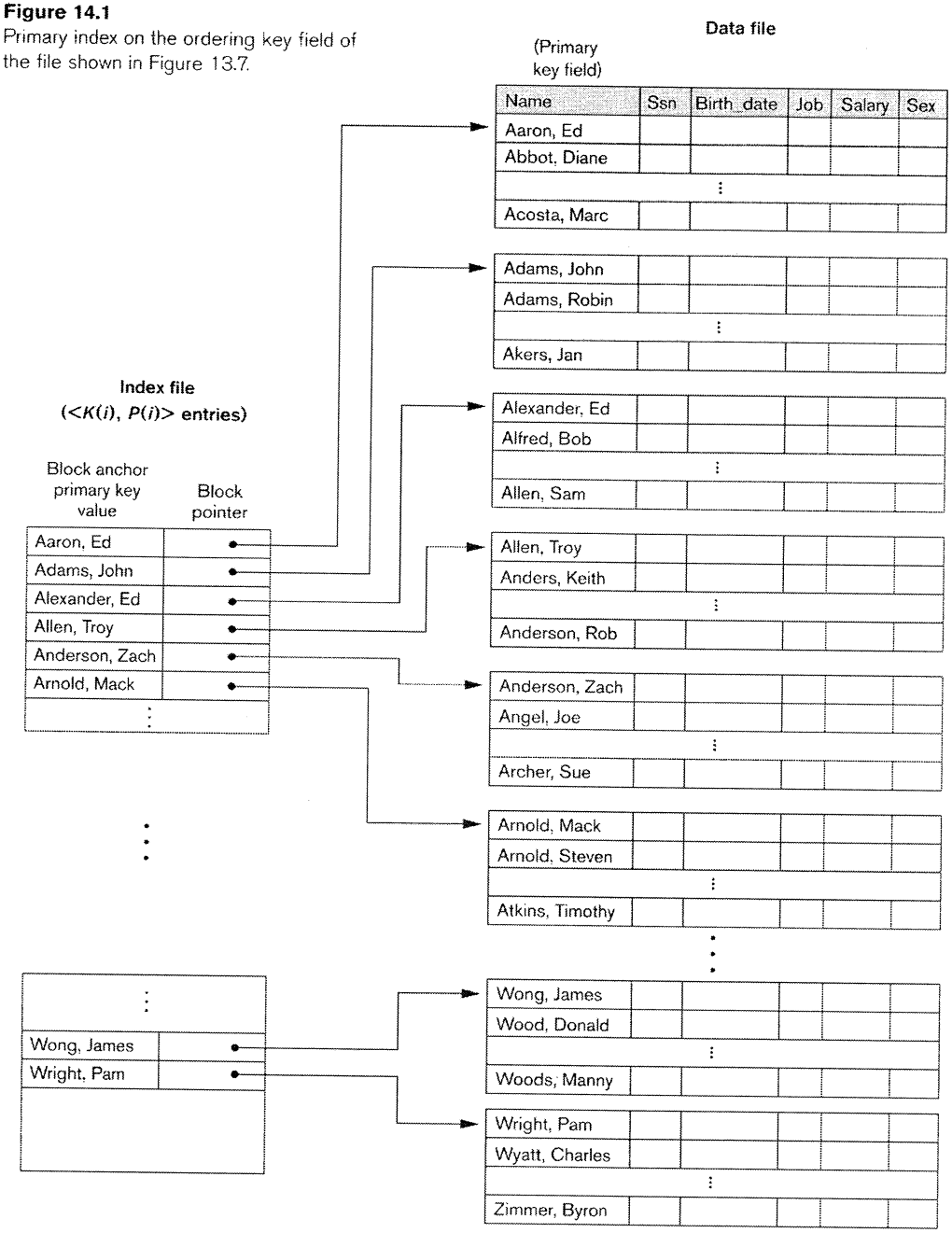
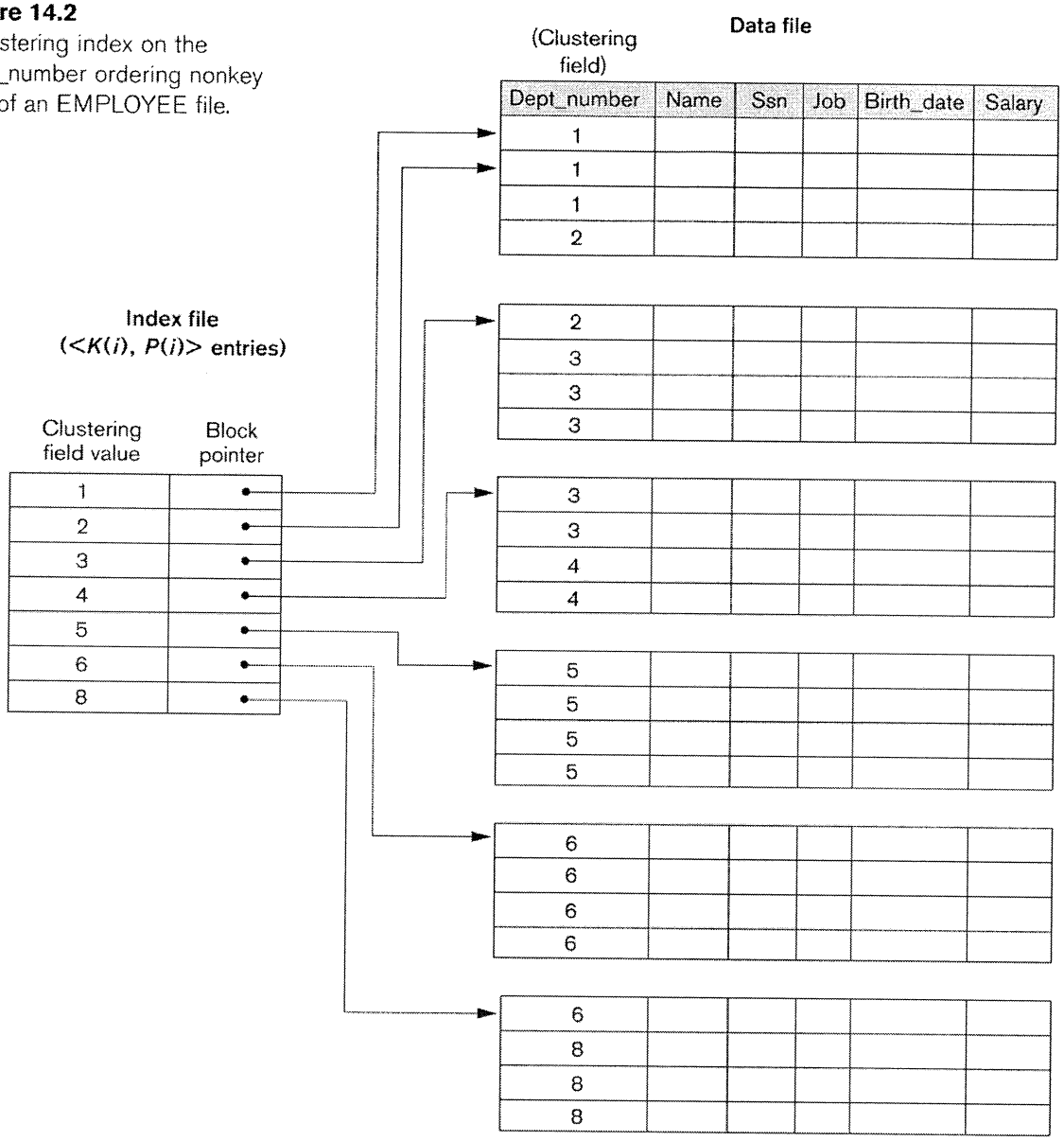
Primary

Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.



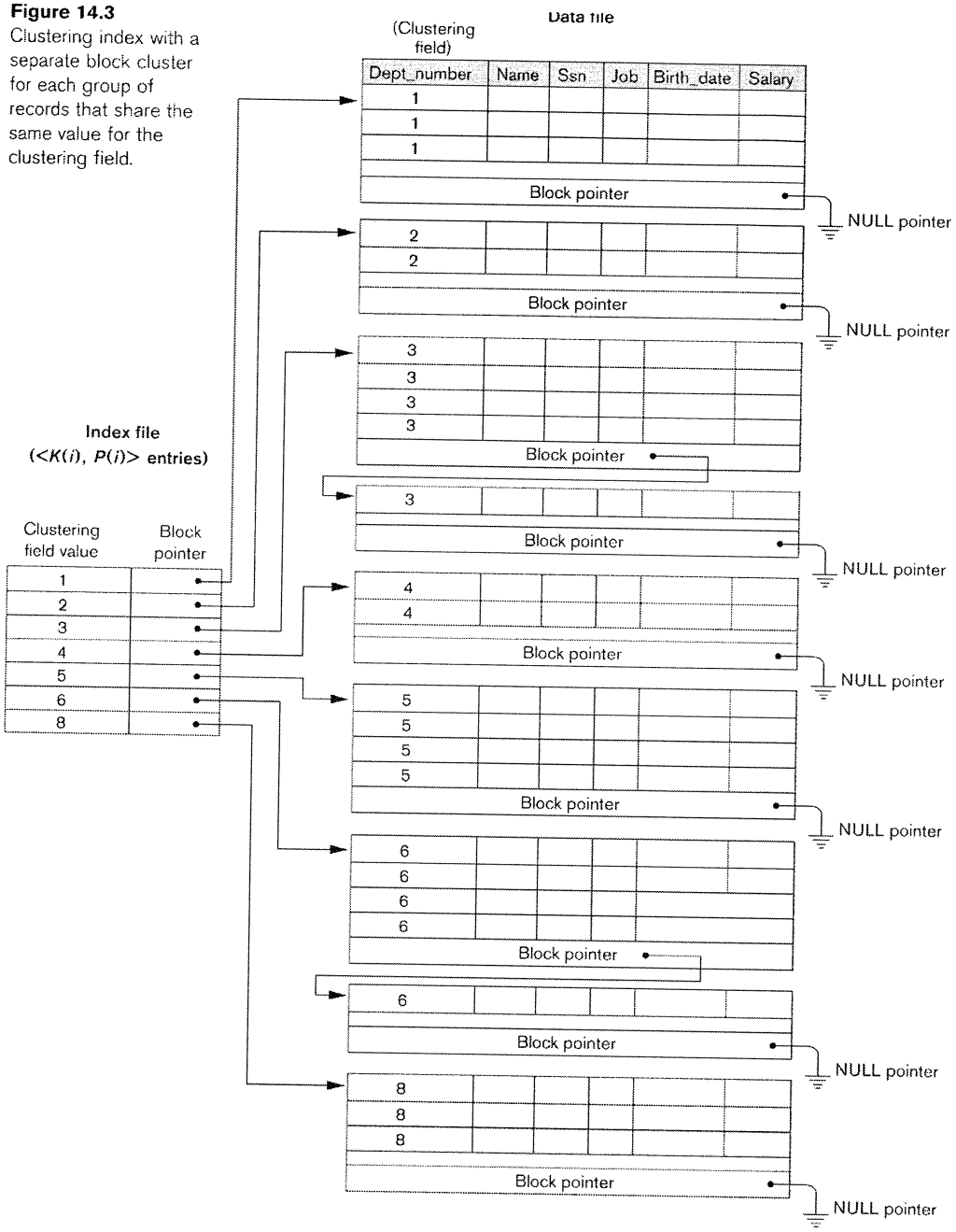
Clustering

Figure 14.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.



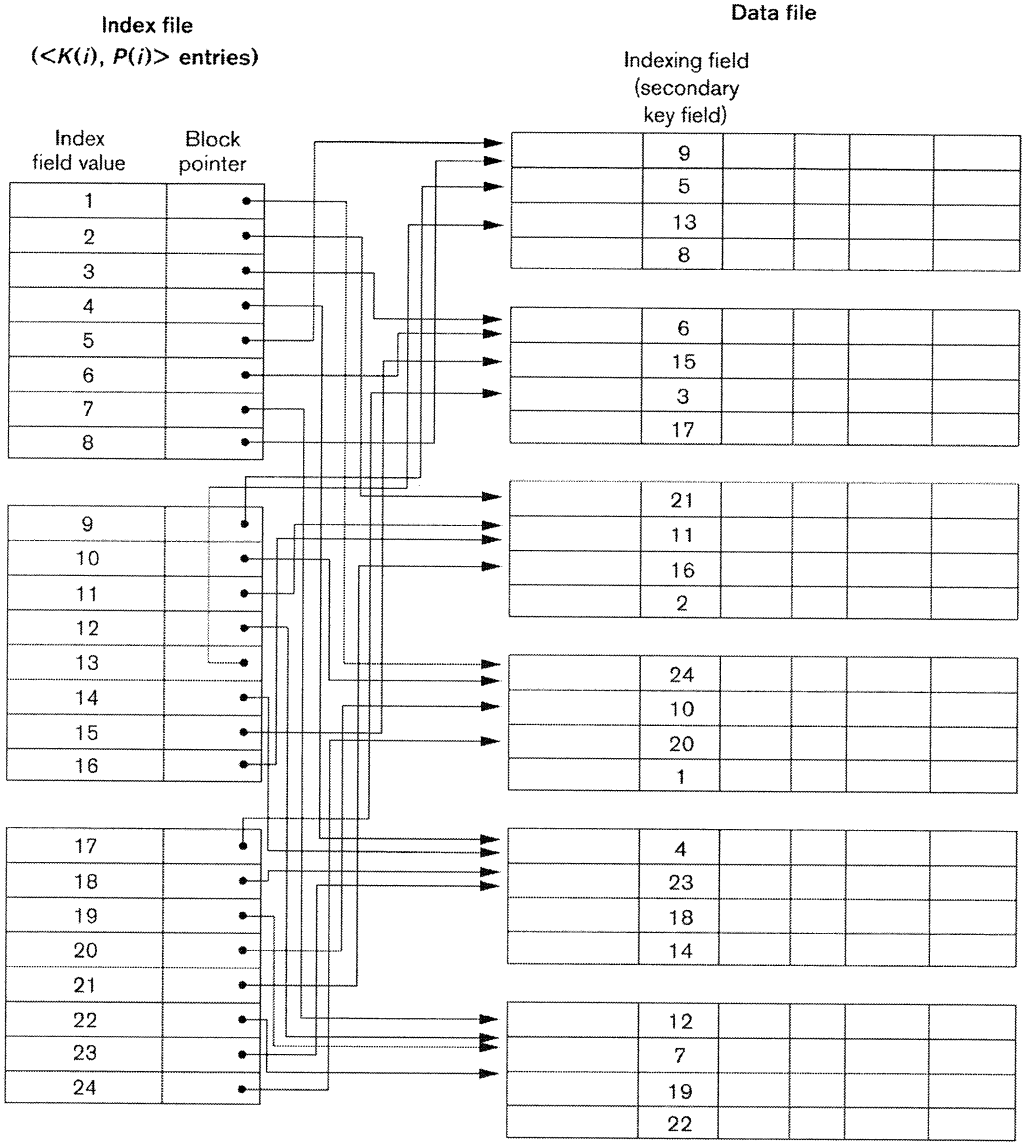
Clustering

Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Dense Secondary Non ordering key

Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



multi level

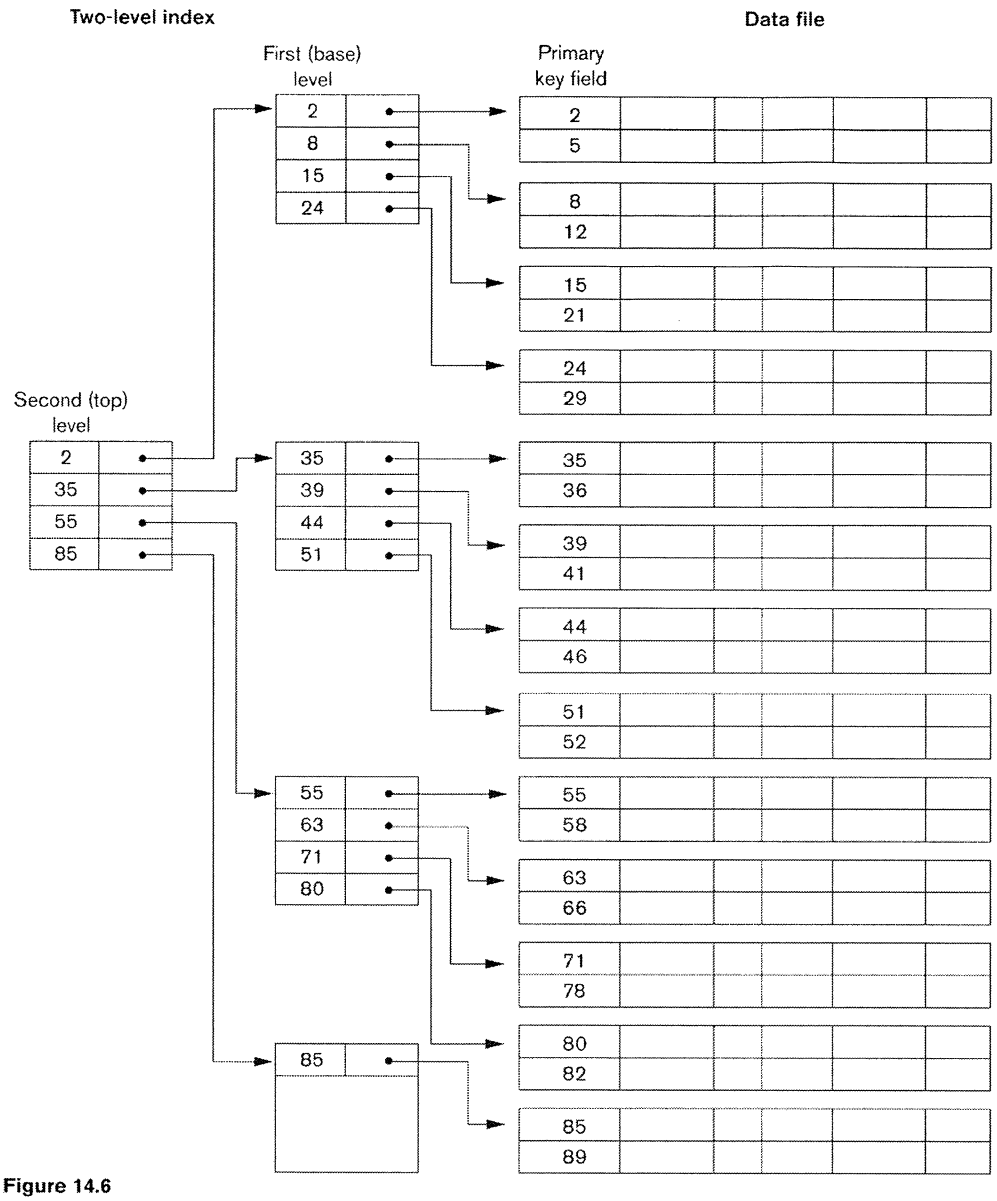
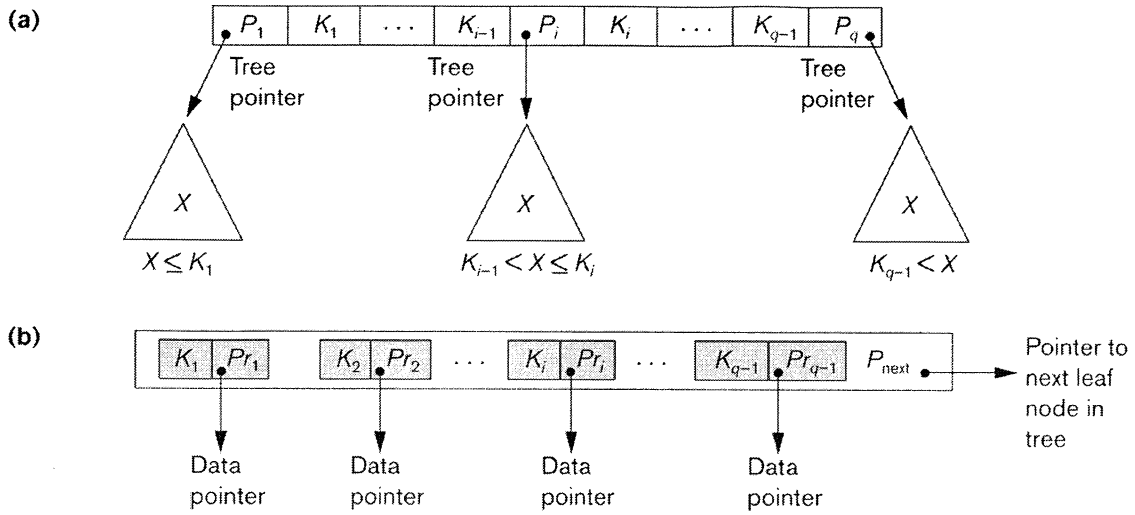


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

B+ Tree

Figure 14.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.



RELATIONAL OPERATOR -- DIVISION (÷)

Applied to two relations R(Z) ÷ S (X)

where attributes of S are a subset of the attributes of R; that $X \subseteq Z$. *** typo on page 163
Let Y be the set of attributes of Z that are not attributes of S. ($Y = Z - X$) → $Z = X \cup Y$.

$R(Z) \div S(X) = T(Y)$ t belongs to T(Y) if t_R appears in R with $t_R[Y] = t$ and $t_R[X] = t_S$ for every tuple t_S in S.

The DIVISION operator has no exact counterpart in SQL. However it can be expressed as a sequence of project, Cartesian product, and minus operations:

$T1 \leftarrow \prod_Y (R)$
$T2 \leftarrow \prod_Y ((S \times T1) - R)$
$T \leftarrow T1 - T2$

Example:

Retrieve the names of employees who work on **ALL** the projects that 'Joyce English' works on.

$ENGLISH \leftarrow \prod_{ssn} (\sigma_{Fname='Joyce' \text{ AND } Lname='English'} (EMPLOYEE))$	//453453453
$ENGLISH\_Pnos \leftarrow \prod_{pno} (ENGLISH \ \|X\|_{ssn=essn} WORKS\_ON)$	//1,2
$SSN\_PNOS \leftarrow \prod_{essn,pno} (WORKS\_ON)$	//all WORKS_ON tuples w/o hours
SSNs (SSN) ← SSN_PNOS ÷ ENGLISH_Pnos	// emps working on 1 and 2
$RESULT \leftarrow \prod_{fname, lname} (SSNs * EMPLOYEE)$	// Natural Join

Use user_ for all tables.

How to read an Oracle SQL Execution Plan?

To execute any SQL statement Oracle has to derive an 'execution plan' . The execution plan of a query is a description of how Oracle will implement the retrieval of data to satisfy a given SQL statement. It is nothing but a tree which contains the order of steps and relationship between them

The basic rules of the execution plan tree are:

- An execution plan will contain a root, which has no parents
- A parent can have one or more children, and its ID will be less than the child(ren's) ID
- A child can have only one parent, it is indented to the right; in case of many children, it will have the same indentation.

The following is a sample execution plan.

Grant table

user_tab_privs

Grantor
Grantee
privledge.
Grantable.
table_name

```
SQL> explain plan for
  2  SELECT e.ssn, e.lname, d.dname
  3  FROM employee e, department d
  4  WHERE e.dno = d.dnumber
  5     AND  e.dno = 5;

Explained.
```

To display the execution plan --

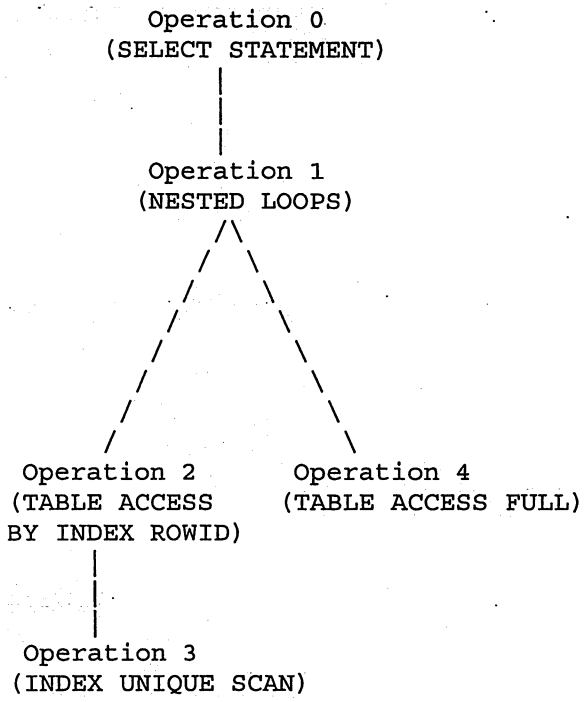
```
SQL> SELECT *
  2  FROM table (dbms_xplan.display(null, null, 'basic'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 25239992

-------------------------------------------------------------------------------
| Id  | Operation                     | Name                |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |
|   1 |  NESTED LOOPS                 |                     |
|   2 |   TABLE ACCESS BY INDEX ROWID | DEPARTMENT          |
|   3 |    INDEX UNIQUE SCAN          | DEPARTMENT_DNUMBER_PK |
|   4 |   TABLE ACCESS FULL           | EMPLOYEE            |
-------------------------------------------------------------------------------
```

user_views
set long 200

Using the rules above, you could say;

- Operation 0 is the root of the tree; it has one child, Operation 1
- Operation 1 has two children, which is Operation 2 and 4
- Operation 2 has one child, which is Operation 3

Below is the graphical representation of the execution plan. If you read the tree; in order to perform Operation 1, you need to perform Operation 2 and 4. Operation 2 comes first; in order to perform 2, you need to perform its Child Operation 3. In order to perform Operation 4, you need to perform Operation 2



- Operation 3 accesses DEPARTMENT table using INDEX UNIQUE SCAN and passes the ROWID to Operation 2
- Operation 2 returns all the rows from DEPARTMENT table to Operation 1
- Operation 1 performs Operation 4 for each row returned by Operation 2
- Operation 4 performs a full table scan (TABLE ACCESS FULL) scan and applies the filter E.DNO=5 and returns the rows to Operation 1
- Operation 1 returns the final results to Operation 0

ICE - Query Tree / Query Graph **WORTH: 50 points**

For the query **"For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor"**,

a. Express the solution in SQL.

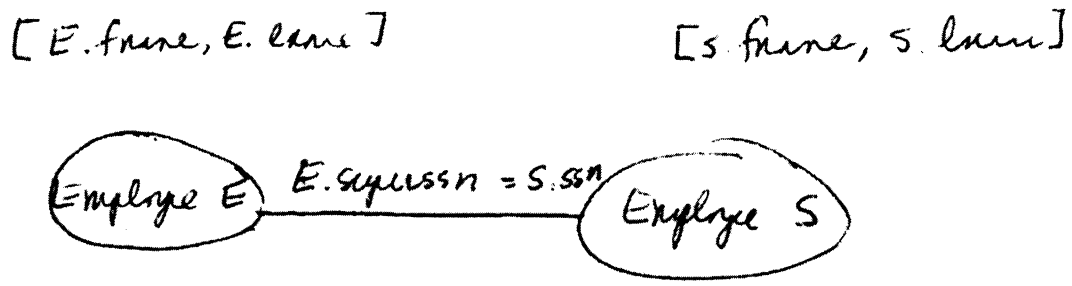
```
SELECT e.fname, e.lname, s.fname, s.lname
FROM  EMPLOYEE E, EMPLOYEE S
WHERE E.super_ssn = S.ssn;
```

b. Express the solution in relational algebra.

$\Pi_{\text{e.fname, e.lname. s.fname, s.lname}}$ $((\rho_E (\text{EMPLOYEE}))$ $|X|_{\text{E.super\_ssn = S.ssn}}$ $(\rho_S (\text{EMPLOYEE}))$

c. Which is appropriate to use for this query, a query tree or a query graph? Why?
Since this is a select –project-join query, a query graph is most appropriate.

d. Depending on your answer on c above, draw the appropriate the structure corresponding to the query.



For the query **"For each project, retrieve the project number, the project name, and the number of employees from the Research Department who work on the project",**

a. Express the solution in SQL.

SELECT pnumber, pname, count(*)
FROM PROJECT, WORKS_ON, EMPLOYEE, DEPARTMENT
WHERE pnumber = pno AND ssn = essn AND dname = 'Research' AND dno=dnumber
GROUP BY pnumber, pname

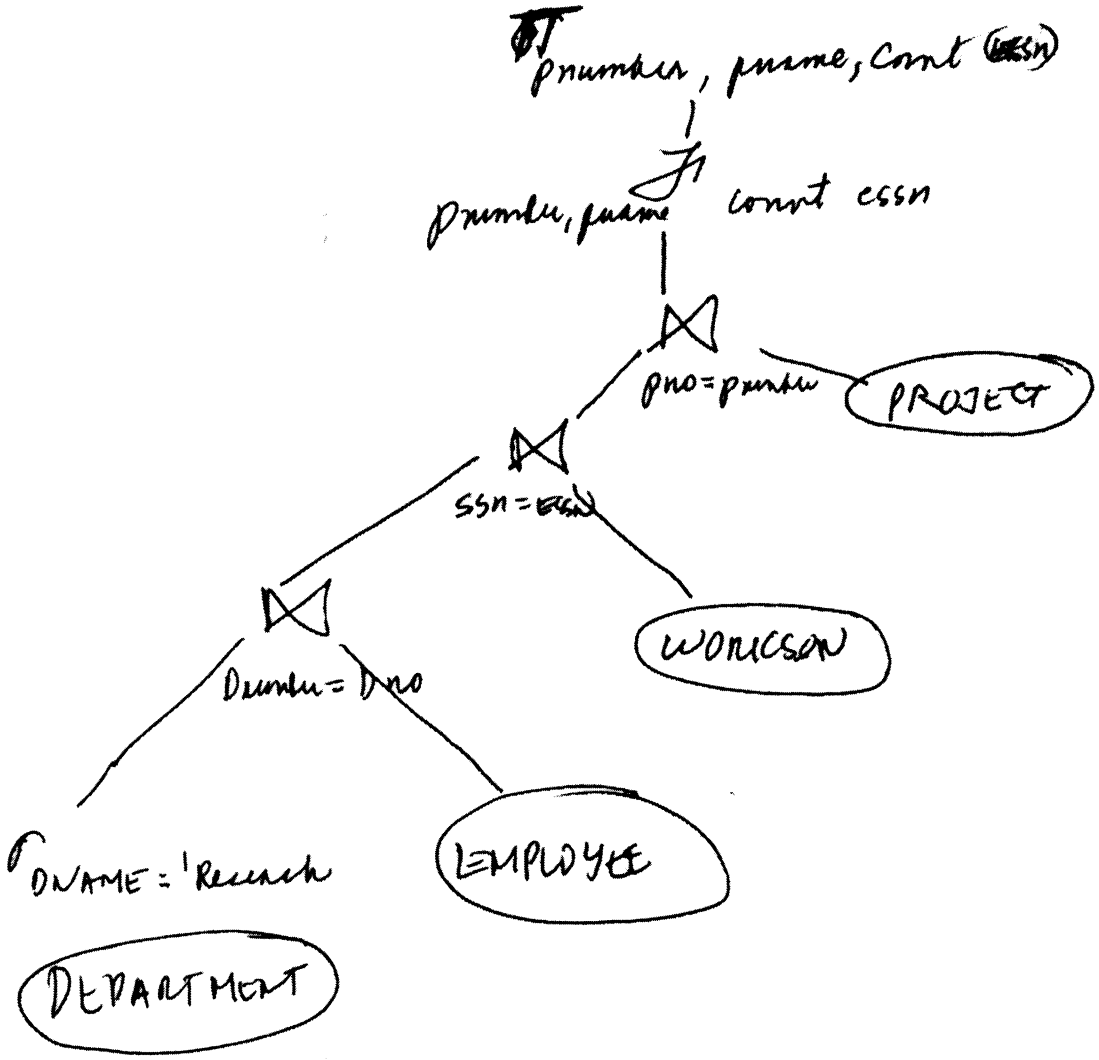
b. Express the solution in relational algebra.

$_{pname, pnumber}F_{COUNT\ essn}(((\sigma_{dname = 'Research'}(Department)) \mid X \mid_{dnumber=dno} Employee) \mid X \mid_{ssn=essn} Works\_On) \mid X \mid_{pno=pnumber} Project)$

c. Which is appropriate to use for this query, a query tree or a query graph? Why?

Since the query involves other operators than just select-project-join, use a query tree.

d. Depending on your answer on c above, draw the appropriate the structure corresponding to the query.



Query Optimization --- JOINS

- Inner Join – tuples without a matching tuple are eliminated from the JOIN result

Type of Inner Join	Relational Algebra Notation	Definition/Description
Theta Join	R $\|X\|_{\text{<join condition>}}$ S	produces all combinations of tuples from R and S that satisfy the join condition
Equi Join	R $\|X\|_{\text{<join condition>}}$ S R $\|X\|_{\text{<join attributesR>, <join attributesS>}}$ S	produces all combinations of tuples from R and S that satisfy the join condition with only equality comparisons
Natural Join	R $*_{\text{<join condition>}}$ S R $*_{\text{<join attributesR>, <join attributesS>}}$ S	Same as EQUIJOIN EXCEPT join attributes of S are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified

ORACLE NOTATION

SELECT ...
FROM R, S
WHERE <join condition>

SELECT ...
FROM R JOIN S on <join_condition>

EXAMPLE: Retrieve the name and address of all employees who work in Research department.

RELATIONAL ALGEBRA:

$\Pi_{fname,lname,\ address}$ (EMPLOYEE $|X|_{dno=dnunber}$ ($\sigma_{dname='Research'}$ DEPARTMENT))

ORACLE:

```
SELECT lname, fname, address
FROM EMPLOYEE, DEPARTMENT
WHERE dno=dnumber AND dname='Research';

SELECT lname, fname, address
FROM EMPLOYEE, (SELECT dnumber
                FROM DEPARTMENT
                WHERE dname='Research')
WHERE dno=dnumber;

SELECT lname, fname, address
FROM EMPLOYEE JOIN DEPARTMENT ON dno=dnumber
WHERE dname='Research';
```

- Outer Join – tuples in R or in S or in both are included in the JOIN result regardless whether or not there are matching tuples.

Type of Outer Join	Relational Algebra Notation ***ORACLE NOTATION***	Definition/Description
Left Outer Join	R]X$_{<join condition>}$S *SELECT ...* *FROM R, S* *WHERE R.a=S.b(+)* SELECT ... FROM R LEFT OUTER JOIN S on R.a=S.b	Keeps every tuple in the first/left relation R and if no matching tuple in S, attributes of S in the join result are filled or padded with NULL values *Example: List all employees and the names of the department they manage; if they do not manage one, indicate this with NULL.* *SELECT lname, dname* *FROM employee, department* *WHERE ssn=mgrssn(+);* *SELECT lname, dname* *FROM employee LEFT OUTER JOIN department ON ssn = mgrssn;*
Right Outer Join	R X[$_{<join condition>}$S *SELECT ...* *FROM R, S* *WHERE R.a(+)=S.b* SELECT ... FROM R RIGHT OUTER JOIN S on R.a=S.b	Keeps every tuple in the second/right relation S and if no matching tuple in R, attributes of R in the join result are filled or padded with NULL values
Full Outer Join	R]X[$_{<join condition>}$S SELECT ... FROM R FULL OUTER JOIN S on R.a=S.b	Keeps all tuples in both the left and right relations when no matching tuples are found, padding them with NULL as needed

Methods for Implementing NATURAL JOIN – R $|X|_{A=B}$ S

Method	Description	Cost
J1: Nested-Loop Join (Nested-Block Join)	*For each block in R (outer loop)* *Retrieve a record from R* *Retrieve every block in S (inner loop)* *Retrieve a record from S* *Test whether records satisfy the join condition*	b_R + (ceiling (b_R/(n_B-2)) * b_S) + ((js * |R| * |S|)/bfr_{RS}) *//number of blocks accessed fro outer loop file R +* *Number of blocks accessed for inner loop file S +* *Number of blocks of join result written back to disk* *//* *n_B = number of buffers in MM to implement join* *2 in (n_B – 2): 1 buffer for inner loop and 1 buffer for result ofjoin* *js = join selectivity = number of tuples in the join result* js = | (R $|X|_c$ S)| / |R X S| = | (R $|X|_c$ S)| / (|R| * |S|) If no condition c, js = 1 If no tuples satisfy join, js = 0 0 <= js <= 1 If A is key of R, js = 1/|R| If B is key of S, js =1/|S| **<u>USE file with fewer blocks as the outer loop!</u>**
J2: Single-loop Join(Using an access structure to retrieve matching records)	An index or hash key exists for one of the join attributes say B of S *For each block in R* *Retrieve a record from R* *Use the access structure for S to retrieve all matching records s from S that satisfy the join condition*	Use for the loop, the file (say S) that is either the smaller or the file with high join selection factor (i.e. has a match for every record) and an index exists. $b_S + (r_S + (x_B + 1))$
J3: Sort-Merge	R and S are sorted by the value of the join attributes Scan both files concurrently in order of the join attributes, match the records that have the same values for A and B	Case 1: R and S are sorted by join attribute $b_R + b_S$ Case 2: R and S are not sorted $b_R + b_S + b_R \log_2 b_R + b_S \log_2 b_S$

Example for J1: EMPLOYEE $|X|_{dno = dnumber}$ DEPARTMENT

Use DEPARTMENT file in the outer loop since it has smaller number of blocks

Variable	Meaning/Definition	Value
n_B	Number of buffers in MM	7
bfr_{RES}	Blocking factor for resulting join file	4
js	Join Selectivity	1/50
DEPARTMENT (D) FILE		
r_D	Number of records	50
b_D	Number of blocks	10
primary index on dnumber,		
x	Number of index levels	1
EMPLOYEE (E) FILE		
r_E	Number of records	6000
b_E	Number of blocks	2000

b_D+ (ceiling (b_D/(n_B-2)) * b_E) + b_{RES} = 10 + (ceiling(10/(7-2))*2000) + ((js * |D| * |E|)/bfr_{RS})

= 10 + (ceiling(10/5) * 2000) + ((1/50 * 50 * 6000)/4)

= 10 + 2* 2000 + 1500

= 4010 blocks + 1500 = 5510 blocks

California State University, San Bernardino
School of Computer Science & Engineering

LAB 01 – Subqueries

Create a subdirectory CSE580/LAB01

For each of the queries below,

1. **save your SQL statement in a text file named LAB01_Q#.sql where # is the number of the query in the list below.**
2. **Run your query to see if it is correct.**

Once you have tested each of your queries, submit a spool file using filename yourlastname_firstinitialLAB 01 that will contain the contents of each query and the result of the execution. See the example in the box below where user is Mendoza_J.

Email the spooled file by Sunday, January 12 by midnight.
SUBJECT: CSE580 – LAB01

```
spool Mendoza_JLAB01.lst
get LAB01_Q1.sql
/

.....
get LAB01_Q?.sql
/

exit
```

Each of the queries below is worth 5 pts.

1. Write a query to display the last name and birthdate of any employee in the same department as Wallace. Exclude Wallace.
2. Create a query to display the employee numbers and lastnames of all employees who earn more than the average salary. Sort the results in ascending order of salary.
3. Write a query that displays the employee ssn and last name of all employees who work in a department with any employee whose last name contains an l. (letter L).
4. Display the last name, department number, and gender of all employees whose department is located in Bellaire.
5. Display the last name and salary of every employee who reports to Wallace.
6. Display the department number, last name and gender for every employee in the Administration department.

EXTRA CREDIT: (10 pts)

Display the employee ssn, last name, and salary of all employees who earn more than the average salary and who work in a department with any employee with an 'n' in their name.

```
select lname, bdate
from jmendoza.employee
where dno = (select dno
         from jmendoza.employee
         where lname = 'Wallace')

         minus

             (select lname, bdate
             from jmendoza.employee
             where lname = 'Wallace')
```

James

1

```
select ssn, lname
from jmendoza.employee
where salary > (select avg(salary)
        from jmendoza.employee)
order by salary
```

2

```
select ssn, lname
from jmendoza.employee
where dno in (select dno
          from jmendoza.employee
          where lower(lname) like '%l%')
```

3

```
select e.lname, e.dno, e.gender
from jmendoza.employee e, jmendoza.department d, jmendoza.dept_locations l
where e.dno = d.dnumber and d.dnumber = l.dnumber and l.dlocation = 'Bellaire'
```

4

```
select e.lname, e.salary
from jmendoza.employee e, jmendoza.employee d
where e.superssn = d.ssn and d.lname = 'Wallace'
```

5

```
select dno, lname, gender
from jmendoza.employee, jmendoza.department
where dno = dnumber and dname = 'Administration'
```

6

```
SELECT lname, bdate
FROM employee
where dno = (SELECT dno
             FROM employee
             WHERE initcap(lname) = 'Wallace'
             )
AND ssn <> (SELECT dno
            FROM employee
            WHERE upper(lname) = 'WALLACE'
            )
/
SELECT ssn, lname
FROM employee
WHERE salary > (SELECT AVG(salary)
                FROM employee)
ORDER BY salary
/
SELECT ssn, lname
FROM employee
WHERE dno IN (SELECT dno
              FROM employee
              WHERE lname LIKE '%l%')

/
SELECT lname, dno, gender
FROM employee
WHERE dno IN (SELECT dnumber
              FROM dept_locations
              WHERE initcap(dlocation) = 'Bellaire'
             )

/
SELECT lname, salary
FROM employee EMP
WHERE superssn = (SELECT ssn
                  FROM employee Super
                  WHERE lname = 'Wallace'
                 )
/
SELECT dno, lname, gender
FROM employee
WHERE dno = (SELECT dnumber
             FROM department
             WHERE dname = 'Administration'
             )
/
```

1

Mendoza

2

3

4

5

6

CSE 580 W2014 LAB 02 – VIEWS

1. CREATE a subdirectory LAB02 under CSE580 directory.

   For this Lab exercise you will be using the COMPANY DB tables created by user jmendoza. This lab must be finished by 7:30 PM and email me the script file no later than 10 PM tonight, March 6 (Thursday).

2. SUBMIT a script file that contains the answers to the following questions/tasks.

   TASK 01 – Create a view called Employees_Vu based on the employee social security number, employee last name and department number from the EMPLOYEE table created by jmendoza. Change the heading for the employee name to Employee, employee social security number to Employee Id, and department number to Department Id.

   TASK 02 – Display the contents of Employees_Vu.

   TASK 03 – SELECT the view name and text from the USER_VIEWS data dictionary view.
   NOTE: Make sure that you display the entire SELECT clause that contains the definition of the view.

   TASK 04 – Using the EMPLOYEES_Vu view, enter a query to display all employee names and department numbers.

   TASK 05 - Create a view named DEPT04_Vu that contains the employee social security number, employee name (lastname, firstname) and department number for all employees in department no 4. Label the view columns, EmpNo, Employee and DeptNo. Do not allow an employee to be reassigned to another department through the view.

   TASK 06 - Display the structure and contents of the Dept04_Vu.

   TASK 07 - Attempt to reassign Ahmad Jabbar to department 5.

   TASK 08 - Create a view called Salary_Vu based on the employee last names, department names, salaries. Use the appropriate COMPANY DB tables created by user jmendoza. Label the columns, Employee, Department and Salary respectively.

```
SQL> start task1.sql
SQL> ;
  1  CREATE VIEW Employees_Vu("Employee Id", Employee, "Department Id")
  2  as select lname, ssn, dno
  3*    from durbach.employee
SQL> /

View created.

SQL> select *
  2  from Employees_Vu;

Employee Id     EMPLOYEE  Department Id
--------------- --------- -------------
Smith           123456789             5
Wong            333445555             5
Zelaya          999887777             4
Wallace         987654321             4
Narayan         666884444             5
English         453453453             5
Jabbar          987987987             4
Borg            888665555             1

8 rows selected.

SQL> set long 200
SQL> select view_name, text
  2  from user_views;

VIEW_NAME
------------------------------
TEXT
--------------------------------------------------------------------------------
EMPLOYEES_VU
select lname, ssn, dno
   from
durbach.employee

SQL> select Employee, "Department Id"
  2  from Employees_Vu;

EMPLOYEE  Department Id
--------- -------------
123456789             5
333445555             5
999887777             4
987654321             4
666884444             5
453453453             5
987987987             4
888665555             1

8 rows selected.

SQL> start task5.sql
SQL> ;
  1  CREATE VIEW DEPT04_Vu (EmpNo, Employee, DeptNo)
  2  as select ssn, lname || ', ' || fname, dno
  3     from durbach.employee
  4     where dno = 4
  5* WITH CHECK OPTION CONSTRAINT DNO04_VU_CK
SQL> /

View created.

SQL> desc Dept04_Vu;
 Name                                      Null?    Type
```

```
 ---------------------------------------- -------- ----------------------------
 EMPNO                                    NOT NULL CHAR(9)
 EMPLOYEE                                          VARCHAR2(32)
 DEPTNO                                            NUMBER

SQL> select *
  2  from Dept04_Vu;

EMPNO     EMPLOYEE                            DEPTNO
--------- -------------------------------- ----------
999887777 Zelaya, Alicia                            4
987654321 Wallace, Jennifer                         4
987987987 Jabbar, Ahmad                             4


SQL> update Dept04_Vu
  2  set DeptNo = 5
  3  where employee = 'JABBAR, AHMAD';
       *
ERROR at line 1:
ORA-01031: insufficient privileges


SQL> start task8.sql
SQL> ;
  1  CREATE VIEW Salary_Vu (Employee, Department, Salary)
  2  as select lname, dname, salary
  3     from durbach.employee, durbach.department
  4*    where dno = dnumber
SQL> /

View created.

SQL> spool off
```

DATA DICTIONARY TABLE/VIEW	DESCRIPTION	COMMENTS
USER_TABLES table_name	describes the relational tables owned by the current user. Its columns (except for OWNER) are the same as those in ALL_TABLES	http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_2105.htm#i1592091 The above URL gives the columns and meaning of these columns in USER_TABLES. owner – table name,
USER_CATALOG table_name ... – type	lists tables, views, clusters, synonyms, and sequences owned by the current user. Its columns are the same as those in "ALL_CATALOG".	This view does not display the OWNER column. http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_1028.htm#i1575120 The above URL gives the columns and meaning of these columns in USER_CATALOG
USER_COL_PRIVS	describes the column object grants for which the current user is the object owner, grantor, or grantee.	http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_1038.htm The above URL gives the columns and meaning of these columns in USER_COL_PRIVS.
USER_COL_PRIVS_MADE	describes the column object grants for which the current user is the object owner. Its columns (except for OWNER) are the same as those in ALL_COL_PRIVS_MADE.	http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_1039.htm#i1575578 The above URL gives the columns and meaning of these columns in USER_COL_PRIVS_MADE
USER_CONS_COLUMNS	describes columns that are owned by the current user and that are specified in constraint definitions. Its columns are the same as those in "ALL_CONS_COLUMNS".	http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_1042.htm#i1575870 The above URL gives the columns and meaning of these columns in USER_CONS_COLUMNS
USER_VIEWS	describes the views owned by the current user. Its columns (except for OWNER) are the same as those in ALL_VIEWS.	http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_2118.htm#i1593583 The above URL gives the columns and meaning of these columns in USER_VIEWS

Granting Privileges on Columns

You can grant INSERT, UPDATE, or REFERENCES privileges on individual columns in a table.

Caution:
Before granting a column-specific INSERT privilege, determine if the table contains any columns on which NOT NULL constraints are defined. Granting selective insert capability without including the NOT NULL columns prevents the user from inserting any rows into the table. To avoid this situation, ensure that each NOT NULL column can either be inserted into or has a non-NULL default value. Otherwise, the grantee will not be able to insert rows into the table and will receive an error.

The following statement grants the INSERT privilege on the acct_no column of the accounts table to user psmith:

```
GRANT INSERT (acct_no) ON accounts TO psmith;
```

In the following example, object privilege for the ename and job columns of the emp table are granted to the users jfee and tsmith:

```
GRANT INSERT(ename, job) ON emp TO jfee, tsmith;
```

Revoke insert on emp from jfee;

Revoking ?

Limit Propigation ?

with grant option.

Select
insert
update
delete
references
alter
index
all

All —

USER —

DBA_TAB_PRIVS

DBA_TAB_PRIVS describes all object grants in the database.

Related View

USER_TAB_PRIVS describes the object grants for which the current user is the object owner, grantor, or grantee.

Column	Datatype	NULL	Description
GRANTEE	VARCHAR2(30)	NOT NULL	Name of the user to whom access was granted
OWNER	VARCHAR2(30)	NOT NULL	Owner of the object
TABLE_NAME	VARCHAR2(30)	NOT NULL	Name of the object. The object can be any object, including tables, packages, indexes, sequences, and so on.
GRANTOR	VARCHAR2(30)	NOT NULL	Name of the user who performed the grant
PRIVILEGE	VARCHAR2(40)	NOT NULL	Privilege on the object
GRANTABLE	VARCHAR2(3)		Indicates whether the privilege was granted with the GRANT OPTION (YES) or not (NO)
HIERARCHY	VARCHAR2(3)		Indicates whether the privilege was granted with the HIERARCHY OPTION (YES) or not (NO)

SOURCE: http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_5046.htm#i1627646

SELECT COST FUNCTIONS

COST – Access Cost to Secondary Storage = number of block transfers between disk and main memory (MM) buffers

CATALOG Information

VARIABLE	DEFINITION	COMMENTS
r	File size= number of records or tuples	
R	Record or tuple size in bytes	
b	Number of blocks needed by the file	
bfr	Blocking factor for file	
	Primary file organization • Ordered • Unordered • With/without primary index • With/without clustering index • Hashed index on a key attribute	
	Indices Available • Primary • Secondary • Clustering • Indexing attributes	
x	Number of levels of a multi-level index	
b_{I1}	Number of first-level index blocks	
d	Number of distinct values of an attribute	
sl	Selectivity = fraction of records satisfying an equality condition on the attribute	usually a decimal
s	Selection cardinality = average number of records that satisfy the equality condition on the attribute = sl * r	Key attribute: d = r, sl = 1/r s= 1 Non-key attribute: (assume uniform distribution of d values among r records) sl = 1/d s = r/d

$d = R/s$

COST FUNCTIONS for SELECT

SELECT ALGORITHM	DESCRIPTION	COST FORMULA
S1: Linear Search	Search all file blocks that satisfies select condition Search: equality on a key attrib Case 1: found Case 2: not found	$C_{S1a} = b$ $C_{S1b} = b/2$ $C_{S1b} = b$
S2: Binary Search	Binary Search	$C_{S2} = \log_2 b +$ ceiling (s/bfr) – 1 Inequality or duplicate values $C_{S2} = \log_2 b$: equality on unique key
$S3_a$: Using a primary index to get one record	Get one disk block at each index level + one disk block from data file	$C_{S3a} = x + 1$
$S3_b$: Using a hash key index to get one record		$C_{S3b} = 1$
S4: Using an ordering index to get many records		Inequality: $C_{S4} = x + (b/2)$
S5: Using a clustering index to get many records	Get one disk block at each index level which gives the address of the first file disk block in the cluster and s records will satisfy the equality condition	$C_{S5} = x +$ ceiling(s/bfr)
S6: Using a secondary B+ tree index	Equality:Secondary Index on Unique Key Equality:Secondary Index on Non-unique Key (Non clustered) Inequality: search half of first level index + half of file records via the index	$C_{S6} = x + 1$ $C_{S6a} = x + 1 + s$ $C_{S6b} = x +$ $(b_{I1}/2) + r/2$
S7: Conjunctive Select	Any of S1-S6 to retrieve the records that satisfy one condition and check in the mm buffers where each retrieved record satisfies the remaining conditions of the conjunction Multiple Indexes: Use the indexes to retrieve the record pointers or record ids and use the intersection of the record ptrs in the mm to retrieve the resulting records	

S8: Conjunctive Select Using Composite Index	Same as S3a, S5 or S6a depending on the index type	

CHAPTER 15 – Functional Dependencies and Normalization: Relational DBs

CHAPTER GOAL:

- Learn how to measure the goodness of relational db design (relation schemas)

Two levels at looking at the goodness of relation schemas:

1. Logical (Conceptual) Level
    - How good users interpret the relation schemas (base + virtual)
    - How good users interpret the relation schema attribute

    ⇨ Users clearly understand the meaning of data
    ⇨ Users can correctly formulate queries

2. Implementation (Physical Storage) Level
    - How tuples in base relations are stored
    - How tuples in base relations are updated

Goals of RDB:

1. Preserve information
2. Minimize redundancy`

Two approaches to RDB :

- Design By Synthesis (Bottom-Up)
    - Start: Attributes
    - Output: Relation Schemas
    - Process: Determine the binary relationships among attributes
- Design By Analysis (Top-Down)
    - Start: Relation Schemas
    - Output: Decomposed Relation Schemas
    - Process: Analyze individual relation schemas and continue to decompose until all desirable properties (no redundancy, no update/insert/delete anomalies) are met

4 Guidelines for Good RDB Design →

- No anomalies that result in redundant data during insertion/modification of a relation and cause accidental loss of information during a delete from a relation.
- Avoid too many NULLS → waste of storage space and difficulty in performing select, aggregation and joins.
- Avoid generating spurious tuples during joins on base relations by not using join attributes that may not represent a proper FK-PK relationship.

FD $X \rightarrow Y$ R(X,Y)

- Constraint on two sets of attributes, X and Y
- Whenever two different tuples in R have the same X-value, then these tuples must also have the same Y-value
- If a constraint on R says there can be only one tuple for a given X-value (X is then a candidate key) then $X \rightarrow Y$ for any subset of attributes Y of R. [RATIONALE: definition of key constraint!]
- FD is a function of the semantics or meaning of the attributes
- Constraints **must hold at all times.**
- Property of a relation schema R.

Kinds of Keys

- Superkey
- Key
- Candidate Key
- Primary Key
- Prime Attribute (PA)
- Non-prime Attribute (NP)

Normalization

- Process
- Series of tests to certify whether the relation is in a certain NF.
- Based on FDs and PK
- 1NF – no MVA and no composite attributes OR only SVA or atomic attributes
- 2NF – 1NF + no partial FD ie. Every non-prime attribute must be fd on full key.

2NF Definition vs General Definition of 2NF
(PK of R) vs (any key of R)

- 3NF – 2NF + no transitive FD ie. There should be no FD of the form NP → NP

3NF Definition vs General Definition of 3NF
(No NP → NP) vs nontrivial FD X→ A holds for R → X is a superkey of R or A is a prime attribute of R
General Definition : no need to prove 2NF to be in 3NF
Violation of General Definition of 3NF * NP → NP //transitive dependency * proper subset of key of R → NP //partial fd that violates 3NF and 2NF

REFER to TABLE 15.1, page 525

BCNF (Boyce-Codd NF): for a nontrivial fd X → A in R, X is a superkey of R.

- Only requires that some key be on the left hand side of any nontrivial FD, not that all keys are contained on the left side.
- How to Decompose a BCNF Violation
  Given: R with a set of FDs F
  OUTPUT: Decomposition of R into a collection of BCNF relations

  METHOD:
  a. If there are BCNF violations, say X → Y.
  b. Compute X^+.
  c. Choose R1 = X^+ as one relation schema
  d. Choose R2 = {X U attributes of R not in X^+}
  e. Determine the FDs for R1 and R2.
  f. Recursively decompose R1 and R2 as long as there are any BCNF violations.

  EXAMPLE:

  Given: R (Title, Year, StudioName, President, PresAddr) //R tells about the movie, its studio, the president of the studion and address of the president of the studio.

  FD1: Title, Year --> StudioName

  FD2: StudioName --> President

  FD3: President --> PresAddr

  Key: Title, Year

  FD2,FD3 violate BCNF.

  Decompose R by starting with FD2: StudioName --> President

  Add to the RHS of FD2 any other attributes in the closure of StudioName (which is PresAddr) ===> R1 (StudioName, President, PresAddr)

  FD: StudioName --> president, PresAddr

  R is decomposed into (Title, Year, StudioName) and (StudioName, President, PresAddr)

  Title, Year --> Studio Name StudioName --> President President--> PresAddr

  Key: Title, Year .. BCNF Key: StudioName but President --> PresAddr violates BCNF

  Decompose StudioName, President, PresAddr into

  (President, PresAddr) President --> PresAddr ... BCNF

  (StudioName, President) StudioName --> President ... BCNF

1NF:

A relation is in 1NF if it has no multivalued attribute **and** no composite attribute.

Or

A relation is in 1NF if all the attributes are single-valued **and** atomic.

A relation R which is not in 1NF because it has a multivalued attribute can be converted into 1NF in one of three ways: (see pages 520-521)

EXAMPLE: From the COMPANY DB: DEPARTMENT(Dname, Dnumber, Mgr_SSN, {dlocations})

Method 1 (Preferred) : Remove the multivalued attribute locations and place in a separate relation Dept_locations
together with the primary key (Dnumber) of Department. The PK of Dept_locations is the combination of both Dnumber and location.

DEPT_LOCATIONS (Dnumber, Dlocation)

Method 2: If a maximum number of values is known for the attribute, e.g. 3 locations, replace the multivalued dlocations with three atomic values dlocation1, dlocation2, dlocation3.

DEPARTMENT

Dname	Dnumber	Mgr_SSN	Location1	Location2	Location3
Research	5	33344555	Bellaire	Sugarland	Houston
Administration	4	987654321	Stafford	NULL	NULL
Headquarters	1	888665555	Houston	NULL	NULL

Method 3: Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT. The primary key is the combination {Dnumber, Dlocation}

DEPARTMENT: Composite Key {Dnumber, Dlocation}

Dname	Dnumber	Mgr_SSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

What is wrong with 1NF of DEPARTMENT?

1. Redundancy
2. Anomalies
    - *INSERT ANOM ALY: Cannot insert a new department that does not have a location yet.*
    - *DELETE ANOMALY: Does not apply for this example.*
    - *UPDATE ANOMALY: Changing the manager of Research will involve changing all 3 tuples. Changing only in one tuple will result in inconsistency.*

2NF

EXAMPLE 1: *EMP_DEPT* (Ename, SSN, Bdate, Address, Dnumber, Dname, DMgr_SSN)

FD1: SSN → Ename, Bdate, Address	*An employee has a name, bdate and address and ssn uniquely identifies an employee.*
FD2: SSN → Dnumber	*An employee works for one department.*
FD3: Dnumber → Dname, DMgr_SSN	*A department has a name and manager and a unique dnumber.*

EMP_DEPT is in 2NF since it is in 1NF (only atomic and single-valued attributes) and

its PK is SSN which has only one attribute and therefore cannot violate full-functional dependency!

What is wrong if EMP_DEPT is left in 2NF?

Refer to EMP_DEPT Table in Figure 15.4, page 508

- Too much redundancy: Since a department can have many employees, the department info (dname, dmgr_ssn) is repeated for every employee that works in a department.
- DELETE Anomaly: If we delete employee James Borg, then we accidentally lose information on the Headquarters Department.
- UPDATE Anomaly: If we change the name of Administration Department and there are x employees working for Administration Department, we must change the name of the Administration department in these x tuples. Failure to do so will result in inconsistent data.
- INSERT ANOMALY: We cannot add a new department unless that department has at least one employee!

EXAMPLE 2:

EMP_PROJ (SSN, Pnumber, Hours, Ename, Pname, Plocation)

FD1: SSN → Ename	*An employee has a name and SSN uniquely identifies an employee.*
FD2: Pnumber → Pname, Plocation	*A project has a name and location and a unique number.*
FD3: SSN, Pnumber → Hours	*An employee works in many projects and a project has many employees. We need to track the number of hours worked by an employee in a project.*

EMP_PROJ is in 1NF since all attributes are atomic and single-valued!

EMP_PROJ is not in 2NF since FD1 and FD2 are partial functional dependencies!

PK of EMP_PROJ: composite of SSN, Pnumber i.e {SSN, Pnumber}

Prime Attributes (PA): SSN, Pnumber

Non Prime Attributes (NPA): Hours, Ename, Pname, Plocation

FD1: SSN → Ename is a partial FD since the right hand side (RHS) of the FD, Ename is NPA and the left hand side (LHS) of the FD, SSN is part of key {**SSN**, Pnumber}

Similarly, FD2: Pnumber → Pname, Plocation which can be written as 2 FDs

FD21: Pnumber → Pname

FD22: Pnumber → Plocation

In both FD21 and FD22, the RHS of FD is NPA and the LHS of FD is part of key {SSN, **Pnumber**}

WHAT is wrong with EMP_PROJ?

- Redundancy .... Since a project can have more than one employee, the name and location of the project is redundant in all the records for those employees working in a project.
- INSERT ANOMALY:
  1. We cannot add a new project unless that project has at least one employee working in it.
  2. We cannot add a new employee unless that employee is working in at least one project.
- DELETE ANOMALY:
  1. If a project happens to have only one employee working for it and we delete this employee then we accidentally will delete the information about the project.
  2. If an employee works in only one project and we delete the project, then we will accidentally delete that employee from the database!
- UPDATE ANOMALY:
  1. If an employee works in more than one project and this employee changes name, then we will need to update this employee's name in all tuples for consistency.
  2. If a project has more than one employee, then if we change the name or location of this project, then we need to update all the tuples for this project for consistency.

How to Convert EMP_PROJ to 2NF?

EMP_PROJ (SSN, Pnumber, Hours, Ename, Pname, Plocation)

FD1: SSN → Ename — *An employee has a name and SSN uniquely identifies an employee.*
FD2: Pnumber → Pname, Plocation — *A project has a name and location and a unique number.*
FD3: SSN, Pnumber → Hours — *An employee works in many projects and a project has many employees. We need to track the number of hours worked by an employee in a project.*

1. Remove the partially dependent attribute(s) from the relation by placing them in a new relation along with a copy of their determinant (ie. LHS)

two attributes always in 3nf.

Using FD1: Ename is a partially dependent attribute

Create a new relation EMPLOYEE (SSN, Ename) SSN → Ename

NOTE: EMPLOYEE is 1NF, 2NF and 3NF!

===→ EMP_PROJ (SSN, Pnumber, Hours, Pname, Plocation)

Using FD2: Pname, Plocation are partially dependent attributes

Create a new relation PROJECT (Pnumber, Pname, Plocation) Pnumber → Pname, Plocation

NOTE: PROJECT is 1NF, 2NF and 3NF!

===→ EMP_PROJ (SSN, Pnumber, Hours) SSN, Pnumber → Hours

NOTE: EMP_PROJ is 1NF, 2NF, and 3NF!

Thus, **EMP_PROJ** (SSN, Pnumber, Hours, Ename, Pname, Plocation) has been normalized into three smaller 3NF relations!

EMPLOYEE (SSN, Ename) SSN → Ename
PROJECT (Pnumber, Pname, Plocation) Pnumber → Pname, Plocation
EMP_PROJ (SSN, Pnumber, Hours) SSN, Pnumber → Hours

Normal forms are necessary to preserve consistency after operations (Delete (D), Insert (I), and Update (U)).

Third Normal Form -- 3NF

Formally, a relation is in 3NF if it is in 2NF and has no transitive FDs.

An FD $X \rightarrow Y$ in R is a **transitive dependency**

if there is a set of attributes Z that is neither a candidate key (CK) nor a subset of any key of R and
both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

Informally

a relation is in 3NF if it is 2NF and no nonprime attribute of R is transitively dependent on the primary key.

Ie. There is no FD of the form NPA → NPA where NPA = nonprime attribute

EXAMPLE: *EMP_DEPT* (Ename, SSN, Bdate, Address, Dnumber, Dname, DMgr_SSN)

FD1: SSN → Ename, Bdate, Address	*An employee has a name, bdate and address and ssn uniquely identifies an employee.*
FD2: SSN → Dnumber	*An employee works for one department.*
FD3: Dnumber → Dname, DMgr_SSN	*A department has a name and manager and a unique dnumber.*

EMP_DEPT is in 2NF since it is in 1NF (only atomic and single-valued attributes) and

its PK is SSN which has only one attribute and therefore cannot violate full-functional dependency!

What is wrong if EMP_DEPT is left in 2NF?

Refer to EMP_DEPT Table in Figure 15.4, page 508

- Too much redundancy: Since a department can have many employees, the department info (dname, dmgr_ssn) is repeated for every employee that works in a department.
- DELETE Anomaly: If we delete employee James Borg, then we accidentally lose information on the Headquarters Department.

- UPDATE Anomaly: If we change the name of Administration Department and there are x employees working for Administration Department, we must change the name of the Administration department in these x tuples. Failure to do so will result in inconsistent data.
- INSERT ANOMALY: We cannot add a new department unless that department has at least one employee!

EXAMPLE: *EMP_DEPT* (Ename, SSN, Bdate, Address, Dnumber, Dname, DMgr_SSN)

FD1: SSN → Ename, Bdate, Address	*An employee has a name, bdate and address and ssn uniquely identifies an employee.*
FD2: SSN → Dnumber	*An employee works for one department.*
FD3: Dnumber → Dname, DMgr_SSN	*A department has a name and manager and a unique dnumber.*
FD31: Dnumber → Dname	
FD32: Dnumber → DMgr_SSN	

EMP_DEPT is in 2NF but is not in 3NF!

PK: SSN
PA: SSN
NPA: Ename, Bdate, Address, Dnumber, Dname, DMgr_SSN

FD31 is a transitive dependency since DMgr_SSN is transitively dependent on SSN via Dnumber (FD2 and FD32)
Similary, FD32 is a transitive dependency since Dname is transitively dependent on SSN via Dnumber (FD2 and FD31)

How to convert to 3NF?

1. Remove the transitive dependent attribute from the original relation and creating a new relation with the transitive dependent attribute together with its determinant.

   DEPARTMENT (Dnumber, Dname, DMgr_SSN) Dnumber → Dname, DMgr_SSN

   DEPARTMENT is 1NF, 2NF, and 3NF!

   EMP_DEPT (SSN, Ename, Bdate, Address, Dnumber)

   SSN → Dnumber
   SSN → Ename, Bdate, Address

   EMP_DEPT is in 1NF, 2NF, 3NF!

RATIONALE for BCNF:

Definition of 3NF assumes that the relation has only one relation key.

Problems arise with the definition when the relation has more than one key.

EXAMPLE: GRADES (Student_Id, Phone_No, Course, Grade) //Stores Student's Grades in Courses as well as phone numbers.

Each student has one unique non-shared Phone_No. Student_Id → Phone_No and
Phone_No → Student_Id

Each student takes any number of courses and gets a grade for each course.
Student_Id, Course → Grade

Since Phone_No is unique .... Phone_No, Course → Grade

So GRADES has two keys: (Student_Id, Course) and (Phone_No, Course)

GRADES is in 3NF! Why?

Although GRADES is in 3NF it has undesirable characteristics:

- Phone number of each student is stored more than once → unnecessary redundancy.
- A student's phone cannot be stored until the student takes at least one course → insert anomaly
- A student's grade cannot be entered until the student's phone is known → insert anomaly
- If a student drops all courses, all tuples that include this student are deleted which inadvertently deletes student's phone number → delete anomaly.
- If the student's phone number changes, more than one tuple must be changed. →update anomaly

Problems arise because there are dependencies between key attributes Student_Id and Phone_No.
To remove these problems, Boyce proposed a normal form know as BCNF.

Normalize GRADES:

- Student_Id → Phone_No
- Phone_No → Student_Id

Student (Student_Id, Phone_No) Student_Id → Phone_No .... BCNF
Phone_No → Student_Id

Results (Student_Id, Course, Grade) Student_Id, Course → Grade .... BCNF

How to Get the Closure of X -- Simplified Version

The Closure finds the attributes that are functionally dependent on X, given a set of attributes and FDs.

1. Let $X_0 = X$; $N = 0$;

2. If there is a dependency $A \rightarrow B$ in the original list of FDs whose left hand side (A) is contained in X_N but whose right hand side (B) is not in X_N
   then add B to X_N to make X_{N+1}; i.e. $X_{N+1} = X_N \cup B$
   otherwise, terminate.

3. Increment N and go to step 2.

Example:

Given: R={A,B,C,D,E,F,G,H,I,J} and
FDs: 1. AB → C 2. A→D,E 3. B → F 4. F→ G,H 5. D→I,J

Find: Closure of B (B^+) Using the above algorithm, X is now B

1. Let $X_0 = B$; $N = 0$
2. Look in FDs which has B on the left hand side (LHS) and whose right hand side (RHS) is not in X_0

..... FD3 has B on LHS but F on the RHS
$X_1 = X_0 + F = \{B, F\}$ $N = 1$

3. Look in FDs which has F on (LHS) and whose (RHS) is not in X_1

..... FD4 has F on LHS but G,H on the RHS are not in X_1
$X_2 = X_1 + G + H = \{B, F, G, H\}$ $N = 3$

4. Look in FDs which has G on (LHS) and whose (RHS) is not in X_2
   No FD exists
5. Look in FDs which has H on (LHS) and whose (RHS) is not in X_2
   No FD exists

Therefore no other attribute can be added to $X_2 = \{B,F,G,H\}$

How can we use the Closure Property to indicate if an attribute is a key of a relation?

Aggregate Functions and Grouping in Relational Algebra and SQL

Relational Algebra

$$\langle \text{grouping attributes} \rangle \; \mathcal{F} \; \langle \text{function list} \rangle \; (R)$$

Where <function list> = (<function> <attribute>) pairs
<function> = SUM | AVERAGE | MAXIMUM | MINIMUM | COUNT

<grouping attributes> = list of attributes of R
<attribute> = attribute of R

OUTPUT/RESULT attributes: grouping attributes, function list

EXAMPLES in Relational Algebra:

1. For each department, retrieve the number of employees in the department, and their average salary.

$$\text{DNO } \mathcal{F} \text{ COUNT SSN, AVERAGE SALARY } (\text{EMPLOYEE})$$

2. For each department, retrieve the number of employees in the department, and their average salary and rename the resulting attributes.

$$\rho_{R(\text{DNO, No\_of\_Employees, Average\_SAL})} (\text{DNO } \mathcal{F} \text{ COUNT SSN, AVERAGE SALARY } (\text{EMPLOYEE}))$$

Equivalent Examples in SQL:

1. For each department, retrieve the number of employees in the department, and their average salary.

```
SELECT DNO, COUNT(SSN), AVG(SALARY)
FROM EMPLOYEE
GROUP BY DNO;
```

2. For each department, retrieve the number of employees in the department, and their average salary and rename the resulting attributes.

$$\rho_{R(\text{DNO, No\_of\_Employees, Average\_SAL})} (\text{DNO } \mathcal{F} \text{ COUNT SSN, AVERAGE SALARY } (\text{EMPLOYEE}))$$

```
SELECT DNO, COUNT(SSN)  No_of_Employees, AVG(SALARY)  Average_SAL
FROM EMPLOYEE
GROUP BY DNO;
```

RENAME in Relational Algebra and SQL

RENAME in RELATIONAL ALGEBRA

- The rename operator enables one to rename the relation name or the attribute names or both.
- It is a unary operator – operating on one relation only.
- SYNTAX: Given relation R of degree n (degree = number of attributes in R)

$P_{S(B1, B2, ..., Bn)}(R)$ → *renames relation R with S and attributes of R with* $_{B1, B2, ..., Bn}$

Or $P_{S}(R)$ → *renames relation R with S*

Or $P_{(B1, B2, ..., Bn)}(R)$ → *renames attributes of R with* ***B1, B2, ..., Bn***

RENAME in SQL R (A1,A2, ..., An)

```
SELECT a1 B1, a2 AS B2, ...., an Bn
FROM R S
[WHERE <condition>];
```

Steps in Determining all the Candidate Keys of a Relation/Table

GIVEN: Relation R, its attributes and its set of FDs, F.
OUTPUT: Candidate keys of R

PROCEDURE:

1. Rewrite the FDs in canonical form; i.e. an FD should have only one attribute on the right hand side (RHS).
2. Remove the extraneous attribute from any FD whose left hand side (LHS) has more than one attribute! .... Apply the Closure Property.
3. Remove the redundant FDs from the given set of FDs, F. .... Apply the Membership Algorithm.
4. Determine the likely combinations of attributes that may comprise the key
   Using the set of non-redundant FDs from Step 3 above

   a. Eliminate attributes that will **never participate in any key.**
      If an attribute is **never on the LHS of an FD and is on the RHS of an FD** then it is **not in any key**.
   b. Find attributes that will **always participate in all keys**
      If an attribute is **never on the RHS of an FD** then it must be in **every key.**
      This includes attributes in R which are not listed in **any FD**, i.e. attribute not found in any functional dependency in F.
   c. If an attribute Z is found on both sides of the FDs then we cannot use 4a or 4b to indicate if Z is definitely part of all keys or definitely not part of any key.
      List all such attributes (Z_1, Z_2, ..., Z_k) and its subsets

5. The possible candidate keys are
   (1) set of attributes from 4b
   (2) set of attributes from 4b U subset 1 of Z_1, Z_2, ..., Z_k
   (3) set of attributes from 4b U subset 2 of Z_1, Z_2, ..., Z_k
   (4) set of attributes from 4b U subset 3 of Z_1, Z_2, ..., Z_k
   (5) .....
   (6) set of attributes from 4b U subset 2^{kth} of Z_1, Z_2, ..., Z_k

6. Compute the X-closure of each of the sets in step 5 above.
   Start with the set that has the smallest number of attributes.
   If the closure gives the set of all attributes of R then the set S is a key.
   You do not need to find the closure of any set that contains S.
   However you still need to do the closure of other sets that have the same number of attributes as S. These other sets are candidate keys if their closure is the set of all attributes of R.

JOINS

Joins can be used to connect any number of tables.
The number of joins you will need in your WHERE clause is total_number_of_tables – 1

Types of join conditions based on the operator.

- Equijoins - You use the equality operator (=) in the join.
- Non-equijoins - You use an operator other than equals in the join, such as

1. not-equal (<>),
2. less than (<),
3. greater than (>),
4. less than or equal to (<=),
5. greater than or equal to (>=),
6. LIKE,
7. IN, and
8. BETWEEN.

Three different types of joins:

1. Inner joins - Return a row only when the columns in the join contain values that satisfy the join condition. This means that if a row has a null value in one of the columns in the join condition, that row isn't returned.
2. Outer joins - Can return a row even when one of the columns in the join condition contains a null value.
3. Self joins - Return rows joined on the same table.

Decision Matrix for Writing Joins

Example: If you want to display the name and department name of all the employees who are in the same department as Narayan.

Columns to Display	Originating Table	Condition
lname, fname,	Employee	Lname = 'Narayan'
Dname	Department	Employee.dno = Department.dnumber

SQL statement can be easily formulated by looking at the decision matrix. The first column gives the column list in the SELECT clause, the second column gives the tables for the FROM clause, and the third column gives the condition for the WHERE clause.

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

Table Aliases

- Qualifying column names with lengthy table names can be time consuming, so use table aliases instead of table names.
- Table aliases can be up to 30 characters in length, but shorter is better.
- If a table alias is used for a table name in the FROM clause, use the table alias instead of the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- Table alias is valid only for the current SELECT statement.

BETWEEN lovalue AND hivalue

- Specify low value first and the high value last.
- a BETWEEN lo AND hi is equivalent to (a >= lo) AND (a <= hi)
- has no performance benefits, just logical simplicity.

IN operator [a IN (....)]

- a IN (v1, v2, ..., vn) is equivalent to (a = v1 OR a=v2 ... OR a=vn).
- has no performance benefits, just logical simplicity.

Inner Join

- Syntax
    - SELECT table1.column, table2.column
      FROM table1, table2
      WHERE table1.column = table2.column

Outer Join

- Use an outer join to also see rows that do not meet the join condition.
- Outer join operator is the plus sign (+) and is placed on the 'side' of the join that is deficient in information.
- The operator has the effect of creating one or more null rows, to which one or more rows from the nondeficient table can be joined.
- SYNTAX

a. SELECT table1.column, table2.column
FROM table1 [LEFT|RIGHT|FULL OUTER] JOIN table2
ON (table1.column_name = table2.column_name)

b. SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column (+) = table2.column;

c. SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column = table2.column (+);

table1.column (+) is the outer join symbol, that can be placed on either side of the WHERE clause condition, but not on both sides.

Place the + sign following the name of the column in the table without the matching rows

- A condition involving an outer join cannot use the IN operator or be linked to anther condition by the OR operator.

OTHER Ways to Express Inner Join

- Join two tables based on the same column name

```
SELECT table1.column, table2.column
FROM table1  NATURAL JOIN table2
```

- Do equijoin based on the column name

```
SELECT table1.column, table2.column
FROM table1  JOIN table2  USING (column_name)
```

  - If several columns have the same names but the data types do not match, use the USING clause instead of NATURAL JOIN to specify the columns that should be used for an equijoin.
  - Use the USING clause to match only one column when more than one column matches.
  - Do not use a table name or alias in the referenced columns

- Do equijoin based on the condition in the ON clause

```
SELECT table1.column, table2.column
FROM table1  JOIN table2  ON  (table1.column_name = table2.column_name)
```

EXAMPLE:

LEFT OUTER JOIN

```
SELECT e.lname, e.dno, d.dname
FROM employee e    LEFT OUTER JOIN  department d   ON (e.dno = d.dnumber);
```

This retrieves all rows in the EMPLOYEE table, which is the left table even if there is no match in the Department table.

```
SELECT e.lname, e.dno, d.dname
 FROM employee e, department d
 WHERE d.dumber (+) = e.dno;
```

RIGHT OUTER JOIN

```
SELECT e.lname, e.dno, d.dname
FROM employee e    RIGHT OUTER JOIN  department d   ON (e.dno = d.dnumber);
```

This retrieves all rows in the DEPARTMENT table, which is the right table even if there is no match in the Employee table.

```
SELECT e.lname, e.dno, d.dname
 FROM employee e, department d
 WHERE d.dumber = e.dno (+);
```

FULL OUTER JOIN

```
SELECT e.lname, e.dno, d.dname
FROM employee e    FULL OUTER JOIN  department d   ON (e.dno = d.dnumber);
```

This retrieves all rows in the EMPLOYEE table, even if there is no match in the Department table. It also retrieves all rows in the DEPARTMENT table, even if there is no match in the EMPLOYEE table.

```
SELECT e.lname, e.dno, d.dname
 FROM employee e, department d
 WHERE d.dumber = e.dno (+)
  UNION
 SELECT e.lname, e.dno, d.dname
 FROM employee e, department d
 WHERE d.dumber (+) = e.dno ;
```

SUBQUERY

Syntax: SELECT select_list
FROM table
WHERE expr operator
(SELECT select_list
FROM table);

NOTES:

1. Subquery (inner query) is also called nested SELECT, sub-SELECT or inner SELECT.
2. Subquery (inner query) executes once before the main query.
3. Result of the subquery is used by the main query (outerquery).
4. Can place the subquery in
    a. WHERE clause
    b. HAVING clause
    c. FROM clause
    d. CREATE VIEW
    e. CREATE TABLE
    f. UPDATE statement
    g. INTO clause of an INSERT statement
    h. SET clause of an UPDATE statement
5. Operator includes a comparison condition
    a. Single-row operators (=,>, >=, <, <=,, <>)
    b. Multiple-row operators (IN, ANY, ALL)

Operator	Meaning
IN	Equal to any member in the list
ANY	Compare value to each value returned by the subquery < ANY means less than the maximum > ANY means more than the minimum = ANY is equivalent to IN
ALL	Compare value to **every** value returned by the subquery < ALL means less than the minimum > ALL means more than the maximum

When using ANY use DISTINCT keyword to prevent rows from being selected several times.

The NOT operator cannot be used with IN, ANY and ALL operators.

6. Enclose subqueries in parenthesis.
7. Place subqueries on the right side of the comparison operator.
8. Use single-row operators with single-row subqueries and use multiple-row operators with multiple-row subqueries.
    a. Single-row subqueries return only one row from the inner SELECT statement.
    b. Multiple-row subqueries return more than one row from the inner SELECT statement.
9. There are multiple-column queries – queries that return more than one column from the inner SELECT statement.
10. The outer and inner queries can get data from different tables.

11. A subquery can execute multiple times in correlated subqueries.
12. There is no limit on the number of subqueries; the limit is related to the buffer size that the query uses.

6. SELECT ssn, lname, gender, salary
   FROM employee
   WHERE salary < ANY
                (SELECT salary
                 FROM employee
                 WHERE gender = 'F')
   AND gender <> 'F';

7. SELECT ssn, lname, gender, salary
   FROM employee
   WHERE salary < ALL
                (SELECT salary
                 FROM employee
                 WHERE gender = 'F')
   AND gender <> 'F';

UNION, INTERSECT, MINUS

EXAMPLE:

```
 /* Make a list of all project numbers for projects that involve an
    employee whose last name is 'Smith', either as a worker or as a
    manager of the department that controls that project.
 */

(SELECT DISTINCT Pnumber
 FROM jmendoza.PROJECT, jmendoza.DEPARTMENT, jmendoza.EMPLOYEE
 WHERE    Dnum=Dnumber    AND
          MgrSsn=Ssn      AND
          Lname='Smith')

UNION

(SELECT DISTINCT Pnumber
 FROM jmendoza.PROJECT, jmendoza.WORKS_ON, jmendoza.EMPLOYEE
 WHERE    Pno=Pnumber     AND
          Essn=Ssn        AND
          Lname='Smith');

 /* The first SELECT retrieves the projects that involve 'Smith'
    as manager of the dept that controls project, and the
    second retrieves the projects that involve a 'Smith' as a
    worker on the project.
 */
```

ANOTHER SOLUTION Q4A using the IN and OR operators.

```
SELECT DISTINCT  pnumber
FROM jmendoza.project
WHERE pnumber  IN  (SELECT pnumber
                    FROM jmendoza.project, jmendoza.department, jmendoza.employee
                    WHERE dnum=dnumber AND
                          Mgrsnn = ssn    AND
                          lname = 'Smith'
                    )
   OR   pnumber  IN  (SELECT pno
                    FROM jmendoza.works_on, jmendoza.employee
                    WHERE essn = ssn          AND
                          lname = 'Smith'
                    );
```

SET OPERATORS

The set operators combine the results of two or more component queries into one result. Queries containing SET operators are called ***compound queries***.

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows that are selected by the first SELECT statement and not selected by the second SELECT statement.

All SET operators have equal precedence. If a SQL statement has multiple SET operators, the statement is evaluated from left (top) to right (bottom) if no parenthesis explicitly specify another order.

You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with the other SET operators.

The UNION Operator

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- The IN operator has higher precedence than the UNION operator.
- By default, the output is sorted in ascending order of the first column of the SELECT clause.

Example: Make a list of all project numbers for projects that involve an employee with last name 'Smith' either as worker or as a manager of the department that controls the project.

```
(SELECT   DISTINCT Pnumber                                          // Smith as a manager of dept that controls project
 FROM     jmendoza.PROJECT, jmendoza.DEPARTMENT, jmendoza.EMPLOYEE
 WHERE    Dnum=Dnumber AND MgrSSN = SSN AND UPPER( Lname) = 'SMITH')

 UNION

(SELECT   DISTINCT Pnumber                                          //Smith as a worker in a project
 FROM     jmendoza.PROJECT, jmendoza.WORKS_ON, jmendoza.EMPLOYEE
 WHERE    Pno=Pnumber AND ESSN = SSN AND UPPER( Lname) = 'SMITH');
```

The UNION ALL Operator

- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- The DISTINCT keyword cannot be used.

The INTERSECT Operator

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- INTERSECT does not ignore NULL values.

The MINUS Operator

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- All of the columns in the WHERE clause must be in the select clause for the MINUS operator to work.

SET Operator Guidelines

- The expressions in the SELECT lists must match in number and data type.
- Parentheses can be used to alter the sequence of execution.
- The ORDER BY clause
    - Can appear only at the very end of the statement
    - Will accept the column name, aliases from the first SELECT statement, or the positional notation
    - The column name or alias, if used in the ORDER BY clause, must be from the first SELECT list.
- SET operators can be used in subqueries.

AGGREGATING DATA USING GROUP FUNCTIONS

Group functions operate on sets of rows to give one result per group. These sets may be the whole table or the table split into groups.

Group Functions

Function	Description
AVG([DISTINCT | ALL] *n*)	Average value of *n*, ignoring null values
COUNT({ * | DISTINCT | ALL] *expr*})	Number of rows, where *expr* evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)
MAX([DISTINCT | ALL] *expr*)	Maximum value of *expr*, ignoring null values
MIN([DISTINCT | ALL] *expr*)	Minimum value of *expr*, ignoring null values
STDDEV([DISTINCT | ALL] *x*)	Standard deviation of *n*, ignoring null values
SUM ([DISTINCT | ALL] *n*)	Sum values of *n*, ignoring null values
VARIANCE([DISTINCT | ALL] *x*)	Variance of *n*, ignoring null values

GUIDELINES

- DISTINCT makes the function consider only nondupliate values; ALL makes it consider every value including duplicates. The default is ALL and therefore does not need to be specified.
- The data types for the functions with an *expr* argument may be CHAR, VARCHAR2, NUMBER or DATE.
- All group functions ignore null values.
- The ORACLE server implicitly sorts the result set in ascending order when using a GROUP BY clause. To override this default ordering, DESC can be used in an ORDER BY clause.
- You can use MIN and MAX for any data type.
- COUNT(*) returns the number of rows in a table.
- COUNT(expr) returns the number of rows with non-null values for the expr.
- COUNT(DISTINCT *expr*) returns the number of unique, non-null values in the column identified by *expr*.

EXAMPLES

1. SELECT AVG(SALARY), MAX(SALARY), MIN(SALARY), SUM(SALARY)
   FROM jmendoza.employee;

2. SELECT MIN(bdate), MAX(bdate)
   FROM jmendoza.employee;

3. SELECT COUNT (*)
   FROM jmendoza.employee
   WHERE dno = 4;

4. SELECT COUNT(dno)
   FROM jmendoza.employee;

5. SELECT COUNT(distinct dno)
   FROM jmendoza.employee;

GROUP FUNCTIONS and NULL VALUES

Group functions ignore null values in the column. For example if we added a COMMISSION in the EMPLOYEE table and only four of the employees have commissions then

```
SELECT AVG(COMMISSION)
FROM jmendoza.EMPLOYEE;
```

This will calculate the total commission paid to all employees divided by the number of employees receiving a commission (four).

USING NVL FUNCTION with GROUP Functions

NVL forces group functions to include null values.

```
SELECT AVG(NVL(COMMISSION, 0))
FROM jmendoza.employee;
```

The average is calculated based on *all* rows in the table, regardless of whether null values are stored in COMMISSION column. The average is calculated as the total commission paid to all employees divided by the total number of employees in the company (let's say 20).

NVL FUNCTION

Converts a null to an actual value.

- Data types that can be used are date, character, and number.
- Data types must match:
  - NVL (COMMISSION, 0)
  - NVL (dbate '01-JAN-97')
  - NVL(superssn, 'No Boss ') /superssn is char(9)

SYNTAX

NVL (*expr1*, *expr2*)

expr1 - source value or expression that may contain a null
expr2 - target value for converting the null

CREATING GROUPS OF DATA

At times, it is needed to divide the table of information into smaller groups. This can be done by GROUP BY clause.

SYNTAX

```
SELECT      column, group_function(column)
FROM        table
[WHERE      condition]
[GROUP BY   group_by_expression]
[ORDER BY   column];
```

This divides rows in a table into smaller groups by using the GROUP BY clause.

GUIDELINES:

- If you include a group function in a SELECT clause, you cannot select individual results as well, unless the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the columns in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.

EXAMPLES:

1. SELECT **dno**, AVG(salary)
   FROM jmendoza.employee
   GROUP BY **dno**;

   *This calculates the average salary for **each** department.*

2. SELECT AVG(salary)
   FROM jmendoza.employee
   GROUP BY dno;

   This can be done but results are not meaningful.

3. SELECT dno, AVG(salary)
   FROM jmendoza.employee
   GROUP BY dno
   ORDER BY AVG(salary)

4. SELECT dno, AVG(salary) "Average Salary"
   FROM jmendoza.employee
   GROUP BY dno
   ORDER BY "Average Salary"

GROUPS WITHIN GROUPS

Let's assume that employee table has job_id column

```
SELECT dno, job_id, sum(salary)
FROM jmendoza.employee
GROUP BY dno, job_id;
```

The select statement computes the sum of the salaries for job-ids within each dno group.

RESTRICTING GROUP RESULTS – HAVING CLAUSE

Use the HAVING clause to restrict groups:

1. Rows are grouped.
2. Group function is applied.
3. Groups matching the HAVING clause are displayed.

SYNTAX:

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY   group_by expression]
[HAVING     group_condition]
[ORDER BY   column];
```

EXAMPLE:

Query: Find the maximum salary of each department, but show only the departments that have a maximum salary of more than $10,000.

```
SELECT dno, MAX(salary)
FROM jmendoza.employee
GROUP BY dno
HAVING MAX(salary) > 10000;
```

NESTING GROUP FUNCTIONS

```
SELECT  MAX(AVG(SALARY)
FROM jmendoza.employee
GROUP BY dno;
```

NOTE: Group functions can be nested to a depth of two.

Figure 3.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

HORIZONTAL PROPAGATION == limits the breath of propagation of privileges

Limiting horizontal propagation to i **means that an account A given the grant option can grant the privilege to at most i other accounts.**

VERTICAL PROPAGATION == limits the depth of propagation of privileges

Vertical Propagation = 0 → granting the privilege with no grant option.

Account A grants Account B with vertical propagation j → B can grant the privilege to other accounts only with a vertical propagation < j.

ORACLE 11g VIEWS

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations thru a view	Yes	Not always

Creating a View

SYNTAX:

```
CREATE [OR REPLACE] [FORCE|NOFORCE} VIEW viewname
    [(alias[, alias] ...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

NOTES:

1. Subquery can contain complex SELECT syntax. It is a complete SELECT statement. You can use aliases for the columns in the select list.
2. Column aliases in the CREATE VIEW clause are listed in the same order as the columns in the subquery.
3. OR REPLACE recreates the view if it already exists.
4. FORCE creates the view regardless of whether or not the base tables exist
5. NOFORCE creates the view only if the base tables exist (this is the default)
6. viewname is the name of the view
7. alias specifies names for the expressions selecte4d by the view's query
   The number of aliases must match the number of expressions selected by the view.
8. WITH CHECK OPTION specifies that only rows accessible to the view can be inserted or updated.
   constraint is the name assigned to the CHECK OPTION constraint
9. WITH READ ONLY ensures that no DML operations can be performed on this view

EXAMPLES of SIMPLE VIEW:

```
CREATE VIEW empvu05
  AS SELECT  ssn, lname, salary
     FROM   employee
    WHERE dno = 5;

CREATE VIEW empvu05 (IdNum, EmpName, MonthlySalary)
  AS SELECT  ssn, lname, salary
     FROM   employee
    WHERE dno = 5;

CREATE VIEW empvu05 (Id_Number,  Name,  Annual_Salary)
  AS SELECT  ssn, lname, salary * 12
     FROM employee
    WHERE dno= 5;
```

EXAMPLES OF COMPLEX VIEW

```
CREATE VIEW Dept_Summary_Vu (name, minsal, maxsal, avgsal)
AS  SELECT  d.dname, MIN(e.salary), MAX(e.salary), AVG(e.salary)
    FROM   employee e, department d
    WHERE   e.dno = d.dnumber
    GROUP BY d.dname;
```

RETRIEVING from a VIEW

SELECT *
FROM viewname;

Views in the Data Dictionary

USER_VIEWS -- shows the name of the view and the definition of the view. The text of the SELECT statement that constitutes the view is stored in a LONG column. To see more contents of a LONG column, use SET LONG n where n is the value of the number of characters of the LONG column that you want to see.

Data Access Using Views

When data is accessed using a view, the ORACLE server performs the following:

1. It retrieves the view definition from USER_VIEWS
2. It checks access privileges for the view base table.
3. It converts the view query into an equivalent operation on the underlying base tables or tables. Data is retrieved from , or an update is made to, the base tables.

RULES for PERFORMING DML Operations on a VIEW

1. DML operation can be performed on simple views.
2. A row cannot be removed if the view contains the following:
   - Group functions
   - A GROUP BY clause
   - DISTINCT keyword
3. Data in a view cannot be modified if it contains
   - Group functions
   - A GROUP BY clause
   - DISTINCT keyword
   - Columns defined by expressions
4. Data cannot be added through a view if the view includes
   - Group functions
   - A GROUP BY clause
   - DISTINCT keyword
   - Columns defined by expressions
   - NOT NULL columns in the base tables that are not selected by the view
     NOTE: you are adding values directly into the underlying table through the view.

USING the WITH CHECK OPTION Clause

- To ensure that DML operations performed on the view stay within the domain of the view by using the WITH CHECK OPTION clause.

EXAMPLE:

```
CREATE or REPLACE VIEW Emp04_Vu
AS SELECT *
    FROM employee
   WHERE dno = 4
   WITH CHECK OPTION CONSTRAINT Emp04_Vu_CK;
```

NOTE: Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

- It is possible to perform referential integrity checks through views. Constraints can be enforced at the database level. The view can be used to protect data integrity, but the use is very limited.
- The WITH CHECK OPTION clause specifies that INSERTS and UPDATES performed through the view cannot create rows which the view cannot select, and therefore it allows integrity constraints and data validation checks to be enforced on data being inserted or updated.
- If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, with the constraint name if that has been specified.
  ERROR is ORA-01402: view with CHECK OPTION where-clause violation

```
UPDATE Emp04_Vu
   SET dno=5
  WHERE ssn = '987654321';
```

No rows are updated because if the department number were to change to 5, the view would no longer be able to see that employee. Therefore, with the CHECK OPTION clause, the view can see only employees in department 4 and does not allow the department number for those employees to be changed through the view.

DENYING DML Operations

- To ensure that no DML operations can occur, add WITH READ ONLY option to the view definition.
- Any attempt to remove a row from the view with a read-only constraint results in an ORACLE server error
  ORA-01752: cannot delete from view without exactly one key-preserved table.
- Any attempt to insert or modify a row using the view with a read-only constraint results in an ORACLE server error
  ORA-01733: virtual column not allowed here.

REMOVING A VIEW

DROP VIEW viewname;

- Only the view definition is removed from the database.
- Dropping views has no effect on the tables on which the view was based.
- Views or other applications based on deleted views become invalid.
- Only the creator or a user with the DROP ANY VIEW privilege can remove a view.

Inline Views

- Is a subquery with an alias (or correlation name) that you can use within a SQL statement.
- A named subquery in the FROM clause of the main query is an example of an inline view.
- Is created by placing a subquery in the FROM clause and giving the subquery an alias.

EXAMPLE:

```
SELECT e.lname, e.salary, e.dno,  d.maxsal
FROM employee e, (SELECT dno, max(salary)   maxsal
                  FROM  employee
                  GROUP BY dno)  d
WHERE e.dno = d.dno
AND e.salary < d.maxsal;
```

Implementing Equi & Natural Two-Way Joins and their Cost

R |X|$_{A=B}$ S

1. **Nested-Loop Join or Nested-Block Join**

```
For each block of R
  For each block of S
    For each tuple in the R block
      For each tuple in the S block
        If the tuples satisfy the condition then add to join
      End
    End
  End
```

Minimum number of buffers needed: 3 one for outerloop R, one for the inner loop S and one for join result.
If n_B = number of buffers > 3 then we read into as many buffers as many blocks as possible from the file in the outer loop and save one buffer for only one block from the file in the inner loop, plus the space for writing the result.
We should read n_B -2 blocks of R into the buffer at a time and only 1 block of S, saving one buffer for the result. The total number of blocks of R accessed is still b_R but the total number of S blocks that need to be accessed is approximately b_S * ceiling(b_R/(n_B -2)). NOTE: ceiling(b_R/(n_B -2)) = no of times (n_B -2) blocks are loaded in MM

The join algorithm uses a buffer to hold the joined records of the result file. Once the buffer is filled, it is written to disk and its contents are appended to the result file, and then refilled with join result records.

Therefore the total cost of access = b_R + (b_S * ceiling(b_R/(n_B -2))) + cost of writing resulting file to disk

Use for outer block, the file with fewer blocks!

What's cost of writing the result file?
Join Selectivity (js) = ratio of the number of tuples of the resulting join file to size of the Cartesian Product file
= | R |X|$_C$ S| / (R X S)
= | R |X|$_C$ S| / (|R| * |S|)

If no join condition: js =1 and the join = Cartesian Product
If no tuples satisfy the join condition: js = 0.
General: 0 <= js <= 1.
Special cases if c is R.A = S.B
1. If A is a key of R, then | R |X|$_C$ S | <= |S| → js <= |S|/ (|R| * |S|) → js <= 1 / |R|
   [each record of S will be joined with at most one record of R since A is a key of R].
2. If B is a key of S, then | R |X|$_C$ S | <= |R| → js <= |R|/ (|R| * |S|) → js <= 1 / |S|

take bigger if time for both

$$|R|X|_C S| = js * (|R| * |S|)$$

$125 \times 10000 = \frac{1}{125} \times 1000 \times 75$

Cost of writing the result (no of blocks to be written back to disk): (js * |R| * |S|) / bfr_{RS}

Cost of Nested Join Loop = b_R + (b_S * ceiling(b_R/(n_B -2))) + ((js * |R| * |S|) / bfr_{RS})

outer should have less blocks.

ICE: Feb 18 – Implementation & Cost of JOIN

NAME 1:_____ **NAME 2:**_____

GIVEN: 2 relations: Department (D) and Employee (E)

where $r_D = |D| = 125$ records; $b_D = 13$ blocks;
primary index on Dnumber: $x_{Dnumber} = 1$;
secondary index on Mgr_ssn: $x_{Mgr\_ssn}=2$; $s_{Mgr\_ssn}= 1$

$r_E = |E| = 10000$ records; $b_E = 2000$ blocks; $bfr_E = 5$ records/block;
secondary index on key attribute SSN: $X_{ssn}= 4$; $s_{ssn} = 1$; $sl_{ssn}= .0001$

$bfr_{ED} = 4$ records/block

$n_B = 3$ memory buffers

Calculate the cost functions for different options of executing JOIN operation

OP7: DEPARTMENT $|X|_{Mgr\_ssn=ssn}$ EMPLOYEE

nb = 3

Implementation J1 (nested-loop join):

Since D has fewer blocks (13) than E (2000), we will use D in the outer loop

js = 1/125 since every Department has exactly one manager

$C_{J1} = b_D + (\text{ceiling}((b_D / (n_B - 2))) * b_E) + ((js * |D| * |E|)/bfr_{ED})$
= 13 blocks + (ceiling((13 blocks/(3-2) blocks)) * 2000 blocks) + (((1/125 records) * 125 records * 10000 records) / 4 records/block)

= 13 blocks + (13 * 2000 blocks) + (10000 records/ 4 records/block)

= 13 blocks + 26000 blocks + 2500 blocks = 28513 blocks

Implementation J2A (single-loop join: D as loop; secondary index on ssn for E)

$C_{J2A} = b_D + (|D| * (x_{SSN} + 1)) + (((1/125) * |D|*|E|)/bfr_{ED})$

= 13 + (125 * (4 + 1)) + (((1/125) * 125 * 10000)/4)

= 13 + (125 * 5) + (2500) = 13 + 625 + 2500 = 3138

Implementation J2B (single-loop join: E as loop; mgr_ssn as secondary key for D)

$C_{J2B} = b_E + (|E| * (x_{MGR\_SSN} + 1)) + (((1/125) * |D|*|E|)/bfr_{ED})$

= 2000 + (10000 * (2 + 1)) + 2500 = 2000 + (10000 * 3) + 2500 = 34,500

Min(34500, 3138) = 3138 ..... Choose C_{J2A}

Sort-Merge

Employee is already sorted on SSN.

Department is not sorted on MGR_SSN → cost to do sort = $b_D \log_2 b_D$

Cost of Sort-Merge Join = cost to sort Department + cost to merge

$$= b_D \log_2 b_D + b_D + b_E + ((js * |D| * |E|)/bfr_{DE})$$

$$=13 * \log_2 13 + 13 + 2000 = 48.1 + 2013 = 49 + 2013 = 2062$$

Total Cost = Cost of Sort-Merge Join + Cost to Write Result to Disk

$$= 2062 + (((1/125) * 125 * 10000)/4) = 2062 + 2500 = 4562$$

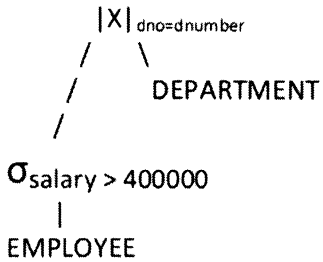
EXERCISE 19.22, p 725

Compare the cost of two different query plans for the query

$$\sigma_{salary > 400000} (Employee \; |X|_{dno=dnumber} \; DEPARTMENT)$$

Use the db statistics in Fig 19.8, page 720

QUERY PLAN A:



Use the salary index on EMPLOYEE for $\sigma_{salary > 400000}$.
Table a of Fig 19.8 shows there are 500 unique salary values (d) with a low value of 1 and high value of 500 (in units of $1000) ➔ low value is $1000 and high value is $500,000.

The selectivity (sl) for salary > 400000 ➔ salary > 400 can be estimated as
(500-400)/500 = 1/5 =0.20

Cost (in block accesses) of accessing the salary index (B+ tree)= blevel + (0.20 * Leaf_blocks)
= 1 + (0.20 * 50)
= 1 + 10 = 11

Since the index is non-unique, the employees can be stored on any of the data blocks.
So the number of records to be accessed = 0.20 * Num_Rows = 0.20 * 10000 = 2000

Since the 10000 rows are stored in 2000 blocks (Table b of Fig 19.8) ➔ 5 rows/block
So we can store these 2000 records in 400 blocks. [Temporary Table to store result of SELECT]

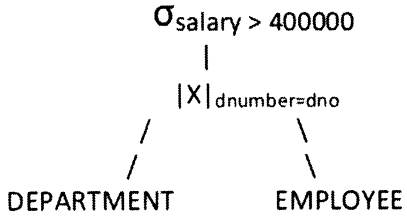
Cost of writing the Temporary Table to disk is 400 blocks.

Do NESTED LOOP Join between Temporary Table and Department Table. $N_B = 3$
Department is the outer loop; Temporary Table is the inner loop.
So the cost of the nested loop join = $b_{DEPARTMENT} + (b_{DEPARMENT} * b_{TEMPORARY}) = 5 + (5 * 400) = 2005$ accesses

Total Cost= Cost to Access Index + Cost to Access 2000 records + cost to write Temporary + Cost to Do Join

= 11 + 2000 + 400 + 2005 = 4416 blocks

QUERY PLAN B



Use nested loop join but use pipelining instead of creating a temporary table. That is, pass the joining rows to the select operator as they are selected. DEPARTMENT – outer loop; EMPLOYEE – inner loop and $n_B = 3$.

Cost of Nested Loop Join = 5 + (5 * 2000) = 10,005.

Pipeline the result to select operator to choose only those rows whose salary values > 400,000.

February 18, 2014 **InClass Exercise on Computing Cost of Select Relational Algebra Operator**
SAMPLE SOLUTION

GIVEN:

EMPLOYEE FILE
r = 10,000 records b=2000 disk blocks bfr = 5 records/block

ACCESS PATHS:

- Clustering Index on Salary: $x_{salary} = 3$; $s_{salary} = 20$;
  $sl_{salary} = 20/10000 = .002$ NOTE: $s = sl * r \rightarrow 20 = sl * 10000$

- Secondary Index on key attribute Ssn: $x_{ssn} = 4$;
  $s_{ssn} = 1$ since ssn is a primary key only one record satisfies each value
  $sl_{ssn} = 1/10000 = .0001$

- Secondary index on nonkey attribute dno: $x_{dno=2}$; $b_{I1dno} = 4$; $d_{dno} = 125$;
  $sl_{dno} = s/r = 80/ 10000 = .008$

  $s_{dno} = 10000/125 = 80$ NOTE: $s_{nonkey\ attribute} = r/d$

- Secondary index on gender, with $x_{gender} = 1$;
  $d_{gender} = 2$ (male, female)

  $s_{gender} = 10000/2 = 5000$ NOTE: $s_{nonkey\ attribute} = r/d$

COMPUTE the cost of each of the relational algebra operator select applied to the Employee relation in the COMPANY DB.

1. $\sigma_{ssn='333445555'}$ (EMPLOYEE)
   There are three ways to implement this: linear search, binary search, and use the secondary index on ssn.

   Using the secondary index on SSN, cost = $x_{ssn} + 1 = 4 + 1 = 5$ disk block accesses.

   Using linear search, cost = $b_{EMPLOYEE} = 2000$ blocks.

   Using binary search, cost = $b_{EMPLOYEE}/2 = 1000$ blocks. (NOTE: employee file has a primary index on ssn since ssn is the key of EMPLOYEE.
   CHOOSE min(5, 2000, 1000) = 5 → secondary index on SSN

2. $\sigma_{dno>4}$ (EMPLOYEE)
   There are two ways to do this: linear search and using the secondary index on dno.
   Use the secondary index on dno: cost = $x_{dno} + (b_{I1dno}/2) + (r_{EMPLOYEE}/2) = 2 + 4/2 + 10000/2 = 2 + 2 + 5000 = 5004$
   Linear search: cost = $b_{EMPLOYEE} = 2000$ blocks.
   CHOOSE min(2000, 5004) = 2000 → linear search

3. $\sigma_{dno=4}$ (EMPLOYEE)
   There are two ways to do this: linear search and using the secondary index on dno.
   equality condition on nonkey attribute → cost = $x_{dno} + 1 + s_{dno} = 2 + 1 + 80 = 83$

   linear search: cost = $b_{EMPLOYEE} = 2000$ blocks.
   CHOOSE min(2000, 83) = 83 → secondary index on dno

4. $\sigma_{dno=4\ AND\ salary>25000\ AND\ gender='F'}$ (EMPLOYEE)
   There are two main ways to do this: linear search and cost for each select condition in the conjunctive select.
   linear search: cost = $b_{EMPLOYEE} = 2000$ blocks

   Cost using dno=4.: from 3 above cost $_{dno=4}$ =83
   Cost using salary > 25000: cost $_{salary>25000}$ = x_{salary} +($b_{EMPLOYEE}/2$) = 3 + 2000/2 = 1003
   Cost using gender = 'F': cost $_{gender='F'}$ =x_{gender} + 1 + s_{gender} = 1 + 1 +5000 = 5002

   CHOOSE min(1003, 83,5002,2000) = 83 → secondary index on dno
   So we will choose to use the secondary index on dno and then check each selected record for the conditions salary > 25000 and gender = 'F'.

Example 19.8.6, page 719 – Cost Based Query Optimization

SELECT pnumber, dnum, lname, address, bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE dnum=dnumber AND mgr_ssn = ssn AND plocation='Stafford';

$\Pi_{pnumber,dnum,lname,\ address,\ bdate}\ (\sigma_{plocation='Stafford'}\ (PROJECT)\ |X|_{dnum=dnumber}\ DEPARTMENT\ |X|_{mgr\_ssn=ssn}\ EMPLOYEE)$

PROJECT will retrieve all Stafford Projects: Use Proj_Ploc non-unique index

$d_{PROJECT}$ = 200 $r_{PROJECT}$ = 2000 ➔ 10 project records per plocation value (s = r/d)

$bfr_{PROJECT}$ = floor(2000 rows/100 blocks) = 20 records/block

SELECT will retrieve 10 records ➔ will need 1 block to store result of select : TEMP1

$cost_{s6a} = x + 1 + s = 1 + 1 + 10 = 12$

DEPARTMENT has no index

$b_{DEPARTMENT}$ = 5 blocks $r_{DEPARTMENT}$ = 50 $bfr_{DEPARTMENT}$ = floor(50/5) = 10 records/block

$b_{PROJECT}$ = 100 blocks

smallest on outside.

JOIN (NLJ): TEMP2 ← TEMP1 $|X|_{dnum=dnumber}$ DEPARTMENT (outer: TEMP1 has 10 records; inner: Department : 50 recs

$r_{TEMP2} = r_{TEMP1}$ = 10 recs bfr_{TEMP2} = 5 recs/block TEMP2 will need 2 blocks to store result of join

JOIN (SLJ): TEMP3 ←TEMP2 $|X|_{mgr\_ssn=ssn}$ EMPLOYEE

Employee has a unique index EMP_SSN

$r_{TEMP3} = r_{TEMP2}$ = 10 recs

Single-loop Join: Loop over TEMP2 and then probe EMP_SSN index to do join
Look-up each of the 10 department managers for the matching ssn in EMPLOYEE using EMP_SSN index
Cost to search EMP_SSN index for one ssn is number of levels (root + leaf) + data block = 3
TOTAL COST to search 10 mgrs = 3 accesses per ssn * 10 mgr_ssns = 30 accesses

TOTAL COST of JOIN: cost for TEMP2 + cost for TEMP3 = 2 + 30 = 32 block accesses

+ 12 for temp1 + write back

$\pi_{pnumber,dnum,lname,\ address,\ bdate}$ (DEPARTMENT $|X|_{dnumber=dnum}$ ($\sigma_{plocation='Stafford'}$ (PROJECT)) $|X|_{mgr\_ssn=ssn}$ EMPLOYEE)

$\pi_{pnumber,dnum,lname,\ address,\ bdate}$ (DEPARTMENT $|X|_{mgr\_ssn=ssn}$ EMPLOYEE $|X|_{dnumber=dnum}$ ($\sigma_{plocation='Stafford'}$ (PROJECT)))

$\Pi_{pnumber,dnum,lname,\ address,\ bdate}$ (EMPLOYEE $|X|_{ssn=mgr\_ssn}$ DEPARTMENT $|X|_{dnumber=dnum}$ ($\sigma_{plocation='Stafford'}$ (PROJECT)))

Figure 19.8
Sample statistical information for relations in Q2. (a) Column information. (b) Table information. (c) Index information.

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.

CHAPTER 18 --- Indexing Structures for Files

1. Dense vs Sparse Index
   - Dense index = one index entry for every search key value or every record in the data file
   - Sparse (Non-dense) index = an index entry for some of the search values

2. Single-level Ordered Indexes

   A. Primary Index – ordered file on a key field → distinct value for each record
      - One index entry <K(i), P(i)> for each block in the data file
        - K(i) = key for the first record (*anchor record or block anchor*) in a block
        - P(i) = ptr to the block
      - No of index entries = no of disk blocks in the ordered data file
      - Sparse index → entry for each disk block vs for each record
      - Occupies smaller storage than data file
        - Fewer index entries vs data records
        - Smaller index entry size vs data record size
      - Cost (No of disk block accesses) to do binary search for a record with a search key
        = cost to binary search primary index + access block containing the record
        = $\log_2 b_i$ + 1 accesses
        Where b_i = number of blocks of the primary index

   See Figure 18.1, page 634

   B. Clustering Index
   File records are physically ordered on a nonkey field (clustering field) and data file is called a clustered file.
      - One index entry <K(i), P(i)> for each distinct value of clustering field
        - K(i) =clustering field value
        - P(i) = ptr to the first block in the data file that has a record with K(i)
      - Sparse Index
      -

   See Figure 18.2, page 637

3. Secondary Index
    - Data file can be ordered, unordered, or hashed
    - Can be created on a candidate key with a unique value or nonkey field with duplicate values
    - Index is ordered <indexing field on nonordering field of data file, block/record ptr)
    - Can have many secondary indices for a file -- one for each nonordering field
    - Secondary Index on a Key (Unique) field
        - One index entry for each record in the data file
          < K(i), P(i)>
            - K(i) =secondary key field value
            - P(i) = ptr to the block in the data file that has a record with K(i) or to the record itself
        - Entries are ordered by value of K(i).
        - Data file records are not physically ordered by values of secondary key field
        - Dense index
          See Figure 18.4, page 639; Figure 18.5, page 641

See TABLE 18.1 and TABLE 18.2, page 642 (SUMMARY)

4. Multi-Level Ordered Indexes = index to the indices
    - first or base level of multilevel index : Index File to the Data File
        - Ordered file with a distinct value for each K(i)
        - Needs ceiling (r_1/fo) blocks where r_1 = number of index entries for first level
    - Second level: primary index to the first level index
        - One entry for each block of the first level
        - r_2 = number of 2nd level entries = ceiling (r_1/fo)
        - needed only if the first level needs more than one disk block
    - Third level: primary index to the second level index
        - One entry for each block of the second level
        - r_3 = number of 3rd level entries = ceiling (r_2/fo)
        - needed only if the second level needs more than one disk block

          .....
    - t^{th} level : t = ceiling ($\log_{fo}(r_1)$) = number of levels of multi-level index
    - a single disk block is retrieved at each level
    - t disk blocks are accessed for a multi-level index where t = no of index levels.
      SEE Figure 18.6, page 645

bfr_i = fo (fan out) of the multi-level index

DISK:

B = Disk Block Size

DATA FILE:

R = record length in bytes (record size)
bfr = blocking factor = floor (B/R)
r = number of records in the file
b = number of blocks needed by file = ceiling (r/bfr)

INDEX FILE:

V = size of index key in bytes
P = size of pointer in bytes
R_i = Index entry size = V+ P bytes
bfr_i = floor(B/R_i)
r_i = number of index entries
b_i = number of index blocks = ceiling(r_i / bfr_i)

ICE 03 – Finding a Minimal Cover

GIVEN: R (A,B,C,D,E,F,G,H)

FDs:
FD1: A → B
FD2: ABCD → E
FD3: EF → G
FD4: EF → H
FD5: ACDF → EG

Step 2. Rewrite each FD in canonical form. FD5 is not in canonical form
FD1: A → B
FD2: ABCD → E
FD3: EF → G
FD4: EF → H
FD51: ACDF → E
FD52: ACDF → G

Step 3. Remove extraneous attributes from (FD2, FD3, FD4, FD5), FD52

For FD2: ABCD → E ..... since FD1 A→B, then B is extraneous in ABCD → E .... ACD → E (FD22)
For FD51: ... since FD22 ACD → E, then F is extraneous in ACDF → E .... ACD → E (FD511)

Thus far,
FD1: A → B
FD22: ACD → E
FD3: EF → G
FD4: EF → H
FD511: ACD → E
FD52: ACDF → G

FD22 and FD511 are the same so we can remove FD511.

FD1: A → B
FD22: ACD → E
FD3: EF → G
FD4: EF → H
FD52: ACDF → G

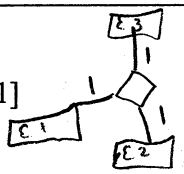
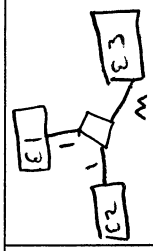
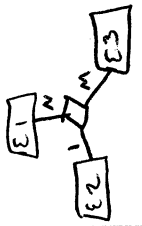
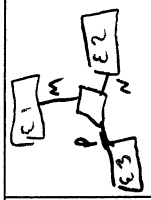
ACD → E and EF → G ===> ACDF → G .... ACDF is redundant!

Final set of FDs: (Minimal Cover)
FD1: A → B
FD22: ACD → E
FD3: EF → G
FD4: EF → H

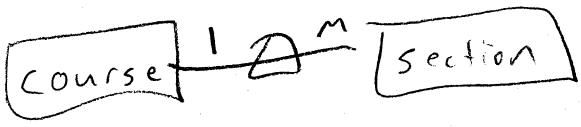
w to Determine Functional Dependencies from the Requirements and ER/EER Diagrams

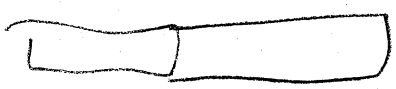
Need to analyze

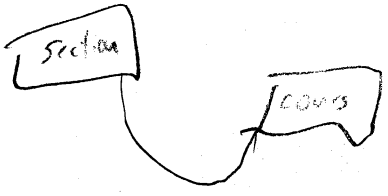
- explicit FDs (secondary FDs) from the db requirements analysis
- FDs (primary FDs) derived from ER diagram
- FDs derived from intuition

Degree	Cardinality	Primary FD
Binary	1:1	2 ways: key(one side) → key (one side) E1 – E2 E2 – E1
Binary	1:M	key(many side) → key (one side)  E2 → E1
Recursive	M:N	None (composite key from both sides) reflexivity
Ternary	1:1:1	3 ways: key(one), key(one) → key(one) Ex. Technician (empid)[1], Project (projname)[1], Notebook (nbno)[1] Empid, projname → nbno Empid, nbno → projname Nbno, projname → empid 
	1:1:M E2, E3 – E1 E1, E3 – E2	2 ways: key(one), key(many) → key(one) Ex. Each EMPLOYEE (empid) [M]assigned to a PROJECT (projname) [1]works at only one LOCATION (locname) [1] for that PROJECT, but can be at a different location for a different project. At a given location, an employee works on only one project. At a particular location, there can be many employees assigned to a given project. empid, projname → locname empid, locname → projname
	1:M:N E1, E3 → E2	1 way: key(many), key(many) → key (one) Ex. Each ENGINEER(empid) [M] working on a particular project has exactly one MANAGER (mgrid), but a PROJECT (projname) [M] may have many managers and an engineer may have many managers and many projects. A manager may manage several projects. projname , empid → mgrid
	M:N:P E1, E2, E3 → E1, E2, E3	None (composite key from all 3 sides) reflexive. Ex. Employees (empid) can use different skills (skilltype) on any one of the many projects (projname) and each project has many employees with various skills. none
Generalization	None	None (secondary FD only)

Any table B that is subsumed by another table A can potentially be eliminated. Table B is subsumed by another table A when all the attributes in B are also contained in A, and all data dependencies in B also occur in A.







ICE 04 – February 4

Given R: (A,B,C,D,E,F,G,H,J)

FDs: FD1: A →> B; FD2: A →> C; FD3: B → C
FD4: B → D; FD5: D → B; FD6: ABE → F
FD7: E → J; FD8: EG → H; FD9: H → G

1. Find a minimal cover for R. Show your solution.
2. Find all the candidate keys for R. Show your solution.
3. What is the highest normal form of R? Support your answer.

(1)

A – B
B – C
B – D
ABE – F
E – J
EG – H

A - B
B - C
B - D
AE - F
E - J
EG - H

(2)

L	M	R
AE	B	C, D F
G		H, J

A^+: ABCD

E^+: EJ

G^+: G

AEG^+: AEGBCDFJH

(3) Inf because there are no multi valued or composite,
2Nf: not in second because there are non prime attributes dependent on part of the key. Since there is only one key that's all we check.

RELATIONAL ALGEBRA and SQL

SQL Operators	Relational Algebra Operators
SELECT * FROM R WHERE <select condition>;	SELECT $\sigma_{\text{<selection condition>}} R$
SELECT <attribute list> FROM R;	PROJECT $\Pi_{\text{<attribute list>}} R$
SELECT R1.join_attributes, R2. join_attributes FROM R1, R2 WHERE <join condition>	THETA JOIN $R_1 \ |X|_{\text{<join condition>}} R_2$
SELECT R1.join_attributes, R2. join_attributes FROM R1, R2 WHERE < equal join condition>	EQUIJOIN $R_1 \ |X|_{\text{<equality join condition>}} R_2$ $R_1 \ |X|_{(\text{<join attribute 1>}), (\text{<join attribute 1>})} R_2$
SELECT R1.join_attributes FROM R1, R2 WHERE < equal join condition>	NATURAL JOIN $R_1 *_{\text{<equality join condition>}} R_2$ $R_1 *_{(\text{<join attribute 1>}), (\text{<join attribute 1>})} R_2$ Join attributes have same names: $R_1 * R_2$
SELECT <attribute list> FROM R1 UNION SELECT <attribute list> FROM R2	UNION $R_1 \cup R_2$
SELECT <attribute list> FROM R1 INTERSECT SELECT <attribute list> FROM R2	INTERSECT $R_1 \cap R_2$
SELECT <attribute list> FROM R1 MINUS SELECT <attribute list> FROM R2	DIFFERENCE $R_1 - R_2$
SELECT <attribute list> FROM R1, R2	CARTESIAN PRODUCT $R_1 \times R_2$

In Class Exercise – January 16, 2014:
Subquery & Update

COYOTE IDS: _____ **SCORE:** _____/ 15 pts

_____

1. Display the names, salaries and addresses for all employees whose salary is larger than the average salary in their department. Provide two solutions using correlated subquery. One solution should use a subquery in the FROM clause.

SOLUTION 1:

```
select e.fname, e.lname, e.salary, e.address
from employee e
where e.salary > ( select avg(ee.salary)
                   from employee ee
                   where ee.dno = e.dno);
```

SOLUTION 2:

```
Select e.fname, e.lname, e.salary, e.address
from employee e, (select avg(ee.salary) avg_salary, ee.dno
                  from employee ee
                  group by ee.dno) d
where e.dno = d.dno and e.salary > d.avg_salary;
```

2. Update the salary and department of Joyce English to match that of Alicia Zelaya. Use a correlated subquery to update.

SOLUTION:

~~Set (e.salary, e.dno) = (sele~~

```
update employee e
set (e.salary, e.dno) = (select ee.salary, ee.dno
                         from employee ee
                         where ee.fname = 'Alicia')
where e.fname = 'Joyce';
```

ICE 01 – February 4

Given R: (X, A, B, C) FDs: FD1: AX --> B; FD2: AX --> C; FD3: B --> A

1. Find all the candidate keys of R.
2. What is the highest normal form of R.
3. If R is not in BCNF
   a. why is it not in BCNF?
   b. Decompose R so that it will be in BCNF

① $AX \to B$
$AX \to C$
$B \to A$

L	M	R
X	AB	C

X^+
X

XA^+
XABC

XB^+
XBAC

② Inf: because there are no multi or composite.

2Nf: because FD3 has A dependent on part of a key, and B is not one of the two candidate keys

it is in 2Nf because all values on right are prime except for C in FD2 which AX is candidate key

3Nf it is in 3Nf because all left values are superkeys or the right side is prime.

3. ③ Not in BCNf because in FD3, B is not a superkey of R.

B. To decompose R to bcnf, move the problem attribute to a new relation leaving the left side in R. copy left side into new relation and use as primary key.

R: (X, B, C) – FD $BX \to C$

R1: (B, A) FD $B \to A$

ICE 02 – February 4

Consider the relation instance of the Student-Course relation schema.
Student-Course (Snum, Sname, Major, Cname,Time, Room)

Snum	Sname	Major	Cname	Time	Room
0110	Manning	CSE	CSE572	TTh4	JB-359
0110	Manning	CSE	Math272	MW3	JB-116
0110	Manning	CSE	NSCI306	TTh2	CE-150
1000	Rosenberg	Anthropology	Anth326	MWF9	UH-412
1000	Rosenberg	Anthropology	Math272	MW3	JB-116
2000	Manning	Statistics	NSCI306	TTh2	CE-150
2000	Manning	Statistics	Math272	MW3	JB-116
3000	Wilson	Accounting	CSE572	TTh4	JB-359
3000	Wilson	Accounting	ACCT211	MW4	JB-146

a. Identify at least one update anomaly, one insertion anomaly, and one deletion anomaly.
b. What are the undesirable functional dependencies in the relation instance of the Student-Course relation schema shown above?

(a) insert

Student can't be inserted unless enrolled in a class.

delete

if you delete major ~~and~~, else then the major is deleted as well

Student 0110 is deleted as well

if delete a Student, could delete all references to a course.

update

if you wanted to change 0110's major, you'd have to update it in multiple tuples.

(b) Major

ICE 03 – February 4

yes gets all attributes

Consider the relation REFRIG (Model#, Year, Price, Manuf_Plant, Color) and the following FDs:

FD1: Model# → Manuf_Plant
FD2: Model#, Year → Price
FD3: Manuf_Plant → Color

model#, year
model#
year
manuf-plant
color
price

a. Evaluate each of the following as a candidate key for REFRIG, giving reasons why it can or cannot be a key:
   1) Model#
   2) Model#, Year
   3) Model#, Color

model# +
model#
manuf-plant
color

model#, color +
model#
color
manuf-plant

no because dont get all values from closure.

b. Based on the above (a) key determination, state whether the relation REFRIG is in 3NF and in BCNF, giving appropriate reasons.

c. Consider the decomposition of REFRIG into
   R1(Model#, Year, Price) and
   R2 (Model#, Manuf_Plant, Color)
   Is this decomposition lossless? Show why.

(b) 3NF: not in 2nf because manuf-plant is determined by part of the only candidate key, therefore can't be in 3NF. also FD3 has a transitive dependency because left is not superkey and right is not prime.

BCNF: it is not in BCNF because not in 2NF and because on FD1, model# is not a superkey of Refrig.

(c)

	model #	year	price	manuf plant	color
R1	a1	a2	a3	b14 a4	b15 a5
R2	a1	b22	b23	a4	a5

it is lossless because first row is all A's

California State University, San Bernardino
School of Computer Science & Engineering
CSE580 Winter 2014 Midterm ----- SOLUTIONS

There are three pages to this QUESTIONS Sheets and 5 questions and 1 Extra Credit question. Please use the provided answer sheets to write your answer.

Return these QUESTIONS SHEETS together with your ANSWER SHEETS.

QUESTIONS

1. Given **FACULTY** (InstructorName, Faculty#, StudentName, Student#, DeptName, Course#, CourseDescription, Grade, Committee#)

   and the following FDs:

   FD1: Faculty# → InstructorName
   FD2: Faculty#→ Committee#
   FD3: Faculty# → DeptName, Course#
   FD4: Student# → StudentName
   FD5: Student#, Course# → Grade
   FD6: Course# → CourseDescription

a. [5 pts] Find all the candidate keys of FACULTY. Must show how you got the candidate key(s).

Left Only (Include)	Both Sides (Not Sure)	Right Only (Exclude)
Faculty#	Course#	InstructorName
Student#		StudentName
		CourseDescription
		DeptName
		Grade
		Committee#

Faculty#$^+$ = {Faculty, InstructorName, DeptName, Course#, Committee#, CourseDescription}

Student#$^+$ = {Student#, StudentName}

(Faculty#, Student#)$^+$ = { Faculty, InstructorName, DeptName, Course#, Committee#, CourseDescription, Student#, StudentName, Grade} = Faculty

... **Candidate Key**: composite (Faculty#,Student#)

b. [10 pts] What is the highest normal form of FACULTY? Support your answer.

FACULTY in is 1NF since each attribute is single-valued and atomic.
It is not in 2NF since FD1, FD2, FD3,FD4 violate full functional dependency.
The LHS is only part of full key (Faculty#, Student#).

c. [10 pts] If FACULTY is not in BCNF,
   i.) Why is it not in BCNF? Since all FDs do not have CK on the LHS.
   ii.) Decompose FACULTY so that it will be in BCNF

Need to remove Faculty, InstructorName, Committee#, DeptName,, StudentName, CourseDescription from FACULTY.

GRADE (,Student#,Course#, Grade) Composite Key (Course#,Student#)
FD5: Student#, Course# → Grade

This is in 1NF, 2NF, 3NF and BCNF

STUDENT(Student#, StudentName)
FD4: Student# → StudentName

This is in 1NF, 2NF, 3NF and BCNF

FACULTY2(Faculty#, FacultyName, Committee#, DeptName, Course#)
FD1: Faculty# → InstructorName **FD2**: Faculty#→ Committee#
FD31: Faculty# → DeptName
FD32: Faculty# → Course#

This is in 1NF, 2NF, 3NF and BCNF

COURSE(Course#, CourseDescription)
FD6: Course# → CourseDescription

2. Consider the NIH Grants Database described below:
In this question, you will write SQL queries over recent research grants from the National Institute of Health (NIH)awarded to professors at several major universities. The NIH is a significant funder of academic research in the United States. Researchers submit proposals to various programs. Panels of peer reviewers (other academics in related fields) decide which proposals will be funded. After grants are funded (typical amounts range from $50,000 to many millions of dollars), they are administered by the university on behalf of the researchers. Researchers pay students and staff, purchase equipment and supplies, and pay expenses (such as travel) from these grants.

The NIH Grants database consists of six entities:

- Grants, with their associated meta-data (e.g., amount of funding, start and end date, the researchers who work on them, etc.),
- Organizations (e.g., universities) which receive grants,

- Researchers who receive grants,
- Programs run by NIH that award grants,
- Managers at NIH who run programs that award grants, and
- Fields (research areas) that describe high level topic-areas for grants.

Nine relational tables implement the entities and relationships for the NIH Grants Database above:

NOTE: PK is denoted by underlined attribute(s); FK is denoted by bold italicized ***attribute.***

ORGS (Id, Name, StreetAddr, City, State, Zip, Phone)

RESEARCHERS (Id, Name, ***Org***) //Org refers to the ORGS(Id)

PROGRAMS (Id, Name, Directorate) //Directorate – top level part of NIH that runs this program

MANAGERS (Id, Name)

GRANTS(Id, Title, Amount, Org, ***PI***, ***Manager***, Started, Ended, Abstract)
//PI – principal investigator on the grant and refers to RESEARCHES (Id)
//Manager – refers to MANAGERS(Id)

GRANT_RESEARCHERS (***ResearcherID, GrantId***) // M-N RESEARCHERS - GRANTS
// (ResearcherId, GrantId) : composite PK
// ResearcherId refers to RESEARCHERS(Id)
// GrantId refers to GRANTS(Id)

FIELDS(Id, Name)

GRANT_FIELDS (***GrantId, FieldId***) //M-N GRANTS - FIELDS
// (GrantId, FieldId) : composite PK
// FieldId refers to FIELDS(Id)
// GrantId refers to GRANTS(Id)

GRANT_PROGRAMS (***GrantId, ProgramId***) //M-N GRANTS - PROGRAMS
// (GrantId, ProgramId) : composite PK
// ProgramId refers to PROGRAMS(Id)
// GrantId refers to GRANTS(Id)

2a. [10 pts] Write a SQL query that uses a subquery in the FROM clause to find the name of the grant with the smallest id.

```
SELECT title
FROM grants, (SELECT min(id) AS minid
              FROM grants) AS MinGrant
WHERE grants.id = MinGrant.minid;
```

2b. [10 pts] Write a SQL query that uses a subquery in theWHERE clause to find the name of the grant with the smallest id.

```
SELECT title
FROM grants
WHERE grants.id = (SELECT min(id)
                   FROM grants);
```

2c. [10 pts] CHOOSE one from (i) and (ii) below. Circle your choice.

(i) Find the grants with researchers that matched a list of researcher ids from California State University, San Bernardino.

```
SELECT grants.title
FROM grants, grant_researchers
WHERE grants.id = grant_researchers.grantid
   AND grant_researchers.researcherid IN (SELECT  Id
                                          FROM  researchers
                                          WHERE org = (SELECT Id
                                                       FROM orgs
                                                       WHERE UPPER(name) like
                                                            'CSUSB')
```

(ii) Display the list of the ids of all grants written by the researcher, Albert Einstein. [5 pts]

```
SELECT GrantId
FROM grant_researchers
WHERE ResearcherId  = (SELECT id
                       FROM researchers
                       WHERE lower(name) like 'albert einstein')
```

3. Consider the relation Contracts below.

Contracts (ContractId, SupplierId, ProjectId, DeptId, PartId, Qty, Value) and its FDs:

ContractId → ContractId, SupplierId, ProjectId, DeptId, PartId, Qty, Value
ProjectId, PartId → ContractId //A project purchases a given part using a single contract
SupplierId, DeptId → PartId //A department purchases at most one part from a supplier
ProjectId → SupplierId //A project uses only with one supplier

In this relation, the contract C with ContractId is an agreement that supplier S (SupplierId) will supply Q items of part P (PartId) to project J(ProjectId) associated with department D (DeptId); the value V of this contract is equal to Value.

[20 pts] Find a minimal cover for Contracts. Show all the details – step by step.

Step 1: Rewrite each FD in canonical form ... only one attribute on the RHS.

Fd11: ContractId → ContractId //Trivial ... can remove

Fd12: ContractId → SupplierId
Fd13: ContractId → ProjectId
Fd14: ContractId → DeptId
Fd15: ContractId → PartId
Fd16: ContractId → Qty
Fd17: ContractId → Value
Fd2: ProjectId, PartId → Contract
Fd3: SupplierId, DeptId → PartId
Fd4: ProjectId → SupplierId

Step 2: Remove extraneous attributes ... Look only at FDs with more than one attribute on LHS .... Fd2, Fd3 .... These do not have extraneous attributes

Step 3: Remove redundant fds.

Fd15: ContractId → PartId is redundant Check if ContractId$^+$ contains PartId without using Fd15!

Because ContractId$^+$ = {ContractId //Fd11,
SupplierId //Fd12,
ProjectId //Fd13,
DeptId //Fd14,
Qty //Fd16,
Value //Fd17,
PartId //Fd3 ..... since PartId is in ContractId$^+$
We can stop at this point and say that ContractId → PartId is redundant and therefore remove Fd15

Another solution: Fd12: ContractId → SupplierId;
Fd14: ContractId → DeptId;
Union or Additive Rule: ContractId → SupplierId, DeptId
Fd3: SupplierId, DeptId → PartId
ContractId →PartId .... transitivity

Fd12: ContractId → SupplierId is also redundant Check if ContractId$^+$ contains SupplierId without using Fd12!

Another solution is to use inference rules and the given Fds except Fd12.
Fd13: ContractId → ProjectId and Fd4: ProjectId → SupplierId
implies ContractId → SupplierId (Transitivity Rule)

So the minimal cover for Contracts is Fd13, Fd14, Fd16, Fd17,Fd2, Fd3, Fd4!

4. a. [5 pts] Identify a delete anomaly for relation EMP_DEPT in Figure 15.4, page 508.
If we delete Borg, James we inadvertently lose information about Headquarters Dept!
b. [5 pts] Identify an update anomaly for relation EMP_PROJ in Figure 15.4, page 508.
Changing ename,/pname,/plocation will imply changing ALL records with that ename/pname/plocation
c. [5 pts] Identify an insert anomaly for relation EMP_PROJ in Figure 15.4, page 508.
Since (ssn, pnumber) is composite key.... Cannot insert a new employee without this employee working in at least 1 project; similarly cannot insert a new project without this project having at least one employee working in it.

5. Given R(A,B,C,D) and its decomposition into R1(A,D); R2(A,C); R3(B,C,D) and FDs for R:
FD1: A→B FD2: B → C FD3: CD → A

[10 pts] Is this decomposition lossless? Support your answer.

	A	B	C	D
(AD)	a1	b12	b13	a4
(AC)	a1	b22	a3	b24
(BCD)	b31	a2	a3	a4

FD1: A → B

	A	B	C	D
(AD)	a1	b12	b13	a4
(AC)	a1	b12	a3	b24
(BCD)	b31	a2	a3	a4

FD2: B → C

	A	B	C	D
(AD)	a1	b12	a3	a4
(AC)	a1	b12	a3	b24
(BCD)	b31	a2	a3	a4

FD2: CD → A

	A	B	C	D
(AD)	a1	b12	a3	a4
(AC)	a1	b12	a3	b24
(BCD)	a1	a2	a3	a4

Since there is a row where all the variables are a's ... the decomposition is lossless!

EXTRA CREDIT:

Given: Instructor(InstructorName, InstructorId, InstructorOffice#, StudentId, StudentName, StudentOffice#,StudentDesignatedRefrigeratorId, RefrigeratorOwnerId, RefrigeratorId, RefrigeratorSize, AssistantName, AssistantId, AssistantOffice)

Suppose the data has the following properties or constraints:

A. Students can work for multiple instructors. // M:N Students - Instructors

StudentId, InstructorId → StudentId, InstructorId //Trivial FD

B. Refrigerators are owned by one instructor. //1:M Instructors - Refrigerators
RefrigeratorId → InstructorId

C. Instructors can own multiple refrigerators. //1:M Instructors - Refrigerators

D. Students can only use one refrigerator. //1:M Students - Refrigerators
StudentId → RefrigeratorId

E. Assistants can work for multiple instructors. //1:M Instructors - Assistants

F. Instructors only have a single Assistant. //1:M Instructors – Assistants
InstructorId → AssistantId

Obvious FDs:

InstructorId → InstructorName, InstructorOffice# , AssistantId //instructor info
//AssistantId: Constraint F

StudentId → StudentName, StudentOffice# //student info

RefrigeratorId → RefrigeratorSize, RefrigeratorOwnerId //refrigerator info
//RefrigeratorOwner: Constraint C

AssistantId → AssistantName, AssistantOffice //Assistant info

StudentId, InstructorId → StudentId, InstructorId //Constraint A

[10 points]:

(1) Put this relation into 3rd normal form by writing out the decomposed tables;
RELATIONS:

INSTRUCTOR (InstructorId, InstructorName, InstructorOffice#, AssistantId)
InstructorId → InstructorName, InstructorOffice# , AssistantId
FK: AssistantId refers to ASSISTANT(AssistantId)

STUDENT (StudentId, StudentName, StudentOffice#, StudentDesignatedRefrigeratorId)
StudentId → StudentName, StudentOffice#
StudentId → RefrigeratorId //rename to StudentDesignatedRefrigeratorId
FK: StudentDesignatedRefrigeratorID refers to REFRIGERATOR(RefrigeratorId)

REFRIGERATOR (RefrigeratorId, RefrigeratorSize, RefrigeratorOwnerId
RefrigeratorId → RefrigeratorSize, RefrigeratorOwnerId
FK: RefrigeratorOwnerId refers to Instructor(InstructorId)

ASSISTANT(AssistantId, AssistantName, AssistantOffice)
AssistantId → AssistantName, AssistantOffice

STUDENT_WORKS_FOR (StudentId, InstructorId) //composite key (StudentId, InstructorID
FK: StudentId refers to Student(StudentId)
FK: InstructorId refers to Instructor(InstructorId)

(2) Designate keys in your tables by underlining them; and

(3) Designate foreign keys by drawing an arrow from a foreign key to the primary key it refers to.