**STRUCTURED QUERY LANGUAGE (SQL)**

**Introduction.**

* The **Oracle RDBMS** is available on many different operating system platforms including **Windows** and **UNIX**.
* Oracle is a **relational** DBMS - even the data dictionary is simply a collection of tables of data along with indexes and other objects such as sequences and triggers.
* **SQL** has a basic grammar and syntax.
* The functionally of the SQL language is virtually identical across these operating system platforms.
* Using SQL does not require programming experience, but programming experience can help you conceptualize what a particular SQL command will accomplish.
* The keywords for SQL queries that retrieve data are **SELECT**, **FROM**, **WHERE**, and **ORDER BY**.  Other command options also exist, but these are the basic ones.

**Connecting to the ORACLE Database.**

* Directions for connecting to the **ORACLE** database are given in the web handout, [Getting Started with Oracle](http://www.siue.edu/~dbock/cis564/ostart.htm) that is available on the course web site.

**Naming Difficulties.**

* When building a **schema** (meaning the definition of the tables, indexes, and other objects that comprise a database), it is important to name objects like tables, columns, indexes with **meaningful names**.  A failure to follow this naming rule can result in difficulties for users when they are creating queries.
* Oracle allows object names to be up to 30 characters long.  Make use of this capability.
* Here are some rules to follow.
  + **Don't abbreviate** unless it is necessary. Example, it is better to name a column **WorkerName** than **Wname**.
  + Be **consistent** when abbreviating. Don't use **EmpNo** in one table and **Eno** in another table and **EmpNumber** in still another table.
  + The **Purpose** or **Meaning** of a column or table must be apparent from the name.  What would be the meaning of a table named **Scde** (Storage Code)? A better name would be **STORAGECODE**.
  + If you use **Underscores** as part of a naming convention, then be consistent.  If you are not consistent, then no one can remember when to use them and when not to use them.
  + Be consistent in using **Plurals**.  Is the name of that table "EMP" or "EMPS"?  Is it "NOTE" or "NOTES"?

**SQL BASIC INFORMATION**

**SQL and UNIX Syntax.**

* The **SQL syntax** used in these notes complies with Oracle’s SQL\*PLUS product.
* UNIX syntax used in these notes complies with most UNIX operating systems.
* The American National Standards Institute (ANSI) first published **ANSI SQL standards** in 1986.
* Oracle Corporation - provides **SQL\*PLUS** (an enhanced version of SQL) as part of the Oracle RDBMS product.  This means that SQL\*PLUS has some non-ANSI standard SQL commands that will not run on other RDBMS platforms.
* Microsoft Corporation - provides the SQL Server RDBMS software -- a competitor with Oracle Corporation.

**SQL Standards - Their Characteristics and Benefits.**

* The ANSI SQL Standard provides:
  + Specific syntax and semantics of **SQL data definition** and **datamanipulation languages**.
  + Basic data structures and operations for designing, assessing, maintaining, controlling, and protecting an SQL database.
  + **Portability** of applications and database definition.  Easy to move applications from one machine to another.
  + Minimal standards for adoption among products claiming to be SQL compatible.
  + **Reduced training costs** - IS professionals share a common language.
  + **Productivity** - IS professionals can become proficient in its use.
  + **Application longevity** - A language tends to remain standard for a long time.
  + **Reduced dependence** on a single vendor.

**SQL Data Types.**

* Data stored in a relational database can be stored using a variety of **data types**. The primary ORACLE data types are **NUMBER**, **VARCHAR**, and **CHAR** for storing numbers anda text data; however, there are additional data types that are supported to support backward compatability with products.given below:

|  |  |
| --- | --- |
| **KEY DATA TYPES** |  |
| CHAR(*size*) | Fixed-length character data, *size* characters long. Maximum size=255; default=1 byte. Padded on right with blanks to full length of *size*. |
| DATE | Valid dates range from Jan 1, 4712 B.C. to Dec 31, 4712 A.D. |
| NUMBER | For NUMBER column with space for 40 digits, plus space for a decimal point and sign. Numbers may be expressed in two ways: first, with numbers 0 to 9, the signs + and -, and a decimal point(.); second, in scientific notation, e.g. 1.85E3 for 1850. Valid values are 0 and positive and negative numbers from 1.0E-130 to 9.99…E125. |
| VARCHAR2(*size*) | Variable length character string, maximum *size* up to 2000 bytes. |
| **MISCELLANEOUS DATA TYPES AND VARIATIONS** |  |
| DECIMAL | Same as NUMBER. |
| FLOAT | Same as NUMBER. |
| INTEGER | Same as NUMBER. |
| INTEGER(*size*) | Integer of specified *size* digits wide; same as NUMBER(*size*) of specific size digits wide. |
| LONG | Character data of variable size up to 2Gb in length. **Only one LONG column may be defined per table.** LONG columns may not be used in subqueries, functions, expressions, where clauses, or indexes. A table containing LONG data may not be clustered. |
| LONG RAW | Raw binary data; otherwise the same as LONG (used for images). |
| LONG VARCHAR | Same as LONG |
| NUMBER(*size*) | For NUMBER column of specified *size* in digits. |
| NUMBER(*size,d*) | For NUMBER column of specified *size* with *d* digits after the decimal point, e.g. NUMBER(5,2) could contain nothing larger than 999.99 without an error being generated. |
| NUMBER(\*) | Same as NUMBER. |
| SMALLINT | Same as NUMBER. |
| RAW(*size*) | Raw binary data, *size* bytes long, maximum *size*=255 bytes. |
| ROWID | A value that uniquely identifies a row in an Oracle database - it is returned by the pseudo-column ROWID. Table columns may not be assigned this type. |
| VARCHAR(*size*) | Same as VARCHAR2.  Always use VARCHAR2. |

   
**SQL and SQLPLUS Basics and Error Messages.**

* We have already seen that SQL can be used to **select** data.
* SQL is used for data manipulation in terms of adding, updating, or deleting data.
* The primary verbs for data manipulation are **insert, update**, or **delete**data.
* Taken together, these are the four primary verbs you use in SQL.
* We will start learning to write SQL commands by connecting through a **telnet** session to connect to the **ORACLE**database,  and then by using the SQLPLUS command to enter the **SQLPLUS environment**.
* Start using **SQLPLUS** by typing the command at the prompt SQLPLUS (or by using the appropriate graphical user interface if it is provided). The typical prompt you receive is: **SQL >**
* Sometimes your connection will fail and you will receive an **error message**.
  + If you get the following errors, it may be that access to the **ORACLE** binary files is not available because the **PATH**command for your account is incorrectly established - see your instructor to correct this.
  + The problem could also be that the database has been shutdown and so ORACLE is not running.
  + **ERROR: ORA-1017:**invalid username/password; logon denied.  Unable to CONNECT to ORACLE after 3 attempts, exiting SQL\*Plus
* When you are finished with your work, you quit SQLPLUS with the **QUIT** or **EXIT** command.

**Case in Names and Data.**

* Oracle ignores **case** (upper versus lower case letters) when you type table and column names.  It converts table and column names to uppercase when you type a query and checks the query's validity in the data dictionary.

Data, however, may be stored as a combination of upper and lower case.  You can use Oracle functions to convert data between upper and lower case if that becomes necessary as part of a programming application.

* Case matters only if SQLPLUS is checking a value for equality in the database. For example, the following conditions are **not** identical.
  + **Where Section = 'f'**
  + **Where Section = 'F'**
* Here the values "f"  and "F" inside quotes are called literals, meaning literal values.

**SELECT QUERIES**

**SQL Query Suntax.**

* This section teaches you to write a number of different types of SQL queries.
* SQL is based on the data transform language first named **SEQUEL**.
* This is a relational calculus type of language which provides three capabilities in a very simple syntax.
* The basic syntax as we have already seen is:

**SELECT (column list)**   
**FROM (table list)**   
**WHERE (condition clause)**

**A Sample Query.**

* This result comes from the **ORACLE** database which is in use at SIUE.  This database contains all tables given in the ***Oracle Press: The Complete Reference***, but you do not need to have this book in order to use this set of notes.

**SQL> Select City, Temperature, Humidity from WEATHER**   
**2 where Temperature > 80 and Humidity < 90**   
**3 order by Temperature ;**

**CITY        TEMPERATURE HUMIDITY**   
**----------- ----------- ----------**   
**PARIS       81          62**   
**ATHENS      97          89** 

**Describing a Table.**

* Often it is useful to be able to describe the structure of a table because you may not be familiar with a particular table.
* The **Describe** command is used to accomplish this.  This example gives the description of the **WEATHER** and **LOCATION** tables in the **ORACLE** database.

**SQL> describe weather;**   
**Name            Null?    Type**   
**--------------- -------- ----**   
**CITY                     VARCHAR2(11)**   
**TEMPERATURE              NUMBER**   
**HUMIDITY                 NUMBER**   
**CONDITION                VARCHAR2(9)**

**SQL> describe location;**   
**Name            Null?    Type**   
**--------------- -------- ----**   
**CITY                     VARCHAR2(25)**   
**COUNTRY                  VARCHAR2(25)**   
**CONTINENT                VARCHAR2(25)**   
**LATITUDE                 NUMBER**   
**NORTHSOUTH               CHAR(1)**   
**LONGITUDE                NUMBER**   
**EASTWEST                 CHAR(1)** 

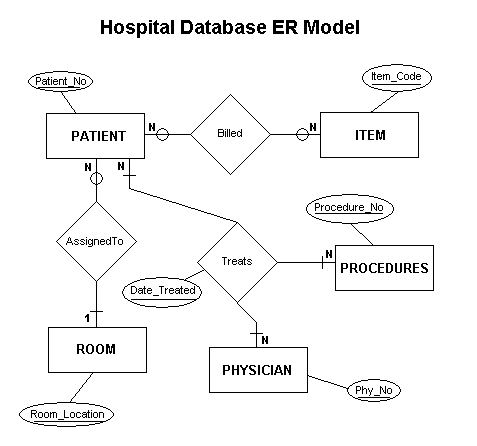
**Result of a SELECT Command.**

* The **SELECT** command produces a **"virtual table"**  -- this is a combination of columns and rows from one or more selected tables given in the query.  This normally produces a subset of the columns and rows that comprise the tables in the query.
* The **virtual table** produced by a query is stored on disk in a **Temporary** tablespace area or in **RAM** that is sometimes called a "**scratch**" space because it is used over and over.  The actual location of the virtual table is a function of the size of the table.
* When the virtual table is no longer needed, Oracle disposes of it.
* The basic operation of each of the three components of the SELECT command are explained below.

|  |  |
| --- | --- |
| SELECT | List the columns including expressions involving columns from base tables or views to be projected into the table that will be the result of the command. |
| FROM | Identifies the tables or views from which columns will be chosen to appear in the result table. This must also list the tables or views needed to join tables to process the query. |
| WHERE | Includes conditions that restrict the rows selected as well as the conditions for joining two or more tables or views. |

**THE COMMUNITY HOSPITAL**   
**EXAMPLE DATABASE**

* The table examples given below assume the database schema represented by the ER diagram given here.  The ER diagram only shows primary key attributes.



|  |  |
| --- | --- |
| **BILLED** |  |
| **BILL\_NO** | **NUMBER(5) - PRI KEY** |
| **PATIENT\_NO** | **NUMBER(9)** |
| **ITEM\_CODE** | **NUMBER(5)** |
| **CHARGE** | **NUMBER(7,2)** |

|  |  |
| --- | --- |
| **TREATS** |  |
| **PHY\_ID** | **NUMBER(4) - PRI KEY** |
| **PATIENT\_NO** | **NUMBER(4) - PRI KEY** |
| **PROCEDURE\_NO** | **NUMBER(4) - PRI KEY** |
| **DATE\_TREATED** | **DATE - PRI KEY** |
| **TREAT\_RESULT** | **VARCHAR2(50)** |

|  |  |
| --- | --- |
| **ITEM** |  |
| **ITEM\_CODE** | **NUMBER(4) - PRI KEY** |
| **DESCRIPTION** | **VARCHAR2(50)** |
| **NORMAL\_CHARGE** | **NUMBER(7,2)** |

|  |  |
| --- | --- |
| **PHYSICIANS** |  |
| **PHY\_ID** | **NUMBER(4) - PRI KEY** |
| **PHY\_PHONE** | **CHAR(8)** |
| **PHY\_NAME** | **VARCHAR2(50)** |

|  |  |
| --- | --- |
| **PATIENT** |  |
| **PATIENT\_NO** | **NUMBER(4) - PRI KEY** |
| **DATE\_LAST\_TREATED** | **DATE** |
| **PAT\_NAME** | **VARCHAR2(50)** |
| **ROOM\_LOCATION** | **CHAR(4)** |

|  |  |
| --- | --- |
| **ROOM** |  |
| **ROOM\_LOCATION** | **CHAR(4) - PRI KEY** |
| **ROOM\_ACCOMODATION** | **CHAR(2)** |
| **ROOM\_EXTENSION** | **NUMBER(4)** |

|  |  |
| --- | --- |
| **PROCEDURES** |  |
| **PROCEDURE\_NO** | **NUMBER(4) - PRI KEY** |
| **PROC\_DESCRIPTION** | **VARCHAR2(50)** |

**EXAMPLE #1 – SIMPLE SELECT TO RETURN A SET OF ROWS BASED ON A SIMPLE SINGLE VALUE IN THE WHERE CLAUSE.**

* Consider the table named **BILLED** as defined above. The command to select the **PATIENT\_NO**, **ITEM\_CODE**, and **CHARGE** and from the **BILLED** table for a specific **PATIENT\_NO** is:

**SELECT patient\_no, item\_code, charge**   
**FROM billed**   
**WHERE patient\_no = 1117;**

**PATIENT\_NO  ITEM\_CODE     CHARGE**   
**---------- ---------- ----------**   
**1117       2222       7.54**   
**1117       2255         25**

* Notice the **SELECT** clause always ends in the semi-colon.

**EXAMPLE #2 – ELIMINATE DUPLICATE ROWS.**

* To eliminate duplicate rows in order to display just distinct rows, the **SELECT DISTINCT** command may be used.  This gives all of the different charges that are stored to the table.

**SELECT DISTINCT charge**   
**FROM billed;**

**CHARGE**   
**----------**   
**2.21**   
**4.56**   
**6.68**   
**7.54**   
**7.75**   
**25**   
**...not all rows are displayed here.**

**EXAMPLE #3 – SIMPLEST QUERY DISPLAYS ALL COLUMNS IN A TABLE.**

* To display all columns from the **BILLED** table use the wildcard **\***.

**SELECT \***   
**FROM billed**   
**WHERE patient\_no = 1116;**

**PATIENT\_NO  ITEM\_CODE     CHARGE**   
**---------- ---------- ----------**   
**1116       2242       4.56**   
**1116       2243       6.68**

**EXAMPLE #4 – QUERY DISPLAYS ALL COLUMNS AND ROWS.**

* To display all columns and all rows from the **BILLED** table eliminate the **WHERE** clause.

**SELECT \***   
**FROM billed ;**

**PATIENT\_NO  ITEM\_CODE     CHARGE**   
**---------- ---------- ----------**   
**1117       2222       7.54**   
**1113       2233       2.21**   
**1117       2255         25**   
**1113       2222       7.75**   
**1115       2245        175**   
**...not all rows are displayed here.**

**ARITHMETIC, LOGICAL, AND OTHER OPERATORS.**

* The various arithmetic, logical, and additional operators are shown in the table below.
* Examples using some of the operators are provided after the table.
* Arithmetic operators are used to qualify expressions in **WHERE** clauses.
* There are operators such as **LIKE** for performing character string pattern matching.
* The **IS NULL** and **IS NOT NULL** operators are used to see if data exists in a row - a column that is completely empty is **NULL**. The word **IS** is required.
* The logical operators **AND**, **OR**, and **NOT** may be used to create complicated **WHERE** clauses. You may use parentheses to group logical operators properly.

|  |  |
| --- | --- |
| **ARITHMETIC OPERATORS** | |
| Page = 6 | Page is equal to 6 |
| Page > 6 | Page is greater than 6 |
| Page >= 6 | Page is greater than or equal to 6 |
| Page < 6 | Page is less than 6 |
| Page <= 6 | Page is less than or equal to 6 |
| Page != 6 or Page ^= 6  or Page <> 6 | Page is not equal to 6 |

|  |  |
| --- | --- |
| **USING THE LIKE OPERATOR** | |
| Feature LIKE ‘Mo%’ | Feature begins with the letters Mo. Note the % wildcard represents any number of spaces or characters. |
| Feature LIKE ‘\_ \_I%’ | Feature has the letter I in the third position. Note the underline (\_) represents one space. |

|  |  |
| --- | --- |
| **USING THE IS NULL and IS NOT NULL OPERATORS** | |
| Precipitation IS NULL | The column named Precipitation contains no value or is unknown. |
| Precipitation IS NOT NULL | The column named Precipitation contains some value or is known. |

|  |  |
| --- | --- |
| **USING THE IN and BETWEEN OPERATORS** | |
| Page IN (1,2,3)Page is in the list (1,2,3) |  |
| Page BETWEEN 6 AND 10 | Page is equal to 6, 10, or anything in between. |
| Section IN (‘A’, ‘B’, ‘F’) | Section is in the list (A, B, or F ) Note use of ‘’ quotes. |

|  |  |
| --- | --- |
| **USING LOGICAL OPERATORS – OR, AND, NOT** | |
| Section = ‘A’ OR Section = ‘B’ | The value of the column Section is either A or B for this to evaluate to TRUE. |
| Section = ‘A’ AND Page > 2 | The value of the column Section must be A and the value of the column Page must be greater than 2 for this to evaluate to TRUE. |
| NOT Section = ‘A’ | Section must have some value other than A for this to evaluate to TRUE. |

**EXAMPLE #5  USES LOGICAL AND OPERATOR.**

* To display all **charges** greater than **$5.00** for the **PATIENT\_NO 1116**.

**SELECT charge**   
**FROM billed**   
**WHERE patient\_no = 1116 AND charge > 5.00;**

**CHARGE**   
**----------**   
**6.68**

**EXAMPLE #6 – USES LOGICAL OR OPERATOR.**

* To display all **charges** for either patient 1116 or patient 1117.

**SELECT patient\_no, charge**   
**FROM billed**   
**WHERE patient\_no = 1116 OR patient\_no = 1117;**

**PATIENT\_NO     CHARGE**   
**---------- ----------**   
**1117       7.54**   
**1117         25**   
**1116       4.56**   
**1116       6.68**   
**1117     167.67**   
**1117     222.21**   
**1117       4.92**

**EXAMPLE #7 – USES COUNT FUNCTION.**

* SQL provides many **built-in functions** (such as **COUNT**, **MAX**, **MIN**, **SUM**, and **AVG**) to be used as operators for columns.
* An example of counting the number of times patient 1116 has been charged for items. This counts the number of rows for the patient.

**SELECT COUNT(\*)**   
**FROM billed**   
**WHERE patient\_no = 1116;**

**COUNT(\*)**   
**---------**   
**2**

**EXAMPLE #8 – DISPLAYS DISTINCT ROWS – ELIMINATES DUPLICATE ROWS ON THE OUTPUT.**

* Suppose you want the number of **DISTINCT** procedures performed on a patient. This counts the number of distinct values that appear in the column named **PROCEDURE\_NO** for the patient.

**SELECT COUNT ( DISTINCT procedure\_no )**   
**FROM treats**   
**WHERE patient\_no = 1113;**

**COUNT(DISTINCTPROCEDURE\_NO)**   
**---------------------------**   
**2**

**EXAMPLE #9 – SPECIFY MEANINGFUL COLUMN NAMES.**

* To "jazz up" the output, you may specify creative **column names** to appear at the top of a column of output.
* The default is to list the column name above the column.

**SELECT COUNT (DISTINCT procedure\_no) AS "No. of Procedures"**   
**FROM treats**   
**WHERE patient\_no = 1113;**

**No. of Procedures**   
**-----------------**   
**2**

**DISPLAYING CONSTANTS AND CALCULATED VALUES**

* Before using data aggregate functions such as **COUNT** or **SUM**, you should understand that you **cannot** mix row-level data output in the same SELECT clause with data aggregate functions.

**EXAMPLE #10.**

* To display a calculated value such as the current charge and the amount that would be charged if the charge were increased by 6% for all rows in the **ITEM** table, the command is:

**SELECT normal\_charge, normal\_charge\*1.06**   
**FROM item;**

**NORMAL\_CHARGE NORMAL\_CHARGE\*1.06**   
**------------- ------------------**   
**7.54             7.9924**   
**2.21             2.3426**   
**4.5               4.77**   
**284.59           301.6654**   
**25               26.5**   
**...not all items are listed here.**

**EXAMPLE #11.**

* To compute how long a patient was in the hospital, you can perform arithmetic operations on both numeric as well as date type data.
* The example below shows the use of date type data to list all patients hospitalized more than 6 days. This also shows an example of using an arithmetic expression in a **WHERE** clause.

**SELECT patient\_no, date\_discharged, date\_admitted,**   
**(date\_discharged - date\_admitted + 1)**   
**As "No. of Days In"**   
**FROM patient**   
**WHERE (date\_discharged - date\_admitted + 1 ) > 6;**

**PATIENT\_NO DATE\_DISC DATE\_ADMI No. of Days In**   
**---------- --------- --------- --------------**   
**1117 22-DEC-01 16-DEC-01              7**   
**1114 24-DEC-01 15-DEC-01             10**   
**1115 25-DEC-01 15-DEC-01             11** 

**SORTING, GROUPING and the BETWEEN OPERATOR**

**EXAMPLE #12 – USING SORTING AND GROUPING.**

* The **ORDER BY** optional clause will sort results in either ascending or descending order.  The optional clause for descending is **DESC**.
* The **GROUP BY** optional clause is used to group rows with common values.
* The example below produces a list of the total charges per patient for expensive medical items (**CHARGE** greater than $100 for an item) where patients owe the hospital a  sum (total charges over $500).
* Also note that we can only include column data such as the **PATIENT\_NO** and aggregate data such as **SUM(CHARGE)**together in a query when we use the **GROUP BY** clause.
* The **BETWEEN** operator is used to simplify query writing for a range of qualifications, for example, when a value is BETWEEN *x* and *y*, using the BETWEEN *x* and *y* is equivalent to saying the value is >=*x* AND =<*y*.

**SELECT patient\_no, SUM(charge)**   
**FROM billed**   
**WHERE (patient\_no BETWEEN 1110 AND 1200) AND**   
**charge > 100**   
**GROUP BY patient\_no**   
**HAVING SUM(charge) > 200**   
**ORDER BY patient\_no;**

**PATIENT\_NO SUM(CHARGE)**   
**---------- -----------**   
**1112      782.55**   
**1117      389.88**

* The **GROUP BY**specifies control breaks for subtotals.
* The **HAVING** clause **must be used** because the group-level data is aggregated.
* The **HAVING** clause is like **WHERE** but must be used when the condition involves group-level data.

**EXAMPLE #13 – MAJOR AND MINOR SORTS.**

* You can sort by more than one variable (major and minor sort).
* The query below shows the results in descending total charge sequence (use of the **DESC** option since the default is ascending), then by ascending **PATIENT\_NO** as the minor sort variable.

**SELECT patient\_no, SUM(charge) total\_charge**   
**FROM billed**   
**WHERE (patient\_no BETWEEN 1110 AND 1200) AND**   
**charge > 100**   
**GROUP BY patient\_no**   
**HAVING SUM(charge) > 50**   
**ORDER BY total\_charge DESC, patient\_no;**

**PATIENT\_NO TOTAL\_CHARGE**   
**---------- ------------**   
**1112       782.55**   
**1117       389.88**   
**1115          175**

**EXAMPLE #14 – ANOTHER BETWEEN EXAMPLE**.

* This is another example of the **BETWEEN** operator and lists all patients who were charged between $160 and $170 for item 2245.
* The second query shows all charges for item 2245.

**SELECT patient\_no, charge**   
**FROM billed**   
**WHERE item\_code = 2245 AND**   
**charge BETWEEN 160 AND 170;**

**PATIENT\_NO     CHARGE**   
**---------- ----------**   
**1112     167.67**   
**1112     167.67**   
**1117     167.67**

**SELECT patient\_no, charge**   
**FROM billed**   
**WHERE item\_code = 2245;**

**PATIENT\_NO     CHARGE**   
**---------- ----------**   
**1115        175**   
**1112     167.67**   
**1112     167.67**   
**1117     167.67**

**THE IN OPERATOR**

* The **IN** operator may be used to replace the **OR** operator to simplify a query.
* Later you will see examples where **IN** must be used for complex queries.

**EXAMPLE #15 – USING THE IN OPERATOR.**

* This example lists the patients who had either Dr. Hawkeye Pierce or Dr. Trapper John or Dr. Jonas Salkman as a physician.

**SELECT DISTINCT patient\_no, phy\_id**   
**FROM treats**   
**WHERE phy\_id IN**   
**(8887, 8886, 8882);**

**PATIENT\_NO     PHY\_ID**   
**---------- ----------**   
**1112       8887**   
**1113       8882**   
**1113       8887**   
**1115       8886**   
**1116       8886**   
**1117       8887** 

* It would be nice to show the patient names (**PAT\_NAME** field) and associated physician names (**PHY\_NAME** field) along with the Patient information.  We shall see how to accomplish this in the next section.

**MULTIPLE TABLE QUERIES**

* SQL can produce data from several related tables by performing either a physical or virtual join of the tables. The default is a **virtual join**.
* The **WHERE** clause is most often used to perform the **JOIN** function where two or more tables have common columns such as is the case for the **PATIENT** and **BILLED** tables.

**EQUI-JOIN and NATURAL JOIN**

* The **Equi-Join** and **Natural Join** are equivalent except that duplicate columns are eliminated in the **Natural Join** that would otherwise appear in the **Equi-Join**.
* Recall the immediate example above where the query listed the **PATIENT\_NO** and **PHY\_ID**, but not the **PAT\_NAME** or **PHY\_NAME** fields.  This query joints the **PATIENT**, **PHYSICIAN**, and **TREATS** tables to produce the desired information.
* The query also demonstrates the use of an **ALIAS** for a table name - here we have three aliases (**PA**, **PH**, and **TR**), one for each table.
* When two or more tables have a column with the **same name**, the name can be **qualified** by using the table name combined with the period (.) in referring to the column, e.g. **billed.patient\_no**.  Note that columns with unique names do **not** have to be qualified by the table name.
* The **COLUMN** commands given prior to the **SELECT** are used to format the output.

**EXAMPLE #16a – JOIN THREE TABLES.**

**COLUMN pat\_name FORMAT A18;**   
**COLUMN phy\_name FORMAT A18;**   
**SELECT DISTINCT PA.patient\_no, pat\_name, PH.phy\_id, phy\_name**   
**FROM patient PA, physician PH, treats TR**   
**WHERE PA.patient\_no = TR.patient\_no AND**   
**PH.phy\_id = TR.phy\_id AND**   
**PH.phy\_id IN (8887, 8886, 8882);**

**PATIENT\_NO PAT\_NAME           PHY\_ID PHY\_NAME**   
**---------- ------------------ ------ ------------------**   
**1112 Eminem               8887 Dr. Hawkeye Pierce**   
**1113 George Marshall      8882 Dr. Jonas Salkman**   
**1113 George Marshall      8887 Dr. Hawkeye Pierce**   
**1115 Barbara Streisand    8886 Dr. Trapper John**   
**1116 G. W. Bush           8886 Dr. Trapper John**   
**1117 Sally Field          8887 Dr. Hawkeye Pierce**

* The next example below shows the **PATIENT\_NO** and **DATE\_DISCHARGED** from the **PATIENT** table and the associated **CHARGE** from the **BILLED** table.
* The sequence of table names in the FROM list is immaterial. If a column is indexed, the DBMS will automatically optimize the use of the index in parsing the query.

**EXAMPLE #16b – JOIN TWO TABLES.**

**SELECT billed.patient\_no, date\_discharged, charge**   
**FROM billed, patient**   
**WHERE patient.patient\_no = billed.patient\_no;**

**PATIENT\_NO DATE\_DISC     CHARGE**   
**---------- --------- ----------**   
**1117 22-DEC-01       7.54**   
**1113 21-DEC-01       2.21**   
**1117 22-DEC-01         25**   
**1113 21-DEC-01       7.75**   
**1115 25-DEC-01        175**   
**1116 22-DEC-01       4.56**   
**...not all of the rows are shown here.**

   
**OUTER JOIN and TABLE NAME ALIASES (ABBREVIATIONS)**

* There may be situations where a row in one table, such as a **PATIENT** table has no corresponding rows in another table such as the **CHARGE** table (the patient has not yet incurred any charges - a most unusual circumstance in any hospital).
* The **JOIN** of these tables will produce rows where some columns have **NULL** values. This is called the **Outer Join**. The Oracle RDBMS supports Outer Joins.

**EXAMPLE #17 – OUTER JOIN and USING ALIAS.**

* The following command might produce the sample table shown after the command.
* This example also shows another example of using "alias" names for tables in order to shorten the query that is written.
* Here the alias for **BILLED** has been shortened to the letter "**B**" and **PATIENT** to "**P**".
* Note the resulting **NULL** cells for some rows.
* The (**+**) for the table for which there will be **NULL** values in terms of matching the **P.PATIENT\_NO** to the **B.PATIENT\_NOis required**.

**SELECT p.patient\_no, b.charge**   
**FROM billed b, patient p**   
**WHERE p.patient\_no = b.patient\_no(+);**

**PATIENT\_NO     CHARGE**   
**---------- ----------**   
**1111**   
**1112        225**   
**1112     167.67**   
**1112     167.67**   
**1112       4.92**   
**1112     222.21**   
**1113       2.21**   
**1113       7.75**   
**1114**   
**1115        175**   
**1116       4.56**   
**...not all rows are shown here.**

* If we only want a list of patients who have no charges yet, the following query with a WHERE clause testing for NULL values will work.  Note that this result is a subset of the above result.

**SELECT p.patient\_no, b.charge**   
**FROM billed b, patient p**   
**WHERE p.patient\_no = b.patient\_no(+) AND**   
**b.charge IS NULL;**

**PATIENT\_NO     CHARGE**   
**---------- ----------**   
**1111**   
**1114** 

**SUBQUERIES**

* The object of a **WHERE** clause can be another **SELECT** query.
* Sometimes this makes it simpler to write a query that would otherwise have a **very complex** **WHERE** clause.
* The two queries shown in the example below are equivalent. Both queries produce a list of **ITEM\_CODEs** and the associated description of the items for a specific patient.  To produce the list the **ITEM** and **BILLED** tables must be joined.

**EXAMPLE #18 – A COMPLEX QUERY AND THE SUBQUERY.**

**SELECT item.item\_code, description**   
**FROM item, billed**   
**WHERE item.item\_code = billed.item\_code AND**   
**patient\_no = 1116;**

**ITEM\_CODE DESCRIPTION**   
**---------- ----------------**   
**2242 4 inch dressing**   
**2243 Syringe, 8 gauge** 

**SELECT item\_code, description**   
**FROM item**   
**WHERE item\_code IN**   
**(SELECT item\_code FROM billed**   
**WHERE patient\_no = 1116);**

**ITEM\_CODE DESCRIPTION**   
**--------- ----------------**   
**2242 4 inch dressing**   
**2243 Syringe, 8 gauge**

**EXAMPLE #19 – USING IN WITH THE SUBQUERY.**

* When an inner query (**subquery**) returns a set or list of values, we can use the **IN** operator to match on equality conditions.
* In fact, the **IN** operator was used for the above as shown here.

**SELECT item\_code, description**   
**FROM item**   
**WHERE item\_code IN**   
**(SELECT item\_code FROM billed**   
**WHERE patient\_no = 1116);**

* There are many other types of subqueries. You can spend many weeks studying the topic of SQL query writing.

**VIEW DEFINITION**

* A **VIEW** is a virtual table that does not exist in reality, but is a **logical definition** of a **set of related columns**, usually from **multiple tables**.
* A **VIEW** presents data to the end user of an application system the way that the end user is used to seeing the data.
  + For example, the **Customer Order Form** is a view of data from several different tables including **CUSTOMER**, **ORDERS**, **PRODUCT**, **ORDERLINE**, and **SALESPERSON**.
* A **VIEW** can also be used to simplify query generation and to add data security to a database by limiting the data that an end user can access.
* A **VIEW** **definition** is permanently stored as part of the database.
* The example below creates a view named **PATIENT\_BILL** that includes the **PATIENT\_NO**, **ITEM\_CODE** and **CHARGE**columns from the **BILLED** table and the **DESCRIPTION** column from the **ITEM** table, and the **DATE\_DISCHARGED** from the **PATIENT** table.

**EXAMPLE #20 – CREATE A VIEW.**   
   
**CREATE VIEW patient\_bill AS**   
**SELECT B.patient\_no, P.pat\_name,**   
**B.item\_code, charge,**   
**description, date\_discharged**   
**FROM patient P, billed B, item I**   
**WHERE P.patient\_no = B.patient\_no AND**   
**I.item\_code = B.item\_code;**   
   
**View created.** 

* Note that the relationship from **PATIENT** to **BILLED** is **1:N** and the relationship from **ITEM** to **BILLED** is **1:N**.
* In other words, the **BILLED** table is the intersection table linking **PATIENT** and **ITEM**.
* Now you can query the view PATIENT\_BILL just as you would a table.  When you execute the query, the view is generated by the DBMS and loaded with data, then your query is executed.

**Column Description Format A22;**   
**SELECT patient\_no, item\_code, charge,**   
**description**   
**FROM patient\_bill**   
**WHERE patient\_no = 1117;**   
   
**PATIENT\_NO ITEM\_CODE   CHARGE DESCRIPTION**   
**---------- --------- -------- ---------------------**   
**1117      2222     7.54 Syringe, 19 gauge**   
**1117      2255       25 Saline Soln, 1 liter**   
**1117      2245   167.67 Surgical Prep Pack #8**   
**1117      2224   222.21 Surgical Prep Pack #4**   
**1117      2267     4.92 Bed Pan** 

**DERIVED COLUMNS**

* A view may contain **derived** (or virtual) columns.
* For example, the total charges by patient for room and special items in a room (item codes between 2200 and 2250).
* A view for this aggregate data can be created from the **PATIENT\_BILL** view that was created in the example above.
* The new view (code shown below for this view of a view) named **ROOM\_CHARGE** has two virtual columns named **OCCUPANT** and **ROOM\_CHGS**.
* Note that **ROOM\_CHARGES** is the sum of the charges for an occupant of the room.

**EXAMPLE #21 – DERIVED COLUMNS.**   
   
**CREATE VIEW room\_charge (occupant, room\_chgs) AS**   
**SELECT pat\_name, sum(CHARGE)**   
**FROM patient\_bill**   
**WHERE item\_code BETWEEN 2200 AND 2250**   
**GROUP BY pat\_name;** 

**EXAMPLE #21 – SELECTING FROM A VIEW.**

* Now you can select charges for an occupant with the simple query given below.

**Column occupant Format A20;**   
**SELECT occupant, room\_chgs**   
**FROM room\_charge**   
**WHERE room\_chgs > 100;**   
   
**OCCUPANT              ROOM\_CHGS**   
**-------------------- ----------**   
**Barbara Streisand           175**   
**Eminem                   782.55**   
**Sally Field              397.42**

 which is equivalent to:

**SELECT pat\_name, SUM(charge)**   
**FROM patient\_bill**   
**WHERE item\_code BETWEEN 2200 AND 2250**   
**GROUP BY pat\_name**   
**HAVING SUM(charge) > 100;**

**PAT\_NAME             SUM(charge)**   
**-------------------- -----------**   
**Barbara Streisand            175**   
**Eminem                    782.55**   
**Sally Field               397.42**

Attached here is the Script file used to create and populate the tables used in this handout.  They files assume the existence of the DATA tablespace and the DATA\_INDEX tablespace for storage of tables and indexes.  The user account assumed is named DBOCK.  If you run the script to create the tables with data in your own Oracle account, replace the name of the user account with your own account name.  Do not attempt to create the PUBLIC SYNONYMS since these already exist for the DBOCK account.

REM  Table ROOM   
        drop table ROOM;   
        create table ROOM (   
          ROOM\_LOCATION  char(4)   
            constraint PK\_ROOM primary key   
              using index tablespace DATA\_INDEX   
              pctfree 5,   
          ROOM\_ACCOMMODATION  char(2)   
            constraint NN\_ACCOMMODATION NOT NULL,   
          ROOM\_EXTENSION      number(4) default NULL   
    )   
        tablespace DATA;

Insert into ROOM values ('GM01', 'P2', 1111);   
Insert into ROOM values ('GM02', 'P2', 1112);   
Insert into ROOM values ('GM03', 'P2', 1113);   
Insert into ROOM values ('GM04', 'P2', 1114);   
Insert into ROOM values ('IC01', 'P1', 2221);   
Insert into ROOM values ('IC02', 'P1', 2222);   
Insert into ROOM values ('IC03', 'P1', 2223);

REM  Table PATIENT   
        drop table PATIENT;   
        create table PATIENT (   
          PATIENT\_NO        number(4)   
            constraint PK\_PATIENT primary key   
              using index tablespace DATA\_INDEX   
              pctfree 5,   
          DATE\_ADMITTED     date  default sysdate   
            constraint NN\_DATE\_ADMITTED NOT NULL,   
          DATE\_DISCHARGED   date default NULL,   
          PAT\_NAME          varchar2(50)   
            constraint NN\_PAT\_NAME NOT NULL,   
          ROOM\_LOCATION     char(4)   
            constraint NN\_ROOM\_LOCATION NOT NULL,   
       constraint FK\_PATIENT\_ROOM foreign key (ROOM\_LOCATION)   
          references ROOM   
          on delete cascade   
    )   
        tablespace DATA;

Insert into PATIENT values (1111, '15-DEC-01', '20-DEC-01', 'Tom Thumb', 'GM03');   
Insert into PATIENT values (1117, '16-DEC-01', '22-DEC-01', 'Sally Field', 'GM01');   
Insert into PATIENT values (1113, '17-DEC-01', '21-DEC-01', 'George Marshall', 'GM03');   
Insert into PATIENT values (1114, '15-DEC-01', '24-DEC-01', 'William Clinton', 'GM02');   
Insert into PATIENT values (1116, '18-DEC-01', '22-DEC-01', 'G. W. Bush', 'IC01');   
Insert into PATIENT values (1115, '15-DEC-01', '25-DEC-01', 'Barbara Streisand', 'IC03');   
Insert into PATIENT values (1112, '18-DEC-01', '20-DEC-01', 'Eminem', 'GM01');

REM  Table ITEM   
        drop table ITEM;   
        create table ITEM (   
          ITEM\_CODE      number(4)   
            constraint PK\_ITEM primary key   
              using index tablespace DATA\_INDEX   
              pctfree 5,   
          DESCRIIPTION   varchar2(50) default NULL,   
          NORMAL\_CHARGE  number(7,2)   
    )   
        tablespace DATA;

Insert into ITEM values (2222, 'Syringe, 19 gauge', 7.54);   
Insert into ITEM values (2233, '9 inch gauze', 2.21);   
Insert into ITEM values (2244, 'Tylenol 800MG', 4.50);   
Insert into ITEM values (2223, 'Morphine Drip, 0.2 liter', 284.59);   
Insert into ITEM values (2255, 'Saline Soln, 1 liter', 25.00);   
Insert into ITEM values (2243, 'Syringe, 8 gauge', 6.68);   
Insert into ITEM values (2267, 'Bed Pan', 4.92);   
Insert into ITEM values (2266, 'Surgical Prep Pack #2', 195.95);   
Insert into ITEM values (2265, '4 inch gauze', 1.19);   
Insert into ITEM values (2245, 'Surgical Prep Pack #8', 167.67);   
Insert into ITEM values (2268, 'Bandaid, 3 inch', 0.76);   
Insert into ITEM values (2242, '4 inch dressing', 4.56);   
Insert into ITEM values (2224, 'Surgical Prep Pack #4', 222.21);

REM  Table PHYSICIAN   
        drop table PHYSICIAN;   
        create table PHYSICIAN (   
          PHY\_ID         number(4)   
            constraint PK\_PHYSICIAN primary key   
              using index tablespace DATA\_INDEX   
              pctfree 5,   
          PHY\_PHONE      char(8) default NULL,   
          PHY\_NAME       varchar2(50)   
            constraint NN\_PHY\_NAME NOT NULL   
    )   
        tablespace DATA;

Insert into PHYSICIAN values (8880, '888-0001', 'Dr. Oprah Heart');   
Insert into PHYSICIAN values (8887, '888-0007', 'Dr. Hawkeye Pierce');   
Insert into PHYSICIAN values (8882, '888-0002', 'Dr. Jonas Salkman');   
Insert into PHYSICIAN values (8884, '888-0004', 'Dr. Winifred Winfield');   
Insert into PHYSICIAN values (8886, '888-0006', 'Dr. Trapper John');

REM  Table PROCEDURES   
        drop table PROCEDURES;   
        create table PROCEDURES (   
          PROCEDURE\_NO       number(4)   
            constraint PK\_PROCEDURES primary key   
              using index tablespace DATA\_INDEX   
              pctfree 5,   
          PROC\_DESCRIPTION  varchar2(50)   
    )   
        tablespace DATA;

Insert into PROCEDURES values (3225, 'Appendectomy');   
Insert into PROCEDURES values (3233, 'Lobotomy');   
Insert into PROCEDURES values (3234, 'Thoracic Procedure');   
Insert into PROCEDURES values (3232, 'Cervical Disk');   
Insert into PROCEDURES values (3230, 'Hip Replacement');   
Insert into PROCEDURES values (3231, 'Arthroscopic Procedure');   
Insert into PROCEDURES values (3237, 'Ingrown Toenail');

REM  Table TREATS   
 drop table TREATS;   
  create table TREATS (   
    PHY\_ID       number(4),   
          PATIENT\_NO   number(4),   
    PROCEDURE\_NO number(5),   
    DATE\_TREATED date,   
          TREAT\_RESULT varchar2(50) default NULL,   
        constraint FK\_TREATS\_PATIENT foreign key (PATIENT\_NO)   
          references PATIENT   
          on delete cascade,   
        constraint FK\_TREATS\_PHYSICIAN foreign key (PHY\_ID)   
          references PHYSICIAN   
          on delete cascade,   
       constraint FK\_TREATS\_PROCEDURES foreign key (PROCEDURE\_NO)   
          references PROCEDURES   
          on delete cascade,   
        constraint PK\_TREATS   
          primary key (PHY\_ID, PATIENT\_NO, PROCEDURE\_NO, DATE\_TREATED)   
          using index tablespace DATA\_INDEX   
          pctfree 5   
   )   
        tablespace DATA;

Insert into TREATS values (8887, 1117, 3232, '17-DEC-01','Repaired Disk OK');   
Insert into TREATS values (8882, 1113, 3230, '18-DEC-01','Repaired Right Hip OK');   
Insert into TREATS values (8887, 1113, 3232, '17-DEC-01','Repaired Disk OK');   
Insert into TREATS values (8884, 1114, 3233, '15-DEC-01','Brain Partially Removed');   
Insert into TREATS values (8886, 1116, 3234, '19-DEC-01','Removed Left Lung');   
Insert into TREATS values (8880, 1111, 3237, '15-DEC-01','Removed toenail');   
Insert into TREATS values (8886, 1115, 3230, '17-DEC-01','Repaired Left Hip OK');   
Insert into TREATS values (8887, 1112, 3231, '18-DEC-01','Repaired Left Knee OK');   
Insert into TREATS values (8887, 1112, 3231, '19-DEC-01','Repaired Right Knee OK');

REM  Table BILLED   
 drop table BILLED;   
  create table BILLED (   
    BILL\_NO    number(5),   
          PATIENT\_NO number(4),   
    ITEM\_CODE  number(5),   
    CHARGE     number(7,2),   
        constraint FK\_BILLED\_PATIENT foreign key (PATIENT\_NO)   
          references PATIENT   
          on delete cascade,   
        constraint FK\_BILLED\_ITEM foreign key (ITEM\_CODE)   
          references ITEM   
          on delete cascade,   
        constraint PK\_BILLED   
          primary key (BILL\_NO)   
          using index tablespace DATA\_INDEX   
          pctfree 5   
   )   
        tablespace DATA;

Insert into BILLED values (00001, 1117, 2222, 7.54);   
Insert into BILLED values (00002, 1113, 2233, 2.21);   
Insert into BILLED values (00003, 1117, 2255, 25.00);   
Insert into BILLED values (00004, 1113, 2222, 7.75);   
Insert into BILLED values (00005, 1115, 2245, 175.00);   
Insert into BILLED values (00006, 1116, 2242, 4.56);   
Insert into BILLED values (00007, 1112, 2224, 225.00);   
Insert into BILLED values (00008, 1112, 2245, 167.67);   
Insert into BILLED values (00009, 1116, 2243, 6.68);   
Insert into BILLED values (00010, 1112, 2245, 167.67);   
Insert into BILLED values (00011, 1112, 2224, 222.21);   
Insert into BILLED values (00012, 1112, 2267, 4.92);   
Insert into BILLED values (00013, 1117, 2245, 167.67);   
Insert into BILLED values (00014, 1117, 2224, 222.21);   
Insert into BILLED values (00015, 1117, 2267, 4.92); 

REM End of Database Table and Data Insertion Code   
REM Now create the public synonyms to allow easy access   
REM by students to the tables.

drop public synonym BILLED;   
create public synonym BILLED for DBOCK.BILLED;   
drop public synonym ROOM;   
create public synonym ROOM for DBOCK.ROOM;   
drop public synonym PATIENT;   
create public synonym PATIENT for DBOCK.PATIENT;   
drop public synonym ITEM;   
create public synonym ITEM for DBOCK.ITEM;   
drop public synonym TREATS;   
create public synonym TREATS for DBOCK.TREATS;   
drop public synonym PHYSICIAN ;   
create public synonym PHYSICIAN for DBOCK.PHYSICIAN;   
drop public synonym PROCEDURES;   
create public synonym PROCEDURES for DBOCK.PROCEDURES; 

REM End of Script