-145-1	
31	978-1-11111-145-1	
32	978-1-11111-145-1	
43	978-1-11111-132-1	
44	978-1-11111-138-1	
45	978-1-11111-138-1	
46	978-1-11111-131-1	
47	978-1-11111-140-1	
48	978-1-11111-123-1	
49	978-1-11111-127-1	
63	978-1-11111-130-1	
64	978-1-11111-136-1	
65	978-1-11111-136-1	
66	978-1-11111-137-1	
67	978-1-11111-137-1	
68	978-1-11111-138-1	
69	978-1-11111-138-1	
70	978-1-11111-139-1	
71	978-1-11111-139-1	

FIGURE 16.16 **Using IS NULL to retrieve rows containing nulls.**

Four-Valued Logic

Codd's 330 rules for the relational data model include an enhancement to three-valued logic that he called *four-valued logic*. In four-valued logic, there are actually two types of null: "null and it doesn't matter that it's null" and "null and we've really got a problem because it's null." For example, if a company sells internationally, then it probably has a column for the country of each customer. Because it is essential to know a customer's country, a null in the *country* column would fall into the category of "null and we've really got a problem." In contrast, a missing value in a *company name* column would be quite acceptable in a customer table for rows that represented individual customers. Then the null would be "null and it doesn't matter that it's null." Four-valued logic remains purely theoretical at this time, however, and hasn't been widely implemented. (In fact, as far as I know, only one DBMS uses four-valued logic: FirstSQL.)

---

[1] This date format is a fairly generic one that is recognized by many DMBSs. However, you should consult the documentation for your DBMS to determine exactly what will work with your product.

[2] It is actually faster for the DBMS to evaluate *asking_price* > 50 because it involves only one action (is the asking price greater than 50) rather than three: Is the asking price equal to 50? Is the asking price less than 50? Take the opposite of the result of the first two questions.

[3] Whether you need quotes around date expressions depends on the DBMS.