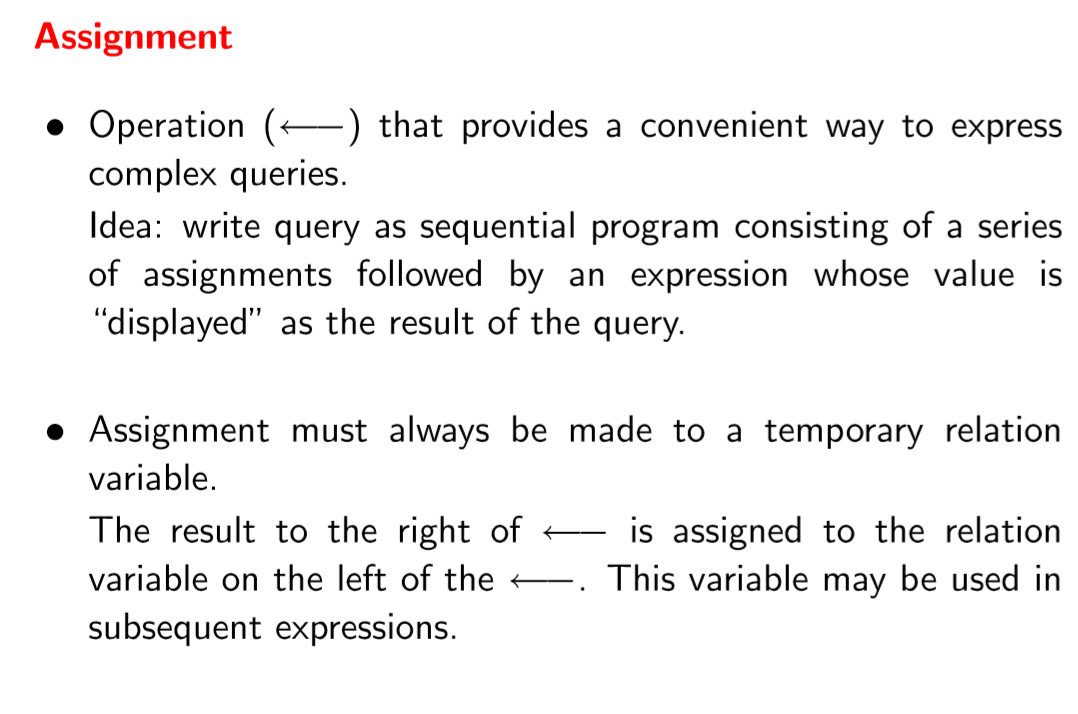
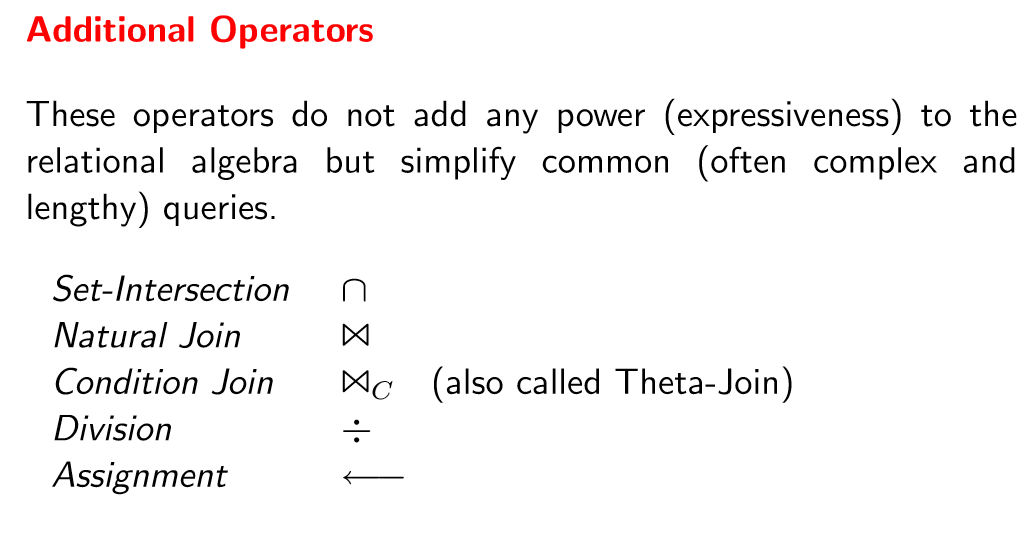
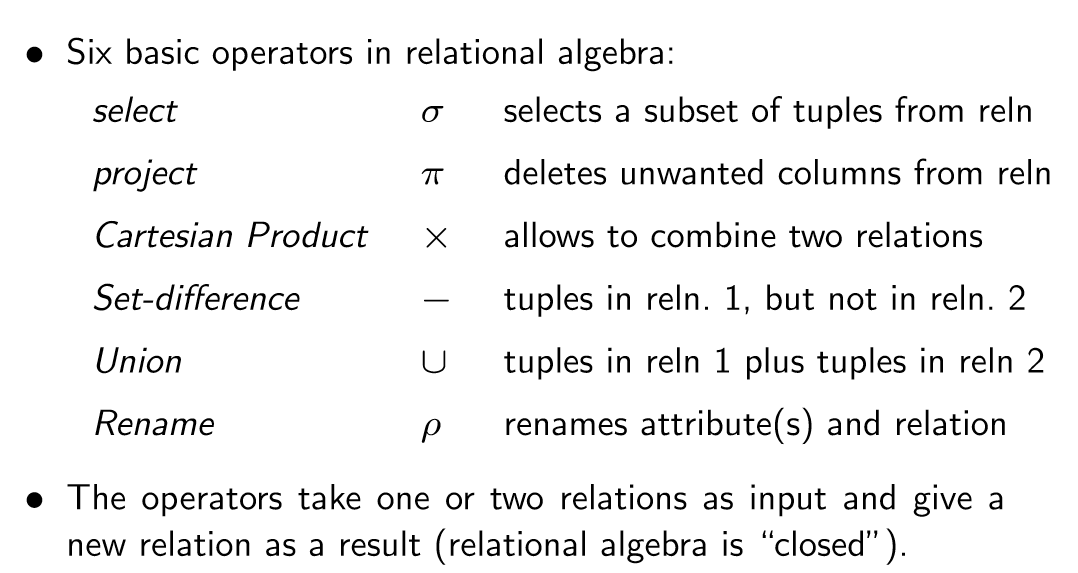
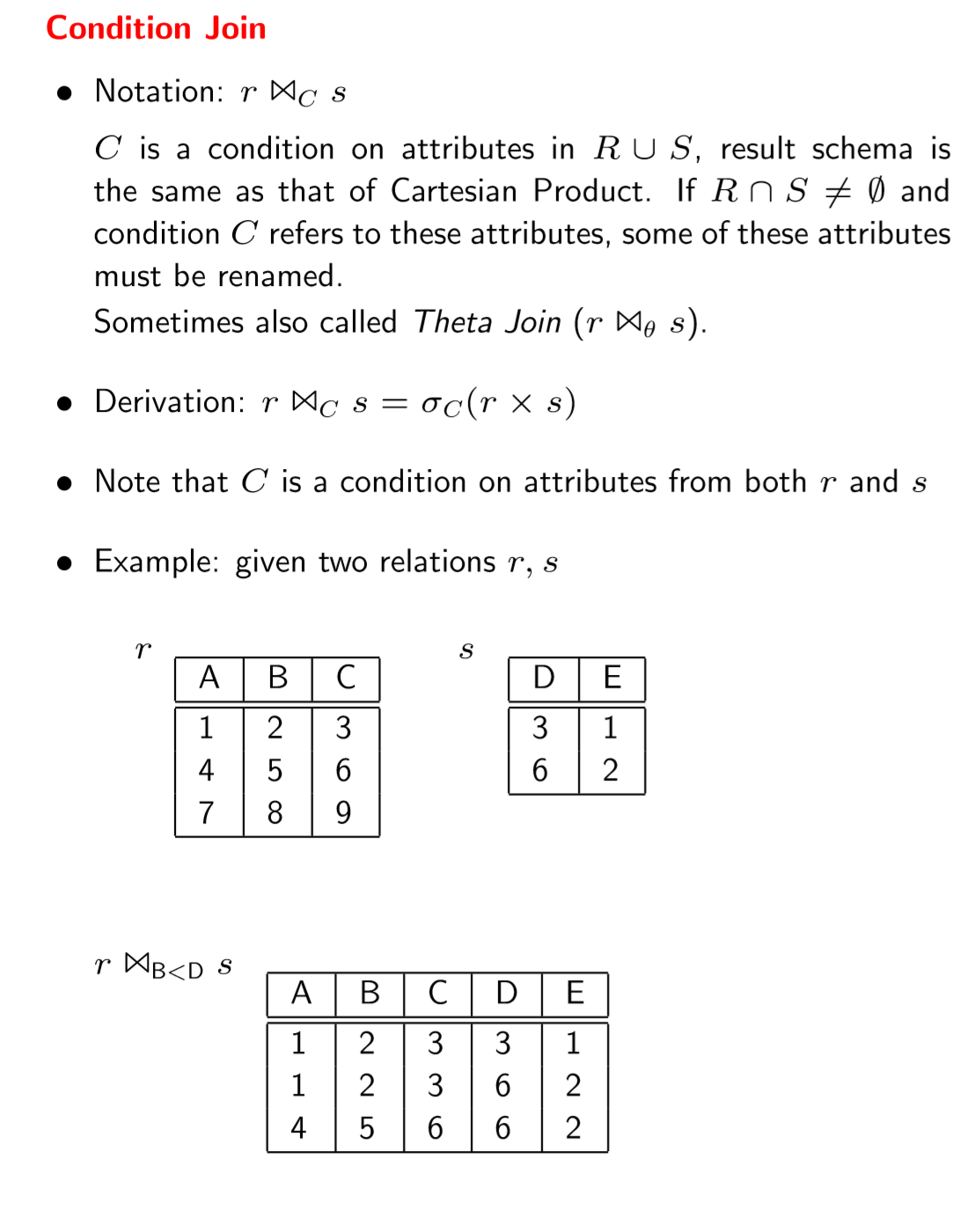
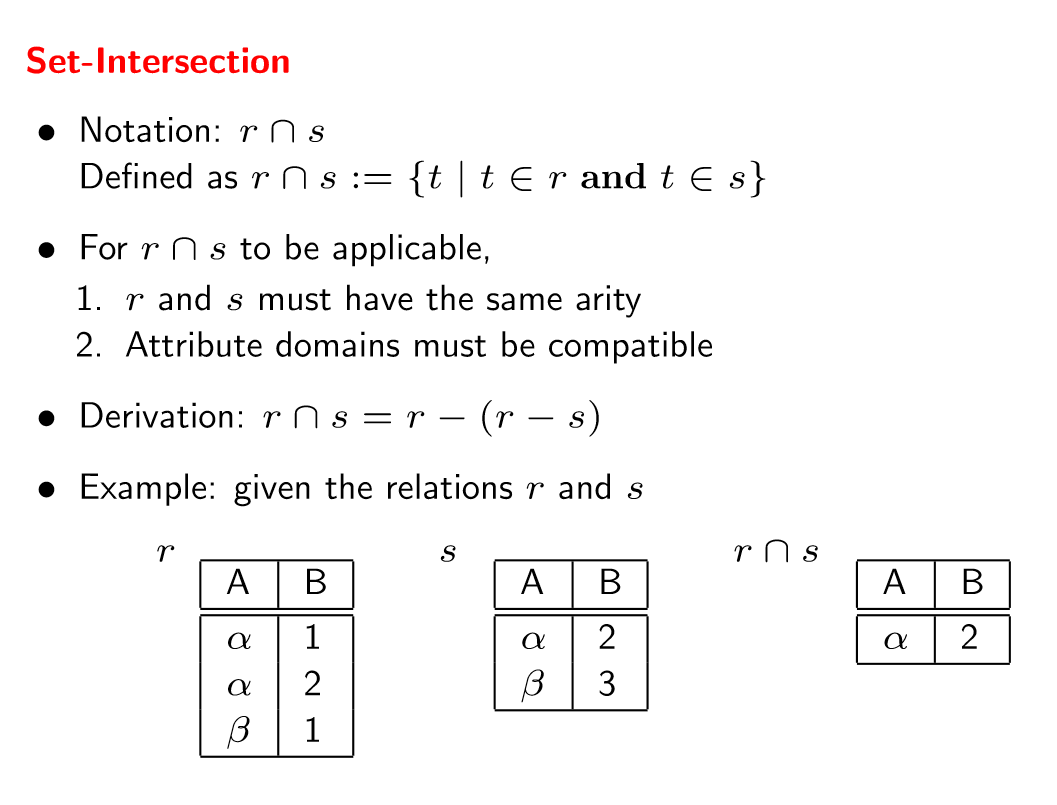
|  |  |  |
| --- | --- | --- |
| **SubQueries HWK**  **Using a single SQL statement display fname,lname of all the students who are taking Database Programming regardless of case.**  SELECT fname, lname  FROM student  WHERE SSN IN(SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE upper(class\_description) = ('DATABASE PROGRAMMING')));  **Using a single SQL statement display all the rows from the student\_class table where class description is not null**  SELECT \* FROM STUDENT\_CLASS  WHERE class\_code IN  (SELECT class\_code FROM class WHERE class\_description IS NOT NULL);  **Using a single SQL statement display fname, lname of all the students whose first name is anything except John, Jack or Bob. and are taking the operating systems class and their phone number is null**  SELECT fname, lname  FROM student  WHERE UPPER(fname) NOT IN('JOHN','JACK','BOB') AND phone IS NOT NULL  AND SSN IN(SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE upper(class\_description) = ('OPERATING SYSTEMS')));  **Using a single SQL statement display ssn, fname, lname, age/2 of all the students whose first name begins with the letter J and age is greater than 25 and are taking any class that contains ‘Intro’ in its description (Have to convert the dob into a number). Order the results by age/2 in descending order. Use an alias for the order by clause**  SELECT fname, lname, ssn, TRUNC(MONTHS\_BETWEEN(SYSDATE,DOB)/12)/2 age  FROM student  WHERE UPPER(fname) LIKE 'J%' AND ROUND(MONTHS\_BETWEEN(SYSDATE,DOB)/12)>25  AND SSN IN(SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE upper(class\_description) LIKE '%INTRO%'))  ORDER BY ROUND(MONTHS\_BETWEEN(SYSDATE,DOB)/12)/2 DESC;  **Using a single SQL statement display fname, lname from the student table where last name contains the letters ‘nn’ (e.g. Benny, Bonny, Sonny) and is enrolled in any class that contains the letter ‘h’ in its description regardless of case. Order the results by lname. When using order by use the position and not the name of the column**  SELECT fname, lname  FROM student  WHERE UPPER(lname) LIKE '%NN%'  AND SSN IN(SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE upper(class\_description) LIKE '%H%'))  ORDER BY 2;  **Using a single SQL statement, delete all the rows from the class table for all classes that are associated with students who live in Sacramento and earn less than 15000 (NOTE: you are deleting from the class table)**  CREATE TABLE class2 AS SELECT class\_code, class\_description FROM class WHERE class\_code IN  (SELECT class\_code FROM student\_class WHERE ssn IN  (SELECT ssn FROM student WHERE TRUNC(MONTHS\_BETWEEN(SYSDATE, dob)/12)>30));  **Update the salary to 75000 for all students who are enrolled in ‘Database programming’ regardless of case and live in CA**  UPDATE student SET salary=75000  WHERE ssn IN  (SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE UPPER(class\_description)='DATABASE PROGRAMMING'  AND UPPER(state) = 'CA')); | **Group BY HWK**  **Write a single SQL statement that displays the number of of people with the same lastname. The results should contain the lastname and the count for each lastname. Exclude from the list all those who live in CA**  SELECT lname AS "Last Name", COUNT(\*) AS "Count"  FROM student  WHERE UPPER(state) != 'CA' GROUP BY lname ORDER BY lname;  **Write a single SQL statement that displays the number of people living in each of the states. The results should display the state and the number of people living in each state. Exclude from the list all those who are living in cities that contains the letter ‘h’**  SELECT state AS "State", COUNT(\*) AS "Count"  FROM student  WHERE LOWER(city) NOT LIKE '%h%'  GROUP BY state  ORDER BY state;  **Use a single SQL statement that displays the ssn and the number of classes a student is taking with the column heading “number of classes” where the number of classes is less than 2 , order by ssn descending.**  SELECT ssn, COUNT(class\_code)  FROM student\_class  GROUP BY ssn HAVING COUNT(class\_code)<2  ORDER BY ssn DESC;  **Write a single SQL statement that displays the average age for each city, state combination for all students whose salary is greater than the average salary and are taking some kind of ‘Intro’ class. Also exclude the city ‘Berkeley’ from the list regardless of case. Sort by city in ascending order and state in descending order**  SELECT city, state, AVG(TRUNC(MONTHS\_BETWEEN(SYSDATE, dob)/12)) AS "Age"  FROM student WHERE salary > (  SELECT AVG(salary) FROM student) AND ssn IN  (SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE class\_description LIKE 'Intro%'))  AND LOWER(city) != 'berkeley' GROUP BY city, state ORDER BY city ASC, state DESC;  **Write a single SQL statement that displays the States in lower case along with the rounded average age for the different states with the alias name “average of ages” for all the students who are taking a class that contains ‘principles’ in its description regardless of case.**  SELECT LOWER(state) AS "states", ROUND(AVG(TRUNC(MONTHS\_BETWEEN(SYSDATE, dob)/12))) AS "Average of Ages"  FROM student  WHERE ssn IN  (SELECT ssn FROM student\_class WHERE class\_code IN  (SELECT class\_code FROM class WHERE LOWER(class\_description) LIKE '%principles'))  GROUP BY state ORDER BY state; | **Group Union HW**  **1)Display the name and the corresponding description based on the candidates salary (Use union and be cautious of the types) Order by name**  **< 50000 Honest**  **>50000 and < 100000 Not so honest**  **>100000 80000**  SELECT fname 11"i! lname AS name, NVL(salary,0) AS salary, 'Honest' FROM candidate WHERE NVL(salary,0) < 50000  UNION  SELECT fname 11"i! lname AS name, NVL(salary,0) AS salary, 'Not so honest' FROM candidate WHERE NVL(salary,0) > 50000 AND NVL(salary,0) < 100000  UNION  SELECT fname 11"i! lname AS name, NVL(salary,0) AS salary, '80000' FROM candidate WHERE NVL(salary,0) > 100000;  **Display the name and the corresponding description based on the candidates salary (Use a plain case statement in chapter 5)**  **< 50000 Honest**  **>50000 and < 100000 Not so honest**  **>100000 80000**  SELECT fname 11"i! lname AS name, NVL(salary,0) AS salary,  CASE WHEN NVL(salary,0) < 50000 THEN 'Honest'  WHEN NVL(salary,0) > 50000 AND NVL(salary,0) < 100000 THEN 'Not so honest'  WHEN NVL(salary,0) > 100000 THEN '80000' END FROM candidate;  **Display the name of all the people who are not associated with a party (use not in). This is a bit tricky because people who don’t have a party\_id will have a null and you cannot compare a null using not in because it doesn’t know how to deal with non-data. You can put the party\_id in an NVL function in the where clause to resolve this issue**  SELECT fname ||’ ‘|| lname AS name FROM candidate  WHERE NVL(partyid,0( NOT IN (SELECT partyid FROM party);  **Repeat question 3 using (not exists)**  SELECT fname ||’ ‘|| lname AS name FROM candidate  WHERE NOT EXISTS( SELECT \* FROM party WHERE candidate.partyid=party.partyid);  **Repeat question 3 using (minus)**  SELECT fname ||’ ‘|| lname AS name, NVL(partyid,0) AS partyid FROM candidate WHERE NVL  (partyid, 0) = 0 MINUS SELECT partydesc, partyid FROM party; |





Relational algebra: answer is (2,3,3,5) for 3 of them and the answer for the other is (4,3)

Code: select lname, nvl(partydesc, ‘no description’) partydesc from candidate c, party p where c.partyid = p.partyid(+);

Select count(\*) count, partydesc from party natural join candidate group by partydesc;

|  |  |  |  |
| --- | --- | --- | --- |
| **Joins HWK**  **Give a listing of all the ssns,first names and the class descriptions of**  **all the classes the students are taking. If there are no class \_descriptions**  **display 'No description is available yet'. (USE NVL)**  SELECT student.ssn, student,fname, NVL(class.class\_description, ‘No Description is avialable yet’) AS ‘’Class description” FROM class, student, student\_class WHERE class.class\_code=student\_class.class\_code AND student\_class.ssn=student.ssn;  Give a listing of only the lname and the class\_code for  students who are taking 'Introduction to C programming'. (Inner join)  SELECT lname, class.class\_code  FROM class, student, student\_class  WHERE class.class\_code=student\_class.CLASS\_CODE AND student\_class.SSN=student.SSN AND class\_description=’Intro to C programming’:  **Give a listing of only the lname and the class\_code for students who are taking ‘Intro to C programming’. (Inner join)**  SELECT lname, class.class\_code  FROM class, student, student\_class  WHERE class.class\_code = student\_class.CLASS\_CODE AND student\_class.SSN = student.SSN AND class\_description=’Intro to C programming’;  **Give a lising of all the class\_descriptions and the number of**  **students enrolled in each class for all students who are older**  **than the average age where the total number of students for the class is more than 1 student. Order by the number of students. If thereis no class description replace it with 'Other Classes'(Note: Take it in steps. First do all those who are older than the average age, then do the group by, then add**  **the having clause and then the order and then combine everything together**  SELECT NVL(class.class\_description, ‘other classes’) as “class description”, COUNT(\*) AS “Number of Students”  FROM class, student, Student\_class  WHER student.ssn=student\_class.ssn AND class.class\_code=student\_class.CLASS\_CODE AND TUNC(minutes\_between(sysdate, dob)/12)> (Select AVG(trunc(months\_between(sysdate,dob)/12) FROM student) GROUP by class.class\_description HAVING COUNT(\*)>1 ORDER BY COUnT(\*);  **Give a listing of all the classes for which no students are**  **enrolled in (use in or not in clause) (subquery)**  SELECT class\_code FROM class WHERE class\_code NOT IN (SELECT DISTINCT(class\_code) FROM student\_class);  **Give a listing of all the students who are not enrolled in any**  **classes (Note: Use Exists or not Exists)**  (continued on next column) 🡪 | **Joins HWK continued**  SELECT fname ||’ ‘|| lname AS”Name” FROM sturdent WHERE  NOT EXISTS (SELECT \* FROM student\_class WHERE student.ssn=student\_class.ssn);  **create a new table that contains the list of all the studentsand class\_descriptions. Include In this table the list of allstudents who are not enrolled in any classes (display no classes). If there are no class descriptions then display ‘no description’(Use combination of inner join, union and minus)(Note: minus will deal with the students who are not enrolled in any classes)**  CREATE TABLE new\_table AS(SELECT fname||’ ‘||lname AS”name”, NVL(class\_description, ‘No dscription AS”Class description” FROM student, class, student\_class WHERE student.ssn=student\_class.ssn AND student\_class.class\_code=class.class\_code UNION(SELECT fname||’ ‘||lname AS “name”,’No class’ FROM(SELECT ssn FROM student MINUS SELECT ssn FROM student\_class) NATURAL JOIN student));  **repeat question 6 using a combination of inner join, union and not exists (Note: Not exists will deal with the students who are not enrolled in any classes)**  CREATE TABLE new\_table AS(SELECT fname || ‘ ‘|| lname AS “Name”, NVL(class\_description, ‘No Description’) AS “Class Description” FROM student, class, student\_class WHERE student.ssn=student\_class.ssn AND student\_class.class\_code=class.class\_code UNION (SELECT fname|| ‘ ‘|| lname AS “Name”, ‘No Classes’ FROM (SELECT ssn FROM student WHERE NOT EXISTS (SELECT ssn FROM student\_class WHERE student.ssn=student\_class.ssn)) NATURAL JOIN student));  **We want to find out which courses are being taken by the different students for all those whose age is greater than the average age. Give a listing of the course descriptions and student names (Inner join)**  SELECT fname||’ ‘||lname AS “name”, NVL(class\_desccription, ‘other classes’) AS “class description” FROM class, student, student\_class WHERE student.ssn=student\_class.ssn AND class.class\_code=student\_class.class\_code AND TRUNC(MONTHS\_between(sysdate, dob)/12)>(SELECT AVG(trunc(months\_between(sysdate,dob)/12)) from student);  **We want to find out the courses that each student is not enrolled in.**  **Give a listing of the course descriptions, and the students (lname) who are not taking that specific course**  **(Use a cartesian product and inner join it with a minus)**  SELECT classs\_description, lname FROM class, student MINUS SELECT lname, class\_description FROM student, class, student\_class WHERE class.class\_code = student\_class.class\_code AND student\_class.ssn=student.ssn | | **Example 9b (using separate queries)**  --In this example, we want to know the names of all the people who are infected with Malaria.  --Using separate queries, we first have to find out what the disease\_id is for malaria which is in the disease table. We then use the disease\_id to find out which patient\_id(s) have malaria.  --For this we need to go to the patient\_disease table. Finally, we take the list of patient\_ids and feed it into the patient table to get their names. Based on what we know so far, we would write three separate queries but in the next example, we will connect them into a single query. This would be done through the subquery mechanism.  SELECT disease\_id FROM disease WHERE disease\_desc='Malaria';  SELECT patient\_id FROM patient\_disease WHERE disease\_id=22;  SELECT fname, lname FROM patient WHERE patient\_id=111;  **Example 9c (Using subqueries-one single row)**  A single- row subquery can return only one row of results consisting of only one column to the outer query. A single- row subquery can also be nested in the outer query's SELECT clause.  —Display all the patients who have malaria. Returns a single row.  —Start with inner most query and work your way to the outer query. Notice the number of —parantheses. The indentation is used for readability.  SELECT fname, 1 name FROM patient WHERE (  SELECT patient id FROM patient\_disease WHERE disease SELECT¯disease id FROM disease WHERE disease desc='Ma1aria') ) ;  --lnvalid. Notice the asterisk does not match up with disease\_id.  —The disease id must match with disease\_id and not the asterisk.  SELECT fname, Iname FROM patient WHERE patient (  SELECT patient id FROM patient disease WHERE disease (  SELECT FROM disease¯WHERE disease desc='Ma1aria') ) ;  --lnvalid. Notice the asterisk does not match up with patient\_id.  SELECT fname, Inarne FROM patient WHERE patient  SELECT FROM patient disease WHERE  SELECT disease\_id FROM disease WHERE disease desc—'Ma1aria' )) ; |
| **Example 9d (Multiple rows)**  Multiple- row subqueries are nested queries that can return more than one row of results to the parent query. The main rule to keep in mind when working with multiple- row subqueries is that you must use multiple- row operators. If a single- row operator is used with a subquery that returns more than one row of results, Oracle returns an error message. Valid multiple- row operators include IN, ALL, and ANY must be used. Of the three, the IN Operator is used most often.  --lnvalid. Display all the patients who have Cancer. Returns multiple rows.  --Start from the inner-most query. The result would then bubble up to the outer queries. The problem with this query is that there are multiple patients who suffer from cancer. This would mean that the second subquery would return multiple rows; however, the from the outer —most query can only handle one single piece of information.  SELECT fname, Iname FROM patient WHERE (  SELECT patient\_id FROM patient disease WHERE disease  SELECT disease\_id PROü disease WIIERE 'Cancer' ) ) ;  —To correct the above problem, we change from (z) to (in). The in operator can handle multiple values.  SELECT fname, Iname FPC)M pat j ent WHERE patient id  SELECT patient id FP.OM patient \_disease WHERE disease ide (  SELECT disease id FROM disease WHERE disease desc—'Cancer' ) ) ; | **Example 9e (Single and multiple rows)**  —Display all the diseases that "jill crane" has.  SELECT disease desc FROM disease WHERE disease  SELECT disease\_id FROM patient disease WHERE (  SELECT patient id FROM patient WHERE fname—' j ill' and Iname=' Crane' ) ) ;  --lnvalid. Display all the diseases that "John Doe" has. Must use in clause.  SELECT disease desc FROM disease WHERE disease id —  SELECT disease id FROM patient disease WHERE patient (  SELECT patient id FROM patient WHERE fname=' john' and Iname=' Doe' ) ) ;  SELECT disease desc FROM disease WHERE disease id IN (  SELECT disease id FROM patient disease WHERE patient id= (  SELECT patient id FROM patient WHERE fname—' john' and Inane—' Doe' ) ) ;    --Notice the concatenation operator in the subquery.  --The concatenation operator takes the two pieces of data and connect them together --to make it appear as one single piece.  SELECT disease desc FROM disease WHERE disease id IN (  SELECT disease id FROM patient disease WHERE, patient id= (  SELECT patient\_id FROM patient WHERE (fname I I 'johnDoe') ) ) ; | | **Example 9f (Multiple column subquery)**  Multiple- column subquery returns more than one column to the outer query. The syntax of the outer WHERE clause is WHERE ( columnname, columnname, ...) IN subquery.  Keep these rules in mind: • Because the WHERE clause contains more than one column name, the column list must be enclosed in parentheses. Column names listed in the WHERE clause must be in the same order as they’re listed in the subquery’s SELECT clause.  --Notice that fname & lname are enclosed in parantheses in the outer query. It works like the concatenation operator in that fname and lname become a single piece of data which are compared against fname & lname in the inner query. The (IN) operator is used because it is possible that the inner query may yield multiple rows.  SELECT patient\_id FROM patient WHERE (fname,lname) IN (  SELECT fname,lname FROM special\_names); |
| **Example 9g (Group functions and subqueries)**  —That single number will be fed to the outer query which can be used to compare every row in the patient table.  SELECT fname, Iname, salary FROM patient WHERE salary > (SELECT AVG (salary) FROM patient) ;  --To make AVG work, dates have to be converted to numbers which can be done by using MONTHS BETWEEN. Notice that a subquery has to be used to deal with the AVG first.  SELECT fname, Iname, DOB FROM patient where MONTHS BETWEEN (sysdate, DOB) > (SELECT AVG (MONTHS BETWEEN(s sdate, DOB) ) FROM patient) ; | **Example 9h (Create table and subqueries)**  You can also perform CREATE TABLE AS by using subqueries.  --lnstead of displaying the information on to the screen, it can be fed into a brand new table.  CREATE TABLE NEW TABLEI ms SELECT patient id FROM patient WHERE (fname, 1 name) IN (SELECT fname, 1 name FROM special \_names) ;  SELECT \* FPOM NEW TABLEI;  --lnvalid. Mur.t use an alias because the new table will be using the information between --the select and from to come up with the column names for the new tables. Since —salary' 2 is not a valid column name, an alias has to be used.  CREATE TABLE NEW TABLE2 AS SELECT patient id, salary \* 2 FROM patient WHERE (fname, Iname) IN (SELECT fname, I name FROM special \_names) ;  --This query corrects the problem from the previous example.  CREATE TABLE NEW TABLE2 AS SELECT patient id, salary  2 Increase FROM patient WHERE (fname, I name) IN ( SELECT fname, Inarne FROM special names) ;  SELECT \* FROM NEW TABLE2; | | **Exmnple 9i (Update and delete using subqueries)**  You can also perform UPDATE and DELETE statements  SELECT pat-ienL\_id, salary FROM patient;  --This example updates the salary for all those patients who have cancer.  UPDATE patient SET salary—salary "2 WIIERE patient id IN (  SELECT patient id FROM patient \_disease WHERE disease id— (  SELECT disease id FROM disease WHERE )  SELECT patient id, salary FROM patient;  —This example deletes all the records from the patient\_disease table for all those who have cancer.  DELETE FROM patient disease WHERE disease\_id IN (  \_  SELECT disease id FROM disease WHERE disease desc='Cancer') ;  SELECT \* FROM patient disease; |
| **Summary Examples Chapter 9**  --This example updates the salary for all those patients who have cancer.  --Notice the IN clause will have to be used to deal with multiple rows.  --Also, note that the where condition for the outer query filters on both the patient\_id and the salary.  SELECTpatient\_id, salary FROM patient WHERE patient\_id IN (  SELECT patient\_id FROM patient\_disease WHERE disease\_id=(  SELECT disease\_id FROM disease WHERE to\_lower(disease\_desc)='cancer')) AND salary IS NOT NULL ;  --Notice that the AVG function is enclosed in its own query. Salary>AVG(salary) is wrong.  SELECT patient\_id, salary FROM patient WHERE salary >  (SELECT AVG(salary) FROM Patient) | **10.1 Cartesian/Cross JOIN**  In a Cartesian join, also called a Cartesian product or cross join, each record in the first table is matched with each record in the second table. This type of join is useful when you're performing certain statistical procedures for data analysis. Therefore, if you have three records in the first table and four in the second table, the first record from the first table is matched with each of the four records in the second table. Then the second record of the first table is matched with each of the four records from the second table, and so on. The CROSS keyword, combined with the JOIN keyword, can be used in the FROM clause to explicitly instruct Oracle to create a Cartesian (cross) join. The CROSS JOIN keywords instruct the database system to create cross- products, using all records of the tables listed in the query. | | **Example 10.1 a (Cartesian product, cross join)**  SELECT \* FROM skills;  SELECT FROM heroes;  --Cartesian product gives every combination.  SELECT FROM hero, skills; OR SELECT FROM hero CROSS JOIN skills;  **Exantple 10.1 b (Patient example)**  patient d, fname, name,  SELECT FROM disease; salary FROM patient;  SELECT FROM patient disease;  --Cartesian product  SELECT fname, Iname, disease desc FROM patient, disease, patient disease;  --Another way of doing a cartesian product.  SELECT fname, Iname, disease desc FROM patient CROSS JOIN disease CROSS JOIN patient disease; |
| **10.2 Inner Join**  The most common type of join is based on two ( or more) tables having equivalent data stored in a common column. These joins are called equality joins but are also referred to as equijoins, inner joins, or simple joins.The traditional way to include join conditions and avoid an unintended Cartesian result is to use the WHERE clause. The WHERE clause can perform two different activities: joining tables and providing conditions to limit or filter the rows that are affected.A column qualifier indicates the table containing the column being referenced.With the equality, non- equality, and self- joins, a row is returned only if a corresponding record in each table is queried. These types of joins can be categorized as inner joins because records are listed in the results only if a match is found in each table. | **Example 10.2a (Simple join)**  —We want to retrieve all the heroes and their corresponding skills. We have to connect the common --columns. For this we need an inner join (Also referred to as equi-join).  --Since skill code appears in both tables, we have to prefix the columns with the table name --to avoid am Igulty.  SELECT \* FROM heroes, skills WHERE heroes. skill code=skills.skill code;  --Since the name of the tables can be long, we can use aliases to refer to the tables. Once —aliases are assigned, we cannot use the table names.  SELECT \* FROM heroes h, skills s WHERE h. skill code—s. skill code;  --lnvalid: cannot use table names once aliases have been assigned.  SELECT \* FROM heroes h, skills s WHERE heroes. skill code—skills. skill code;  --h.\* referes to all the columns in the heroes table. S. \* refers to all the columns in the skills table.  SELECT h. \* , s.\* FROM heroes h, skills s WHERE h. skill code—s.skill code;  --Display only the needed columns.  SELECT name, skill name FROM heroes h, skills s WHERE  h. skill code=s. skill code; | | **Exantple 10.2b (Patient exantple)**  --We want to display the natues and the diseases of the different people. Since names appear  -in one table and descriptions in another, we have to do a join. The join is done by connecting  -the common columns. All three tables have to be connected. If a table is included in the FROM --clause but is not connected in the WHERE clause, then the results will look like a Cartesian --product, which is more than likely not what we want.  --The columns can be connected in any order as long as all three tables have the connection.  SELECT fname, 1 name, disease desc FROM patient p, disease d, patient disease pd WHERE p. patient: id—pd. patient id AND pd. disease id—d. disease id;  --This is not what we want. Notice the disease table is not joined. The result is that patient and --patient\_disease will be inner joined. Their result will then be cross joined with --the disease table, which would logically be erroneous.  SELECT fname, Iname, disease desc FROM patient p, disease d, patient disease pd WHERE at lent id— d. at fent id; |
| You can use three approaches to create an equality join that uses the JOIN keyword: NATURAL JOIN, JOIN . . . USING, and JOIN . . . ON:   * The NATURAL JOIN keywords create a join automatically between two tables, based on columns with matching names. * The USING clause allows you to create joins based on a column that has the same name and definition in both tables. * When the tables to be joined in a USING clause don't have a commonly named and defined field, you must add the ON clause to the JOIN keyword to specify how the tables are related. * There are two main differences between using the USING and ON clauses with the JOIN keyword: * The USING clause can be used only if the tables being joined have a common column with the same name. This rule isn't a requirement for the ON clause. * A condition is specified in the ON clause; this isn't allowed in the USING clause. The USING clause can contain only the name of the common column. | **Example 10.2c (Natural join)**  --lnner joins can be done using a variety of syntax. Instead of using the WHERE clause as in --last example, the key words NATURAL JOIN can be used. It will automatically find the commonly --named columns and connect them together.  --Also, table aliases are not allowed. Notice the skill\_code appears in both tables but with this --new syntax, we don't need to prefix the column with the table name. Natural join can figure --things out by itself.  SELECT name, skill code FROM heroes NATURAL JOIN skills;  --The order of NATURAL JOIN does not matter.  SELECT FROM skills NATURAL JOIN heroes;  --Can also use the plain JOIN syntax and the USING clause to identify the common column.  SELECT name, skill \_name FROM heroes JOIN skills USING (skill code) ;  --Can use the JOIN clause and the ON keyword. This syntax begins to look like the first join that --we did using a WHERE clause.  SELECT name, skill nameFROM heroesh JOIN skills s ON h.skill code=s.skill code; | | **Example 10.2d (Patient Example)**  --Can do a natural join against multiple tables.  --0rder does not matter when using natural join.  SELECT fname, 1 name, disease descFROM patient NATURAL JOIN disease NATURAL JOIN patient disease;  ORDER DOES NOT MATTER (NATURAL JOIN)  SELECT fname, 1 name, disease desc FROM patient NATURAL JOIN patient disease NATURAL JOIN disease |
| **Example 10.2e (Natural join with multiple columns)**  Drop TABLE a;  DROP TABLE b;  CREATE taBLE A  COLA NUMBER,  COLB NUMBER, COLC NUMBER  INSERT INTO a VALUES  INSERT INTO a VALUES ;  INSERT INTO a VALUES  INSERT INTO a VALUES  CREATE TABLE B (COLd NUMBER, COLB NUMBER, COLA NUMBER) ;  INSERT INTO b VALUES  INSERT INTO b VALUES  INSERT INTO b VALUES  INSERT INTO b VALUES  —Matches on all the columns that have the same name.  --lf it doesn't find any matches between the column names, then it works like a cross join.  SELECT \* FROM a NATURAL JOIN b; | **Example 10.2F (Patient Example with GROUP B Y)**  --ln this example, we are tryjng to find the number of diseases each person has as long as they  •-they have more than one disease. Since the name cornes from the patient table but the actual disease --association comes from patient\_disease, we have to do an inner Join.  --ERROR: Alias names cannot be used in the GROUP BY or the HAVING clause.  SELECT fname first: name, count ( w ) Di3easeCount  FROM patient p, patient digeage pd  WHERE p. patient id—pd. patient åd  GROUP BY first-Name HAVING DiseaseCount >1;  --This is a correction to the above statement.  SELECT fname, count ( E ) DiseaseCount FROM patient p, disease d, patient\_disease pd WHERE p. patient id and  d. disease id—d. disease idGROtJP BY fname HAVING count ( w ) > 1; | **Example 10.2g (Non-equijoin)**  SELECT FROM students;  SELECT FROM grade range;  —A non-equi join is like a inner join in that it joins records from multiple tables but the columns —may not have the same name. In other words, there may not be a foreign key relationship.  SELECT name, score, grade FROM students, grade range WHERE  \_ score BETWEEN be score AND end score; | |

|  |  |  |
| --- | --- | --- |
| **Example 10.2h (Numbering each line)**  CREATE TABLE student  Name VARCHAR2 (10) ,  Class VARCHAR2 (10)  INSERT INTO student VALUES ( abdul , 'philosophy') ;  INSERT INTO student VALUES ( 'bob' , 'philosophy') ;  /\*Below is the desired result set that we are looking for. We want to get a count sequence for each of the different categories. Notice this is not like a group by in that we are not trying to suppress any information.  We want to number our records. \*/  /• Rownum is a pseudo-column that is available to us. It is a number that Oracle assigns to each record in the order in which the records were either physically or virtually inserted into the table. Create a table (templ) with new rownumber. \*/  CREATE TABLE templ AS  SELECT rownum AS line number, name, class PROM student;  —Create a table (temp2) that identifies the beginning point.  CREATE TABLE temp2 AS  SELECT min (rownum) AS beg line number, class  FROM student GROUP BY class | **10.3 Self Join**  Sometimes data in one column of a table has a relationship with another column in the same table.This type of join is known as a self- join.  DROP TABLE employee;  CREATE TABLE employee  s sn VARCHAR2 (11) , name VARCHAR2 (11) , manager VARCHAR2 (11) , salary NUMBER  INSERT INTO employee VALUES ('111', 'jack', '222' , 10000) ; INSERT INTO employee VALUES ( ' 333 ' 'john', '222', 20000) ; INSERT INTO employee VALUES ( ' 444 ' 'jill', '111',10000) ;  INSERT INTO em 10 ee VALUES ( '222' , ' oe' , ' 999 ' , 10000) ;  SELECT \* FROM employee;  /\*In this example, we want to find the names of all the employees and their managers. Notice that both names reside in the same table. We can do a join against the same table and assign a different alias to each table making it appear as if we have two separate tables. We can then connect the foreign key (manager) to the primary key (ssn). Notice the use of the alias before each of the columns because they appear in both tables. Without the alias we would get an ambiguously defined column error. Also Joe is not included in the result because Manager (999) does not match with any ssns. \*/  (Continued on the right column) | SELECT e . name EMPLOYEE, m. name Manager FROM employee e, employee m  WHERE e. manager=m. ssn;  --We want to find all the people who are making the same salary.  Problem: This is not what we want because it duplicates the entries.  SELECT el . name, el . salary FROM employee el, employee e2 WHERE el. salary=e2.sa1ary AND el.ssn!=e2.ssn ;  "This resolves the duplicate issue.  SELECT DISTINCT el . name, el. salary FROM employee el, employee e2  WHERE el. salar =e2.sa1ar AND el.ssn!=e2.ssn ;  **10.4 Outer Join**  To tell Oracle to create NULL rows for records that don't have a matching row, use an outer join operator, which looks like this: (+). It's placed in the WHERE clause immediately after the column name from the table that's missing the matching row. It tells Oracle to create a NULL row in that table to join with the row in the other table.You need to be aware of some limitations when using the traditional approach to outer joins: The outer join operator can be used for only one table in the joining condition. In other words, you can't create NULL rows in both tables at the same time. A condition that includes the outer join operator can't use the IN or OR operator |
| **Example 10.4a (Inner join)**  Inner join: Connect all three tables otherwise we will get a Cartesian product. select \* from a, b, c where coll=c012 and c012=c013;  **Example 10.4d (One outer join condition)**  —Common to all and also to (a) and (b).  —It looks for (ab) in c013 but doesn't find a match. So it sets it to NULL. It checks (ab) to c012 and  —does find a match.  select \* from a, b, c where coll=c013 (+) and coll=c012; | **Example 10.4b (One outer join condition)**  --The (+) is used for outer join which means (You don't really care). When you don't care, it  --creates a virtual NULL record behind the scenes. An outer join is first and foremost an inner join. --Then if there is a record for which there is not a match, the (+) says, it is okay and will allow it to --go through.  --ln this case, if there is something in coll for which there is no match in c012, it will automatically --create a NULL record in c012. The problem is that then the NULL record will be compared to the data  --in c013. NULLs cannot be checked in this way and so in this case the (+) means nothing. select \* from a, b, c where coll=c012 (+) and c012=c013; | **Example 10.4c (One outer join condition)**  --ln this example, we find all the common records to a, b and c. In addition, we find all the --records that are common to (a) and (c).  --lt checks (ac) against matches in c012 but it cannot find any. So it sets it to NULL because of (+). It then --checks for (ac) in c013 and it finds a match.  select \* from a, b, c where coll=c012 (+) and coll=c013; |
| **Example 10.4e (One outer join condition)**  --Common to all and also to b and c.  --Notice that coll is not included, which means that we will have a Cartesian product. (abc) in c012 will be found in c013 and also (bc) will be found in c013. The other records that are --in c012, which are not in c013, will go through because of the (+). However, they would be ignored because it is looking to match that data with COB again. The results (ab, abc) will be cross joined -- with every record in table (a).  select \* from a, b, c where c012=c013 (+) and c012=c013; | **Example 10.4f (Two outer join conditions)**  —All the intersecting points are displayed.  —If there is something that is coll but not in c012, the (+) says it is okay. It then creates a NULL record. —If there is something in coll but not in c013. the (+) says it is okay. It also creates a NULL record. select from a, b, c where coll=c012 (+) and coll=c013 (+) ; | **Example 10.4g (Two outer join conditions)**  —Displays all the intersecting points.  —If there is something that is in coll but not in c012, the (+) says it is okay and it creates a NULL record. —If there is something in c012 but not in c013, the (+) says it is okay and also creates a NULL record. select from a, b, c where coll—c012 (+) and c012—c013 (+) ;  **Example 10.4h(Plus sign on only one side)**  — Invalid: Only one + sign can be used.  select \* from a,b where coll(+)=c012(+) ; |
| **Example 10.4i (Inner join)**  SELECT \* FROM patient;  SELECT \* FROM disease;  SELECT \* FROM patient disease;  —This is an inner join, which gives us a listing of all the patient names and their disease descriptions.  SELECT fname, FROM patient p, patient disease pd, disease d  WHERE p. patient id=pd. atient id and pd. disease id=d. disease id; | **Example 10.4j (Including records that don't match with anything else)**  —Select all the people and their disease descriptions, ALO, include in the '.et the  --that are not associated with any individual. In this case (flu) is not associated with anyone,  SELECT fname, Iname, digease desc FPOM patient p, patient digeane pd, digeaze d patient id d. diøeage id (+) -d. digeaze |  |
| **Example 10.4m (Left and right outer join syntax)**  --Notice that in this syntax, patient is on the left of the LEFT OUTER JOIN syntax whereas —patient\_disease is on the right. This would rnean that aside from the inner join. we want to --include records in the patient table that are not the patient\_disease table.  SELECT fname, Inarne,digeage idfrorn patient p LEFT OUTER JOIN patient disease pd ON p. patient id —pd. Patient id;  --Notice that in this syntax, patient is on the right of the RIGHT OUTER JOIN syntax whereas --patient\_disease is on the left. This would mean that aside from the inner join, we want to --include records in the patient table that are not the patient\_disease table.  SELECT fname, patient disease pd RIGHT OUTEP JOIN patient p  ON p. patient i cd—pd. patient id;  •-Here is how we accomplish the same thing using the old (+) syntax.  SELECT fname, Iname, disease idFROM patient digeaoe pd. patient p WHERE pd. atient id at lent id; | **Example 10.4k (All join conditions must be included)**  —This is not what we want. It is missing a join condition. Given the inner join between the patient and —patient disease, along with the individuals who are not associated with any diseases, which is --the outer join, we will Cartesian product the result set with the disease table.  SELECT fname, Inarne,disease d\*SC FROM patient p, patient disease pd, disease d WHERE p. patient id=pd. patient id(+) ;  --This is not what we want, In this case the (+) is associated with the wrong table. In outer joins we want --to include records that are not matched in some other table. Jn this case, all the records in the --patient\_disease are matched up in the patient table. The (+) is extreneous.  SELECT fname, Inarne, disease desc FROM patient p, patient disease pd, disease d  WHERE p. patient id (+) —pd. patient id AND pd. disease id —d. disease id;  When creating a traditional outer j01n with the outer join operator, the join can be applied to only one table— not both. However, with the JOIN keyword, you can specify which table the join should be applied to by using a left, right, or full outer join. Left and right outer joins specify which table the outer join should be applied to, based on the table's location in the join condition. For example, a left outer join instructs Oracle to keep any rows in the table listed on the left side of the join condition, even if no matches are found with the table listed on the right. A full outer join keeps all rows from both tables in the results, no matter which table is deficient when matching rows. ( That is, it performs a combination of left and right outer joins.) | **Example 10.4L (Using join syntax)**  --Can use the alternate syntax of LEFT or RIGHT OUTER JOIN to replace the old (+).  —Notice that in this syntax, disease is on the left of the LEFT OUTER JOIN syntax whereas --patient\_disease is on the right. This would mean that aside from the inner join, we want to --include records in the disease table that are not the patient\_disease table.  SELECT patient id, disease desc from disease d LEFT OUTER JOIN patient disease pd ON d. disease id —pd. disease id;  —Notice that in this syntax, disease is on the right of the RIGHT OUTER JOIN syntax whereas  —patient\_disease is on the left. This would mean that aside from the inner join, we want to  —include records in the disease table that are not in the patient\_disease table,  SELECT patient\_id, disease desc from patient disease pd RIGHT OUTER JOIN disease d ON pd. disease id —d. disease id;  —Here is how we accomplish the same thing using the old (+) syntax.  SELECT patient id, disease\_desc FROM patient\_disease pd, disease d WHERE pd. disease id (+) —d. disease id; |

|  |  |  |
| --- | --- | --- |
| **Example 10.4n(Left and right outer join syntax)**  --First inner join between the three tables. Second we will do the LEFT OUTER JOIN between the disease and patient\_disease which means that we are including the diseases that are in the disease table that are not associated with anyone. "This extra record would now be sitting on the left hand side of the second LEFT OUTER JOIN which —says to include that in the result set as well.  SELECT fname,lname,disease\_desc from disease d LEFT OUTER JOIN patient\_disease pd ON d. disease\_id =pd.disease\_id LEFT OUTER JOIN patient p ON pd.patient\_id=p.patient\_id;  --First inner join. Second we will do the RIGHT OUTER JOIN between the disease and which means that we are including the diseases that are in the disease table that are not associated with anyone.  --This extra record would now be sitting on the left hand side of the LEFT OUTER JOIN which says to include that in the result set as well,  SELECT fname, 1 name, disease desc from patient\_disease pd RIGHT OUTER JOIN disease d ON pd. disease id=d.disease\_id LEFT OUTER JOIN patient p ON pd.patient id;  --The same thing can be done with the old syntax.  SELECT fname, 1 name, disease\_desc FROM patient disease pd, disease d, patient p WHEREpd.disease id (+) —d. disease id AND d. at lent id at lent id (+) ; | **Example 10.4o (Full outer join conditions)**  --Full outer is equivalent to a LEFT and RIGHT OUTER JOIN at the same time. This means that in addition to an inner join, include the stuff on the left hand side for which there is no match on the right hand side. Also include the stuff on the right-hand side for which there is no match on the left hand side.  SELECT pd. disease\_id, d.disease\_desc FROM disease d FULL OUTER JOIN patient\_disease pd ON d. disease\_id=pd. disease\_id;  --This is the same as above because all the records that are in the patient\_disease table are matched against the records in the disease table because it is a bridge table.  --SELECT pd. disease\_id, d.disease\_desc FROM disease d LEFT OUTER JOIN patient\_disease pd ON d. disease\_id =pd. disease\_id;  --Same scenario as above except that it is using the patient table instead of the disease table.  SELECT pd.disease\_id, p.fname FROM patient p FULL OUTER JOIN patient\_disease pd ON p. patient\_id =pd. patient\_id;  --This is the same as above. Don’t need the FULL OUTER JOIN because all the records in the patient\_disease table match up with the records in the patient table because it is a bridge table. (Continued on right column) | 10.4 o (Continued)  SELECT pd.disease\_id, p.fname FROM patient p LEFT OUTER JOIN patient\_disease pd ON p.patient\_id =pd. patient\_id;  --Include all the data from the disease table in the FULL OUTER JOIN. Take that result set which includes the common records and the orphan records in the disease table and include them with  --all the records in the patient table, even the records in the patient table that don’t match with the patient disease table.  SELECT p.fname, d.disease\_desc FROM disease d FULL OUTER JOIN patient\_disease pd ON d. disease\_id =pd. disease\_id FULL OUTER JOIN patient p ON p. patient\_id =pd. patient\_id;  --This is the same as above. Because the patient\_diseaes table is a bridge table, we can do a LEFT OUTER JOIN to include all the stuff from the disease table. Next, we can do a full outer join to include those results along with all the records from the patient table.  SELECT p.fname, d.disease\_desc FROM disease d LEFT OUTER JOIN patient\_disease pd ON d.disease\_id =pd. disease\_id FULL OUTER JOIN patient p ON p. patient\_id =pd. patient\_id; |
| **10.5 Set Operators**  Set operators are used to combine the results of two ( or more) SELECT statements. Valid set operators in Oracle are UNION, UNION ALL, INTERSECT, and MINUS. When used with two SELECT statements, the UNION set operator returns the results of both queries. However, if there are any duplicates, they are removed, and the duplicated record is listed only once. To include duplicates in the results, use the UNION ALL set operator. INTERSECT lists only records that are returned by both queries; the MINUS set operator removes the second query's results from the output if they are also found in the first quen/s results. INTERSECT and MINUS set operations produce unduplicated results  Keep in mind some guidelines for multiple- column set operations: All columns are included to perform the set comparison. Each query must contain the same number of columns, which are compared positionally. Column names can be different in the queries. | **Example 10.5a (UNION)**  --Appends the result from the second query to the first query. Make sure the number of columns in both queries are the same. Also their types must be the same. The column headings come from the first query and ORDER BY clause appears at the end of the last query. --The order by clause refers to columns from the first query  SELECT Iname FROM hourly  UNION  SELECT Iname FROM salaried; | **Example 10.5b (UNION)**  —Instead of doing DECODE or a CASE statement we can just append the results of —one query to another. Notice the number of columns and their types match.  SELECT Iname, hrly\_wage, 'poor’ FROM hourly WHERE hrIy\_wage<=15 UNION  SELECT Iname , hr Iy wage, 'okay' FROM hourly WHERE hrly\_wage>15;  —Notice that the number of columns and data types match using the TO\_CHAR and TO NUMBER functions.  SELECT Iname , hr Iy wage poor FROM hourly WHERE hr Iy wage<=15 UNION  SELECT Iname , TO CHAR (hrly wage) , TO NUMBER(") FROM hourly WHERE hr I wage>15; |
| **Example 10.5c (UNION ALL)**  --Find the Inames that are common between the two tables.  —Duplicates are suppressed. SELECT Iname FROM hourly  INTERSECT  SELECT Iname FROM salaried; | **Example 10.5d (INTERSECT)**  --Find the lnames that are common between the two tables. Duplicates are suppressed.  SELECT lname FROM hourly INTERSECT SELECT lname FROM salaried;  --Notice that the literal text hello appears in both queries, which means that it works just like the above queries.  SELECT lname, 'hello' FROM hourly  INTERSECT  SELECT lname, 'hello' FROM salaried;  --Same as above except the duplicates are not suppressed. Have to use DISTINCT to get the same results.  SELECT lname FROM hourly where lname IN  (SELECT lname FROM Salaried); | **Example 10.5e (MINUS)**  --AII the records that are in salaried that are not in hourly are displayed.  SELECT Iname FROM salaried  MINUS  SELECT Iname FROM hourl ;  --AII the records that are in hourly which are not in salaried are displayed.  --Must have the same number of columns and type.  SELECT Inane FROM hourly  MINUS  SELECT Iname FROM salaried; |
| **Example 10.5F (EXISTS Uncorrelated)**  The EXISTS function searches for the presence of a single row meeting the stated criteria as opposed to the IN statement which looks for all occurrences.  Rule of thumb:  • If the majority of the filtering criteria are in the subquery then the IN variation may be rnore efficient.  • If the majority of the filtering criteria are in the top query then the EXISTS variation may be more efficient.  EXISTS is usually more efficient that IN Because EXISTS use indexes of the table and hence scans the table faster as well as it returns the boolean value (T or F) If T is received for EXISTS clause than the rows will be returned otherwise not. Whereas IN works as simple query where it will scan all possible values in the table and then compares the condition given by you and then the result.  —The following displays all the patients who have diseases. The inner query is done first. The results are then passed on to the outer query. Notice that the patient\_id in the outer query connects to the patient\_id in the inner query. Note that patient\_ids that are NULL in the outer query will not be considered because the IN clause only 100ks at data. NULL is void of data.  SELECT patient\_id, Iname FROM patient WHERE patient\_id IN (SELECT patient\_id FROM patient\_disease);  —All the patients, even the ones that don't have a disease are displayed. For every row that is being processed in the outer query, the inner query is executed. Notice —that unlike the IN clause, there is nothing to connect the outer to the inner query. There is no column in the where clause. If EXISTS (SELECT • FROM patient\_disease) which appears after the WHERE clause comes back with a TRUE result, then the row that is being processed by the outer query is accepted, otherwise it is rejected. When we get rows back from (Continued to next column) | --(SELECT • FROM patient\_disease) then we have a true condition. If we get no rows back then it is false.  SELECT patient\_id, Iname FROM patient WHERE EXISTS (SELECT FROM patient disease);  **Example 10.5g (EXISTS correlated)**  So far you have studied mostly uncorrelated subqueries: The subquery is executed first, its results are passed to the outer query, and then the outer query is executed. In a correlated subquery, Oracle uses a different procedure to execute a query. A correlated subquery references one or more columns in the outer query, and the EXISTS operator is used to test whether the relationship or link is present.  —All the patients who have a disease are displayed.  --Unlike the previous example, this one connects the outer query to the inner query. For every  --row that is processed in the outer query, the inner query also gets processed. For every row in the outer --query, the patient-id is compared against the patient\_id in the inner query.  SELECT patient\_id, Iname FROM patient p WHERE EXISTS (SELECT \* FROM patient disease pd WHERE p. patient id=pd.patient id ) ;  --AII the patients that don't have a disease are displayed.  --The opposite of above is done using the NOT EXISTS clause.  SELECT patient\_id, Iname FROM patient p WHERE NOT EXISTS (SELECT FROM patient disease pd WHERE p. patient id=pd.patient id) ;  **Summary Examples**  --Cartesian product (All combinations)  —No connection is made between the tables.  SELECT FROM patient, patient disease;  SELECT FROM patient CROSS JOIN patient\_disease;  --lnner join (Only commonalities)  --AII tables are connected. (Continued to next column) | **Summary examples continued**  SELECT p. patient id, disease desc FROM patient p, patient disease pd, disease d WHERE p. patient\_id=pd.patient id AND pd. disease id=d.disease id;  --No alias is needed with natural join.  SELECT patient id, disease desc FROM patient NATURAL JOIN patient disease NATURAL JOIN disease;  --Connect on the common column.  SELECT patient id, disease id FROM patient JOIN patient disease USING (patient id)  --Replace the keyword WHERE with ON.  SELECT p. patient id, disease id FROM patient p JOIN patient\_disease pd ON p.patient id=pd.patient id;  --Example: For each name and description category, display the sum for the salary for only those categories whose sum is greater than 10000.  SELECT Iname, description, sum (salary) FROM patient p, patient\_disease pd, disease d WHERE p. patient id=pd.patient id AND pd. disease id=d. disease id GROUP BY 1 name, description HAVING sum (salary) >10000;  --0uter join (Commonalities plus orphan records from one side)  --(+) means that if you cannot find a match then create an implicit NULL row which means we don't care --if we find a match or not. (+) is associated with the table that does not have the matching record.  --(+) should be associated with the table only once and cannot appear on both sides of  SELECT p. patient i d, disease\_desc FROM patient p, patient\_disease pd, disease d WHERE p. patient id=pd.patient id(+) AND pd. disease id=d.disease id (+) ; |
| **Relational Algebra**  Select: σ <selection condition> (R); it is like where clause  Project p <list> (R) selects columns from the table and discards the other columns; The number of tuples in the result of projection p **<list>** (R)is always less or equal to the number of tuples in R.  Union R U S includes all tuples that are either in R or in S or in both R and S; Duplicate tuples are eliminated.  Intersection R ∩ S includes all tuples that are in both R and S; The two operands must be Union compatible  MINUS R – S The two operands must be Union compatible; Result: a relation that includes all tuples that are in R but not in S.  Union and Intersection are *commutative operations* **R U S = S U R, and R** ∩ **S = S** ∩ **R**; Union and intersection are *associative operations* **R U (S U T) = (R U S) U T, and (R** ∩ **S)** ∩ **T = R** ∩ **(S** ∩ **T)**; The minus operation is *not commutative* **R - S ≠ S – R**  Cartesian Product R X S Combine tuples from two relations in a combinatorial fashion  Join R ∞ <join condition> S Equijoin with “=” only; Always have one or more pairs of attributes that have *identical values* in every tuple  Natural Join \* A equijoin without superfluous attributes; Any two join attributes have the same name in both relations; Join attributes; Equating all attributes pairs that have the same name in the two relations; Rename when necessary before applying nature join.  **Example Problems**  List the year and title of each book:  (p **title, year** (book)  List all information about students whose major is CS:  (σ major=’cs’ (student)  List all students and all the possible books they can borrow:  book x students {cartesian product of two tables}  List all books published by Pearson before 1995  (σ year <1995 and publisher = ‘Pearson’ (books))  List the name of authors who live in Sac  p **Sname** (σ Address = ‘Sacramento’ (Authors))  List the name of students who are older than 25 and are not studying CS  p **Sname** (σ Age > 25 & major != ’CS’ (students))  Rename AName in the relation AUTHORS to Name (rename symbol is p [rho])  P name🡨 Aname (authors) | **Example Problems**  List each book with its keywords p **keyword, title** (books ∞ describes)  List each student with the books s/he has borrowed p **sname** (Book ∞ (student ∞ Borrows))  List the title of books written by the author ’Ullman’  p **title** (Books ∞ (σ Aname=’Ulman’ (Has\_written))  List the names of all students who have borrowed a book and who are CS majors p **sname** (Borrows ∞ (σ major=’cs’(student)) )  Rewrite Query 4 using assignments r1 = (σ major=’cs’(student))  r2= r1 ∞ Borrows r3 = (p **sname** (r2)  List the authors of the books the student ’Smith’ has borrowed  (p **Aname** (Has\_written ∞ (Borrows ∞ (σ**sname=’Smith’** (student)))))  Which books have both keywords ’database’ and ‘Programming’  p**title** (Books ∞ (p **isbn** (σ keyword=’database’ (Describes)) ∩ (p **isbn** (σ keyword=’programming’ (Describes))))  List the title of books written by the author ‘Expert’ but not books that have the keyword ’database’.  p**title** (σ Aname = ‘Expert’ and keyword != ‘database’  ((Books ∞ Has\_written) ∞ Describes))  **Relational Algebra Example Assignment**  Employees(eid,name, salary,dID,m\_dID)  Projects(pID, description)  Works\_on(eID, pID, hours)  Departmets(dID, location)  List the name of the project that have employees from the systems department working less than 5 hours. pID is also the name of the project  Πpid(σhours<5(works\_on)⨝σ dID =’Systems(employees)) | **Coding Questions**  Write a single SQL statement that gives a listing of all country names that don’t have any person in (use not exists clause)  SELECT country FROM <table name> a WHERE NOT EXISTS (SELECT person FROM <other table name> b WHERE a.country = b.country);  Repeat last questions (not in clause)  SELECT country FROM <table name> WHERE country NOT IN (SELECT person FROM <other table name>);  Repeat last questions using the (minus clause)  SELECT country FROM <table name>  MINUS  SELECT country FROM <table name> WHERE person IS NOT NULL;  Write an SQL statement that gives a listing of all the people and the name of the countries they live in. If there is a country code but no country name display “no country named”  SELECT countrycode, NVL(countryname, ‘No Country Named’), people FROM <table name> a, <other table name> b WHERE a.countrycode = b.countrycode (+);  Write an SQL statement that gives a listing of all the countries and the people that live in those countries. If there is a country that does not have any person living in it then display “No Person”  SELECT countrycode, countryname, NVL(people, ‘No Person’) FROM <table name> a, <other table name> b WHERE a.countrycode = b.countrycode (+);  Write an SQL statement that give a listing of the country names and the number of people living in each country  **SELECT countryname, COUNT (\*) FROM table\_name GROUP BY countryname;** |

|  |  |
| --- | --- |
| Cartesian product or CROSS JOIN | Replicates each row from the first table with every row from the second table |
| Equality join also known as equijoin, inner join or a simple join | Creates a join by using a commonly named and defined column |
| Non-equality join | Joins tables when there are no equivalent rows in the tables to be joined |
| Self-join | Joins a table to itself. |
| Outer join | Includes records of a table when there is no matching record in the other table. |
| Set operators UNION, UNION ALL, INTERSECT and MINUS | Combines results from multiple select statements |
| WHERE | In the traditional approach, the WHERE clause indicates which columns should be joined |
| NATURAL JOIN | The keywords are used in the FROM clause to join tables containing a common column with the same name and definition |
| JOIN … USING | The JOIN keyword is used in the FROM clause; combined with the USING clause, it identifies the common column used to join the tables. |
| JOIN … ON | The JOIN keyword is used in the FROM clause. The ON clause identifies the columns used to join the tables |
| OUTER JOIN can be used with LEFT, RIGHT, FULL | Indicates that at least one of the tables doesn’t have a matching row in the other table |
| UNION | Returns the results of both queries and removes duplicates |
| UNION ALL | Returns the results of both queries but includes duplicates |
| INTERSECT | Returns only the rows included in the results of both queries |
| MINUS | Subtracts the second query’s results if they’re also returned in the first query’s result |

