

# MIS 381 INTRO. TO DATABASE MANAGEMENT

---

Indexes and Sequences

Data Manipulation Language

**Tayfun Keskin**

Visiting Clinical Professor, The University of Texas at Austin, McCombs School of Business  
Associate Teaching Professor, University of Washington Seattle, Foster School of Business

# QUESTIONS

Any questions  
before we begin ...



# AGENDA



Lecture

Indexes  
Sequences



Hands-On

Exercises



Looking Forward

Exam 1  
Homework 2



The University of Texas at Austin  
McCombs School of Business

# REVIEW QUESTION

What is a good primary key?

# QUESTION

What do you think will happen if you try to input a primary key identical to an existing cell?

# WHAT ARE INDEXES?

- Oracle schema objects (like tables) created to improve the performance of data access
- Oracle provides several types of indexes
  - Default (most common) type is known as B-tree
  - A composite index can be created on multiple columns
  - Primary keys or unique constraints implicitly create indexes

# QUESTION

Imagine a large dataset :

How many rows will Oracle DB will scan if you want to retrieve the sales performed yesterday?

# INDEXES

- Why?
- When do you NOT use an index?

Speeds up:

- Joins
- Searches

Columns that are updated frequently

But why is this bad?

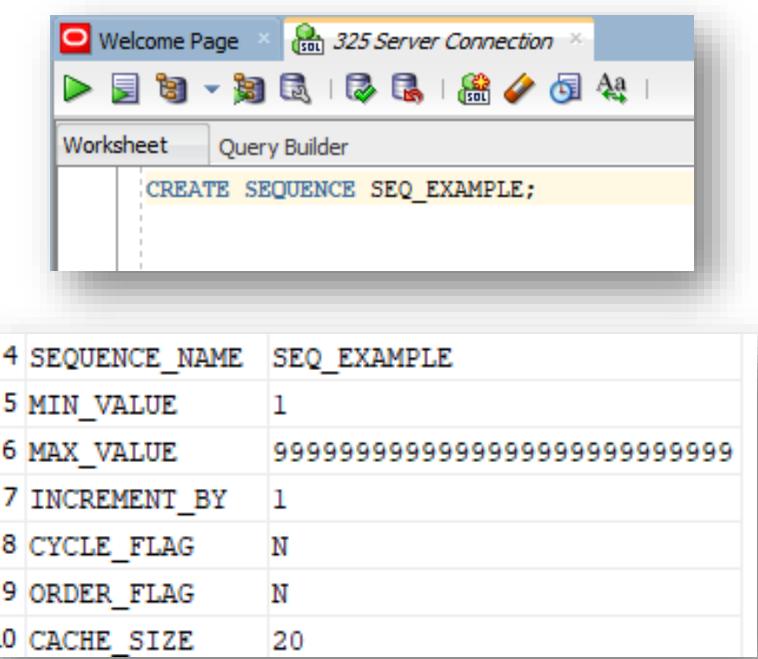


# WHAT ARE SEQUENCES?

- Automatically generated sequence of integer values
- Sequences are independent Oracle schema objects (are not linked to any specific table in a database)
- You can reference the next value of a sequence when inserting data as many tables as you'd like (they can be used freely as a part of your insert statements)

# SEQUENCE CHARACTERISTICS

- Min value
  - Max value
  - Increment by
  - Cache
  - Cycle or no cycle



# QUESTION

Why would I ever need a sequence?  
(an automatically incrementing column)

# USE CASE FOR A SEQUENCE

- A database object that automatically generate a sequence of integer values
- Typically, it is used to generate a value for the primary key
- Much easier when adding new data

# HANDS ON PRACTICE: OPEN ORACLE SQL DEVELOPER



# IN-CLASS EXERCISE FILES

- ICE 1 – Drop / create script
- ICE 2 – Hands-on exercise on sequences and indexes
- ICE 3 – Self (team) practice on members table



# PRACTICE FOR INDEXES

1. Start by discussing the following questions with your partner
  - a. What would be likely fields to create an index on for the invoices table based on rules discussed?
  - b. Should we create an index on vendor\_id? If so, why?
2. Assuming we do want to create an index on invoices.vendor\_id, write that syntax out and run it. Try to do this by reference only the syntax and not the examples. Confirm if your index was created successfully (i.e. without error).
3. Discuss if you should create an index on the invoice\_date. If so, should we consider sorting the index in a specific way? Based on your discussion, create the index you think would be most valuable.

# TOO MANY INDEXES?

- Creating too many indexes aren't ideal
- The more indexes you create on your tables, the slower your insert, update, delete commands will be
  - Because each time you manipulate a row in your table, indexes associated with that table will also have to be updated as well
- Only frequently queried columns should be indexed

# LOOKING FORWARD

We covered Chapter 10

**What's next?**

- DML

We will start Chapter 7

Exam 1

Homework 2



# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# DATA MANIPULATION LANGUAGE

- Insert
- Update
- Delete



# REVIEW QUESTION

Which command do we use to start a SQL query?

```
SELECT *
FROM Employees
```

# SQL STATEMENTS (SO FAR)

- **Queries:** Retrieving data from database tables

```
SELECT * FROM Employees;
```

- **DDL:** Creating/defining new database objects

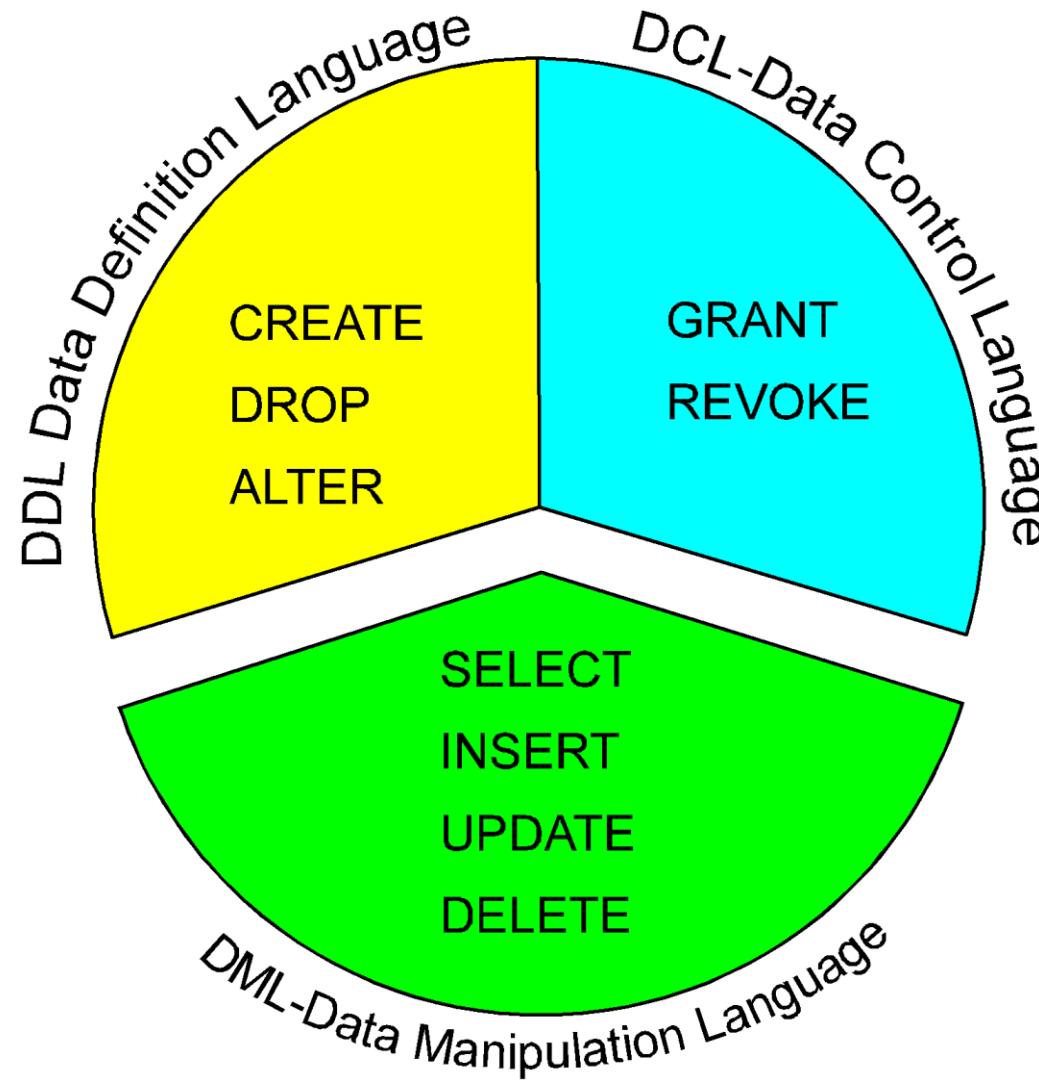
```
CREATE TABLE Employees  
(EmpID Number, FName Varchar2(20), LName Varchar2(20));
```

- **DML:** Modifying/manipulating existing data in your database

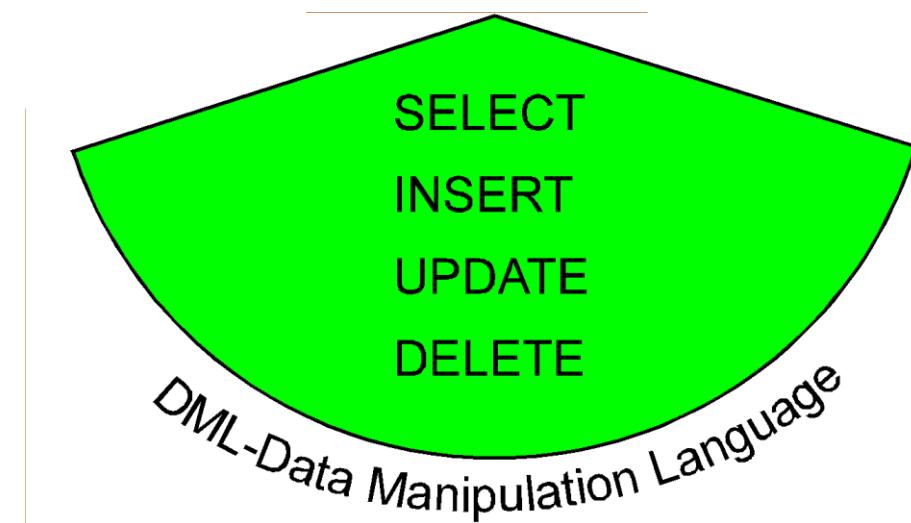
```
INSERT INTO Employees  
VALUES (1, 'John', 'Smith');
```



# SQL OVERVIEW



# SQL OVERVIEW



# IMPORTANT TERMS

- **Transaction:** autonomous unit of work where data is modified
  - Any DML command will implicitly create a transaction
- **Commit:** make the changes persistent (closes the transaction)
  - Data isn't officially saved to table for other users to see until you commit
  - You'll be able to see data you insert before a commit
- **Rollback:** undo the changes
  - If you rollback before you commit, data will not be saved to table
  - Rollback will have no effect right after commit

# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# **BACKUP SLIDES**



# PART 1

ICE 2 – Chapter 10 (Indexes and Sequences)



The University of Texas at Austin  
McCombs School of Business

## The syntax of the CREATE SEQUENCE statement

```
CREATE SEQUENCE sequence_name
  [START WITH starting_integer]
  [INCREMENT BY increment_integer]
  [{MINVALUE minimum_integer | NOMINVALUE}]
  [{MAXVALUE maximum_integer | NOMAXVALUE}]
  [{CYCLE | NOCYCLE}]
  [{CACHE cache_size | NOCACHE}]
  [{ORDER | NOORDER}]
```

### A statement that creates a sequence

```
CREATE SEQUENCE vendor_id_seq
```

### A statement that specifies a starting integer

```
CREATE SEQUENCE vendor_id_seq
  START WITH 124
```

### A statement that specifies all parameters

```
CREATE SEQUENCE test_seq
  START WITH 100 INCREMENT BY 10
  MINVALUE 0 MAXVALUE 1000000
  CYCLE CACHE 100;
```

### Let's Clarify:

- **Cycle** = return to zero after seq hits the max value
- **Cache** = Saves the next set of numbers in the seq in memory to speed up performance.  
Default cache is 20.
- **Order** = Guarantees the numbers generated in order of request. *Only used in unique application design situations*

## A statement that creates a sequence

```
CREATE SEQUENCE vendor_id_seq
```

## A statement that specifies a starting integer

```
CREATE SEQUENCE vendor_id_seq
    START WITH 124
```

## A statement that specifies all parameters

```
CREATE SEQUENCE test_seq
    START WITH 100 INCREMENT BY 10
    MINVALUE 0 MAXVALUE 1000000
    CYCLE CACHE 100;
```

## Practice:

1. Create a sequence called ***member\_id\_seq*** that starts at 10 and increments by 1.
4. Run test INSERT statements and confirm sequence was created correctly and works
6. Update the DDL provided to default the member\_id column to the next value of the newly created sequence
7. Run 2<sup>nd</sup> test INSERT statements and then confirm member\_id defaults correctly

## Example: How to use a sequence as the default value

```
CREATE TABLE invoices
(
    invoice_id      NUMBER      DEFAULT invoice_id_seq.NEXTVAL PRIMARY KEY,
    invoice_date    DATE        NOT NULL
)
```

## The syntax of the ALTER SEQUENCE statement

```
ALTER SEQUENCE sequence_name  
[sequence_attributes]
```

### A statement that alters a sequence

```
ALTER SEQUENCE test_seq  
INCREMENT BY 9  
MINVALUE 99 MAXVALUE 999999  
NOCYCLE CACHE 9 NOORDER;
```

### A statement that drops a sequence

```
DROP SEQUENCE test_seq;
```

### Practice

5. Update member\_id\_seq to increment by 10 instead of 1
6. Rerun 2<sup>nd</sup> test INSERTS and SELECT to see if they still worked.
7. Drop member\_id\_seq

# The sequences for the AP schema

The screenshot shows the Oracle SQL Developer interface. The title bar reads "Oracle SQL Developer : Sequence AP.INVOICE\_ID\_SEQ@ap". The menu bar includes File, Edit, View, Navigate, Run, Team, Tools, Window, and Help. The toolbar has various icons for connection management, file operations, and search. The left sidebar has a "Find Database Object" search bar and lists database objects under "Connections" for the "ap" connection, including Tables, Views, Indexes, Packages, Procedures, Functions, Queues, Queues Tables, Triggers, Crossedition Triggers, Types, Sequences, Materialized Views, Materialized View Logs, Synonyms, Public Synonyms, Database Links, Public Database Links, and Directories. The main workspace displays the "create\_ap\_tables.sql" script and the "INVOICE\_ID\_SEQ" sequence details. The "Details" tab shows the following properties:

Name	Value
1 CREATED	02-JUN-14
2 LAST_DDL_TIME	02-JUN-14
3 SEQUENCE_OWNER	AP
4 SEQUENCE_NAME	INVOICE_ID_SEQ
5 MIN_VALUE	1
6 MAX_VALUE	99999999999999999999999999999999
7 INCREMENT_BY	1
8 CYCLE_FLAG	N
9 ORDER_FLAG	N
10 CACHE_SIZE	20
11 LAST_NUMBER	115

# How to work with Indexes

- Speeds up joins and searches. How?
- By default, Oracle creates an index on each Primary Key
- Best to add Indexes on columns:
  - Frequently used in joins (i.e. Foreign Keys especially)
  - Frequently used in searches
  - Has a UNIQUE integrity constraint
  - Not updated a lot - indexes slow down insert, update, deletes
- FAQ: Is indexing everything bad? Who typically makes indexes?

## The syntax of the CREATE INDEX statement

```
CREATE [UNIQUE] INDEX index_name
    ON table_name (column_name_1 [ASC|DESC]
                    [, column_name_2 [ASC|DESC]]...)
```

A statement that creates an index  
based on a single column

```
CREATE INDEX invoices_vendor_id_ix
    ON invoices (vendor_id);
```

A statement that creates an index  
based on two columns

```
CREATE INDEX invoices_vendor_id_inv_no_ix
    ON invoices (vendor_id, invoice_number);
```

A statement that creates a unique index

```
CREATE UNIQUE INDEX vendors_vendor_phone_ix
    ON vendors (vendor_phone);
```

A statement that creates an index  
that's sorted in descending order

```
CREATE INDEX invoices_invoice_total_ix
    ON invoices (invoice_total DESC);
```

**TIP:** Use a standard naming convention  
for readability. e.g. table\_column\_ix

## Index Syntax using *table\_field\_ix* naming standard

```
-- Create the indexes
CREATE INDEX vendors_terms_id_ix
    ON vendors (default_terms_id);
CREATE INDEX vendors_account_number_ix
    ON vendors (default_account_number);
```

### Practice:

8. What would be likely fields to create an index on for the invoice table?
9. Create an index on vendor\_id. Why?
10. Create an index on invoice\_date why? Should we consider sorting?  
Create index.

## A statement that creates a function-based index

```
CREATE INDEX vendors_vendor_name_upper_ix  
  ON vendors (UPPER(vendor_name));
```

## Another statement for a function-based index

```
CREATE INDEX invoices_balance_due_ix  
  ON invoices (invoice_total - payment_total - credit_total DESC);
```

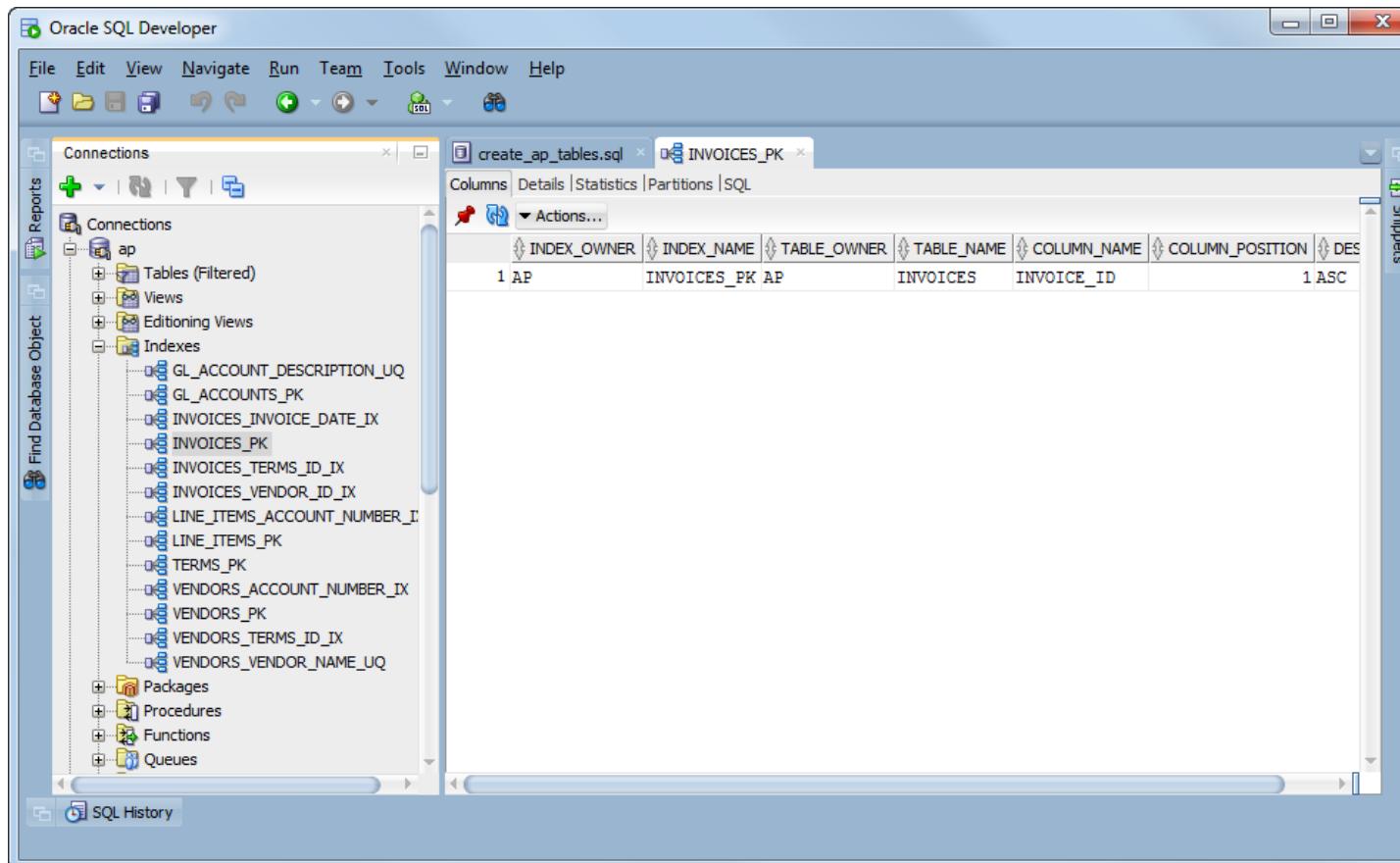
## How to enable function-based indexes

```
CONNECT system/system;  
ALTER SYSTEM SET QUERY_REWRITE_ENABLED=TRUE;
```

## A statement that drops an index

```
DROP INDEX vendors_vendor_state_ix
```

# The indexes for the AP schema



# PART 2

Chapter 7

DML – Insert, Update, Delete



The University of Texas at Austin  
McCombs School of Business

## INSERT statements add data to tables

```
INSERT INTO invoices
VALUES (115, 97, '456789', '01-AUG-14', 8344.50, 0, 0, 1, '31-AUG-14', NULL)
```

The response from the system

```
1 rows inserted
```

## A COMMIT statement that commits the changes

```
COMMIT
```

The response from the system

```
COMMIT succeeded
```

## A statement that rolls back the changes

```
ROLLBACK
```

The response from the system

```
ROLLBACK succeeded
```

# Terms you should know

- Transaction
- Commit
  - NOTE: data isn't officially saved to table for other users to see until you Commit.
  - NOTE: You'll be able to see data you insert before a commit
- Rollback
  - NOTE: If you rollback before you commit, data will not be saved to table

# The syntax of the CREATE TABLE AS statement

```
CREATE TABLE table_name AS
SELECT select_list
FROM table_source
[WHERE search_condition]
```

## Example

```
CREATE TABLE invoices_copy AS
SELECT *
FROM invoices
```

## When would this be useful?

- Avoid touching production tables in a live DB
- Let's you practice on non-production tables.
- NOTE: It's not likely you'll be creating "test" tables in a Production environment

## A statement that creates a partial copy of the Invoices table

```
CREATE TABLE old_invoices AS
SELECT *
FROM invoices
WHERE invoice_total - payment_total - credit_total = 0
```

## A statement that creates a table with summary rows from the Invoices table

```
CREATE TABLE vendor_balances AS
SELECT vendor_id, SUM(invoice_total) AS sum_of_invoices
FROM invoices
WHERE (invoice_total - payment_total - credit_total) <> 0
GROUP BY vendor_id
```

## A statement that deletes a table

```
DROP TABLE old_invoices
```

### Warning

- When you use the SELECT statement to create a table, only the column definitions and data are copied.
- Definitions of primary keys, foreign keys, indexes, default values, and so on are not included in the new table.

## INSERT syntax for inserting a single row

```
INSERT INTO table_name [(column_list)]
    VALUES (value_1 [, expression_2]...)
```

### An INSERT statement that adds a new row without using a column list

```
INSERT INTO invoices
VALUES (115, 97, '456789', '01-AUG-14', 8344.50, 0, 0, 1, '31-AUG-14', NULL)
(1 rows inserted)
```

### An INSERT statement that adds the new row using a column list

```
INSERT INTO invoices
    (invoice_id, vendor_id, invoice_number, invoice_total, payment_total,
     credit_total, terms_id, invoice_date, invoice_due_date)
VALUES
    (115, 97, '456789', 8344.50, 0,
     0, 1, '01-AUG-14', '31-AUG-14')
(1 rows inserted)
```

NOTE: It's okay to continue on 2 lines

## The definition of the Color\_Sample table

Column name	Data Type	Not Null	Default Value
color_id	NUMBER	Yes	
color_number	NUMBER	Yes	0
color_name	VARCHAR2		

## 5 INSERT statements examples (NULL & Default)

```
INSERT INTO color_sample (color_id, color_number)
VALUES (1, 606)
```

```
INSERT INTO color_sample (color_id, color_name)
VALUES (2, 'Yellow')
```

```
INSERT INTO color_sample
VALUES (3, DEFAULT, 'Orange')
```

```
INSERT INTO color_sample
VALUES (4, 808, NULL)
```

```
INSERT INTO color_sample
VALUES (5, DEFAULT, NULL)
```

COLOR_ID	COLOR_NUMBER	COLOR_NAME
1	606	(null)
2	0	Yellow
3	0	Orange
4	808	(null)
5	0	(null)

The Color\_Sample table after the rows are inserted

## The syntax of the INSERT statement for inserting rows selected from another table

```
INSERT [INTO] table_name [(column_list)]
SELECT column_list
FROM table_source
[WHERE search_condition]
```

An INSERT statement that inserts paid invoices in the Invoices table into the Invoice\_Archive table

```
INSERT INTO invoice_archive
SELECT *
FROM invoices
WHERE invoice_total - payment_total - credit_total = 0
(74 rows inserted)
```

Commonly used when copying data from one table to another that have similar data structure.

## The same INSERT statement with a column list

```
INSERT INTO invoice_archive
  (invoice_id, vendor_id, invoice_number,
   invoice_total, credit_total,
   payment_total, terms_id, invoice_date,
   invoice_due_date)
SELECT
  invoice_id, vendor_id, invoice_number, invoice_total,
  credit_total, payment_total, terms_id,
  invoice_date, invoice_due_date
FROM invoices
WHERE invoice_total - payment_total - credit_total = 0
(74 rows inserted)
```

Commonly used when copying data from one table to another that have similar data structure.

## The syntax of the UPDATE statement

```
UPDATE table_name
SET column_name_1 = expression_1 [, column_name_2 = expression_2]...
[WHERE search_condition]
```

## An UPDATE statement that assigns new values to two columns of a single row in the Invoices table

```
UPDATE invoices
SET payment_date = '21-SEP-14',
    payment_total = 19351.18
WHERE invoice_number = '97/522'
(1 rows updated)
```

## An UPDATE statement that assigns a new value to one column of all invoices for a vendor

```
UPDATE invoices
SET terms_id = 1
WHERE vendor_id = 95
(6 rows updated)
```

## An UPDATE statement that uses an arithmetic expression to assign a value to a column

```
UPDATE invoices
SET credit_total = credit_total + 100
WHERE invoice_number = '97/522'
(1 rows updated)
```

### Warning

- If you omit the WHERE clause, all rows in the table will be updated.

## An UPDATE statement that assigns the maximum due date in the Invoices table to a specific invoice

```
UPDATE invoices
SET credit_total = credit_total + 100,
    invoice_due_date =
        (SELECT MAX(invoice_due_date)
         FROM invoices)
WHERE invoice_number = '97/522'
(1 rows updated)
```

## An UPDATE statement that updates all invoices for a vendor based on the vendor's name

```
UPDATE invoices
SET terms_id = 1
WHERE vendor_id =
    (SELECT vendor_id
     FROM vendors
     WHERE vendor_name = 'Pacific Bell')
(6 rows updated)
```

## An UPDATE statement that changes the terms of all invoices for vendors in three states

```
UPDATE invoices
SET terms_id = 1
WHERE vendor_id IN
  (SELECT vendor_id
   FROM vendors
   WHERE vendor_state IN ('CA', 'AZ', 'NV'))
(51 rows updated)
```

## The syntax of the DELETE statement

```
DELETE [FROM] table_name  
[WHERE search_condition]
```

### A DELETE statement that deletes one row

```
DELETE FROM invoice_line_items  
WHERE invoice_id = 100 AND invoice_sequence = 1  
(1 rows deleted)
```

### A DELETE statement that deletes four rows

```
DELETE FROM invoice_line_items  
WHERE invoice_id = 100  
(4 rows deleted)
```

## A DELETE statement that uses a subquery to delete all invoice line items for a vendor

```
DELETE FROM invoice_line_items
WHERE invoice_id IN
  (SELECT invoice_id
   FROM invoices
   WHERE vendor_id = 115)
(4 rows deleted)
```

### Warning

- If you omit the WHERE clause from a DELETE statement, all the rows in the table will be deleted.

# MIS 381N INTRO. TO DATABASE MANAGEMENT

---

Select

**Tayfun Keskin**

Visiting Clinical Professor, The University of Texas at Austin, McCombs School of Business  
Associate Teaching Professor, University of Washington Seattle, Foster School of Business

# QUESTIONS

Any questions  
before we begin ...



# AGENDA



Lecture

Select



Hands-On

Exercises



Looking Forward

Homework 3  
Quiz 3



The University of Texas at Austin  
McCombs School of Business

# Select



The University of Texas at Austin  
McCombs School of Business

# **SIX STEPS (WHICH TRANSLATE INTO THE SIX CLAUSES OF THE SELECT STATEMENT)**

- What do you want to display?
- Where are you getting the data?
- Are there any conditions on attributes or join conditions?
- Do you need to group by any attribute?
- Are there any conditions on the aggregate functions?
- Do you want the results to appear in a certain order?

# SIX STEPS (WHICH TRANSLATE INTO THE SIX CLAUSES OF THE SELECT STATEMENT)

- What do you want to display? **SELECT**
- Where are you getting the data? **FROM**
- Are there any conditions on attributes or join conditions? **WHERE**
- Do you need to group by any attribute? **GROUP BY**
- Are there any conditions on the aggregate functions? **HAVING**
- Do you want the results to appear in a certain order? **ORDER BY**



**SELECT**

Columns

- Column names
- Arithmetic expressions
- Literals (text or numeric)
- Scalar functions

**FROM**

Table or view names

**WHERE**

Conditions (qualifies rows)

**ORDER BY**

Sorts result rows



The University of Texas at Austin  
McCombs School of Business

SELECT	Columns - Column names - Arithmetic expressions - Literals (text or numeric) - Scalar functions - “Aggregate” functions
FROM	Table or view names
WHERE	Conditions (qualifies rows)
ORDER BY	Sorts result rows
<b>GROUP BY</b>	<b>Creates sub totals in conjunction with column functions</b>
<b>HAVING</b>	<b>Conditions the sub totals</b>



# LET'S TEST OUR UNDERSTANDING

How do you select all columns from a table?

How do you select specific columns from a table?

How do you filter records?

How do you sort records

# LOOKING FORWARD

Read Chapters 3 and 7

Quiz 3

Homework 3



The University of Texas at Austin  
McCombs School of Business

# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# **BACKUP SLIDES**



The University of Texas at Austin  
McCombs School of Business

# PART 1

Chapter 3

Select



The University of Texas at Austin  
McCombs School of Business

# What are 4 main parts of a SELECT statement?

- SELECT
- FROM
- WHERE
- ORDER BY

```
SELECT column_list  
FROM table_source  
[WHERE filter_condition]  
[ORDER BY order_by_list]
```

## Knowledge check

- How do you select all columns from a table?
- How do you select specific columns from a table?
- How do you filter records?
- How do you sort records?

## SELECT syntax

```
SELECT column_list
  FROM table_source
  [WHERE filter_condition]
  [ORDER BY order_by_list]
```

### e.g. SELECT all columns

```
SELECT *
  FROM invoices
```

### e.g. SELECT particular columns

```
SELECT invoice_number, invoice_date
  FROM invoices
```

### e.g. Sort with ORDER BY clause

```
SELECT invoice_number, invoice_date
  FROM invoices
 ORDER by invoice_date [DESC]
```

### e.g. Select with WHERE clause

```
SELECT invoice_number, invoice_date
  FROM invoices
 WHERE invoice_date = '02-MAY-2014'
```

## Ways to specify columns in a query's SELECT clause

- All columns** in base table
- Column name** in base table
- String Expression**
  - Use the string operator || to concatenate column or literal strings together
  - Literal strings are defined by you like the following:
  - **Select city || ',' || state** or **Select 'Mr. or Mrs.' || last\_name**
- Arithmetic Expression**
  - Use +, -, /, \* to do arithmetic calcs on numeric columns
  - e.g. **Select invoice\_total - payment\_total**
- Alias** – give custom column names for calculated columns
  - e.g. **Select (invoice\_total - payment\_total) as "Amount Owed"**
  - NOTE: Include Quotes (not ticks) around column alias if your column name has spaces or includes spaces. Good rule of thumb is to always use **AS w/ no spaces**

## How to include apostrophes in literal values

```
SELECT vendor_name || '''s address: ',  
      vendor_city || ',',  
      || vendor_state  
      || ''  
      || vendor_zip_code  
  
FROM vendors
```

VENDOR_NAME  "ADDRESS:"	VENDOR_CITY  ','  VENDOR_STATE  "  VENDOR_ZIP_CODE
1 Data Reproductions Corp's address:	Auburn Hills, MI 48326
2 Executive Office Products's address:	Fresno, CA 93710
3 Leslie Company's address:	Olathe, KS 66061
4 Retirement Plan Consultants's address:	Fresno, CA 93704
5 Simon Direct Inc's address:	East Brunswick, NJ 08816

## Terms to know

- String expression
- Literal value
- String literal (string constant)
- Concatenation operator

## The arithmetic operators in order of precedence

- \* Multiplication
- / Division
- + Addition
- Subtraction

## A SELECT statement that calculates balance due

```
SELECT invoice_total, payment_total, credit_total,  
       invoice_total - payment_total - credit_total  
             AS balance_due  
FROM invoices
```

	INVOICE_TOTAL	PAYMENT_TOTAL	CREDIT_TOTAL	BALANCE_DUE
1	116.54	116.54	0	0
2	1083.58	1083.58	0	0
3	20551.18	0	1200	19351.18
4	26881.4	26881.4	0	0
5	936.93	936.93	0	0

## Practice String/Numeric Expressions & Column Alias:

1. Select all vendors from NY but concatenate Address into this format: **Address; City, State Zip**  
Don't include alias this time
2. Add a column alias to this newly created column called "**Vendor Address**"
3. Pull ***vendor\_id, invoice total, payment total, and difference of invoice total & payment total.***  
Rename calculate calculated to "Amount Owed"

**String expression to derive full name**

```
SELECT first_name || ' ' || last_name
```

**String expression to derive full name w/ alias**

```
SELECT first_name || ' ' || last_name as "Full Name"
```

**Arithmetic expression to calculate a value**

```
SELECT invoice_number,  
       invoice_total - payment_total - credit_total  
FROM invoices
```

# Order of operations

## A SELECT statement that uses parentheses

```
SELECT invoice_id,  
       invoice_id + 7 * 3 AS order_of_precedence,  
       (invoice_id + 7) * 3 AS add_first  
FROM invoices  
ORDER BY invoice_id
```

INVOICE_ID	ORDER_OF_PRECEDENCE	ADD_FIRST
1	1	22
2	2	23
3	3	24
4	4	25
5	5	26

# Scalar functions

- **Scalar** = Operates on a single value and returns a single value
  
- **SYSDATE** – returns today's date/time. Like NOW() function in Excel
- **ROUND** – round decimals to whole numbers
- **SUBSTR** – Returns certain part of a string. Like MID() function in Excel
- **TO\_CHAR** – convert number/date to string
- **TO\_DATE** – convert string to a date
- **MOD** – returns remainder of division of two numbers

## A SELECT statement that uses the SYSDATE and ROUND functions

```
SELECT invoice_date,  
       SYSDATE AS today,  
       ROUND(SYSDATE - invoice_date)  AS invoice_age_in_days  
FROM invoices
```

	INVOICE_DATE	TODAY	INVOICE_AGE_IN_DAYS
1	18-JUL-14	19-JUL-14	1
2	20-JUN-14	19-JUL-14	29
3	14-JUN-14	19-JUL-14	35

## ROUND(number to found [,number of decimals])

```
SELECT 3.33333,  
       ROUND(3.3333),  
       ROUND(3.3333,1)  
FROM dual
```

	3.33333	ROUND(3.3333)	ROUND(3.3333,1)
1	3.33333	3	3.3

## A SELECT statement that uses SUBSTR

```
SELECT vendor_contact_first_name,  
vendor_contact_last_name,  
    SUBSTR(vendor_contact_first_name, 1, 1) ||  
    SUBSTR(vendor_contact_last_name, 1, 1) AS initials  
FROM vendors
```

	VENDOR_CONTACT_FIRST_NAME	VENDOR_CONTACT_LAST_NAME	INITIALS
1	Cesar	Arodondo	CA
2	Rachael	Danielson	RD
3	Zev	Alondra	ZA
4	Salina	Edgardo	SE
5	Daniel	Bradlee	DB

## A SELECT statement that uses TO\_CHAR

```
SELECT 'Invoice: #'  
       || invoice_number  
       || ', dated '  
       || TO_CHAR(payment_date, 'MM/DD/YYYY')  
       || ' for $'  
       || TO_CHAR(payment_total)  
  AS "Invoice Text"  
  
FROM invoices
```

Invoice Text
1 Invoice: # QP58872, dated 04/11/2014 for \$116.54
2 Invoice: # Q545443, dated 05/14/2014 for \$1083.58
3 Invoice: # P-0608, dated for \$0
4 Invoice: # P-0259, dated 05/12/2014 for \$26881.4
5 Invoice: # MAB01489, dated 05/13/2014 for \$936.93

## A SELECT statement that uses the MOD function

```
SELECT invoice_id,  
       MOD(invoice_id, 10) AS Remainder  
FROM invoices  
ORDER BY invoice_id
```

INVOICE_ID	REMAINDER
9	9
10	0
11	1

## A SELECT statement that returns all rows

```
SELECT vendor_city, vendor_state
FROM vendors
ORDER BY vendor_city
```

VENDOR_CITY	VENDOR_STATE
1 Anaheim	CA
2 Anaheim	CA
3 Ann Arbor	MI
4 Auburn Hills	MI
5 Boston	MA

(122 rows selected)

## A SELECT statement with no duplicate rows

```
SELECT DISTINCT vendor_city, vendor_state
FROM vendors
ORDER BY vendor_city
```

VENDOR_CITY	VENDOR_STATE
1 Anaheim	CA
2 Ann Arbor	MI
3 Auburn Hills	MI
4 Boston	MA
5 Brea	CA

(53 rows selected)

## A SELECT statement that uses the ROWNUM pseudo column to limit the number of rows

```
SELECT vendor_id, invoice_total  
FROM invoices  
WHERE ROWNUM <= 5
```

	VENDOR_ID	INVOICE_TOTAL
1	34	116.54
2	34	1083.58
3	110	20551.18
4	110	26881.4
5	81	936.93

## A SELECT statement that sorts the result set after the WHERE clause

```
SELECT vendor_id, invoice_total
FROM invoices
WHERE ROWNUM <= 5
ORDER BY invoice_total DESC
```

VENDOR_ID	INVOICE_TOTAL	
1	110	26881.4
2	110	20551.18
3	34	1083.58
4	81	936.93
5	34	116.54

## A SELECT statement that sorts the result set before the WHERE clause

```
SELECT vendor_id, invoice_total
FROM (SELECT * FROM invoices
      ORDER BY invoice_total DESC)
WHERE ROWNUM <= 5
```

VENDOR_ID	INVOICE_TOTAL	
1	110	37966.19
2	110	26881.4
3	110	23517.58
4	72	21842
5	110	20551.18

## A SELECT statement that uses the Dual table

```
SELECT 'test' AS test_string,  
       10-7   AS test_calculation,  
       SYSDATE AS test_date  
FROM Dual
```

	TEST_STRING	TEST_CALCULATION	TEST_DATE
1	test	3	28-MAY-14

# MIS 381N INTRO. TO DATABASE MANAGEMENT

---

SQL Essentials

**WHERE, ORDER BY, JOIN**

**Tayfun Keskin**

Visiting Clinical Professor, The University of Texas at Austin, McCombs School of Business  
Associate Teaching Professor, University of Washington Seattle, Foster School of Business

# QUESTIONS

Any questions  
before we begin ...



# AGENDA



Lecture

WHERE, ORDER BY  
FROM, JOIN



Hands-On

Exercises



Looking Forward

Homework 3  
Harvard case & article



The University of Texas at Austin  
McCombs School of Business

# First, a little review

# USING COLUMN FUNCTIONS

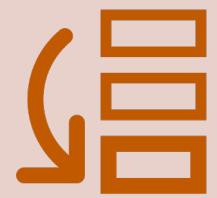
- **String Expression**
  - Use the string operator || to concatenate column or literal strings together
  - Select city || ',' || state or Select 'Mr. or Mrs.' || last\_name
- **Arithmetic Expression**
  - Use +, -, /, \* to do arithmetic calcs on numeric columns
  - e.g. Select invoice\_total – payment\_total
- **Alias – give custom column names for calculated columns**
  - e.g. Select (invoice\_total – payment\_total) AS “Amount Owed”
  - NOTE: Include Quotes (not ticks) around column alias if your column name has spaces or includes spaces. Good rule of thumb is to always use AS and quotes

# TYPES OF COLUMN FUNCTIONS

- Scalar = Operates on a single value and returns a single value
- SYSDATE – returns today's date/time. Like NOW() function in Excel
- ROUND – round decimals to whole numbers
- SUBSTR – Returns certain part of a string. Like MID() function in Excel
- TO\_CHAR – convert number/date to string
- TO\_DATE – convert string to a date
- MOD – returns remainder of division of two numbers

# DISTINCT AND ROWNUM

- DISTINCT – finds the unique occurrence of a value
- Find all the city and states where our vendors are from
- ROWNUM – pseudo column that can be used to limit the rows that are retrieved
- Retrieve the first five rows from the vendors table



**ORDER BY**



**WHERE**



**FOSTER**  
The University of Texas at Austin  
**SCHOOL OF BUSINESS**  
McCombs School of Business

# QUESTION

Is ordering and filtering required in queries?

Which commands are mandatory in a query?

# IN A NUTSHELL: SQL ESSENTIALS

- WHERE is a row filter command
- ORDER BY is a column sort command
- FETCH/OFFSET is used for filtering after ORDER BY



# WHERE CLAUSES

- WHERE <expression1> <operator> <expression2>  
Operator can be =, >, <, <=, >=, <>
- Boolean or logical operators AND, OR, NOT
- WHERE <expression1>  
[NOT] BETWEEN begin\_expression AND end\_expression
- WHERE <expression>  
[NOT] IN ({subquery|expression\_1 [, expression\_2]...})
- WHERE <expression> [NOT] LIKE pattern  
Wildcard symbols %, \_
- IS NULL and IS NOT NULL



# COMPOUND CONDITIONS

## A compound condition without parentheses

```
SELECT invoice_number, invoice_date, invoice_total  
FROM invoices  
WHERE invoice_date > '01-MAY-2014' OR invoice_total > 500  
      AND invoice_total - payment_total - credit_total > 0  
ORDER BY invoice_number
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 0-2058	08-MAY-14	37966.19
2 0-2060	08-MAY-14	23517.58
3 0-2436	07-MAY-14	10976.06

(91 rows selected)

## The order of precedence for compound conditions

1. NOT
2. AND
3. OR

## The same compound condition with parentheses

```
WHERE (invoice_date > '01-MAY-2014'  
      OR invoice_total > 500)  
      AND invoice_total - payment_total - credit_total > 0  
ORDER BY invoice_number
```

Tip: Just use parenthesis to be sure



The University of Texas at Austin  
McCombs School of Business

# ORDER OF OPERATIONS

ORDER	CLAUSE	FUNCTION
1	from	Choose and join tables to get base data.
2	where	Filters the base data.
3	group by	Aggregates the base data.
4	having	Filters the aggregated data.
5	select	Returns the final data.
6	order by	Sorts the final data.
7	limit	Limits the returned data to a row count.



# LOOKING FORWARD

Read Chapters 3 and 7

Quiz 3

Homework 3

Harvard case and article



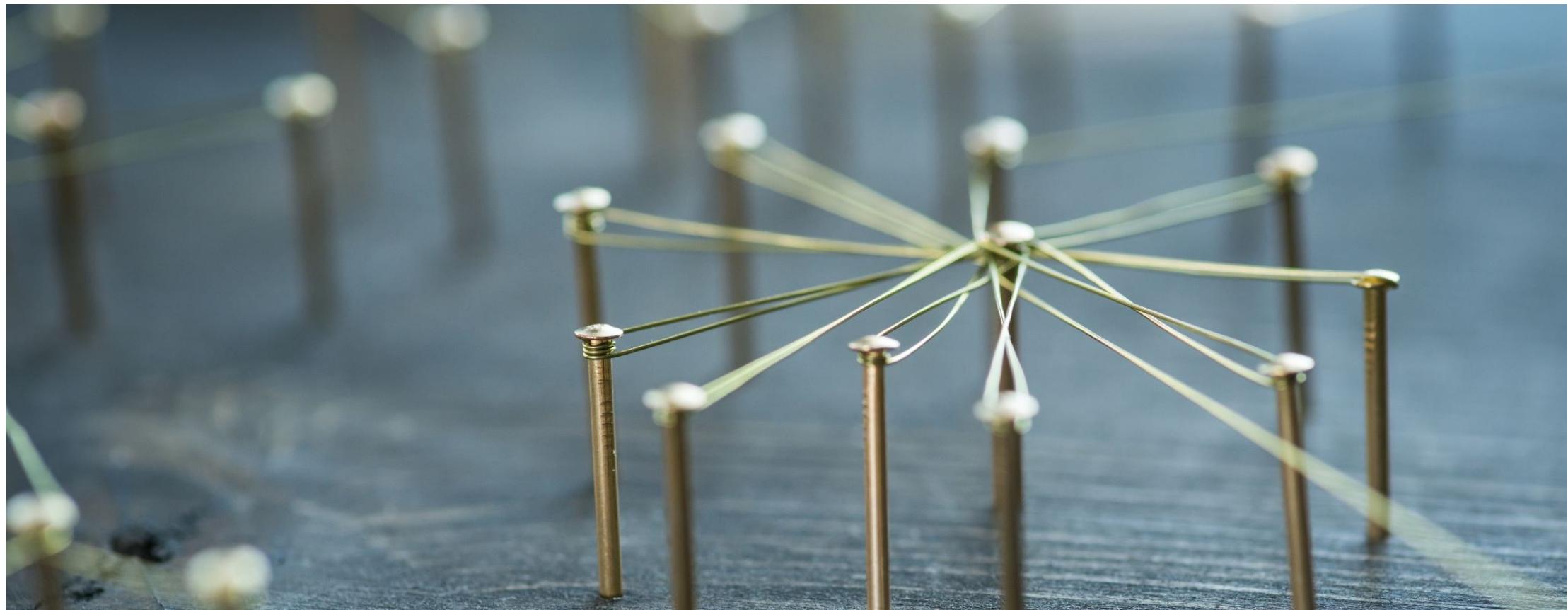
The University of Texas at Austin  
McCombs School of Business

# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# SELECT USING JOINS



The University of Texas at Austin  
McCombs School of Business

# QUESTION

When could we want to join tables?

Hint: think about vendor and their invoices



FROM

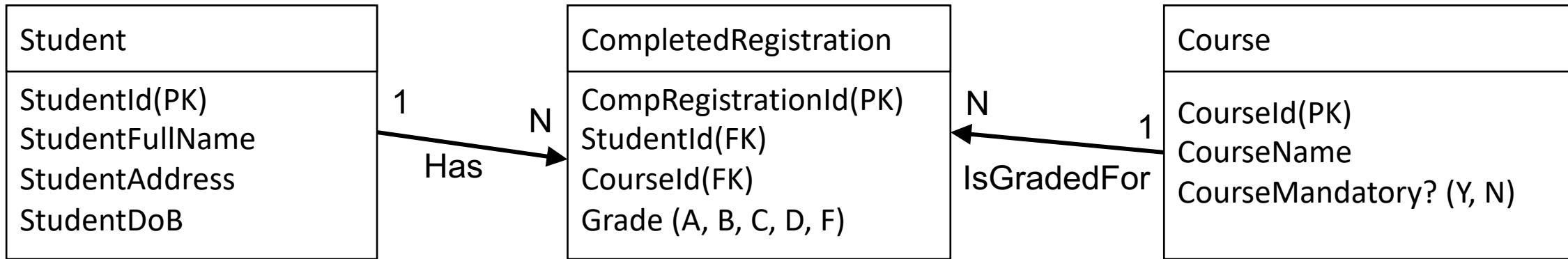


JOIN



FOSTER  
The University of Texas at Austin  
SCHOOL OF BUSINESS  
McCombs School of Business

# SOMETIMES THE DATA WE NEED IS IN MULTIPLE TABLES



- How can we retrieve **StudentFullName**, **CourseName** and **Grade**?
- How can we retrieve data correctly from the **Student**, **Course** and **CompletedRegistration** tables?
- We need to look for columns that are the same in multiple tables – the primary and foreign keys?



# JOIN SYNTAX FROM CLAUSE

```
SELECT column_list  
FROM table_1 INNER JOIN table_2 ON join_condition_1
```

```
SELECT StudentFullName, CourseId, Grade  
FROM CompletedRegistration INNER JOIN Student  
    ON CompletedRegistration.StudentId = Student.StudentId
```

```
SELECT StudentFullName, CourseName, Grade  
FROM CompletedRegistration INNER JOIN Student  
    ON CompletedRegistration.StudentId = Student.StudentId  
INNER JOIN Course  
    ON CompletedRegistration.CourseId = Course.CourseID
```



# JOIN SYNTAX TABLE ALIASES

```
SELECT column_list  
FROM table_1 INNER JOIN table_2 ON join_condition_1
```

```
SELECT StudentFullName, CourseId, Grade  
FROM CompletedRegistration CR INNER JOIN Student S  
    ON CR.StudentId = S.StudentId
```

```
SELECT StudentFullName, CourseName, Grade  
FROM CompletedRegistration CR INNER JOIN Student S  
    ON CR.StudentId = S.StudentId  
INNER JOIN Course C  
    ON CR.CourseId = C.CourseID
```



# JOIN SYNTAX – PRECISELY DEFINING COLUMNS

```
SELECT StudentId, StudentFullName, CourseName, Grade  
FROM CompletedRegistration CR INNER JOIN Student S  
    ON CR.StudentId = S.StudentId  
INNER JOIN Course C  
    ON CR.CourseId = C.CourseID
```

ORA-00918: column ambiguously defined  
00918. 00000 - "column ambiguously defined"

# JOIN SYNTAX – PRECISELY DEFINING COLUMNS

```
SELECT CR.StudentId, S.StudentFullName, C.CourseName, CR.Grade  
FROM CompletedRegistration CR INNER JOIN Student S  
      ON CR.StudentId =           S.StudentId  
INNER JOIN Course C  
      ON CR.CourseId = C.CourseID
```

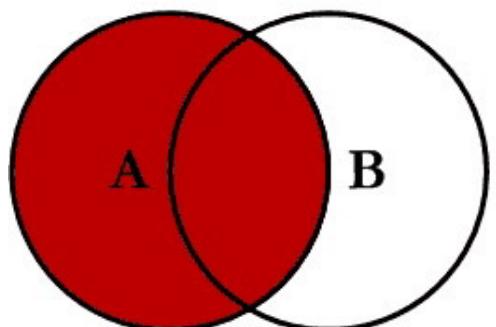


# JOIN SYNTAX – PRECISELY DEFINING COLUMNS

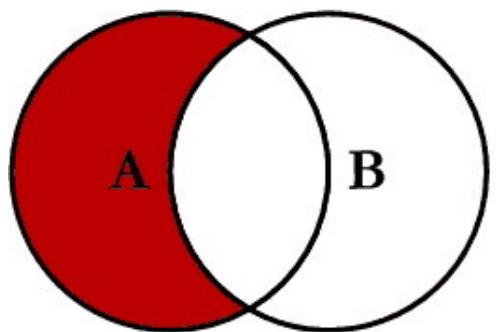
```
SELECT S.StudentFullName, C.CourseName, CR.Grade  
FROM CompletedRegistration CR, Student S, Course C  
WHERE CR.StudentId = S.StudentId  
      CR.CourseId = C.CourseID
```

NOT RECOMMENDED  
Correct syntax but not best practice

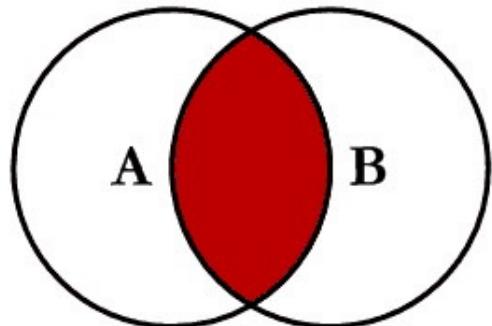
# SQL JOINS



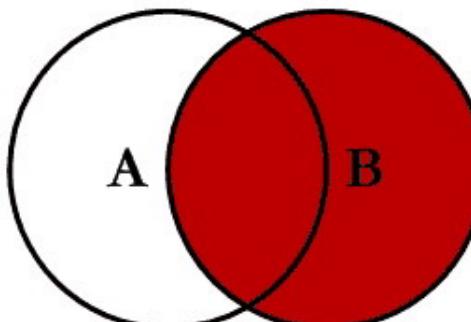
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



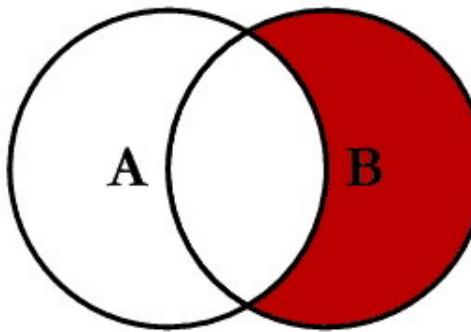
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



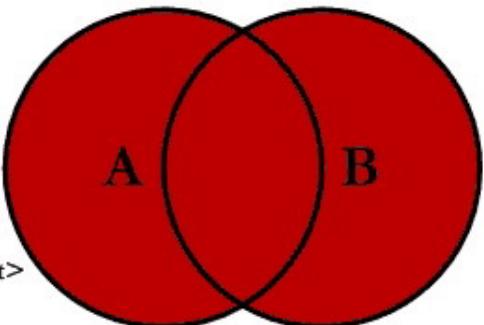
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



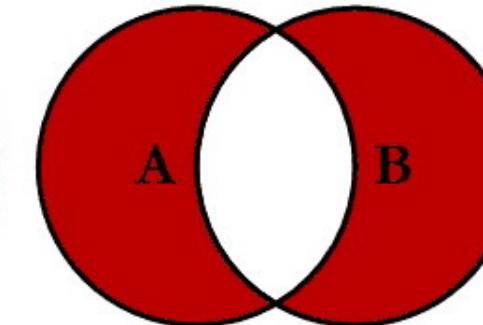
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```



important

# LOOKING FORWARD

Read Chapters 3 and 7

Homework 3

Harvard case and article



# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# **BACKUP SLIDES**



The University of Texas at Austin  
McCombs School of Business

# PART 1

Chapter 3

WHERE, ORDER BY



The University of Texas at Austin  
McCombs School of Business

## SELECT syntax

```
SELECT column_list
FROM table_source
[WHERE filter_condition]
[ORDER BY order_by_list]
```

### e.g. Sort with ORDER BY clause

```
SELECT invoice_number, invoice_date
FROM invoices
ORDER by invoice_date [DESC]
```

### e.g. Select with WHERE clause

```
SELECT invoice_number, invoice_date
FROM invoices
WHERE invoice_date = '02-MAY-2014'
```

# Practice Expressions in WHERE clause:

## e.g. Arithmetic expression in WHERE clause

```
SELECT invoice_number,  
       (invoice_total - payment_total) As "Money Due"  
FROM invoices  
WHERE (invoice_total - payment_total) = 0
```

## The syntax of the WHERE clause with the IN operator

```
WHERE test_expression
  [NOT] IN ({subquery|expression_1 [, expression_2]...})
```

### Examples of the IN operator

#### The IN operator with a list of numeric literals

```
WHERE terms_id IN (1, 3, 4)
```

#### The IN operator preceded by NOT

```
WHERE vendor_state NOT IN ('CA', 'NV', 'OR')
```

#### The IN operator with a subquery

```
WHERE vendor_id IN
  (SELECT vendor_id
    FROM invoices
   WHERE invoice_date = '01-MAY-2014')
```

## The syntax of the WHERE clause with the BETWEEN operator

```
WHERE test_expression
      [NOT] BETWEEN begin_expression AND end_expression
```

## Examples of the BETWEEN operator

### The BETWEEN operator with literal values

```
WHERE invoice_date
      BETWEEN '01-MAY-2014' AND '31-MAY-2014'
```

### The BETWEEN operator preceded by NOT

```
WHERE vendor_zip_code NOT BETWEEN 93600 AND 93799
```

### The BETWEEN operator with a calculated value

```
WHERE invoice_total - payment_total - credit_total
      BETWEEN 200 AND 500
```

### The BETWEEN operator with upper and lower limits

```
WHERE invoice_due_date BETWEEN SYSDATE AND (SYSDATE + 30)
```

## A SELECT statement that sorts the result set after the WHERE clause

```
SELECT vendor_id, invoice_total
FROM invoices
WHERE ROWNUM <= 5
ORDER BY invoice_total DESC
```

VENDOR_ID	INVOICE_TOTAL	
1	110	26881.4
2	110	20551.18
3	34	1083.58
4	81	936.93
5	34	116.54

## A SELECT statement that sorts the result set before the WHERE clause

```
SELECT vendor_id, invoice_total
FROM (SELECT * FROM invoices
      ORDER BY invoice_total DESC)
WHERE ROWNUM <= 5
```

VENDOR_ID	INVOICE_TOTAL	
1	110	37966.19
2	110	26881.4
3	110	23517.58
4	72	21842
5	110	20551.18

## The syntax of the WHERE clause with comparison operators

```
WHERE expression_1 operator expression_2
```

### The comparison operators

- =
- >
- <
- <=
- >=
- ◊

## The comparison operators

- =
- >
- <
- <=
- >=
- <>

## Examples of WHERE clauses that retrieve...

### Vendors located in Iowa

```
WHERE vendor_state = 'IA'
```

### Invoices with a balance due (two variations)

```
WHERE (invoice_total - payment_total) > 0
```

```
WHERE invoice_total > (payment_total + credit_total)
```

### Vendors with names from A to L

```
WHERE vendor_name < 'M'
```

### Invoices on or before a specified date

```
WHERE invoice_date <= '31-MAY-14'
```

### Invoices on or after a specified date

```
WHERE invoice_date >= '01-MAY-14'
```

### Invoices with credits that don't equal zero

```
WHERE credit_total <> 0
```

## A compound condition without parentheses

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_date > '01-MAY-2014' OR invoice_total > 500
      AND invoice_total - payment_total - credit_total > 0
ORDER BY invoice_number
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 0-2058	08-MAY-14	37966.19
2 0-2060	08-MAY-14	23517.58
3 0-2436	07-MAY-14	10976.06

(91 rows selected)

### The order of precedence for compound conditions

1. NOT
2. AND
3. OR

## The same compound condition with parentheses

```
WHERE (invoice_date > '01-MAY-2014'
      OR invoice_total > 500)
      AND invoice_total - payment_total - credit_total > 0
ORDER BY invoice_number
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 0-2436	07-MAY-14	10976.06
2 109596	14-JUN-14	41.8
3 111-92R-10092	04-JUN-14	46.21

(39 rows selected)

**Tip: Just use parenthesis to be sure**

## The syntax of the WHERE clause with the LIKE operator

```
WHERE match_expression [NOT] LIKE pattern
```

### Wildcard symbols

%

\_

## WHERE clauses that use the LIKE operator

### Example 1

```
WHERE vendor_city LIKE 'SAN%'
```

#### Cities that will be retrieved

“San Diego” and “Santa Ana”

### Example 2

```
WHERE vendor_name LIKE 'COMPU_ER%'
```

#### Vendors that will be retrieved

“Compuserve” and “Computerworld”

## The syntax of the WHERE clause with the Is null condition

```
WHERE expression IS [NOT] NULL
```

### The contents of the Null\_Sample table

```
SELECT *
FROM null_sample
```

	INVOICE_ID	INVOICE_TOTAL
1	1	125
2	2	0
3	3	(null)
4	4	2199.99
5	5	0

## The expanded syntax of the ORDER BY clause

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC]]...
```

### An ORDER BY clause that sorts by one column

```
SELECT vendor_name,  
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code  
AS address  
FROM vendors  
ORDER BY vendor_name
```

VENDOR_NAME	ADDRESS
1 ASC Signs	Fresno, CA 93703
2 AT&T	Phoenix, AZ 85062
3 Abbey Office Furnishings	Fresno, CA 93722

### An ORDER BY clause that sorts by three columns

```
SELECT vendor_name,  
       vendor_city || ', ' || vendor_state || ' ' ||  
       vendor_zip_code AS address  
FROM vendors  
ORDER BY vendor_state, vendor_city, vendor_name
```

VENDOR_NAME	ADDRESS
1 AT&T	Phoenix, AZ 85062
2 Computer Library	Phoenix, AZ 85023
3 Wells Fargo Bank	Phoenix, AZ 85038
4 Aztek Label	Anaheim, CA 92807
5 Blue Shield of California	Anaheim, CA 92850
6 Diversified Printing & Pub	Brea, CA 92621
7 ASC Signs	Fresno, CA 93703

## An ORDER BY clause that uses an alias

```
SELECT vendor_name,  
       vendor_city || ', ' || vendor_state || ' ' ||  
             vendor_zip_code AS address  
FROM vendors  
ORDER BY address, vendor_name
```

VENDOR_NAME	ADDRESS
1 Aztek Label	Anaheim, CA 92807
2 Blue Shield of California	Anaheim, CA 92850
3 Malloy Lithographing Inc	Ann Arbor, MI 48106
4 Data Reproductions Corp	Auburn Hills, MI 48326

## An ORDER BY clause that uses an expression

```
SELECT vendor_name,  
       vendor_city || ', ' || vendor_state || ' ' ||  
       vendor_zip_code AS address  
FROM vendors  
ORDER BY vendor_contact_last_name  
        || vendor_contact_first_name
```

VENDOR_NAME	ADDRESS
1 Dristas Groom & McCormick	Fresno, CA 93720
2 Internal Revenue Service	Fresno, CA 93888
3 US Postal Service	Madison, WI 53707
4 Yale Industrial Trucks-Fresno	Fresno, CA 93706

## An ORDER BY clause that uses column positions

```
SELECT vendor_name,  
       vendor_city || ', ' || vendor_state || ' ' ||  
             vendor_zip_code AS address  
FROM vendors  
ORDER BY 2, 1
```

VENDOR_NAME	ADDRESS
1 Aztek Label	Anaheim, CA 92807
2 Blue Shield of California	Anaheim, CA 92850
3 Malloy Lithographing Inc	Ann Arbor, MI 48106
4 Data Reproductions Corp	Auburn Hills, MI 48326

## The syntax of the row limiting clause (12c and later)

```
[ OFFSET offset { ROW | ROWS } ]
[ FETCH { FIRST | NEXT } [ { rowcount | percent PERCENT }
{ ROW | ROWS } { ONLY | WITH TIES } ]
```

## A **FETCH** clause that retrieves the first five rows

```
SELECT vendor_id, invoice_total
FROM invoices
ORDER BY invoice_total DESC
FETCH FIRST 5 ROWS ONLY
```

	VENDOR_ID	INVOICE_TOTAL
1	110	37966.19
2	110	26881.4
3	110	23517.58
4	72	21842
5	110	20551.18

## An OFFSET clause that starts with the third row and fetches three rows

```
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
OFFSET 2 ROWS FETCH NEXT 3 ROWS ONLY
```

	INVOICE_ID	VENDOR_ID	INVOICE_TOTAL
1	3	110	20551.18
2	4	110	26881.4
3	5	81	936.93

## An OFFSET clause that starts with the 101<sup>st</sup> row

```
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
OFFSET 100 ROWS FETCH NEXT 1000 ROWS ONLY
```

	INVOICE_ID	VENDOR_ID	INVOICE_TOTAL
1	101	103	1367.5
2	102	48	856.92
3	103	95	19.67
4	104	114	290

## A SELECT statement that retrieves rows with zero values

```
SELECT *
FROM null_sample
WHERE invoice_total = 0
```

	INVOICE_ID	INVOICE_TOTAL
1	2	0
2	5	0

## A SELECT statement that retrieves rows with non-zero values

```
SELECT *
FROM null_sample
WHERE invoice_total <> 0
```

	INVOICE_ID	INVOICE_TOTAL
1	1	125
2	4	2199.99

## A SELECT statement that retrieves rows with null values

```
SELECT *
FROM null_sample
WHERE invoice_total IS NULL
```

	INVOICE_ID	INVOICE_TOTAL
1	3	(null)

## A SELECT statement that retrieves rows without null values

```
SELECT *
FROM null_sample
WHERE invoice_total IS NOT NULL
```

	INVOICE_ID	INVOICE_TOTAL
1	1	125
2	2	0
3	4	2199.99
4	5	0

## The default sequence for an ascending sort

- Special characters
- Capital letters
- Lowercase letters
- Null values

### Notes

- This causes problems when sorting mixed-case columns.
- Chapter 8 provides the solutions.

# PART 2

Chapter 4

Select Using Joins – Select from Multiple Tables



The University of Texas at Austin  
McCombs School of Business

# The explicit syntax for an inner join (2 tables only)

```
SELECT column_list
FROM table_1 INNER JOIN table_2
    ON join_condition_1
```

*NOTE: The “join\_condition” defines what columns connect the tables (e.g. table1.pk = table1.fk)*

## A SELECT statement that joins two tables

```
SELECT invoice_number, vendor_name, invoice_total, payment_total
FROM vendors INNER JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
ORDER BY invoice_number
```

## The result set

INVOICE_NUMBER	VENDOR_NAME	INVOICE_TOTAL	PAYMENT_TOTAL
1 0-2058	Malloy Lithographing Inc	37966.19	37966.19
2 0-2060	Malloy Lithographing Inc	23517.58	21221.63
3 0-2436	Malloy Lithographing Inc	10976.06	0
4 1-200-5164	Federal Express Corporation	63.4	63.4
5 1-202-2978	Federal Express Corporation	33	33
6 10843	Yesmed, Inc	4901.26	4901.26
7 109596	Coffee Break Service	41.8	0

# Inner join syntax

```
Select column_list  
FROM table1 inner join table2  
    ON table1.column = table2.column
```

e.g. Inner join 2 tables – pull all columns

```
SELECT *  
FROM invoices inner join vendors  
    ON invoices.vendor_id = vendors.vendor_id
```

Ven_id	Name	...
1	USPS	...
...	...	...
34	IBM	...
37	BCBS	...

Inv_id	Ven_id	Date	...
1	34	1/1	...
2	34	1/2	...
...	...	...	...
			...

# Using a table alias

```
SELECT column_list
FROM table_1 n1 INNER JOIN table_2 n2
    ON n1.column_name operator n2.column_name
```

## A SELECT statement that joins two tables

```
SELECT invoice_number, vendor_name, invoice_total, payment_total
FROM vendors v INNER JOIN invoices i
    ON v.vendor_id = i.vendor_id
ORDER BY invoice_number
```

ORA-00918: column ambiguously defined  
00918. 00000 - "column ambiguously defined"

```
SELECT invoice_number, vendor_name, invoice_total, payment_total, v.vendor_id
FROM vendors v INNER JOIN invoices i
    ON v.vendor_id = i.vendor_id
ORDER BY invoice_number
```

# e.g. Formatted code w/ clear aliases

```
SELECT      i.invoice_number,  
            v.vendor_name,  
            i.invoice_total,  
            i.payment_total,  
            v.vendor_id  
  
FROM        vendors v INNER JOIN invoices i  
            ON v.vendor_id = i.vendor_id  
  
ORDER BY    i.invoice_number
```

## Practice: Try combine expressions with inner join with aliases for all tables

```
SELECT    i.invoice_number,  
        v.vendor_name,  
        i.invoice_due_date,  
        (invoice_total - payment_total - credit_total) AS "balance_due"  
FROM vendors v JOIN invoices i  
    ON v.vendor_id = i.vendor_id  
WHERE (invoice_total - payment_total - credit_total) > 0  
ORDER BY invoice_due_date DESC
```

### The result set

INVOICE_NUMBER	VENDOR_NAME	INVOICE_DUE_DATE	BALANCE_DUE
1 40318	Data Reproductions Corp	20-JUL-14	21842
2 39104	Data Reproductions Corp	20-JUL-14	85.31
3 0-2436	Malloy Lithographing Inc	17-JUL-14	10976.06

(40 rows selected)

# Best Practice: Keep filtering out of FROM

An inner join with two conditions – Not best practice to include filters in FROM

```
SELECT invoice_number, invoice_date,  
       invoice_total, line_item_amt  
  FROM invoices i JOIN invoice_line_items li  
    ON (i.invoice_id = li.invoice_id) AND  
       (i.invoice_total > li.line_item_amt)  
 ORDER BY invoice_number
```

The same join with one condition in a WHERE clause – Best practice

```
SELECT invoice_number, invoice_date,  
       invoice_total, line_item_amt  
  FROM invoices i JOIN invoice_line_items li  
    ON i.invoice_id = li.invoice_id  
 WHERE i.invoice_total > li.line_item_amt  
 ORDER BY invoice_number
```

	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL	LINE_ITEM_AMT
1	97/522	30-APR-14	1962.13	765.13
2	97/522	30-APR-14	1962.13	1197
3	I77271-001	05-JUN-14	662	75.6
4	I77271-001	05-JUN-14	662	58.4

(6 rows selected)

## A SELECT statement that joins four tables

```
SELECT vendor_name, invoice_number, invoice_date,  
       line_item_amt, account_description  
FROM vendors v  
      JOIN invoices i ON v.vendor_id = i.vendor_id  
      JOIN invoice_line_items li  
        ON i.invoice_id = li.invoice_id  
      JOIN general_ledger_accounts gl  
        ON li.account_number = gl.account_number  
WHERE (invoice_total - payment_total - credit_total) > 0  
ORDER BY vendor_name, line_item_amt DESC
```

## The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_DATE	LINE_ITEM_AMT	ACCOUNT_DESCRIPTION
1 Abbey Office Furnishings	203339-13	02-MAY-14	17.5	Office Supplies
2 Blue Cross	547481328	20-MAY-14	224	Group Insurance
3 Blue Cross	547480102	19-MAY-14	224	Group Insurance
4 Blue Cross	547479217	17-MAY-14	116	Group Insurance
5 Cardinal Business Media, Inc.	134116	01-JUN-14	90.36	Card Deck Advertising
6 Coffee Break Service	109596	14-JUN-14	41.8	Meals
7 Compuserve	21-4748363	09-MAY-14	9.95	Books, Dues, and Subscriptions
8 Computerworld	367447	31-MAY-14	2433	Card Deck Advertising

(44 rows selected)

# Joining more than 2 tables

```
SELECT column_list
FROM table_1 n1 INNER JOIN table_2 n2 ON n1.column_name operator n2.column_name
                    INNER JOIN table_3 n3 ON n2.column_name = n3.column_name
```

## e.g. Joining 2+ tables with aliases

```
SELECT i.invoice_number, v.vendor_name, i.invoice_total, i.payment_total
FROM vendors v INNER JOIN invoices i ON v.vendor_id = i.vendor_id
                    INNER JOIN invoice_line_items ili ON i.invoice_id = ili.invoice_id
ORDER BY invoice_number
```

## The implicit syntax for an inner join

```
SELECT select_list
FROM table_1, table_2 [, table_3]...
WHERE table_1.column_name operator table_2.column_name
  [AND table_2.column_name operator table_3.column_name]...
```

### Implicit syntax that joins two tables – simpler but puts the join in WHERE

```
SELECT invoice_number, vendor_name
FROM vendors v, invoices i
WHERE v.vendor_id = i.vendor_id
ORDER BY invoice_number
```

### The result set

INVOICE_NUMBER	VENDOR_NAME
1 0-2058	Malloy Lithographing Inc
2 0-2060	Malloy Lithographing Inc
3 0-2436	Malloy Lithographing Inc
4 1-200-5164	Federal Express Corporation
5 1-202-2978	Federal Express Corporation

(114 rows selected)

## Implicit syntax that joins four tables

```
SELECT vendor_name, invoice_number, invoice_date,  
      line_item_amt, account_description  
FROM   vendors v, invoices i, invoice_line_items li,  
      general_ledger_accounts gl  
WHERE  v.vendor_id = i.vendor_id  
      AND i.invoice_id = li.invoice_id  
      AND li.account_number = gl.account_number  
      AND (invoice_total - payment_total - credit_total) > 0  
ORDER BY vendor_name, line_item_amt DESC
```

## The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_DATE	LINE_ITEM_AMT	ACCOUNT_DESCRIPTION
1 Abbey Office Furnishings	203339-13	02-MAY-14	17.5	Office Supplies
2 Blue Cross	547481328	20-MAY-14	224	Group Insurance
3 Blue Cross	547480102	19-MAY-14	224	Group Insurance
4 Blue Cross	547479217	17-MAY-14	116	Group Insurance
5 Cardinal Business Media, Inc.	134116	01-JUN-14	90.36	Card Deck Advertising

(44 rows selected)

## But...Best practice

- Joins in FROM
- Filters in WHERE

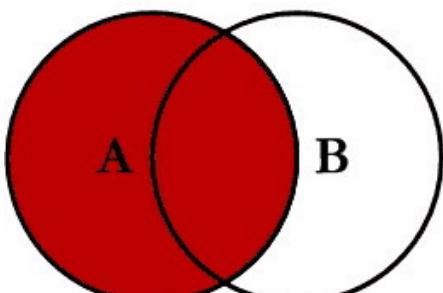
## The explicit syntax for an outer join

```
SELECT select_list
FROM table_1
    {LEFT|RIGHT|FULL} [OUTER] JOIN table_2
        ON join_condition_1
    [{LEFT|RIGHT|FULL} [OUTER] JOIN table_3
        ON join_condition_2]...
```

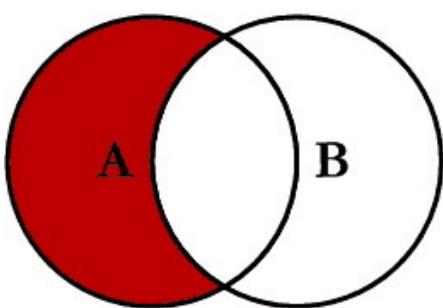
## What outer joins do

Join	Keeps unmatched rows from
Left	The left table
Right	The right table
Full	Both tables

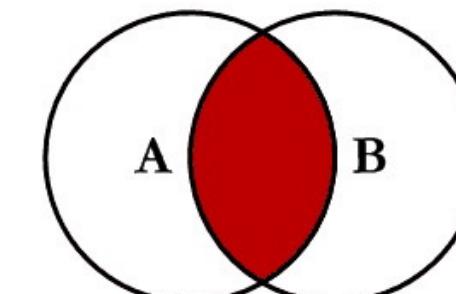
# SQL JOINS



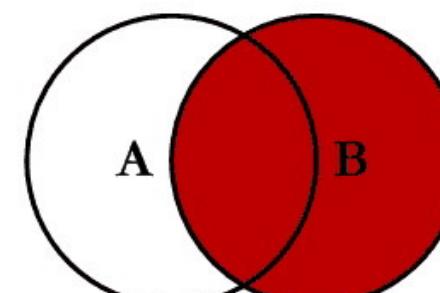
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



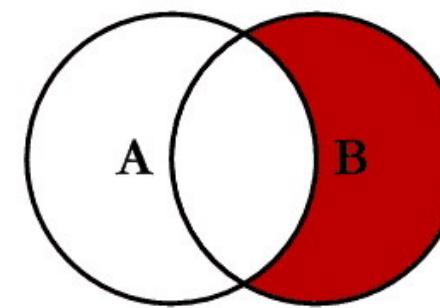
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



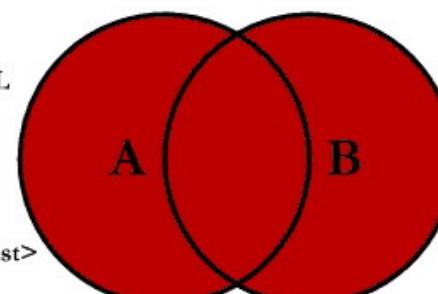
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



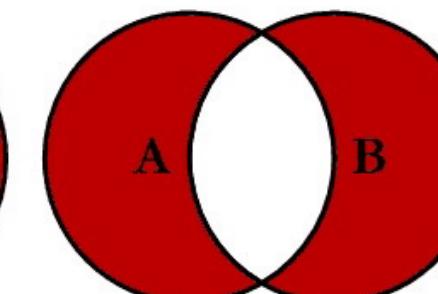
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

## A SELECT statement that uses a left outer join

```
SELECT vendor_name, invoice_number, invoice_total
FROM vendors LEFT JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
ORDER BY vendor_name
```

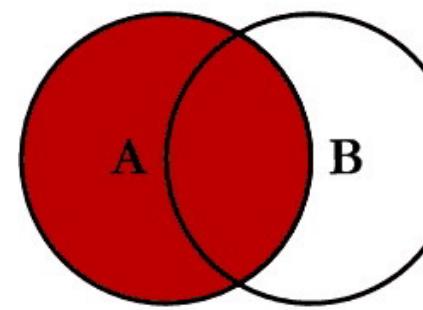
### The result set

	VENDOR_NAME	INVOICE_NUMBER	INVOICE_TOTAL
1	ASC Signs	(null)	(null)
2	AT&T	(null)	(null)
3	Abbey Office Furnishings	203339-13	17.5
4	American Booksellers Assoc	(null)	(null)
5	American Express	(null)	(null)

(202 rows selected)

## The Departments table

	DEPARTMENT_NUMBER	DEPARTMENT_NAME
1		Accounting
2		Payroll
3		Operations
4		Personnel
5		Maintenance



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

## The Employees table

	EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy		2
2	Jones	Elmer		4
3	Simonian	Ralph		2
4	Hernandez	Olivia		1
5	Aaronsen	Robert		2
6	Watson	Denise		6
7	Hardy	Thomas		5
8	O'Leary	Rhea		4
9	Locario	Paulo		6

## A left outer join

```
SELECT department_name AS dept_name,
       d.department_number AS dept_no,
       last_name
  FROM departments d
 LEFT JOIN employees e
    ON d.department_number = e.department_number
 ORDER BY department_name
```

## Query results

DEPT_NAME	DEPT_NO	LAST_NAME
1 Accounting	1 Hernandez	
2 Maintenance	5 Hardy	
3 Operations	3 (null)	
4 Payroll	2 Simonian	
5 Payroll	2 Aaronsen	
6 Payroll	2 Smith	
7 Personnel	4 Jones	
8 Personnel	4 O'Leary	

## The Departments table

DEPARTMENT_NUMBER	DEPARTMENT_NAME
1	1 Accounting
2	2 Payroll
3	3 Operations
4	4 Personnel
5	5 Maintenance

## The Employees table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

## A right outer join – Not needed

```
SELECT department_name AS dept_name,
       e.department_number AS dept_no,
       last_name
  FROM departments d
 RIGHT JOIN employees e
    ON d.department_number = e.department_number
 ORDER BY department_name
```

## Query results

DEPT_NAME	DEPT_NO	LAST_NAME
1 Accounting	1 Hernandez	
2 Maintenance	5 Hardy	
3 Payroll	2 Smith	
4 Payroll	2 Simonian	
5 Payroll	2 Aaronsen	
6 Personnel	4 O'Leary	
7 Personnel	4 Jones	
8 (null)	6 Watson	
9 (null)	6 Locario	

## The Departments table

DEPARTMENT_NUMBER	DEPARTMENT_NAME
1	Accounting
2	Payroll
3	Operations
4	Personnel
5	Maintenance

## The Employees table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

## A full outer join

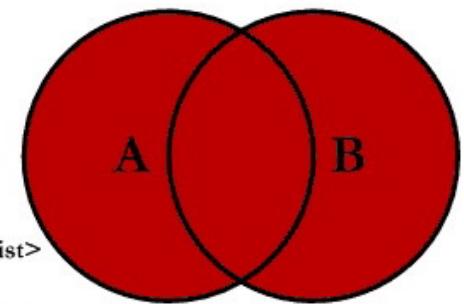
```
SELECT department_name AS dept_name,
       d.department_number AS d_dept_no,
       e.department_number AS e_dept_no,
       last_name
  FROM departments d
    FULL JOIN employees e
      ON d.department_number = e.department_number
 ORDER BY e.department_number, d.department_number
```

DEPT_NAME	D_DEPT_NO	E_DEPT_NO	LAST_NAME
1 Accounting	1	1 Hernandez	
2 Payroll	2	2 Simonian	
3 Payroll	2	2 Smith	
4 Payroll	2	2 Aaronsen	
5 Personnel	4	4 O'Leary	
6 Personnel	4	4 Jones	
7 Maintenance	5	5 Hardy	
8 (null)	(null)	6 Watson	
9 (null)	(null)	6 Locario	
10 Operations	3	(null) (null)	

## A SELECT statement that uses full outer joins

```
SELECT department_name, last_name, project_number AS proj_no
FROM departments dpt
    FULL JOIN employees emp
        ON dpt.department_number = emp.department_number
    FULL JOIN projects prj
        ON emp.employee_id = prj.employee_id
ORDER BY department_name
```

DEPARTMENT_NAME	LAST_NAME	PROJ_NO
1 Accounting	Hernandez	P1011
2 Maintenance	Hardy	(null)
3 Operations	(null)	(null)
4 Payroll	Simonian	P1012
5 Payroll	Aaronsen	P1012
6 Payroll	Smith	P1012
7 Personnel	Jones	(null)
8 Personnel	O'Leary	P1011
9 (null)	Locario	P1013
10 (null)	(null)	P1014
11 (null)	Watson	P1013



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

## Let's practice FULL OUTER join:

1. Join Orders, Order\_Items, Products, and Categories.
2. Return Order\_ID, Product\_Name, Category\_Name
3. Test: What changes when you use INNER versus OUTER join? Try both

## The syntax for a union

```
SELECT_statement_1
UNION [ALL]
  SELECT_statement_2
[UNION [ALL]
    SELECT_statement_3]...
[ORDER BY order_by_list]
```

## Rules for a union

- The number of columns must be the same in all SELECTs.
- The column data types must be compatible.
- The column names are taken from the first SELECT statement.

## e.g. A union with data from just Invoices table

```
SELECT 'Active' AS source, invoice_number, invoice_date,
       invoice_total
  FROM invoices
 WHERE (invoice_total - payment_total - credit_total) > 0
UNION
  SELECT 'Paid' AS source, invoice_number, invoice_date,
         invoice_total
  FROM invoices
 WHERE (invoice_total - payment_total - credit_total)
      <= 0
 ORDER BY invoice_total DESC
```

## The result set

SOURCE	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 Paid	0-2058	08-MAY-14	37966.19
2 Paid	P-0259	16-APR-14	26881.4
3 Paid	0-2060	08-MAY-14	23517.58
4 Active	40318	18-JUL-14	21842
5 Active	P-0608	11-APR-14	20551.18
6 Active	0-2436	07-MAY-14	10976.06

## The syntax for MINUS and INTERSECT operations

```
SELECT_statement_1
{MINUS | INTERSECT}
  SELECT_statement_2
[ORDER BY order_by_list]
```

# Self Joins: A way of finding in common records on a table

A self-join that returns vendors from cities in common with other vendors

```
SELECT DISTINCT v1.vendor_name, v1.vendor_city,  
    v1.vendor_state  
FROM vendors v1 JOIN vendors v2  
    ON (v1.vendor_city = v2.vendor_city) AND  
        (v1.vendor_state = v2.vendor_state) AND  
        (v1.vendor_id <> v2.vendor_id)  
ORDER BY v1.vendor_state, v1.vendor_city
```

## JUST FYI:

- Not going to practice this
- We'll learn a different way to do this later

The result set

VENDOR_NAME	VENDOR_CITY	VENDOR_STATE
1 AT&T	Phoenix	AZ
2 Computer Library	Phoenix	AZ
3 Wells Fargo Bank	Phoenix	AZ
4 Aztek Label	Anaheim	CA
5 Blue Shield of California	Anaheim	CA
6 ASC Signs	Fresno	CA
7 Abbey Office Furnishings	Fresno	CA
8 BFI Industries	Fresno	CA

(84 rows selected)

## A SELECT statement that uses left outer joins

```
SELECT department_name, last_name, project_number AS proj_no
FROM departments d
    LEFT JOIN employees e
        ON d.department_number = e.department_number
    LEFT JOIN projects p
        ON e.employee_id = p.employee_id
ORDER BY department_name, last_name, project_number
```

	DEPARTMENT_NAME	LAST_NAME	PROJ_NO
1	Accounting	Hernandez	P1011
2	Maintenance	Hardy	(null)
3	Operations	(null)	(null)
4	Payroll	Aaronsen	P1012
5	Payroll	Simonian	P1012
6	Payroll	Smith	P1012
7	Personnel	Jones	(null)
8	Personnel	O'Leary	P1011

## A SELECT statement that uses full outer joins

```
SELECT department_name, last_name, project_number AS proj_no
FROM departments dpt
    FULL JOIN employees emp
        ON dpt.department_number = emp.department_number
    FULL JOIN projects prj
        ON emp.employee_id = prj.employee_id
ORDER BY department_name
```

	DEPARTMENT_NAME	LAST_NAME	PROJ_NO
1	Accounting	Hernandez	P1011
2	Maintenance	Hardy	(null)
3	Operations	(null)	(null)
4	Payroll	Simonian	P1012
5	Payroll	Aaronsen	P1012
6	Payroll	Smith	P1012
7	Personnel	Jones	(null)
8	Personnel	O'Leary	P1011
9	(null)	Locario	P1013
10	(null)	(null)	P1014
11	(null)	Watson	P1013

## A SELECT statement with an outer and inner join

```
SELECT department_name AS dept_name,  
       last_name, project_number  
FROM departments dpt  
      JOIN employees emp  
        ON dpt.department_number = emp.department_number  
      LEFT JOIN projects prj  
        ON emp.employee_id = prj.employee_id  
ORDER BY department_name
```

### The result set

	DEPT_NAME	LAST_NAME	PROJECT_NUMBER
1	Accounting	Hernandez	P1011
2	Maintenance	Hardy	(null)
3	Payroll	Simonian	P1012
4	Payroll	Smith	P1012
5	Payroll	Aaronsen	P1012
6	Personnel	Jones	(null)
7	Personnel	O'Leary	P1011

(7 rows selected)

## Terms to know

- Outer join
- Left outer join
- Right outer join
- Equi-join
- Natural join
- Cross join

## The syntax for a join with the USING keyword

```
SELECT select_list
FROM table_1
[ {LEFT|RIGHT|FULL} [OUTER]] JOIN table_2
    USING(join_column_1[, join_column_2]...)
[[ {LEFT|RIGHT|FULL} [OUTER]] JOIN table_3
    USING (join_column_2[, join_column_2]...)]...
```

## A SELECT statement with the USING keyword

```
SELECT invoice_number, vendor_name
FROM vendors
    JOIN invoices USING (vendor_id)
ORDER BY invoice_number
```

## The result set

	INVOICE_NUMBER	VENDOR_NAME
1	0-2058	Malloy Lithographing Inc
2	0-2060	Malloy Lithographing Inc
3	0-2436	Malloy Lithographing Inc
4	1-200-5164	Federal Express Corporation

(114 rows selected)

## JUST FYI:

- Removes the need of the “ON column1 = column2” syntax
- Only works if joining columns have same name
- Best to just learn full syntax and stick to using “ON” keyword.

## The syntax for a join with the NATURAL keyword

```
SELECT select_list
FROM table_1
    NATURAL JOIN table_2
    [NATURAL JOIN table_3]...
```

## A SELECT statement with the NATURAL keyword

```
SELECT invoice_number, vendor_name
FROM vendors
    NATURAL JOIN invoices
ORDER BY invoice_number
```

## The result set

INVOICE_NUMBER	VENDOR_NAME
1 0-2058	Malloy Lithographing Inc
2 0-2060	Malloy Lithographing Inc
3 0-2436	Malloy Lithographing Inc
4 1-200-5164	Federal Express Corporation

(114 rows selected)

## JUST FYI:

- Simpler/lazy way to join
- Only works if there is a single column in common between the two joined tables.
- Best to just learn full syntax and stick to using “INNER” and “ON” keywords.

# How to code a cross join with the explicit syntax

## The explicit syntax for a cross join

```
SELECT select_list  
FROM table_1 CROSS JOIN table_2
```

## A cross join that uses the explicit syntax

```
SELECT departments.department_number, department_name,  
       employee_id, last_name  
  FROM departments CROSS JOIN employees  
 ORDER BY departments.department_number
```

## The result set

	DEPARTMENT_NUMBER	DEPARTMENT_NAME	EMPLOYEE_ID	LAST_NAME
1	1 Accounting		4 Hernandez	
2	1 Accounting		3 Simonian	
3	1 Accounting		9 Locario	
4	1 Accounting		8 O'Leary	
5	1 Accounting		7 Hardy	
6	1 Accounting		6 Watson	
7	1 Accounting		5 Aaronsen	

(45 rows selected)

## How to code a cross join with the implicit syntax

### The implicit syntax for a cross join

```
SELECT select_list  
FROM table_1, table_2
```

### A cross join that uses the implicit syntax

```
SELECT departments.department_number, department_name,  
       employee_id, last_name  
FROM departments, employees  
ORDER BY departments.department_number
```

### The result set

	DEPARTMENT_NUMBER	DEPARTMENT_NAME	EMPLOYEE_ID	LAST_NAME
1	1 Accounting		4 Hernandez	
2	1 Accounting		3 Simonian	
3	1 Accounting		9 Locario	
4	1 Accounting		8 O'Leary	
5	1 Accounting		7 Hardy	
6	1 Accounting		6 Watson	
7	1 Accounting		5 Aaronsen	

(45 rows selected)

### **JUST FYI:**

- Cross joins typically won't happen if you avoid use of implicit joins
- Just be aware of what a Cartesian join can look like so you can know how to correct it

## The syntax for a union

```
SELECT_statement_1
UNION [ALL]
  SELECT_statement_2
[UNION [ALL]
    SELECT_statement_3]...
[ORDER BY order_by_list]
```

## Rules for a union

- The number of columns must be the same in all SELECTs.
- The column data types must be compatible.
- The column names are taken from the first SELECT statement.

## A union with data from two different tables

```
SELECT 'Active' AS source, invoice_number, invoice_date,  
       invoice_total  
  FROM active_invoices  
 WHERE invoice_date >= '01-JUN-2014'  
  
UNION  
SELECT 'Paid' AS source, invoice_number, invoice_date,  
       invoice_total  
  FROM paid_invoices  
 WHERE invoice_date >= '01-JUN-2014'  
ORDER BY invoice_total DESC
```

## The result set

SOURCE	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 Active	40318	18-JUL-14	21842
2 Paid	P02-3772	03-JUN-14	7125.34
3 Paid	10843	04-JUN-14	4901.26
4 Paid	77290	04-JUN-14	1750
5 Paid	RTR-72-3662-X	04-JUN-14	1600
6 Paid	75C-90227	06-JUN-14	1367.5
7 Paid	P02-88D77S7	06-JUN-14	856.92
8 Active	I77271-001	05-JUN-14	662
9 Active	9982771	03-JUN-14	503.2

(22 rows selected)

## A union with data from just the Invoices table

```
SELECT 'Active' AS source, invoice_number, invoice_date,  
       invoice_total  
  FROM invoices  
 WHERE (invoice_total - payment_total - credit_total) > 0  
UNION  
  SELECT 'Paid' AS source, invoice_number, invoice_date,  
        invoice_total  
  FROM invoices  
 WHERE (invoice_total - payment_total - credit_total)  
      <= 0  
ORDER BY invoice_total DESC
```

## The result set

SOURCE	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 Paid	0-2058	08-MAY-14	37966.19
2 Paid	P-0259	16-APR-14	26881.4
3 Paid	0-2060	08-MAY-14	23517.58
4 Active	40318	18-JUL-14	21842
5 Active	P-0608	11-APR-14	20551.18
6 Active	0-2436	07-MAY-14	10976.06

(114 rows selected)

## A union with payment data from the same tables

```
SELECT invoice_number, vendor_name,
      '33% Payment' AS payment_type,
      invoice_total AS total,
      (invoice_total * 0.333) AS payment
FROM invoices JOIN vendors
  ON invoices.vendor_id = vendors.vendor_id
WHERE invoice_total > 10000
UNION
SELECT invoice_number, vendor_name,
      '50% Payment' AS payment_type,
      invoice_total AS total,
      (invoice_total * 0.5) AS payment
FROM invoices JOIN vendors
  ON invoices.vendor_id = vendors.vendor_id
WHERE invoice_total BETWEEN 500 AND 10000
```

## The union (continued)

UNION

```
SELECT invoice_number, vendor_name,
       'Full amount' AS payment_type,
       invoice_total AS Total, invoice_total AS Payment
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
   WHERE invoice_total < 500
ORDER BY payment_type, vendor_name, invoice_number
```

## The result set

	INVOICE_NUMBER	VENDOR_NAME	PAYMENT_TYPE	TOTAL	PAYMENT
1	40318	Data Reproductions Corp	33% Payment	21842	7273.386
2	0-2058	Malloy Lithographing Inc	33% Payment	37966.19	12642.74127
3	0-2060	Malloy Lithographing Inc	33% Payment	23517.58	7831.35414
4	0-2436	Malloy Lithographing Inc	33% Payment	10976.06	3655.02798
5	P-0259	Malloy Lithographing Inc	33% Payment	26881.4	8951.5062
6	P-0608	Malloy Lithographing Inc	33% Payment	20551.18	6843.54294
7	509786	Bertelsmann Industry Svcs. Inc	50% Payment	6940.25	3470.125

(114 rows selected)

## The syntax for MINUS and INTERSECT operations

```
SELECT_statement_1
{MINUS | INTERSECT}
  SELECT_statement_2
[ORDER BY order_by_list]
```

### The Customers table

CUSTOMER_LAST_NAME	CUSTOMER_FIRST_NAME
Anders	Maria
Trujillo	Ana
Moreno	Antonio
Hardy	Thomas
Berglund	Christina
Moos	Hanna

(24 rows selected)

### The Employees table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

(9 rows selected)

## A query that excludes rows from the first query if they also occur in the second query

```
SELECT customer_first_name, customer_last_name
  FROM customers
MINUS
  SELECT first_name, last_name
    FROM employees
   ORDER BY customer_last_name
```

### The result set

	CUSTOMER_FIRST_NAME	CUSTOMER_LAST_NAME
1	Maria	Anders
2	Christina	Berglund
3	Art	Braunschweiger
4	Donna	Chelan

(23 rows selected)

## A query that only includes rows that occur in both queries

```
SELECT customer_first_name, customer_last_name
FROM customers
INTERSECT
SELECT first_name, last_name
FROM employees
```

## The result set

	CUSTOMER_FIRST_NAME	CUSTOMER_LAST_NAME
1	Thomas	Hardy

(1 rows selected)

# MIS 381 INTRO. TO DATABASE MANAGEMENT

---

Advanced SQL

Summary queries, subqueries, functions

**Tayfun Keskin**

Visiting Clinical Professor, The University of Texas at Austin, McCombs School of Business  
Associate Teaching Professor, University of Washington Seattle, Foster School of Business

# QUESTIONS

- Any questions before we begin?



# AGENDA



Lecture

Summary Queries  
Subqueries, Functions



Hands-On

Exercises



Looking Forward

Homework 4  
PL/SQL



The University of Texas at Austin  
McCombs School of Business

# QUESTION

Let's speculate on this student entity:

What interesting questions might we ask?

## Student

StudentId(PK)  
StudentFirstName  
StudentLastName  
StudentStreet  
StudentCity  
StudentState  
StudentZip  
StudentGender  
StudentDoB  
Major  
Minor  
GPA  
CreditsCompleted  
DateEnrolled

# POSSIBLE QUERIES

- How many students do we have in a particular major?
  - Minor?
  - Gender?
  - State?
  - City?
- What is the average GPA of students in a particular major?
- What are the total and average credits completed by major based on date enrolled?
  - Does this change by minor?
  - Gender?

Student
StudentId(PK)
StudentFirstName
StudentLastName
StudentStreet
StudentCity
StudentState
StudentZip
StudentGender
StudentDoB
Major
Minor
GPA
CreditsCompleted
DateEnrolled

# SELECT SYNTAX FOR SUMMARY DATA

SELECT	Columns
	- Column names
	- Arithmetic expressions
	- Literals (text or numeric)
	- Scalar functions
	- <b>Column functions</b>
FROM	Table or view names
WHERE.	Conditions (qualifies rows)
ORDER BY	Sorts result rows
GROUP BY	Creates sub totals with column functions

- How many students do we have in a particular major?

```
SELECT count (studentid) , major  
FROM student  
GROUP BY major
```

- What is the average GPA of students in a particular major?

```
SELECT avg (GPA) , major  
FROM student  
GROUP BY major
```



# COMMON ORACLE COLUMN FUNCTIONS

SELECT	Columns
	- Column names
	- Arithmetic expressions
	- Literals (text or numeric)
	- Scalar functions
	- Column functions
FROM	Table or view names
WHERE.	Conditions (qualifies rows)
ORDER BY	Sorts result rows
GROUP BY	Creates sub totals with column functions

- AVG
- COUNT
- MAX
- MEDIAN
- MIN
- SUM
- **Also known as aggregate and analytic functions**



# COMMON ERRORS

```
SELECT count(studentid), major, minor FROM student  
GROUP BY major
```

ORA-00979: not a GROUP BY expression  
00979. 00000 - "not a GROUP BY expression"

\*Cause:

\*Action:

```
SELECT count(studentid), major, minor FROM student  
GROUP BY major, minor
```



# COMMON ERRORS

```
SELECT studentid, major FROM student  
GROUP BY major
```

ORA-00979: not a GROUP BY expression  
00979. 00000 - "not a GROUP BY expression"

\*Cause:

\*Action:

```
SELECT count(studentid), major FROM student  
GROUP BY major
```

# HAVING CLAUSE

SELECT	Columns
	- Column names
	- Arithmetic expressions
	- Literals (text or numeric)
	- Scalar functions
	- “Aggregate” functions
FROM	Table or view names
WHERE.	Conditions (qualifies rows)
ORDER BY	Sorts result rows
GROUP BY	Creates sub totals with column functions
HAVING	Aggregate condition

- How many majors do we have where the average GPA is less than 3.0? What is the count of student is such majors?

```
SELECT count(studentid), major
FROM student
GROUP BY major
HAVING (avg(GPA) < 3)
```

- How many students do we have by major with a GPA < 3

```
SELECT count(studentid), major
FROM student
WHERE GPA < 3
GROUP BY major
```



# LOOKING FORWARD

## Read Chapters:

- 3: Single table queries
- 4: Multiple table (Joins)
- 7: DML (insert, update...)
- 5: Summary queries
- 6: Subqueries
- 8: Data types and functions

## Homework 4, Quiz 4

## HBSP Package

## Exam 2



# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# **PART 2: SUBQUERIES AND FUNCTIONS**



# REMINDER QUESTION

What is a subquery?

# WHAT IS A SUBQUERY?

- Sometime the logic needed in a SELECT statement requires another SELECT statement – this second SELECT statement is called a subquery
- The syntax for a subquery is the same as the syntax for the SELECT statement
- A subquery can return a single value, a result set that contain a single column, or a result set that contains one or more columns
- The first SELECT statement is called the outer query

# SUBQUERIES CAN BE PART OF ...

- ... a WHERE clause as a search condition
- ... a HAVING clause as a search condition
- ... the FROM clause as a table specification
- ... a SELECT clause as a column specification

# SUBQUERIES CAN BE PART OF ...

- ... a WHERE clause as a search condition
- ... a HAVING clause as a search condition
  - When a subquery returns a single value, it can be used anywhere an expression is evaluated
  - When a subquery returns a single-column result set with two or more rows, it can be used in a place of a list of values, such as the list for an IN operator

# SUBQUERIES CAN BE PART OF ...

- ... the FROM clause as a table specification
  - When a subquery returns a result set with two or more columns, it can be used in a FROM clause
  - A subquery coded in a FROM clause is called an inline view
  - An inline view should have an alias
  - All calculated values should have a name (alias)
  - Inline views are not as efficient as views

# SUBQUERIES CAN BE PART OF ...

- ... a SELECT clause as a column specification
  - A subquery must return a single value
  - Usually a correlated subquery (i.e., a query that is executed once for each row processed by the outer query)
  - Such subqueries can usually be restated as joins

# SUBQUERIES CAN BE PART OF ...

- ... while syntactically correct subqueries are not used in the GROUP BY and ORDER BY clauses
- ... since joins are usually more efficient than subqueries, they are rarely used in a SELECT clause

# JOINS VS. SUBQUERIES

- **Advantages of Joins:**

- A join can include columns from both tables.
- A join is more intuitive when it uses an existing relationship.

- **Advantages of Subqueries:**

- A subquery can pass an aggregate value to the outer query.
- A subquery is more intuitive when it uses an ad hoc relationship.
- Long, complex queries can be easier to code with subqueries.



# PROCEDURE FOR BUILDING COMPLEX QUERIES

- State the problem to be solved in English.
- Use pseudocode to outline the query.
- If necessary, use pseudocode to outline each subquery.
- Code the subqueries and test them.
- Code and test the final query.



# LOOKING FORWARD

## Read Chapters:

- 3: Single table queries
- 4: Multiple table (Joins)
- 7: DML (insert, update...)
- 5: Summary queries
- 6: Subqueries
- 8: Data types and functions

## Homework 4, Quiz 4

## HBSP Package

## Exam 2



# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# **BACKUP SLIDES**



# PART 1

Chapter 5

Summary Queries



The University of Texas at Austin  
McCombs School of Business

# AGGREGATE FUNCTIONS

## The syntax of the aggregate functions

```
AVG ( [ALL|DISTINCT] expression)
SUM ( [ALL|DISTINCT] expression)
MIN ( [ALL|DISTINCT] expression)
MAX ( [ALL|DISTINCT] expression)
COUNT ( [ALL|DISTINCT] expression)
COUNT (*)
```

## A summary query example

```
SELECT COUNT(*) AS number_of_invoices,
       SUM(invoice_total) AS sum_of_invoice_totals
FROM   invoices;
```

## The result set

NUMBER_OF_RECORDS	Invoice Total Sum
114	214290.51

## Practice:

Write a query that returns the following:

- Count of invoice records
- Sum of invoice\_total
- Average invoice\_total
- Lowest invoice\_total
- Largest invoice\_total
- Count of distinct vendors
- Should we clean up any formatting data that you returned in query?

## A summary query with COUNT(\*), AVG, and SUM

```
SELECT 'After 1/1/2008' AS selection_date,  
       COUNT(*) AS number_of_invoices,  
       ROUND(AVG(invoice_total), 2) AS avg_invoice_amt,  
       SUM(invoice_total) AS total_invoice_amt  
FROM invoices  
WHERE invoice_date > '01-JAN-2014'
```

### The result set

SELECTION_DATE	NUMBER_OF_INVOICES	AVG_INVOICE_AMT	TOTAL_INVOICE_AMT
1 After 1/1/2014	114	1879.74	214290.51

## A summary query with MIN and MAX functions

```
SELECT 'After 1/1/2008' AS selection_date,  
       COUNT(*) AS number_of_invoices,  
       MAX(invoice_total) AS highest_invoice_total,  
       MIN(invoice_total) AS lowest_invoice_total  
  FROM invoices  
 WHERE invoice_date > '01-JAN-2014'
```

## The result set

SELECTION_DATE	NUMBER_OF_INVOICES	HIGHEST_INVOICE_TOTAL	LOWEST_INVOICE_TOTAL
1 After 1/1/2014	114	37966.19	6

## A summary query example

```
SELECT      COUNT(*) AS number_of_invoices,  
            SUM(invoice_total) AS sum_of_invoice_totals  
FROM invoices;
```

### Practice aggregating an arithmetic expression:

1. Update the above query to pull the SUM of what's due.

NOTE: The “amount due” = (invoice\_total – payment\_total – credit\_total)

2. Update query to only consider invoices created on/after a certain date

*Hint: Dates follow a “DD-MMM-YYYY” format (e.g. 07-Aug-2018)*

## A summary query with the DISTINCT keyword

```
SELECT COUNT(DISTINCT vendor_id) AS number_of_vendors,  
       COUNT(vendor_id) AS number_of_invoices,  
       ROUND(AVG(invoice_total),2) AS avg_invoice_amt,  
       SUM(invoice_total) AS total_invoice_amt  
FROM invoices  
WHERE invoice_date > '01-JAN-2008'
```

## The result set

	NUMBER_OF_VENDORS	NUMBER_OF_INVOICES	AVG_INVOICE_AMT	TOTAL_INVOICE_AMT
1	34	114	1879.74	214290.51

# GROUPING

## The syntax with GROUP BY and HAVING clauses

```
SELECT select_list
FROM table_source
[WHERE search_condition] --row filter
[GROUP BY group_by_list]
[HAVING search_condition] --aggregate filter
[ORDER BY order_by_list]
```

## A summary query that counts the number of invoices by vendor

```
SELECT vendor_id, COUNT(*) AS invoice_qty
FROM invoices
GROUP BY vendor_id
ORDER BY vendor_id
```

### The result set

VENDOR_ID	INVOICE_QTY
1	34
2	37
3	48
4	72

(34 rows selected)

First, let's look at Excel first...

## A summary query that calculates average invoice amount by vendor

```
SELECT vendor_id, vendor_name,  
       ROUND(AVG(invoice_total), 2) AS average_invoice_amount  
FROM invoices  
GROUP BY vendor_id, vendor_name  
ORDER BY average_invoice_amount DESC
```

The result set: (34 records)

	VENDOR_ID	AVERAGE_INVOICE_AMOUNT
1	110	23978.48
2	72	10963.66
3	104	7125.34
4	99	6940.25
5	119	4901.26
6	122	2575.33
7	86	2433
8	100	2184.5
9	113	1750
10	107	1600
11	103	1367.5
12	83	1077.21
13	81	936.93

# HAVING

## A summary query that calculates average invoice amount by vendor

```
SELECT vendor_id,  
       ROUND(AVG(invoice_total), 2) AS average_invoice_amount  
FROM invoices  
GROUP BY vendor_id  
HAVING ROUND(AVG(invoice_total), 2) > 2000  
ORDER BY average_invoice_amount DESC
```

The result set: (8 records)

	VENDOR_ID	AVERAGE_INVOICE_AMOUNT
1	110	23978.48
2	72	10963.66
3	104	7125.34
4	99	6940.25
5	119	4901.26
6	122	2575.33
7	86	2433
8	100	2184.5

## Practice:

1. Write a query that returns the vendor state and the count of vendors in that state
2. Update query to only show states with more than 2 vendors
3. Add in City between state and count
4. Sort by Count DESC

## A summary query with a join

```
SELECT vendor_state, vendor_city,
       COUNT(*) AS invoice_qty,
       ROUND(AVG(invoice_total),2) AS invoice_avg
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
 GROUP BY vendor_state, vendor_city
 ORDER BY vendor_state, vendor_city
```

## The result set

VENDOR_STATE	VENDOR_CITY	INVOICE_QTY	INVOICE_AVG
1 AZ	Phoenix	1	662
2 CA	Fresno	19	1208.75
3 CA	Los Angeles	1	503.2
4 CA	Oxnard	3	188

(20 rows selected)

## A summary query that limits the groups to those with two or more invoices

```
SELECT vendor_state, vendor_city,
       COUNT(*) AS invoice_qty,
       ROUND(AVG(invoice_total),2) AS invoice_avg
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
 GROUP BY vendor_state, vendor_city
 HAVING COUNT(*) >= 2
 ORDER BY vendor_state, vendor_city
```

## The result set

	VENDOR_STATE	VENDOR_CITY	INVOICE_QTY	INVOICE_AVG
1	CA	Fresno	19	1208.75
2	CA	Oxnard	3	188
3	CA	Pasadena	5	196.12
4	CA	Sacramento	7	253

(12 rows selected)

# HAVING vs WHERE

# HAVING vs WHERE?

## Summary query with filter condition in the HAVING clause

```
SELECT vendor_name, COUNT(*) AS invoice_qty,  
       ROUND(AVG(invoice_total),2) AS invoice_avg  
FROM vendors JOIN invoices  
    ON vendors.vendor_id = invoices.vendor_id  
GROUP BY vendor_name  
HAVING AVG(invoice_total) > 500  
ORDER BY invoice_qty DESC
```

## The result set

VENDOR_NAME	INVOICE_QTY	INVOICE_AVG
1 United Parcel Service	9	2575.33
2 Zylka Design	8	867.53
3 Malloy Lithographing Inc	5	23978.48
4 IBM	2	600.06

## Summary query with filter condition in the WHERE clause

```
SELECT vendor_name, COUNT(*) AS invoice_qty,  
       ROUND(AVG(invoice_total),2) AS invoice_avg  
FROM vendors JOIN invoices  
    ON vendors.vendor_id = invoices.vendor_id  
WHERE invoice_total > 500 (row level)  
GROUP BY vendor_name  
ORDER BY invoice_qty DESC
```

## The result set

VENDOR_NAME	INVOICE_QTY	INVOICE_AVG
1 United Parcel Service	9	2575.33
2 Zylka Design	7	946.67
3 Malloy Lithographing Inc	5	23978.48
4 Ingram	2	1077.21

# HAVING vs WHERE?

```
Select *
From invoices
where vendor_id in (
    select vendor_id
    from vendors
    where vendor_name like 'Zylka%'
)
```

	INVOICE_ID	VENDOR_ID	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL	PAYMENT_TOTAL	CREDIT_TOTAL	TERMS_ID	INVOICE_DUE_DATE	PAYMENT_DATE
1	15	121	97/553B	26-APR-14	313.55	0	0	4	09-JUL-14	(null)
2	18	121	97/553	27-APR-14	904.14	0	0	4	09-JUL-14	(null)
3	19	121	97/522	30-APR-14	1962.13	0	200	4	10-JUL-14	(null)
4	20	121	97/503	30-APR-14	639.77	639.77	0	4	11-JUN-14	05-JUN-14
5	21	121	97/488	30-APR-14	601.95	601.95	0	3	03-JUN-14	27-MAY-14
6	22	121	97/486	30-APR-14	953.1	953.1	0	2	21-MAY-14	13-MAY-14
7	23	121	97/465	01-MAY-14	565.15	565.15	0	1	14-MAY-14	05-MAY-14
8	24	121	97/222	01-MAY-14	1000.46	1000.46	0	3	03-JUN-14	25-MAY-14

## A summary query with a compound condition in the HAVING clause

```
SELECT
    invoice_date,
    COUNT(*) AS invoice_qty,
    SUM(invoice_total) AS invoice_sum
FROM invoices
GROUP BY invoice_date
HAVING invoice_date
    BETWEEN '01-MAY-2014' AND '31-MAY-2014'
    AND COUNT(*) > 1
    AND SUM(invoice_total) > 100
ORDER BY invoice_date DESC
```

### The result set

	INVOICE_DATE	INVOICE_QTY	INVOICE_SUM
1	31-MAY-14	3	11557.75
2	23-MAY-14	6	2761.17
3	22-MAY-14	2	442.5
4	20-MAY-14	3	308.64

(15 rows selected)

## The same query with a WHERE clause

```
SELECT
    invoice_date,
    COUNT(*) AS invoice_qty,
    SUM(invoice_total) AS invoice_sum
FROM invoices
WHERE invoice_date
    BETWEEN '01-MAY-2014' AND '31-MAY-2014'
GROUP BY invoice_date
HAVING COUNT(*) > 1
    AND SUM(invoice_total) > 100
ORDER BY invoice_date DESC
```

### TIP:

Key aggregate filters in HAVING because you have to.

Keep row-level filters in WHERE for readability

## Practice:

- Pull all invoices after June 1 and give the average invoice\_total by vendor\_state. Only show states with avg > 2000.

## Follow these steps:

1. Select \* From table...
2. Code the JOIN (if applicable)
3. Code WHERE
4. Specify the columns and add aggregate functions AND group by
5. Code HAVING filter aggregate columns

# ROLLUP / CUBE

## A summary query with a final summary row

```
SELECT vendor_id, COUNT(*) AS invoice_count,  
       SUM(invoice_total) AS invoice_total  
  FROM invoices  
 GROUP BY ROLLUP(vendor_id)
```

### The result set

VENDOR_ID	INVOICE_COUNT	INVOICE_TOTAL
32	121	8 6940.25
33	122	9 23177.96
34	123	47 4378.02
35	(null)	114 214290.51

(35 rows selected)

## Practice:

- Pull vendor state and count of vendors like before but use a ROLLUP keyword in group by
- Try adding in an additional column like city into the select and rollup
- Try updating ROLLUP to CUBE

## A summary query with a summary row at the start of the result set

```
SELECT vendor_id, COUNT(*) AS invoice_count,  
       SUM(invoice_total) AS invoice_total  
  FROM invoices  
 GROUP BY CUBE(vendor_id)
```

### The result set

	VENDOR_ID	INVOICE_COUNT	INVOICE_TOTAL
1	(null)	114	214290.51
2	34	2	1200.12
3	37	3	564
4	48	1	856.92

(35 rows selected)

## A summary query with a summary row for each set of groups

```
SELECT vendor_state, vendor_city, COUNT(*) AS qty_vendors
FROM vendors
WHERE vendor_state IN ('IA', 'NJ')
GROUP BY CUBE(vendor_state, vendor_city)
ORDER BY vendor_state, vendor_city
```

## The result set

	VENDOR_STATE	VENDOR_CITY	QTY_VENDORS
1	IA	Fairfield	1
2	IA	Washington	1
3	IA	(null)	2
4	NJ	East Brunswick	2
5	NJ	Fairfield	1
6	NJ	Washington	1
7	NJ	(null)	4
8	(null)	East Brunswick	2
9	(null)	Fairfield	2
10	(null)	Washington	2
11	(null)	(null)	6

## A summary query for non-numeric columns

```
SELECT MIN(vendor_name) AS first_vendor,  
       MAX(vendor_name) AS last_vendor,  
       COUNT(vendor_name) AS number_of_vendors  
FROM vendors
```

### The result set

FIRST_VENDOR	LAST_VENDOR	NUMBER_OF_VENDORS
1 ASC Signs	Zylka Design	122

## Practice:

1. Pull the Last Name in vendor\_contacts closest to A (e.g., Adams). Pull the Last Name closest to Z (e.g., Zilker)
2. Pop Quiz – Do you think we can pull the AVG of last\_name?

# PART 2

Chapter 6

Subqueries



The University of Texas at Austin  
McCombs School of Business

# 4 ways to use a subquery in a SELECT statement

- In a WHERE clause as a search condition
- In a HAVING clause as a search condition
- In the FROM clause as a table specification
- In the SELECT clause as a column specification

**Subquery in  
WHERE**

# Basic use case: Use a Subquery in the WHERE clause

## Part 1: Write subquery

```
SELECT AVG(invoice_total)  
FROM invoices
```

## Value returned by query

1879.7413

## Part 2. Use query as a subquery in a WHERE clause

```
SELECT *  
FROM invoices  
WHERE invoice_total >  
(SELECT AVG(invoice_total)  
FROM invoices)  
ORDER BY invoice_total
```

## The result set

	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1	989319-487	18-APR-14	1927.54
2	97/522	30-APR-14	1962.13
3	989319-417	26-APR-14	2051.59
4	989319-427	25-APR-14	2115.81
5	989319-477	19-APR-14	2184.11

## A query that uses an inner join

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
WHERE vendor_state = 'CA'
ORDER BY invoice_date
```

### The result set

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 QP58872	25-FEB-14	116.54
2 Q545443	14-MAR-14	1083.58
3 MAB01489	16-APR-14	936.93
4 97/553B	26-APR-14	313.55

(40 rows)

## The same query but with a subquery

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE vendor_id IN
    (SELECT vendor_id
     FROM vendors
     WHERE vendor_state = 'CA')
ORDER BY invoice_date
```

### Advantages of joins

- A join can include columns from both tables.
- A join is more intuitive when it uses an existing relationship.

### Advantages of subqueries

- A subquery can pass an aggregate value to the outer query.
- A subquery is more intuitive when it uses an ad hoc relationship.
- Long, complex queries can be easier to code with subqueries.

## Build first query

```
SELECT vendor_id  
FROM vendors  
where vendor_name like 'B%'
```

## Subquery Results

VENDOR_ID
8
11
17
37
47
51
67
84
99

## Final Query w/ Subquery

```
SELECT *  
from invoices  
WHERE vendor_id IN (  
    SELECT vendor_id  
    FROM vendors  
    where vendor_name like 'B%')
```

## Same as the following

```
SELECT *  
from invoices  
WHERE vendor_id IN  
(8,11,17,37,47,51,67,84,99)
```

# A procedure for building complex queries

- State the problem to be solved in English.
- Use pseudocode to outline your first query.
- If necessary, use pseudocode to outline each subquery.
- Code the subqueries and test them individually
- Combine inner queries with outer queries and test the final query.

# ANY, SOME, ALL

## Keywords

## The syntax of a subquery in WHERE

```
Select column_list
From table
WHERE expression comparison_operator
[SOME|ANY|ALL] (subquery)
```

## Example

```
SELECT *
from invoices
WHERE vendor_id IN (
    SELECT vendor_id
    FROM vendors
    where vendor_name like 'B%')
```

## The syntax of a WHERE clause that uses an IN phrase with a subquery

```
WHERE test_expression [NOT] IN (subquery)
```

### A query that returns vendors without invoices

```
SELECT vendor_id, vendor_name, vendor_state
FROM vendors
WHERE vendor_id NOT IN
    (SELECT DISTINCT vendor_id
     FROM invoices)
ORDER BY vendor_id
```

### How would you do this another way?

```
SELECT v.vendor_id, vendor_name, vendor_state
FROM vendors v LEFT JOIN invoices i
    ON v.vendor_id = i.vendor_id
WHERE i.vendor_id IS NULL
ORDER BY v.vendor_id
```

### The result of the subquery

VENDOR_ID
1
2
3
4
5
6

(34 rows)

### The result set

VENDOR_ID	VENDOR_NAME	VENDOR_STATE
32	33 Nielson	OH
33	35 Cal State Termite	CA
34	36 Graylift	CA
35	38 Venture Communications Int'l	NY
36	39 Custom Printing Company	MO
37	40 Nat Assoc of College Stores	OH

(88 rows)

## How the ALL keyword works

Condition	Equivalent expression
<code>x &gt; ALL (1, 2)</code>	<code>x &gt; 2</code>
<code>x &lt; ALL (1, 2)</code>	<code>x &lt; 1</code>
<code>x = ALL (1, 2)</code>	<code>(x = 1) AND (x = 2)</code>
<code>x &lt;&gt; ALL (1, 2)</code>	<code>(x &lt;&gt; 1) AND (x &lt;&gt; 2)</code>

## A procedure for building complex queries

- State the problem to be solved in English.
- Use pseudocode to outline the query.
- If necessary, use pseudocode to outline each subquery.
- Code the subqueries and test them individually
- Combine inner queries with outer queries and test the final query.

```
select vendor_id  
from vendors  
where vendor_name = 'IBM'
```

```
Select invoice_total  
from invoices  
where vendor_id in (  
    select vendor_id  
    from vendors  
    where vendor_name = 'IBM')
```

```
select *  
from invoices  
where invoice_total > ALL (  
    Select invoice_total from invoices  
    where vendor_id in (  
        select vendor_id  
        from vendors  
        where vendor_name = 'IBM')  
);
```

## Query that uses join and ALL

```
SELECT vendor_name, invoice_number, invoice_total  
FROM invoices i JOIN vendors v  
    ON i.vendor_id = v.vendor_id  
WHERE invoice_total > ALL  
    (SELECT invoice_total  
        FROM invoices  
        WHERE vendor_id = 34)  
ORDER BY vendor_name
```

## The result of the subquery

INVOICE_TOTAL
116.54
1083.58

## The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_TOTAL
Bertelsmann Industry Svcs. Inc	509786	6940.25
Cahners Publishing Company	587056	2184.5
Computerworld	367447	2433
Data Reproductions Corp	40318	21842

(25 rows)

## How the ANY and SOME keywords work

Condition	Equivalent expression
<code>x &gt; ANY (1, 2)</code>	<code>x &gt; 1</code>
<code>x &lt; ANY (1, 2)</code>	<code>x &lt; 2</code>
<code>x = ANY (1, 2)</code>	<code>(x = 1) OR (x = 2)</code>
<code>x &lt;&gt; ANY (1, 2)</code>	<code>(x &lt;&gt; 1) OR (x &lt;&gt; 2)</code>

## How the ANY and SOME keywords work

Condition	Equivalent expression
<code>x &gt; ANY (1, 2)</code>	<code>x &gt; 1</code>
<code>x &lt; ANY (1, 2)</code>	<code>x &lt; 2</code>
<code>x = ANY (1, 2)</code>	<code>(x = 1) OR (x = 2)</code>
<code>x &lt;&gt; ANY (1, 2)</code>	<code>(x &lt;&gt; 1) OR (x &lt;&gt; 2)</code>

## Practice:

- Pull all the invoices that had invoice totals bigger than ANY of IBM's invoices

# **Subquery in FROM**

# When is it useful to select FROM a query?

- When you want to aggregate an aggregate value
- e.g. You want to find the average count of vendors that sent us more than 1 invoice.
  - First you need to find the count of invoices for each vendor
  - Then you need to filter out vendors with a count of 1
  - Then you can find the average of the counts remaining
- Example that won't work

```
SELECT VENDOR_ID, avg(count(invoice_id))
FROM INVOICES
GROUP BY VENDOR_ID....
```

ORA-00935: group function is nested too deeply  
00935. 00000 - "group function is nested too deeply"  
--

## A query that uses an inline view

```
SELECT i.vendor_id,  
       MAX(invoice_date) AS last_invoice_date,  
       AVG(invoice_total) AS average_invoice_total  
FROM invoices i JOIN  
  (  
    SELECT vendor_id,  
           AVG(invoice_total) AS average_invoice_total  
    FROM invoices  
   HAVING AVG(invoice_total) > 4900  
  GROUP BY vendor_id  
  ) v  
  ON i.vendor_id = v.vendor_id  
GROUP BY i.vendor_id  
ORDER BY MAX(invoice_date) DESC
```

NOTE: You technically don't need a subquery to do this. Just review this as a syntax example of how to join a table to query

### The result of the subquery (an inline view)

VENDOR_ID	AVERAGE_INVOICE_TOTAL
1	10963.655
2	6940.25
3	7125.34
4	23978.482
5	4901.26

### The result set

VENDOR_ID	LAST_INVOICE_DATE	AVERAGE_INVOICE_TOTAL
1	72 18-JUL-14	10963.655
2	119 04-JUN-14	4901.26
3	104 03-JUN-14	7125.34
4	99 31-MAY-14	6940.25
5	110 08-MAY-14	23978.482

# COMPLEX QUERIES

## Problem to solve:

Retrieve all vendor\_ids, their most recent invoice\_date, and average invoice\_total for all vendors that have AVG invoice total great than \$4900

## How?

Join a **table** to a **select statement**

INVOICE_ID	VENDOR_ID	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1	34	QP58872	25-FEB-14	116.54
2	34	Q545443	14-MAR-14	1083.58
3	110	P-0608	11-APR-14	20551.18
4	110	P-0259	16-APR-14	26881.4
5	81	MAB01489	16-APR-14	936.93
6	122	989319-497	17-APR-14	2312.2
7	82	C73-24	17-APR-14	600
8	122	989319-487	18-APR-14	1927.54
9	122	989319-477	19-APR-14	2184.11
10	122	989319-467	24-APR-14	2318.03
11	122	989319-457	24-APR-14	3813.33
12	122	989319-447	24-APR-14	3689.99
13	122	989319-437	24-APR-14	2765.36
14	122	989319-427	25-APR-14	2115.81

VENDOR_ID	AVERAGE_INVOICE_TOTAL
72	10963.655
99	6940.25
104	7125.34
110	23978.482
119	4901.26

# When a *in line* subquery is needed (i.e. in FROM)

- Pull the vendor in each state with highest invoice total (Vendor\_Name, State, Invoice Total)
- What do we need to first?

## STEP 1 – Code first subquery

```
SELECT v.vendor_id,
       v.vendor_name,
       v.vendor_state,
       SUM(i.invoice_total) AS sum_of_invoices
  FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id
 GROUP BY v.vendor_id, v.vendor_name, v.vendor_state
 ORDER BY vendor_state
```

VENDOR_ID	VENDOR_NAME	VENDOR_STATE	SUM_OF_INVOICES
96	Wells Fargo Bank	AZ	662
94	Abbey Office Furnishings	CA	17.5
99	Bertelsmann Industry Svcs. Inc	CA	6940.25
37	Blue Cross	CA	564
102	Coffee Break Service	CA	41.8
86	Computerworld	CA	2433
104	Digital Dreamworks	CA	7125.34
105	Dristas Groom & McCormick	CA	220
89	Evans Executone Inc	CA	95
106	Ford Motor Credit Company	CA	503.2
107	Franchise Tax Board	CA	1600
48	Fresno County Tax Collector	CA	856.92
108	Gostanian General Building	CA	450
34	IBM	CA	1200.12

## STEP 2 – Move subquery into outer query

```
SELECT *
  FROM (SELECT v.vendor_id,
               v.vendor_name,
               v.vendor_state,
               SUM(i.invoice_total) AS sum_of_invoices
          FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id
         GROUP BY v.vendor_id, v.vendor_name, v.vendor_state
        ORDER BY vendor_state)
```

VENDOR_ID	VENDOR_NAME	VENDOR_STATE	SUM_OF_INVOICES
96	Wells Fargo Bank	AZ	662
94	Abbey Office Furnishings	CA	17.5
99	Bertelsmann Industry Svcs. Inc	CA	6940.25
37	Blue Cross	CA	564
102	Coffee Break Service	CA	41.8
86	Computerworld	CA	2433
104	Digital Dreamworks	CA	7125.34
105	Dristas Groom & McCormick	CA	220
89	Evans Executone Inc	CA	95
106	Ford Motor Credit Company	CA	503.2
107	Franchise Tax Board	CA	1600
48	Fresno County Tax Collector	CA	856.92
108	Gostanian General Building	CA	450

# When a *in line* subquery is needed (i.e. in FROM)

- Pull the vendor in each state with highest invoice total (Vendor\_Name, State, Invoice Total)
- What do we need to first?

## STEP 1 – Code first subquery

```

SELECT v.vendor_id,
       v.vendor_name,
       v.vendor_state,
       SUM(i.invoice_total) AS sum_of_invoices
  FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id
 GROUP BY v.vendor_id, v.vendor_name, v.vendor_state
 ORDER BY vendor_state
    
```

VENDOR_ID	VENDOR_NAME	VENDOR_STATE	SUM_OF_INVOICES
96	Wells Fargo Bank	AZ	662
94	Abbey Office Furnishings	CA	17.5
99	Bertelsmann Industry Svcs. Inc	CA	6940.25
37	Blue Cross	CA	564
102	Coffee Break Service	CA	41.8
86	Computerworld	CA	2433
104	Digital Dreamworks	CA	7125.34
105	Dristas Groom & McCormick	CA	220
89	Evans Executone Inc	CA	95
106	Ford Motor Credit Company	CA	503.2
107	Franchise Tax Board	CA	1600
48	Fresno County Tax Collector	CA	856.92
108	Gostanian General Building	CA	450
34	IBM	CA	1200.12

## STEP 2 – Move subquery into outer query

```

SELECT vendor_state, MAX(sum_of_invoices) AS invoice_total
  FROM (SELECT v.vendor_id,
               v.vendor_name,
               v.vendor_state,
               SUM(i.invoice_total) AS sum_of_invoices
          FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id
         GROUP BY v.vendor_id, v.vendor_name, v.vendor_state
        ORDER BY vendor_state)
    GROUP BY vendor_state
    
```

VENDOR_STATE	INVOICE_TOTAL
CA	7125.34
MA	1367.5
OH	207.78
TN	4378.02
MI	119892.41
DC	600
NV	23177.96
TX	2154.42
AZ	662
PA	265.36

# When a *in line* subquery is needed (i.e. in FROM)

## STEP 3 – Join both query outputs together like they're tables

**SELECT \* FROM**

```
(SELECT v.vendor_id,  
v.vendor_name,  
v.vendor_state,  
SUM(i.invoice_total) AS sum_of_invoices  
FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id  
GROUP BY v.vendor_id, v.vendor_name, v.vendor_state  
order by vendor_state)
```

```
(select vendor_state, max(sum_of_invoices) as invoice_total  
from
```

```
(SELECT v.vendor_id,  
v.vendor_state,  
SUM(i.invoice_total) AS sum_of_invoices  
FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id  
GROUP BY v.vendor_id, v.vendor_state  
order by v.vendor_state)  
group by vendor_state)
```

VENDOR_ID	VENDOR_NAME	VENDOR_STATE	SUM_OF_INVOICES	VENDOR_STATE_1	INVOICE_TOTAL
104	Digital Dreamworks	CA	7125.34	CA	7125.34
103	Dean Witter Reynolds	MA	1367.5	MA	1367.5
88	Edward Data Services	OH	207.78	OH	207.78
123	Federal Express Corporation	TN	4378.02	TN	4378.02
110	Malloy Lithographing Inc	MI	119892.41	MI	119892.41
82	Reiter's Scientific & Pro Books	DC	600	DC	600
122	United Parcel Service	NV	23177.96	NV	23177.96
83	Ingram	TX	2154.42	TX	2154.42
96	Wells Fargo Bank	AZ	662	AZ	662
80	Cardinal Business Media, Inc.	PA	265.36	PA	265.36

# When a *in line* subquery is needed (i.e. in FROM)

## STEP 4 – Update your final select LAST

**SELECT select summary\_1.vendor\_name, summary\_1.vendor\_state, invoice\_total FROM**

**(SELECT v.vendor\_id,  
v.vendor\_name,  
v.vendor\_state,  
SUM(i.invoice\_total) AS sum\_of\_invoices  
FROM invoices i JOIN vendors v ON i.vendor\_id = v.vendor\_id  
GROUP BY v.vendor\_id, v.vendor\_name, v.vendor\_state  
order by vendor\_state) summary\_1**

**INNER JOIN**

**(select vendor\_state, max(sum\_of\_invoices) as invoice\_total  
from**

**(SELECT v.vendor\_id,  
v.vendor\_state,  
SUM(i.invoice\_total) AS sum\_of\_invoices  
FROM invoices i JOIN vendors v ON i.vendor\_id = v.vendor\_id  
GROUP BY v.vendor\_id, v.vendor\_state  
order by v.vendor\_state)**

**group by vendor\_state) summary\_2**

**ON summary\_1.vendor\_state = summary\_2.vendor\_state  
and summary\_1.sum\_of\_invoices = summary\_2.invoice\_total  
Order by summary\_1.vendor\_state**

VENDOR_NAME	VENDOR_STATE	INVOICE_TOTAL
Wells Fargo Bank	AZ	662
Digital Dreamworks	CA	7125.34
Reiter's Scientific & Pro Books	DC	600
Dean Witter Reynolds	MA	1367.5
Malloy Lithographing Inc	MI	119892.41
United Parcel Service	NV	23177.96
Edward Data Services	OH	207.78
Cardinal Business Media, Inc.	PA	265.36
Federal Express Corporation	TN	4378.02
Ingram	TX	2154.42

# MORE ON COMPLEX QUERIES

## A query that uses three subqueries

```

SELECT summary1.vendor_state, summary1.vendor_name,
       top_in_state.sum_of_invoices
  FROM
    (
      SELECT v_sub.vendor_state, v_sub.vendor_name,
             SUM(i_sub.invoice_total) AS sum_of_invoices
        FROM invoices i_sub JOIN vendors v_sub
          ON i_sub.vendor_id = v_sub.vendor_id
       GROUP BY v_sub.vendor_state, v_sub.vendor_name
    ) summary1
   JOIN
    (
      SELECT summary2.vendor_state,
             MAX(summary2.sum_of_invoices) AS sum_of_invoices
        FROM
          (
            SELECT v_sub.vendor_state, v_sub.vendor_name,
                   SUM(i_sub.invoice_total) AS sum_of_invoices
              FROM invoices i_sub JOIN vendors v_sub
                ON i_sub.vendor_id = v_sub.vendor_id
             GROUP BY v_sub.vendor_state, v_sub.vendor_name
          ) summary2
         GROUP BY summary2.vendor_state
    ) top_in_state
  ON summary1.vendor_state =
     top_in_state.vendor_state AND
summary1.sum_of_invoices =
     top_in_state.sum_of_invoices
 ORDER BY summary1.vendor_state

```

## The result set

VENDOR_STATE	VENDOR_NAME	SUM_OF_INVOICES
1 AZ	Wells Fargo Bank	662
2 CA	Digital Dreamworks	7125.34
3 DC	Reiter's Scientific & Pro Books	600
4 MA	Dean Witter Reynolds	1367.5
5 MI	Malloy Lithographing Inc	119892.41
6 NV	United Parcel Service	23177.96
7 OH	Edward Data Services	207.78

(10 rows)

## Pseudocode for the query

```
SELECT summary1.vendor_state, summary1.vendor_name,
       top_in_state.sum_of_invoices
  FROM (inline view returning
            vendor_state, vendor_name, sum_of_invoices)
                  AS summary1
   JOIN (inline view returning vendor_state,
             max(sum_of_invoices))
                  AS top_in_state
      ON summary1.vendor_state =
         top_in_state.vendor_state
     AND summary1.sum_of_invoices =
         top_in_state.sum_of_invoices
 ORDER BY summary1.vendor_state
```

## Pseudocode for the Top\_In\_State subquery

```
SELECT summary2.vendor_state,
       MAX(summary2.sum_of_invoices)
  FROM (inline view returning vendor_state,
            vendor_name, sum_of_invoices)
                  AS summary2
 GROUP BY summary2.vendor_state
```

## The Summary1 and Summary2 subqueries

```
SELECT v_sub.vendor_state, v_sub.vendor_name,
       SUM(i_sub.invoice_total) AS sum_of_invoices
  FROM invoices i_sub JOIN vendors v_sub
    ON i_sub.vendor_id = v_sub.vendor_id
 GROUP BY v_sub.vendor_state, v_sub.vendor_name
 ORDER BY v_sub.vendor_state, v_sub.vendor_name
```

## The result of Summary1 and Summary2

VENDOR_STATE	VENDOR_NAME	SUM_OF_INVOICES
1 AZ	Wells Fargo Bank	662
2 CA	Abbey Office Furnishings	17.5
3 CA	Bertelsmann Industry Svcs. Inc	6940.25

(34 rows)

## The result of the Top\_In\_State subquery

VENDOR_STATE	SUM_OF_INVOICES
1 CA	7125.34
2 MA	1367.5
3 OH	207.78

(10 rows)

# CORRELATED SUBQUERIES

**Out of scope for class**

## Problem to solve:

Pull vendors with an above average invoice total

### A query that uses a correlated subquery

```
SELECT vendor_id, invoice_number, invoice_total
FROM invoices inv_main
WHERE invoice_total >
    (SELECT AVG(invoice_total)
     FROM invoices inv_sub
      WHERE inv_sub.vendor_id = inv_main.vendor_id)
ORDER BY vendor_id, invoice_total
```

INSTEAD...try the SYNTAX to use an IN-LINE JOIN INSTEAD

```
SELECT *
FROM table1 alias_1 inner join
    (sub-query) alias_2
  on alias_1.key = alias_2.key
```

### The value returned by the subquery for vendor 95

28.50166...

### The result set

VENDOR_ID	INVOICE_NUMBER	INVOICE_TOTAL
6	83 31359783	1575
7	95 111-92R-10095	32.7
8	95 111-92R-10093	39.77
9	95 111-92R-10092	46.21
10	110 P-0259	26881.4

(36 rows) **But, correlated can be complicated!**

## A query that uses a correlated subquery

```
SELECT vendor_name,  
       (SELECT MAX(invoice_date) FROM invoices  
        WHERE invoices.vendor_id =  
              vendors.vendor_id) AS latest_inv  
FROM vendors  
ORDER BY latest_inv
```

### The result set

VENDOR_NAME	LATEST_INV
1 IBM	14-MAR-14
2 Wang Laboratories, Inc.	16-APR-14
3 Reiter's Scientific & Pro Books	17-APR-14
4 United Parcel Service	26-APR-14
5 Wakefield Co	26-APR-14
6 Zylka Design	01-MAY-14
7 Abbey Office Furnishings	02-MAY-14

(122 rows)

## The same query restated using a join

```
SELECT vendor_name,  
       MAX(invoice_date) AS latest_inv  
FROM vendors v  
      LEFT JOIN invoices i  
        ON v.vendor_id = i.vendor_id  
GROUP BY vendor_name  
ORDER BY latest_inv
```

### The same result set

VENDOR_NAME	LATEST_INV
1 IBM	14-MAR-14
2 Wang Laboratories, Inc.	16-APR-14
3 Reiter's Scientific & Pro Books	17-APR-14
4 United Parcel Service	26-APR-14
5 Wakefield Co	26-APR-14
6 Zylka Design	01-MAY-14
7 Abbey Office Furnishings	02-MAY-14

(122 rows)

## Problem to solve:

### Pull vendors without invoices

#### The syntax of a subquery with EXISTS

```
WHERE [NOT] EXISTS (subquery)
```

#### A query that returns vendors without invoices

```
SELECT vendor_id, vendor_name, vendor_state
FROM vendors
WHERE NOT EXISTS
  (SELECT *
   FROM invoices
   WHERE invoices.vendor_id = vendors.vendor_id)
```

#### The result set

VENDOR_ID	VENDOR_NAME	VENDOR_STATE
53	33 Nielson	OH
54	35 Cal State Termite	CA
55	36 Graylift	CA
56	38 Venture Communications Int'l	NY
57	39 Custom Printing Company	MO
58	40 Nat Assoc of College Stores	OH

(88 rows)

# PART 3

Chapter 8

Data types and functions



The University of Texas at Austin  
McCombs School of Business

# Built-in data type categories

- **Character** – (i.e. strings)
- **Numeric** – (integers and decimals)
- **Temporal** – (e.g. dates, times)
- Large object (LOB) – (e.g. Text, images, sounds, video)
- Rowid – (address for each row in a database)

## ANSI type

CHARACTER (n)  
CHAR (n)  
CHARACTER VARYING (n)  
CHAR VARYING (n)  
NATIONAL CHARACTER (n)  
NATIONAL CHAR (n)  
NCHAR (n)  
NATIONAL CHARACTER VARYING (n)  
NATIONAL CHAR VARYING (n)  
NCHAR VARYING (n)  
NUMERIC (p , s)  
DECIMAL (p , s)  
INTEGER  
INT  
SMALLINT  
FLOAT  
DOUBLE PRECISION  
REAL

## Oracle equivalent

CHAR (n)  
VARCHAR2 (n)  
NCHAR (n)  
NVARCHAR2 (n)  
NUMBER (p , s)  
NUMBER (38)  
FLOAT (126)  
FLOAT (126)  
FLOAT (63)

1 byte = 256  
characters

ABC  
123

2-3 bytes =  
65,000 characters

七  
米

p = total number of digits stored  
s = number of decimal digits that can be stored

# The date/time data types

**DATE**

**TIMESTAMP [ (fsp) ]**

**TIMESTAMP [ (fsp) ] WITH TIME ZONE**

**TIMESTAMP [ (fsp) ] WITH LOCAL TIME ZONE**

**INTERVAL YEAR [ (yp) ] TO MONTH**

**INTERVAL DAY [ (dp) ] TO SECOND [ (fsp) ]**

## Terms to know

- Temporal data types
- Date/time data type

# The large object data types

**CLOB – characters**

**NCLOB – Unicode characters**

**BLOB – unstructured data (image, sound, video)**

**BFILE – pointer to a large file outside database**

## Oracle functions for converting data

`TO_CHAR(expr[, format])`

`TO_NUMBER(expr[, format])`

`TO_DATE(expr[, format])`

## Examples that use TO functions

`TO_CHAR(1975.5)`

`TO_CHAR(1975.5, '$99,999.99')`

`TO_CHAR(SYSDATE)`

`TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS')`

`TO_NUMBER('1975.5')`

`TO_NUMBER('$1,975.5', '$99,999.99')`

`TO_DATE('15-APR-14')`

`TO_CHAR(TO_DATE('15-APR-14'), 'DD-MON-YYYY HH24:MI:SS')`

NOTE: format will use a default if not specified

### Practice:

1. Select SYSDATE from *dual* and format with the following but convert to string with the following formats

- OCT-2019
- October 23, 2019
- 10/23/19 06:10:01
- WED, 10-23-2019

2. Select invoice\_total from invoices and format as char, \$999.99

# FYI other option: CAST functions

```
SELECT invoice_id,  
       invoice_date,  
       invoice_total,  
       CAST(invoice_date AS VARCHAR2(9)),  
       CAST(invoice_total AS NUMBER(9))  
FROM   invoices
```

## Same query using TO\_ functions

```
SELECT invoice_id,  
       invoice_date,  
       invoice_total,  
       to_char(invoice_date),  
       to_number(invoice_total)  
FROM   invoices
```

## NOTE:

- ANSI-standard so you can use on other DBMS
- TO\_ functions gives more control
- Expect that if you are using different DBMSs (MySQL, SQL Server, DB2) that you may have to use different functions to convert data. Googling never ends!

# Working with Numbers

## Number format elements

9	= digits
. or D	= decimals
G or ,	= group separator
0	= leading/trailing 0
\$	= \$ sign
L	= local currency
U	
C	= currency symbol
S	= - or + sign
MI	= - for negatives
PR	= brackets for neg.
FM	= remove trail/lead 0
EEEE	

## Examples

Value	Format	Output
1975.5	(none specified)	1975.5
1975.5	999	###
1975.5	9999	1976
1975.5	9,999.9	1,975.5
1975.5	9G999D9	1,975.5
1975.5	99,999.99	1,975.50
1975.5	09,999.990	01,975.500
1975.5	\$99,999.99	\$1,975.50
1975.5	L9,999.99	\$1,975.50
1975.5	U9,999.99	\$1,975.50
1975.5	C9,999.99	USD1,975.50
1975.5	S9,999.99	+1,975.50
-1975.5	9,999.99S	1,975.50-
-1975.5	9,999.99MI	1,975.50-
-1975.5	9,999.99PR	<1,975.50>
1975.5	9,999.99PR	1,975.50
01975.50	FM9,999.99	1,975.5
1975.5	9.99EEEE	1.98E+03

**Practice:** Experiment with format of currency fields on invoices, make a negative

## Some common numeric functions

`ROUND(number[, length])`

`TRUNC(number[, length])`

`CEIL(number)`

`FLOOR(number)`

`ABS(number)`

`SIGN(number)`

`MOD(number, number_divisor)`

`POWER(number, number_exponent)`

`SQRT(number)`

Example	Result
<code>ROUND(12.5)</code>	13
<code>ROUND(12.4999, 0)</code>	12
<code>ROUND(12.4999, 1)</code>	12.5
<code>ROUND(12.4944, 2)</code>	12.49
<code>ROUND(1264.99, -2)</code>	1300
<code>TRUNC(12.5)</code>	12
<code>TRUNC(12.4999, 1)</code>	12.4
<code>TRUNC(12.4944, 2)</code>	12.49
<code>TRUNC(1264.99, -2)</code>	1200
<code>CEIL(1.25)</code>	2
<code>CEIL(-1.25)</code>	-1
<code>FLOOR(1.25)</code>	1
<code>FLOOR(-1.25)</code>	-2

Example	Result
<code>ABS(1.25)</code>	1.25
<code>ABS(-1.25)</code>	1.25
<code>SIGN(1.25)</code>	1
<code>SIGN(0)</code>	0
<code>SIGN(-1.25)</code>	-1
<code>MOD(10, 10)</code>	0
<code>MOD(10, 9)</code>	1
<code>POWER(2, 2)</code>	4
<code>POWER(2, 2.5)</code>	5.65685...
<code>SQRT(4)</code>	2
<code>SQRT(5)</code>	2.23606...

## The Float\_Sample table

	FLOAT_ID	FLOAT_VALUE
1		1 0.9999999999999999
2		2 1.0
3		3 1.0000000000000001
4		4 1234.56789012345
5		5 999.04440209348
6		6 24.04849

## A statement that searches for an exact value

```
SELECT *
FROM float_sample
WHERE float_value = 1
```

## The result set

	FLOAT_ID	FLOAT_VALUE
1		2 1.0

## A statement that searches for a range of values

```
SELECT *
FROM float_sample
WHERE float_value BETWEEN 0.99 AND 1.01
```

### The result set

FLOAT_ID	FLOAT_VALUE
1	10.99999999999999
2	21.0
3	31.000000000000001

## A statement that searches for rounded values

```
SELECT *
FROM float_sample
WHERE ROUND(float_value, 2) = 1
```

### The result set

	FLOAT_ID	FLOAT_VALUE
1		1 0.9999999999999999
2		2 1.0
3		3 1.0000000000000001

# Working with Dates

## Common date/time format elements

Element	Description	Element	Description
<b>AD</b>	Anno Domini	<b>DAY</b>	Name of day padded with spaces
<b>BC</b>	Before Christ	<b>DY</b>	Abbreviated name of day
<b>CC</b>	Century	<b>DDD</b>	Day of year (1-366)
<b>YEAR</b>	Year spelled out	<b>DD</b>	Day of month (01-31)
<b>YYYY</b>	Four-digit year	<b>D</b>	Day of week (1-7)
<b>YY</b>	Two-digit year	<b>HH</b>	Hour of day (01-12)
<b>RR</b>	Two-digit round year	<b>HH24</b>	Hour of day (01-24)
<b>Q</b>	Quarter of year (1-4)	<b>MI</b>	Minute (00-59)
<b>MONTH</b>	Name of month padded with spaces	<b>SS</b>	Second (00-59)
<b>MON</b>	Abbreviated name of month	<b>SSSSS</b>	Seconds past midnight (0-86399)
<b>MM</b>	Month (01-12)	<b>FF[1-9]</b>	Fractional seconds
<b>WW</b>	Week of year (1-52)	<b>PM</b>	Post Meridian
<b>W</b>	Week of month (1-5)	<b>AM</b>	Ante Meridian

**Practice:**  
Make something like this based on *invoices*

INVOICE_ID	INVOICE_DATE	INVOICE_TOTAL
1	TUE - FEB 25, 2014	\$116.54
2	FRI - MAR 14, 2014	\$1,083.58
3	FRI - APR 11, 2014	\$20,551.18
4	WED - APR 16, 2014	\$26,881.40
5	WED - APR 16, 2014	\$936.93
6	THU - APR 17, 2014	\$2,312.20
7	THU - APR 17, 2014	\$600.00
8	FRI - APR 18, 2014	\$1,927.54
9	SAT - APR 19, 2014	\$2,184.11
10	THU - APR 24, 2014	\$2,318.03

## Date Examples

(none specified)	19-AUG-14
DD-MON-YYYY	19-AUG-2014
DD-Mon-YY	19-Aug-14
MM/DD/YY	08/19/14
YYYY-MM-DD	2014-08-19
Dy Mon DD, YY	Tue Aug 19, 14
Month DD, YYYY B.C.	August 19, 2014 A.D.

## Time Examples

HH:MI	04:20
HH24:MI:SS	16:20:36
HH:MI AM	04:20 PM
HH:MI A.M.	04:20 P.M.
HH:MI:SS.FF5	04:20:36.12345
HH:MI:SS.FF4	04:20:36.1234
YYYY-MM-DD HH:MI:SS AM	2014-08-19 04:20:36 PM

## Examples that parse a date/time value

### Example

```
TO_CHAR(SYSDATE, 'DD-MON-RR HH:MI:SS') 19-AUG-14 04:20:36 PM
```

### Result

```
TO_CHAR(SYSDATE, 'YEAR') TWO THOUSAND FOURTEEN
```

```
TO_CHAR(SYSDATE, 'YEAR') Two Thousand Fourteen
```

```
TO_CHAR(SYSDATE, 'YYYY') 2014
```

```
TO_CHAR(SYSDATE, 'YY') 14
```

```
TO_CHAR(SYSDATE, 'MONTH') AUGUST
```

```
TO_CHAR(SYSDATE, 'MON') AUG
```

```
TO_CHAR(SYSDATE, 'MM') 08
```

```
TO_CHAR(SYSDATE, 'DD') 19
```

```
TO_CHAR(SYSDATE, 'DAY') TUESDAY
```

```
TO_CHAR(SYSDATE, 'DY') TUES
```

```
TO_CHAR(SYSDATE, 'HH24') 16
```

```
TO_CHAR(SYSDATE, 'HH') 04
```

### Example

```
TO_CHAR(SYSDATE, 'MI')
```

### Result

```
20
```

```
TO_CHAR(SYSDATE, 'SS')
```

```
36
```

```
TO_CHAR(SYSDATE, 'CC')
```

```
21
```

```
TO_CHAR(SYSDATE, 'Q')
```

```
3
```

```
TO_CHAR(SYSDATE, 'WW')
```

```
34
```

```
TO_CHAR(SYSDATE, 'W')
```

```
3
```

```
TO_CHAR(SYSDATE, 'DDD')
```

```
232
```

```
TO_CHAR(SYSDATE, 'D')
```

```
3
```

### Example

```
TO_NUMBER(TO_CHAR(SYSDATE, 'HH24')) 16
```

### Result

```
TO_NUMBER(TO_CHAR(SYSDATE, 'HH')) 4
```

```
TO_NUMBER(TO_CHAR(SYSDATE, 'SS')) 36
```

## Some common date/time functions

```

SYSDATE
CURRENT_DATE
ROUND(date[, date_format])
TRUNC(date[, date_format])
MONTHS_BETWEEN(date1, date2)
ADD_MONTHS(date, integer_months)
LAST_DAY(date)
NEXT_DAY(date, day_of_week)

```

## Two operators for working with dates

```
+  
-  
/
```

## Examples that use the date/time functions

Example	Result
SYSDATE	19-AUG-14 04:20:36 PM
ROUND(SYSDATE)	20-AUG-14 12:00:00 AM
TRUNC(SYSDATE, 'MI')	19-AUG-14 04:20:00 PM
MONTHS_BETWEEN('01-SEP-14','01-AUG-14')	1
MONTHS_BETWEEN('15-SEP-14','01-AUG-14')	1.451...
ADD_MONTHS('19-AUG-14', -1)	19-JUL-14
ADD_MONTHS('19-AUG-14', 11)	19-JUL-15
LAST_DAY('15-FEB-14')	29-FEB-14
NEXT_DAY('15-AUG-14', 'FRIDAY')	22-AUG-14
NEXT_DAY('15-AUG-14', 'THURS')	21-AUG-14
SYSDATE - 1	18-AUG-14
SYSDATE + 7	26-AUG-14
SYSDATE - TO_DATE('01-JAN-14')	231
TO_DATE('01-JAN-14') - SYSDATE	-231

## The Date\_Sample table

	DATE_ID	START_DATE
1		1 01-MAR-79
2		2 28-FEB-99
3		3 31-OCT-03
4		4 28-FEB-05
5		5 28-FEB-06
6		6 01-MAR-06

### A SELECT statement that fails to return a row

```
SELECT *
FROM date_sample
WHERE start_date = '28-FEB-06'
```

## A SELECT statement that searches for a range

```
SELECT *
FROM date_sample
WHERE start_date >= '28-FEB-06'
    AND start_date < '01-MAR-06'
```

### The result set

	DATE_ID	START_DATE
1	5	28-FEB-06

## A statement with TRUNC to remove time values

```
SELECT *
FROM date_sample
WHERE TRUNC(start_date) = '28-FEB-06'
```

### The result set

	DATE_ID	START_DATE
1	5	28-FEB-06

## The Date\_Sample table

	DATE_ID	START_DATE
1	1	01-MAR-79
2	2	28-FEB-99
3	3	31-OCT-03
4	4	28-FEB-05
5	5	28-FEB-06
6	6	01-MAR-06

### A SELECT statement that fails to return a row

```
SELECT *
FROM date_sample
WHERE start_date = TO_DATE('10:00:00', 'HH24:MI:SS')
```

## A statement that ignores the date component

```
SELECT *
FROM date_sample
WHERE TO_CHAR(start_date, 'HH24:MI:SS') = '10:00:00'
```

### The result set

	DATE_ID	START_DATE
1	4	28-FEB-05

## Another statement that fails to return a row

```
SELECT * FROM date_sample
WHERE start_date >= TO_DATE('09:00:00', 'HH24:MI:SS')
  AND start_date < TO_DATE('12:59:59', 'HH24:MI:SS')
```

## Another statement that ignores the date

```
SELECT * FROM date_sample
WHERE TO_CHAR(start_date, 'HH24:MI:SS') >= '09:00:00'
  AND TO_CHAR(start_date, 'HH24:MI:SS') < '12:59:59'
```

## The result set

DATE_ID	START_DATE
1	4 28-FEB-05
2	6 01-MAR-06

# Working with **Characters/Strings**

## Some common character functions

```
LTRIM(string[, trim_string])
RTRIM(string[, trim_string])
TRIM(string) Removes leading/trailing spaces
TRIM([trim_char FROM ]string)

LPAD(string, length[, pad_string])
RPAD(string, length[, pad_string])

LOWER(string)
UPPER(string)
INITCAP(string)
```

### Practice:

1. Select invoice\_date with MONTH format and then use TRIM to remove spaces
2. Left pad invoice\_total with 15 periods ''.
3. Select vendor\_name in ALL CAPS
4. Select vendor\_state in lowercase

## Character function examples

Example	Result
LTRIM(' John Smith ')	'John Smith '
RTRIM(' John Smith ')	' John Smith'
TRIM(' John Smith ')	'John Smith'
LTRIM('\$0019.99', '\$0')	'19.99'
TRIM('\$' FROM '\$0019.99')	'0019.99'
LPAD('\$19.99', 15)	'\$19.99'
LPAD('\$2150.78', 15)	'\$2150.78'
LPAD('\$2150.78', 15, '.')	'.....\$2150.78'
RPAD('John', 15)	'John'
RPAD('John', 15, '.')	'John.....'
LOWER('CA')	'ca'
UPPER('ca')	'CA'
INITCAP('john smith')	'John Smith'
INITCAP('JOHN SMITH')	'John Smith'

## More character functions

```
SUBSTR(string, start[, length])  
LENGTH(string)  
INSTR(string, find [,start])  
REPLACE(string, find, replace)
```

### Practice:

1. Select the vendor\_name and vendor\_name length
2. Select vendor\_phone twice. Once with no formatting and a 2<sup>nd</sup> time in the following format 999-999-9999. Try doing it two ways (i.e. one with REPLACE and another with SUBSTR)

## Character function examples (continued)

Example	Result
SUBSTR(' (559) 555-1212', 1, 5)	' (559) '
SUBSTR(' (559) 555-1212', 7, 3)	'555'
SUBSTR(' (559) 555-1212', 7)	'555-1212'
INSTR(' (559) 555-1212', ' ')	6
INSTR('559-555-1212', '-')	4
INSTR('559-555-1212', '-', 5)	8
INSTR('559-555-1212', '1212')	9
LENGTH(' (559) 555-1212')	14
LENGTH(' (559) 555-1212 ')	18
REPLACE('559-555-1212', '-', '.')	'559.555.1212'
REPLACE('559-555-1212', '-', '')	'5595551212'

## The String\_Sample table

ID	NAME
1 1	Lizbeth Darien
2 2	Darnell O'Sullivan
3 17	Lance Pinos-Potter
4 20	Jean Paul Renard
5 3	Alisha von Strump

## A SELECT statement that parses a string

```
SELECT SUBSTR(name, 1, (INSTR(name, ' ') - 1))  
      AS first_name,  
      SUBSTR(name, (INSTR(name, ' ') + 1))  
      AS last_name  
FROM string_sample
```

## The result set

	FIRST_NAME	LAST_NAME
1	Lizbeth	Darien
2	Darnell	O'Sullivan
3	Lance	Pinos-Potter
4	Jean	Paul Renard
5	Alisha	von Strump

## Practice:

The product\_name in **products** starts with the brand and is followed by the actual product's name.

Parse the field into two new fields:

1. *Brand* which contains the first word in the product\_name
2. *Instrument\_Name* which is any text that comes after the brand.

# Sorting char/varchar columns like #s

## A table sorted by a character column

```
SELECT * FROM string_sample  
ORDER BY id
```

## The result set (questionable results)

ID	NAME
1 1	Lizbeth Darien
2 17	Lance Pinos-Potter
3 2	Darnell O'Sullivan
4 20	Jean Paul Renard
5 3	Alisha von Strump



In this example table,  
“ID” is CHAR so it won’t  
sort like a number

## Solution: Convert character column to number

```
SELECT * FROM string_sample  
ORDER BY TO_NUMBER(id)
```

## The result set

ID	NAME
1 1	Lizbeth Darien
2 2	Darnell O'Sullivan
3 3	Alisha von Strump
4 17	Lance Pinos-Potter
5 20	Jean Paul Renard

## Another option: Pad character column with leading 0 to make sortable

```
SELECT LPAD(id, 2, '0') AS lpad_id, name
FROM string_sample
ORDER BY lpad_id
```

### The result set

LPAD_ID	NAME
1 01	Lizbeth Darien
2 02	Darnell O'Sullivan
3 03	Alisha von Strumpf
4 17	Lance Pinos-Potter
5 20	Jean Paul Renard

# Handling Conditional with CASE

## The syntax of the simple CASE expression

```
CASE input_expression
    WHEN when_expression_1 THEN result_expression_1
    [WHEN when_expression_2 THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

## The syntax of the searched CASE expression

```
CASE
    WHEN conditional_expression_1 THEN result_expression_1
    [WHEN conditional_expression_2
        THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

## The syntax of the simple CASE expression

```
CASE input_expression
    WHEN when_expression_1 THEN result_expression_1
    [WHEN when_expression_2 THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

## A statement that uses a simple CASE expression

```
SELECT invoice_number, terms_id,
CASE terms_id
    WHEN 1 THEN 'Net due 10 days'
    WHEN 2 THEN 'Net due 20 days'
    WHEN 3 THEN 'Net due 30 days'
    WHEN 4 THEN 'Net due 60 days'
    WHEN 5 THEN 'Net due 90 days'
END AS terms
FROM invoices
```

## The result set

INVOICE_NUMBER	TERMS_ID	TERMS
1 QP58872	1	Net due 10 days
2 Q545443	1	Net due 10 days
3 P-0608	5	Net due 90 days

## Category Names

- 1 = 'Guitars'
- 2 = 'Bass'
- 3 = 'Drums'
- 4 = 'Keyboard'

Sort by category, price

PRODUCT_NAME	LIST_PRICE	CATEGORY_NAME
Hofner Icon	499.99	Bass
Fender Precision	799.99	Bass
Ludwig 5-piece Drum Set with Cymbals	699.99	Drums
Tama 5-Piece Drum Set with Cymbals	799.99	Drums
Washburn D10S	299	Guitars
Rodriguez Caballero 11	415	Guitars
Yamaha FG700S	489.99	Guitars
Fender Stratocaster	699	Guitars
Gibson Les Paul	1199	Guitars
Gibson SG	2517	Guitars

## The syntax of the searched CASE expression

CASE

```
    WHEN conditional_expression_1 THEN result_expression_1
    [WHEN conditional_expression_2
        THEN result_expression_2]...
    [ELSE else_result_expression]
```

END

## A statement that uses a searched CASE

```
SELECT invoice_number, invoice_total, invoice_date,
       invoice_due_date,
CASE
    WHEN (SYSDATE - invoice_due_date) > 30
        THEN 'Over 30 days past due'
    WHEN (SYSDATE - invoice_due_date) > 0
        THEN '1 to 30 days past due'
    ELSE 'Current'
END AS status
FROM invoices
WHERE invoice_total - payment_total - credit_total > 0
```

**Product Grade**

>= 1000 = 'Professional'  
>= 500 = 'Intermediate'  
Everything else = 'Beginner'

## The result set

INVOICE_NUMBER	INVOICE_TOTAL	INVOICE_DATE	INVOICE_DUE_DATE	STATUS
37 547481328	224	20-MAY-08	25-JUN-08	Over 30 days past due
38 40318	21842	18-JUL-08	20-JUL-08	Over 30 days past due
39 31361833	579.42	23-MAY-08	09-JUN-08	Over 30 days past due
40 456789	8344.5	01-AUG-14	31-AUG-14	Current

# Handling NULL values

# The COALESCE, NVL, and NVL2 syntax

```
COALESCE(expression1 [, expression2] [, expression3]...)
NVL(expression, null_replacement)
NVL2(expression, not_null_replacement, null_replacement)
```

## A statement that uses the COALESCE function

```
SELECT payment_date, invoice_due_date,
       COALESCE(payment_date, invoice_due_date,
                TO_DATE('01-JAN-1900'))
      AS payment_date_2
   FROM invoices
```

## The result set

	PAYMENT_DATE	INVOICE_DUE_DATE	PAYMENT_DATE_2
1	11-APR-08	22-APR-08	11-APR-08
2	14-MAY-08	23-MAY-08	14-MAY-08
3	(null)	30-JUN-08	30-JUN-08
4	12-MAY-08	16-MAY-08	12-MAY-08
5	13-MAY-08	16-MAY-08	13-MAY-08
6	(null)	26-JUN-08	26-JUN-08

# The COALESCE, NVL, and NVL2 syntax

```
COALESCE(expression1 [, expression2] [, expression3]...)
NVL(expression, null_replacement)
NVL2(expression, not_null_replacement, null_replacement)
```

## A SELECT statement that uses the NVL function

```
SELECT payment_date,
       NVL(TO_CHAR(payment_date), 'Unpaid') AS payment_date_2
FROM invoices
```

### The result set

	PAYMENT_DATE	PAYMENT_DATE_2
1	11-APR-08	11-APR-08
2	14-MAY-08	14-MAY-08
3	(null)	Unpaid
4	12-MAY-08	12-MAY-08

## A SELECT statement that uses the NVL2 function

```
SELECT payment_date,
       NVL2(payment_date, 'Paid', 'Unpaid') AS payment_date_2
FROM invoices
```

### The result set

	PAYMENT_DATE	PAYMENT_DATE_2
1	11-APR-08	Paid
2	14-MAY-08	Paid
3	(null)	Unpaid
4	12-MAY-08	Paid

## Practice:

1. Pull a report that checks vendor contact is up-to-date like so.

VENDOR_ID	VENDOR_NAME	CONTACT_CHECK
81	Wang Laboratories, Inc.	(800) 555-0344
82	Reiter's Scientific & Pro Books	(202) 555-5561
83	Ingram	Update contact
84	Boucher Communications Inc	(215) 555-8000
85	Champion Printing Company	(800) 555-1957
86	Computerworld	(617) 555-0700
87	DMV Renewal	Update contact
88	Edward Data Services	(513) 555-3043
89	Evans Executone Inc	Update contact
90	Wakefield Co	(559) 555-4744

2. Update to show not null values as 'Okay' & sort

VENDOR_ID	VENDOR_NAME	CONTACT_CHECK
40	Nat Assoc of College Stores	Update contact
58	Fresno Rack & Shelving Inc	Update contact
60	The Mailers Guide Co	Update contact
64	Texaco	Update contact
65	The Drawing Board	Update contact
66	Ascom Hasler Mailing Systems	Update contact
87	DMV Renewal	Update contact
72	Data Reproductions Corp	Okay
73	Executive Office Products	Okay
74	Leslie Company	Okay
75	Retirement Plan Consultants	Okay
76	Simon Direct Inc	Okay
77	State Board Of Equalization	Okay
78	The Present Center	Okay

# Rank & Row Numbers (Using Window Functions)

## A query that uses RANK and DENSE\_RANK

```
SELECT RANK() OVER (ORDER BY invoice_total) AS rank,  
       DENSE_RANK() OVER (ORDER BY invoice_total)  
           AS dense_rank,  
       invoice_total, invoice_number  
FROM invoices
```

### The result set

RANK	DENSE_RANK	INVOICE_TOTAL	INVOICE_NUMBER
1	1	1	6 25022117
2	1	1	6 24863706
3	1	1	6 24780512
4	4	2	9.95 21-4748363
5	4	2	9.95 21-4923721
6	6	3	10 4-342-8069

### Practice:

Pull a list of products and their rank by list\_price descending.

Which is the 2<sup>nd</sup> ranked product by price?

PRODUCT_NAME	LIST_PRICE	RANK
Gibson SG	2517	1
Gibson Les Paul	1199	2
Tama 5-Piece Drum Se...	799.99	3
Fender Precision	799.99	3
Ludwig 5-piece Drum ...	699.99	5
Fender Stratocaster	699	6
Hofner Icon	499.99	7
Yamaha FG700S	489.99	8
Rodriguez Caballero 11	415	9
Washburn D10S	299	10

## A query that uses the ROW\_NUMBER function

```
SELECT ROW_NUMBER() OVER(ORDER BY vendor_name)
      AS row_number, vendor_name
FROM vendors
```

### The result set

	ROW_NUMBER	VENDOR_NAME
1	1	ASC Signs
2	2	AT&T
3	3	Abbey Office Furnishings
4	4	American Booksellers Assoc
5	5	American Express

# MIS 381N INTRO. TO DATABASE MANAGEMENT

---

PL/SQL

**Tayfun Keskin**

Visiting Clinical Professor, The University of Texas at Austin, McCombs School of Business  
Associate Teaching Professor, University of Washington Seattle, Foster School of Business

# QUESTIONS

- Any questions before we begin?



# AGENDA



Lecture

PL/SQL



Hands-On

Exercises



Looking Forward

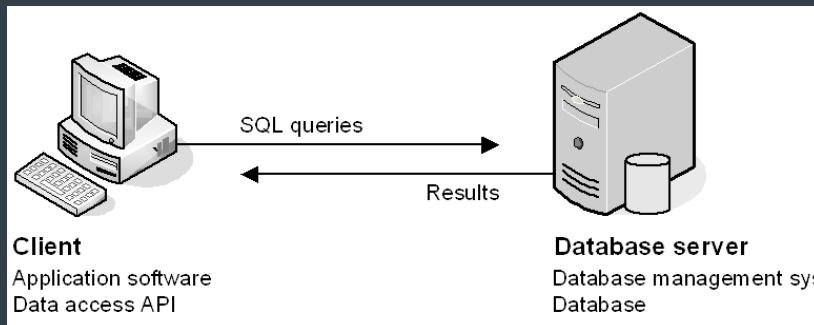
Homework 4  
Exam 2



The University of Texas at Austin  
McCombs School of Business

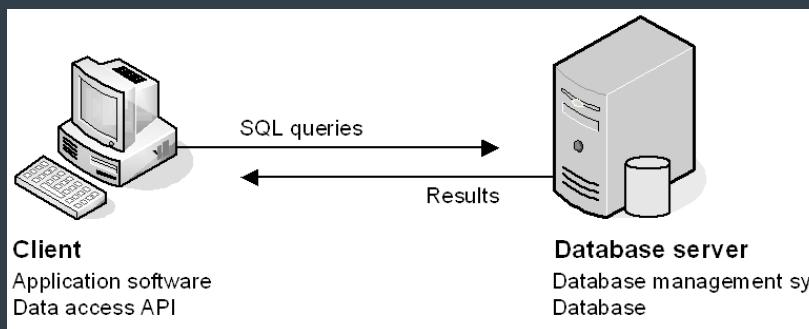
# REMINDER QUESTION

Where do you write and run SQL?



# REMINDER QUESTION

Where do tables get created and saved?



# Oracle 19c has a Programming Language Built in

- It's called PL/SQL
- It stands for “**Procedural Language extensions to the Structured Query Language**”
- Expands your ability to fine tune control of the Oracle database
- PL/SQL is a full 3<sup>rd</sup> generation programming language, such as C++



# WHAT CAN I DO WITH PL/SQL?

- Declare variables
- Limiting scope of blocks of code
- Using loops and conditional statements (FOR, WHILE, IF)
- Call external functions
- Processing results

# WHY IS PL/SQL IMPORTANT?

- The integration of Oracle DB and PL/SQL is very tight
- All data types used by one are available to the other
  - Such as VARCHAR2, NUMBER, DATE
- Transparently uses declared data types with %TYPE and %ROWTYPE without knowing the exact data type at runtime

# QUESTION

What is the difference between a procedural and declarative programming language?

(Hint: SQL is primarily declarative)

# DIFFERENCE BETWEEN SQL AND PL/SQL

- SQL is declarative:
  - You write the code as “Here’s what I need to do”
  - Optimizer figures out the best way
- PL/SQL is procedural:
  - You write the code as “Here’s how to do what I want”
  - Optimizer is not involved



# PL/SQL FEATURES

- Functions: process zero or more variables and return a variable of any data type
- Procedures: process zero or more arguments in a stored block of code
- Variable declarations: common data types and cursors
- FOR and WHILE loops with CONTINUE and EXIT
- IF-THEN-ELSE statement
- Exception handling

# WHAT IS A DB FUNCTION?

- A sequence of SQL and PL/SQL statements stored by name
- Stored in the DB's data dictionary that you can call again later
- It can have zero arguments or dozens (usually a few)
- Returns one value of any datatype, even a pointer
- A.K.A.: user function, user-defined function, stored function

# CREATE FUNCTION SYNTAX

```
CREATE [OR REPLACE] FUNCTION function_name
()
  parameter_name_1 data_type
  [, parameter_name_2 data_type]...
)
RETURN data_type
{IS | AS}
pl_sql_block
```

```
-- Example: A function that returns a vendor ID
CREATE OR REPLACE FUNCTION get_vendor_id
(
    vendor_name_param VARCHAR2
)
RETURN NUMBER
AS
    vendor_id_var NUMBER;
BEGIN
    SELECT vendor_id
    INTO vendor_id_var
    FROM vendors
    WHERE vendor_name = vendor_name_param;

    RETURN vendor_id_var;
END;
/
```



# QUESTION

If your function compiles without any errors, does it mean your function is without an error?

Or, useful?

# WHAT IS A DB PROCEDURE?

- Companion to stored functions
- A sequence of SQL and PL/SQL statements stored by name
- It can be anonymous (it doesn't get stored permanently)
- It can have zero or more arguments, but **do not return values**
- It can be stored by itself, or stored in a package

# WHAT? NO RETURN VALUE?

- It cannot be called directly from a SELECT statement  
(because it's not a function, it's not part of the SQL language)
- You can still run it with an EXEC or CALL statement
- You can modify and return values if one of the parameters is declared as OUT in the argument list (not very common)

# CREATE PROCEDURE SYNTAX

```
CREATE [OR REPLACE] PROCEDURE proc_name
()
  parameter_name_1 data_type
  [, parameter_name_2 data_type]...
[])
{IS | AS}

pl_sql_block
```

```
-- Example: A sproc that updates a table
CREATE OR REPLACE PROCEDURE
update_invoices_credit_total
(
  invoice_number_param  VARCHAR2,
  credit_total_param    NUMBER
)
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```



# TRUE OR FALSE?

A stored procedure can modify and return  
parameters defined as **OUT**

# WHAT IS A VARIABLE?

- A variable is a meaningful name which facilitates a programmer to store data temporarily during the execution of code
- Declaring variables is optional, but you'll need to put something in the declaration section to take full advantage of PL/SQL
- A variable can be CONSTANT (I know it's ironic)

# DATA TYPES ALLOWED

- All data types are allowed in the declaration section
- If you don't want to change the code, you can define a variable with the `%TYPE` keyword (it automatically finds the data type of the column at runtime)
- You can define an entire column with `%ROWTYPE`



# DECLARING VARIABLES

- A procedure of any type can have an optional DECLARE section
- You can only DECLARE in an anonymous block; otherwise, use IS
- Declared variables can be initialized, uninitialized (NULL)
  - A constant, by definition, is always initialized
- If you want to avoid changing your code as your table or column attributes change, use %TYPE or %ROWTYPE keywords and it will help you avoid recompilations in the future

# WHAT IS A CURSOR?

- When an SQL statement is processed, Oracle creates a memory area known as context area
- A cursor is a pointer to this context area
- It contains all information needed for processing the statement
- In PL/SQL, the context area is controlled by Cursor
- A cursor contains information on a select statement and the rows of data accessed by it.

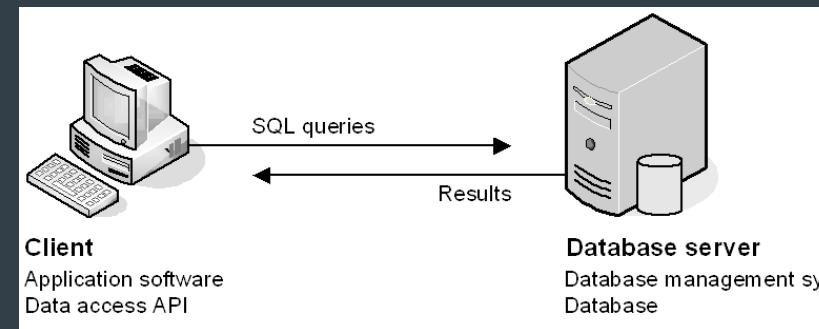
# TYPES OF CURSORS

- **Implicit Cursors:** are automatically generated by Oracle while an SQL statement is executed (if you don't use an explicit cursor)
- **Explicit Cursors:** are defined by the programmers to gain more control over the context area
  - These cursors should be defined in the declaration section of the PL/SQL block
  - It is created on a SELECT statement which returns more than one row

# DISCUSSION QUESTIONS

Where does anonymous PL/SQL is written and run?

Where does a stored procedure is saved and run?



# LOOKING FORWARD

**Read Chapters 3-8**

- 13-15: PL/SQL

**Quiz 4**

**Homework 4**

**Exam 2**



The University of Texas at Austin  
McCombs School of Business

# THANK YOU



The University of Texas at Austin  
McCombs School of Business

# **BACKUP SLIDES**



# PART 1

Chapter 13

PL/SQL Introduction



The University of Texas at Austin  
McCombs School of Business

# The syntax for an anonymous PL/SQL block

Optional    **DECLARE**

```
    declaration_statement_1;
    [declaration_statement_2;]...
```

**BEGIN**

```
    body_statement_1;
    [body_statement_2;]...
```

Optional    **EXCEPTION**

```
    WHEN OTHERS THEN
        exception_handling_statement_1;
        [exception_handling_statement_2;]...
```

**END;**

          /

## A script with an anonymous PL/SQL block (i.e. code snippet is nameless)

```
--CONNECT ap/ap;
SET SERVEROUTPUT ON;

DECLARE
    sum_balance_due_var NUMBER(9, 2);
BEGIN
    SELECT SUM(invoice_total - payment_total - credit_total)
    INTO sum_balance_due_var
    FROM invoices
    WHERE vendor_id = 95;
    IF sum_balance_due_var > 0 THEN
        DBMS_OUTPUT.PUT_LINE('Balance due: $' || ROUND(sum_balance_due_var, 2));
    ELSE
        DBMS_OUTPUT.PUT_LINE('Balance paid in full');
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred');
END;
/
```

# Procedures for printing output to the screen

**DBMS\_OUTPUT.ENABLE()** --out of scope and only for old versions

**DBMS\_OUTPUT.PUT(string)** = output string

**DBMS\_OUTPUT.PUT\_LINE(string)** = output string and move to next line

# Commands for working with scripts

CONNECT

**SET SERVEROUTPUT ON;**

## PL/SQL statements for controlling the flow

**IF...ELSIF...ELSE**

CASE...WHEN...ELSE

**FOR...IN...LOOP**

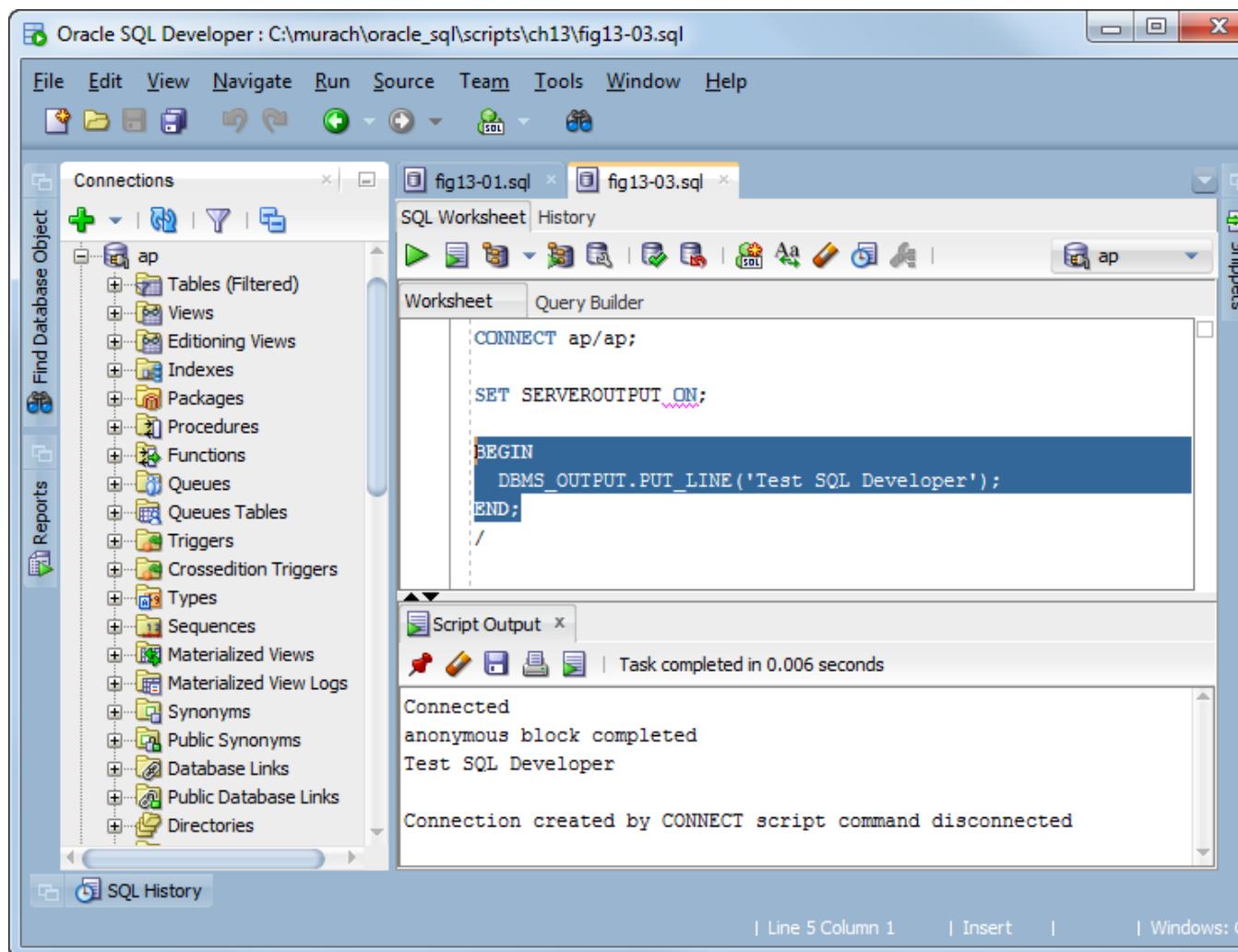
WHILE...LOOP

LOOP...EXIT WHEN

**CURSOR...IS**

EXECUTE IMMEDIATE

# How to print data to the Script Output window



# Declaring **VARIABLES**

## The syntax for declaring a variable

```
variable_name_1 DATA_TYPE;
```

```
percent_difference NUMBER;
```

## The syntax for declaring a variable with the same data type as a column

```
variable_name_1 table_name.column_name%TYPE;
```

```
max_invoice_total invoices.invoice_total%TYPE;
```

## The syntax for setting a variable to a selected value

```
SELECT column_1[, column_2]...  
INTO variable_name_1[, variable_name_2]...;
```

```
SELECT MAX(invoice_total), MIN(invoice_total)  
INTO max_invoice_total, min_invoice_total
```

## The syntax for setting a variable to a literal value or the result of an expression

```
variable_name := literal_value_or_expression
```

```
vendor_id_var
```

```
NUMBER := 95;
```

# A SQL script that uses variables

```
DECLARE
    max_invoice_total    invoices.invoice_total%TYPE;
    min_invoice_total    invoices.invoice_total%TYPE;
    percent_difference  NUMBER;
    count_invoice_id    NUMBER;
    vendor_id_var       NUMBER := 95;
BEGIN
    SELECT MAX(invoice_total), MIN(invoice_total), COUNT(invoice_id)
    INTO max_invoice_total, min_invoice_total, count_invoice_id
    FROM invoices WHERE vendor_id = vendor_id_var;

    percent_difference :=
        (max_invoice_total - min_invoice_total) /
        min_invoice_total * 100;

    DBMS_OUTPUT.PUT_LINE('Maximum invoice: $' ||
                         max_invoice_total);
    DBMS_OUTPUT.PUT_LINE('Minimum invoice: $' ||
                         min_invoice_total);
    DBMS_OUTPUT.PUT_LINE('Percent difference: %' ||
                         ROUND(percent_difference, 2));
    DBMS_OUTPUT.PUT_LINE('Number of invoices: ' ||
                         count_invoice_id);

END;
/
```

# Handling CONDITIONS

# The syntax of the IF statement

```
IF boolean_expression THEN
    statement_1;
    [statement_2;]...
[ELSIF boolean_expression THEN
    statement_1;
    [statement_2;]...]...
[ELSE
    statement_1;
    [statement_2;]...]
END IF;
```

NOTE spelling is  
ELSIF, not ELSEIF

## A script that uses an IF statement

```
CONNECT ap/ap;
SET SERVEROUTPUT ON;

DECLARE
    first_invoice_due_date DATE;
BEGIN
    SELECT MIN(invoice_due_date)
    INTO first_invoice_due_date
    FROM invoices
    WHERE invoice_total - payment_total - credit_total > 0;

    IF first_invoice_due_date < SYSDATE() THEN
        DBMS_OUTPUT.PUT_LINE('Invoices overdue!');
    ELSIF first_invoice_due_date = SYSDATE() THEN
        DBMS_OUTPUT.PUT_LINE('Invoices are due today!');
    ELSE
        DBMS_OUTPUT.PUT_LINE('No invoices are overdue.');
    END IF;
END;
/
```

### The response from the system

Invoices overdue!

# The syntax of the Simple CASE statement

```
CASE expression
    WHEN expression_value_1 THEN
        statement_1;
        [statement_2;]...
    [WHEN expression_value_2 THEN
        statement_1;
        [statement_2;]...]...
    [ELSE
        statement_1;
        [statement_2;]...]...
END CASE;
```

## A script that uses a Simple CASE statement

```
CONNECT ap/ap;
SET SERVEROUTPUT ON;

DECLARE
    terms_id_var NUMBER;
BEGIN
    SELECT terms_id INTO terms_id_var
    FROM invoices WHERE invoice_id = 4;

    CASE terms_id_var
        WHEN 1 THEN
            DBMS_OUTPUT.PUT_LINE('Net due 10 days');
        WHEN 2 THEN
            DBMS_OUTPUT.PUT_LINE('Net due 20 days');
        WHEN 3 THEN
            DBMS_OUTPUT.PUT_LINE('Net due 30 days');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Net due more than 30 days');
    END CASE;
END;
/
```

# The syntax of a Searched CASE expression

```
CASE
    WHEN boolean_expression THEN
        statement_1;
        [statement_2;]...
    [WHEN boolean_expression THEN
        statement_1;
        [statement_2;]...]...
[ELSE
    statement_1;
    [statement_2;]...]
END CASE;
```

No variable included  
in this kind of CASE

# Repeating code with LOOPS

## The syntax of the FOR loop

```
FOR counter_var IN [REVERSE]
    counter_start..counter_end LOOP
    statement_1;
    [statement_2;] ...
END LOOP;
```

### A FOR loop

```
FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE('i: ' || i);
END LOOP;
```

### The output for the loop

```
i: 1
i: 2
i: 3
```

## The syntax for a WHILE loop

```
WHILE boolean_expression LOOP
    statement_1;
    [statement_2;] ...
END LOOP;
```

### A WHILE loop

```
i := 1;
WHILE i < 4 LOOP
    DBMS_OUTPUT.PUT_LINE('i: ' || i);
    i := i + 1;
END LOOP;
```

### The output for the loop

```
i: 1
i: 2
i: 3
```

## The syntax for a simple loop

```
LOOP
    statement_1;
    [statement_2]...
    EXIT WHEN boolean_expression;
END LOOP;
```

## A simple loop

```
i := 1;
LOOP
    DBMS_OUTPUT.PUT_LINE('i: ' || i);
    i := i + 1;
    EXIT WHEN i >= 4;
END LOOP;
```

## The output for the loop

```
i: 1
i: 2
i: 3
```

## **CAUTION**

- Stick to FOR or WHILE for simplicity's sake.
- Also beware using CONTINUE and CONTINUE WHEN. These are rarely used and EXIT WHEN can suffice.
- Also careful about writing an ENDLESS LOOP which is.....?

# Storing a list many values with Arrays, **BULK COLLECT**, & **CURSOR**

## A script that uses a BULK COLLECT clause to populate a nested table

```
DECLARE
    TYPE names_table          IS TABLE OF VARCHAR2(40);
    vendor_names;
BEGIN
    SELECT vendor_name
    BULK COLLECT INTO vendor_names
    FROM vendors
    WHERE rownum < 4
    ORDER BY vendor_id;

    FOR i IN 1..vendor_names.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Vendor name: ' || vendor_names(i));
    END LOOP;
END;
/
```

Helpful when pulling a column (i.e., a list) of values

## The response from the system

```
Vendor name: US Postal Service
Vendor name: National Information Data Ctr
Vendor name: Register of Copyrights
```

## Practice

Update example to respond with first 10 vendor names.

Also add in vendor # (e.g. Vendor Name 1: ) on each line like so:

```
Vendor name 1: US Postal Service
Vendor name 2: National Information Data Ctr
```

# Cursors

```
DECLARE
  CURSOR invoices_cursor IS          --declare a variable for cursor
    SELECT invoice_id, invoice_total
    FROM invoices
    WHERE invoice_total - payment_total - credit_total > 0;

  invoice_row invoices%ROWTYPE;      --declare a variable for each row

BEGIN
  FOR invoice_row IN invoices_cursor LOOP  --for each row in the dataset
    . . .
  END LOOP;
END;
/
```

Helpful when pulling a table of values

	INVOICE_ID	INVOICE_TOTAL
1	3	20551.18
2	6	2312.2
3	8	1927.54
4	15	313.55
5	18	904.14
6	19	1962.13
7	30	17.5
8	34	10976.06
9	38	61.5
10	39	158
11	40	26.75
12	41	23.5
13	42	9.95
14	44	52.25

# Handling ERRORS

# A script that doesn't handle exceptions

```
CONNECT ap/ap;  
INSERT INTO general_ledger_accounts VALUES (130, 'Cash');
```

## The response from the system

SQL Error: ORA-00001: unique constraint  
(AP.GL\_ACCOUNT\_DESCRIPTION\_UQ) violated 00001. 00000 -  
"unique constraint (%s.%s) violated"

\*Cause: An UPDATE or INSERT statement attempted to insert  
duplicate key. For Trusted Oracle configured in  
DBMS MAC mode, you may see this message if a  
duplicate entry exists at a different level.

\*Action: Either remove the unique restriction or do not  
insert the key.

# The syntax of the EXCEPTION block

**EXCEPTION**

```
WHEN most_specific_exception THEN
    statement_1;
    [statement_2;]...
```

```
[WHEN less_specific_exception THEN
    statement_1;
    [statement_2;]...]...
```

## An EXCEPTION block that handles exceptions

```
CONNECT ap/ap;
SET SERVEROUTPUT ON;

BEGIN
    INSERT INTO general_ledger_accounts VALUES (130, 'Cash');

    DBMS_OUTPUT.PUT_LINE('1 row inserted.');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE(
            'You attempted to insert a duplicate value.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(
            'An unexpected exception occurred.');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

## The response from the system

You attempted to insert a duplicate value.

# A list of common exceptions

ORA Error	Exception
00001	DUP_VAL_ON_INDEX
01403	NO_DATA_FOUND
01476	ZERO_DIVIDE
01722	INVALID_NUMBER
06502	VALUE_ERROR

## A script that will display an error if the object doesn't already exist

```
CONNECT ap/ap;  
DROP TABLE test1;  
CREATE TABLE test1 (test_id NUMBER);
```

## The response from the system

Connected

Error starting at line 2 in command:

```
DROP TABLE test1
```

Error report:

```
SQL Error: ORA-00942: table or view does not exist  
00942. 00000 - "table or view does not exist"
```

\*Cause:

\*Action:

```
CREATE TABLE succeeded.
```

# Executing SQL PL/SQL using **EXECUTE IMMEDIATE**

# The EXECUTE IMMEDIATE statement

```
EXECUTE IMMEDIATE 'sql_string'
```

## A script that won't display an error

```
CONNECT ap/ap;

BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE test1';
EXCEPTION
    WHEN OTHERS THEN
        NULL;
END;
/
```

```
CREATE TABLE test1 (test_id NUMBER);
```

## The response from the system

```
Connected
anonymous block completed
CREATE TABLE succeeded.
```

# Prompting user entry with **SUBSTITUTION VARIABLES**

# A script that uses substitution variables

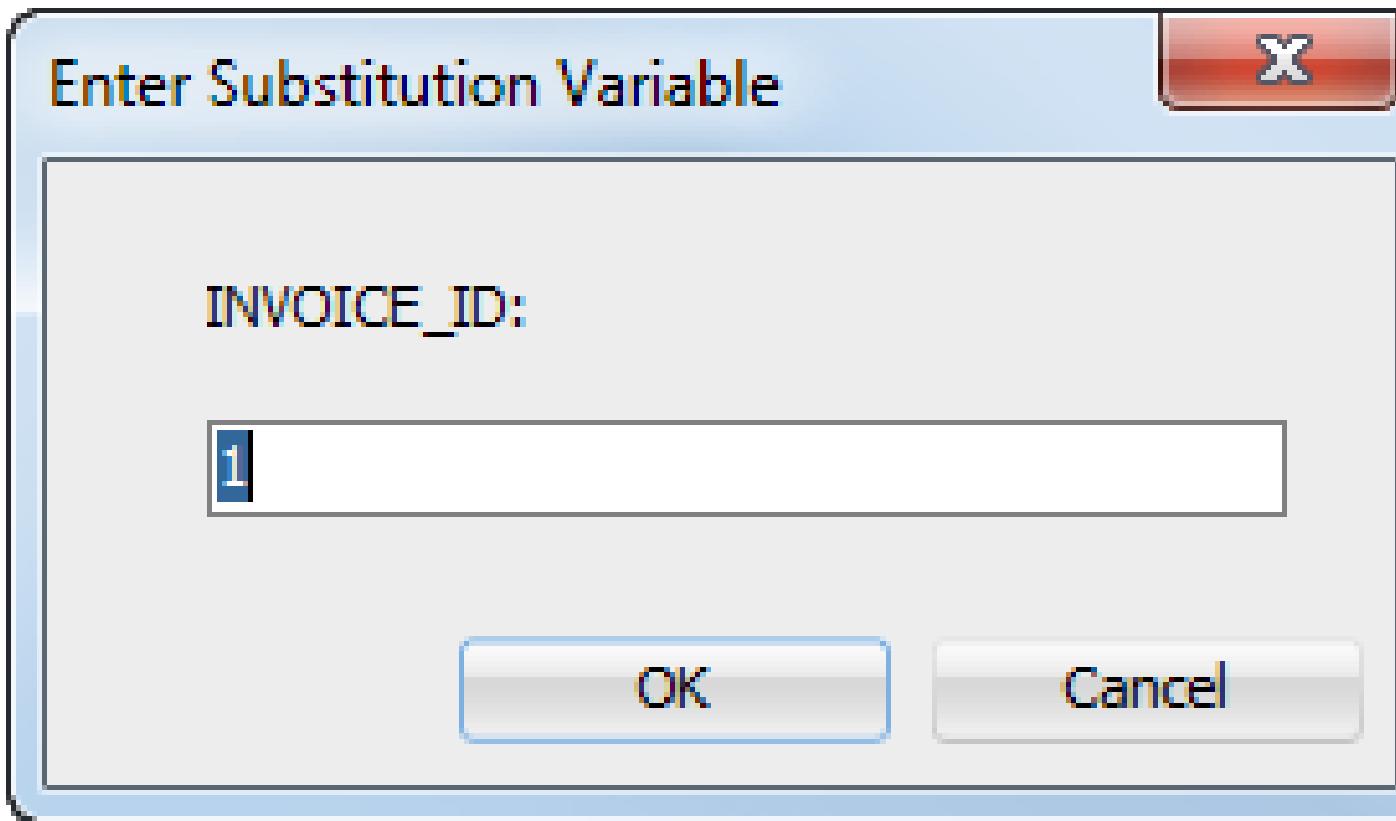
```
-- Use the VARIABLE keyword to declare a bind variable
VARIABLE invoice_id_value NUMBER;

-- Use a PL/SQL block to set the value of a bind variable
-- to the value of a substitution variable
BEGIN
    invoice_id_value := &invoice_id;
END;
/

-- Use a bind variable in a SELECT statement
SELECT invoice_id, invoice_number
FROM invoices
WHERE invoice_id = invoice_id_value;

-- Use a bind variable in another PL/SQL block
BEGIN
    DBMS_OUTPUT.PUT_LINE('invoice_id_value: ' || invoice_id_value);
END;
/
```

# The dialog box for a substitution variable



# The response from the system

Connected

anonymous block completed

INVOICE_ID	INVOICE_NUMBER
------------	----------------

1	QP58872
---	---------

1 rows selected

anonymous block completed

invoice\_id\_value: 1

# How to concatenate text/variables in **DYNAMIC SQL**

# Dynamic SQL that updates a specified invoice

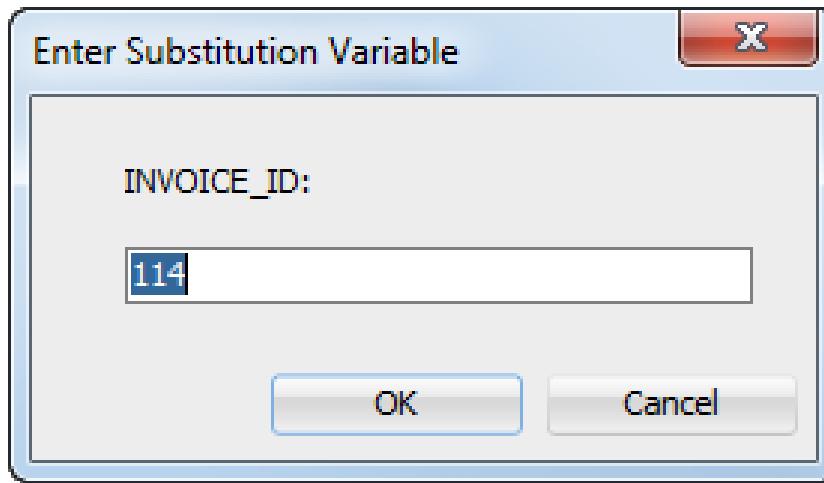
```
CONNECT ap/ap;

DECLARE
    invoice_id_var NUMBER;
    terms_id_var NUMBER;
    dynamic_sql VARCHAR2(400);
BEGIN
    invoice_id_var := &invoice_id;
    terms_id_var := &terms_id;

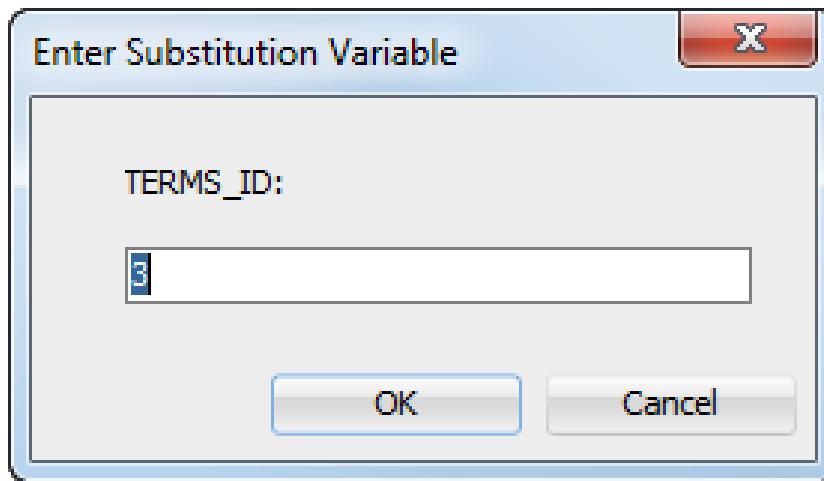
    dynamic_sql := 'UPDATE invoices ' ||
                   'SET terms_id = ' || terms_id_var || ' ' ||
                   'WHERE invoice_id = ' || invoice_id_var;

    EXECUTE IMMEDIATE dynamic_sql;
END;
/
```

## The first dialog box for a substitution variable



## The second dialog box for a substitution variable



# The contents of the variable named dynamic\_sql at runtime

```
UPDATE invoices SET terms_id = 3 WHERE invoice_id = 114
```

# Appendix

## The syntax for declaring a cursor

```
CURSOR cursor_name IS select_statement;
```

## The syntax for declaring a variable for a row

```
row_variable_name table_name%ROWTYPE;
```

## The syntax for getting a column value from a row variable

```
row_variable_name.column_name
```

# A script that uses a cursor

```
DECLARE
    CURSOR invoices_cursor IS
        SELECT invoice_id, invoice_total FROM invoices
        WHERE invoice_total - payment_total - credit_total > 0;

    invoice_row invoices%ROWTYPE;

BEGIN
    FOR invoice_row IN invoices_cursor LOOP

        IF (invoice_row.invoice_total > 1000) THEN
            UPDATE invoices
            SET credit_total = credit_total + (invoice_total * .1)
            WHERE invoice_id = invoice_row.invoice_id;
            DBMS_OUTPUT.PUT_LINE(
                '1 row updated where invoice_id = ' ||
                invoice_row.invoice_id);
        END IF;

    END LOOP;
END;
/
```

# The response from the system

```
1 row updated where invoice_id = 3
1 row updated where invoice_id = 6
1 row updated where invoice_id = 8
1 row updated where invoice_id = 19
1 row updated where invoice_id = 34
1 row updated where invoice_id = 81
1 row updated where invoice_id = 88
1 row updated where invoice_id = 113
```

# PART 2

Chapter 15

Stored Procedures and Functions



The University of Texas at Austin  
McCombs School of Business

## CREATE PROCEDURE syntax

```
CREATE [OR REPLACE] PROCEDURE proc_name
[(
    parameter_name_1 data_type
    [, parameter_name_2 data_type]...
)]
{IS | AS}
```

pl\_sql\_block

## Example: sproc that updates a table

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
    invoice_number_param    VARCHAR2,
    credit_total_param      NUMBER
)

AS

BEGIN
    UPDATE invoices
    SET credit_total = credit_total_param
    WHERE invoice_number = invoice_number_param;

    COMMIT;

    EXCEPTION
        WHEN OTHERS THEN
            ROLLBACK;
    END;
    /

```

## Example: sproc that updates a table

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
    invoice_number_param  VARCHAR2,
    credit_total_param     NUMBER
)
AS
BEGIN
    UPDATE invoices
    SET credit_total = credit_total_param
    WHERE invoice_number = invoice_number_param;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END;
/
```

### SQL statement that calls the sproc

```
CALL update_invoices_credit_total ('367447', 300);
```

### Script that calls the sproc

```
BEGIN
    update_invoices_credit_total ('367447', 300);
END;
/
```

### Script that passes parameters by name

```
BEGIN
    update_invoices_credit_total(
        credit_total_param => 300,
        invoice_number_param => '367447'
    );
END;
/
```

## The syntax for declaring an optional parameter

```
parameter_name_1 data_type [DEFAULT default_value]
```

## A statement that uses an optional parameter

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
    invoice_number_param VARCHAR2,
    credit_total_param    NUMBER      DEFAULT 100
)
AS
BEGIN
    UPDATE invoices
    SET credit_total = credit_total_param
    WHERE invoice_number = invoice_number_param;

    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END;
/
```

## Practice

### A statement that calls the stored procedure

```
CALL update_invoices_credit_total('367447', 200);
```

### Another statement that calls the stored procedure

```
CALL update_invoices_credit_total('367447');
```

# User -Defined FUNCTIONS

# User Defined Functions

- A *user-defined function (UDF)*, which can also be called a *stored function* or just a *function*, is an executable database object that contains a block of PL/SQL code.
- A *scalar-valued function* returns a single value of any data type.
- A function can accept input parameters that work like the input parameters for a stored procedure.
- A function always returns a value. You use the RETURN keyword to specify the value that's returned by the function.
- A function can't make changes to the database such as executing an INSERT, UPDATE, or DELETE statement.

## The syntax for creating a function

```
CREATE [OR REPLACE] FUNCTION function_name
[(
    parameter_name_1 data_type
    [, parameter_name_2 data_type]...
)]
RETURN data_type
{IS | AS}
pl_sql_block
```

## e.g. A function that returns a vendor ID

```
CREATE OR REPLACE FUNCTION get_vendor_id
(
    vendor_name_param VARCHAR2
)
RETURN NUMBER
AS
    vendor_id_var NUMBER;
BEGIN
    SELECT vendor_id
    INTO vendor_id_var
    FROM vendors
    WHERE vendor_name = vendor_name_param;

    RETURN vendor_id_var;
END;
/
```

## e.g. A function that returns a vendor ID

```
CREATE OR REPLACE FUNCTION get_vendor_id
(
    vendor_name_param VARCHAR2
)
RETURN NUMBER
AS
    vendor_id_var NUMBER;
BEGIN
    SELECT vendor_id
    INTO vendor_id_var
    FROM vendors
    WHERE vendor_name = vendor_name_param;

    RETURN vendor_id_var;
END;
/
```

## A SELECT statement that uses the function

```
SELECT invoice_number, invoice_total
FROM invoices
WHERE vendor_id = get_vendor_id('IBM')
```

## The response from the system

	INVOICE_NUMBER	INVOICE_TOTAL
1	QP58872	116.54
2	Q545443	1083.58

## A function that calculates balance due

```
CREATE OR REPLACE FUNCTION get_balance_due
(
    invoice_id_param NUMBER
)
RETURN NUMBER
AS
    balance_due_var NUMBER;
BEGIN
    SELECT invoice_total - payment_total - credit_total
        AS balance_due
    INTO balance_due_var
    FROM invoices
    WHERE invoice_id = invoice_id_param;

    RETURN balance_due_var;
END;
/
```

## Statement that uses expression for balance\_due

```
SELECT vendor_id, invoice_number,  
       invoice_total - payment_total - credit_total AS balance_due  
FROM invoices  
WHERE vendor_id = 37;
```

## Statement that calls function to get balance\_due

```
SELECT vendor_id, invoice_number,  
       get_balance_due(invoice_id) AS balance_due  
FROM invoices  
WHERE vendor_id = 37;
```

## The response from the system

VENDOR_ID	INVOICE_NUMBER	BALANCE_DUE
1	37 547479217	116
2	37 547480102	224
3	37 547481328	224

## Advantages

1. Code is shorter & simpler
2. Easier to maintain - Logic to calculate balance due is stored in a single, centralized location. (i.e. not each individual query).

## A statement that creates a function

```
CREATE FUNCTION get_sum_balance_due
(
    vendor_id_param NUMBER
)
RETURN NUMBER
AS
    sum_balance_due_var NUMBER;
BEGIN
    SELECT SUM(get_balance_due(invoice_id))
        AS sum_balance_due
    INTO sum_balance_due_var
    FROM invoices
    WHERE vendor_id = vendor_id_param;

    RETURN sum_balance_due_var;
END;
/
```

## A statement that calls the function

```
SELECT vendor_id, invoice_number,
       get_balance_due(invoice_id) AS balance_due,
       get_sum_balance_due(vendor_id) AS sum_balance_due
  FROM invoices
 WHERE vendor_id = 37;
```

## The response from the system

VENDOR_ID	INVOICE_NUMBER	BALANCE_DUE	SUM_BALANCE_DUE
1	37 547479217	116	564
2	37 547480102	224	564
3	37 547481328	224	564

# Appendix

## The syntax for declaring parameters

```
parameter_name_1 [IN|OUT|IN OUT] data_type
```

## A stored procedure that uses parameters

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
    invoice_number_param IN VARCHAR2,
    credit_total_param   IN NUMBER,
    update_count          OUT INTEGER
)
AS
BEGIN
    UPDATE invoices
    SET credit_total = credit_total_param
    WHERE invoice_number = invoice_number_param;

    SELECT COUNT(*)
        INTO update_count
    FROM invoices
    WHERE invoice_number = invoice_number_param;

    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        SELECT 0 INTO update_count FROM dual;
        ROLLBACK;
END;
/
```

3<sup>rd</sup> parameter outputs  
into the variable  
declared when the  
procedure is executed

## A script that calls the stored procedure

```
SET SERVEROUTPUT ON;

DECLARE
    row_count INTEGER;

BEGIN
    update_invoices_credit_total('367447', 200, row_count);
    DBMS_OUTPUT.PUT_LINE('row_count: ' || row_count);

END;
/
```

**Practice – See #5**

## The syntax of the RAISE statement

```
RAISE exception_name
```

### e.g. A procedure that raises a predefined exception

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
    invoice_number_param VARCHAR2,
    credit_total_param    NUMBER
)
AS
BEGIN
    IF credit_total_param < 0 THEN
        RAISE VALUE_ERROR;
    END IF;

    UPDATE invoices
    SET credit_total = credit_total_param
    WHERE invoice_number = invoice_number_param;

    COMMIT;
END;
```

#### Note:

- This is a predefined error that can be used to identify invalid entries

## A statement that calls the procedure

```
CALL update_invoices_credit_total('367447', -100);
```

### e.g. The response from the system if not error catching

Error report:

```
SQL Error: ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "AP.UPDATE_INVOICES_CREDIT_TOTAL", line 9
```

### e.g. Script that calls the procedure with error catching

```
SET SERVEROUTPUT ON;

BEGIN
    update_invoices_credit_total('367447', -100);
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Updating credit total to a negative is not allowed');
END;
/
```

### The response from the system

```
Updating credit total to a negative is not allowed
```

## The RAISE\_APPLICATION\_ERROR procedure

```
RAISE_APPLICATION_ERROR(error_number, error_message);
```

### e.g. A statement that raises an application error

```
RAISE_APPLICATION_ERROR(-20001, 'Credit total may not be negative.');
```

### Response if the error isn't caught

Error report:

```
SQL Error: ORA-20001: Credit total may not be negative.  
ORA-06512: at "AP.UPDATE_INVOICES_CREDIT_TOTAL", line 10
```

### e.g. A script that catches an application error

```
BEGIN  
    update_invoices_credit_total('367447', -100);  
EXCEPTION  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE(  
            'An unknown exception occurred.');
```

END;  
/

### The response from the system

An unknown exception occurred.

### Note:

- This is your own custom error message that we can add into the sproc. NOTE: you don't have the "catch" this error type in the EXCEPTION block
- While you don't have the "handle" this error type in the EXCEPTION block you can capture this with all OTHERS if you wish

## Three statements that call the stored procedure

```
CALL insert_invoice(34, 'ZXA-080', '30-AUG-08', 14092.59,  
0, 0, 3, '30-SEP-08', NULL);
```

```
CALL insert_invoice(34, 'ZXA-080', '30-AUG-08', 14092.59,  
0, 0, 3, '30-SEP-08');
```

```
CALL insert_invoice(34, 'ZXA-080', '30-AUG-08',  
14092.59);
```

## The response for a successful insert

```
CALL insert_invoice(34, succeeded.
```

## A statement that raises an error

```
CALL insert_invoice(34, 'ZXA-080', '30-AUG-08', -14092.59);
```

## The response when a validation error occurs

Error report:

```
SQL Error: ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "AP.INSERT_INVOICE", line 20
```

## A stored procedure that drops a table

```
CREATE OR REPLACE PROCEDURE drop_table
(
    table_name VARCHAR2
)
AS
BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE ' || table_name;
EXCEPTION
    WHEN OTHERS THEN
        NULL;
END;
/
```

## A statement that calls the stored procedure

```
CALL drop_table('test1');
```

## The response from the system

```
CALL drop_table('test1') succeeded.
```

## The syntax of the DROP PROCEDURE statement

```
DROP PROCEDURE procedure_name
```

e.g. A statement that creates a stored procedure

```
CREATE PROCEDURE clear_invoices_credit_total
(
    invoice_number_param  VARCHAR2
)
AS
BEGIN
    UPDATE invoices
    SET credit_total = 0
    WHERE invoice_number = invoice_number_param;

    COMMIT;
END;
/
```

A statement that drops a stored procedure

```
DROP PROCEDURE clear_invoices_credit_total
```

## The syntax of the DROP FUNCTION statement

```
DROP FUNCTION function_name
```

### A statement that drops a function

```
DROP FUNCTION get_sum_balance_due;
```

# How to retrieve data from a single table

In this chapter, you'll learn how to code SELECT statements that retrieve data from a single table. You should realize, though, that the skills covered here are the essential ones that apply to any SELECT statement, no matter how many tables it operates on and no matter how complex the retrieval. So you'll want to have a good understanding of the material in this chapter before you go on to the chapters that follow.

<b>An introduction to the SELECT statement .....</b>	<b>80</b>
The basic syntax of the SELECT statement.....	80
SELECT statement examples .....	82
<b>How to code the SELECT clause .....</b>	<b>84</b>
How to code column specifications.....	84
How to name the columns in a result set.....	86
How to code string expressions .....	88
How to code arithmetic expressions.....	90
How to use scalar functions.....	92
How to use the Dual table.....	94
How to use the DISTINCT keyword to eliminate duplicate rows .....	96
How to use the ROWNUM pseudo column to limit the number of rows .....	98
<b>How to code the WHERE clause .....</b>	<b>100</b>
How to use the comparison operators .....	100
How to use the AND, OR, and NOT logical operators.....	102
How to use the IN operator.....	104
How to use the BETWEEN operator .....	106
How to use the LIKE operator.....	108
How to use the IS NULL condition.....	110
<b>How to code the ORDER BY clause .....</b>	<b>112</b>
How to sort a result set by a column name.....	112
How to sort a result set by an alias, an expression, or a column number.....	114
<b>How to code the row limiting clause .....</b>	<b>116</b>
How to limit the number of rows.....	116
How to return a range of rows .....	116
<b>Perspective .....</b>	<b>118</b>

## An introduction to the SELECT statement

---

To help you learn to code the SELECT statement, this chapter starts by presenting its basic syntax. Next, it presents several examples that will give you an idea of what you can do with this statement. Then, the rest of this chapter will teach you the details of coding this statement.

### The basic syntax of the SELECT statement

---

Figure 3-1 presents the basic syntax of the SELECT statement. The syntax summary at the top of this figure uses some conventions that are used throughout this book. First, capitalized words are called *keywords*, and you must spell them exactly as shown, though you can use whatever capitalization you prefer. For example, the following are equivalent: “SELECT”, “Select”, “select” and “sELeCt”. Second, you must provide replacements for words in lowercase. For example, you must enter a list of columns in place of *select\_list*, and you must enter a table name in place of *table\_source*.

Beyond that, you can choose between the items in a syntax summary that are separated by pipes (|) and enclosed in braces ({{}}) or brackets ([ ]). And you can omit items enclosed in brackets. If you have a choice between two or more optional items, the default item is underlined. And if an element can be coded multiple times in a statement, it’s followed by an ellipsis (...). You’ll see examples of pipes, braces, default values, and ellipses in syntax summaries later in this chapter. For now, compare the syntax in this figure with the coding examples in figure 3-2 to see how the two are related.

The syntax summary in this figure has been simplified so that you can focus on the four main clauses of the SELECT statement: the SELECT clause, the FROM clause, the WHERE clause, and the ORDER BY clause. Most of the SELECT statements you code will contain all four clauses. However, only the SELECT and FROM clauses are required.

The SELECT clause is always the first clause in a SELECT statement. It identifies the columns you want to include in the result set. These columns are retrieved from the *base tables* named in the FROM clause. Since this chapter focuses on retrieving data from a single table, the FROM clauses in all of the statements in this chapter name a single base table. In the next chapter, though, you’ll learn how to retrieve data from two or more tables. And as you progress through this book, you’ll learn how to select data from other sources such as views and expressions.

The WHERE and ORDER BY clauses are optional. The ORDER BY clause determines how the rows in the result set are to be sorted, and the WHERE clause determines which rows in the base table are to be included in the result set. The WHERE clause specifies a search condition that’s used to *filter* the rows in the base table. This search condition can consist of one or more *Boolean expressions*, or *predicates*. A Boolean expression is an expression that evaluates to True or False. When the search condition evaluates to True, the row is included in the result set.

## The simplified syntax of the SELECT statement

```
SELECT select_list  
FROM table_source  
[WHERE search_condition]  
[ORDER BY order_by_list]
```

## The four clauses of the SELECT statement

Clause	Description
<b>SELECT</b>	Describes the columns that will be included in the result set.
<b>FROM</b>	Names the table from which the query will retrieve the data.
<b>WHERE</b>	Specifies the conditions that must be met for a row to be included in the result set. This clause is optional.
<b>ORDER BY</b>	Specifies how the rows in the result set will be sorted. This clause is optional.

### Description

- You use the basic SELECT statement shown above to retrieve the columns specified in the SELECT clause from the *base table* specified in the FROM clause and store them in a result set.
- The WHERE clause is used to *filter* the rows in the base table so that only those rows that match the search condition are included in the result set. If you omit the WHERE clause, all of the rows in the base table are included.
- The search condition of a WHERE clause consists of one or more *Boolean expressions*, or *predicates*, that result in a value of True, False, or Unknown. If the combination of all the expressions is True, the row being tested is included in the result set. Otherwise, it's not.
- If you include the ORDER BY clause, the rows in the result set are sorted in the specified sequence. Otherwise, the sequence of the rows is not guaranteed by Oracle.

### Note

- The syntax shown above does not include all of the clauses of the SELECT statement. You'll learn about the other clauses later in this book.

In this book, we won't use the terms "Boolean expression" or "predicate" because they don't clearly describe the content of the WHERE clause. Instead, we'll just use the term "search condition" to refer to an expression that evaluates to True or False.

## **SELECT statement examples**

---

Figure 3-2 presents five SELECT statement examples. All of these statements retrieve data from the Invoices table.

The first statement in this figure retrieves all of the rows and columns from the Invoices table. This statement uses an asterisk (\*) as a shorthand to indicate that all of the columns should be retrieved, and the WHERE clause is omitted so there are no conditions on the rows that are retrieved. You can see the results after this statement as they're displayed by SQL Developer. Here, both horizontal and vertical scroll bars are displayed, indicating that the result set contains more rows and columns than can be displayed on the screen at one time.

Notice that this statement doesn't include an ORDER BY clause. Without an ORDER BY clause, Oracle doesn't guarantee the sequence in which the rows are presented. They might be in the sequence you expect, or they might not. As a result, if the sequence matters to you, you should include an ORDER BY clause.

The second statement retrieves selected columns from the Invoices table. As you can see, the columns to be retrieved are listed in the SELECT clause. Like the first statement, this statement doesn't include a WHERE clause, so all the rows are retrieved. Then, the ORDER BY clause causes the rows to be sorted by the invoice\_total column in ascending sequence. Later in this chapter, you'll learn how to sort rows in descending sequence.

The third statement also lists the columns to be retrieved. In this case, though, the last column is calculated from two columns in the base table (credit\_total and payment\_total), and the resulting column is given the name total\_credits. In addition, the WHERE clause specifies that only the invoice with an invoice\_id of 17 should be retrieved.

The fourth SELECT statement includes a WHERE clause whose condition specifies a range of values. In this case, only invoices with invoice dates between May 1, 2014 and May 31, 2014 are retrieved. In addition, the rows in the result set are sorted by invoice date.

The last statement in this figure shows another variation of the WHERE clause. In this case, only those rows with an invoice\_total greater than 50,000 are retrieved. Since none of the rows in the Invoices table satisfies this condition, the result set is empty.

## A SELECT statement that retrieves all the data from the Invoices table

```
SELECT *
FROM invoices
```

INVOICE_ID	VENDOR_ID	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL	PAYMENT_TOTAL	CREDIT_TOTAL
1	1	34 QP58872	25-FEB-14	116.54	116.54	0
2	2	34 Q545443	14-MAR-14	1083.58	1083.58	0
3	3	110 P-0608	11-APR-14	20551.18	0	1200
4	4	110 P-0259	16-APR-14	26881.4	26881.4	0

(114 rows selected)

## A SELECT statement that retrieves three columns from each row, sorted in ascending sequence by invoice\_total

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
ORDER BY invoice_total
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 25022117	24-MAY-14	6
2 24863706	27-MAY-14	6
3 24780512	29-MAY-14	6
4 21-4748363	09-MAY-14	9.95

(114 rows selected)

## A SELECT statement that retrieves two columns and a calculated value for a specific invoice

```
SELECT invoice_id, invoice_total,
       (credit_total + payment_total) AS total_credits
FROM invoices
WHERE invoice_id = 17
```

INVOICE_ID	INVOICE_TOTAL	TOTAL_CREDITS
1	17	356.48

## A SELECT statement that retrieves all invoices between given dates

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_date BETWEEN '01-MAY-2014' AND '31-MAY-2014'
ORDER BY invoice_date
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 7548906-20	01-MAY-14	27
2 4-321-2596	01-MAY-14	10
3 4-327-7357	01-MAY-14	162.75
4 4-342-8069	01-MAY-14	10

(70 rows selected)

## A SELECT statement that returns an empty result set

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_total > 50000
```

INVOICE...	INVOICE...	INVOICE...

Figure 3-2 SELECT statement examples

## How to code the SELECT clause

---

Figure 3-3 presents an expanded syntax for the SELECT clause. The keywords shown in the first line allow you to restrict the rows that are returned by a query. You'll learn how to code them in a moment. But first, you'll learn various techniques for identifying which columns are to be included in a result set.

### How to code column specifications

---

Figure 3-3 summarizes the techniques you can use to code column specifications. You saw how to use some of these techniques in the previous figure. For example, you can code an asterisk in the SELECT clause to retrieve all of the columns in the base table, and you can code a list of column names separated by commas. Note that when you code an asterisk, the columns are returned in the order that they occur in the base table.

You can also code a column specification as an *expression*. For example, you can use an arithmetic expression to perform a calculation on two or more columns in the base table, and you can use a string expression to combine two or more string values. An expression can also include one or more functions. You'll learn more about each of these techniques in the topics that follow.

## The expanded syntax of the SELECT clause

```
SELECT [ALL|DISTINCT]
    column_specification [[AS] result_column]
    [, column_specification [[AS] result_column]] ...
```

## Five ways to code column specifications

Source	Option	Syntax
Base table value	All columns	*
	Column name	column_name
Calculated value	Result of a concatenation	String expression (see figure 3-5)
	Result of a calculation	Arithmetic expression (see figure 3-6)
	Result of a scalar function	Scalar function (see figure 3-7)

### Column specifications that use base table values

**The \* is used to retrieve all columns**

```
SELECT *
```

**Column names are used to retrieve specific columns**

```
SELECT vendor_name, vendor_city, vendor_state
```

### Column specifications that use calculated values

**An arithmetic expression is used to calculate balance\_due**

```
SELECT invoice_number,
       invoice_total - payment_total - credit_total AS balance_due
```

**A string expression is used to derive full\_name**

```
SELECT first_name || ' ' || last_name AS full_name
```

### Description

- Use SELECT \* only when you need to retrieve all columns from a table. Otherwise, list the names of the columns you need.
- An *expression* is a combination of column names and operators that evaluate to a single value. In the SELECT clause, you can code arithmetic expressions, string expressions, and expressions that include one or more functions.
- After each column specification, you can code an AS clause to specify the name for the column in the result set. See figure 3-4 for details.

### Note

- The other elements shown in the syntax summary above let you control the number of rows that are returned by a query. You can use the DISTINCT keyword to eliminate duplicate rows. See figure 3-9 for details.

## How to name the columns in a result set

---

By default, a column in a result set is given the same name as the column in the base table. You can specify a different name, however, if you need to. You can also name a column that contains a calculated value. When you do that, the new column name is called a *column alias*. Figure 3-4 presents two techniques for creating column aliases.

The first technique is to code the column specification followed by the AS keyword and the column alias. This is the coding technique specified by the American National Standards Institute (ANSI, pronounced ‘ann-see’), and it’s illustrated by the first example in this figure.

The second technique is to code the column specification followed by a space and the column alias. This coding technique is illustrated by the second example. Whenever possible, though, you should use the first technique since the AS keyword makes it easier to identify the alias for the column, which makes your SQL statement easier to read and maintain.

When you code an alias, you must enclose the alias in double quotes if the alias contains a space or is a keyword that’s reserved by Oracle. In this figure, the first two examples specify an alias for the invoice\_number column that uses two words with a space between them.

In addition, these two examples specify an alias for the invoice\_date column that uses a keyword that’s reserved by Oracle: the DATE keyword. If you don’t enclose this keyword in double quotes, you will get an error when you attempt to execute either of these SQL statements. When you enter a statement into SQL Developer, it boldfaces keywords that are reserved by Oracle. This makes it easy to identify Oracle keywords when you’re writing SQL statements.

When you enclose an alias in double quotes, the result set uses the capitalization specified by the alias. Otherwise, the result set capitalizes all letters in the column name. In this figure, for instance, the first two columns in the first result set use the capitalization specified by the aliases. However, since no alias is specified for the third column, all letters in the name of this column are capitalized.

When you code a column that contains a calculated value, it’s a good practice to specify an alias for the calculated column. If you don’t, Oracle will assign the entire calculation as the name, which can be unwieldy, as shown in the third example. As a result, you usually assign a name to any column that’s calculated from other columns in the base table.

## Two SELECT statements that name the columns in the result set

### A SELECT statement that uses the AS keyword

```
-- DATE is a reserved keyword.
-- As a result, it must be enclosed in quotations.
SELECT invoice_number AS "Invoice Number", invoice_date AS "Date",
       invoice_total AS total
FROM invoices
```

### A SELECT statement that omits the AS keyword

```
SELECT invoice_number "Invoice Number", invoice_date "Date",
       invoice_total total
FROM invoices
```

### The result set for both SELECT statements

Invoice Number	Date	TOTAL
1 QP58872	25-FEB-14	116.54
2 Q545443	14-MAR-14	1083.58
3 P-0608	11-APR-14	20551.18
4 P-0259	16-APR-14	26881.4
5 MAB01489	16-APR-14	936.93

## A SELECT statement that doesn't provide a name for a calculated column

```
SELECT invoice_number, invoice_date, invoice_total,
       invoice_total - payment_total - credit_total
FROM invoices
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL	INVOICE_TOTAL-PAYMENT_TOTAL-CREDIT_TOTAL
1 QP58872	25-FEB-14	116.54	0
2 Q545443	14-MAR-14	1083.58	0
3 P-0608	11-APR-14	20551.18	19351.18
4 P-0259	16-APR-14	26881.4	0
5 MAB01489	16-APR-14	936.93	0

## Description

- By default, a column in the result set is given the same name as the column in the base table. If that's not what you want, you can specify a *column alias* for the column.
- One way to name a column is to use the AS keyword as shown in the first example above. Although the AS keyword is optional, it enhances readability.
- If an alias includes spaces or an Oracle reserved keyword, you must enclose it in double quotes.
- When you enclose an alias in quotes, the result set uses the capitalization specified by the alias. Otherwise, the result set capitalizes all letters in the column name.

## How to code string expressions

---

A *string expression* consists of a combination of one or more character columns and *literal values*. To combine, or *concatenate*, the columns and values, you use the *concatenation operator* (||). This is illustrated by the examples in figure 3-5.

The first example shows how to concatenate the vendor\_city and vendor\_state columns in the Vendors table. Notice that because no alias is assigned to this column, Oracle assigns a name, which is the entire expression. Also notice that the data in the vendor\_state column appears immediately after the data in the vendor\_city column in the results. That's because of the way vendor\_city is defined in the database. Because it's defined as a variable-length column (the VARCHAR2 data type), only the actual data in the column is included in the result. In contrast, if the column had been defined with a fixed length, any spaces after the name would have been included in the result. You'll learn about data types and how they affect the data in your result set in chapter 8.

The second example shows how to format a string expression by adding spaces and punctuation. Here, the vendor\_city column is concatenated with a *string literal*, or *string constant*, that contains a comma and a space. Then, the vendor\_state column is concatenated with that result, followed by a string literal that contains a single space and the vendor\_zip\_code column.

Occasionally, you may need to include a single quotation mark or an apostrophe within a literal string. If you simply type a single quote, however, the system will misinterpret it as the end of the literal string. As a result, you must code two single quotation marks in a row. This is illustrated by the third example in this figure.

## How to concatenate string data

```
SELECT vendor_city, vendor_state, vendor_city || vendor_state
FROM vendors
```

VENDOR_CITY	VENDOR_STATE	VENDOR_CITY  VENDOR_STATE
1 Auburn Hills	MI	Auburn HillsMI
2 Fresno	CA	FresnoCA
3 Olathe	KS	OlatheKS
4 Fresno	CA	FresnoCA
5 East Brunswick	NJ	East BrunswickNJ

## How to format string data using literal values

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
```

VENDOR_NAME	ADDRESS
1 Data Reproductions Corp	Auburn Hills, MI 48326
2 Executive Office Products	Fresno, CA 93710
3 Leslie Company	Olathe, KS 66061
4 Retirement Plan Consultants	Fresno, CA 93704
5 Simon Direct Inc	East Brunswick, NJ 08816

## How to include apostrophes in literal values

```
SELECT vendor_name || "'s address: ",
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code
FROM vendors
```

VENDOR_NAME  "SADDRESS:"	VENDOR_CITY  ','  VENDOR_STATE  " "  VENDOR_ZIP_CODE
1 Data Reproductions Corp's address:	Auburn Hills, MI 48326
2 Executive Office Products's address:	Fresno, CA 93710
3 Leslie Company's address:	Olathe, KS 66061
4 Retirement Plan Consultants's address:	Fresno, CA 93704
5 Simon Direct Inc's address:	East Brunswick, NJ 08816

## Description

- A *string expression* can consist of one or more character columns, one or more *literal values*, or a combination of character columns and literal values.
- The columns specified in a string expression must contain string data (that means they're defined with the CHAR or VARCHAR2 data type).
- The literal values in a string expression also contain string data, so they can be called *string literals* or *string constants*. To create a literal value, enclose one or more characters within single quotation marks ('').
- You can use the *concatenation operator* (||) to combine columns and literals in a string expression.
- You can include a single quote within a literal value by coding two single quotation marks, as shown in the third example above.

Figure 3-5 How to code string expressions

## How to code arithmetic expressions

---

Figure 3-6 shows how to code *arithmetic expressions*. To start, it summarizes the four *arithmetic operators* you can use in this type of expression. Then, it presents two examples that show how to use these operators.

The SELECT statement in the first example includes an arithmetic expression that calculates the balance due for an invoice. This expression subtracts the payment\_total and credit\_total columns from the invoice\_total column. The resulting column is given the name balance\_due.

When Oracle evaluates an arithmetic expression, it performs the operations from left to right based on the *order of precedence*. This order says that multiplication and division are done first, followed by addition and subtraction. If that's not what you want, you can use parentheses to specify how you want an expression evaluated. Then, the expressions in the innermost sets of parentheses are evaluated first, followed by the expressions in outer sets of parentheses. Within each set of parentheses, the expression is evaluated from left to right in the order of precedence.

To illustrate how parentheses and the order of precedence affect the evaluation of an expression, consider the second example in this figure. Here, the expressions in the second and third columns both use the same operators. However, when Oracle evaluates the expression in the second column, it performs the multiplication operation before the addition operation because multiplication comes before addition in the order of precedence. In contrast, when Oracle evaluates the expression in the third column, it performs the addition operation first because it's enclosed in parentheses. As you can see in the result set, these two expressions result in different values.

Unlike some other databases, Oracle doesn't provide a modulo operator that can be used to return the remainder of a division operation. Instead, you must use the MOD function as described in the next figure.

## The arithmetic operators in order of precedence

*	Multiplication
/	Division
+	Addition
-	Subtraction

## A SELECT statement that calculates the balance due

```
SELECT invoice_total, payment_total, credit_total,
       invoice_total - payment_total - credit_total AS balance_due
FROM invoices
```

	INVOICE_TOTAL	PAYMENT_TOTAL	CREDIT_TOTAL	BALANCE_DUE
1	116.54	116.54	0	0
2	1083.58	1083.58	0	0
3	20551.18	0	1200	19351.18
4	26881.4	26881.4	0	0
5	936.93	936.93	0	0

## A SELECT statement that uses parentheses to control the sequence of operations

```
SELECT invoice_id,
       invoice_id + 7 * 3 AS order_of_precedence,
       (invoice_id + 7) * 3 AS add_first
FROM invoices
ORDER BY invoice_id
```

	INVOICE_ID	ORDER_OF_PRECEDENCE	ADD_FIRST
1	1	22	24
2	2	23	27
3	3	24	30
4	4	25	33
5	5	26	36

## Description

- Unless parentheses are used, the operations in an expression take place from left to right in the *order of precedence*. For arithmetic expressions, multiplication and division are done first, followed by addition and subtraction.
- Whenever necessary, you can use parentheses to clarify or override the sequence of operations. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets, and so on.

Figure 3-6 How to code arithmetic expressions

## How to use scalar functions

---

Figure 3-7 introduces you to *scalar functions*, which operate on a single value and return a single value. These functions work differently than the *aggregate functions* described in chapter 5 that are used to summarize data. For now, don't worry about the details of how these functions work, because you'll learn more about them in chapter 8. Instead, just focus on how they're used in column specifications.

To code a function, you begin by entering its name followed by a set of parentheses. If the function requires one or more *parameters*, you enter them within the parentheses and separate them with commas. When you enter a parameter, you need to be sure it has the correct data type.

The first example in this figure shows how to use the SUBSTR function to extract the first character of the vendor\_contact\_first\_name and vendor\_contact\_last\_name columns. The first parameter of this function specifies the column name; the second parameter specifies the starting position; and the third parameter specifies the number of characters to return. The results of the two functions are then concatenated to form initials, as shown in the result set for this statement.

The second example shows how to use the TO\_CHAR function. This function converts a column with a DATE or NUMBER data type to a character string. A common use for it is in concatenation operations, where all the data being concatenated *must* be string data. This function has two parameters. The first parameter, which specifies the column name, is required. The second parameter, which specifies a format mask for the column, is optional. In this example, a format mask of 'MM/DD/YYYY' is used to convert the payment\_date column from a DATE type to a CHAR type. This format mask specifies that the date should be displayed with a two-digit month, followed by a forward slash, followed by a two-digit day, followed by another forward slash, followed by a four-digit year.

In the second example, the payment\_date column for Invoice # P-0608 is NULL. Note that this causes the TO\_CHAR function to return an empty string for the payment date.

The third example uses the SYSDATE function to return the current date. Since this function doesn't accept any parameters, you don't need to code any parentheses after the name of the function. In fact, if you do code parentheses after the name of the function, you will get an error when you execute the statement. In this example, the second column uses the SYSDATE function to return the current date, and the third column uses the SYSDATE function to calculate the number of days between the two dates. Here, the third column also uses the ROUND function to round the decimal value that's returned by the calculation to an integer.

The fourth example shows how to use the MOD function to return the remainder of a division of two integers. Here, the second column contains an expression that returns the remainder of the division operation when the invoice-id column is divided by 10. In the result set, you can see the results for IDs 9 through 11 (the remainders are 9, 0, and 1).

### A SELECT statement that uses the SUBSTR function

```
SELECT vendor_contact_first_name, vendor_contact_last_name,
       SUBSTR(vendor_contact_first_name, 1, 1) ||
       SUBSTR(vendor_contact_last_name, 1, 1) AS initials
FROM vendors
```

VENDOR_CONTACT_FIRST_NAME	VENDOR_CONTACT_LAST_NAME	INITIALS
1 Cesar	Arodondo	CA
2 Rachael	Danielson	RD
3 Zev	Alondra	ZA
4 Salina	Edgardo	SE
5 Daniel	Bradlee	DB

### A SELECT statement that uses the TO\_CHAR function

```
SELECT 'Invoice: #' || invoice_number || ', dated ' ||
       TO_CHAR(payment_date, 'MM/DD/YYYY') ||
       ' for $' || TO_CHAR(payment_total) AS "Invoice Text"
FROM invoices
```

Invoice Text
1 Invoice: # QP58872, dated 04/11/2014 for \$116.54
2 Invoice: # Q545443, dated 05/14/2014 for \$1083.58
3 Invoice: # P-0608, dated for \$0
4 Invoice: # P-0259, dated 05/12/2014 for \$26881.4
5 Invoice: # MAB01489, dated 05/13/2014 for \$936.93

### A SELECT statement that uses the SYSDATE and ROUND functions

```
SELECT invoice_date,
       SYSDATE AS today,
       ROUND(SYSDATE - invoice_date) AS invoice_age_in_days
FROM invoices
```

INVOICE_DATE	TODAY	INVOICE_AGE_IN_DAYS
1 18-JUL-14	19-JUL-14	1
2 20-JUN-14	19-JUL-14	29
3 14-JUN-14	19-JUL-14	35

### A SELECT statement that uses the MOD function

```
SELECT invoice_id,
       MOD(invoice_id, 10) AS Remainder
FROM invoices
ORDER BY invoice_id
```

INVOICE_ID	REMAINDER
9	9
10	0
11	1

### Description

- A SQL statement can include a *function*. A function performs an operation and returns a value.
- For more information on using functions, see chapter 8.

## How to use the Dual table

---

The Dual table is automatically available to all users. This table is useful for testing expressions that use literal values, arithmetic calculations, and functions as shown in figure 3-8. In particular, the Dual table is often used in the documentation that shows how Oracle's built-in scalar functions work.

In the example in this figure, the second column in the result set shows the value of the calculation 10 minus 7, and the third column shows the date that's returned by the SYSDATE function. This shows that you can perform test calculations in more than one column of the Dual table.

## A SELECT statement that uses the Dual table

```
SELECT 'test' AS test_string,  
       10-7   AS test_calculation,  
      SYSDATE AS test_date  
FROM Dual
```

TEST_STRING	TEST_CALCULATION	TEST_DATE
1 test	3	28-MAY-14

### Description

- The Dual table is automatically created and made available to users.
- The Dual table is useful for testing expressions that use literal values, arithmetic, operators, and functions.

---

Figure 3-8 How to use the Dual table

## How to use the DISTINCT keyword to eliminate duplicate rows

---

By default, all of the rows in the base table that satisfy the search condition in the WHERE clause are included in the result set. In some cases, though, that means that the result set will contain duplicate rows, or rows whose column values are identical. If that's not what you want, you can include the DISTINCT keyword in the SELECT clause to eliminate the duplicate rows.

Figure 3-9 illustrates how this works. Here, both SELECT statements retrieve the vendor\_city and vendor\_state columns from the Vendors table. The first statement, however, doesn't include the DISTINCT keyword. Because of that, the same city and state can appear in the result set multiple times. In the results shown in this figure, for example, you can see that "Anaheim CA" occurs twice. In contrast, the second statement includes the DISTINCT keyword, so each city/state combination is included only once.

## A SELECT statement that returns all rows

```
SELECT vendor_city, vendor_state  
FROM vendors  
ORDER BY vendor_city
```

VENDOR_CITY	VENDOR_STATE
1 Anaheim	CA
2 Anaheim	CA
3 Ann Arbor	MI
4 Auburn Hills	MI
5 Boston	MA

(122 rows selected)

## A SELECT statement that eliminates duplicate rows

```
SELECT DISTINCT vendor_city, vendor_state  
FROM vendors  
ORDER BY vendor_city
```

VENDOR_CITY	VENDOR_STATE
1 Anaheim	CA
2 Ann Arbor	MI
3 Auburn Hills	MI
4 Boston	MA
5 Brea	CA

(53 rows selected)

## Description

- The DISTINCT keyword prevents duplicate (identical) rows from being included in the result set.
- The ALL keyword causes all rows matching the search condition to be included in the result set, regardless of whether rows are duplicated. Since this is the default, it's a common practice to omit the ALL keyword.
- To use the DISTINCT or ALL keyword, code it immediately after the SELECT keyword.

Figure 3-9 How to use the DISTINCT keyword to eliminate duplicate rows

## How to use the ROWNUM pseudo column to limit the number of rows

---

In addition to eliminating duplicate rows, you can limit the number of rows that are retrieved by a SELECT statement. To do that, you can use the ROWNUM pseudo column as shown in figure 3-10. A *pseudo column* works similarly to a column in a table. However, you can only use a pseudo column to select data. In other words, you can't insert, update, or delete the values stored in a pseudo column.

If you want to learn more about how pseudo columns work, you can search the Oracle Database SQL Reference manual for *pseudocolumn*. Note that the Oracle documentation doesn't include a space between the words *pseudo* and *column*.

The first example shows how to limit the number of rows in the result set to the first five rows. Here, the ROWNUM pseudo column is used in the WHERE clause to return the first five rows in the Invoices table.

The second example shows how to add an ORDER BY clause to sort the first five rows of the table so the largest invoice total is displayed first. Since the sort operation is applied after the first five rows are retrieved, this doesn't retrieve the five largest invoice totals in the Invoices table. Instead, it returns the first five rows of the table and then sorts them.

If you want to return the five largest invoice totals for the entire Invoices table, you need to sort the result set before you use the ROWNUM pseudo column to limit the number of rows included in the result set. To do that, you can use a *subquery* as shown in the third example. In the FROM clause, this example supplies a SELECT statement that sorts the Invoices table instead of supplying the name of the Invoices table. As a result, the table is sorted before the WHERE clause is applied.

For more information about working with subqueries, see chapter 6. In addition, if the ROWNUM pseudo column isn't adequate for your needs, you might want to use the ROW\_NUMBER function described in chapter 8.

## A SELECT statement that uses the ROWNUM pseudo column to limit the number of rows in the result set

```
SELECT vendor_id, invoice_total  
FROM invoices  
WHERE ROWNUM <= 5
```

VENDOR_ID	INVOICE_TOTAL
1	34
2	34
3	110
4	110
5	81

## A SELECT statement that sorts the result set after the WHERE clause

```
SELECT vendor_id, invoice_total  
FROM invoices  
WHERE ROWNUM <= 5  
ORDER BY invoice_total DESC
```

VENDOR_ID	INVOICE_TOTAL
1	110
2	110
3	34
4	81
5	34

## A SELECT statement that sorts the result set before the WHERE clause

```
SELECT vendor_id, invoice_total  
FROM (SELECT * FROM invoices  
      ORDER BY invoice_total DESC)  
WHERE ROWNUM <= 5
```

VENDOR_ID	INVOICE_TOTAL
1	110
2	110
3	110
4	72
5	110

## Description

- You can use the ROWNUM pseudo column to limit the number of rows included in the result set.
- If you want to sort the result set before you use the ROWNUM pseudo column to limit the number of rows included in the result set, you can use a *subquery* as shown in the third example. For more information about working with subqueries, see chapter 6.

Figure 3-10 How to use the ROWNUM pseudo column

## How to code the WHERE clause

---

The WHERE clause in a SELECT statement filters the rows in the base table so that only the rows you need are retrieved. In the topics that follow, you'll learn a variety of ways to code this clause.

### How to use the comparison operators

---

Figure 3-11 shows you how to use the *comparison operators* in the search condition of a WHERE clause. As you can see in the syntax summary at the top of this figure, you use a comparison operator to compare two expressions. If the result of the comparison is True, the row being tested is included in the query results.

The examples in this figure show how to use some of the comparison operators. The first WHERE clause, for example, uses the equal operator (=) to retrieve only those rows whose vendor\_state column have a value of IA. Since the state code is a string literal, it must be enclosed in single quotes.

In contrast, a numeric literal like the one in the second WHERE clause isn't enclosed in quotes. This clause uses the greater than (>) operator to retrieve only those rows that have a balance due greater than zero.

The third WHERE clause illustrates another way to retrieve all the invoices with a balance due. Like the second clause, it uses the greater than operator. Instead of comparing the balance due to a value of zero, however, it compares the invoice total to the total of the payments and credits that have been applied to the invoice.

The fourth WHERE clause illustrates how you can use comparison operators other than the equal operator with string data. In this example, the less than operator (<) is used to compare the value of the vendor\_name column to a literal string that has the letter M in the first position. That will cause the query to return all vendors with names that begin with the letters A through L.

You can also use the comparison operators with *date literals*, as illustrated by the fifth and sixth WHERE clauses. The fifth clause will retrieve rows with invoice dates on or before May 31, 2014, and the sixth clause will retrieve rows with invoice dates on or after May 1, 2014. Like string literals, date literals must be enclosed in single quotes. In addition, you can use different formats to specify dates, as shown by the two date literals shown in this figure. You'll learn more about the acceptable date formats and date comparisons in chapter 8.

The last WHERE clause shows how you can test for a *not equal* condition. To do that, you code a *less than* sign followed by a *greater than* sign. In this case, only rows with a credit total that's not equal to zero will be retrieved.

Whenever possible, you should compare expressions that have similar data types. If you attempt to compare expressions that have different data types, Oracle may implicitly convert the data types for you. Although implicit conversions are often acceptable, they will occasionally yield unexpected results. In chapter 8, you'll learn how to explicitly convert data types so your comparisons will always yield the results that you want.

## The syntax of the WHERE clause with comparison operators

```
WHERE expression_1 operator expression_2
```

### The comparison operators

=	Equal
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal

### Examples of WHERE clauses that retrieve...

#### Vendors located in Iowa

```
WHERE vendor_state = 'IA'
```

#### Invoices with a balance due (two variations)

```
WHERE invoice_total - payment_total - credit_total > 0  
WHERE invoice_total > payment_total + credit_total
```

#### Vendors with names from A to L

```
WHERE vendor_name < 'M'
```

#### Invoices on or before a specified date

```
WHERE invoice_date <= '31-MAY-14'
```

#### Invoices on or after a specified date

```
WHERE invoice_date >= '01-MAY-14'
```

#### Invoices with credits that don't equal zero

```
WHERE credit_total <> 0
```

### Description

- You can use a comparison operator to compare any two expressions that result in like data types. Although unlike data types may be converted to data types that can be compared, the comparison may produce unexpected results.
- If the result of a comparison results in a True value, the row being tested is included in the result set. If it's False or Unknown, the row isn't included.
- To use a string literal or a *date literal* in a comparison, enclose it in quotes. To use a numeric literal, enter the number without quotes.
- Character comparisons are case-sensitive. ‘CA’ and ‘Ca’, for example, are not equivalent.

## How to use the AND, OR, and NOT logical operators

---

Figure 3-12 shows how to use *logical operators* in a WHERE clause. You can use the AND and OR operators to combine two or more search conditions into a *compound condition*. And you can use the NOT operator to negate a search condition. The examples in this figure illustrate how these operators work.

The first two examples illustrate the difference between the AND and OR operators. When you use the AND operator, both conditions must be true. So in the first example, only those vendors located in Springfield, New Jersey, are retrieved from the Vendors table. When you use the OR operator, though, only one of the conditions must be true. So in the second example, all vendors located in New Jersey *and* all the vendors located in Pittsburgh are retrieved.

The third example shows a compound condition that uses two NOT operators. As you can see, this expression is difficult to understand. Because of that, you should avoid using this operator. The fourth example in this figure, for instance, shows how the search condition in the third example can be rephrased to eliminate the NOT operator. As a result, the condition in the fourth example is much easier to understand.

The last two examples in this figure show how the order of precedence for the logical operators and the use of parentheses affect the result of a search condition. By default, the NOT operator is evaluated first, followed by AND and then OR. However, you can use parentheses to override the order of precedence or to clarify a logical expression, just as you can with arithmetic expressions. In the next to last example, for instance, no parentheses are used, so the two conditions connected by the AND operator are evaluated first. In the last example, though, parentheses are used so the two conditions connected by the OR operator are evaluated first.

## The syntax of the WHERE clause with logical operators

```
WHERE [NOT] search_condition_1 {AND|OR} [NOT] search_condition_2 ...
```

### Examples of queries using logical operators

#### A search condition that uses the AND operator

```
WHERE vendor_state = 'NJ' AND vendor_city = 'Springfield'
```

#### A search condition that uses the OR operator

```
WHERE vendor_state = 'NJ' OR vendor_city = 'Pittsburgh'
```

#### A search condition that uses the NOT operator

```
WHERE NOT (invoice_total >= 5000 OR NOT invoice_date <= '01-JUL-2014')
```

#### The same condition rephrased to eliminate the NOT operator

```
WHERE invoice_total < 5000 AND invoice_date <= '01-JUL-2014'
```

### A compound condition without parentheses

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_date > '01-MAY-2014' OR invoice_total > 500
AND invoice_total - payment_total - credit_total > 0
ORDER BY invoice_number
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 0-2058	08-MAY-14	37966.19
2 0-2060	08-MAY-14	23517.58
3 0-2436	07-MAY-14	10976.06

(91 rows selected)

### The same compound condition with parentheses

```
WHERE (invoice_date > '01-MAY-2014' OR invoice_total > 500)
AND invoice_total - payment_total - credit_total > 0
ORDER BY invoice_number
```

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 0-2436	07-MAY-14	10976.06
2 109596	14-JUN-14	41.8
3 111-92R-10092	04-JUN-14	46.21

(39 rows selected)

### Description

- You can use the AND and OR *logical operators* to create *compound conditions* that consist of two or more conditions. You use the AND operator to specify that the search must satisfy both of the conditions, and you use the OR operator to specify that the search must satisfy at least one of the conditions.
- You can use the NOT operator to negate a condition, but that can make the search condition difficult to understand. If it does, you should rephrase the condition to eliminate NOT.
- When Oracle evaluates a compound condition, it evaluates the operators in this sequence: (1) NOT, (2) AND, and (3) OR. You can use parentheses to override this order of precedence or to clarify the sequence in which the operations will be evaluated.

Figure 3-12 How to use the AND, OR, and NOT logical operators

## How to use the IN operator

---

Figure 3-13 shows how to code a WHERE clause that uses the IN operator. When you use this operator, the value of the test expression is compared with the list of expressions in the IN phrase. If the test expression is equal to one of the expressions in the list, the row is included in the query results. This is illustrated by the first example in this figure, which will return all rows whose terms\_id column is equal to 1, 3, or 4.

You can also use the NOT operator with the IN phrase to test for a value that's not in a list of expressions. This is illustrated by the second example in this figure. In this case, only those vendors who are not in California, Nevada, or Oregon are retrieved.

If you look at the syntax of the IN phrase shown at the top of this figure, you'll see that you can code a *subquery* in place of a list of expressions. Subqueries are a powerful tool that you'll learn about in chapter 6. For now, though, you should know that a subquery is simply a SELECT statement within another statement. In the third example in this figure, for instance, a subquery is used to return a list of vendor\_id values for vendors who have invoices dated May 1, 2014. Then, the WHERE clause retrieves a vendor row only if the vendor is in that list. Note that for this to work, the subquery must return a single column, in this case, vendor\_id.

## The syntax of the WHERE clause with the IN operator

```
WHERE test_expression [NOT] IN ({subquery|expression_1 [, expression_2]...})
```

### Examples of the IN operator

#### The IN operator with a list of numeric literals

```
WHERE terms_id IN (1, 3, 4)
```

#### The IN operator preceded by NOT

```
WHERE vendor_state NOT IN ('CA', 'NV', 'OR')
```

#### The IN operator with a subquery

```
WHERE vendor_id IN  
  (SELECT vendor_id  
   FROM invoices  
   WHERE invoice_date = '01-MAY-2014')
```

### Description

- You can use the IN operator to test whether an expression is equal to a value in a list of expressions. Each of the expressions in the list must evaluate to the same type of data as the test expression.
- The list of expressions can be coded in any order without affecting the order of the rows in the result set.
- You can use the NOT operator to test for an expression that's not in the list of expressions.
- You can also compare the test expression to the items in a list returned by a *subquery* as illustrated by the third example above. You'll learn more about coding subqueries in chapter 6.

## How to use the BETWEEN operator

---

Figure 3-14 shows how to use the BETWEEN operator in a WHERE clause. When you use this operator, the value of a test expression is compared to the range of values specified in the BETWEEN phrase. If the value falls within this range, the row is included in the query results.

The first example in this figure shows a simple WHERE clause that uses the BETWEEN operator. It retrieves invoices with invoice dates between May 1, 2014 and May 31, 2014. Note that the range is inclusive, so invoices with invoice dates of May 1 and May 31 are included in the results.

The second example shows how to use the NOT operator to select rows that are not within a given range. In this case, vendors with zip codes that aren't between 93600 and 93799 are included in the results.

The third example shows how you can use a calculated value in the test expression. Here, the payment\_total and credit\_total columns are subtracted from the invoice\_total column to give the balance due. Then, this value is compared to the range specified in the BETWEEN phrase.

The last example shows how you can use calculated values in the BETWEEN phrase. Here, the first value selects the function SYSDATE (which represents the current date), and the second value is SYSDATE plus 30 days. So the query results will include all those invoices that are due between the current date and 30 days from the current date.

However, please note the warning about date comparisons in this figure. In particular, an invoice-date of May 31, 2014 at 2:00 PM isn't less than or equal to "31-May-2008", and it isn't between "01-May-2014" and "31-May-2014". To learn more about date comparisons, please read chapter 8.

## The syntax of the WHERE clause with the BETWEEN operator

```
WHERE test_expression [NOT] BETWEEN begin_expression AND end_expression
```

### Examples of the BETWEEN operator

#### The BETWEEN operator with literal values

```
WHERE invoice_date BETWEEN '01-MAY-2014' AND '31-MAY-2014'
```

#### The BETWEEN operator preceded by NOT

```
WHERE vendor_zip_code NOT BETWEEN 93600 AND 93799
```

#### The BETWEEN operator with a test expression coded as a calculated value

```
WHERE invoice_total - payment_total - credit_total BETWEEN 200 AND 500
```

#### The BETWEEN operator with the upper and lower limits coded as calculated values

```
WHERE invoice_due_date BETWEEN SYSDATE AND (SYSDATE + 30)
```

### Description

- You can use the BETWEEN operator to test whether an expression falls within a range of values. The lower limit must be coded as the first expression, and the upper limit must be coded as the second expression. Otherwise, the result set will be empty.
- The two expressions used in the BETWEEN operator for the range of values are inclusive. That is, the result set will include values that are equal to the upper or lower limit.
- You can use the NOT operator to test for an expression that's not within the given range.

### Warning about date comparisons

- All columns that have the DATE data type include both a date and time, and so does the value returned by the SYSDATE function. But when you code a date literal like '31-May-2014', the time defaults to 00:00:00 on a 24-hour clock, or 12:00 AM (midnight). As a result, a date comparison may not yield the results you expect. For instance, May 31, 2014 at 2:00 PM isn't between '01-May-2014' and '31-May-2014'.
- To learn more about date comparisons, please see chapter 8.

## How to use the LIKE operator

---

One final operator you can use in a search condition is the LIKE operator, shown in figure 3-15. You use this operator along with the *wildcards* shown at the top of this figure to specify the *string pattern*, or *mask*, that you want to match. The examples in this figure show how this works.

In the first example, the LIKE phrase specifies that all vendors in cities that start with the letters SAN should be included in the query results. Here, the percent sign (%) indicates that any characters can follow these three letters. So San Diego and Santa Ana are both included in the results.

The second example selects all vendors whose vendor name starts with the letters COMPU, followed by any one character, the letters ER, and any characters after that. Two vendor names that match that pattern are Compuserve and Computerworld.

The LIKE operator provides a powerful technique for finding information in a database that can't be found using any other technique.

## The syntax of the WHERE clause with the LIKE operator

```
WHERE match_expression [NOT] LIKE pattern
```

### Wildcard symbols

Symbol	Description
%	Matches any string of zero or more characters.
_	Matches any single character.

### WHERE clauses that use the LIKE operator

Example	Results that match the mask
WHERE vendor_city LIKE 'SAN%'	“San Diego” and “Santa Ana”
WHERE vendor_name LIKE 'COMPU_ER%'	“Compuserve” and “Computerworld”

### Description

- You use the LIKE operator to retrieve rows that match a *string pattern*, called a *mask*. Within the mask, you can use special characters, called *wildcard* characters, that determine which values in the column satisfy the condition.
- You can use the NOT operator before the LIKE operator. Then, only those rows with values that don't match the string pattern will be included in the result set.

## How to use the IS NULL condition

---

In chapter 1, you learned that a column can contain a *null value*. A null isn't the same as zero, a blank string that contains one or more spaces (' '), or an empty string that doesn't contain any spaces (""). Instead, a null value indicates that the data is not applicable, not available, or unknown. When you allow null values in one or more columns, you need to know how to test for them in search conditions. To do that, you can use the IS NULL condition, as shown in figure 3-16.

This figure uses a table named `null_sample` to illustrate how to search for null values. This table contains two columns. The first column, `invoice_id`, is an identification column. The second column, `invoice_total`, contains the total for the invoice, which can be a null value. As you can see in the first example, the invoice with an `invoice_id` of 3 contains a null value.

The second example in this figure shows what happens when you retrieve all the invoices with invoice totals equal to zero. Notice that the row with a null invoice total isn't included in the result set. Likewise, it isn't included in the result set that contains all the invoices with invoices totals that aren't equal to zero, as illustrated by the third example. Instead, you have to use the IS NULL condition to retrieve rows with null values, as shown in the fourth example.

You can also use the NOT operator with the IS NULL condition as illustrated in the last example in this figure. When you use this operator, all of the rows that don't contain null values are included in the query results.

## The syntax of the WHERE clause with the IS null condition

```
WHERE expression IS [NOT] NULL
```

### The contents of the Null\_Sample table

```
SELECT *
FROM null_sample
```

INVOICE_ID	INVOICE_TOTAL
1	125
2	0
3	(null)
4	2199.99
5	0

### A SELECT statement that retrieves rows with zero values

```
SELECT *
FROM null_sample
WHERE invoice_total = 0
```

INVOICE_ID	INVOICE_TOTAL
1	0
2	0

### A SELECT statement that retrieves rows with non-zero values

```
SELECT *
FROM null_sample
WHERE invoice_total <> 0
```

INVOICE_ID	INVOICE_TOTAL
1	125
2	2199.99

### A SELECT statement that retrieves rows with null values

```
SELECT *
FROM null_sample
WHERE invoice_total IS NULL
```

INVOICE_ID	INVOICE_TOTAL
1	(null)

### A SELECT statement that retrieves rows without null values

```
SELECT *
FROM null_sample
WHERE invoice_total IS NOT NULL
```

INVOICE_ID	INVOICE_TOTAL
1	125
2	0
3	2199.99
4	0

Figure 3-16 How to use the IS NULL condition

## How to code the ORDER BY clause

---

The ORDER BY clause specifies the sort order for the rows in a result set. In most cases, you can use column names from the base table to specify the sort order as you saw in some of the examples earlier in this chapter. However, you can also use other techniques to sort the rows in a result set, as described in the topics that follow.

### How to sort a result set by a column name

---

Figure 3-17 presents the expanded syntax of the ORDER BY clause. As you can see, you can sort by one or more expressions in either ascending or descending sequence. This is illustrated by the three examples in this figure.

The first two examples show how to sort the rows in a result set by a single column. In the first example, the rows in the vendors table are sorted in ascending sequence by the vendor\_name column. Since ascending is the default sequence, the ASC keyword is omitted. In the second example, the rows are sorted by the vendor\_name column in descending sequence.

To sort by more than one column, you simply list the names in the ORDER BY clause separated by commas as shown in the third example. Here, the rows in the Vendors table are first sorted by the vendor\_state column in ascending sequence. Then, within each state, the rows are sorted by the vendor\_city column in ascending sequence. Finally, within each city, the rows are sorted by the vendor\_name column in ascending sequence. This can be referred to as a *nested sort* because one sort is nested within another.

Although all of the columns in this example are sorted in ascending sequence, you should know that doesn't have to be the case. For example, we could have sorted by the vendor\_name column in descending sequence like this:

```
ORDER BY vendor_state, vendor_city, vendor_name DESC
```

Note that the DESC keyword in this example applies only to the vendor\_name column. The vendor\_state and vendor\_city columns are still sorted in ascending sequence.

If you study the first example in this figure, you can see that capital letters come before lowercase letters in an ascending sort. As a result, "ASC Signs" comes before "Abbey Office Furnishings" in the result set. For some business applications, this is acceptable. But if it isn't, you can use the LOWER function to convert the column to lowercase letters in the ORDER BY clause like this:

```
ORDER BY LOWER(vendor_name)
```

Then, the rows will be sorted in the correct alphabetical sequence. In chapter 8, you can learn more about this function.

## The expanded syntax of the ORDER BY clause

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC]] ...
```

### An ORDER BY clause that sorts by one column in ascending sequence

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_name
```

VENDOR_NAME	ADDRESS
1 ASC Signs	Fresno, CA 93703
2 AT&T	Phoenix, AZ 85062
3 Abbey Office Furnishings	Fresno, CA 93722

### An ORDER BY clause that sorts by one column in descending sequence

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_name DESC
```

VENDOR_NAME	ADDRESS
1 Zylka Design	Fresno, CA 93711
2 Zip Print & Copy Center	Fresno, CA 93777
3 Zee Medical Service Co	Washington, IA 52353

### An ORDER BY clause that sorts by three columns

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_state, vendor_city, vendor_name
```

VENDOR_NAME	ADDRESS
1 AT&T	Phoenix, AZ 85062
2 Computer Library	Phoenix, AZ 85023
3 Wells Fargo Bank	Phoenix, AZ 85038
4 Aztek Label	Anaheim, CA 92807
5 Blue Shield of California	Anaheim, CA 92850
6 Diversified Printing & Pub	Brea, CA 92621
7 ASC Signs	Fresno, CA 93703

## Description

- The ORDER BY clause specifies how you want the rows in the result set sorted. You can sort by one or more columns, and you can sort each column in either ascending (ASC) or descending (DESC) sequence. ASC is the default.
- By default, in an ascending sort, special characters appear first in the sort sequence, followed by numbers, then by capital letters, then by lowercase letters, and then by null values. In a descending sort, this sequence is reversed.
- With one exception, you can sort by any column in the base table, regardless of whether it's included in the SELECT clause. The exception is if the query includes the DISTINCT keyword. Then, you can only sort by columns included in the SELECT clause.

Figure 3-17 How to sort a result set by a column name

## How to sort a result set by an alias, an expression, or a column number

---

Figure 3-18 presents three more techniques that you can use to specify sort columns. First, you can use a column alias that's defined in the SELECT clause. The first SELECT statement in this figure, for example, sorts by a column named address, which is an alias for the concatenation of the vendor\_city, vendor\_state, and vendor\_zip\_code columns. Within the address column, the result set is sorted by the vendor\_name column.

You can also use an arithmetic or string expression in the ORDER BY clause, as illustrated by the second example in this figure. Here, the expression consists of the vendor\_contact\_last\_name column concatenated with the vendor\_contact\_first\_name column. Here, neither of these columns is included in the SELECT clause.

The last example in this figure shows how you can use column numbers to specify a sort order. To use this technique, you code the number that corresponds to the column of the result set, where 1 is the first column, 2 is the second column, and so on. In this example, the ORDER BY clause sorts the result set by the second column, which contains the concatenated address, then by the first column, which contains the vendor name. The result set returned by this statement is the same as the result set returned by the first statement.

Notice, however, that the statement that uses column numbers is more difficult to read because you have to look at the SELECT clause to see what columns the numbers refer to. In addition, if you add or remove columns from the SELECT clause, you may also have to change the ORDER BY clause to reflect the new column positions. As a result, we don't recommend this coding technique.

### An ORDER BY clause that uses an alias

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY address, vendor_name
```

VENDOR_NAME	ADDRESS
1 Aztek Label	Anaheim, CA 92807
2 Blue Shield of California	Anaheim, CA 92850
3 Malloy Lithographing Inc	Ann Arbor, MI 48106
4 Data Reproductions Corp	Auburn Hills, MI 48326

### An ORDER BY clause that uses an expression

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_contact_last_name || vendor_contact_first_name
```

VENDOR_NAME	ADDRESS
1 Dristas Groom & McCormick	Fresno, CA 93720
2 Internal Revenue Service	Fresno, CA 93888
3 US Postal Service	Madison, WI 53707
4 Yale Industrial Trucks-Fresno	Fresno, CA 93706

### An ORDER BY clause that uses column positions

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY 2, 1
```

VENDOR_NAME	ADDRESS
1 Aztek Label	Anaheim, CA 92807
2 Blue Shield of California	Anaheim, CA 92850
3 Malloy Lithographing Inc	Ann Arbor, MI 48106
4 Data Reproductions Corp	Auburn Hills, MI 48326

### Description

- The ORDER BY clause can include a column alias that's specified in the SELECT clause.
- The ORDER BY clause can include any valid expression. The expression can refer to any column in the base table, even if it isn't included in the result set.
- The ORDER BY clause can use numbers to specify the columns to use for sorting. In that case, 1 represents the first column in the result set, 2 represents the second column, and so on.

## How to code the row limiting clause

---

With Oracle 12c and later, you can use the OFFSET and FETCH clauses to limit the number of rows that are returned in the result set and to set the starting point for the rows that are returned. For most queries, you want to retrieve the entire result set. As a result, you typically don't need this clause. However, there may be times when you want to use this clause to retrieve a subset of a larger result set as shown in figure 3-19.

### How to limit the number of rows

---

In the simplest form of the row limiting clause, you omit the optional OFFSET clause and code the FETCH clause. Then, the number of rows in the result set is, at most, the number you specify. However, if the result set is smaller than the number you specify, the FETCH clause has no effect.

The SELECT statement in the first example includes the FETCH FIRST 5 ROWS ONLY clause. As a result, the entire result set is five rows. Without the FETCH clause, this statement would return 114 rows. Because the result set is sorted by invoice total in descending order, this result set represents the five largest invoices.

If necessary, you can also use the PERCENT keyword to limit the number of rows by percentage. In the first example, for instance, you could code FETCH FIRST 30 PERCENT ROWS ONLY to retrieve the first 30% of rows. Because the result set is sorted by invoice total in descending sequence, this result set would represent the top 30% of largest invoices.

Similarly, you can use the WITH TIES keywords to include rows where the sort value is the same as the sort value for the last row in the result set. In the first example, for instance, the result set is sorted by the invoice\_total column. As a result, you could code FETCH NEXT 5 ROWS ONLY WITH TIES to add any other rows that have an invoice total of 20551.18 to the result set. This would cause the result set to return more than 5 rows.

### How to return a range of rows

---

If you code the optional OFFSET clause, it represents an *offset*, or starting point for the result set. This offset starts from a value of 0, which refers to the first row in the result set. In the second example, then, the offset is 2 so the result set starts with the third invoice. Then, the FETCH clause limits the result set to 3 rows.

The third example has an offset of 100, so the result set starts with row 101. Then, the FETCH clause limits the result set to 1000 rows. Since the table contains only 114 rows, though, the result set contains just the last 14 rows in the table.

If you want to retrieve all rows from the offset to the end of the result set, you can omit the FETCH clause. In the third example, for instance, you could code OFFSET 100 ROWS to retrieve the same result set.

## The syntax of the row limiting clause

```
[ OFFSET offset { ROW | ROWS } ]
[ FETCH { FIRST | NEXT } [ { rowcount | percent PERCENT } ]
{ ROW | ROWS } { ONLY | WITH TIES } ]
```

## A FETCH clause that retrieves the first five rows

```
SELECT vendor_id, invoice_total
FROM invoices
ORDER BY invoice_total DESC
FETCH FIRST 5 ROWS ONLY
```

	VENDOR_ID	INVOICE_TOTAL
1	110	37966.19
2	110	26881.4
3	110	23517.58
4	72	21842
5	110	20551.18

## An OFFSET clause that starts with the third row and fetches three rows

```
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
OFFSET 2 ROWS FETCH NEXT 3 ROWS ONLY
```

	INVOICE_ID	VENDOR_ID	INVOICE_TOTAL
1	3	110	20551.18
2	4	110	26881.4
3	5	81	936.93

## An OFFSET clause that starts with the 101<sup>st</sup> row

```
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
OFFSET 100 ROWS FETCH NEXT 1000 ROWS ONLY
```

	INVOICE_ID	VENDOR_ID	INVOICE_TOTAL
1	101	103	1367.5
2	102	48	856.92
3	103	95	19.67
4	104	114	290

(14 rows)

## Description

- Typically, you use an ORDER BY clause to sort the result set before you apply the row limiting clause.
- You can use the FETCH clause to limit the number of rows returned by the SELECT statement.
- You can use the OFFSET clause to specify the first row to return, where the first row is 0, the second row is 1, and so on.
- You can use the PERCENT keyword to fetch the specified percentage of rows.
- You can use the WITH TIES keyword to include additional rows if the sort value is the same as the sort value for the last row in the result set.

Figure 3-19 How to code the row limiting clause

## Perspective

---

The goal of this chapter has been to teach you the basic skills for coding SELECT statements. You'll use these skills in almost every SELECT statement you code. As you'll see in the chapters that follow, however, there's a lot more to coding SELECT statements than what's presented here. In the next three chapters, then, you'll learn additional skills for coding SELECT statements.

## Terms

---

base table	order of precedence
keyword	function
filter	parameter
Boolean expression	date literal
predicate	comparison operator
expression	logical operator
column alias	compound condition
string expression	pseudo column
concatenate	subquery
concatenation operator	string pattern
literal value	mask
string literal	wildcard
string constant	null value
arithmetic expression	nested sort
arithmetic operator	

## Exercises

### Run some of the examples in this chapter

In these exercises, you'll use Oracle SQL Developer to run some of the scripts for the examples in this chapter. This assumes that you already know how to use SQL Developer, as described in chapter 2.

1. Start Oracle SQL Developer.
2. Open the script for fig3-02a that you should find in this directory:  
`c:\murach\oracle_sql\scripts\ch03.`  
Then, press the F9 key or click on the Execute Statement button to run the script. This shows you the data that's in the *Invoices* table that you'll be working with in this chapter.
3. Open and run the script for fig3-02b.
4. Open and run the scripts for any of the other examples in this chapter that you're interested in reviewing.

## Enter and run your own SELECT statements

In these exercises, you'll enter and run your own SELECT statements. To do that, you can open the script for an example that is similar to the statement you need to write, copy the statement into a new Worksheet window, and then modify the statement. That can save you both time and syntax errors.

5. Write a SELECT statement that returns three columns from the Vendors table: vendor\_name, vendor\_contact\_last\_name, and vendor\_contact\_first\_name. Then, run this statement.

Next, add code to this statement so it sorts the result set by last name and then first name. Then, run this statement again. This is a good way to build and test a statement, one clause at a time.

6. Write a SELECT statement that returns one column from the Vendors table named full\_name. Create this column from the vendor\_contact\_first\_name and vendor\_contact\_last\_name columns, and format it like this: last name, comma, space, first name (for example, “Doe, John”). Next, sort the result set by last name and then first name. Then, filter the result set for contacts whose last name begins with the letter A, B, C, or E.
7. Write a SELECT statement that returns four columns from the Invoices table named Due Date, Invoice Total, 10%, and Plus 10%. These columns should contain this data:

Due Date	The invoice_due_date column
Invoice Total	The invoice_total column
10%	10% of the value of invoice_total
Plus 10%	The value of invoice_total plus 10%

(For example, if invoice\_total is 100, 10% is 10, and Plus 10% is 110.) Next, filter the result set so it returns only those rows with an invoice total that's greater than or equal to 500 and less than or equal to 1000. Then, sort the result set in descending sequence by invoice\_due\_date.

8. Write and run a SELECT statement that returns four columns from the Invoices table named Number, Total, Credits, and Balance Due. These columns should include this data:

Number	The invoice_number column
Total	The invoice_total column
Credits	Sum of the payment_total and credit_total columns
Balance Due	Invoice_total minus the sum of payment_total and credit_total

Next, filter for invoices with a balance due that's greater than or equal to \$500. Then, sort the result set by balance due in descending sequence. Last, use the ROWNUM pseudo column so the result set contains only the rows with the 10 largest balance dues.

### Work with nulls and use the Dual table

9. Write a SELECT statement that returns the balance due and the payment date from the Invoices table, but only when the payment\_date column contains a null value. Then, modify the WHERE clause so it returns any invalid rows (rows in which the balance due is zero and the payment date is null).
10. Use the Dual table to create a row with these columns:

Starting Principal	Starting principle which should be equal to \$51,000
New Principal	Starting principal plus a 10% increase
Interest	6.5% of the new principal
Principal + Interest	The new principal plus the interest (add the expression you used for the new principal calculation to the expression you used for the interest calculation)

Now, add a column named “System Date” that uses the TO\_CHAR function to show the results of the SYSDATE function when it's displayed with this format:

```
'dd-mon-yyyy hh24:mi:ss'
```

This format will display the day, month, year, hours, minutes, and seconds of the system date, and this will show that the system date also includes a time. (You should be able to figure out how to use the TO\_CHAR and SYSDATE functions by studying figure 3-7.)

# 4

## How to retrieve data from two or more tables

In the last chapter, you learned how to create result sets that contain data from a single table. Now, this chapter will show you how to create result sets that contain data from two or more tables. To do that, you can use either a join or a union.

<b>How to work with inner joins.....</b>	<b>122</b>
How to code an inner join.....	122
When and how to use table aliases.....	124
How to work with tables from different schemas.....	126
How to use compound join conditions .....	128
How to use a self-join .....	130
Inner joins that join more than two tables.....	132
How to use the implicit inner join syntax.....	134
<b>How to work with outer joins.....</b>	<b>136</b>
How to code an outer join.....	136
Outer join examples .....	138
Outer joins that join more than two tables .....	140
How to use the implicit outer join syntax.....	142
<b>Other skills for working with joins.....</b>	<b>144</b>
How to combine inner and outer joins .....	144
How to join tables with the USING keyword.....	146
How to join tables with the NATURAL keyword.....	148
How to use cross joins .....	150
<b>How to work with unions .....</b>	<b>152</b>
The syntax of a union .....	152
Unions that combine data from different tables .....	152
Unions that combine data from the same table .....	154
How to use the MINUS and INTERSECT operators .....	156
<b>Perspective .....</b>	<b>158</b>

## How to work with inner joins

---

A *join* lets you combine columns from two or more tables into a single result set. In the topics that follow, you'll learn how to use the most common type of join, an *inner join*. You'll learn how to use other types of joins later in this chapter.

### How to code an inner join

---

Figure 4-1 presents the *explicit syntax* for coding an inner join. This syntax was introduced in version 9i of Oracle. As you'll see later in this chapter, Oracle also provides an implicit syntax that you can use to code inner joins. However, the syntax shown in this figure is the one you'll use most often.

To join data from two tables, you code the names of the two tables in the FROM clause along with the JOIN keyword and an ON phrase that specifies the *join condition*. The join condition indicates how the two tables should be compared. In most cases, they're compared based on the relationship between the primary key of the first table and a foreign key of the second table.

The SELECT statement in this figure, for example, joins data from the Vendors and Invoices tables based on the vendor\_id column in each table. Notice that because the equals operator is used in this condition, the value of the vendor\_id column in a row in the Vendors table must match the vendor\_id in a row in the Invoices table for that row to be included in the result set. In other words, only vendors with one or more invoices will be included. Although you'll code most inner joins using the equals operator, you should know that you can compare two tables based on other conditions too.

In this example, the Vendors table is joined with the Invoices table using a column that has the same name in both tables: vendor\_id. Because of that, the columns must be qualified to indicate which table they come from. As you can see, you code a *qualified column name* by entering the table name and a period in front of the column name. Although this example uses qualified column names only in the join condition, you must qualify a column name anywhere it appears in the statement if the same name occurs in both tables. If you don't, Oracle will return an error indicating that the column name is ambiguous. Of course, you can also qualify column names that aren't ambiguous. However, we recommend you do that only if it clarifies your code.

## The explicit syntax for an inner join

```
SELECT select_list
FROM table_1
    [INNER] JOIN table_2
        ON join_condition_1
    [[INNER]] JOIN table_3
        ON join_condition_2]...
```

## A SELECT statement that joins the Vendors and Invoices tables

```
SELECT invoice_number, vendor_name
FROM vendors INNER JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
ORDER BY invoice_number
```

### The result set

INVOICE_NUMBER	VENDOR_NAME
1 0-2058	Malloy Lithographing Inc
2 0-2060	Malloy Lithographing Inc
3 0-2436	Malloy Lithographing Inc
4 1-200-5164	Federal Express Corporation

(114 rows selected)

## Description

- A *join* is used to combine columns from two or more tables into a result set based on the *join conditions* you specify. For an *inner join*, only those rows that satisfy the join condition are included in the result set.
- A join condition names a column in each of the two tables involved in the join and indicates how the two columns should be compared. In most cases, you use the equals operator to retrieve rows with matching columns. However, you can also use any of the other comparison operators in a join condition.
- In most cases, you'll join two tables based on the relationship between the primary key in one table and a foreign key in the other table. However, you can also join tables based on relationships not defined in the database. These are called *ad hoc relationships*.
- If the two columns in a join condition have the same name, you have to qualify them with the table name so Oracle can distinguish between them. To code a *qualified column name*, type the table name, followed by a period, followed by the column name.

## Notes

- The INNER keyword is optional and is seldom used.
- This syntax for coding an inner join can be referred to as the *explicit syntax*. It is also called the *SQL-92 syntax* because it was introduced by the SQL-92 standards.
- You can also code an inner join using the *implicit syntax*. See figure 4-7 for more information.

## When and how to use table aliases

---

When you name a table to be joined in the FROM clause, you can refer to the table by an *alias*. A *table alias* is similar to a column alias, except that you do not use the word AS, like you do when you assign a column alias. After you assign a table alias, you must use the alias in place of the original table name throughout the query. This is illustrated in figure 4-2.

The first SELECT statement in this figure joins data from the Vendors and Invoices tables. Here, both tables have been assigned aliases that consist of a single letter. Table aliases are used to reduce typing and to make a query more understandable, particularly when table names are lengthy.

The alias used in the second SELECT statement in this figure, for example, simplifies the name of the Invoice\_Line\_Items table to just Line\_Items. That way, the shorter name can be used to refer to the invoice\_id column of the table in the join condition. Although this doesn't improve the query in this example much, it can have a dramatic effect on a query that refers to the Invoice\_Line\_Items table several times.

## The syntax for an inner join that uses table aliases

```
SELECT select_list
FROM table_1 n1
    [INNER] JOIN table_2 n2
        ON n1.column_name operator n2.column_name
    [[INNER]] JOIN table_3 n3
        ON n2.column_name operator n3.column_name]...
```

## An inner join with aliases for all tables

```
SELECT invoice_number, vendor_name, invoice_due_date,
       (invoice_total - payment_total - credit_total) AS balance_due
  FROM vendors v JOIN invoices i
    ON v.vendor_id = i.vendor_id
 WHERE (invoice_total - payment_total - credit_total) > 0
 ORDER BY invoice_due_date DESC
```

### The result set

INVOICE_NUMBER	VENDOR_NAME	INVOICE_DUE_DATE	BALANCE_DUE
1 40318	Data Reproductions Corp	20-JUL-14	21842
2 39104	Data Reproductions Corp	20-JUL-14	85.31
3 0-2436	Malloy Lithographing Inc	17-JUL-14	10976.06

(40 rows selected)

## An inner join with an alias for only one table

```
SELECT invoice_number, line_item_amt, line_item_description
  FROM invoices JOIN invoice_line_items line_items
    ON invoices.invoice_id = line_items.invoice_id
 WHERE account_number = 540
 ORDER BY invoice_date
```

### The result set

INVOICE_NUMBER	LINE_ITEM_AMT	LINE_ITEM_DESCRIPTION
1 97/553B	313.55	Card revision
2 97/553	904.14	DB2 Card decks
3 97/522	765.13	SCMD Flyer

(8 rows selected)

## Description

- A *table alias* is an alternative table name assigned in the FROM clause. You can use an alias when a long table name would make qualified column names long or confusing.
- If you assign an alias to a table, you must use that alias to refer to the table throughout your query. You can't use the original table name.
- You can use an alias for any table in a join without using an alias for any other table.
- Use table aliases whenever they simplify or clarify the query. Avoid using them when they make a query more confusing or difficult to read.

## How to work with tables from different schemas

---

If you use the procedure in figure A-5 of appendix A to create the tables for this book, all of the tables will be organized into three *schemas*. First, all tables pertaining to accounts payable such as the Vendors and Invoices tables are stored in the schema named AP. Then, all tables pertaining to order management are stored in a schema named OM. Finally, all tables that are used by the smaller examples presented in this book are stored in a schema named EX.

In addition, the procedure in figure A-5 creates one *user* that is the *owner* for each schema. In particular, a user named AP is the owner of the AP schema, a user named OM is the owner of the OM schema, and a user named EX is the owner of the EX schema.

When you connect to Oracle as a user that's the owner of a schema, you do not need to qualify any table name with the schema name. This is the case most of the time. If, for example, you connect as the AP user, you can access any of the tables in the AP schema. However, you may occasionally need to join to a table that's in another schema. To do that, you must qualify the table name in the other schema by prefixing the table name with the schema name. This is shown in figure 4-3.

Before you can work with tables in another schema, you must log on as a user that has appropriate permissions to work with the tables in that schema. The first example in this figure shows a statement that can be used to grant permissions to the AP user that allows that user to select data from the Customers table in the OM schema. To run this statement, you must log on as the OM user, which has the appropriate permissions to grant this permission to the AP user.

For now, that's all you need to know about granting permissions. However, you'll learn more about users and permissions in chapter 12. Usually, though, you won't need to worry about this since your database administrator (DBA) will be responsible for granting the appropriate permissions to each user.

Once you have granted the appropriate permissions to the AP user, you can log on as the AP user and run the SELECT statement shown in the second example. This statement joins data from two tables (Vendors and Customers) in two different schemas (AP and OM). To do that, this statement qualifies the Customers table with the name of the OM schema. Since this example assumes that you're logged on as the AP user, it isn't necessary to qualify the Vendors table with the name of the AP schema. However, if you thought it improved the readability of the statement, you could qualify the Vendors table too, and the result of the query would be the same.

**The syntax of a table name that's qualified with a schema name**

```
schema_name.table_name
```

**A SQL statement that grants the SELECT permission  
on the Customers table in the OM schema to the AP schema**

```
GRANT SELECT ON customers TO ap
```

**A SQL statement that joins to a table from another schema**

```
SELECT vendor_name, customer_last_name, customer_first_name,  
      vendor_state AS state, vendor_city AS city  
FROM vendors v  
JOIN om.customers c  
ON v.vendor_zip_code = c.customer_zip  
ORDER BY state, city
```

**The result set**

VENDOR_NAME	CUSTOMER_LAST_NAME	CUSTOMER_FIRST_NAME	STATE	CITY
1 Wells Fargo Bank	Marissa	Kyle	AZ	Phoenix
2 Aztek Label	Irvin	Ania	CA	Anaheim
3 Lou Gentile's Flower Basket	Damien	Deborah	CA	Fresno
4 Shields Design	Damien	Deborah	CA	Fresno
5 Costco	Neftaly	Thalia	CA	Fresno
6 Costco	Holbrooke	Rashad	CA	Fresno
7 Gary McKeighan Insurance	Holbrooke	Rashad	CA	Fresno
8 Zylka Design	Neftaly	Thalia	CA	Fresno
9 Zylka Design	Holbrooke	Rashad	CA	Fresno

(37 rows selected)

**Description**

- Before you can access a table in another schema, you must log on as a user that has appropriate access permissions.
- To grant access permissions to a user, you can use a GRANT statement. For more information about using the GRANT statement, see chapter 12.
- To join to a table in another schema, you must prefix the table name with the schema name.

---

Figure 4-3 How to work with tables from different schemas

## How to use compound join conditions

---

Although a join condition typically consists of a single comparison, you can include two or more comparisons in a join condition using the AND and OR operators. Figure 4-4 illustrates how this works.

In the first SELECT statement in this figure, you can see that the Invoices and Invoice\_Line\_Items tables are joined based on two comparisons. First, the primary key of the Invoices table, invoice\_id, is compared with the foreign key of the Invoice\_Line\_Items table, also named invoice\_id. As in previous examples, this comparison uses an equals condition. Then, the invoice\_total column in the Invoices table is tested for a value greater than the value of the invoice\_line\_item\_amt column in the Invoice\_Line\_Items table. That means that only those invoices that have two or more line items will be included in the result set. You can see part of this result set in this figure.

Another way to code these conditions is to code the primary join condition in the FROM clause and the other condition in the WHERE clause. This is illustrated by the second SELECT statement in this figure.

When you code separate compound join conditions like this, you may wonder which technique is most efficient. However, Oracle typically does a good job of optimizing queries so they run efficiently regardless of how you write the query. As a result, you can usually focus on writing a query so the code is easy to read and maintain. Then, if you encounter performance problems with your queries, you can work on optimizing the query later.

## An inner join with two conditions

```
SELECT invoice_number, invoice_date,  
       invoice_total, line_item_amt  
  FROM invoices i JOIN invoice_line_items li  
    ON (i.invoice_id = li.invoice_id) AND  
        (i.invoice_total > li.line_item_amt)  
 ORDER BY invoice_number
```

## The same join with the second condition coded in a WHERE clause

```
SELECT invoice_number, invoice_date,  
       invoice_total, line_item_amt  
  FROM invoices i JOIN invoice_line_items li  
    ON i.invoice_id = li.invoice_id  
 WHERE i.invoice_total > li.line_item_amt  
 ORDER BY invoice_number
```

### The result set

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL	LINE_ITEM_AMT
1 97/522	30-APR-14	1962.13	765.13
2 97/522	30-APR-14	1962.13	1197
3 I77271-001	05-JUN-14	662	75.6
4 I77271-001	05-JUN-14	662	58.4

(6 rows selected)

## Description

- A join condition can include two or more conditions connected by AND or OR operators.
- In most cases, your code will be easier to read if you code the join condition in the ON expression and search conditions in the WHERE clause.

## How to use a self-join

---

A *self-join* is a join where a table is joined with itself. Although self-joins are rare, there are some unique queries that are best solved using self-joins.

Figure 4-5 presents an example of a self-join that uses the Vendors table. Notice that since the same table is used twice, aliases are used to distinguish one occurrence of the table from the other. In addition, each column name used in the query is qualified by the alias since the columns occur in both tables.

The join condition in this example uses three comparisons. The first two match the vendor\_city and vendor\_state columns in the two tables. As a result, the query will return rows for vendors that reside in the same city and state as another vendor. Because a vendor resides in the same city and state as itself, however, a third comparison is included to exclude rows that match a vendor with itself. To do that, this condition uses the not equals operator to compare the vendor\_id columns in the two tables.

Notice that the DISTINCT keyword is also included in this SELECT statement. That way, a vendor appears only once in the result set. Otherwise, it would appear once for each row with a matching city and state.

This example also shows how you can use columns other than key columns in a join condition. Keep in mind, however, that this is an unusual situation and you're not likely to code joins like this often.

## A self-join that returns vendors from cities in common with other vendors

```
SELECT DISTINCT v1.vendor_name, v1.vendor_city,  
    v1.vendor_state  
FROM vendors v1 JOIN vendors v2  
    ON (v1.vendor_city = v2.vendor_city) AND  
        (v1.vendor_state = v2.vendor_state) AND  
        (v1.vendor_id <> v2.vendor_id)  
ORDER BY v1.vendor_state, v1.vendor_city
```

### The result set

VENDOR_NAME	VENDOR_CITY	VENDOR_STATE
1 AT&T	Phoenix	AZ
2 Computer Library	Phoenix	AZ
3 Wells Fargo Bank	Phoenix	AZ
4 Aztek Label	Anaheim	CA
5 Blue Shield of California	Anaheim	CA
6 ASC Signs	Fresno	CA
7 Abbey Office Furnishings	Fresno	CA
8 BFI Industries	Fresno	CA

(84 rows selected)

### Description

- A *self-join* is a join that joins a table with itself.
- When you code a self-join, you must use aliases for the tables, and you must qualify each column name with the alias.

Figure 4-5 How to use a self-join

## Inner joins that join more than two tables

---

So far in this chapter, you've seen how to join data from two tables. Depending on the requirement, however, you may have to join many more tables. For example, you sometimes need to join 10 or more tables.

The SELECT statement in figure 4-6 joins data from four tables: Vendors, Invoices, Invoice\_Line\_Items, and General\_Ledger\_Accounts. Each of the joins is based on the relationship between the primary key of one table and a foreign key of the other table. For example, the account\_number column is the primary key of the General\_Ledger\_Accounts table and a foreign key of the Invoice\_Line\_Items table.

This SELECT statement also begins to show how table aliases make a statement easier to code and read. Here, the one-letter and two-letter aliases that are used by the tables allow you to code the ON clause more concisely.

## A SELECT statement that joins four tables

```
SELECT vendor_name, invoice_number, invoice_date,  
      line_item_amt, account_description  
FROM vendors v  
    JOIN invoices i  
      ON v.vendor_id = i.vendor_id  
    JOIN invoice_line_items li  
      ON i.invoice_id = li.invoice_id  
    JOIN general_ledger_accounts gl  
      ON li.account_number = gl.account_number  
 WHERE (invoice_total - payment_total - credit_total) > 0  
 ORDER BY vendor_name, line_item_amt DESC
```

### The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_DATE	LINE_ITEM_AMT	ACCOUNT_DESCRIPTION
1 Abbey Office Furnishings	203339-13	02-MAY-14	17.5	Office Supplies
2 Blue Cross	547481328	20-MAY-14	224	Group Insurance
3 Blue Cross	547480102	19-MAY-14	224	Group Insurance
4 Blue Cross	547479217	17-MAY-14	116	Group Insurance
5 Cardinal Business Media, Inc.	134116	01-JUN-14	90.36	Card Deck Advertising
6 Coffee Break Service	109596	14-JUN-14	41.8	Meals
7 Compuserve	21-4748363	09-MAY-14	9.95	Books, Dues, and Subscriptions
8 Computerworld	367447	31-MAY-14	2433	Card Deck Advertising

(44 rows selected)

### Description

- You can think of a multi-table join as a series of two-table joins proceeding from left to right.

Figure 4-6 Inner joins that join more than two tables

## How to use the implicit inner join syntax

---

Earlier in this chapter, we mentioned that Oracle provides an *implicit syntax* for joining tables. This syntax was used prior to Oracle 9i. Although we recommend that you use the explicit syntax when you're developing SQL statements for newer versions of Oracle, you should be familiar with the implicit syntax in case you ever need to maintain existing SQL statements that use it.

Figure 4-7 presents the implicit syntax for an inner join, along with two statements that use it. As you can see, the tables to be joined are simply listed in the FROM clause. Then, the join conditions are included in the WHERE clause.

The first SELECT statement joins data from the Vendors and Invoices tables. Like the SELECT statement you saw back in figure 4-1, these tables are joined based on an equals comparison between the vendor\_id columns in the two tables. In this case, though, the comparison is coded as the search condition of the WHERE clause. If you compare the result set shown in this figure with the one in figure 4-1, you'll see that they're identical.

The second SELECT statement uses the implicit syntax to join data from four tables. This is the same join you saw in figure 4-6. Notice in this example that the three join conditions are combined in the WHERE clause using the AND operator. In addition, an AND operator is used to combine the join conditions with the search condition.

Because the explicit syntax for joins lets you separate join conditions from search conditions, statements that use the explicit syntax are typically easier to read than those that use the implicit syntax. In addition, the explicit syntax helps you avoid a common coding mistake with the implicit syntax: omitting the join condition. As you'll learn later in this chapter, an implicit join without a join condition results in a cross join, which can return a large number of rows. For these reasons, we recommend that you use the explicit syntax in all your new SQL code.

## The implicit syntax for an inner join

```
SELECT select_list
FROM table_1, table_2 [, table_3]...
WHERE table_1.column_name operator table_2.column_name
    [AND table_2.column_name operator table_3.column_name]...
```

## A SELECT statement that joins the Vendors and Invoices tables

```
SELECT invoice_number, vendor_name
FROM vendors v, invoices i
WHERE v.vendor_id = i.vendor_id
ORDER BY invoice_number
```

### The result set

INVOICE_NUMBER	VENDOR_NAME
1 0-2058	Malloy Lithographing Inc
2 0-2060	Malloy Lithographing Inc
3 0-2436	Malloy Lithographing Inc
4 1-200-5164	Federal Express Corporation
5 1-202-2978	Federal Express Corporation

(114 rows selected)

## A statement that joins four tables

```
SELECT vendor_name, invoice_number, invoice_date,
       line_item_amt, account_description
  FROM vendors v, invoices i, invoice_line_items li,
       general_ledger_accounts gl
 WHERE v.vendor_id = i.vendor_id
   AND i.invoice_id = li.invoice_id
   AND li.account_number = gl.account_number
   AND (invoice_total - payment_total - credit_total) > 0
 ORDER BY vendor_name, line_item_amt DESC
```

### The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_DATE	LINE_ITEM_AMT	ACCOUNT_DESCRIPTION
1 Abbey Office Furnishings	203339-13	02-MAY-14	17.5	Office Supplies
2 Blue Cross	547481328	20-MAY-14	224	Group Insurance
3 Blue Cross	547480102	19-MAY-14	224	Group Insurance
4 Blue Cross	547479217	17-MAY-14	116	Group Insurance
5 Cardinal Business Media, Inc.	134116	01-JUN-14	90.36	Card Deck Advertising

(44 rows selected)

## Description

- Instead of coding a join condition in the FROM clause, you can code it in the WHERE clause along with any search conditions. Then, you simply list the tables you want to join in the FROM clause separated by commas.
- This syntax for coding joins is referred to as the *implicit syntax*, or the *theta syntax*. It was used prior to the SQL-92 standards, which introduced the explicit syntax.
- If you omit the join condition from the WHERE clause, a cross join is performed. You'll learn about cross joins later in this chapter.

Figure 4-7 How to use the implicit inner join syntax

## How to work with outer joins

---

Although inner joins are the type of join you'll use most often, Oracle also supports *outer joins*. Unlike an inner join, an outer join returns all of the rows from one or both tables involved in the join, regardless of whether the join condition is true. You'll see how this works in the topics that follow.

### How to code an outer join

---

Figure 4-8 presents the explicit syntax for coding an outer join. Because this syntax is similar to the explicit syntax for inner joins, you shouldn't have any trouble understanding how it works. The main difference is that you include the LEFT, RIGHT, or FULL keyword to specify the type of outer join you want to perform. As you can see in the syntax, you can also include the OUTER keyword, but it's optional and is usually omitted.

The table in this figure summarizes the differences between left, right, and full outer joins. When you use a *left outer join*, the result set includes all the rows from the first, or left, table. Similarly, when you use a *right outer join*, the result set includes all the rows from the second, or right, table. And when you use a *full outer join*, the result set includes all the rows from both tables.

The example in this figure illustrates a left outer join. Here, the Vendors table is joined with the Invoices table. Notice that the result set includes vendor rows even if no matching invoices are found. In that case, null values are returned for the columns in the Invoices table.

When coding outer joins, it's a common practice to avoid using right joins. To do that, you can substitute a left outer join for a right outer join by reversing the order of the tables in the FROM clause and using the LEFT keyword instead of RIGHT. This often makes it easier to read statements that join more than two tables.

## The explicit syntax for an outer join

```
SELECT select_list
FROM table_1
    {LEFT|RIGHT|FULL} [OUTER] JOIN table_2
        ON join_condition_1
    [{LEFT|RIGHT|FULL} [OUTER] JOIN table_3
        ON join_condition_2]...
```

## What outer joins do

Joins of this type	Keep unmatched rows from
Left outer join	The first (left) table
Right outer join	The second (right) table
Full outer join	Both tables

## A SELECT statement that uses a left outer join

```
SELECT vendor_name, invoice_number, invoice_total
FROM vendors LEFT JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
ORDER BY vendor_name
```

### The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_TOTAL
1 ASC Signs	(null)	(null)
2 AT&T	(null)	(null)
3 Abbey Office Furnishings	203339-13	17.5
4 American Booksellers Assoc	(null)	(null)
5 American Express	(null)	(null)

(202 rows selected)

## Description

- An *outer join* retrieves all rows that satisfy the join condition, plus unmatched rows in one or both tables.
- In most cases, you use the equals operator to retrieve rows with matching columns. However, you can also use any of the other comparison operators.
- When a row with unmatched columns is retrieved, any columns from the other table that are included in the result set are given null values.

## Notes

- The OUTER keyword is optional and typically omitted.
- You can also code left outer joins and right outer joins using the implicit syntax. See figure 4-11 for more information.

## Outer join examples

---

To give you a better understanding of how outer joins work, figure 4-9 presents three more examples. These examples use the Departments and Employees tables shown at the top of this figure. In each case, the join condition joins the tables based on the values in their department\_number columns.

The first SELECT statement performs a left outer join on these two tables. In the result set produced by this statement, you can see that department number 3 (Operations) is included in the result set even though none of the employees in the Employees table work in that department. Because of that, a null value is assigned to the last\_name column from that table.

The second SELECT statement uses a right outer join. In this case, all of the rows from the Employees table are included in the result set. Notice, however, that two of the employees, Locario and Watson, are assigned to a department that doesn't exist in the Departments table. Of course, if the department\_number column in this table had been defined as a foreign key to the Departments table, this would not have been allowed. In this case, though, a foreign key wasn't defined, so null values are returned for the department\_name column in these two rows.

The third SELECT statement in this figure illustrates a full outer join. If you compare the results of this query with the results of the queries that use a left and right outer join, you'll see that this is a combination of the two joins. In other words, each row in the Departments table is included in the result set, along with each row in the Employees table. Because the department\_number column from both tables is included in this example, you can clearly identify the row in the Departments table that doesn't have a matching row in the Employees table and the two rows in the Employees table that don't have matching rows in the Departments table.

### The Departments table

DEPARTMENT_NUMBER	DEPARTMENT_NAME
1	Accounting
2	Payroll
3	Operations
4	Personnel
5	Maintenance

### The Employees table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

### A left outer join

```
SELECT department_name AS dept_name,
       d.department_number AS dept_no,
       last_name
  FROM departments d
 LEFT JOIN employees e
    ON d.department_number =
       e.department_number
 ORDER BY department_name
```

DEPT_NAME	DEPT_NO	LAST_NAME
1 Accounting		1 Hernandez
2 Maintenance		5 Hardy
3 Operations		3 (null)
4 Payroll		2 Simonian
5 Payroll		2 Aaronsen
6 Payroll		2 Smith
7 Personnel		4 Jones
8 Personnel		4 O'Leary

### A right outer join

```
SELECT department_name AS dept_name,
       e.department_number AS dept_no,
       last_name
  FROM departments d
 RIGHT JOIN employees e
    ON d.department_number =
       e.department_number
 ORDER BY department_name
```

DEPT_NAME	DEPT_NO	LAST_NAME
1 Accounting		1 Hernandez
2 Maintenance		5 Hardy
3 Payroll		2 Aaronsen
4 Payroll		2 Simonian
5 Payroll		2 Smith
6 Personnel		4 Jones
7 Personnel		4 O'Leary
8 (null)		6 Locario
9 (null)		6 Watson

### A full outer join

```
SELECT department_name
      AS dept_name,
       d.department_number
      AS d_dept_no,
       e.department_number
      AS e_dept_no,
       last_name
  FROM departments d
 FULL JOIN employees e
    ON d.department_number =
       e.department_number
 ORDER BY department_name
```

DEPT_NAME	D_DEPT_NO	E_DEPT_NO	LAST_NAME
1 Accounting	1	1	Hernandez
2 Maintenance	5	5	Hardy
3 Operations	3	(null)	(null)
4 Payroll	2	2	Simonian
5 Payroll	2	2	Smith
6 Payroll	2	2	Aaronsen
7 Personnel	4	4	Jones
8 Personnel	4	4	O'Leary
9 (null)	(null)	(null)	Locario
10 (null)	(null)	(null)	Watson

Figure 4-9 Outer join examples

## Outer joins that join more than two tables

---

Like inner joins, you can use outer joins to join data from more than two tables. The two examples in figure 4-10 illustrate how this works. These examples use the Departments and Employees tables you saw in the previous figure, along with a Projects table. All three tables are shown at the top of this figure.

The first example in this figure uses left outer joins to join the data in the three tables. Here, you can see once again that none of the employees in the Employees table are assigned to the Operations department. Because of that, null values are returned for the columns in both the Employees and Projects tables. In addition, you can see that two employees, Hardy and Jones, aren't assigned to a project.

The second example in this figure uses full outer joins to join the three tables. This result set includes unmatched rows from the Departments and Employees table, just like the result set you saw in figure 4-9 that was created using a full outer join. In addition, the result set in this example includes an unmatched row from the Projects table: the one for project number P1014. In other words, none of the employees are assigned to this project.

**The Departments table**

DEPARTMENT_NUMBER	DEPARTMENT_NAME
1	Accounting
2	Payroll
3	Operations
4	Personnel
5	Maintenance

**The Employees table**

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

**The Projects table**

PROJECT_NUMBER	EMPLOYEE_ID
1 P1011	8
2 P1011	4
3 P1012	3
4 P1012	1
5 P1012	5
6 P1013	6
7 P1013	9
8 P1014	10

**A SELECT statement that joins the three tables using left outer joins**

```
SELECT department_name,
       last_name,
       project_number AS proj_no
  FROM departments d
    LEFT JOIN employees e
      ON d.department_number =
         e.department_number
    LEFT JOIN projects p
      ON e.employee_id =
         p.employee_id
 ORDER BY department_name, last_name,
          project_number
```

DEPARTMENT_NAME	LAST_NAME	PROJ_NO
1 Accounting	Hernandez	P1011
2 Maintenance	Hardy	(null)
3 Operations	(null)	(null)
4 Payroll	Aaronsen	P1012
5 Payroll	Simonian	P1012
6 Payroll	Smith	P1012
7 Personnel	Jones	(null)
8 Personnel	O'Leary	P1011

**A SELECT statement that joins the three tables using full outer joins**

```
SELECT department_name, last_name,
       project_number AS proj_no
  FROM departments dpt
    FULL JOIN employees emp
      ON dpt.department_number =
         emp.department_number
    FULL JOIN projects prj
      ON emp.employee_id =
         prj.employee_id
 ORDER BY department_name
```

DEPARTMENT_NAME	LAST_NAME	PROJ_NO
1 Accounting	Hernandez	P1011
2 Maintenance	Hardy	(null)
3 Operations	(null)	(null)
4 Payroll	Simonian	P1012
5 Payroll	Aaronsen	P1012
6 Payroll	Smith	P1012
7 Personnel	Jones	(null)
8 Personnel	O'Leary	P1011
9 (null)	Locario	P1013
10 (null)	(null)	P1014
11 (null)	Watson	P1013

Figure 4-10 Outer joins that join more than two tables

## How to use the implicit outer join syntax

---

When you're writing new queries that use outer joins, you may use either the explicit outer join syntax described in figure 4-8 or the implicit outer join syntax shown in figure 4-11. However, to make your code easier to read and maintain, you should try to be consistent with the syntax that's used elsewhere in the application you are working on.

Note, however, that you can't perform a full outer join using the implicit syntax. As a result, if you need to perform a full outer join, you'll have to use the explicit outer join syntax.

### The implicit syntax for an outer join

```
SELECT select_list
FROM table_1, table_2 [, table 3]...
WHERE table_1.column_name [(+)] table_2.column_name [(+)]
      [table_2.column_name [(+)] table_3.column_name [(+)]]...
```

### A SELECT statement that joins two tables using a left outer join

```
SELECT department_name AS dept_name,
       dpt.department_number AS dept_no,
       last_name
  FROM departments dpt, employees emp
 WHERE dpt.department_number = emp.department_number (+)
 ORDER BY department_name
```

#### The result set

DEPT_NAME	DEPT_NO	LAST_NAME
1 Accounting	1 Hernandez	
2 Maintenance	5 Hardy	
3 Operations	3 (null)	
4 Payroll	2 Simonian	
5 Payroll	2 Aaronsen	
6 Payroll	2 Smith	
7 Personnel	4 Jones	
8 Personnel	4 O'Leary	

(8 rows selected)

### A SELECT statement that joins two tables using a right outer join

```
SELECT department_name AS dept_name,
       emp.department_number AS dept_no,
       last_name
  FROM departments dpt, employees emp
 WHERE dpt.department_number (+) = emp.department_number
 ORDER BY department_name
```

#### The result set

DEPT_NAME	DEPT_NO	LAST_NAME
1 Accounting	1 Hernandez	
2 Maintenance	5 Hardy	
3 Payroll	2 Aaronsen	
4 Payroll	2 Simonian	
5 Payroll	2 Smith	
6 Personnel	4 Jones	
7 Personnel	4 O'Leary	
8 (null)	6 Locario	
9 (null)	6 Watson	

(9 rows selected)

### Description

- The implicit syntax for outer joins is an alternative to the SQL-92 standards.

## Other skills for working with joins

---

The topics that follow present other skills for working with joins. In the first topic, you'll learn how to use inner and outer joins in the same statement. Next, you'll learn how to join tables with the USING and NATURAL keywords. Then, you'll learn how to use another type of join, called a cross join.

### How to combine inner and outer joins

---

Figure 4-12 shows how you can combine inner and outer joins. In this example, the Departments table is joined with the Employees table using an inner join and the Employees table is joined to the Projects table with a left outer join. The result is a table that includes all of the departments that have employees assigned to them, all of the employees assigned to those departments, and the projects those employees are assigned to. Here, you can clearly see that two employees, Hardy and Jones, haven't been assigned projects.

**The Departments table**

	DEPARTMENT_NUMBER	DEPARTMENT_NAME
1		1 Accounting
2		2 Payroll
3		3 Operations
4		4 Personnel
5		5 Maintenance

**The Employees table**

	EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy		2
2	Jones	Elmer		4
3	Simonian	Ralph		2
4	Hernandez	Olivia		1
5	Aaronsen	Robert		2
6	Watson	Denise		6
7	Hardy	Thomas		5
8	O'Leary	Rhea		4
9	Locario	Paulo		6

**The Projects table**

	PROJECT_NUMBER	EMPLOYEE_ID
1	P1011	8
2	P1011	4
3	P1012	3
4	P1012	1
5	P1012	5
6	P1013	6
7	P1013	9
8	P1014	10

**A SELECT statement that combines an outer and an inner join**

```
SELECT department_name AS dept_name, last_name, project_number
FROM departments dpt
    JOIN employees emp
        ON dpt.department_number = emp.department_number
    LEFT JOIN projects prj
        ON emp.employee_id = prj.employee_id
ORDER BY department_name
```

**The result set**

DEPT_NAME	LAST_NAME	PROJECT_NUMBER
1 Accounting	Hernandez	P1011
2 Maintenance	Hardy	(null)
3 Payroll	Simonian	P1012
4 Payroll	Smith	P1012
5 Payroll	Aaronsen	P1012
6 Personnel	Jones	(null)
7 Personnel	O'Leary	P1011

(7 rows selected)

**Description**

- You can combine inner and outer joins within a single SELECT statement using either the explicit or the implicit join syntax.

## How to join tables with the USING keyword

---

When you use the equals operator to join two tables on a common column, the join can be referred to as an *equijoin* (or an *equi-join*). When you code an equijoin, it's common for the columns that are being compared to have the same name. For joins like these, you can simplify the query with the USING keyword that was introduced with Oracle 9i. In other words, you can code a USING clause instead of an ON clause to specify the join as shown in figure 4-13.

Here, the first example shows how to join the Vendors and Invoices tables on the vendor\_id column with a USING clause. This returns the same results as the query shown in figure 4-1 that uses the ON clause. Note that the USING clause only works because the vendor\_id column exists in both the Vendors and Invoices tables.

The second example shows how to join the Departments, Employees, and Projects tables with the USING keyword. Here, the first USING clause uses an inner join to join the Departments table to the Employees table on the department\_number column. Then, the second USING clause uses a left join to join the Employees table to the Projects table on the employee\_id column. This shows that you can use a USING clause for inner and outer joins, and it returns the same result as the query shown in figure 4-12.

In some rare cases, you may want to join a table by multiple columns. To do that with a USING clause, you can code multiple column names within the parentheses, separating each column name with a comma. This yields the same result as coding two equijoins connected with the AND operator.

Since the USING clause is more concise than the ON clause, it can make your code easier to read and maintain. As a result, it often makes sense to use the USING clause when you're developing new statements. However, if you can't get the USING clause to work correctly for a complex query, you can always use an ON clause instead.

## The syntax for a join that uses the USING keyword

```
SELECT select_list
FROM table_1
[ {LEFT|RIGHT|FULL} [OUTER] ] JOIN table_2
    USING(join_column_1[, join_column_2]...)
[ [{LEFT|RIGHT|FULL} [OUTER] ] JOIN table_3
    USING (join_column_2[, join_column_2]...) ]...
```

## A SELECT statement that uses the USING keyword to join two tables

```
SELECT invoice_number, vendor_name
FROM vendors
    JOIN invoices USING (vendor_id)
ORDER BY invoice_number
```

### The result set

INVOICE_NUMBER	VENDOR_NAME
1 0-2058	Malloy Lithographing Inc
2 0-2060	Malloy Lithographing Inc
3 0-2436	Malloy Lithographing Inc
4 1-200-5164	Federal Express Corporation

(114 rows selected)

## A SELECT statement that uses the USING keyword to join three tables

```
SELECT department_name AS dept_name, last_name, project_number
FROM departments
    JOIN employees USING (department_number)
    LEFT JOIN projects USING (employee_id)
ORDER BY department_name
```

### The result set

DEPT_NAME	LAST_NAME	PROJECT_NUMBER
1 Accounting	Hernandez	P1011
2 Maintenance	Hardy	(null)
3 Payroll	Simonian	P1012
4 Payroll	Smith	P1012
5 Payroll	Aaronsen	P1012
6 Personnel	Jones	(null)
7 Personnel	O'Leary	P1011

(7 rows selected)

## Description

- You can use the USING keyword to simplify the syntax for joining tables.
- The join can be an inner join or an outer join.
- The tables must be joined by a column that has the same name in both tables.
- If you want to include multiple columns, you can separate each table with a comma.
- The join must be an *equijoin*, which means that the equals operator is used to compare the two columns.

Figure 4-13 How to join tables with the USING keyword

## How to join tables with the NATURAL keyword

---

Figure 4-14 shows how to use the NATURAL keyword that was introduced with Oracle 9i to code a *natural join*. When you code a natural join, you don't specify the column that's used to join the two tables. Instead, the database automatically joins the two tables based on all columns in the two tables that have the same name. As a result, this type of join only works correctly if the database is designed in a certain way.

For instance, if you use a natural join to join the Vendors and Invoices tables as shown in the first example, the join works correctly because these tables only have one column in common: the vendor\_id column. As a result, the database joins these two tables on the vendor\_id column. However, if these tables had another column in common, this query would attempt to join these tables on both columns and would yield unexpected results.

Although natural joins are easy to code, these joins don't explicitly specify the join column. As a result, they might not work correctly if the structure of the database changes later. Because of that, you'll usually want to avoid using natural joins for production code.

In addition, if you use natural joins, you may get unexpected results for more complex queries. In that case, you can use the USING or ON clauses to explicitly specify the join since these clauses give you more control over the join. If necessary, you can mix a natural join with the USING or ON clauses within a single SELECT statement. In this figure, for example, the second SELECT statement uses a natural join for the first join and uses a USING clause for the second join. The result is the same as the result for the second statement in figure 4-13.

## The syntax for a join that uses the NATURAL keyword

```
SELECT select_list
FROM table_1
    NATURAL JOIN table_2
    [NATURAL JOIN table_3]...
```

## A SELECT statement that uses the NATURAL keyword to join tables

```
SELECT invoice_number, vendor_name
FROM vendors
    NATURAL JOIN invoices
ORDER BY invoice_number
```

### The result set

INVOICE_NUMBER	VENDOR_NAME
1 0-2058	Malloy Lithographing Inc
2 0-2060	Malloy Lithographing Inc
3 0-2436	Malloy Lithographing Inc
4 1-200-5164	Federal Express Corporation

(114 rows selected)

## A SELECT statement that uses the NATURAL keyword to join three tables

```
SELECT department_name AS dept_name, last_name, project_number
FROM departments
    NATURAL JOIN employees
    LEFT JOIN projects USING (employee_id)
ORDER BY department_name
```

### The result set

DEPT_NAME	LAST_NAME	PROJECT_NUMBER
1 Accounting	Hernandez	P1011
2 Maintenance	Hardy	(null)
3 Payroll	Simonian	P1012
4 Payroll	Smith	P1012
5 Payroll	Aaronsen	P1012
6 Personnel	Jones	(null)
7 Personnel	O'Leary	P1011

(7 rows selected)

## Description

- You can use the NATURAL keyword to create a *natural join* that simplifies the syntax for joining tables. A natural join can be used to join two tables based on all columns in the two tables that have the same name.
- Although the code for a natural join is shorter than the code for joins that use the ON or USING clauses, a natural join only works correctly for certain types of database structures. In addition, a natural join often yields unexpected results for complex queries. As a result, it's more common to use the ON or USING clauses to join tables.

Figure 4-14 How to join tables with the NATURAL keyword

## How to use cross joins

---

A *cross join* produces a result set that includes each row from the first table joined with each row from the second table. The result set is known as the *Cartesian product* of the tables. Figure 4-15 shows how to code a cross join using either the explicit or implicit syntax.

To use the explicit syntax, you include the CROSS JOIN keywords between the two tables in the FROM clause. Notice that because of the way a cross join works, you don't include a join condition. The same is true when you use the implicit syntax. In that case, you simply list the tables in the FROM clause and omit the join condition from the WHERE clause.

The two SELECT statements in this figure illustrate how cross joins work. Both of these statements combine data from the Departments and Employees tables. As you can see, the result is a table that includes 45 rows. That's each of the five rows in the Departments table combined with each of the nine rows in the Employees table. Although this result set is relatively small, you can imagine how large it would be if the tables included hundreds or thousands of rows.

As you study these examples, you should realize that cross joins have few practical uses. As a result, you'll rarely, if ever, need to use one. In fact, you're most likely to code a cross join by accident if you use the implicit join syntax and forget to code the join condition in the WHERE clause. That's one of the reasons why it's generally considered a good practice to use the explicit join syntax.

## How to code a cross join using the explicit syntax

### The explicit syntax for a cross join

```
SELECT select_list  
FROM table_1 CROSS JOIN table_2
```

### A cross join that uses the explicit syntax

```
SELECT departments.department_number, department_name, employee_id,  
      last_name  
FROM departments CROSS JOIN employees  
ORDER BY departments.department_number
```

## How to code a cross join using the implicit syntax

### The implicit syntax for a cross join

```
SELECT select_list  
FROM table_1, table_2
```

### A cross join that uses the implicit syntax

```
SELECT departments.department_number, department_name, employee_id,  
      last_name  
FROM departments, employees  
ORDER BY departments.department_number
```

## The result set

DEPARTMENT_NUMBER	DEPARTMENT_NAME	EMPLOYEE_ID	LAST_NAME
1	1 Accounting	4 Hernandez	
2	1 Accounting	3 Simonian	
3	1 Accounting	9 Locario	
4	1 Accounting	8 O'Leary	
5	1 Accounting	7 Hardy	
6	1 Accounting	6 Watson	
7	1 Accounting	5 Aaronsen	

(45 rows selected)

## Description

- A *cross join* joins each row from the first table with each row from the second table. The result set returned by a cross join is known as a *Cartesian product*.
- To code a cross join using the explicit syntax, use the CROSS JOIN keywords in the FROM clause.
- To code a cross join using the implicit syntax, list the tables in the FROM clause and omit the join condition from the WHERE clause.

Figure 4-15 How to use cross joins

## How to work with unions

---

Like a join, a *union* combines data from two or more tables. Instead of combining columns from base tables, however, a union combines rows from two or more result sets. You'll see how that works in the topics that follow.

### The syntax of a union

---

Figure 4-16 shows how to code a union. As the syntax shows, you create a union by connecting two or more SELECT statements with the UNION keyword. For this to work, the result of each SELECT statement must have the same number of columns, and the data types of the corresponding columns in each table must be compatible.

In this syntax, we have indented all of the SELECT statements that are connected by the UNION operator to make it easier to see how this statement works. However, in a production environment, it's common to see the SELECT statements and the UNION operator coded at the same level of indentation.

If you want to sort the result of a union operation, you can code an ORDER BY clause after the last SELECT statement. Note that the column names you use in this clause must be the same as those used in the first SELECT statement. That's because the column names you use in the first SELECT statement are the ones that are used in the result set.

By default, a union operation removes duplicate rows from the result set. If that's not what you want, you can include the ALL keyword. In most cases, though, you'll omit this keyword.

### Unions that combine data from different tables

---

The example in this figure shows how to use a union to combine data from two different tables. In this case, the Active\_Invoices table contains invoices with outstanding balances, and the Paid\_Invoices table contains invoices that have been paid in full. Both of these tables have the same structure as the Invoices table you've seen in previous figures.

This union operation combines the rows in both tables that have an invoice date on or after June 1, 2014. Notice that the first SELECT statement includes a column named Source that contains the literal value "Active." The second SELECT statement includes a column by the same name, but it contains the literal value "Paid." This column is used to indicate which table each row in the result set came from.

Although this column is assigned the same name in both SELECT statements, you should realize that doesn't have to be the case. In fact, none of the columns have to have the same names. Corresponding columns do have to have compatible data types. But the corresponding relationships are determined by the order in which the columns are coded in the SELECT clauses, not by their names. When you use column aliases, though, you'll typically assign the same name to corresponding columns so that the statement is easier to understand.

## The syntax for a union operation

```

SELECT_statement_1
UNION [ALL]
  SELECT_statement_2
[UNION [ALL]
  SELECT_statement_3]...
[ORDER BY order_by_list]

```

## A union that combines invoice data from two different tables

```

SELECT 'Active' AS source, invoice_number, invoice_date, invoice_total
FROM active_invoices
WHERE invoice_date >= '01-JUN-2014'
UNION
  SELECT 'Paid' AS source, invoice_number, invoice_date, invoice_total
  FROM paid_invoices
  WHERE invoice_date >= '01-JUN-2014'
ORDER BY invoice_total DESC

```

## The result set

SOURCE	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 Active	40318	18-JUL-14	21842
2 Paid	P02-3772	03-JUN-14	7125.34
3 Paid	10843	04-JUN-14	4901.26
4 Paid	77290	04-JUN-14	1750
5 Paid	RTR-72-3662-X	04-JUN-14	1600
6 Paid	75C-90227	06-JUN-14	1367.5
7 Paid	P02-88D77S7	06-JUN-14	856.92
8 Active	I77271-001	05-JUN-14	662
9 Active	9982771	03-JUN-14	503.2

(22 rows selected)

## Description

- A *union* combines the result sets of two or more SELECT statements into one result set.
- Each result set must return the same number of columns, and the corresponding columns in each result set must have compatible data types.
- By default, a union eliminates duplicate rows. If you want to include duplicate rows, code the ALL keyword.
- The column names in the final result set are taken from the first SELECT clause. Column aliases assigned by the other SELECT clauses have no effect on the final result set.
- To sort the rows in the final result set, code an ORDER BY clause after the last SELECT statement. This clause must refer to the column names assigned in the first SELECT clause.

---

Figure 4-16 How to combine data from different tables

## Unions that combine data from the same table

---

Figure 4-17 shows how to use unions to combine data from a single table. In the first example, rows from the Invoices table that have a balance due are combined with rows from the same table that are paid in full. As in the example in the previous figure, a column named Source is added at the beginning of each interim table. That way, the final result set indicates whether each invoice is active or paid.

The second example in this figure shows how you can use a union with data that's joined from two tables. Here, each SELECT statement joins data from the Invoices and Vendors tables. The first SELECT statement retrieves invoices with totals greater than \$10,000. Then, it calculates a payment of 33% of the invoice total. The two other SELECT statements are similar. The second one retrieves invoices with totals between \$500 and \$10,000 and calculates a 50% payment. And the third one retrieves invoices with totals less than \$500 and sets the payment amount at 100% of the total. Although this is unrealistic, it helps illustrate the flexibility of union operations.

Notice in this example that the same column aliases are assigned in each SELECT statement. Although the aliases in the second and third SELECT statements have no effect on the query, they make the query easier to read. In particular, it makes it easy to see that the three SELECT statements have the same number and types of columns.

## A union that combines information from the Invoices table

```

SELECT 'Active' AS source, invoice_number, invoice_date, invoice_total
FROM invoices
WHERE (invoice_total - payment_total - credit_total) > 0
UNION
SELECT 'Paid' AS source, invoice_number, invoice_date, invoice_total
FROM invoices
WHERE (invoice_total - payment_total - credit_total) <= 0
ORDER BY invoice_total DESC

```

### The result set

SOURCE	INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 Paid	0-2058	08-MAY-14	37966.19
2 Paid	P-0259	16-APR-14	26881.4
3 Paid	0-2060	08-MAY-14	23517.58
4 Active	40318	18-JUL-14	21842
5 Active	P-0608	11-APR-14	20551.18
6 Active	0-2436	07-MAY-14	10976.06

(114 rows selected)

## A union that combines payment data from the same joined tables

```

SELECT invoice_number, vendor_name, '33% Payment' AS payment_type,
       invoice_total AS total, (invoice_total * 0.333) AS payment
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
   WHERE invoice_total > 10000
UNION
      SELECT invoice_number, vendor_name, '50% Payment' AS payment_type,
             invoice_total AS total, (invoice_total * 0.5) AS payment
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
   WHERE invoice_total BETWEEN 500 AND 10000
UNION
      SELECT invoice_number, vendor_name, 'Full amount' AS payment_type,
             invoice_total AS Total, invoice_total AS Payment
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
   WHERE invoice_total < 500
ORDER BY payment_type, vendor_name, invoice_number

```

### The result set

INVOICE_NUMBER	VENDOR_NAME	PAYMENT_TYPE	TOTAL	PAYMENT
1 40318	Data Reproductions Corp	33% Payment	21842	7273.386
2 0-2058	Malloy Lithographing Inc	33% Payment	37966.19	12642.74127
3 0-2060	Malloy Lithographing Inc	33% Payment	23517.58	7831.35414
4 0-2436	Malloy Lithographing Inc	33% Payment	10976.06	3655.02798
5 P-0259	Malloy Lithographing Inc	33% Payment	26881.4	8951.5062
6 P-0608	Malloy Lithographing Inc	33% Payment	20551.18	6843.54294
7 509786	Bertelsmann Industry Svcs. Inc	50% Payment	6940.25	3470.125

(114 rows selected)

Figure 4-17 Unions that combine data from the same table

## How to use the MINUS and INTERSECT operators

---

Like the UNION operator, the MINUS and INTERSECT operators work with two or more result sets, as shown in figure 4-18. Because of that, all three of these operators can be referred to as *set operators*. In addition, the MINUS and INTERSECT operators follow many of the same rules as the UNION operator.

The first query shown in this figure uses the MINUS operator to return the first and last names of all customers in the Customers table except any customers whose first and last names also exist in the Employees table. Since Thomas Hardy is the only name that's the same in both tables, this is the only record that's excluded from the result set for the query that comes before the MINUS operator.

The second query shown in this figure uses the INTERSECT operator to return the first and last names of all customers in the Customers table whose first and last names also exist in the Employees table. Since Thomas Hardy is the only name that exists in both tables, this is the only record that's returned for the result set for this query.

When you use the MINUS and INTERSECT operators, you must follow many of the same rules for working with the UNION operator. To start, both of the statements that are connected by these operators must return the same number of columns. In addition, the data types for these columns must be compatible. Finally, when two queries are joined by a MINUS or INTERSECT operator, the column names in the final result set are taken from the first query. When you code the ORDER BY clause, you can specify the column names in the first query (customer\_last\_name in our example), or you can specify the column position (2). If you understand how the UNION operator works, you shouldn't have any trouble understanding these rules.

## The syntax for the MINUS and INTERSECT operations

```
SELECT_statement_1
{MINUS | INTERSECT}
  SELECT_statement_2
[ORDER BY order_by_list]
```

### The Customers table

CUSTOMER_LAST_NAME	CUSTOMER_FIRST_NAME
Anders	Maria
Trujillo	Ana
Moreno	Antonio
Hardy	Thomas
Berglund	Christina
Moos	Hanna

### The Employees table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

## A query that excludes rows from the first query if they also occur in the second query

```
SELECT customer_first_name, customer_last_name
  FROM customers
MINUS
  SELECT first_name, last_name
    FROM employees
   ORDER BY customer_last_name
```

### The result set

CUSTOMER_FIRST_NAME	CUSTOMER_LAST_NAME
1 Maria	Anders
2 Christina	Berglund
3 Art	Braunschweiger
4 Donna	Chelan

(23 rows selected)

## A query that only includes rows that occur in both queries

```
SELECT customer_first_name, customer_last_name
  FROM customers
INTERSECT
  SELECT first_name, last_name
    FROM employees
```

### The result set

CUSTOMER_FIRST_NAME	CUSTOMER_LAST_NAME
1 Thomas	Hardy

(1 row selected)

## Description

- The number of columns must be the same in both SELECT statements.
- The data types for each column must be compatible.
- The column names in the final result set are taken from the first SELECT statement.

## Perspective

---

In this chapter, you learned a variety of techniques for combining data from two or more tables into a single result set. In particular, you learned how to use the SQL-92 syntax for combining data using inner joins. Of all the techniques presented in this chapter, this is the one you'll use most often. So you'll want to be sure you understand it thoroughly before you go on.

### Terms

---

join	theta syntax
join condition	outer join
inner join	left outer join
ad hoc relationship	right outer join
qualified column name	full outer join
explicit syntax	equijoin
table alias	natural join
schema	cross join
user	Cartesian product
schema owner	union
self-join	set operator
implicit syntax	

### Exercises

1. Write a SELECT statement that returns all columns from the Vendors table inner-joined with all columns from the Invoices table.
2. Write a SELECT statement that returns four columns:

vendor_name	vendor_name from the Vendors table
invoice_number	invoice_number from the Invoices table
invoice_date	invoice_date from the Invoices table
balance_due	invoice_total minus payment_total minus credit_total from the Invoices table

The result set should have one row for each invoice with a non-zero balance.  
Sort the result set by vendor\_name in ascending order.

3. Write a SELECT statement that returns three columns:

vendor_name	vendor_name from the Vendors table
default_account	default_account_number from the Vendors table
description	account_description from the General_Ledger_Accounts table

The result set should have one row for each vendor, and it should be sorted by account\_description and then by vendor\_name.

4. Write a SELECT statement that returns five columns from three tables:

vendor_name	vendor_name from the Vendors table
invoice_date	invoice_date from the Invoices table
invoice_number	invoice_number from the Invoices table
li_sequence	invoice_sequence from the Invoice_Line_Items table
li_amount	line_item_amt from the Invoice_Line_Items table

Use these aliases for the tables: Ven for the Vendors table, Inv for the Invoices table, and LI for the Invoice\_Line\_Items table. Also, sort the final result set by vendor\_name, invoice\_date, invoice\_number, and invoice\_sequence.

5. Write a SELECT statement that returns three columns:

vendor_id	vendor_id from the Vendors table
vendor_name	vendor_name from the Vendors table
contact_name	A concatenation of vendor_contact_first_name and vendor_contact_last_name with a space in between

The result set should have one row for each vendor whose contact has the same last name as another vendor's contact, and it should be sorted by vendor\_contact\_last\_name. *Hint: Use a self-join.*

6. Write a SELECT statement that returns two columns from the General\_Ledger\_Accounts table: account\_number and account\_description. The result set should have one row for each account number that has never been used. Sort the final result set by account\_number. *Hint: Use an outer join to the Invoice\_Line\_Items table.*
7. Use the UNION operator to generate a result set consisting of two columns from the Vendors table: vendor\_name and vendor\_state. If the vendor is in California, the vendor\_state value should be "CA"; otherwise, the VendorState value should be "Outside CA." Sort the final result set by vendor\_name.



# 5

## How to code summary queries

In this chapter, you'll learn how to code queries that summarize data. For example, you can use summary queries to report sales totals by vendor or state, or to get a count of the number of invoices that were processed each day of the month. You'll also learn how to use a special type of function called an aggregate function. Aggregate functions allow you to do jobs like calculate averages, summarize totals, or find the highest value for a given column.

<b>How to work with aggregate functions .....</b>	<b>162</b>
How to code aggregate functions .....	162
Queries that use aggregate functions.....	164
<b>How to group and summarize data.....</b>	<b>166</b>
How to code the GROUP BY and HAVING clauses .....	166
Queries that use the GROUP BY and HAVING clauses.....	168
How the HAVING clause compares to the WHERE clause .....	170
How to code complex search conditions .....	172
<b>How to summarize data using Oracle extensions .....</b>	<b>174</b>
How to use the ROLLUP operator .....	174
How to use the CUBE operator .....	176
<b>Perspective .....</b>	<b>178</b>

## How to work with aggregate functions

---

In chapter 3, you were introduced to *scalar functions*, which operate on a single value and return a single value. In this chapter, you'll learn how to use *aggregate functions*, which operate on a series of values and return a single summary value. Because aggregate functions typically operate on the values in columns, they are sometimes referred to as *column functions*. A query that contains one or more aggregate functions is typically referred to as a *summary query*.

### How to code aggregate functions

---

Figure 5-1 presents the syntax of the most common aggregate functions. Since the purpose of these functions is self-explanatory, we'll focus mainly on how you use them.

All of these functions but one operate on an expression. In the query in this figure, for example, the expression that's coded for the SUM function calculates the balance due of an invoice using the invoice\_total, payment\_total, and credit\_total columns. The result is a single value that represents the total amount due for all the selected invoices. If you look at the WHERE clause in this example, you'll see that it includes only those invoices with a balance due.

In addition to an expression, you can also code the ALL or DISTINCT keyword in these functions. ALL is the default, which means that all values are included in the calculation. The exceptions are null values, which are always excluded from these functions.

If you don't want duplicate values included, you can code the DISTINCT keyword. In most cases, you'll use DISTINCT only with the COUNT function. You'll see an example of that in the next figure. You won't use it with MIN or MAX because it has no effect on those functions. And it doesn't usually make sense to use it with the AVG and SUM functions.

Unlike the other aggregate functions, you can't use the ALL or DISTINCT keywords or an expression with COUNT(\*). Instead, you code this function exactly as shown in the syntax. The value returned by this function is the number of rows in the base table that satisfy the search condition of the query, including rows with null values. The COUNT(\*) function in the query in this figure, for example, indicates that the Invoices table contains 40 invoices with a balance due.

## The syntax of the aggregate functions

Function syntax	Result
<code>AVG( [ALL DISTINCT] expression)</code>	The average of the non-null values in the expression.
<code>SUM( [ALL DISTINCT] expression)</code>	The total of the non-null values in the expression.
<code>MIN( [ALL DISTINCT] expression)</code>	The lowest non-null value in the expression.
<code>MAX( [ALL DISTINCT] expression)</code>	The highest non-null value in the expression.
<code>COUNT( [ALL DISTINCT] expression)</code>	The number of non-null values in the expression.
<code>COUNT(*)</code>	The number of rows selected by the query.

## A summary query that counts unpaid invoices and calculates the total due

```
SELECT COUNT(*) AS number_of_invoices,
       SUM(invoice_total - payment_total - credit_total) AS total_due
  FROM invoices
 WHERE invoice_total - payment_total - credit_total > 0
```

NUMBER_OF_INVOICES	TOTAL_DUE
1	40 66796.24

## Description

- *Aggregate functions*, also called *column functions*, perform a calculation on the values in a set of selected rows. You specify the values to be used in the calculation by coding an expression for the function's argument. In many cases, the expression is just the name of a column.
- A SELECT statement that includes an aggregate function can be called a *summary query*.
- The expression you specify for the AVG and SUM functions must result in a numeric value. The expression for the MIN, MAX, and COUNT functions can result in a numeric, date, or string value.
- By default, all values are included in the calculation regardless of whether they're duplicated. If you want to omit duplicate values, code the DISTINCT keyword. This keyword is typically used only with the COUNT function.
- All of the aggregate functions except for COUNT(\*) ignore null values.
- Aggregate functions are often used with the GROUP BY clause of the SELECT statement, which is used to group the rows in a result set. See figure 5-3 for more information.
- If you code an aggregate function in the SELECT clause, that clause can't include non-aggregate columns from the base table.

## Queries that use aggregate functions

---

Figure 5-2 presents four more queries that use aggregate functions. Before we describe these queries, you should know that with two exceptions, a SELECT clause that contains an aggregate function can contain only aggregate functions. The first exception is if the column specification results in a literal value. This is illustrated by the first column in the first two queries in this figure. The second exception is if the query includes a GROUP BY clause. Then, the SELECT clause can include any columns specified in the GROUP BY clause. You'll see how you use the GROUP BY clause later in this chapter.

The first two queries in this figure use the COUNT(\*) function to count the number of rows in the Invoices table that satisfy the search condition. In both cases, only those invoices with invoice dates after 1/1/2014 are included in the count. In addition, the first query uses the AVG function to calculate the average amount of those invoices and the SUM function to calculate the total amount of those invoices. In contrast, the second query uses the MIN and MAX functions to calculate the minimum and maximum invoice amounts.

Although the MIN, MAX, and COUNT functions are typically used on columns that contain numeric data, they can also be used on columns that contain character or date data. In the third query, for example, they're used on the vendor\_name column in the Vendors table. Here, the MIN function returns the name of the vendor that's lowest in the sort sequence, the MAX function returns the name of the vendor that's highest in the sort sequence, and the COUNT function returns the total number of vendors. Note that since the vendor\_name column can't contain null values, the COUNT(\*) function would have returned the same result.

The fourth query illustrates how using the DISTINCT keyword can affect the result of a COUNT function. Here, the first COUNT function uses the DISTINCT keyword to count the number of vendors that have invoices dated 1/1/2014 or later in the Invoices table. To do that, it looks for distinct values in the vendor\_id column. In contrast, because the second COUNT function doesn't include the DISTINCT keyword, it counts every invoice that's dated 1/1/2014 or later. Of course, you could accomplish the same thing using the COUNT(\*) function. COUNT(vendor\_id) is used here only to illustrate the difference between coding and not coding the DISTINCT keyword.

## A summary query that uses the COUNT(\*), AVG, and SUM functions

```
SELECT 'After 1/1/2014' AS selection_date,
       COUNT(*) AS number_of_invoices,
       ROUND(AVG(invoice_total), 2) AS avg_invoice_amt,
       SUM(invoice_total) AS total_invoice_amt
  FROM invoices
 WHERE invoice_date > '01-JAN-2014'
```

SELECTION_DATE	NUMBER_OF_INVOICES	AVG_INVOICE_AMT	TOTAL_INVOICE_AMT
1 After 1/1/2014	114	1879.74	214290.51

## A summary query that uses the MIN and MAX functions

```
SELECT 'After 1/1/2014' AS selection_date, COUNT(*) AS number_of_invoices,
       MAX(invoice_total) AS highest_invoice_total,
       MIN(invoice_total) AS lowest_invoice_total
  FROM invoices
 WHERE invoice_date > '01-JAN-2014'
```

SELECTION_DATE	NUMBER_OF_INVOICES	HIGHEST_INVOICE_TOTAL	LOWEST_INVOICE_TOTAL
1 After 1/1/2014	114	37966.19	6

## A summary query that works on non-numeric columns

```
SELECT MIN(vendor_name) AS first_vendor,
       MAX(vendor_name) AS last_vendor,
       COUNT(vendor_name) AS number_of_vendors
  FROM vendors
```

FIRST_VENDOR	LAST_VENDOR	NUMBER_OF_VENDORS
1 ASC Signs	Zylka Design	122

## A summary query that uses the DISTINCT keyword

```
SELECT COUNT(DISTINCT vendor_id) AS number_of_vendors,
       COUNT(vendor_id) AS number_of_invoices,
       ROUND(AVG(invoice_total),2) AS avg_invoice_amt,
       SUM(invoice_total) AS total_invoice_amt
  FROM invoices
 WHERE invoice_date > '01-JAN-2014'
```

NUMBER_OF_VENDORS	NUMBER_OF_INVOICES	AVG_INVOICE_AMT	TOTAL_INVOICE_AMT
1	34	114	1879.74

## Notes

- If you want to count all of the selected rows, you'll typically use the COUNT(\*) function as illustrated by the first two examples above. An alternative is to code the name of any column in the base table that can't contain null values, as illustrated by the third example.
- If you want to count only the rows with unique values in a specified column, you can code the COUNT function with the DISTINCT keyword followed by the name of the column, as illustrated in the fourth example.

## How to group and summarize data

---

Now that you understand how aggregate functions work, you're ready to learn how to group data and use aggregate functions to summarize the data in each group. To do that, you need to learn about two new clauses of the SELECT statement: GROUP BY and HAVING.

### How to code the GROUP BY and HAVING clauses

---

Figure 5-3 presents the syntax of the SELECT statement with the GROUP BY and HAVING clauses. The GROUP BY clause determines how the selected rows are grouped, and the HAVING clause determines which groups are included in the final results. As you can see, these clauses are coded after the WHERE clause but before the ORDER BY clause. That makes sense because the search condition in the WHERE clause is applied before the rows are grouped, and the sort sequence in the ORDER BY clause is applied after the rows are grouped.

In the GROUP BY clause, you list one or more columns or expressions separated by commas. Then, the rows that satisfy the search condition in the WHERE clause are grouped by those columns or expressions in ascending sequence. That means that a single row is returned for each unique set of values in the GROUP BY columns. This will make more sense when you see the examples in the next figure that group by two columns. For now, take a look at the example in this figure that groups by a single column.

This example calculates the average invoice amount for each vendor who has invoices in the Invoices table that average over \$2,000. To do that, it groups the invoices by vendor\_id. Then, the AVG function calculates the average of the invoice\_total column. Because this query includes a GROUP BY clause, this function calculates the average invoice total for each group rather than for the entire result set. In that case, the aggregate function is called a *vector aggregate*. In contrast, aggregate functions like the ones you saw earlier in this chapter that return a single value for all the rows in a result set are called *scalar aggregates*.

The example in this figure also includes a HAVING clause. The search condition in this clause specifies that only those vendors with invoices that average over \$2,000 should be included. Note that this condition must be applied after the rows are grouped and the average for each group has been calculated.

In addition to the AVG function, the SELECT clause includes the vendor\_id column. That makes sense since the rows are grouped by this column. However, the columns used in the GROUP BY clause don't have to be included in the SELECT clause.

## The syntax of the SELECT statement with the GROUP BY and HAVING clauses

```
SELECT select_list  
FROM table_source  
[WHERE search_condition]  
[GROUP BY group_by_list]  
[HAVING search_condition]  
[ORDER BY order_by_list]
```

### A summary query that calculates the average invoice amount by vendor

```
SELECT vendor_id, ROUND(AVG(invoice_total), 2) AS average_invoice_amount  
FROM invoices  
GROUP BY vendor_id  
HAVING AVG(invoice_total) > 2000  
ORDER BY average_invoice_amount DESC
```

VENDOR_ID	AVERAGE_INVOICE_AMOUNT
1	110
2	72
3	104
4	99
5	119
6	122
7	86
8	100

(8 rows selected)

### Description

- The GROUP BY clause groups the rows of a result set based on one or more columns or expressions. It's typically used in SELECT statements that include aggregate functions.
- If you include aggregate functions in the SELECT clause, the aggregate is calculated for each set of values that result from the columns named in the GROUP BY clause.
- If you include two or more columns or expressions in the GROUP BY clause, they form a hierarchy where each column or expression is subordinate to the previous one.
- When a SELECT statement includes a GROUP BY clause, the SELECT clause can include aggregate functions, the columns used for grouping, and expressions that result in a constant value.
- A group-by list typically consists of the names of one or more columns separated by commas. However, it can contain any expression except for those that contain aggregate functions.
- The HAVING clause specifies a search condition for a group or an aggregate. This condition is applied after the rows that satisfy the search condition in the WHERE clause are grouped.

---

Figure 5-3 How to code the GROUP BY and HAVING clauses

## Queries that use the GROUP BY and HAVING clauses

---

Figure 5-4 presents three more queries that group data. If you understood the query in the last figure, you shouldn't have any trouble understanding how the first query in this figure works. It groups the rows in the Invoices table by vendor\_id and returns a count of the number of invoices for each vendor.

The second query in this figure illustrates how you can group by more than one column. Here, a join is used to combine the vendor\_state and vendor\_city columns from the Vendors table with a count and average of the invoices in the Invoices table. Because the rows are grouped by both state and city, a row is returned for each state and city combination. Then, the ORDER BY clause sorts the rows by city within state. Without this clause, the rows would be returned in no particular sequence.

The third query is identical to the second query except that it includes a HAVING clause. This clause uses the COUNT function to limit the state and city groups that are included in the result set to those that have two or more invoices. In other words, it excludes groups that have only one invoice.

### A summary query that counts the number of invoices by vendor

```
SELECT vendor_id, COUNT(*) AS invoice_qty
FROM invoices
GROUP BY vendor_id
ORDER BY vendor_id
```

VENDOR_ID	INVOICE_QTY
1	34
2	37
3	48
4	72

(34 rows selected)

### A summary query that calculates the number of invoices and the average invoice amount for the vendors in each state and city

```
SELECT vendor_state, vendor_city, COUNT(*) AS invoice_qty,
       ROUND(AVG(invoice_total),2) AS invoice_avg
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
 GROUP BY vendor_state, vendor_city
 ORDER BY vendor_state, vendor_city
```

VENDOR_STATE	VENDOR_CITY	INVOICE_QTY	INVOICE_AVG
1 AZ	Phoenix	1	662
2 CA	Fresno	19	1208.75
3 CA	Los Angeles	1	503.2
4 CA	Oxnard	3	188

(20 rows selected)

### A summary query that limits the groups to those with two or more invoices

```
SELECT vendor_state, vendor_city, COUNT(*) AS invoice_qty,
       ROUND(AVG(invoice_total),2) AS invoice_avg
  FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
 GROUP BY vendor_state, vendor_city
 HAVING COUNT(*) >= 2
 ORDER BY vendor_state, vendor_city
```

VENDOR_STATE	VENDOR_CITY	INVOICE_QTY	INVOICE_AVG
1 CA	Fresno	19	1208.75
2 CA	Oxnard	3	188
3 CA	Pasadena	5	196.12
4 CA	Sacramento	7	253

(12 rows selected)

### Note

- You can use a join with a summary query to group and summarize the data in two or more tables.

## How the HAVING clause compares to the WHERE clause

---

As you've seen, you can limit the groups included in a result set by coding a search condition in the HAVING clause. In addition, you can apply a search condition to each row before it's included in a group. To do that, you code the search condition in the WHERE clause just as you would for any SELECT statement. To make sure you understand the differences between search conditions coded in the HAVING and WHERE clauses, figure 5-5 presents two examples.

In the first example, the invoices in the Invoices table are grouped by vendor name, and a count and average invoice amount are calculated for each group. Then, the HAVING clause limits the groups in the result set to those that have an average invoice total greater than \$500.

In contrast, the second example includes a search condition in the WHERE clause that limits the invoices included in the groups to those that have an invoice total greater than \$500. In other words, the search condition in this example is applied to every row. In the previous example, it was applied to each group of rows.

Beyond this, there are also two differences in the expressions that you can include in the WHERE and HAVING clauses. First, the HAVING clause can include aggregate functions as you saw in the first example in this figure, but the WHERE clause can't. That's because the search condition in a WHERE clause is applied before the rows are grouped. Second, although the WHERE clause can refer to any column in the base tables, the HAVING clause can only refer to columns included in the SELECT clause. That's because it filters the summarized result set that's defined by the SELECT, FROM, WHERE, and GROUP BY clauses. In other words, it doesn't filter the base tables.

## A summary query with a search condition in the HAVING clause

```
SELECT vendor_name, COUNT(*) AS invoice_qty,
       ROUND(AVG(invoice_total),2) AS invoice_avg
  FROM vendors JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
 GROUP BY vendor_name
 HAVING AVG(invoice_total) > 500
 ORDER BY invoice_qty DESC
```

VENDOR_NAME	INVOICE_QTY	INVOICE_AVG
1 United Parcel Service	9	2575.33
2 Zylka Design	8	867.53
3 Malloy Lithographing Inc	5	23978.48
4 IBM	2	600.06

(19 rows selected)

## A summary query with a search condition in the WHERE clause

```
SELECT vendor_name, COUNT(*) AS invoice_qty,
       ROUND(AVG(invoice_total),2) AS invoice_avg
  FROM vendors JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
 WHERE invoice_total > 500
 GROUP BY vendor_name
 ORDER BY invoice_qty DESC
```

VENDOR_NAME	INVOICE_QTY	INVOICE_AVG
1 United Parcel Service	9	2575.33
2 Zylka Design	7	946.67
3 Malloy Lithographing Inc	5	23978.48
4 Ingram	2	1077.21

(20 rows selected)

## Description

- When you include a WHERE clause in a SELECT statement that uses grouping and aggregates, the search condition is applied before the rows are grouped and the aggregates are calculated. That way, only the rows that satisfy the search condition are grouped and summarized.
- When you include a HAVING clause in a SELECT statement that uses grouping and aggregates, the search condition is applied after the rows are grouped and the aggregates are calculated. That way, only the groups that satisfy the search condition are included in the result set.
- A HAVING clause can only refer to a column included in the SELECT clause. A WHERE clause can refer to any column in the base tables.
- Aggregate functions can only be coded in the HAVING clause. A WHERE clause can't contain aggregate functions.

Figure 5-5 How the HAVING clause compares to the WHERE clause

## How to code complex search conditions

---

You can code compound search conditions in a HAVING clause just as you can in a WHERE clause. This is illustrated by the first query in figure 5-6. This query groups invoices by invoice date and calculates a count of the invoices and the sum of the invoice totals for each date. In addition, the HAVING clause specifies three conditions. First, the invoice date must be between 5/1/2014 and 5/31/2014. Second, the invoice count must be greater than 1. And third, the sum of the invoice totals must be greater than \$100.

Because the second and third conditions in the HAVING clause in this query include aggregate functions, they must be coded in the HAVING clause. The first condition, however, doesn't include an aggregate function, so it could be coded in either the HAVING or WHERE clause. The second statement in this figure, for example, shows this condition coded in the WHERE clause. Note that the query returns the same result set regardless of where you code this condition.

So how do you know where to code a search condition? In general, I think your code will be easier to read if you include all the search conditions in the HAVING clause. If, on the other hand, you prefer to code non-aggregate search conditions in the WHERE clause, that's OK, too.

Since a search condition in the WHERE clause is applied before the rows are grouped while a search condition in the HAVING clause isn't applied until after the grouping, you might expect a performance advantage by coding all search conditions in the HAVING clause. However, Oracle takes care of this performance issue for you when it optimizes the query. To do that, it automatically moves search conditions to whichever clause will result in the best performance, as long as that doesn't change the logic of your query. As a result, you can code search conditions wherever they result in the most readable code without worrying about system performance.

## A summary query with a compound condition in the HAVING clause

```
SELECT
    invoice_date,
    COUNT(*) AS invoice_qty,
    SUM(invoice_total) AS invoice_sum
FROM invoices
GROUP BY invoice_date
HAVING invoice_date BETWEEN '01-MAY-2014' AND '31-MAY-2014'
    AND COUNT(*) > 1
    AND SUM(invoice_total) > 100
ORDER BY invoice_date DESC
```

## The same query coded with a WHERE clause

```
SELECT
    invoice_date,
    COUNT(*) AS invoice_qty,
    SUM(invoice_total) AS invoice_sum
FROM invoices
WHERE invoice_date BETWEEN '01-MAY-2014' AND '31-MAY-2014'
GROUP BY invoice_date
HAVING COUNT(*) > 1
    AND SUM(invoice_total) > 100
ORDER BY invoice_date DESC
```

## The result set returned by both queries

INVOICE_DATE	INVOICE_QTY	INVOICE_SUM
1 31-MAY-14	3	11557.75
2 23-MAY-14	6	2761.17
3 22-MAY-14	2	442.5
4 20-MAY-14	3	308.64

(15 rows selected)

## Description

- You can use the AND and OR operators to code compound search conditions in a HAVING clause just as you can in a WHERE clause.
- If a search condition includes an aggregate function, it must be coded in the HAVING clause. Otherwise, it can be coded in either the HAVING or the WHERE clause.
- In most cases, your code will be easier to read if you code all the search conditions in the HAVING clause, but you can code non-aggregate search conditions in the WHERE clause if you prefer.

## How to summarize data using Oracle extensions

---

So far, this chapter has discussed standard SQL keywords and functions. However, you should also know about two extensions that Oracle provides for summarizing data: the ROLLUP and CUBE operators. You'll learn how to use these operators in the topics that follow.

### How to use the ROLLUP operator

---

You can use the ROLLUP operator to add one or more summary rows to a result set that uses grouping and aggregates. The two examples in figure 5-7 illustrate how this works.

The first example shows how the ROLLUP operator works when you group by a single column. Here, the invoices in the Invoices table are grouped by vendor\_id, and an invoice count and invoice total are calculated for each vendor. In addition, because the ROLLUP operator is included in the GROUP BY clause, a summary row is added at the end of the result set. This row summarizes all the aggregate columns in the result set. In this case, it summarizes the invoice\_count and invoice\_total columns. Because the vendor\_id column can't be summarized, it's assigned a null value.

The second query in this figure shows how the ROLLUP operator works when you group by two columns. This query groups the vendors in the Vendors table by state and city and counts the number of vendors in each group. Then, in addition to a summary row at the end of the result set, summary rows are included for each state.

When you use an ORDER BY clause with the ROLLUP operator, the rows are sorted after the summary rows are added. Then, because null values come after other values in the Oracle sort sequence, the summary rows come after the rows that they summarize. Incidentally, if the ORDER BY clause in the second example were omitted, the only difference in the result set would be that the cities within each state wouldn't necessarily be in the right sequence.

You can also use another function, the GROUPING function, to work with null columns in a summary row. However, this function is typically used in conjunction with the CASE function, which you'll learn about in chapter 8. So we'll present the GROUPING function in that chapter.

## A summary query that includes a final summary row

```
SELECT vendor_id, COUNT(*) AS invoice_count,
       SUM(invoice_total) AS invoice_total
  FROM invoices
 GROUP BY ROLLUP(vendor_id)
```

VENDOR_ID	INVOICE_COUNT	INVOICE_TOTAL
32	121	8 6940.25
33	122	9 23177.96
34	123	47 4378.02
35	(null)	114 214290.51

(35 rows selected)

## A summary query that includes a summary row for each grouping level

```
SELECT vendor_state, vendor_city, COUNT(*) AS qty_vendors
  FROM vendors
 WHERE vendor_state IN ('IA', 'NJ')
 GROUP BY ROLLUP (vendor_state, vendor_city)
 ORDER BY vendor_state, vendor_city
```

VENDOR_STATE	VENDOR_CITY	QTY_VENDORS
1 IA	Fairfield	1
2 IA	Washington	1
3 IA	(null)	2
4 NJ	East Brunswick	2
5 NJ	Fairfield	1
6 NJ	Washington	1
7 NJ	(null)	4
8 (null)	(null)	6

## Description

- You can use the ROLLUP operator in the GROUP BY clause to add summary rows to the final result set. A summary is provided for each aggregate column included in the select list. All other columns, except the ones that identify which group is being summarized, are assigned null values.
- The ROLLUP operator adds a summary row for each group specified in the GROUP BY clause except for the rightmost group, which is summarized by the aggregate functions. It also adds a summary row to the end of the result set that summarizes the entire result set. If the GROUP BY clause specifies a single group, only the final summary row is added.
- The sort sequence in the ORDER BY clause is applied after the summary rows are added.
- When you use the ROLLUP operator, you can't use the DISTINCT keyword in any of the aggregate functions.
- You can also use the GROUPING function with the ROLLUP operator to determine if a summary row has a null value assigned to a given column. See chapter 8 for details.

Figure 5-7 How to use the ROLLUP operator

## How to use the CUBE operator

---

Figure 5-8 shows you how to use the CUBE operator. This operator is similar to the ROLLUP operator, except that it adds summary rows for every combination of groups. This is illustrated by the two examples in this figure. As you can see, these examples are the same as the ones in figure 5-7 except that they use the CUBE operator instead of the ROLLUP operator.

In the first example, the result set is grouped by a single column. In this case, a single row is added to the result set that summarizes all the groups. In other words, this works the same as it does with the ROLLUP operator. The only difference is that the summary row is at the start of the result set instead of the end of the result set.

In the second example, you can see how CUBE differs from ROLLUP when you group by two or more columns. In this case, the result set includes a summary row for each state just as it did when the ROLLUP operator was used. For instance, the third row in this example indicates that there are two vendors in the state of Iowa. In addition, though, the CUBE operator includes a summary row for each city. For instance, the ninth row in this example indicates that there are two vendors for the city of Fairfield. But if you look at the first and fifth rows in the result set, you'll see that one of those vendors is in Fairfield, Iowa and one is in Fairfield, New Jersey. Similarly, there are two vendors in two different states for the city named Washington. There are also two vendors in the city of East Brunswick, but both are in New Jersey.

Here again, the ORDER BY clause is applied after the summary rows are added. So in the second example, the ORDER BY clause has two effects. First, it insures that the cities are in sequence within the states. Second, it moves the four extra summary rows that are created by the CUBE operator from the start of the result set to the end of the result set.

Now that you've seen how the CUBE operator works, you may be wondering when you would use it. One obvious use is to add a summary row to a result set that's grouped by a single column, but you could just as easily use the ROLLUP operator for that. Beyond that, the CUBE operator can occasionally get useful information that you can't get any other way.

## A summary query that includes a final summary row

```
SELECT vendor_id, COUNT(*) AS invoice_count,
       SUM(invoice_total) AS invoice_total
  FROM invoices
 GROUP BY CUBE(vendor_id)
```

	VENDOR_ID	INVOICE_COUNT	INVOICE_TOTAL
1	(null)	114	214290.51
2	34	2	1200.12
3	37	3	564
4	48	1	856.92

(35 rows selected)

## A summary query that includes a summary row for each set of groups

```
SELECT vendor_state, vendor_city, COUNT(*) AS qty_vendors
  FROM vendors
 WHERE vendor_state IN ('IA', 'NJ')
 GROUP BY CUBE(vendor_state, vendor_city)
 ORDER BY vendor_state, vendor_city
```

	VENDOR_STATE	VENDOR_CITY	QTY_VENDORS
1	IA	Fairfield	1
2	IA	Washington	1
3	IA	(null)	2
4	NJ	East Brunswick	2
5	NJ	Fairfield	1
6	NJ	Washington	1
7	NJ	(null)	4
8	(null)	East Brunswick	2
9	(null)	Fairfield	2
10	(null)	Washington	2
11	(null)	(null)	6

## Description

- You can use the CUBE operator in the GROUP BY clause to add summary rows to the final result set. A summary is provided for each aggregate column included in the select list. All other columns, except the ones that identify which group is being summarized, are assigned null values.
- The CUBE operator adds a summary row for every combination of groups specified in the GROUP BY clause. It also adds a summary row to the end of the result set that summarizes the entire result set.
- The sort sequence in the ORDER BY clause is applied after the summary rows are added.
- When you use the CUBE operator, you can't use the DISTINCT keyword in any of the aggregate functions.
- You can also use the GROUPING function with the CUBE operator to determine if a summary row has a null value assigned to a given column. See chapter 8 for details.

Figure 5-8 How to use the CUBE operator

## Perspective

---

In this chapter, you learned how to code queries that group and summarize data. In most cases, you'll be able to use the techniques presented here to get the summary information you need.

## Terms

---

scalar function  
aggregate function  
column function  
summary query  
scalar aggregate  
vector aggregate

## Exercises

1. Write a SELECT statement that returns one row for each vendor that contains these columns from the Invoices table:  
The vendor\_id column  
The sum of the invoice\_total column  
The result set should be sorted by vendor\_id.
2. Write a SELECT statement that returns one row for each vendor that contains these columns:  
The vendor\_name column from the Vendors table  
The sum of the payment\_total column in the Invoices table.  
The result set should be sorted in descending sequence by the payment total sum for each vendor.
3. Write a SELECT statement that returns one row for each vendor that contains three columns:  
The vendor\_name column from the Vendors table  
The count of the invoices for each vendor in the Invoices table  
The sum of the invoice\_total column for each vendor in the Invoices table  
Sort the result set so the vendor with the most invoices appears first.

4. Write a SELECT statement that returns one row for each general ledger account number that contains three columns:
  - The account\_description column from the General\_Ledger\_Accounts table
  - The count of the entries in the Invoice\_Line\_Items table that have the same account\_number
  - The sum of the line item amounts in the Invoice\_Line\_Items table that have the same account-numberFilter the result set to include only those rows with a count greater than 1; group the result set by account description; and sort the result set in descending sequence by the sum of the line item amounts.
5. Modify the solution to exercise 4 to filter for invoices dated in the second quarter of 2014 (April 1, 2014 to June 30, 2014). *Hint: Join to the Invoices table to code a search condition based on invoice\_date.*
6. Write a SELECT statement that answers this question: What is the total amount invoiced for each general ledger account number? Use the ROLLUP operator to include a row that gives the grand total. *Hint: Use the line\_item\_amt column of the Invoice\_Line\_Items table.*
7. Write a SELECT statement that answers this question: Which vendors are being paid from more than one account? Return two columns: the vendor name and the total number of accounts that apply to that vendor's invoices. *Hint: Use the DISTINCT keyword to count the account\_number column in the Invoice\_Line\_Items table.*



# How to code subqueries

A subquery is a SELECT statement that's coded within another SQL statement. As a result, you can use subqueries to build queries that would be difficult or impossible to do otherwise. In this chapter, you'll learn how to use subqueries within SELECT statements. Then, in the next chapter, you'll learn how to use them when you code INSERT, UPDATE, and DELETE statements.

<b>An introduction to subqueries .....</b>	<b>182</b>
How to use subqueries .....	182
How subqueries compare to joins.....	184
<b>How to code subqueries in search conditions.....</b>	<b>186</b>
How to use subqueries with the IN operator .....	186
How to compare the result of a subquery with an expression.....	188
How to use the ALL keyword .....	190
How to use the ANY and SOME keywords .....	192
How to code correlated subqueries.....	194
How to use the EXISTS operator .....	196
<b>Other ways to use subqueries .....</b>	<b>198</b>
How to code subqueries in the FROM clause .....	198
How to code subqueries in the SELECT clause.....	200
<b>Guidelines for working with complex queries.....</b>	<b>202</b>
A complex query that uses subqueries .....	202
A procedure for building complex queries.....	204
<b>Two more skills for working with subqueries.....</b>	<b>206</b>
How to code a subquery factoring clause.....	206
How to code a hierarchical query .....	208
<b>Perspective .....</b>	<b>210</b>

## An introduction to subqueries

---

Since you know how to code SELECT statements, you already know how to code a *subquery*. It's simply a SELECT statement that's coded within another SQL statement. The trick to using subqueries is knowing where and when to use them. You'll learn the specifics of using subqueries throughout this chapter. The two topics that follow, however, will give you an overview of where and when to use them.

### How to use subqueries

---

In figure 6-1, you can see that a subquery can be coded, or *introduced*, in the WHERE, HAVING, FROM, or SELECT clause of a SELECT statement. The SELECT statement in this figure, for example, illustrates how you can use a subquery in the search condition of a WHERE clause. When it's used in a search condition, a subquery can be referred to as a *subquery search condition* or a *subquery predicate*.

The statement in this figure retrieves all the invoices from the Invoices table that have invoice totals greater than the average of all the invoices. To do that, the subquery calculates the average of all the invoices. Then, the search condition tests each invoice to see if its invoice total is greater than that average.

When a subquery returns a single value as it does in this example, you can use it anywhere you would normally use an expression. However, a subquery can also return a single-column result set with two or more rows. In that case, it can be used in place of a list of values, such as the list for an IN operator. In addition, if a subquery is coded within a FROM clause, it can return a result set with two or more columns. You'll learn about all of these different types of subqueries in this chapter.

You can also code a subquery within another subquery. In that case, the subqueries are said to be nested. Because *nested subqueries* can be difficult to read, you should use them only when necessary.

## Four ways to introduce a subquery in a SELECT statement

1. In a WHERE clause as a search condition
2. In a HAVING clause as a search condition
3. In the FROM clause as a table specification
4. In the SELECT clause as a column specification

## A SELECT statement that uses a subquery in the WHERE clause

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_total >
    (SELECT AVG(invoice_total)
     FROM invoices)
ORDER BY invoice_total
```

### The value returned by the subquery

1879.7413

### The result set

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 989319-487	18-APR-14	1927.54
2 97/522	30-APR-14	1962.13
3 989319-417	26-APR-14	2051.59
4 989319-427	25-APR-14	2115.81
5 989319-477	19-APR-14	2184.11

(15 rows)

## Description

- A *subquery* is a SELECT statement that's coded within another SQL statement.
- A subquery can return a single value, a result set that contains a single column, or a result set that contains one or more columns.
- A subquery that returns a single value can be coded, or *introduced*, anywhere an expression is allowed. A subquery that returns a single column can be introduced in place of a list of values, such as the values for an IN phrase. And a subquery that returns one or more columns can be introduced in place of a table in the FROM clause.
- The syntax for a subquery is the same as for a standard SELECT statement. However, a subquery doesn't typically include the GROUP BY or HAVING clause.
- A subquery that's used in a WHERE or HAVING clause is called a *subquery search condition* or a *subquery predicate*. This is the most common use for a subquery.
- Although you can introduce a subquery in a GROUP BY or ORDER BY clause, you usually won't need to.
- Subqueries can be *nested* within other subqueries. However, subqueries that are nested more than two or three levels deep can be difficult to read.

## How subqueries compare to joins

---

In the last figure, you saw an example of a subquery that returns an aggregate value that's used in the search condition of a WHERE clause. This type of subquery provides for processing that can't be done any other way. However, most subqueries can be restated as joins, and most joins can be restated as subqueries. This is illustrated by the SELECT statements in figure 6-2.

Both SELECT statements in this figure return a result set that consists of selected rows and columns from the Invoices table. In this case, only the invoices for vendors in California are returned. The first statement uses a join to combine the Vendors and Invoices tables so the vendor\_state column can be tested for each invoice. In contrast, the second statement uses a subquery to return a result set that consists of the vendor\_id column for each vendor in California. Then, that result set is used with the IN operator in the search condition so that only invoices with a vendor\_id in that result set are included in the final result set.

So if you have a choice, which technique should you use? In general, we recommend that you use the technique that results in the most readable code. For example, a join tends to be more intuitive than a subquery when it uses an existing relationship between two tables. That's the case with the Vendors and Invoices tables used in the examples in this figure. On the other hand, a subquery tends to be more intuitive when it uses an ad hoc relationship.

As your queries get more complex, you may find that they're easier to code by using subqueries, regardless of the relationships that are involved. Later in this chapter, for example, you'll see a couple examples of this.

You should also realize that when you use a subquery in a search condition, its results can't be included in the final result set. For instance, the second example in this figure can't be changed to include the vendor\_name column from the Vendors table. That's because the Vendors table isn't named in the FROM clause of the outer query. So if you need to include information from both tables in the result set, you need to use a join.

## A query that uses an inner join

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices JOIN vendors
    ON invoices.vendor_id = vendors.vendor_id
WHERE vendor_state = 'CA'
ORDER BY invoice_date
```

## The same query restated with a subquery

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE vendor_id IN
    (SELECT vendor_id
    FROM vendors
    WHERE vendor_state = 'CA')
ORDER BY invoice_date
```

## The result set returned by both queries

INVOICE_NUMBER	INVOICE_DATE	INVOICE_TOTAL
1 QP58872	25-FEB-14	116.54
2 Q545443	14-MAR-14	1083.58
3 MAB01489	16-APR-14	936.93
4 97/553B	26-APR-14	313.55

(40 rows)

## Advantages of joins

- The result of a join operation can include columns from both tables.
- A join tends to be more intuitive when it uses an existing relationship between the two tables, such as a primary key to foreign key relationship.

## Advantages of subqueries

- You can use a subquery to pass an aggregate value to the outer query.
- A subquery tends to be more intuitive when it uses an ad hoc relationship between the two tables.
- Long, complex queries can sometimes be easier to code using subqueries.

## Description

- Like a join, a subquery can be used to code queries that work with two or more tables.
- Most subqueries can be restated as joins and most joins can be restated as subqueries.

---

Figure 6-2 How subqueries compare to joins

## How to code subqueries in search conditions

---

You can use a variety of techniques to work with a subquery in a search condition. You'll learn about those techniques in the topics that follow. As you read these topics, keep in mind that although all of the examples illustrate the use of subqueries in a WHERE clause, all of this information applies to the HAVING clause as well.

### How to use subqueries with the IN operator

---

In chapter 3, you learned how to use the IN operator to test whether an expression is contained in a list of values. One way to provide that list of values is to use a subquery. This is illustrated in figure 6-3.

The example in this figure retrieves the vendors from the Vendors table that don't have invoices in the Invoices table. To do that, it uses a subquery to retrieve the vendor\_id of each vendor in the Invoices table. The result is a result set like the one shown that contains just the vendor\_id column. Then, this result set is used to filter the vendors that are included in the final result set.

Note that this subquery returns a single column. That's a requirement when a subquery is used with the IN operator. Note also that the subquery includes the DISTINCT keyword. That way, if more than one invoice exists for a vendor, the vendor\_id for that vendor will be included only once. The keyword DISTINCT is optional, however. The final result set will be the same if you include it or omit it.

In the previous figure, you saw that a query that uses a subquery with the IN operator can be restated using an inner join. Similarly, a query that uses a subquery with the NOT IN operator can typically be restated using an outer join. The first query shown in this figure, for example, can be restated as shown in the second query. In this case, though, the query with the subquery is more readable.

## The syntax of a WHERE clause that uses an IN phrase with a subquery

```
WHERE test_expression [NOT] IN (subquery)
```

### A query that returns vendors without invoices

```
SELECT vendor_id, vendor_name, vendor_state
FROM vendors
WHERE vendor_id NOT IN
    (SELECT DISTINCT vendor_id
     FROM invoices)
ORDER BY vendor_id
```

#### The result of the subquery

VENDOR_ID
1
2
3
4
5
6

(34 rows)

#### The result set

VENDOR_ID	VENDOR_NAME	VENDOR_STATE
32	33 Nielson	OH
33	35 Cal State Termite	CA
34	36 Graylift	CA
35	38 Venture Communications Int'l	NY
36	39 Custom Printing Company	MO
37	40 Nat Assoc of College Stores	OH

(88 rows)

### The query restated without a subquery

```
SELECT v.vendor_id, vendor_name, vendor_state
FROM vendors v LEFT JOIN invoices i
    ON v.vendor_id = i.vendor_id
WHERE i.vendor_id IS NULL
ORDER BY v.vendor_id
```

### Description

- You can introduce a subquery with the IN operator to provide the list of values that are tested against the test expression.
- When you use the IN operator, the subquery must return a single column of values.
- A query that uses the NOT IN operator with a subquery can typically be restated using an outer join.

## How to compare the result of a subquery with an expression

---

Figure 6-4 illustrates how you can use the comparison operators to compare an expression with the result of a subquery. In the example, the subquery returns the average balance due of the invoices in the *Invoices* table with a balance due greater than zero. Then, it uses that value to retrieve all invoices that have a balance due that's less than the average.

When you use a comparison operator as shown in this figure, the subquery must return a single value. In most cases, that means that it uses an aggregate function. However, you can also use the comparison operators with subqueries that return two or more values. To do that, you use the *SOME*, *ANY*, or *ALL* keyword to modify the comparison operator. You'll learn more about these keywords in the next two topics.

## The syntax of a WHERE clause that compares an expression with the value returned by a subquery

```
WHERE expression comparison_operator [SOME|ANY|ALL] (subquery)
```

## A query that returns invoices with a balance due less than the average

```
SELECT invoice_number, invoice_date,
       invoice_total - payment_total - credit_total AS balance_due
  FROM invoices
 WHERE invoice_total - payment_total - credit_total > 0
   AND invoice_total - payment_total - credit_total <
      (
        SELECT AVG(invoice_total - payment_total - credit_total)
          FROM invoices
         WHERE invoice_total - payment_total - credit_total > 0
      )
 ORDER BY invoice_total DESC
```

### The value returned by the subquery

1669.906

### The result set

	INVOICE_NUMBER	INVOICE_DATE	BALANCE_DUE
1	31359783	23-MAY-14	1575
2	97/553	27-APR-14	904.14
3	I77271-001	05-JUN-14	662
4	31361833	23-MAY-14	579.42
5	9982771	03-JUN-14	503.2
6	97/553B	26-APR-14	313.55

(33 rows)

## Description

- You can use a comparison operator in a search condition to compare an expression with the results of a subquery.
- If you code a search condition without the ANY, SOME, and ALL keywords, the subquery must return a single value.
- If you include the ANY, SOME, or ALL keyword, the subquery can return a list of values. See figures 6-5 and 6-6 for more information on using these keywords.

Figure 6-4 How to compare the result of a subquery with an expression

## How to use the ALL keyword

---

Figure 6-5 shows you how to use the ALL keyword. This keyword modifies the comparison operator so the condition must be true for all the values returned by a subquery. This is equivalent to coding a series of conditions connected by AND operators. The table at the top of this figure describes how this works for some of the comparison operators.

If you use the greater than operator ( $>$ ), the expression must be greater than the maximum value returned by the subquery. Conversely, if you use the less than operator ( $<$ ), the expression must be less than the minimum value returned by the subquery. If you use the equals operator ( $=$ ), the expression must be equal to all of the values returned by the subquery. And if you use the not equals operator ( $\neq$ ), the expression must not be equal to any of the values returned by the subquery. Note that a not equals condition could be restated using a NOT IN condition.

The query in this figure illustrates the use of the greater than operator with the ALL keyword. Here, the subquery selects the `invoice_total` column for all the invoices with a `vendor_id` value of 34. This results in a table with two rows, as shown in this figure. Then, the outer query retrieves the rows from the `Invoices` table that have invoice totals greater than all of the values returned by the subquery. In other words, this query returns all the invoices that have totals greater than the largest invoice for vendor number 34.

When you use the ALL operator, you should realize that if the subquery doesn't return any rows, the comparison operation will always be true. In contrast, if the subquery returns only null values, the comparison operation will always be false.

In many cases, a condition with the ALL keyword can be rewritten so it's easier to read and maintain. For example, the condition in the query in this figure could be rewritten to use the MAX function like this:

```
WHERE invoice_total >
  (SELECT MAX(invoice_total)
   FROM invoices
   WHERE vendor_id = 34)
```

Whenever you can, then, we recommend that you replace the ALL keyword with an equivalent condition.

## How the ALL keyword works

Condition	Equivalent expression	Description
<code>x &gt; ALL (1, 2)</code>	<code>x &gt; 2</code>	<i>x</i> must be greater than all the values returned by the subquery, which means it must be greater than the maximum value.
<code>x &lt; ALL (1, 2)</code>	<code>x &lt; 1</code>	<i>x</i> must be less than all the values returned by the subquery, which means it must be less than the minimum value.
<code>x = ALL (1, 2)</code>	<code>(x = 1) AND (x = 2)</code>	This condition can evaluate to True only if the subquery returns a single value or if all the values returned by the subquery are the same. Otherwise, it evaluates to False.
<code>x &lt;&gt; ALL (1, 2)</code>	<code>(x &lt;&gt; 1) AND (x &lt;&gt; 2)</code>	This condition is equivalent to: <code>x NOT IN (1, 2)</code>

## A query that returns invoices larger than the largest invoice for vendor 34

```
SELECT vendor_name, invoice_number, invoice_total
FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id
WHERE invoice_total > ALL
    (SELECT invoice_total
     FROM invoices
     WHERE vendor_id = 34)
ORDER BY vendor_name
```

### The result of the subquery

INVOICE_TOTAL
1 116.54
2 1083.58

### The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_TOTAL
1 Bertelsmann Industry Svcs. Inc	509786	6940.25
2 Cahners Publishing Company	587056	2184.5
3 Computerworld	367447	2433
4 Data Reproductions Corp	40318	21842

(25 rows)

## Description

- You can use the ALL keyword to test that a comparison condition is true for all of the values returned by a subquery. This keyword is typically used with the comparison operators `<`, `>`, `<=`, and `>=`.
- If no rows are returned by the subquery, a comparison that uses the ALL keyword is always true.
- If all of the rows returned by the subquery contain a null value, a comparison that uses the ALL keyword is always false.

## How to use the ANY and SOME keywords

---

Figure 6-6 shows how to use the ANY and SOME keywords. You use these keywords to test if a comparison is true for any, or some, of the values returned by a subquery. This is equivalent to coding a series of conditions connected with OR operators. Because these keywords are equivalent, you can use whichever one you prefer. The table at the top of this figure describes how these keywords work with some of the comparison operators.

The example in this figure shows how you can use the ANY keyword with the less than operator. This statement is similar to the one you saw in the previous figure, except that it retrieves invoices with invoice totals that are less than at least one of the invoice totals for a given vendor. Like the statement in the previous figure, this condition could be rewritten using the MAX function, as follows:

```
WHERE invoice_total <
      (SELECT MAX(invoice_total)
       FROM invoices
       WHERE vendor_id = 115)
```

Because you can usually replace an ANY condition with an equivalent condition that's more readable, you probably won't use ANY often.

## How the ANY and SOME keywords work

Condition	Equivalent expression	Description
<code>x &gt; ANY (1, 2)</code>	<code>x &gt; 1</code>	$x$ must be greater than at least one of the values returned by the subquery list, which means that it must be greater than the minimum value returned by the subquery.
<code>x &lt; ANY (1, 2)</code>	<code>x &lt; 2</code>	$x$ must be less than at least one of the values returned by the subquery list, which means that it must be less than the maximum value returned by the subquery.
<code>x = ANY (1, 2)</code>	<code>(x = 1) OR (x = 2)</code>	This condition is equivalent to: <code>x IN (1, 2)</code>
<code>x &lt;&gt; ANY (1, 2)</code>	<code>(x &lt;&gt; 1) OR (x &lt;&gt; 2)</code>	This condition will evaluate to True for any non-empty result set containing at least one non-null value that isn't equal to $x$ .

## A query that returns invoice amounts smaller than the largest invoice amount for vendor 115

```
SELECT vendor_name, invoice_number, invoice_total
FROM vendors JOIN invoices ON vendors.vendor_id = invoices.invoice_id
WHERE invoice_total < ANY
    (SELECT invoice_total
     FROM invoices
     WHERE vendor_id = 115)
```

### The result of the subquery

INVOICE_TOTAL
1
2
3
4

### The result set

VENDOR_NAME	INVOICE_NUMBER	INVOICE_TOTAL
1 Executive Office Products	263253257	22.57
2 Reiter's Scientific & Pro Books	25022117	6
3 Boucher Communications Inc	24863706	6
4 Champion Printing Company	24780512	6
5 Dean Witter Reynolds	111-92R-10094	19.67

(17 rows)

## Description

- You can use the ANY or SOME keyword to test that a condition is true for one or more of the values returned by a subquery. ANY and SOME are equivalent keywords. SOME is the ANSI-standard keyword, but ANY is more commonly used.
- If no rows are returned by the subquery or all of the rows returned by the subquery contain a null value, a comparison that uses the ANY or SOME keyword is always false.

Figure 6-6 How to use the ANY and SOME keywords

## How to code correlated subqueries

The subqueries you've seen so far in this chapter have been subqueries that are executed only once for the entire query. However, you can also code subqueries that are executed once for each row that's processed by the outer query. This type of query is called a *correlated subquery*, and it's similar to using a loop to do repetitive processing in a procedural programming language.

Figure 6-7 illustrates how correlated subqueries work. The example in this figure retrieves rows from the Invoices table for those invoices that have an invoice total that's greater than the average of all the invoices for the same vendor. To do that, the search condition in the WHERE clause of the subquery refers to the vendor\_id value of the current invoice. That way, only the invoices for the current vendor will be included in the average.

Each time a row in the outer query is processed, the value in the vendor\_id column for that row is substituted for the column reference in the subquery. Then, the subquery is executed based on the current value. If the vendor\_id value is 95, for example, this subquery will be executed:

```
SELECT AVG(invoice_total)
  FROM invoices inv_sub
 WHERE inv_sub.vendor_id = 95
```

After this subquery is executed, the value it returns is used to determine whether the current invoice is included in the result set. For example, the value returned by the subquery for vendor 95 is 28.5016. Then, that value is compared with the invoice total of the current invoice. If the invoice total is greater than that value, the invoice is included in the result set. Otherwise, it's not. This process is repeated until each of the invoices in the Invoices table has been processed.

As you study this example, notice how the column names in the WHERE clause of the inner query are qualified to indicate whether they refer to a column in the inner query or the outer query. In this case, the same table is used in both the inner and outer queries, so aliases have been assigned to the tables. Then, those alias names are used to qualify the column names. Although you have to qualify a reference to a column in the outer query, you don't have to qualify a reference to a column in the inner query. However, it's common practice to qualify both names, particularly if they refer to the same table.

Because correlated subqueries can be difficult to code, you may want to test a subquery separately before using it within another SELECT statement. To do that, however, you'll need to substitute a constant value for the variable that refers to a column in the outer query. That's what we did to get the average invoice total for vendor 95. Once you're sure that the subquery works on its own, you can replace the constant value with a reference to the outer query so you can use it within a SELECT statement.

## A query that uses a correlated subquery to return each invoice amount that's higher than the vendor's average invoice amount

```
SELECT vendor_id, invoice_number, invoice_total
FROM invoices inv_main
WHERE invoice_total >
    (SELECT AVG(invoice_total)
     FROM invoices inv_sub
     WHERE inv_sub.vendor_id = inv_main.vendor_id)
ORDER BY vendor_id, invoice_total
```

The value returned by the subquery for vendor 95

28.50166...

### The result set

VENDOR_ID	INVOICE_NUMBER	INVOICE_TOTAL
6	83 31359783	1575
7	95 111-92R-10095	32.7
8	95 111-92R-10093	39.77
9	95 111-92R-10092	46.21
10	110 P-0259	26881.4

(36 rows)

## Description

- A *correlated subquery* is a subquery that is executed once for each row processed by the outer query. In contrast, a *noncorrelated subquery* is executed only once. All of the subqueries you've seen so far have been noncorrelated subqueries.
- A correlated subquery refers to a value that's provided by a column in the outer query. Because that value varies depending on the row that's being processed, each execution of the subquery returns a different result.
- To refer to a value in the outer query, a correlated subquery uses a qualified column name that includes the table name from the outer query. If the subquery uses the same table as the outer query, you must assign a table alias to one of the tables to remove ambiguity.

## Note

- Because a correlated subquery is executed once for each row processed by the outer query, a query with a correlated subquery typically takes longer to run than a query with a noncorrelated subquery.

## How to use the EXISTS operator

---

Figure 6-8 shows you how to use the EXISTS operator with a subquery. This operator tests whether or not the subquery returns a result set. In other words, it tests whether the result set exists. When you use this operator, the subquery doesn't actually return a result set to the outer query. Instead, it returns an indication of whether any rows satisfy the search condition of the subquery. Because of that, queries that use this operator execute quickly.

You typically use the EXISTS operator with a correlated subquery, as illustrated in this figure. This query retrieves all the vendors in the Vendors table that don't have invoices in the Invoices table. Notice that this query returns the same vendors as the two queries you saw in figure 6-3 that use the IN operator with a subquery and an outer join. However, the query in this figure executes more quickly than either of the queries in figure 6-3.

In this example, the correlated subquery selects all invoices that have the same vendor\_id value as the current vendor in the outer query. Because the subquery doesn't actually return a result set, it doesn't matter what columns are included in the SELECT clause. So it's customary to just code an asterisk.

After the subquery is executed, the search condition in the WHERE clause of the outer query uses NOT EXISTS to test whether any invoices were found for the current vendor. If not, the vendor row is included in the result set.

## The syntax of a subquery that uses the EXISTS operator

```
WHERE [NOT] EXISTS (subquery)
```

### A query that returns vendors without invoices

```
SELECT vendor_id, vendor_name, vendor_state
FROM vendors
WHERE NOT EXISTS
    (SELECT *
     FROM invoices
     WHERE invoices.vendor_id = vendors.vendor_id)
```

#### The result set

VENDOR_ID	VENDOR_NAME	VENDOR_STATE
53	33 Nielson	OH
54	35 Cal State Termite	CA
55	36 Graylift	CA
56	38 Venture Communications Int'l	NY
57	39 Custom Printing Company	MO
58	40 Nat Assoc of College Stores	OH

(88 rows)

### Description

- You can use the EXISTS operator to test that one or more rows are returned by the subquery. You can also use the NOT operator along with the EXISTS operator to test that no rows are returned by the subquery.
- When you use the EXISTS operator with a subquery, the subquery doesn't actually return any rows. Instead, it returns an indication of whether any rows meet the specified condition.
- Because no rows are returned by the subquery, it doesn't matter what columns you specify in the SELECT clause. So you typically just code an asterisk (\*).
- Although you can use the EXISTS operator with either a correlated or a noncorrelated subquery, it's used most often with correlated subqueries. That's because it's usually better to use a join than a noncorrelated subquery with EXISTS.

## Other ways to use subqueries

---

Although you'll typically use subqueries in the WHERE or HAVING clause of a SELECT statement, you can also use them in the FROM and SELECT clauses. You'll learn how to do that in the topics that follow.

### How to code subqueries in the FROM clause

---

Figure 6-9 shows you how to code a subquery in a FROM clause. As you can see, you can code a subquery in place of a table specification. In this example, the results of the subquery are joined with another table. When you use a subquery in this way, it can return any number of rows and columns. In the Oracle documentation, this type of subquery is sometimes referred to as an *inline view* since it works like a view that's temporarily created and stored in memory.

Subqueries are typically used in the FROM clause to create inline views that provide summarized data to a summary query. The subquery in this figure, for example, creates an inline view that contains the vendor\_id values and the average invoice totals for all vendors with invoice averages over 4900. To do that, it selects all vendors with invoice averages over 4900, and it groups the invoices by vendor\_id. The inline view is then joined with the Invoices table, and the resulting rows are grouped by vendor\_id. Finally, the maximum invoice date and average invoice total are calculated for the grouped rows, and the results are sorted by the maximum invoice date in descending sequence.

You should notice three things about this query. First, the inline view is assigned a table alias so it can be referred to from the outer query. Second, the result of the AVG function in the subquery is assigned a column alias. This is because an inline view can't have unnamed columns. Third, although you might think that you could use the average invoice totals calculated by the subquery in the select list of the outer query, you can't. That's because the outer query includes a GROUP BY clause, so only aggregate functions, columns named in the GROUP BY clause, and constant values can be included in this list. Because of that, the AVG function is repeated in the select list.

When used in the FROM clause, a subquery is similar to a view. As you learned in chapter 1, a view is a predefined SELECT statement that's saved with the database. Because it's saved with the database, a view typically performs more efficiently than an inline view. However, it isn't always practical to use a view. In those cases, inline views can be quite useful. In addition, inline views can be useful for testing possible solutions before creating a view. Then, once the inline view works the way you want it to, you can define the view based on the subquery you used to create the inline view.

## A query that uses an inline view to retrieve all vendors with an average invoice total over 4900

```

SELECT i.vendor_id, MAX(invoice_date) AS last_invoice_date,
       AVG(invoice_total) AS average_invoice_total
  FROM invoices i JOIN
       (
          SELECT vendor_id, AVG(invoice_total) AS average_invoice_total
            FROM invoices
           HAVING AVG(invoice_total) > 4900
          GROUP BY vendor_id
       ) v
     ON i.vendor_id = v.vendor_id
 GROUP BY i.vendor_id
 ORDER BY MAX(invoice_date) DESC

```

### The result of the subquery (an inline view)

VENDOR_ID	AVERAGE_INVOICE_TOTAL
1	10963.655
2	6940.25
3	7125.34
4	23978.482
5	4901.26

### The result set

VENDOR_ID	LAST_INVOICE_DATE	AVERAGE_INVOICE_TOTAL
1	72 18-JUL-14	10963.655
2	119 04-JUN-14	4901.26
3	104 03-JUN-14	7125.34
4	99 31-MAY-14	6940.25
5	110 08-MAY-14	23978.482

## Description

- A subquery that's coded in the FROM clause returns a result set that can be referred to as an *inline view*. When you create an inline view, you must assign an alias to it. Then, you can use the inline view within the outer query just as you would any other table.
- When you code a subquery in the FROM clause, you must assign names to any calculated values in the result set.
- Inline views are most useful when you need to further summarize the results of a summary query.
- An inline view is like a view in that it retrieves selected rows and columns from one or more base tables. Because views are stored as part of the database, they're typically more efficient to use than inline views. However, it may not always be practical to construct and save a view in advance.

## How to code subqueries in the SELECT clause

---

Figure 6-10 shows you how to use subqueries in the SELECT clause. As you can see, you can use a subquery in place of a column specification. Because of that, the subquery must return a single value.

In most cases, the subqueries you use in the SELECT clause will be correlated subqueries. The subquery in this figure, for example, calculates the maximum invoice date for each vendor in the Vendors table. To do that, it refers to the vendor\_id column from the Invoices table in the outer query.

Because subqueries coded in the SELECT clause are difficult to read, you shouldn't use them unless you can't find another solution. In most cases, though, you can replace the subquery with a join. The first query shown in this figure, for example, could be restated as shown in the second query. This query joins the Vendors and Invoices tables, groups the rows by vendor\_name, and then uses the MAX function to calculate the maximum invoice date for each vendor. As you can see, this query is much easier to read than the one with the subquery.

## A query that uses a correlated subquery in its SELECT clause to retrieve the most recent invoice for each vendor

```
SELECT vendor_name,
       (SELECT MAX(invoice_date) FROM invoices
        WHERE invoices.vendor_id = vendors.vendor_id) AS latest_inv
  FROM vendors
 ORDER BY latest_inv
```

### The result set

VENDOR_NAME	LATEST_INV
1 IBM	14-MAR-14
2 Wang Laboratories, Inc.	16-APR-14
3 Reiter's Scientific & Pro Books	17-APR-14
4 United Parcel Service	26-APR-14
5 Wakefield Co	26-APR-14
6 Zylka Design	01-MAY-14
7 Abbey Office Furnishings	02-MAY-14

(122 rows)

## The same query restated using a join

```
SELECT vendor_name, MAX(invoice_date) AS latest_inv
  FROM vendors v
    LEFT JOIN invoices i ON v.vendor_id = i.vendor_id
 GROUP BY vendor_name
 ORDER BY latest_inv
```

## Description

- When you code a subquery for a column specification in the SELECT clause, the subquery must return a single value.
- A subquery that's coded within a SELECT clause is typically a correlated subquery.
- A query that includes a subquery in its SELECT clause can typically be restated using a join instead of the subquery. Because a join is usually faster and more readable, subqueries are seldom coded in the SELECT clause.

## Guidelines for working with complex queries

---

So far, the examples you've seen of queries that use subqueries have been relatively simple. However, these types of queries can get complicated in a hurry, particularly if the subqueries are nested. Because of that, you'll want to be sure that you plan and test these queries carefully. You'll learn a procedure for doing that in a moment. But first, you'll see a complex query that illustrates the type of query we're talking about.

### A complex query that uses subqueries

---

Figure 6-11 presents a query that uses three subqueries. The first subquery is used in the FROM clause of the outer query to create a result set that contains the state, name, and total invoice amount for each vendor in the Vendors table. The second subquery is also used in the FROM clause of the outer query to create a result set that's joined with the first result set. This result set contains the state and total invoice amount for the vendor in each state that has the largest invoice total. To create this result set, a third subquery is nested within the FROM clause of the subquery. This subquery is identical to the first subquery.

After the two result sets are created, they're joined based on the columns in each table that contain the state and the total invoice amount. The final result set includes the state, name, and total invoice amount for the vendor in each state with the largest invoice total. This result set is sorted by state.

As you can see, this query is quite complicated and difficult to understand. In fact, you might be wondering if there isn't an easier solution to this problem. For example, you might think that you could solve the problem simply by joining the Vendors and Invoices tables and creating a grouped aggregate. If you grouped by vendor state, however, you wouldn't be able to include the name of the vendor in the result set. And if you grouped by vendor state and vendor name, the result set would include all the vendors, not just the vendor from each state with the largest invoice total.

If you think about how else you might solve this query, we think you'll agree that the solution presented here is fairly straightforward. However, in figure 6-13, you'll learn one way to simplify this query. In particular, you'll learn how to code a single Summary subquery instead of coding the Summary1 and Summary2 subqueries shown here.

## A query that uses three subqueries

```

SELECT summary1.vendor_state, summary1.vendor_name,
       top_in_state.sum_of_invoices
FROM
  (
    SELECT v_sub.vendor_state, v_sub.vendor_name,
           SUM(i_sub.invoice_total) AS sum_of_invoices
      FROM invoices i_sub JOIN vendors v_sub
        ON i_sub.vendor_id = v_sub.vendor_id
     GROUP BY v_sub.vendor_state, v_sub.vendor_name
  ) summary1
JOIN
  (
    SELECT summary2.vendor_state,
           MAX(summary2.sum_of_invoices) AS sum_of_invoices
      FROM
        (
          SELECT v_sub.vendor_state, v_sub.vendor_name,
                 SUM(i_sub.invoice_total) AS sum_of_invoices
            FROM invoices i_sub JOIN vendors v_sub
              ON i_sub.vendor_id = v_sub.vendor_id
             GROUP BY v_sub.vendor_state, v_sub.vendor_name
        ) summary2
     GROUP BY summary2.vendor_state
  ) top_in_state
  ON summary1.vendor_state = top_in_state.vendor_state AND
     summary1.sum_of_invoices = top_in_state.sum_of_invoices
ORDER BY summary1.vendor_state

```

### The result set

VENDOR_STATE	VENDOR_NAME	SUM_OF_INVOICES
1 AZ	Wells Fargo Bank	662
2 CA	Digital Dreamworks	7125.34
3 DC	Reiter's Scientific & Pro Books	600
4 MA	Dean Witter Reynolds	1367.5
5 MI	Malloy Lithographing Inc	119892.41
6 NV	United Parcel Service	23177.96
7 OH	Edward Data Services	207.78

(10 rows)

## How the query works

- This query retrieves the vendor from each state that has the largest invoice total. To do that, it uses three subqueries: Summary1, Summary2, and Top\_In\_State. The Summary1 and Top\_In\_State subqueries are joined in the FROM clause of the outer query, and the Summary2 subquery is nested within the FROM clause of the Top\_In\_State subquery.
- The Summary1 and Summary2 subqueries are identical. They join data from the Vendors and Invoices tables and produce a result set that includes the sum of invoices for each vendor grouped by vendor name within state.
- The Top\_In\_State subquery produces a result set that includes the vendor state and the largest sum of invoices for any vendor in that state. This information is retrieved from the results of the Summary2 subquery.

Figure 6-11 A complex query that uses subqueries

## A procedure for building complex queries

---

To build a complex query like the one in the previous figure, you can use a procedure like the one in figure 6-12. To start, you should state the problem to be solved so that you’re clear about what you want the query to accomplish. In this case, the question is, “Which vendor in each state has the largest invoice total?”

Once you’re clear about the problem, you should outline the query using *pseudocode*. Pseudocode is simply code that represents the intent of the query, but doesn’t necessarily use SQL code. The pseudocode shown in this figure, for example, uses part SQL code and part English. Notice that this pseudocode identifies the two main subqueries. Because these subqueries define inline views, the pseudocode also indicates the alias that will be used for each: summary1 and top\_in\_state. That way, you can use these aliases in the pseudocode for the outer query to make it clear where the data it uses comes from.

If it isn’t clear from the pseudocode how each subquery will be coded, or, as in this case, if a subquery is nested within another subquery, you can also write pseudocode for the subqueries. For example, the pseudocode for the top\_in\_state query is presented in this figure. Because this subquery has a subquery nested in its FROM clause, that subquery is identified in this pseudocode as Summary2.

The next step in the procedure is to code and test the actual subqueries to be sure they work the way you want them to. For example, the code for the Summary1 and Summary2 queries is shown in this figure, along with the results of these queries and the results of the top\_in\_state query. Once you’re sure that the subqueries work the way you want them to, you can code and test the final query.

If you follow the procedure presented in this figure, you’ll find it easier to build complex queries that use subqueries. Before you can use this procedure, of course, you need to have a thorough understanding of how subqueries work and what they can do. So you’ll want to be sure to experiment with the techniques you learned in this chapter before you try to build a complex query like the one shown here.

## A procedure for building complex queries

1. State the problem to be solved by the query in English.
2. Use pseudocode to outline the query. The pseudocode should identify the subqueries used by the query and the data they return. It should also include aliases used for any inline views.
3. If necessary, use pseudocode to outline each subquery.
4. Code the subqueries and test them to be sure that they return the correct data.
5. Code and test the final query.

### The problem to be solved by the query in figure 6-11

Which vendor in each state has the largest invoice total?

#### Pseudocode for the query

```

SELECT summary1.vendor_state, summary1.vendor_name,
       top_in_state.sum_of_invoices
  FROM (inline view returning vendor_state, vendor_name, sum_of_invoices)
        AS summary1
     JOIN (inline view returning vendor_state, max(sum_of_invoices))
           AS top_in_state
    ON summary1.vendor_state = top_in_state.vendor_state AND
       summary1.sum_of_invoices = top_in_state.sum_of_invoices
 ORDER BY summary1.vendor_state
  
```

#### Pseudocode for the Top\_In\_State subquery

```

SELECT summary2.vendor_state, MAX(summary2.sum_of_invoices)
  FROM (inline view returning vendor_state, vendor_name, sum_of_invoices)
        AS summary2
 GROUP BY summary2.vendor_state
  
```

#### The code for the Summary1 and Summary2 subqueries

```

SELECT v_sub.vendor_state, v_sub.vendor_name,
       SUM(i_sub.invoice_total) AS sum_of_invoices
  FROM invoices i_sub JOIN vendors v_sub
        ON i_sub.vendor_id = v_sub.vendor_id
 GROUP BY v_sub.vendor_state, v_sub.vendor_name
 ORDER BY v_sub.vendor_state, v_sub.vendor_name
  
```

#### The result of the Summary1 and Summary2 subqueries

VENDOR_STATE	VENDOR_NAME	SUM_OF_INVOICES
1 AZ	Wells Fargo Bank	662
2 CA	Abbey Office Furnishings	17.5
3 CA	Bertelsmann Industry Svcs. Inc	6940.25

(34 rows)

#### The result of the Top\_In\_State subquery

VENDOR_STATE	SUM_OF_INVOICES
1 CA	7125.34
2 MA	1367.5
3 OH	207.78

(10 rows)

Figure 6-12 A procedure for building complex queries

## Two more skills for working with subqueries

---

Now that you've learned how to code complex queries, you're ready to learn two more skills that are closely related to coding subqueries. These skills make it possible to simplify queries that could also be coded with subqueries.

### How to code a subquery factoring clause

---

*Subquery factoring* allows you to name a block of code that contains a SELECT statement. For example, figure 6-13 shows how to use subquery factoring to simplify the complex query presented in figure 6-11. To start, the statement for the query begins with the WITH keyword to identify a subquery factoring clause. Then, it specifies Summary as the name for the first subquery, followed by the AS keyword, followed by an opening parenthesis, followed by a SELECT statement that defines the subquery, followed by a closing parenthesis. In this figure, for example, this statement returns the same result set as the subqueries named Summary1 and Summary2 that were presented in figure 6-11.

After the first subquery is defined, this example continues by defining a second subquery named top\_in\_state. To start, a comma is coded to separate the two subqueries. Then, this code specifies top\_in\_state as the name for the second subquery, followed by the AS keyword, followed by an opening parenthesis, followed by a SELECT statement that defines the subquery, followed by a closing parenthesis. Here, this SELECT statement refers to the first subquery, named Summary. When coding multiple subqueries within a subquery factoring clause, a subquery can refer to any subquery coded before it, but it can't refer to subqueries coded after it. For example, this statement wouldn't work if the two subqueries were coded in the reverse order.

Finally, the SELECT statement that's coded immediately after the two named subqueries uses both of these subqueries, just as if they were tables. To do that, this SELECT statement joins the two subqueries, specifies the columns to retrieve, and specifies the sort order. To avoid ambiguous references, each column is qualified by the name for the subquery.

If you compare figure 6-13 with figure 6-11, we think you'll agree that the code in figure 6-13 is easier to read. That's partly because the tables defined by the subqueries aren't nested within the SELECT statement. In addition, the code in figure 6-13 is easier to maintain because the Summary query is coded in one place, not in two.

When using the syntax shown here to define a subquery factoring clause, you must supply distinct names for all columns defined by the SELECT statement, including calculated values. That way, it's possible for other statements to refer to the columns in the result set. Most of the time, that's all you need to know to be able to work with a subquery factoring clause.

## The syntax of a subquery factoring clause

```
WITH query_name1 AS (query_definition1)
[, query_name2 AS (query_definition2)]
[...]
sql_statement
```

## Two query names and a query that uses them

```
WITH summary AS
(
    SELECT vendor_state, vendor_name, SUM(invoice_total) AS sum_of_invoices
    FROM invoices
        JOIN vendors ON invoices.vendor_id = vendors.vendor_id
    GROUP BY vendor_state, vendor_name
),
top_in_state AS
(
    SELECT vendor_state, MAX(sum_of_invoices) AS sum_of_invoices
    FROM summary
    GROUP BY vendor_state
)
SELECT summary.vendor_state, summary.vendor_name,
    top_in_state.sum_of_invoices
FROM summary JOIN top_in_state
    ON summary.vendor_state = top_in_state.vendor_state AND
        summary.sum_of_invoices = top_in_state.sum_of_invoices
ORDER BY summary.vendor_state
```

## The result set

VENDOR_STATE	VENDOR_NAME	SUM_OF_INVOICES
1 AZ	Wells Fargo Bank	662
2 CA	Digital Dreamworks	7125.34
3 DC	Reiter's Scientific & Pro Books	600
4 MA	Dean Witter Reynolds	1367.5
5 MI	Malloy Lithographing Inc	119892.41
6 NV	United Parcel Service	23177.96
7 OH	Edward Data Services	207.78

(10 rows)

## Description

- A *subquery factoring clause* can be thought of as a named subquery block. This name can then be used multiple times in the query.
- To define a subquery factoring block, you code the WITH keyword followed by the definition of the subquery.
- To code multiple subquery factoring clauses, separate them with commas. Then, each clause can refer to itself and any previously defined subquery factoring clauses in the same WITH clause.
- You can use subquery factoring clauses with SELECT, INSERT, UPDATE, and DELETE statements. However, you're most likely to use them with SELECT statements, as shown in this figure.

---

Figure 6-13 How to code a subquery factoring clause

## How to code a hierarchical query

---

A *hierarchical query* loops through a result set and returns rows in a hierarchical sequence. Hierarchical queries are often used to work with organizational charts and other hierarchies, in which a parent element may have one or more child elements, and each child element may have one or more child elements. In figure 6-14, for example, the query shows the hierarchical levels for the employees within a company.

The Employees table uses the manager\_id column to identify the manager for each employee. Here, Cindy Smith is the top-level manager since she doesn't have a manager, Elmer Jones and Paulo Locario report to Cindy Smith, and so on.

In this figure, the hierarchical query uses the LEVEL pseudo column to return a column that identifies the level of the employee within the hierarchy. To start, this query uses the LEVEL pseudo column in the SELECT clause to identify the third column. In addition, this query uses the LEVEL pseudo column in the ORDER BY clause to sort by this column.

After the FROM clause, this query uses the START WITH clause to identify the row to be used as the root of the hierarchy. In this figure, for example, the query uses the employee\_id column to identify Cindy Smith as the root of the hierarchy. However, if you wanted, you could use a difference expression after the START WITH keywords to identify this row or a different row.

Finally, the CONNECT BY clause specifies the condition that identifies the relationship between parent rows and child rows. In this figure, for example, the CONNECT BY clause is followed by the PRIOR keyword to specify that a row is a child row if the row's employee\_id column equals the manager\_id column of the other row.

## A query that returns hierarchical data

```
SELECT select_list
FROM table_source
[WHERE search_condition]
START WITH row_specification
CONNECT BY PRIOR connect_expression
[ORDER BY order_by_list]
```

### The Employees table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_NUMBER
1	Smith	Cindy	2
2	Jones	Elmer	4
3	Simonian	Ralph	2
4	Hernandez	Olivia	1
5	Aaronsen	Robert	2
6	Watson	Denise	6
7	Hardy	Thomas	5
8	O'Leary	Rhea	4
9	Locario	Paulo	6

## A query that returns hierarchical data

```
SELECT employee_id,
       first_name || ' ' || last_name AS employee_name,
       LEVEL
  FROM employees
 START WITH employee_id = 1
 CONNECT BY PRIOR employee_id = manager_id
 ORDER BY LEVEL, employee_id
```

### The result set

EMPLOYEE_ID	EMPLOYEE_NAME	LEVEL
1	1 Cindy Smith	1
2	2 Elmer Jones	2
3	9 Paulo Locario	2
4	3 Ralph Simonian	3
5	4 Olivia Hernandez	3
6	7 Thomas Hardy	3
7	8 Rhea O'Leary	3
8	5 Robert Aaronsen	4
9	6 Denise Watson	4

## Description

- A *hierarchical query* is a query that returns rows in a hierarchical order.
- You can use the LEVEL pseudocolumn to identify the level for each row.
- You can use the START WITH clause to identify the row to be used as the root of the hierarchical query.
- You can use the CONNECT BY clause followed by the PRIOR keyword to specify a condition that identifies the relationship between parent rows and child rows.

Figure 6-14 How to code a hierarchical query

## Perspective

---

As you've seen in this chapter, subqueries provide a powerful tool for solving difficult problems. Before you use a subquery, however, remember that a subquery can often be restated more clearly by using a join. If so, you'll typically want to use a join instead of a subquery.

If you find yourself coding the same subqueries over and over, you should consider creating a view for that subquery, as described in chapter 11. This will help you develop queries more quickly since you can use the view instead of coding the subquery again. In addition, since views typically execute more quickly than subqueries, this may improve the performance of your queries.

## Terms

---

- subquery
- introduce a subquery
- subquery search condition
- subquery predicate
- nested subquery
- correlated subquery
- inline view
- pseudocode
- subquery factoring
- hierarchical query

## Exercises

1. Write a SELECT statement that returns the same result set as this SELECT statement but don't use a join. Instead, use a subquery in a WHERE clause that uses the IN keyword.

```
SELECT DISTINCT vendor_name
  FROM vendors JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
   ORDER BY vendor_name
```

2. Write a SELECT statement that answers this question: Which invoices have a payment\_total that's greater than the average payment\_total for all paid invoices? Return the invoice\_number and invoice\_total for each invoice.
3. Write a SELECT statement that returns two columns from the General\_Ledger\_Accounts table: account\_number and account\_description. The result set should have one row for each account number that has never been used. Use a subquery introduced with the NOT EXISTS operator, and sort the final result set by account\_number.
4. Write a SELECT statement that returns four columns: vendor\_name, invoice\_id, invoice\_sequence, and line\_item\_amt for each invoice that has more than one line item in the Invoice\_Line\_Items table. *Hint: Use a subquery that tests for invoice\_sequence > 1.*
5. Write a SELECT statement that returns a single value that represents the sum of the largest unpaid invoices for each vendor (just one for each vendor). Use an inline view that returns MAX(invoice\_total) grouped by vendor\_id, filtering for invoices with a balance due.
6. Rewrite exercise 5 so it uses subquery factoring.
7. Write a SELECT statement that returns the name, city, and state of each vendor that's located in a unique city and state. In other words, don't include vendors that have a city and state in common with another vendor.
8. Use a correlated subquery to return one row per vendor, representing the vendor's oldest invoice (the one with the earliest date). Each row should include these four columns: vendor\_name, invoice\_number, invoice\_date, and invoice\_total.
9. Rewrite exercise 8 so it gets the same result but doesn't use a correlated subquery.



# How to insert, update, and delete data

In the last four chapters, you learned how to code the SELECT statement to retrieve and summarize data. Now, you'll learn how to code the INSERT, UPDATE, and DELETE statements to modify the data in a table. When you're done with this chapter, you'll know how to code the four statements that are used every day by professional application developers.

<b>How to create test tables .....</b>	<b>214</b>
How to create the tables for this book .....	214
How to create a copy of a table.....	214
<b>How to commit and rollback changes.....</b>	<b>216</b>
How to commit changes .....	216
How to rollback changes.....	216
<b>How to insert new rows .....</b>	<b>218</b>
How to insert a single row .....	218
How to insert default values and null values.....	220
How to use a subquery to insert multiple rows.....	222
<b>How to update existing rows .....</b>	<b>224</b>
How to update rows .....	224
How to use a subquery in an UPDATE statement.....	226
<b>How to delete existing rows .....</b>	<b>228</b>
How to delete rows.....	228
How to use a subquery in a DELETE statement.....	228
<b>Perspective .....</b>	<b>230</b>

## How to create test tables

---

As you practice coding INSERT, UPDATE, and DELETE statements, you need to make sure that your experimentation won't affect "live" data that's used by other people at your business or school. Two ways to get around that are presented next.

### How to create the tables for this book

---

The first procedure in figure A-5 of appendix A shows how to create the tables that are used for the examples in this book. After you create these tables, you can modify them without worrying about how much you change them. If you ever want to restore the tables to their original data, you can use the second procedure in figure A-5.

### How to create a copy of a table

---

If you want to test INSERT, UPDATE, and DELETE statements on tables that are running on a server that's available from your business or school, you can create a copy of some or all of a table before you do any testing. To do that, you can use the CREATE TABLE statement with an embedded SELECT statement as shown in figure 7-1. Then, you can experiment all you want with the test tables and delete them when you're done. When you use this technique, the result set that's defined by the SELECT statement is simply copied into a new table.

The three examples in this figure show some of the ways you can use this statement. Here, the first example copies all of the columns from all of the rows in the Invoices table into a new table named *Invoices\_Copy*. The second example copies all of the columns in the Invoices table into a new table named *Old\_Invoices*, but only for rows where the balance due is zero. And the third example creates a table that contains summary data from the Invoices table.

When you're done experimenting with test tables, you can use the DROP TABLE statement that's shown in this figure to delete any tables you don't need anymore. In this figure, for instance, the fourth example shows how to drop the *Invoices\_Copy* table.

When you use this technique to create tables, though, only the column definitions and data are copied, which means that definitions like those of primary keys, foreign keys, and default values aren't retained. As a result, the results that you get when you test against copied tables may be slightly different than the results you would get with the original tables. You'll understand that better after you read chapters 9 and 10.

## The syntax of the CREATE TABLE AS statement

```
CREATE TABLE table_name AS  
SELECT select_list  
FROM table_source  
[WHERE search_condition]  
[GROUP BY group_by_list]  
[HAVING search_condition]  
[ORDER BY order_by_list]
```

### A statement that creates a complete copy of the Invoices table

```
CREATE TABLE invoices_copy AS  
SELECT *  
FROM invoices
```

### A statement that creates a partial copy of the Invoices table

```
CREATE TABLE old_invoices AS  
SELECT *  
FROM invoices  
WHERE invoice_total - payment_total - credit_total = 0
```

### A statement that creates a table with summary rows from the Invoices table

```
CREATE TABLE vendor_balances AS  
SELECT vendor_id, SUM(invoice_total) AS sum_of_invoices  
FROM invoices  
WHERE (invoice_total - payment_total - credit_total) <> 0  
GROUP BY vendor_id
```

### A statement that deletes a table

```
DROP TABLE old_invoices
```

## Description

- You can create a new table based on the result set defined by the SELECT statement. Since the definitions of the columns in the new table are based on the columns in the result set, the column names assigned in the SELECT clause must be unique.
- You can code the other clauses of the SELECT statement just as you would for any other SELECT statement, including grouping, aggregates, joins, and subqueries.
- If you use calculated values in the select list, you must name the column since that name is used in the definition of the new table.
- The table you name must not exist. If it does, you must delete the table by using the DROP TABLE statement before you execute the SELECT statement.

## Warning

- When you use the SELECT statement to create a table, only the column definitions and data are copied. Definitions of primary keys, foreign keys, indexes, default values, and so on are not included in the new table.

## How to commit and rollback changes

---

When you use SQL Developer to execute INSERT, UPDATE, and DELETE statements, Oracle Database automatically adds those statements to a *transaction*. A transaction is a group of SQL statements that must all be executed successfully before they are saved to the database. To make the changes to the database permanent, you must explicitly *commit* the changes to the database. Otherwise, you can undo, or *rollback*, the changes.

For example, the INSERT statement in figure 7-2 adds a single row to the Invoices table. If you use SQL Developer to execute this statement, the row will be added to the Invoices table. Then, you will be able to see this row if you execute a SELECT statement that selects this row from the Invoices table. However, any other users who may be using this database won't be able to see this row until you commit this change to the database.

When you exit SQL Developer, the Oracle Database will automatically rollback all INSERT, UPDATE, and DELETE statements that haven't explicitly been committed. In other words, if you don't explicitly commit changes, they will be lost when you exit SQL Developer.

When you're practicing with INSERT, UPDATE, and DELETE statements, you may want them rolled back. If you want to make permanent changes to a production system, though, you'll need to make sure to commit the changes. For more information about working with transactions, see chapter 14.

### How to commit changes

---

You can make the changes to the database permanent by executing the COMMIT statement shown in this figure. Or, if you're using SQL Developer, you can commit the changes by clicking on the Commit button or pressing F11. When you do, SQL Developer will display a message that indicates whether the commit succeeded.

Most of the time, you'll want to use one of these techniques to manually commit your changes. However, if you want SQL Developer to automatically commit changes immediately after they are made, you can enable the "Autocommit in SQL Worksheet" feature as described in this figure.

### How to rollback changes

---

You can rollback any INSERT, UPDATE, and DELETE statements that haven't been committed yet by executing the ROLLBACK statement shown in this figure. Or, if you're using SQL Developer, you can undo the changes by clicking on the Rollback button or pressing F12. When you do, SQL Developer will display a message that indicates whether the rollback succeeded.

## An INSERT statement that adds a new row to the Invoices table

```
INSERT INTO invoices  
VALUES (115, 97, '456789', '01-AUG-14', 8344.50, 0, 0, 1, '31-AUG-14', NULL)
```

### The response from the system

```
1 rows inserted
```

## A COMMIT statement that commits the changes

```
COMMIT
```

### The response from the system

```
committed
```

## A ROLLBACK statement that rolls back the changes

```
ROLLBACK
```

### The response from the system

```
rollback complete
```

## Description

- A *transaction* is a group of SQL statements that must all be executed together. By default, Oracle adds INSERT, UPDATE, and DELETE statements to a transaction.
- To *commit* the changes made by a transaction, you can issue the COMMIT statement. Or, if you're using SQL Developer, you can click on the Commit button or press F11.
- To *rollback* the changes made by a transaction, you can issue the ROLLBACK statement. Or, if you're using SQL Developer, you can click on the Rollback button or press F12.
- By default, if you don't explicitly commit the changes made to the database by a transaction, the Oracle Database will rollback the changes when you exit SQL Developer.
- If you want SQL Developer to automatically commit changes to the database immediately after each INSERT, UPDATE, or DELETE statement is executed, use the Tools→Preferences command. Then, expand the Database node, click on the Worksheet Parameters node, and check the “Autocommit in SQL Worksheet” check box.

## How to insert new rows

---

To add new rows to a table, you use the INSERT statement. This statement lets you insert a single row with the values you specify or selected rows from another table. You'll see how to use both forms of the INSERT statement in the topics that follow. In addition, you'll learn how to work with default values and null values when you insert new rows.

### How to insert a single row

---

Figure 7-3 shows how to code an INSERT statement to insert a single row. The two examples in this figure insert a row into the *Invoices* table. The data this new row contains is defined near the top of this figure.

In the first example, you can see that you name the table in which the row will be inserted in the INSERT clause. Then, the VALUES clause lists the values to be used for each column. You should notice three things about this list. First, it includes a value for every column in the table. Second, the values are listed in the same sequence that the columns appear in the table. That way, Oracle knows which value to assign to which column. And third, a null value is assigned to the last column, *payment\_date*, using the NULL keyword. You'll learn more about using this keyword in the next topic.

The second INSERT statement in this figure includes a column list in the INSERT clause. Notice that this list doesn't include the *payment\_date* column since it allows a null value. In addition, the columns aren't listed in the same sequence as the columns in the *Invoices* table. When you include a list of columns, you can code the columns in any sequence you like. Then, you just need to be sure that the values in the VALUES clause are coded in the same sequence.

When you specify the values for the columns to be inserted, you must be sure that those values are compatible with the data types of the columns. For example, you must enclose literal values for dates and strings within single quotes. However, you don't need to enclose literal values for numbers in single quotes. You'll learn more about data types and how to work with them in the next chapter. For now, just realize that if any of the values aren't compatible with the data types of the corresponding columns, an error will occur and the row won't be inserted.

## The syntax of the INSERT statement for inserting a single row

```
INSERT INTO table_name [(column_list)]
VALUES (expression_1 [, expression_2]...)
```

### The values for a new row to be added to the Invoices table

Column	Value	Column	Value
invoice_id	115	payment_total	0
vendor_id	97	credit_total	0
invoice_number	456789	terms_id	1
invoice_date	8/01/2014	invoice_due_date	8/31/2014
invoice_total	8,344.50	payment_date	null

### An INSERT statement that adds the new row without using a column list

```
INSERT INTO invoices
VALUES (115, 97, '456789', '01-AUG-14', 8344.50, 0, 0, 1, '31-AUG-14',
NULL)
(1 rows inserted)
```

### An INSERT statement that adds the new row using a column list

```
INSERT INTO invoices
  (invoice_id, vendor_id, invoice_number, invoice_total, payment_total,
   credit_total, terms_id, invoice_date, invoice_due_date)
VALUES
  (115, 97, '456789', 8344.50, 0, 0, 1, '01-AUG-14', '31-AUG-14')
(1 rows inserted)
```

## Description

- You use the INSERT statement to add a new row to a table.
- In the INSERT clause, you specify the name of the table that you want to add a row to, along with an optional column list. The INTO keyword is required.
- You specify the values to be inserted in the VALUES clause. The values you specify depend on whether you include a column list.
- If you don't include a column list, you must specify the column values in the same order as they appear in the table, and you must code a value for each column in the table.
- If you include a column list, you must specify the column values in the same order as they appear in the column list. You can omit columns with default values and columns that accept null values.
- To insert a null value into a column, you can use the NULL keyword. To insert a default value, you can use the DEFAULT keyword. See figure 7-4 for more information on using these keywords.

## How to insert default values and null values

---

If a column allows null values, you'll want to know how to insert a null value into that column. Similarly, if a column is defined with a default value, you'll want to know how to insert that value. The technique you use depends on whether the INSERT statement includes a column list, as shown by the examples in figure 7-4.

All of these INSERT statements use a table named Color\_Sample. This table contains the three columns shown at the top of this figure. The first column, color\_id, represents the internal ID for the column. The second column, color\_number, is defined with a default value of 0. And the third column, color\_name, is defined so that it allows null values.

The first two statements illustrate how you assign a default value or a null value using a column list. To do that, you simply omit the column from the list. In the first statement, for example, the column list names only the color\_number column, so the color\_name column is assigned a null value. Similarly, the column list in the second statement names only the color\_name column, so the color\_number is assigned its default value.

The next three statements show how you assign a default or null value to a column without including a column list. As you can see, you do that by using the DEFAULT and NULL keywords. For example, the third statement specifies a value for the color\_name column, but uses the DEFAULT keyword for the color\_number column. Because of that, Oracle will assign a value of zero to this column. The fourth statement assigns a value of 808 to the color\_number column, and it uses the NULL keyword to assign a null value to the color\_name column. Finally, the fifth statement uses both the DEFAULT and NULL keywords to assign a value of zero to the color\_number column and a null value to the color\_name column.

## The definition of the Color\_Sample table

Column name	Data Type	Not Null	Default Value
color_id	NUMBER	Yes	
color_number	NUMBER	Yes	0
color_name	VARCHAR2		

## Five INSERT statements for the Color\_Sample table

```
INSERT INTO color_sample (color_id, color_number)
VALUES (1, 606)
```

```
INSERT INTO color_sample (color_id, color_name)
VALUES (2, 'Yellow')
```

```
INSERT INTO color_sample
VALUES (3, DEFAULT, 'Orange')
```

```
INSERT INTO color_sample
VALUES (4, 808, NULL)
```

```
INSERT INTO color_sample
VALUES (5, DEFAULT, NULL)
```

## The Color\_Sample table after the rows are inserted

COLOR_ID	COLOR_NUMBER	COLOR_NAME
1	606 (null)	
2	0 Yellow	
3	0 Orange	
4	808 (null)	
5	0 (null)	

## Description

- If a column is defined so it allows null values, you can use the NULL keyword in the list of values to insert a null value into that column.
- If a column is defined with a default value, you can use the DEFAULT keyword in the list of values to insert the default value for that column.
- If you include a column list, you can omit columns with default values and null values. Then, the default value or null value is assigned automatically.

## How to use a subquery to insert multiple rows

---

Instead of using the VALUES clause of the INSERT statement to specify the values for a single row, you can use a subquery to select the rows you want to insert from another table. Figure 7-5 shows you how to do that.

Both examples in this figure retrieve rows from the Invoices table and insert them into a table named Invoice\_Archive. This table is defined with the same columns as the Invoices table. However, the payment\_total and credit\_total columns aren't defined with default values. Because of that, you must include values for these columns.

The first example in this figure shows how you can use a subquery in an INSERT statement without coding a column list. In this example, the SELECT clause of the subquery is coded with an asterisk so that all the columns in the Invoices table will be retrieved. Then, after the search condition in the WHERE clause is applied, all the rows in the result set are inserted into the Invoice\_Archive table.

The second example shows how you can use a column list in the INSERT clause when you use a subquery to retrieve rows. Just as when you use the VALUES clause, you can list the columns in any sequence. However, the columns must be listed in the same sequence in the SELECT clause of the subquery. In addition, you can omit columns that are defined with default values or that allow null values.

Notice that the subqueries in these statements aren't coded within parentheses as a subquery in a SELECT statement is. That's because they're not coded within a clause of the INSERT statement. Instead, they're coded in place of the VALUES clause.

Before you execute INSERT statements like the ones shown in this figure, you'll want to be sure that the rows and columns retrieved by the subquery are the ones you want to insert. To do that, you can execute the SELECT statement by itself. Then, when you're sure it retrieves the correct data, you can add the INSERT clause to insert the rows into another table.

## The syntax of the INSERT statement for inserting rows selected from another table

```
INSERT [INTO] table_name [(column_list)]
SELECT column_list
FROM table_source
[WHERE search_condition]
```

## An INSERT statement that inserts paid invoices in the Invoices table into the Invoice\_Archive table

```
INSERT INTO invoice_archive
SELECT *
FROM invoices
WHERE invoice_total - payment_total - credit_total = 0
(74 rows inserted)
```

## The same INSERT statement with a column list

```
INSERT INTO invoice_archive
    (invoice_id, vendor_id, invoice_number, invoice_total, credit_total,
     payment_total, terms_id, invoice_date, invoice_due_date)
SELECT
    invoice_id, vendor_id, invoice_number, invoice_total, credit_total,
    payment_total, terms_id, invoice_date, invoice_due_date
FROM invoices
WHERE invoice_total - payment_total - credit_total = 0
(74 rows inserted)
```

## Description

- To insert rows selected from one or more tables into another table, you can code a subquery in place of the VALUES clause. Then, the rows in the derived table that result from the subquery are inserted into the table.
- If you don't code a column list in the INSERT clause, the subquery must return values for all the columns in the table where the rows will be inserted, and the columns must be returned in the same order as they appear in that table.
- If you include a column list in the INSERT clause, the subquery must return values for those columns in the same order as they appear in the column list. You can omit columns with default values and columns that accept null values. However, it is good programming practice to always include a column list.

## How to update existing rows

---

To modify the data in one or more rows of a table, you use the UPDATE statement. Although most of the UPDATE statements you code will perform simple updates, you can also code more complex UPDATE statements that include subqueries.

### How to update rows

---

Figure 7-6 presents the syntax of the UPDATE statement. As you can see in the examples, most UPDATE statements include just the UPDATE, SET, and WHERE clauses. The UPDATE clause names the table to be updated, the SET clause names the columns to be updated and the values to be assigned to those columns, and the WHERE clause specifies the condition a row must meet to be updated. Although the WHERE clause is optional, you'll almost always include it. If you don't, all of the rows in the table will be updated, which usually isn't what you want.

The first UPDATE statement in this figure modifies the values of two columns in the Invoices table: payment\_date and payment\_total. Because the WHERE clause in this statement identifies a specific invoice number, only the columns in that invoice will be updated. Notice in this example that the value to be assigned to payment\_date is coded as a literal. You should realize, however, that you can assign any valid expression to a column as long as it evaluates to a value that's compatible with the data type of the column. You can also use the NULL keyword to assign a null value to a column that allows nulls, and you can use the DEFAULT keyword to assign the default value to a column that's defined with one.

The second UPDATE statement modifies a single column in the Invoices table: terms\_id. This time, however, the WHERE clause specifies that all the rows for vendor 95 should be updated. Because this vendor has six rows in the Invoices table, all six rows will be updated.

The third UPDATE statement illustrates how you can use an expression to assign a value to a column. In this case, the expression increases the value of the credit\_total column by 100. Like the first UPDATE statement, this statement updates a single row.

Before you execute an UPDATE statement, you'll want to be sure that you've selected the correct rows. To do that, you can execute a SELECT statement with the same search condition. Then, if the SELECT statement returns the correct rows, you can change it to an UPDATE statement.

## The syntax of the UPDATE statement

```
UPDATE table_name  
SET column_name_1 = expression_1 [, column_name_2 = expression_2]...  
[WHERE search_condition]
```

### An UPDATE statement that assigns new values to two columns of a single row in the Invoices table

```
UPDATE invoices  
SET payment_date = '21-SEP-14',  
    payment_total = 19351.18  
WHERE invoice_number = '97/522'  
  
(1 rows updated)
```

### An UPDATE statement that assigns a new value to one column of all invoices for a vendor

```
UPDATE invoices  
SET terms_id = 1  
WHERE vendor_id = 95  
  
(6 rows updated)
```

### An UPDATE statement that uses an arithmetic expression to assign a value to a column

```
UPDATE invoices  
SET credit_total = credit_total + 100  
WHERE invoice_number = '97/522'  
  
(1 rows updated)
```

## Description

- You use the UPDATE statement to modify one or more rows in the table named in the UPDATE clause.
- You name the columns to be modified and the value to be assigned to each column in the SET clause. You can specify the value for a column as a literal or an expression.
- You can specify the conditions that must be met for a row to be updated in the WHERE clause.
- You can use the DEFAULT keyword to assign the default value to a column that has one, and you can use the NULL keyword to assign a null value to a column that allows nulls. For more information, see figure 7-4.

## Warning

- If you omit the WHERE clause, all rows in the table will be updated.

## How to use a subquery in an UPDATE statement

---

Figure 7-7 presents three UPDATE statements that illustrate how you can use subqueries in an update operation. In the first statement, a subquery is used in the SET clause to retrieve the maximum invoice due date from the Invoices table. Then, that value is assigned to the invoice\_due\_date column for invoice number 97/522.

In the second statement, a subquery is used in the WHERE clause to identify the invoices to be updated. This subquery returns the vendor\_id value for the vendor in the Vendors table with the name “Pacific Bell.” Then, all the invoices with that vendor\_id value are updated.

The third UPDATE statement also uses a subquery in the WHERE clause. This subquery returns a list of the vendor\_id values for all vendors in California, Arizona, and Nevada. Then, the IN operator is used to update all the invoices with vendor\_id values in that list. Note that although the subquery returns 80 vendors, many of these vendors don’t have invoices. As a result, the UPDATE statement only affects 51 invoices.

### An UPDATE statement that assigns the maximum due date in the Invoices table to a specific invoice

```
UPDATE invoices
SET credit_total = credit_total + 100,
    invoice_due_date = (SELECT MAX(invoice_due_date) FROM invoices)
WHERE invoice_number = '97/522'

(1 rows updated)
```

### An UPDATE statement that updates all invoices for a vendor based on the vendor's name

```
UPDATE invoices
SET terms_id = 1
WHERE vendor_id =
    (SELECT vendor_id
     FROM vendors
     WHERE vendor_name = 'Pacific Bell')

(6 rows updated)
```

### An UPDATE statement that changes the terms of all invoices for vendors in three states

```
UPDATE invoices
SET terms_id = 1
WHERE vendor_id IN
    (SELECT vendor_id
     FROM vendors
     WHERE vendor_state IN ('CA', 'AZ', 'NV'))

(51 rows updated)
```

### Description

- You can code a subquery in the SET or WHERE clause of an UPDATE statement.
- You can use a subquery in the SET clause to return the value that's assigned to a column.
- You can code a subquery in the WHERE clause to provide one or more values used in the search condition.

## How to delete existing rows

---

To delete one or more rows from a table, you use the DELETE statement. Just as you can with the UPDATE statement, you can use subqueries in a DELETE statement to help identify the rows to be deleted.

### How to delete rows

---

Figure 7-8 presents the syntax of the DELETE statement along with three examples that illustrate some basic delete operations. As you can see, you specify the name of the table that contains the rows to be deleted in the DELETE clause. You can also code the FROM keyword in this clause, but this keyword is optional so it can be omitted.

To identify the rows to be deleted, you code a search condition in the WHERE clause. Although this clause is optional, you'll almost always include it. If you don't, all of the rows in the table are deleted. This is a common coding mistake, and it can be disastrous if you're working with live data.

If you want to make sure that you've selected the correct rows before you issue the DELETE statement, you can issue a SELECT statement with the same search condition. Then, if the correct rows are retrieved, you can use the same search for the DELETE statement.

The first DELETE statement in this figure deletes a single row from the Invoice\_Line\_Items table. To do that, it specifies the invoice\_id value of the row to be deleted and the invoice\_sequence value in the WHERE clause.

The second DELETE statement deletes four rows from the Invoice\_Line\_Items table. To do that, it specifies 100 as the invoice\_id value of the row to be deleted in the WHERE clause. Since the invoice for this ID has four line items, this deletes all four line items for the invoice.

If you try to delete a row that has one or more child rows that are defined with a foreign-key constraint, Oracle will return an error message and won't delete the row. For example, if you attempt to delete a row from the Vendors table that has child rows in the Invoices and Invoice\_Line\_Items tables, Oracle will return an error message that indicates that an integrity constraint was violated, and it won't delete the vendor. Usually, that's what you want.

### How to use a subquery in a DELETE statement

---

If you really want to delete a row that has one or more child rows from the Vendors table, you can start by deleting all of the invoices and line items for that vendor. To do that, you can use a statement like the third one in this figure to delete the line items for the vendor from the Invoice\_Line\_Items table. Note how this statement uses a subquery to delete all line items for the vendor with the ID of 115. Then, you can use a similar statement to delete all of the invoices for that vendor from the Invoices table. Finally, you can use a simple DELETE statement to delete the row for the vendor from the Vendors table.

## The syntax of the DELETE statement

```
DELETE [FROM] table_name  
[WHERE search_condition]
```

### A DELETE statement that deletes one row

```
DELETE FROM invoice_line_items  
WHERE invoice_id = 100 AND invoice_sequence = 1  
(1 rows deleted)
```

### A DELETE statement that deletes four rows

```
DELETE FROM invoice_line_items  
WHERE invoice_id = 100  
(4 rows deleted)
```

### A DELETE statement that uses a subquery to delete all invoice line items for a vendor

```
DELETE FROM invoice_line_items  
WHERE invoice_id IN  
(SELECT invoice_id  
FROM invoices  
WHERE vendor_id = 115)  
(4 rows deleted)
```

## Description

- You can use the DELETE statement to delete one or more rows from the table you name in the DELETE clause.
- You specify the conditions that must be met for a row to be deleted in the WHERE clause.
- You can use a subquery within the WHERE clause.
- A foreign-key constraint may prevent you from deleting a row. In that case, you can only delete the row if you delete all child rows for that row first.

## Warning

- If you omit the WHERE clause from a DELETE statement, all the rows in the table will be deleted.

## Perspective

---

In this chapter, you learned how to use the INSERT, UPDATE, and DELETE statements to modify the data in a database. In chapters 9 and 10, you'll learn more about the table definitions that can affect the way these statements work. And in chapter 14, you'll learn more about executing groups of INSERT, UPDATE, and DELETE statements as a single transaction.

## Terms

---

transaction  
rollback  
commit

## Exercises

To test whether a table has been modified correctly as you do these exercises, you can write and run an appropriate SELECT statement. Or, when you're using Oracle SQL Developer, you can click on a table name in the Connections window and then on the Data tab to display the data for all of the columns in the table. To refresh the data on this tab, click the Refresh button.

1. Write an INSERT statement that adds this row to the *Invoices* table:

invoice_id	The next id in sequence (find out what this should be)
vendor_id:	32
invoice_number:	AX-014-027
invoice_date:	8/1/2014
invoice_total:	\$434.58
payment_total:	\$0.00
credit_total:	\$0.00
terms_id:	2
invoice_due_date:	8/31/2014
payment_date:	null

2. Write an UPDATE statement that modifies the *Vendors* table. Change the default account number to 403 for each vendor that has a default account number of 400.
3. Write an UPDATE statement that modifies the *Invoices* table. Change the terms\_id to 2 for each invoice that's for a vendor with a default\_terms\_id of 2.
4. Write a DELETE statement that deletes the row that you added to the *Invoices* table in exercise 1.
5. After you have verified that all of the modifications for the first four exercises have been successful, rollback the changes. Then, verify that they have been rolled back.

# How to work with data types and functions

In chapter 3, you were introduced to some of the built-in scalar functions such as the SUBSTR, TO\_CHAR, and SYSDATE functions. Now, this chapter expands on that coverage by presenting more of the built-in scalar functions. Because most of these functions work with specific data types, this chapter begins by describing the data types for an Oracle database.

<b>The built-in data types .....</b>	<b>232</b>
Data type overview .....	232
The character data types.....	234
The numeric data types .....	236
The temporal data types .....	238
The large object data types.....	240
<b>How to convert data from one type to another .....</b>	<b>242</b>
How to convert characters, numbers, and dates .....	242
Common number format elements .....	244
Common date/time format elements .....	246
How to convert characters to and from their numeric codes .....	248
<b>How to work with character data .....</b>	<b>250</b>
How to use the common character functions .....	250
How to parse a string.....	254
How to sort a string in numerical sequence .....	256
How to sort mixed-case columns in alphabetical sequence.....	256
<b>How to work with numeric data.....</b>	<b>258</b>
How to use the common numeric functions.....	258
How to search for floating-point numbers.....	260
<b>How to work with date/time data .....</b>	<b>262</b>
How to use the common date/time functions.....	262
How to parse dates and times .....	264
How to perform a date search.....	266
How to perform a time search .....	268
<b>Other functions you should know about .....</b>	<b>270</b>
How to use the CASE function.....	270
How to use the COALESCE, NVL, and NVL2 functions.....	272
How to use the GROUPING function .....	274
How to use the ranking functions.....	276
<b>Perspective .....</b>	<b>280</b>

## The built-in data types

---

A column's *data type* specifies the type of information that the column is intended to store. A column's data type also determines the types of operations that can be performed on the data.

### Data type overview

---

The Oracle data types can be divided into the five categories shown in the first table in figure 8-1. In this chapter, you'll learn how to work with the most important data types in the first three categories. Then, you can learn more about working with the data types in the temporal category in chapter 17, and you can learn how to work with the data types in the large object category in chapter 18.

The *character data types* are used to store a string of one or more characters that can include letters, numbers, or special characters like the pound sign (#) or the at sign (@). The terms *character*, *string*, and *text* are used interchangeably to describe this type of data.

The *numeric data types* are used to store numbers that can be used for mathematical calculations. These numbers can be *integers*, which are numbers that don't contain decimal places, or they can be numbers that contain decimal places. They can also be *floating-point numbers* that are used to store approximate values for very large and very small numbers.

The *temporal data types* are used primarily to store dates and times. In Oracle terminology, these data types are referred to as the *datetime*, *date/time*, or just *date* types. Remember, though that these types always include a time component as well as a date component. Besides dates and times, the temporal data types include time intervals and timestamps, which you can learn more about in chapter 17.

The *large object (LOB) data types* are used to store large amounts of text, images, sound, video, and so on. In the old days (you know, way back in the 80s and 90s), databases were primarily used to store character, numeric, and date/time data. Today, however, you can use the large object types to store other types of data. That's why we show how to use these data types in chapter 18.

The *rowid data types* are used to store an address for each row in a database. If you want, you can view the address for a row by using the ROWID pseudo-column. However, since most developers don't need to view these addresses, these data types aren't presented in this book. For more information about the rowid data types, look up "Rowid Datatypes" in the Oracle Database SQL Reference manual.

Most of the Oracle data types correspond to the ANSI-standard data types. These data types are listed in the second table in this figure. Here, the second column lists the Oracle data type names, and the first column lists the synonyms Oracle provides for the ANSI-standard data types. Although you can use these synonyms instead of the Oracle data types, there's no reason to do that. If you do, Oracle simply maps the synonyms to the corresponding Oracle data types.

## Built-in data type categories

Category	Description	Chapter
Character	Strings of characters	8
Numeric	Integer, decimal, and floating-point numbers	8
Temporal	Dates, times, time intervals, and timestamps	8 and 17
Large object (LOB)	Text, images, sound, and video	18
Rowid	Addresses for each row in the database	(not covered)

## ANSI data types and Oracle equivalents

ANSI synonyms	Oracle data type
CHARACTER (n)	CHAR (n)
CHAR (n)	
CHARACTER VARYING (n)	VARCHAR2 (n)
CHAR VARYING (n)	
NATIONAL CHARACTER (n)	NCHAR (n)
NATIONAL CHAR (n)	
NCHAR (n)	
NATIONAL CHARACTER VARYING (n)	NVARCHAR2 (n)
NATIONAL CHAR VARYING (n)	
NCHAR VARYING (n)	
NUMERIC (p,s)	NUMBER (p,s)
DECIMAL (p,s)	
INTEGER	NUMBER (38)
INT	
SMALLINT	
FLOAT	FLOAT (126)
DOUBLE PRECISION	FLOAT (126)
REAL	FLOAT (63)

## Description

- Oracle provides built-in *data types* that can be divided into the five categories shown above.
- Oracle provides a synonym for each of the ANSI-standard data types. These are the SQL data types that are specified by the American National Standards Institute.
- When you use the synonym for an ANSI data type, it's mapped to the appropriate Oracle data type indicated in the table above.

## The character data types

---

Figure 8-2 presents the four character data types. The CHAR and VARCHAR2 types store strings of characters defined by the *ASCII character set*. This character set uses one byte per character but only defines 256 characters, which is usually adequate for working with the English language.

The NCHAR and NVARCHAR2 data types store strings of characters defined by the *Unicode character set*. These types of characters require two or three bytes per character, depending on the type of encoding that's used. However, this character set defines over 65,000 characters including most characters from most of the world's languages. Since the Unicode characters are commonly referred to as *national characters*, these data types begin with the prefix *n*.

You use the CHAR and NCHAR data types to store *fixed-length strings*. Data stored using these data types always occupies the same number of bytes, regardless of the actual length of the string. These data types are typically used to define columns that have a fixed number of characters. For example, the vendor\_state column in the Vendors table is defined as CHAR(2) because it always contains two characters.

You use the VARCHAR2 and NVARCHAR2 data types to store *variable-length strings*. Data stored using these data types occupies only the number of bytes needed to store the string. These types are typically used to define columns whose lengths vary from one row to the next. In general, variable-length strings are more efficient than fixed-length strings because the database only uses the amount of disk space required by the value. For example, if you define a column as VARCHAR2(50) and it stores a value that's three characters long, the database only stores the three characters. In contrast, if you define the column as CHAR(50) and it stores a value that's three characters long, the database stores 50 characters, padding the value with 47 space characters.

In Oracle 8 through 11g, the maximum size for the NVARCHAR2 and VARCHAR2 data types was 4,000 bytes. In Oracle 12c and above, these types have the same maximum size by default. However, a DBA can raise the maximum size to 32,767 bytes by modifying the init.ora file so it sets the MAX\_STRING\_SIZE parameter to a value of EXTENDED.

Although you typically store numeric values using numeric data types, the character data types may be a better choice for some numeric values. For example, you typically store zip codes, telephone numbers, and social security numbers in character columns even though they contain only numbers. That's because their values aren't used in arithmetic operations. In addition, if you store these numbers in numeric columns, leading zeroes are stripped, which usually isn't what you want.

## The character data types used to store standard characters

Type	Description
<code>CHAR[ (size [BYTE CHAR]) ]</code>	Fixed-length strings of character data where size is the number of characters or bytes between 1 and 2000. The default is 1 character.
<code>VARCHAR2 (size [BYTE CHAR])</code>	Variable-length strings of character data where size is the maximum number of characters or bytes between 1 and 4000. The size argument is required.

## The character data types used to store Unicode characters

Type	Description
<code>NCHAR[ (size) ]</code>	Fixed-length Unicode characters where size is the number of characters. The default and minimum size is 1. Maximum size is determined by the national character set definition, with an upper limit of 2000 bytes.
<code>NVARCHAR2 (size)</code>	Variable-length Unicode characters where size is the maximum number of characters. The size argument is required. Maximum size is determined by the national character set definition, with an upper limit of 4000 bytes.

## Description

- The *ASCII character set* provides for 256 characters with 1 byte per character.
- The CHAR and VARCHAR2 types use the ASCII character set.
- The *Unicode character set* provides for over 65,000 characters, usually with 2 bytes per character. In the Unicode character set, the first 256 characters correspond with the 256 ASCII characters.
- The NCHAR and NVARCHAR2 types use the Unicode character set, which is commonly referred to as the *national character set*.
- The CHAR and NCHAR data types are typically used for *fixed-length strings*. These data types use the same amount of storage regardless of the actual length of the string. If you insert a value that's shorter than the specified type, the end of the value will be padded with spaces.
- The VARCHAR2 and NVARCHAR2 types are typically used for *variable-length strings*. These data types use only the amount of storage needed for a given string.
- For Oracle 12c and above, the DBA can raise the maximum size for the VARCHAR2 and NVARCHAR2 types to 32,767 bytes by editing the init.ora file and setting the MAX\_STRING\_SIZE parameter to a value of EXTENDED.

## The numeric data types

---

Figure 8-3 presents the numeric data types supported by Oracle. The NUMBER data type is used to store positive and negative numbers with a fixed number of digits. When you specify this data type, you give the precision and scale in parentheses. The *precision* is the total number of digits that can be stored in the column, and the *scale* is the number of digits that can be stored to the right of the decimal point. For business applications, this will probably be the only numeric data type that you will need.

For columns that contain integers, you code the NUMBER data type with just the precision. For example, a column defined as NUMBER(5) allows for an integer value of up to 99999. In contrast, you code both the precision and scale for numbers that contain decimal positions. For example, a column defined as NUMBER(7, 2) allows for a number seven digits long with two digits to the right of the decimal point. As a result, the maximum value for the column is 99999.99.

In contrast to the fixed numbers stored by the NUMBER data type, the FLOAT data type is used to store *floating-point numbers*. These numbers are useful for storing the values for very large or small numbers, but with a limited number of significant digits. As a result, the FLOAT data type doesn't always provide for exact values.

To express the value of a floating-point number, you can use *scientific notation*. To use this notation, you type the letter E followed by a power of 10. For instance, 3.65E+9 is equal to  $3.65 \times 10^9$ , or 3,650,000,000. If you have a scientific or mathematical background, of course, you're already familiar with this notation. And if you aren't already familiar with this notation, you probably aren't going to need to use floating-point numbers.

The BINARY\_FLOAT and BINARY\_DOUBLE data types were introduced with Oracle Database 10g to conform to the IEEE (Institute of Electrical and Electronic Engineers) standard for floating-point arithmetic. BINARY\_FLOAT is used to represent *single-precision*, 32-bit, floating-point numbers, while BINARY\_DOUBLE is used to represent *double-precision*, 64-bit, floating-point numbers.

## The numeric data types

Type	Bytes	Description.
NUMBER [ (p[, s]) ]	1 to 22	Stores zero as well as positive and negative fixed numbers with absolute values from $1.0 \times 10^{-130}$ to, but not including, $1.0 \times 10^{126}$ . The precision (p) can range from 1 to 38. The scale (s) can range from -84 to 127.
FLOAT [ (p) ]	1 to 22	Stores floating-point numbers. The precision (p) can range from 1 to 126 bits. A FLOAT value is represented internally as a NUMBER value.
BINARY_FLOAT	5	Stores single-precision, 32-bit, floating-point values.
BINARY_DOUBLE	9	Stores double-precision, 64-bit, floating-point.

### Description

- The numeric data types are used to store numbers.
- The *precision* of a NUMBER type indicates the total number of digits that can be stored in the data type. The *scale* indicates the number of decimal digits that can be stored to the right of the decimal point.
- For integers, you use the NUMBER type with just the precision in parentheses. For decimal numbers, you use the NUMBER type with both the precision and scale in parentheses.
- A *floating-point number* provides for very large and very small numbers that require decimal positions, but with a limited number of significant digits.
- A *single-precision* floating-point number provides for up to 7 significant digits. A *double-precision* floating-point number provides for up to 16 significant digits.
- The BINARY\_FLOAT and BINARY\_DOUBLE types were introduced with Oracle 10g.

Figure 8-3 The numeric data types

## The temporal data types

---

Figure 8-4 presents the temporal data types that are used to store dates and times. Of these data types, the DATE type is the oldest and the most commonly used. This data type stores the century, year, month, day, hour, minute and second.

When you use a SELECT statement to select a column that's defined with the DATE type, the date is displayed with the default format that's used by the database. On most Oracle systems, that means that August 19, 2014 is displayed as "19-AUG-14". Later in this chapter, though, you'll learn how to use functions to display date values in other formats such as "2014-08-19" or "8/19/14".

When you store a DATE value, the value includes a time component, but this component isn't displayed by default. Later in this chapter, you'll learn how to use functions to work with the time component of a DATE value. For example, you'll learn how to display the time component in a format like 04:20:36 PM or 16:20:36.

When Oracle Database 9i was released, it provided several new data types for working with temporal data. To start, it introduced the three TIMESTAMP types shown in this figure. These data types provide two advantages over the DATE type. First, you can use them to store fractional seconds. Second, you can use them to store a time zone.

Oracle Database 9i also introduced the INTERVAL types shown in this figure. These data types make it easier to work with time intervals like 2 days, 2 hours, and 12 minutes. Before these data types were introduced, you usually used the NUMBER type to store intervals of time.

Since the DATE type is the most widely used temporal type, you'll learn how to work with it in this chapter. For some applications, this is the only temporal type that you will need. However, if you want to learn how to use the TIMESTAMP and INTERVAL types, you can refer to chapter 17.

## The date/time data types

Type	Bytes	Description
<b>DATE</b>	7	Stores date and time information using fixed-length fields to store the century, year, month, day, minute, hour, and second. Valid dates range from January 1, 4712 BC, to December 31, 9999 AD. This type does not have fractional seconds or a time zone.
<b>TIMESTAMP [ (fsp) ]</b>	7 to 11	An extension of DATE. Stores year, month, day, hour, minute, second, and (optionally) the fractional part of a second where the fractional second precision (fsp) is the number of decimal places used to store the fractional part of a second. The fsp can range from 0 to 9, and the default is 6. This type does not have a time zone.
<b>TIMESTAMP [ (fsp) ] WITH TIME ZONE</b>	13	Works like TIMESTAMP except that it includes either a time zone region name or a time zone offset.
<b>TIMESTAMP [ (fsp) ] WITH LOCAL TIME ZONE</b>	7 to 11	Works like TIMESTAMP WITH TIME ZONE except that the time zone is set to the database time zone when it is stored in the database, and the user sees the session time zone when data is retrieved.
<b>INTERVAL YEAR [ (yp) ] TO MONTH</b>	5	Stores a time interval in years and months. Year precision (yp) is the number of digits in the year, and the default is 2.
<b>INTERVAL DAY [ (dp) ] TO SECOND [ (fsp) ]</b>	11	Stores a time interval in days, hours, minutes and seconds. Day precision (dp) is the maximum number of digits in the day, and the default is 2. The fsp (fractional second precision) can range from 0 to 9, and the default is 6.

## Description

- The *temporal data types* are typically referred to as the *date/time*, *datetime*, or simply *date* types.
- The default date format is determined by the `NLS_DATE_FORMAT` and `NLS_TERRITORY` parameters. For more information about these parameters, you can refer to the Oracle Database Globalization Support Guide.
- In this chapter, you'll learn how to work with the most common temporal data type, the `DATE` type. To learn how to work with the other temporal data types, see chapter 17.

Figure 8-4 The temporal data types

## The large object data types

---

Figure 8-5 presents the new large object (LOB) data types that were introduced with Oracle Database 8. These data types make it easier to store large amounts of character or binary data. For example, they can be used to store text, XML, Word, PDF, image, sound, and video files.

The CLOB (Character Large Object) and NCLOB (National Character Large Object) types can store character data. These data types are commonly used to store large text and XML files. The primary difference between these types is that the CLOB type uses 1 byte per character to store characters in the ASCII character set while the NCLOB type uses 2 or 3 bytes per character to store characters in the Unicode character set.

The BLOB (Binary Large Object) type can store any kind of data in binary format. It can be used to store binary files such as PDF files, and it can be used to store image, sound, and video files.

The BFILE (Binary File) type stores a pointer to a binary file that's stored outside of the database. These binary files can be stored anywhere that's accessible through the host computer's file system.

This figure finishes by presenting the old data types for storing large objects that were used prior to Oracle Database 8. These data types are provided primarily for backward compatibility. As a result, you should use one of the new LOB data types for any new development.

## The large object data types

Type	Description
<b>CLOB</b>	Character large object. Stores up to 8 terabytes of character data inside the database.
<b>NCLOB</b>	National character large object. Stores up to 8 terabytes of national character data inside the database.
<b>BLOB</b>	Binary large object. Stores up to 8 terabytes of unstructured binary data inside the database.
<b>BFILE</b>	Binary file. Stores a pointer to a large binary file stored outside the database in the file system of the host computer.

## The old data types for large objects

Type	Description
<b>RAW (size)</b>	Stores up to 2000 bytes of binary data that is not intended to be converted by Oracle when moving data between different systems.
<b>LONG</b>	Stores up to 2 gigabytes of character data.
<b>LONG RAW</b>	Stores up to 2 gigabytes of raw binary data that is not intended to be converted.

## Description

- The *large object (LOB)* data types can store large and unstructured data such as text, images, sound, and video.
- For more information about large object data types, see chapter 18.
- For Oracle 12c and above, the DBA can raise the maximum size for the RAW type to 32,767 bytes by editing the init.ora file and setting the MAX\_STRING\_SIZE parameter to a value of EXTENDED.

## How to convert data from one type to another

---

As you work with the various data types, you'll find that you frequently need to convert a value from one data type to another. To do that, you can use the functions that are described next.

### How to convert characters, numbers, and dates

---

Figure 8-6 shows how to use three of the Oracle functions to convert data from one type to another. The TO\_CHAR function converts an expression to the VARCHAR2 data type. The TO\_NUMBER function converts an expression to the NUMBER type. And the TO\_DATE function converts an expression to the DATE type.

As the syntax summaries and examples for these functions show, you can code these functions with or without format specifications. When you code a format specification, it determines how the data is converted. Otherwise, the default format for the system is used. In the next two figures, you'll learn how to code the format specifications for converting numbers and dates.

To show how these functions work, these examples use literal values to specify numbers and dates. As a result, date literals are enclosed in single quotation marks, and numeric literals aren't enclosed. In practice, though, you're more likely to use these functions on columns or expressions.

In the first two examples, you can see how the TO\_CHAR function can be used to convert a numeric value to character data. In the next two examples, you can see how the system date is converted to character data with and without a format specification. In the third example, only the date is shown. In the fourth example, both the date and time are shown with a 24-hour clock. In both cases, the system date includes the date and time, but the default date format doesn't show the time.

In the last four examples, you can see how the TO\_NUMBER and TO\_DATE functions work. In the seventh example, the TO\_DATE function converts the characters "15-APR-14" to a DATE data type with a time value of 00:00:00. In the eighth example, this DATE data type is converted back to characters with a format that shows both the date and time with a 24-hour clock.

Besides the three TO functions in this figure, Oracle provides TO functions for all of its built-in types. For example, you can use the TO\_NCHAR function to convert a number or date/time value to an NVARCHAR2 type, and you can use the TO\_TIMESTAMP function to convert a character type to a TIMESTAMP type. For more information about TO functions, navigate to the "Functions" section of the Oracle Database SQL Reference manual and scroll down to the functions that begin with TO.

One other way to convert data is to use the CAST function that's shown in this figure. This is an ANSI-standard function that lets you convert, or *cast*, an expression to the data type you specify. In the example, the SELECT statement casts the invoice\_date column to the VARCHAR2(9) data type, and it casts the

## Oracle functions for converting data

Function	Description
<code>TO_CHAR(expr[, format])</code>	Converts the result of an expression to a value of the VARCHAR2 type.
<code>TO_NUMBER(expr[, format])</code>	Converts the result of an expression to a value of NUMBER type.
<code>TO_DATE(expr[, format])</code>	Converts the result of an expression to a value of DATE type.

## Examples that use the Oracle TO functions

Example	Resulting Value
<code>TO_CHAR(1975.5)</code>	1975.5
<code>TO_CHAR(1975.5, '\$99,999.99')</code>	\$1,975.50
<code>TO_CHAR(SYSDATE)</code>	15-APR-14
<code>TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS')</code>	15-APR-2014 10:53:56
<code>TO_NUMBER('1975.5')</code>	1975.5
<code>TO_NUMBER('\$1,975.5', '\$99,999.99')</code>	1975.5
<code>TO_DATE('15-APR-14')</code>	15-APR-2014 00:00:00
<code>TO_CHAR(TO_DATE('15-APR-14'), 'DD-MON-YYYY HH24:MI:SS')</code>	15-APR-2014 00:00:00

## A statement that uses ANSI-standard CAST functions

```
SELECT invoice_id, invoice_date, invoice_total,
       CAST(invoice_date AS VARCHAR2(9)) AS varchar_date,
       CAST(invoice_total AS NUMBER(9)) AS integer_total
FROM invoices
```

### Description

- If a format element is specified, the TO functions use that format element to format the value that's returned. Otherwise, these functions use the default formats for the system.
- Although this figure only lists three of the TO functions, Oracle provides TO functions for most of the built-in data types.
- CAST is an ANSI-standard function that you can use for simple conversions, but the TO functions give you more control over the conversions that are done.
- For more information on the format elements for TO functions, please see the next two figures.
- For more information about TO functions, navigate to the “Functions” section of the Oracle Database SQL Reference manual and scroll down to the functions that begin with TO.

invoice\_total column to the NUMBER(9) type. Although this can be useful, you're usually better off using the TO functions because they give you more control over the conversions.

## Common number format elements

---

Figures 8-7 summarizes the most common number format elements and provides some examples. These examples show how to specify the currency symbols, group separators, and signs that are used to format numbers. Here, if you specify a format that's too short to accommodate the number, the database will return one or more # characters.

Since these format elements often provide several ways to get the same result, you can use the format element that makes the most sense for your situation. For example, if you always want the currency symbol to be a dollar sign (\$), you can use this sign to specify the currency symbol. However, if you want to use the default currency symbol for the database, you can use one of the other elements to specify the currency symbol. Then, the symbol that's used will vary depending on the NLS parameters that are stored within the database.

When coding some format elements, you can code the element before or after the format for the number. For example, you can code the MI element before or after the format for the number. In other cases, you must code the format element before the format for the number. For example, you must code the FM element before the format for the number.

## Number format elements

Element	Description
9	Digits
.	Decimal point
D	Decimal point
,	Comma
G	Group separator
0	Leading or trailing zeros
\$	Dollar sign
L	Local currency symbol
U	Dual currency symbol
C	Currency symbol
S	Minus sign for negative numbers, plus sign for positive numbers
MI	Minus sign for negative numbers
PR	Negative numbers in brackets, one space before and after positive numbers
FM	Removes leading or trailing spaces or zeros
EEEE	Scientific notation

## Number format examples

Value	Format	Output
1975.5	(none specified)	1975.5
1975.5	999	###
1975.5	9999	1976
1975.5	9,999.9	1,975.5
1975.5	9G999D9	1,975.5
1975.5	99,999.99	1,975.50
1975.5	09,999.990	01,975.500
1975.5	\$99,999.99	\$1,975.50
1975.5	L9,999.99	\$1,975.50
1975.5	U9,999.99	\$1,975.50
1975.5	C9,999.99	USD1,975.50
1975.5	S9,999.99	+1,975.50
-1975.5	9,999.99S	1,975.50-
-1975.5	9,999.99MI	1,975.50-
1975.5	9,999.99MI	1,975.50
-1975.5	9,999.99PR	<1,975.50>
1975.5	9,999.99PR	1,975.50
01975.50	FM9,999.99	1,975.5
1975.5	9.99EEEE	1.98E+03

## Description

- For more information, look up “Number Format Elements” in the Oracle Database SQL Reference manual.

Figure 8-7 Common number format elements

## Common date/time format elements

---

Figures 8-8 shows how to use the most common date/time format elements. After you read through the list of format elements, you shouldn't have much trouble understanding how the examples work.

However, the RR format element requires some explanation. This element allows you to use a two-digit year to specify years in two different centuries. For this century, for example, years from 00 to 49 are interpreted as 2000 to 2049, and years 50 through 99 are interpreted as 1950 through 1999. As a result, 98 is interpreted as 1998, not 2098. In most cases, that's what you want. If that isn't what you want, you can use the YY format to specify the year.

In addition, note that the FF format element works with the fractional seconds component, which isn't available from the DATE type. As a result, you will only use this element when you're working with the TIMESTAMP and INTERVAL types that are described in chapter 17.

If you study the summary and examples in this figure, you will see that you can include periods when using the BC, AD, AM, and PM elements. If, for example, you specify "B.C." as a format element, the formatted date will include a value of "B.C" or "A.D.", depending on whether the date is before or after the birth of Christ.

You will also see that the MONTH and DAY elements pad the end of the value that's returned with spaces to accommodate the longest value that can be returned. Since this makes it easy to align the month and day elements of a date in columns, this is often what you want. If it isn't what you want, you can use the TRIM function described later in this chapter to trim the spaces from the ends of these elements.

Last, if a format element returns letters, the capitalization that you use when you specify the format element corresponds with the capitalization for the returned value. If, for example, you specify "MONTH", the month will be returned in all caps (AUGUST). If you specify "Month", the month will be returned with an initial cap (August). And if you specify "month", the month will be returned in lowercase (august). Note that this also applies to the date format elements for both the full and abbreviated names of months and days.

## Common date/time format elements

Element	Description	Element	Description
<b>AD</b>	Anno Domini	<b>DAY</b>	Name of day padded with spaces
<b>BC</b>	Before Christ	<b>DY</b>	Abbreviated name of day
<b>CC</b>	Century	<b>DDD</b>	Day of year (1-366)
<b>YEAR</b>	Year spelled out	<b>DD</b>	Day of month (01-31)
<b>YYYY</b>	Four-digit year	<b>D</b>	Day of week (1-7)
<b>YY</b>	Two-digit year	<b>HH</b>	Hour of day (01-12)
<b>RR</b>	Two-digit round year	<b>HH24</b>	Hour of day (01-24)
<b>Q</b>	Quarter of year (1-4)	<b>MI</b>	Minute (00-59)
<b>MONTH</b>	Name of month padded with spaces	<b>SS</b>	Second (00-59)
<b>MON</b>	Abbreviated name of month	<b>SSSS</b>	Seconds past midnight (0-86399)
<b>MM</b>	Month (01-12)	<b>FF[1-9]</b>	Fractional seconds
<b>WW</b>	Week of year (1-52)	<b>PM</b>	Post Meridian
<b>W</b>	Week of month (1-5)	<b>AM</b>	Ante Meridian

## Common date and time formats

Format	Example
(none specified)	19-AUG-14
DD-MON-YY	19-AUG-14
DD-MON-RR	19-AUG-14
DD-Mon-YY	19-Aug-14
MM/DD/YY	08/19/14
YYYY-MM-DD	2014-08-19
Dy Mon DD, YY	Tue Aug 19, 14
MONTH DD, YYYY BC	AUGUST 19, 2014 AD
Month DD, YYYY B.C.	August 19, 2014 A.D.
HH:MI	04:20
HH24:MI:SS	16:20:36
HH:MI AM	04:20 PM
HH:MI A.M.	04:20 P.M.
HH:MI:SS	04:20:36
HH:MI:SS.FF5	04:20:36.12345
HH:MI:SS.FF4	04:20:36.1234
YYYY-MM-DD HH:MI:SS AM	2014-08-19 04:20:36 PM

## Description

- If you use the RR format, years from 00 to 49 are interpreted as 2000 to 2049, and years 50 through 99 are interpreted as 1950 through 1999.
- For more information about date/time format elements, look up “Datetime Format Elements” in the Oracle Database SQL Reference manual.

Figure 8-8 Common date/time format elements

## How to convert characters to and from their numeric codes

---

Figure 8-9 shows three functions that are used to convert characters to and from their equivalent numeric code. The CHR and ASCII functions work with standard ASCII characters.

For instance, the CHR function in this figure converts the number 97 to its equivalent ASCII code, the letter *a*. Conversely, the ASCII function converts the letter *a* to its numeric equivalent of 97. Although the string in the ASCII function can include more than one character, please note that only the first character is converted.

The NCHR function works like the CHR function. However, since it usually works with the Unicode character set, it's able to convert thousands of characters.

The CHR function is frequently used to output ASCII control characters that can't be typed on your keyboard. The three most common control characters are presented in this figure. These characters can be used to format output so it's easier to read. The SELECT statement in this figure, for example, uses the CHR(13) control character to start a new line after the vendor name and vendor address in the output.

### Three functions for converting characters to and from their numeric codes

Function	Description
<b>ASCII(string)</b>	Returns a NUMBER value that corresponds to the first character in the specified string of the CHAR, VARCHAR2, NCHAR, or NVARCHAR2 type.
<b>CHR(number)</b>	Returns a VARCHAR2 value that corresponds to the specified NUMBER value in the ASCII character set.
<b>NCHR(number)</b>	Returns a VARCHAR2 value that corresponds to the specified NUMBER value in the national character set.

### Examples that use the ASCII, CHR, and NCHR functions

Example	Result
<code>ASCII('a')</code>	97
<code>ASCII('abc')</code>	97
<code>CHR(97)</code>	a
<code>NCHR(97)</code>	a
<code>NCHR(332)</code>	ó

### ASCII codes for common control characters

Control character	Value
Tab	<code>CHR(9)</code>
Line feed	<code>CHR(10)</code>
Carriage return	<code>CHR(13)</code>

### A SELECT statement that uses the CHR function to format output

```
SELECT vendor_name || CHR(13)
    || vendor_address1 || CHR(13)
    || vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code
AS vendor_address
FROM vendors
WHERE vendor_id = 1
```

#### The value that's returned

VENDOR_ADDRESS
1 US Postal ServiceAttn: Supt. Window ServicesMadison, WI 53707

## How to work with character data

---

Now that you know how to convert the number and date/time data types to and from the character data types, you’re ready to learn some additional functions for working with character data.

### How to use the common character functions

---

Figure 8-10 shows how to use some of the common character functions available from Oracle. To start, you can use the LTRIM and RTRIM functions to remove leading or trailing characters from the left or right side of a string. Or, you can use the TRIM function to remove leading and trailing characters from both sides of a string. Most of the time, you’ll use these functions to remove leading and trailing spaces. However, you can also use these functions to remove other characters such as zeros (0), periods (.), and so on.

Conversely, you can use the LPAD and RPAD functions to add leading or trailing characters to the left or right side of a string. Again, most of the time, you’ll use these functions to add leading and trailing spaces. However, you can also use these functions to add other characters.

You can use the LOWER and UPPER functions to convert the characters in a string to lower or uppercase. In addition, you can use the INITCAP function to convert the characters in a string so the initial letter in each word is capitalized and the rest of the word is lowercase.

You can use the NVL and NVL2 functions to provide a substitute string if the string value is a NULL value. For example, you can use these functions to display a value of “Unknown” instead of displaying the default value of null.

You can use the last four functions presented in this figure to modify a string. To start, you can use the SUBSTR function to return the specified number of characters from anywhere in a string. When you use this function, you can use the second argument to specify the starting point where 1 is the first character in the string. In addition, you can use the third argument to specify the number of characters that you want to return. If you don’t specify this argument, this function will return all characters from the starting point to the end of the string.

To specify the starting point and length arguments for the SUBSTR function, it’s common to use the INSTR and LENGTH functions to return integer values. For example, if you want to find the position of the first space in a string, you can use an INSTR function to return an integer value for its position. Then, if necessary, you can perform arithmetic operations on this integer, and you can nest the resulting expression within a SUBSTR function. Similarly, you can use the LENGTH function to return the number of characters in a string, you can include this function in an arithmetic expression, and you can nest the expression within a SUBSTR function.

Finally, you can use the REPLACE function to replace a substring within a string with another substring. For example, you might want to use this function to replace hypens (-) with periods (.).

## Some common character functions

Function	Description
<b>LTRIM(string[, trim_string])</b>	By default, this function removes any spaces from the left side of the specified string. If you specify a trim string, this function removes all characters specified in the trim string.
<b>RTRIM(string[, trim_string])</b>	Same as LTRIM but removes characters from the right side of the string instead of the left.
<b>TRIM([trim_char FROM ]string)</b>	Removes any spaces from the left and right sides of the specified string. If you specify a trim character, this function removes the trim character from both sides of the string.
<b>LPAD(string, length[, pad_string])</b>	Pads the left side of the string to the specified length with spaces or with the characters specified by the pad string.
<b>RPAD(string, length[, pad_string])</b>	Pads the right side of the string to the specified length with spaces or with the characters specified by the pad string.
<b>LOWER(string)</b>	Converts the string to lowercase letters.
<b>UPPER(string)</b>	Converts the string to uppercase letters.
<b>INITCAP(string)</b>	Converts the initial letter in each word to uppercase.
<b>NVL(string, value)</b>	Returns the value argument if the specified string is a null value. Otherwise, this function returns the specified string.
<b>NVL2(string, value1, value2)</b>	Returns the value2 argument if the specified string is a null value. Otherwise, this function returns the value1 argument.
<b>SUBSTR(string, start[, length])</b>	Returns the specified number of characters (length) from the string (string) at the specified starting position (start).
<b>LENGTH(string)</b>	Returns an integer for the number of characters in the specified string.
<b>INSTR(string, find [,start])</b>	Returns an integer for the position of the first occurrence of the specified find string in the specified string starting at the specified position. If the starting position isn't specified, the search starts at the beginning of the string. If the string isn't found, the function returns zero.
<b>REPLACE(string, find, replace)</b>	Returns the string with all occurrences of the specified find string replaced with the specified replace string.

Figure 8-10 How to use the common character functions (part 1 of 2)

Part 2 of figure 8-10 shows examples of the string functions described in part 1. To start, the LTRIM and RTRIM examples remove spaces from the left and right sides of a string. Then, the TRIM example removes spaces from both sides of a string. Here, the result column uses single quotes to identify strings. This distinguishes string values from number values, and it shows the leading and trailing spaces for the result.

The second LTRIM example shows how to remove other characters besides spaces from a string. Here, the second argument specifies that all dollar signs and zeros should be removed from the left side of the string. This shows that the second argument of the LTRIM and RTRIM functions allows you to trim multiple characters.

Conversely, the second TRIM example uses a different syntax that only allows you to specify a single character. Here, you must specify that character, followed by the FROM keyword, followed by the string.

The LPAD and RPAD examples show how to add spaces or other characters to the left or right side of a string. If you want, you can use these functions to align the columns of a result set. In particular, note how the LPAD function can be used to align numbers with the right side of a column.

The LOWER, UPPER, and INITCAP examples show how to change the case of a string. Note how the INITCAP example works the same regardless of whether the input string is uppercase or lowercase.

The NVL and NVL2 examples show how to substitute a string for a NULL value. In these examples, string literals are used to specify a string value, and the NULL keyword is used to specify a NULL value.

The SUBSTR examples show how to return part of a string. To start, the first example returns the first five characters of a string. To do that, the second argument specifies 1 as the position for first character in the string, and the second argument specifies 5 as the length of the string. The second SUBSTR example works similarly, but it returns a string that begins at the seventh character and is three characters long. Unlike the first two examples, the third SUBSTR example doesn't include a third argument. As a result, it returns all characters in the string from the seventh character to the end of the string.

The INSTR examples show how to search for a string within a string and to return an integer value for its starting position. To start, the first example shows how to return an integer for the position of the first space in a string. In this case, the space is the sixth character of the string. Then, the second example shows how to return an integer for the first hyphen in the string. Next, the third example shows how to return an integer for the second hyphen in the string. To do that, this example uses the third argument of the INSTR function to start the search at the fifth character in the string. Finally, the fourth example shows how to return the position for a string literal of "1212". This shows that you can use this function to search for a single character or a string of multiple characters.

The LENGTH examples show how to return an integer for the length of a string. Note that this includes any leading or trailing spaces.

The REPLACE examples show how to replace part of a string with another string. Here, the first example replaces all of the hyphens within the string with

## Character function examples

Example	Result
LTRIM(' John Smith ')	'John Smith '
RTRIM(' John Smith ')	' John Smith'
TRIM(' John Smith ')	'John Smith'
LTRIM('\$0019.99', '\$0')	'19.99'
TRIM('\$' FROM '\$0019.99')	'0019.99'
LPAD('\$19.99', 15)	'\$19.99'
LPAD('\$2150.78', 15)	'\$2150.78'
LPAD('\$2150.78', 15, '.')	'.....\$2150.78'
RPAD('John', 15)	'John '
RPAD('John', 15, '.')	'John.....'
LOWER('CA') 'ca'	
UPPER('ca') 'CA'	
INITCAP('john smith')	'John Smith'
INITCAP('JOHN SMITH')	'John Smith'
NVL('Fresh Corn Records', 'Unknown Company Name')	'Fresh Corn Records'
NVL(NULL, 'Unknown Company Name')	'Unknown Company Name'
NVL2('Fresh Corn Records', 'Known', 'Unknown')	'Known'
NVL2(NULL, 'Known', 'Unknown')	'Unknown'
SUBSTR('(559) 555-1212', 1, 5)	'(559) '
SUBSTR('(559) 555-1212', 7, 3)	'555'
SUBSTR('(559) 555-1212', 7)	'555-1212'
INSTR('(559) 555-1212', ' ')	6
INSTR('559-555-1212', '-')	4
INSTR('559-555-1212', '-', 5)	8
INSTR('559-555-1212', '1212')	9
LENGTH('(559) 555-1212')	14
LENGTH(' (559) 555-1212 ')	18
REPLACE('559-555-1212', '-', '.')	'559.555.1212'
REPLACE('559-555-1212', '--', '')	'5595551212'

## Description

- For more information about these and other functions, you can look them up in the Oracle Database SQL Reference manual. To do that, you can start by navigating to the “Functions” topic. Then, you can navigate to the function you want to learn more about.

Figure 8-10 How to use the common character functions (part 2 of 2)

periods. Then, the second example replaces all of the hypens within the string with nothing. To do that, the third example uses two single quotes with nothing between them to specify a value of nothing. Although these examples use a string that contains a single character for the second and third arguments, you can specify a string that contains multiple characters for these arguments.

## How to parse a string

---

Figure 8-11 shows how to parse a string. To start, the first SELECT statement shows how you can use the SUBSTR function to format columns in a result set. Here, the second column begins by getting the first name of the vendor contact and appending a space to the end of this name. Then, it uses the SUBSTR function to return the first initial of the last name for the vendor contract and appends this initial to the string. Finally, this column adds a period after the initial for the last name.

The third column displays the vendor's phone number without an area code. To accomplish that, this column specification uses the SUBSTR function to return all characters of the vendor\_phone column starting at the seventh character. Of course, this only works correctly because all of the phone numbers are stored in the same format with the area code in parentheses.

This SELECT statement also shows how you can use a function in the search condition of a WHERE clause. This condition uses the SUBSTRING function to select only those rows with an area code of 559. To do that, it retrieves three characters from the vendor\_phone column starting with the second character. Again, this assumes that the phone numbers are all in the same format and that the area code is enclosed in parentheses.

The second SELECT statement shows how to extract multiple values from a single column. In this example, both a first and a last name are stored in the Name column of the String\_Sample table. As a result, if you want to work with the first and last names independently, you have to parse the string using the string functions. In this example, the first name is considered to be every non-blank character up to the first blank, and the last name is considered to be every character after the first blank.

To extract the first name, this statement uses the SUBSTR and INSTR functions. First, it uses the INSTR function to locate the first space in the Name column. Then, it uses the SUBSTR function to extract all of the characters up to that space. Note that a value of one is subtracted from the value that's returned by the INSTR function, so the space itself isn't included in the first name.

To extract the last name, this statement uses the same functions. First, it uses the INSTR function to locate the first space in the Name column. Then, it uses the SUBSTR function to extract all of the characters from that space to the end of the string. Note that a value of one is added to the value that's returned by the INSTR function, so the space itself isn't included in the last name.

As you review this example, you should keep in mind that I kept it simple so that you can focus on how the string functions are used. You should realize, however, that this code won't work for all names. If, for example, a first name contains a space, such as in the name Jean Paul, this code won't work properly.

## A SELECT statement that uses the SUBSTR function

```
SELECT vendor_name,
       vendor_contact_first_name || ' ' ||
           SUBSTR(vendor_contact_last_name, 1, 1) || '.'
               AS contact_name,
       SUBSTR(vendor_phone, 7) AS phone
FROM vendors
WHERE SUBSTR(vendor_phone, 2, 3) = '559'
ORDER BY vendor_name
```

### The result set

VENDOR_NAME	CONTACT_NAME	PHONE
1 Abbey Office Furnishings	Kyra F.	555-8300
2 BFI Industries	Erick K.	555-1551
3 Bill Marvin Electric Inc	Kaitlin H.	555-5106
4 Cal State Termite	Demetrius H.	555-1534
5 California Business Machines	Anders R.	555-5570

(34 rows selected)

## The String\_Sample table

ID	NAME
1 1	Lizbeth Darien
2 2	Darnell O'Sullivan
3 17	Lance Pinos-Potter
4 20	Jean Paul Renard
5 3	Alisha von Strump

### A SELECT statement that parses a string

```
SELECT SUBSTR(name, 1, (INSTR(name, ' ') - 1)) AS first_name,
       SUBSTR(name, (INSTR(name, ' ') + 1)) AS last_name
FROM string_sample
```

### The result set

FIRST_NAME	LAST_NAME
1 Lizbeth	Darien
2 Darnell	O'Sullivan
3 Lance	Pinos-Potter
4 Jean	Paul Renard
5 Alisha	von Strump

## Description

- When parsing strings, it's common to nest one function within another.

Figure 8-11 How to parse a string

That illustrates the importance of designing a database so that this type of problem doesn't occur. You'll learn more about that in section 3. For now, just realize that if a database is designed correctly, you won't have to worry about this type of problem.

## How to sort a string in numerical sequence

---

Figure 8-12 addresses a common problem that occurs when you store numeric data in a character column and then want to sort the column in numeric sequence. In this figure, the columns in the String\_Sample table are defined with character data types. As a result, the first example sorts the values in the ID column in alphabetical sequence instead of numeric sequence, which isn't what you want.

One way to solve this problem is to convert the values in the ID column to integers for sorting purposes as shown in the SELECT statement in the second example. To do that, this example uses the TO\_NUMBER function to convert the ID column to a NUMBER value. As a result, this example sorts the rows in numeric sequence.

Another way to solve this problem is to pad the numbers with leading zeros or spaces as shown in the third example. To do that, this example uses the LPAD function to pad the left side of the ID column with zeros. When it does, it specifies an alias for this column of lpad\_id. Then, it sorts the result set by this alias, which causes the rows to be returned in numeric sequence.

## How to sort mixed-case columns in alphabetical sequence

---

You may remember from chapter 3 that uppercase letters come before lowercase letters when you sort a string in ascending sequence. As a result, a vendor name like "ASC Signs" comes before "Abbey Office Furnishings."

Now that you know how to use the LOWER and UPPER functions, though, this problem is easy to fix. Just convert the strings to all uppercase or all lowercase before you sort them as in

`ORDER BY LOWER(vendor_name)`

or

`ORDER BY UPPER(vendor_name)`

Either way, the rows will be sorted in the correct alphabetical sequence.

## A table sorted by a character column

```
SELECT * FROM string_sample  
ORDER BY id
```

### The result set

ID	NAME
1 1	Lizabeth Darien
2 17	Lance Pinos-Potter
3 2	Darnell O'Sullivan
4 20	Jean Paul Renard
5 3	Alisha von Strump

## A table sorted by a character column treated as a numeric column

```
SELECT * FROM string_sample  
ORDER BY TO_NUMBER(id)
```

### The result set

ID	NAME
1 1	Lizabeth Darien
2 2	Darnell O'Sullivan
3 3	Alisha von Strump
4 17	Lance Pinos-Potter
5 20	Jean Paul Renard

## A table sorted by a character column that's padded with leading zeros

```
SELECT LPAD(id, 2, '0') AS lpad_id, name  
FROM string_sample  
ORDER BY lpad_id
```

### The result set

LPAD_ID	NAME
1 01	Lizabeth Darien
2 02	Darnell O'Sullivan
3 03	Alisha von Strump
4 17	Lance Pinos-Potter
5 20	Jean Paul Renard

## Description

- If you sort by a character column that contains numbers, you may receive unexpected results. To avoid that, you can convert the string column to a numeric value or you can pad the left side of the character column with leading zeros or spaces.

## How to work with numeric data

---

In addition to the character functions, Oracle provides several functions for working with numeric data. Although you'll probably use only a couple of these functions regularly, you should be aware of them in case you ever need them.

### How to use the common numeric functions

---

Figure 8-13 summarizes some of the common numeric functions that Oracle provides. The function you'll probably use most often is the ROUND function. This function rounds a number to the precision specified by the length argument. Note that you can round the digits to the left of the decimal point by coding a negative value for this argument. However, you're more likely to code a positive number to round the digits to the right of the decimal point.

Another function that you might use regularly is the TRUNC function. This function works like the ROUND function, but it truncates the number instead of rounding to the nearest number. In other words, this function chops off the end of the number without doing any rounding. For example, if you round 19.99 to the nearest integer, you get a value of 20. However, if you truncate 19.99, you get a value of 19.

If you study the examples in this figure, you shouldn't have much trouble understanding how they work. For instance, you can easily see how a negative length argument causes the ROUND and TRUNC functions to round and truncate to the left of the decimal point.

In addition to the functions shown in this figure, Oracle provides many other functions for performing mathematical calculations. In particular, it provides many functions for performing trigonometric calculations. Since you're not likely to use these functions for business applications, they aren't presented in this book. However, if you need a function that isn't shown here, you can search for the function in the Oracle Database SQL Reference manual.

## Some common numeric functions

Function	Description
<code>ROUND(number[, length])</code>	Returns the number rounded to the precision specified by the length argument. If the length argument is positive, the digits to the right of the decimal point are rounded. If it's negative, the digits to the left of the decimal point are rounded.
<code>TRUNC(number[, length])</code>	Returns the number truncated as specified by the length argument. If the length argument is omitted, this function truncates all numbers to the right of the decimal point. If it's negative, this function truncates the specified number of digits to the left of the decimal point.
<code>CEIL(number)</code>	Returns the smallest integer that is greater than or equal to the number.
<code>FLOOR(number)</code>	Returns the largest integer that is less than or equal to the number.
<code>ABS(number)</code>	Returns the absolute value of the number.
<code>SIGN(number)</code>	Returns -1 if the number is less than 0, 0 if the number is equal to 0, and 1 if the number is greater than 0.
<code>MOD(number, number_divisor)</code>	Returns the remainder after the number argument is divided by the divisor.
<code>POWER(number, number_exponent)</code>	Returns the number after it has been raised to the power of the specified exponent.
<code>SQRT(number)</code>	Returns the square root of the number.

## Examples that use the numeric functions

Example	Result	Example	Result
<code>ROUND(12.5)</code>	13	<code>ABS(1.25)</code>	1.25
<code>ROUND(12.4999, 0)</code>	12	<code>ABS(-1.25)</code>	1.25
<code>ROUND(12.4999, 1)</code>	12.5	<code>SIGN(1.25)</code>	1
<code>ROUND(12.4944, 2)</code>	12.49	<code>SIGN(0)</code>	0
<code>ROUND(1264.99, -2)</code>	1300	<code>SIGN(-1.25)</code>	-1
<code>TRUNC(12.5)</code>	12	<code>MOD(10, 10)</code>	0
<code>TRUNC(12.4999, 1)</code>	12.4	<code>MOD(10, 9)</code>	1
<code>TRUNC(12.4944, 2)</code>	12.49	 	
<code>TRUNC(1264.99, -2)</code>	1200	<code>POWER(2, 2)</code>	4
 		<code>POWER(2, 2.5)</code>	5.65685...
<code>CEIL(1.25)</code>	2	 	
<code>CEIL(-1.25)</code>	-1	<code>SQRT(4)</code>	2
<code>FLOOR(1.25)</code>	1	<code>SQRT(5)</code>	2.23606...
<code>FLOOR(-1.25)</code>	-2		

### Note

- You can use these functions to work with any numeric data type or any nonnumeric type that can be implicitly converted to a numeric type.

Figure 8-13 How to use the common numeric functions

## How to search for floating-point numbers

---

Earlier in this chapter, you learned that floating-point numbers don't always contain exact values. From a practical point of view, though, that means that you don't want to search for an exact value when you're working with floating-point numbers. If you do, you'll miss values that are essentially equal to the value you want.

To illustrate, consider the `Float_Sample` table shown in figure 8-14. This table includes a column named `float_value` that's defined with the `BINARY_DOUBLE` data type. Here, the first example shows what happens when a `SELECT` statement retrieves all rows where the value stored in the `float_value` column is equal to 1. As you can see, the result set includes only the second row, even though the first and third rows also contain a value that's approximately equal to 1.

To solve this problem, you can search for an approximate value when you perform a search on a column with a floating-point data type. To do that, you can search for a range of values as shown by the second example in this figure. This `SELECT` statement searches for values between .99 and 1.01.

Another alternative is to round the value that you're searching for. This is illustrated by the third example. In both the second and third examples, the statement returns the three rows in the `Float_Sample` table that have a float value that's approximately equal to 1.

## The Float\_Sample table

FLOAT_ID	FLOAT_VALUE
1	1 0.999999999999999
2	21.0
3	31.000000000000001
4	4 1234.56789012345
5	5 999.04440209348
6	6 24.04849

## A SELECT statement that searches for an exact value

```
SELECT * FROM float_sample  
WHERE float_value = 1
```

### The result set

FLOAT_ID	FLOAT_VALUE
1	21.0

## A SELECT statement that searches for a range of values

```
SELECT * FROM float_sample  
WHERE float_value BETWEEN 0.99 AND 1.01
```

### The result set

FLOAT_ID	FLOAT_VALUE
1	1 0.999999999999999
2	21.0
3	31.000000000000001

## A SELECT statement that searches for rounded values

```
SELECT * FROM float_sample  
WHERE ROUND(float_value, 2) = 1
```

### The result set

FLOAT_ID	FLOAT_VALUE
1	1 0.999999999999999
2	21.0
3	31.000000000000001

## Description

- Because the values of floating-point numbers aren't always exact, you'll want to search for approximate values when you retrieve floating-point numbers. To do that, you can specify a range of values, or you can use the ROUND function to search for rounded values.

Figure 8-14 How to search for floating-point numbers

## How to work with date/time data

---

In the topics that follow, you'll learn how to use some of the functions that Oracle provides for working with dates and times. As you'll see, these include functions for extracting different parts of a date/time value and for performing operations on dates and times.

### How to use the common date/time functions

---

Figure 8-15 shows how the date/time functions work and how the plus and minus signs can be used to work with dates. If you study the summaries and examples, you shouldn't have much trouble using these functions.

To get the current date and time, you can use the SYSDATE or CURRENT\_DATE function, although the SYSDATE function is more commonly used. In most cases, both of these functions return the same value. However, if a session time zone has been set, the value returned by the CURRENT\_DATE function will be adjusted to accommodate that time zone.

When you use the ROUND and TRUNC functions without a date format, the default is to round or truncate to a date at 00:00:00 on a 24-hour clock. But when you specify a date format like MI for minutes, the date/time value is rounded or truncated to that time component.

When you use the MONTHS\_BETWEEN function, it returns a whole number of months when the day component is the same for both date arguments. However, if the day component isn't the same for both date arguments, this function returns a decimal number.

When you use the ADD\_MONTHS function, you can specify a positive value to add months to the specified date or a negative number to subtract months from the specified date. If necessary, the function will increase or decrease the year component.

When you use the NEXT\_DAY function, you specify a day of the week as the second argument. Then, the function returns the next day of the week that comes after the specified date. For instance, the first example of this function returns the first Friday after August 15, 2014, which is August 22, and the second example returns the first Thursday after August 15, 2014, which is August 21. Note that you can use full or abbreviated names for the second argument.

When you use the addition (+) and subtraction (-) operators, you can add a specific number of days to a date or subtract a specific number of days from a date. You can also use the subtraction operator to subtract one date from another. If the first date comes after the second date, a positive integer is returned. Otherwise, a negative integer is returned.

## Some common date/time functions

Function	Description
<code>SYSDATE</code>	Returns the current local date and time based on the operating system's clock.
<code>CURRENT_DATE</code>	Returns the local date and time adjusted for the current session time zone.
<code>ROUND(date[, date_format])</code>	Returns the date rounded to the unit specified by the date format. If the format is omitted, rounds to the nearest day.
<code>TRUNC(date[, date_format])</code>	Works like the ROUND function but truncates the date.
<code>MONTHS_BETWEEN(date1, date2)</code>	Returns the number of months between date1 and date2.
<code>ADD_MONTHS(date, integer_months)</code>	Adds the specified number of months to the specified date and returns the resulting date.
<code>LAST_DAY(date)</code>	Returns the date for last day of the month for the specified date.
<code>NEXT_DAY(date, day_of_week)</code>	Returns the date for the next day of the week that comes after the specified date.

## Two operators for working with dates

Operator	Description
<code>+</code>	Adds the specified number of days to a date.
<code>-</code>	Subtracts a specified number of days from a date. Or, subtracts one date from another and returns the number of days between the two dates.

## Examples that use the date/time functions

Example	Result
<code>SYSDATE</code>	19-AUG-14 04:20:36 PM
<code>ROUND(SYSDATE)</code>	20-AUG-14 12:00:00 AM
<code>TRUNC(SYSDATE, 'MI')</code>	19-AUG-14 04:20:00 PM
<code>MONTHS_BETWEEN('01-SEP-14', '01-AUG-14')</code>	1
<code>MONTHS_BETWEEN('15-SEP-14', '01-AUG-14')</code>	1.4516129032258064...
<code>ADD_MONTHS('19-AUG-14', -1)</code>	19-JUL-14
<code>ADD_MONTHS('19-AUG-14', 11)</code>	19-JUL-15
<code>LAST_DAY('15-FEB-14')</code>	29-FEB-14
<code>NEXT_DAY('15-AUG-14', 'FRIDAY')</code>	22-AUG-14
<code>NEXT_DAY('15-AUG-14', 'THURS')</code>	21-AUG-14
<code>SYSDATE - 1</code>	18-AUG-14
<code>SYSDATE + 7</code>	26-AUG-14
<code>SYSDATE - TO_DATE('01-JAN-14')</code>	231
<code>TO_DATE('01-JAN-14') - SYSDATE</code>	-231

Figure 8-15 How to use the common date/time functions

## How to parse dates and times

---

Figure 8-16 shows you how to use the TO\_CHAR function to return the various parts of a DATE value as a string. To do that, you specify the appropriate date format element for the part of the DATE value that you want to return. For example, the “MONTH” element returns a string that spells out the month in uppercase, the “Month” element spells out the month with an initial cap, and the “MM” element returns a string that contains the number for the month.

If you need to get an integer value for part of the date, you can use the TO\_CHAR function with the appropriate date format element to return a string that contains numeric characters. Then, you can use the TO\_NUMBER function to convert that string to an integer value. For example, you can use the TO\_CHAR function with the “MM” element to return a string that contains the number for the month. Then, you can use the TO\_NUMBER function to convert the string that’s returned to an integer value.

## Examples that parse a date/time value

Example	Result
TO_CHAR(SYSDATE, 'DD-MON-RR HH:MI:SS')	19-AUG-14 04:20:36 PM
TO_CHAR(SYSDATE, 'YEAR')	TWO THOUSAND EIGHT
TO_CHAR(SYSDATE, 'Year')	Two Thousand Eight
TO_CHAR(SYSDATE, 'YYYY')	2014
TO_CHAR(SYSDATE, 'YY')	14
TO_CHAR(SYSDATE, 'MONTH')	AUGUST
TO_CHAR(SYSDATE, 'MON')	AUG
TO_CHAR(SYSDATE, 'MM')	14
TO_CHAR(SYSDATE, 'DD')	19
TO_CHAR(SYSDATE, 'DAY')	TUESDAY
TO_CHAR(SYSDATE, 'DY')	TUES
TO_CHAR(SYSDATE, 'HH24')	16
TO_CHAR(SYSDATE, 'HH')	04
TO_CHAR(SYSDATE, 'MI')	20
TO_CHAR(SYSDATE, 'SS')	36
TO_CHAR(SYSDATE, 'CC')	21
TO_CHAR(SYSDATE, 'Q')	3
TO_CHAR(SYSDATE, 'WW')	34
TO_CHAR(SYSDATE, 'W')	3
TO_CHAR(SYSDATE, 'DDD')	232
TO_CHAR(SYSDATE, 'D')	3

## How to convert a date component to a numeric value

Example	Result
TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'))	16
TO_NUMBER(TO_CHAR(SYSDATE, 'HH'))	4
TO_NUMBER(TO_CHAR(SYSDATE, 'SS'))	36

## Description

- You can use the TO\_CHAR method to retrieve any part of a date as a string.
- If you need to perform mathematical operations on a part of a date, you can use the TO\_NUMBER function to convert any numeric part of a date to a NUMBER value.

## How to perform a date search

---

Because date/time values always contain both a date and a time component, you may need to ignore the time component when you search for a date value. In figure 8-17, for example the Date\_Sample table has a date\_id column defined with the NUMBER type and a start\_date column defined with the DATE type. In this table, the time components in the first three rows have a zero value. In contrast, the time components in the next three rows have non-zero time components.

The first SELECT statement shows a typical problem that you can encounter when searching for dates. Here, the statement searches for rows in the Date\_Sample table that have a date value of “28-FEB-06”. Since a time component isn’t specified by the date literal in the WHERE clause, a default time component of 00:00:00 is added to the date when it is converted to a DATE value. However, because the one row with this date has a time that isn’t 00:00:00, no rows are returned by this statement.

To solve this problem, you can search for a range of dates that includes only the dates you’re looking for as shown by the second SELECT statement in this figure. In this statement, the search is for any date greater than or equal to February 28, 2006, and less than March 1, 2006. As a result, this search will find any date on February 28, no matter what the time component is.

Another alternative is to use the TRUNC function to truncate the time component from the date/time values as shown in the third SELECT statement in this figure. In this statement, the time components of the start dates are set to 00:00:00, which is the same as the time component for the date literal.

A third approach to this problem is to extract the month, day, and year components of each date in the WHERE clause using the techniques in the previous figure. Then, the condition in the WHERE clause can search for the right month and day and year values. Usually, though, you won’t need this technique because the techniques in this figure are easier to use.

## The Date\_Sample table

DATE_ID	START_DATE
1	1 01-MAR-79
2	2 28-FEB-99
3	3 31-OCT-03
4	4 28-FEB-05
5	5 28-FEB-06
6	6 01-MAR-06

## A SELECT statement that fails to return a row

```
SELECT * FROM date_sample  
WHERE start_date = '28-FEB-06'
```

## A SELECT statement that searches for a range of dates

```
SELECT * FROM date_sample  
WHERE start_date >= '28-FEB-06' AND start_date < '01-MAR-06'
```

### The result set

DATE_ID	START_DATE
1	5 28-FEB-06

## A SELECT statement that uses the TRUNC function to remove time values

```
SELECT * FROM date_sample  
WHERE TRUNC(start_date) = '28-FEB-06'
```

### The result set

DATE_ID	START_DATE
1	5 28-FEB-06

## Description

- If you perform a search using a date string that doesn't include the time, the date string is converted implicitly to a date/time value of 12:00:00 AM (midnight) or 00:00:00 on a 24-hour clock. Then, if the date columns you're searching have time values other than 12:00:00 AM, you have to accommodate the times in the search condition.
- You can accommodate non-zero time components by searching for a range of dates rather than specific dates or by using the TRUNC function to remove the time component.

## How to perform a time search

---

When you search for a time value without specifying a date component, Oracle automatically uses a default date of January 1 for the current year. That's why the first SELECT statement in figure 8-18 doesn't return any rows. In other words, even though one row has the correct time value in its search condition, that row doesn't have the correct date value.

The second SELECT statement in this figure shows how you can solve this problem. Here, the search condition in this statement uses the TO\_CHAR function to convert the date/time values in the start\_date column to string values that don't contain a date component. To do that, it uses a date format element of "HH24:MI:SS". That way, the string that's returned will match the string literal for the time that's specified in the search condition.

Like the first SELECT statement, the third SELECT statement doesn't return any rows. The problem, of course, is the same as in the first statement. To solve this problem, you can use the TO\_CHAR function to remove the date component as shown in the fourth SELECT statement. Once you do that, this SELECT statement will return all rows that have a time within the range that's specified in the search condition.

## The Date\_Sample table

DATE_ID	START_DATE
1	1 01-MAR-79
2	2 28-FEB-99
3	3 31-OCT-03
4	4 28-FEB-05
5	5 28-FEB-06
6	6 01-MAR-06

## A SELECT statement that fails to return a row

```
SELECT * FROM date_sample
WHERE start_date = TO_DATE('10:00:00', 'HH24:MI:SS')
```

## A SELECT statement that ignores the date component

```
SELECT * FROM date_sample
WHERE TO_CHAR(start_date, 'HH24:MI:SS') = '10:00:00'
```

### The result set

DATE_ID	START_DATE
1	4 28-FEB-05

## Another SELECT statement that fails to return a row

```
SELECT * FROM date_sample
WHERE start_date >= TO_DATE('09:00:00', 'HH24:MI:SS')
  AND start_date < TO_DATE('12:59:59', 'HH24:MI:SS')
```

## Another SELECT statement that ignores the date component

```
SELECT * FROM date_sample
WHERE TO_CHAR(start_date, 'HH24:MI:SS') >= '09:00:00'
  AND TO_CHAR(start_date, 'HH24:MI:SS') < '12:59:59'
```

### The result set

DATE_ID	START_DATE
1	4 28-FEB-05
2	6 01-MAR-06

## Description

- If you use the TO\_DATE function to return a date that only contains a time component, the date is converted implicitly to the first day of the current year. For 2012, for example, the date is converted to January 1, 2012. Then, if the date columns you're searching have dates other than that date, you have to accommodate those dates in the search condition.
- To ignore the date component of a date/time value, you can convert the DATE type to a CHAR type and only return the part of the time value that you want to use.

## Other functions you should know about

---

In addition to the functions presented so far in this chapter, Oracle provides some other general purpose functions that you should know about.

### How to use the CASE function

---

Figure 8-19 presents the two formats of the CASE function. This function returns a value that's determined by the conditions you specify. The two examples in this figure show how this function works.

The first example uses a simple CASE function. When you use this function, Oracle compares the input expression you code in the CASE clause with the expressions you code in the WHEN clauses. In this example, the input expression is the value in the terms\_id column of the Invoices table, and the WHEN expressions are the valid values for this column. When Oracle finds a WHEN expression that's equal to the input expression, it returns the expression specified in the matching THEN clause. If the value of the terms\_id column is 3, for example, this function returns the value "Net due 30 days." Although it's not shown in this example, you can also code an ELSE clause at the end of the CASE function. Then, if none of the when expressions are equal to the input expression, the function returns the value specified in the ELSE clause.

The simple CASE function is typically used with columns that can contain a limited number of values, such as the terms\_id column used in this example. In contrast, the searched CASE function can be used for a wide variety of purposes. For example, you can test for conditions other than equal with this function. In addition, each condition can be based on a different column or expression. The second example in this figure shows how this function works.

This example determines the status of the invoices in the Invoices table. To do that, the searched CASE function determines the number of days between the current date and the invoice due date. If the difference is greater than 30, the CASE function returns the value "Over 30 days past due." Similarly, if the difference is greater than 0, the function returns the value "1 to 30 days past due." Note that if an invoice is 45 days old, both of these conditions are true. In that case, the function returns the expression associated with the first condition since this condition is evaluated first. In other words, the sequence of the conditions is critical to getting logical results. If neither of the conditions is true, the function uses the ELSE clause to return a value "Current."

## The syntax of the simple CASE expression

```
CASE input_expression
    WHEN when_expression_1 THEN result_expression_1
    [WHEN when_expression_2 THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

## The syntax of the searched CASE expression

```
CASE
    WHEN conditional_expression_1 THEN result_expression_1
    [WHEN conditional_expression_2 THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

## A SELECT statement that uses a simple CASE expression

```
SELECT invoice_number, terms_id,
CASE terms_id
    WHEN 1 THEN 'Net due 10 days'
    WHEN 2 THEN 'Net due 20 days'
    WHEN 3 THEN 'Net due 30 days'
    WHEN 4 THEN 'Net due 60 days'
    WHEN 5 THEN 'Net due 90 days'
END AS terms
FROM invoices
```

### The result set

INVOICE_NUMBER	TERMS_ID	TERMS
1 QP58872	1	Net due 10 days
2 Q545443	1	Net due 10 days
3 P-0608	5	Net due 90 days

## A SELECT statement that uses a searched CASE expression

```
SELECT invoice_number, invoice_total, invoice_date, invoice_due_date,
CASE
    WHEN (SYSDATE - invoice_due_date) > 30 THEN 'Over 30 days past due'
    WHEN (SYSDATE - invoice_due_date) > 0 THEN '1 to 30 days past due'
    ELSE 'Current'
END AS status
FROM invoices
WHERE invoice_total - payment_total - credit_total > 0
```

### The result set

INVOICE_NUMBER	INVOICE_TOTAL	INVOICE_DATE	INVOICE_DUE_DATE	STATUS
37 547481328	224	20-MAY-08	25-JUN-08	Over 30 days past due
38 40318	21842	18-JUL-08	20-JUL-08	Over 30 days past due
39 31361833	579.42	23-MAY-08	09-JUN-08	Over 30 days past due
40 456789	8344.5	01-AUG-14	31-AUG-14	Current

### Note

- The last example assumes the SYSDATE function returns July 18, 2014.

## How to use the COALESCE, NVL, and NVL2 functions

---

Figure 8-20 presents three functions that you can use to substitute non-null values for null values: COALESCE, NVL, and NVL2. Although these functions are similar, COALESCE is the most flexible because it lets you specify a list of values. Then, it returns the first non-null value in the list. In contrast, the NVL and NVL2 functions aren't as flexible since they only let you substitute a single non-null value for a null value. Still, these functions are useful in some situations.

The first example uses the COALESCE function to return the value of the payment\_date column if that column doesn't contain a null value. Otherwise, it returns the value of the invoice\_due\_date column if that column doesn't contain a null value. Otherwise, it returns a date of January 1, 1900. Note that all of the arguments for this function must be of the same date type. In this case, all three arguments are of the DATE type.

The second example performs a similar task using the NVL function. In this example, the NVL function substitutes a string value of "Unpaid" for a null payment date. Since both arguments must be of the same data type, this example uses the TO\_CHAR function to convert the payment\_date column to a string.

The third example performs a similar task using the NVL2 function. In this example, the NVL2 function substitutes a string value of "Unpaid" for a null payment date, and it substitutes a string value of "Paid" for a non-null payment date. With the NVL2 function, only the second and third arguments need to be of the same data type. As a result, in this example, it isn't necessary to convert the first argument to a string.

## The syntax of the COALESCE, NVL, and NVL2 functions

```
COALESCE(expression1 [, expression2][, expression3]...)
NVL(expression, null_replacement)
NVL2(expression, not_null_replacement, null_replacement)
```

## A SELECT statement that uses the COALESCE function

```
SELECT payment_date, invoice_due_date,
       COALESCE(payment_date, invoice_due_date, TO_DATE('01-JAN-1900'))
          AS payment_date_2
FROM invoices
```

### The result set

	PAYMENT_DATE	INVOICE_DUE_DATE	PAYMENT_DATE_2
1	11-APR-08	22-APR-08	11-APR-08
2	14-MAY-08	23-MAY-08	14-MAY-08
3	(null)	30-JUN-08	30-JUN-08
4	12-MAY-08	16-MAY-08	12-MAY-08
5	13-MAY-08	16-MAY-08	13-MAY-08
6	(null)	26-JUN-08	26-JUN-08

## A SELECT statement that uses the NVL function

```
SELECT payment_date,
       NVL(TO_CHAR(payment_date), 'Unpaid') AS payment_date_2
FROM invoices
```

### The result set

	PAYMENT_DATE	PAYMENT_DATE_2
1	11-APR-08	11-APR-08
2	14-MAY-08	14-MAY-08
3	(null)	Unpaid
4	12-MAY-08	12-MAY-08

## A SELECT statement that uses the NVL2 function

```
SELECT payment_date,
       NVL2(payment_date, 'Paid', 'Unpaid') AS payment_date_2
FROM invoices
```

### The result set

	PAYMENT_DATE	PAYMENT_DATE_2
1	11-APR-08	Paid
2	14-MAY-08	Paid
3	(null)	Unpaid
4	12-MAY-08	Paid

## Description

- The COALESCE, NVL, and NVL2 functions let you substitute non-null values for null values.
- The COALESCE function returns the first expression in a list of expressions that isn't null. All of the expressions in the list must have the same data type. If all of the expressions are null, this function returns a null value.

Figure 8-20 How to use the COALESCE, NVL, and NVL2 functions

## How to use the GROUPING function

---

In chapter 5, you learned how to use the ROLLUP and CUBE operators to add summary rows to a summary query. When you do that, a null value is assigned to any column in a summary row that isn't being summarized.

If you want to assign a value other than null to these columns, you can do that by using the GROUPING function, as illustrated in figure 8-21. This function accepts the name of a column as its argument. The column you specify must be one of the columns named in a GROUP BY clause that includes the ROLLUP or CUBE operator.

The example in this figure shows how you can use the GROUPING function in a summary query that summarizes vendors by state and city. This is the same summary query that was described in chapter 5. However, instead of simply retrieving the values of the vendor\_state and vendor\_city columns from the base table, this query uses the GROUPING function within a CASE function to determine the values that are assigned to those columns. If a row is added to summarize the vendor\_state column, for example, the value of the GROUPING function for that column is 1. Then, the CASE function assigns a literal string value of “=====” to that column. Otherwise, it retrieves the value of the column from the Vendors table. Similarly, if a row is added to summarize the vendor\_city column, a literal string value of “=====” is assigned to that column. As you can see in the result set shown here, this makes it more obvious which columns are being summarized.

This technique is particularly useful if the columns you're summarizing can contain null values. In that case, it would be difficult to determine which rows are summary rows and which rows simply contain null values. Then, you may not only want to use the GROUPING function to replace the null values in summary rows, but you may want to use the COALESCE or NVL function to replace null values retrieved from the base table.

## The syntax of the GROUPING function

```
GROUPING(column_name)
```

## A summary query that uses the GROUPING function

```
SELECT
    CASE
        WHEN GROUPING(vendor_state) = 1 THEN '====='
        ELSE vendor_state
    END AS vendor_state,
    CASE
        WHEN GROUPING(vendor_city) = 1 THEN '====='
        ELSE vendor_city
    END AS vendor_city,
    COUNT(*) AS qty_vendors
FROM vendors
WHERE vendor_state IN ('IA', 'NJ')
GROUP BY ROLLUP(vendor_state, vendor_city)
ORDER BY vendor_state DESC, vendor_city DESC
```

### The result set

VENDOR_STATE	VENDOR_CITY	QTY_VENDORS
1 NJ	Washington	1
2 NJ	Fairfield	1
3 NJ	East Brunswick	2
4 NJ	=====	4
5 IA	Washington	1
6 IA	Fairfield	1
7 IA	=====	2
8 =====	=====	6

## Description

- You can use the GROUPING function to determine when a null value is assigned to a column as the result of the ROLLUP or CUBE operator. The column you name in this function must be one of the columns named in the GROUP BY clause.
- If a null value is assigned to the specified column as the result of the ROLLUP or CUBE operator, the GROUPING function returns a value of 1. Otherwise, it returns a value of 0.
- You typically use the GROUPING function with the CASE expression. Then, if the GROUPING function returns a value of 1, you can assign a value other than null to the column.

---

Figure 8-21 How to use the GROUPING function

## How to use the ranking functions

---

Figure 8-22 shows how to use the Oracle *ranking functions*. These functions provide a variety of ways that you can rank the rows that are returned by a result set. All four of these functions have a similar syntax and work similarly.

The first example shows how to use the ROW\_NUMBER function. Here, the SELECT statement retrieves two columns from the Vendors table. The first column uses the ROW\_NUMBER function to sort the result set by vendor\_name and to number each row in the result set. To show that the first column has been sorted and numbered correctly, the second column displays the vendor\_name.

To accomplish the sorting and numbering, you code the name of the ROW\_NUMBER function, followed by a set of parentheses, followed by the OVER keyword and a second set of parentheses. Within the second set of parentheses, you code the required ORDER BY clause that specifies the sort order. In this example, for instance, the ORDER BY clause sorts by vendor\_name in ascending order. However, you can code more complex ORDER BY clauses if necessary. In addition, you can code an ORDER BY clause that applies to the entire result set. In that case, the ORDER BY clause within the ranking function is used to number the rows and the ORDER BY clause outside the ranking function is used to sort the rows after the numbering has been applied.

The second example shows how to use the optional PARTITION BY clause of a ranking function. This clause allows you to specify a column that's used to divide the result set into groups. In this example, for instance, the PARTITION BY clause uses a column within the Vendors table to group vendors by state and to sort these vendors by name within each state.

However, you can also use the PARTITION BY clause when a SELECT statement joins one or more tables, like this:

```
SELECT vendor_name, invoice_number,
       ROW_NUMBER() OVER(PARTITION BY vendor_name
                          ORDER BY invoice_number) AS row_number
  FROM vendors JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
```

Here, the invoices will be grouped by vendor and sorted within each vendor by invoice number. As a result, if a vendor has three invoices, these invoices will be sorted by invoice number and numbered from 1 to 3.

## The syntax for the four ranking functions

```
ROW_NUMBER()          OVER ([partition_by_clause] order_by_clause)
RANK()               OVER ([partition_by_clause] order_by_clause)
DENSE_RANK()         OVER ([partition_by_clause] order_by_clause)
NTILE(integer_expression) OVER ([partition_by_clause] order_by_clause)
```

## A query that uses the ROW\_NUMBER function

```
SELECT ROW_NUMBER() OVER(ORDER BY vendor_name) AS row_number, vendor_name
FROM vendors
```

### The result set

ROW_NUMBER	VENDOR_NAME
1	1 ASC Signs
2	2 AT&T
3	3 Abbey Office Furnishings
4	4 American Booksellers Assoc
5	5 American Express

## A query that uses the PARTITION BY clause

```
SELECT ROW_NUMBER()
      OVER(PARTITION BY vendor_state ORDER BY vendor_name)
      AS row_number, vendor_name, vendor_state
FROM vendors
```

### The result set

ROW_NUMBER	VENDOR_NAME	VENDOR_STATE
1	1 AT&T	AZ
2	2 Computer Library	AZ
3	3 Wells Fargo Bank	AZ
4	1 ASC Signs	CA
5	2 Abbey Office Furnishings	CA
6	3 American Express	CA

## Description

- The ROW\_NUMBER, RANK, DENSE\_RANK, and NTILE functions are known as *ranking functions*.
- The ROW\_NUMBER function returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition.
- The ORDER BY clause of a ranking function specifies the sort order in which the ranking function is applied.
- The optional PARTITION BY clause of a ranking function specifies the column that's used to divide the result set into groups.

Figure 8-22 How to use the ranking functions (part 1 of 2)

The third example shows how the RANK and DENSE\_RANK functions work. You can use these functions to rank the rows in a result set. In this example, both the RANK and the DENSE\_RANK functions sort all invoices in the Invoices table by the invoice total. Since the first three rows have the same invoice total, both of these functions give these three rows the same rank, 1. However, the fourth row has a different value. To calculate the value for this row, the RANK function adds 1 to the total number of previous rows. In other words, since the first three rows are tied for first place, the fourth row gets fourth place and is assigned a rank of 4.

The DENSE\_RANK function, on the other hand, calculates the value for the fourth row by adding 1 to the rank for the previous row. As a result, this function assigns a rank of 2 to the fourth row. In other words, since the first three rows are tied for first place, the fourth row gets second place.

The fourth example shows how the NTILE function works. You can use this function to divide the rows in a partition into the specified number of groups. When the rows can be evenly divided into groups, this function is easy to understand. For example, if a result set returns 100 rows, you can use the NTILE function to divide this result set into 10 groups of 10. However, when the rows can't be evenly divided into groups, this function is a little more difficult to understand. In this figure, for example, the NTILE function is used to divide a result set that contains 5 rows. Here, the first NTILE function divides this result into 2 groups with the first having 3 rows and the second having 2 rows. The second NTILE function divides this result set into 3 groups with the first having 2 rows, the second having 2 rows, and the third having 1 row. And so on. Although this doesn't result in groups with even numbers of rows, the NTILE function creates the number of groups specified by its argument.

In this figure, the examples for the RANK, DENSE\_RANK, and NTILE functions don't include PARTITION BY clauses. As a result, these functions are applied to the entire result set. However, whenever necessary, you can use the PARTITION BY clause to divide the result set into groups just as shown in the second example for the ROW\_NUMBER function.

## A query that uses the RANK and DENSE\_RANK functions

```
SELECT RANK() OVER (ORDER BY invoice_total) AS rank,
       DENSE_RANK() OVER (ORDER BY invoice_total) AS dense_rank,
       invoice_total, invoice_number
  FROM invoices
```

### The result set

RANK	DENSE_RANK	INVOICE_TOTAL	INVOICE_NUMBER
1	1	1	6 25022117
2	1	1	6 24863706
3	1	1	6 24780512
4	4	2	9.95 21-4748363
5	4	2	9.95 21-4923721
6	6	3	10 4-342-8069

## Description

- The RANK and DENSE\_RANK functions both return the rank of each row within the partition of a result set.
- If there is a tie, both of these functions give the same rank to all rows that are tied.
- To determine the rank for the next distinct row, the RANK function adds 1 to the total number of rows, while the DENSE\_RANK function adds 1 to the rank for the previous row.

## A query that uses the NTILE function

```
SELECT terms_description,
       NTILE(2) OVER (ORDER BY terms_id) AS tile2,
       NTILE(3) OVER (ORDER BY terms_id) AS tile3,
       NTILE(4) OVER (ORDER BY terms_id) AS tile4
  FROM terms
```

### The result set

TERMS_DESCRIPTION	TILE2	TILE3	TILE4
1 Net due 10 days	1	1	1
2 Net due 20 days	1	1	1
3 Net due 30 days	1	2	2
4 Net due 60 days	2	2	3
5 Net due 90 days	2	3	4

## Description

- The NTILE function divides the rows in a partition into the specified number of groups.
- If the rows can't be evenly divided into groups, the later groups may have one less row than the earlier groups.

## Perspective

---

In this chapter, you learned about the most common Oracle data types. You also learned how to use the common functions for working with strings, numbers, and dates. At this point, you have the essential skills you need to write SQL statements at a professional level.

However, there's more to learn about Oracle data types and functions. In particular, chapter 17 shows how to work with other temporal types, and chapter 18 shows how to work with the large object (LOB) types. From that point on, you should be able to use the Oracle Database SQL Reference manual to learn about the data types and functions that aren't presented in this book.

## Terms

---

data type  
character data type  
string data type  
text data type  
numeric data type  
integer  
floating-point number  
temporal data type  
date/time data type  
datetime data type  
date data type  
large object (LOB) data type  
rowid data type

ASCII character  
Unicode character  
national character  
fixed-length string  
variable-length string  
precision  
scale  
scientific notation  
single-precision number  
double-precision number  
casting  
ranking functions

## Exercises

1. Write a SELECT statement that returns these columns from the *Invoices* table:

The *invoice\_total* column

Use the *TO\_CHAR* function to return the *invoice\_total* column with 2 digits to the right of the decimal point.

Use the *TO\_CHAR* function to return the *invoice\_total* column with no digits to the right of the decimal point and no decimal point

Use the *CAST* function to return the *invoice\_total* column as an integer with seven digits

2. Write a SELECT statement that returns these columns from the Invoices table

The invoice\_date column

Use the TO\_CHAR function to return the invoice\_date column with its full date and time including a four-digit year on a 24-hour clock

Use the TO\_CHAR function to return the invoice\_date column with its full date and time including a four-digit year on a 12-hour clock with an am/pm indicator

Use the CAST function to return the invoice\_date column as VARCHAR2(10)

3. Write a SELECT statement that returns these columns from the Vendors table:

The vendor\_name column

The vendor\_name column in all capital letters

The vendor\_phone column

The last four digits of each phone number

When you get that working right, add the columns that follow to the result set. This is more difficult because these columns require the use of functions within functions.

The second word in each vendor name if there is one; otherwise, blanks

The vendor\_phone column with the parts of the number separated by dots as in 555.555.5555

4. Write a SELECT statement that returns these columns from the Invoices table:

The invoice\_number column

The invoice\_date column

The invoice\_date column plus 30 days

The payment\_date column

A column named days\_to\_pay that shows the number of days between the invoice date and the payment date

The number of the invoice\_date's month

The four-digit year of the invoice\_date

The last day of the invoice date's month

When you've got this working, add a WHERE clause that retrieves just the invoices for the month of May based on the invoice date, not the number of the invoice month.

5. Write a SELECT statement that returns these columns from the Invoices table:

The invoice\_number column

The balance due (invoice total minus payment total minus credit total) with commas, a decimal point, and two decimal positions

A column named "Balance Rank" that uses the RANK function to return a column that ranks the balance due in descending order.