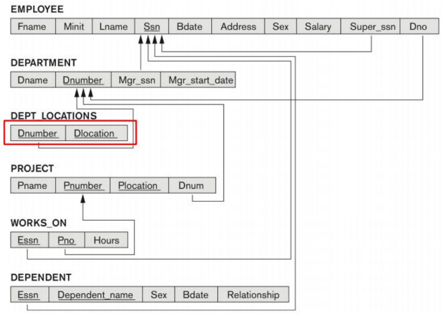
A weak entity can be identified uniquely only by considering the primary key of another (owner) entity

INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, ‘Smith’, ‘smith@ee’, 18, 3.2)

• RESTRICT option: reject the deletion
• CASCADE option: propagate the new primary key value into the foreign keys of the referencing tuples
• SET NULL option: set the foreign keys of the referencing tuples to NULL



Keys:

Superkey An attribute or combination of attributes that uniquely identifies each entity in a table.
Candidate key A minimal superkey that does not contain a subset of attributes that is itself a superkey.
Primary key A candidate key selected to uniquely identify all other attribute values in a given row. No nulls allowed. **Secondary key** An attribute or combination of attributes used strictly for data-retrieval purposes.
Foreign key An attribute or combination of attributes in one table whose value must either - match the primary key in another table, or - be null (contain no value, i.e. empty)

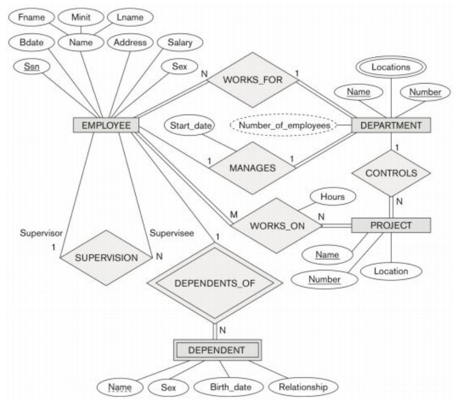
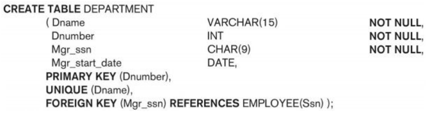


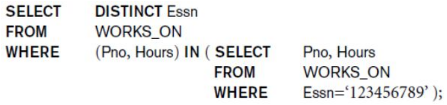
Table Drop behavior options:
CASCADE: all other elements, constraints, views that reference that table are dropped, including the table itself.
RESTRICT: only if not referenced in any constraints.

SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary) FROM EMPLOYEE;

SELECT SUM (Salary) AS Total_Sal, MAX (Salary) AS Highest_Sal, MIN (Salary) AS Lowest_Sal, AVG (Salary) AS Average_Sal FROM EMPLOYEE;

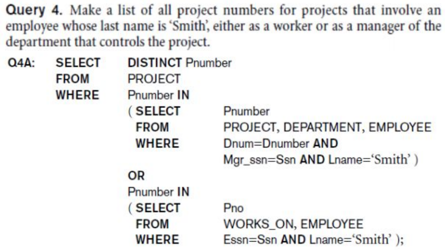


```
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)         NOT NULL,
          Dnumber               INT                 NOT NULL,
          Mgr_ssn               CHAR(9)             NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
```



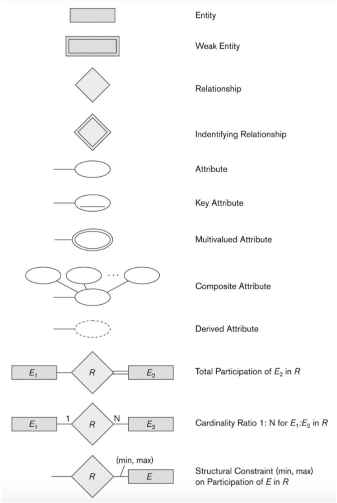
```
SELECT     DISTINCT Essn
FROM       WORKS_ON
WHERE      (Pno, Hours) IN ( SELECT     Pno, Hours
                             FROM       WORKS_ON
                             WHERE      Essn=‘123456789’ );
```

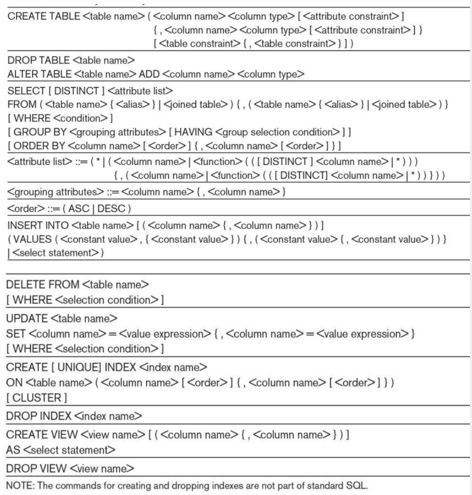
SELECT Dno, COUNT (*), AVG (Salary) FROM EMPLOYEE GROUP BY Dno;



Query 4. 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 the project.

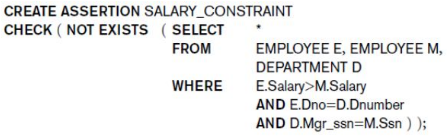
```
Q4A:   SELECT    DISTINCT Pnumber
       FROM      PROJECT
       WHERE     Pnumber IN
                 ( SELECT       Pnumber
                   FROM         PROJECT, DEPARTMENT, EMPLOYEE
                   WHERE        Dnum=Dnumber AND
                                Mgr_ssn=Ssn AND Lname=‘Smith’ )
                 OR
                 Pnumber IN
                 ( SELECT       Pno
                   FROM         WORKS_ON, EMPLOYEE
                   WHERE        Essn=Ssn AND Lname=‘Smith’ );
```


Entity
Weak Entity
Relationship
Indentifying Relationship
Attribute
Key Attribute
Multivalued Attribute
...
Composite Attribute
Derived Attribute
E_1
R
E_2
Total Participation of E_2 in R
E_1
1
R
N
E_2
Cardinality Ratio 1: N for E_1:E_2 in R
(min, max)
R
E
Structural Constraint (min, max) on Participation of E in R



```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                            { , <column name> <column type> [ <attribute constraint> ] }
                            [ <table constraint> { , <table constraint> } ] )
DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>
SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]
<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                     { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )
<grouping attributes> ::= <column name> { , <column name> }
<order> ::= ( ASC | DESC )
INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )
DELETE FROM <table name>
[ WHERE <selection condition> ]
UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[ WHERE <selection condition> ]
CREATE [ UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ]
DROP INDEX <index name>
CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>
DROP VIEW <view name>
NOTE: The commands for creating and dropping indexes are not part of standard SQL.
```

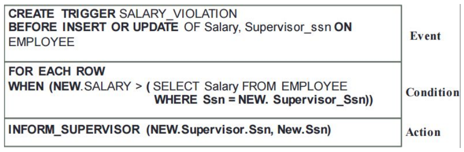
Unary Relational Operations
• SELECT (symbol: σ (sigma))
• PROJECT (symbol: π (pi))
• RENAME (symbol: ρ (rho))
Set Relational Operations
• UNION (∪), INTERSECTION (∩), DIFFERENCE (or MINUS, –)
• CARTESIAN PRODUCT (x)



```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS   ( SELECT     *
                       FROM       EMPLOYEE E, EMPLOYEE M,
                                  DEPARTMENT D
                       WHERE      E.Salary>M.Salary
                                  AND E.Dno=D.Dnumber
                                  AND D.Mgr_ssn=M.Ssn ) );
```

Algebra:
DEP5_EMPS ← σ Dno=5 (EMPLOYEE)
RESULT ← πFname, Lname, Salary(DEP5_EMPS)

Calculus:
Example: Retrieve the name and address of all employees who work for the ‘Research’ department. The query can be expressed as: {t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) and (∃ d) (DEPARTMENT(d) and d.DNAME=‘Research’ and d.DNUMBER=t.DNO) }



```
CREATE  TRIGGER SALARY_VIOLATION
BEFORE  INSERT  OR UPDATE  OF Salary, Supervisor_ssn ON
EMPLOYEE                                                        Event

FOR  EACH  ROW
WHEN  (NEW.SALARY > ( SELECT  Salary FROM  EMPLOYEE
                      WHERE  Ssn = NEW. Supervisor_Ssn))        Condition

INFORM_SUPERVISOR  (NEW.Supervisor.Ssn, New.Ssn)                Action
```

Consider the relation:
EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
Insert Anomaly:
Cannot insert a project unless an employee is assigned to it.
Conversely, cannot insert an employee unless an he/she is assigned to a project.

• Consider the relation:
EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours) • Delete Anomaly:
When a project is deleted, it will result in deleting all the employees who work on that project.
Alternately, if an employee is the sole employee on a project, deleting the corresponding project.

• Relations should be designed such that their tuples will have as few NULL values as possible

• Attributes that are NULL frequently could be placed in separate relations (with the primary key)

For relation R(A, B, C, D) with its extension.
Trivial: AB->A This is when RHS is subset of LHS
Non-Trivial: A -> C, A -> AC RHS is not a subset of LHS

First Normal Form
• Disallows
  • composite attributes
  • multivalued attributes
  • nested relations; attributes whose values for an individual tuple are Non-atomic

Example:
R (ABCD), FD: AB -> D, B -> C
AB is a candidate key, we need to decompose the relation so no partial dependency: R1(ABD), R2(BC)

Example:
R (ABCD), FD: AB -> C, C -> D
AB is a candidate key, no partial dependency, so it is 2NF C is a problem, C may be NULL We need to decompose on R1(ABC), R2(CD) (So C is forced to be a key)

Second Normal Form
Uses the concepts of FDs, primary key
• Definitions
  • Prime attribute: An attribute that is member of the primary key K
  • Full functional dependency:
    a FD Y -> Z where removal of any attribute from Y means the FD does not hold any more
• Examples:
  • {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold
  • {SSN, PNUMBER} -> ENAME is not a full FD (it is called a partial dependency) since SSN -> ENAME also holds

A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on the primary key

Third Normal Form
• Definition:
  • Transitive functional dependency:
    a FD X -> Z that can be derived from two FDs
    X -> Y and Y -> Z
  • Examples:
    • SSN -> DMGRSSN is a transitive FD
      • Since SSN -> DNUMBER and DNUMBER -> DMGRSSN hold
    • SSN -> ENAME is non-transitive
      • Since there is no set of attributes X where SSN -> X and X -> ENAME

A relation schema R is in third normal form (3NF) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key

BCNF
• A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever an FD X → A holds in R, then X is a superkey of R

A relation NOT in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations.

• Two FDs exist in the relation TEACH:
  • fd1: { student, course} ⟶ instructor
  • fd2: instructor ⟶ course
• {student, course} is a candidate key for this relation and that the dependencies shown follow the pattern above.
  • So this relation is in 3NF but not in BCNF

Three possible decompositions for relation TEACH
D1: {student, instructor} and {student, course}
D2: {course, instructor} and {course, student}
D3: {instructor, course } and {instructor, student}

Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).

• R(ABCDEFGH), FDs: AB -> C, A -> DE, B -> F, F -> GH,	CKs: AB	1NF
• R(ABCDE), FDs: CE -> D, D -> B, C -> A,	CKs: CE	1NF
• R(ABCDEF), FDs: AB -> C, DC-> AE, E -> F,	CKs: ABD, BCD	1NF
• R(ABCDEGHI), FDs: AB -> C, BD -> EF, AD -> GH, A -> I,	CKs: ABD	1NF
• R(ABCDE), FDs: AB -> CD, D -> A, BC -> DE,	CKs: AB, BD, BC	3NF
• R(ABCDE), FDs: BC -> ADE, D -> B,	CKs: BC, CD	3NF
• R(ABC), FDs: A -> B, B -> C, C -> A,	CKs: A, B, C	BCNF

• 1st normal form
- All non-prime attributes depend on the key

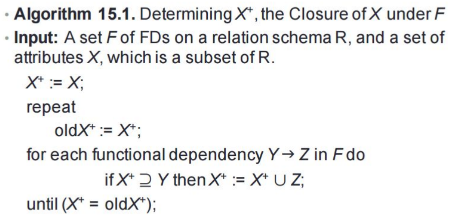
• 2nd normal form
- All non-prime attributes depend on the whole key (no Partial dependency, non-prime attribute depends on prime attribute)

• 3rd normal form
- All non-prime attributes depend on nothing but the key
  (no transitive dependency, non-prime attribute finds another non-prime attribute)

• BCNF
- All attributes depend on nothing but the key (RHS of all rules are superkeys)

Armstrong's inference rules:
- IR1. (Reflexive) If Y subset-of X, then X → Y
- IR2. (Augmentation) If X → Y, then XZ → YZ
  - (Notation: XZ stands for X U Z)
- IR3. (Transitive) If X → Y and Y → Z, then X → Z
- IR4. (Sudo Transitive) If X→Y and (W,Y) → Z then (W,X) → Z
- IR5. (Decomposition) If X → (Y,Z) then X → Y and X → Z

Example:
Contracts(cid,sid,jid,did,pid,qty,value), and:
- C is the key: C -> CSJDPQV
- Project purchases each part using single
  contract: JP -> C
- Dept purchases at most one part from a
  supplier: SD -> P
- JP -> C, C -> CSJDPQV imply JP -> CSJDPQV
- SD -> P implies SDJ -> JP
- SDJ -> JP, JP -> CSJDPQV imply SDJ -> CSJDPQV



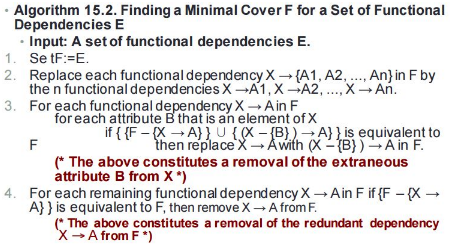
- **Algorithm 15.1.** Determining X^+, the Closure of X under F
- **Input:** A set F of FDs on a relation schema R, and a set of attributes X, which is a subset of R.

```
$X^+ := X$;
repeat
    old$X^+ := X^+$;
for each functional dependency $Y \rightarrow Z$ in $F$ do
            if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z$;
until ($X^+$ = old$X^+$);
```

Example of Closure of attributes:
Example 1: R (ABCDEFG), FD: {A → B, BC → DE, AEF → G}, Find (AC)+
Soln: ABCDE
Example 2: R (ABCDE), FD: {A → BC, CD → E, B → D, E → A}, Find (B)+
Soln: BD
Example 3: R (ABCDEF), FD: {AB → C, BC → AD, D → E, CF → B}, Find (AB)+
Soln: ABCDE

Minimal cover
A set of FDs is minimal if it satisfies the following conditions:
1. Every dependency in F has a single attribute for its RHS (canonical form).
2. We cannot replace any dependency X →A in F with a dependency X-Y →A, where Y is a proper-subset-of X and still have a set of dependencies that is equivalent to F.
3. We cannot remove any dependency from F and have a set of dependencies that is equivalent to F

- Indexes can be:
  - Single level ordered indexes
  - Multilevel indexes
  - Dynamic multilevel indexes
- Indexes can be:
  - Dense: has an index for each key value.
  - Sparse: has index entries for only some of the search values



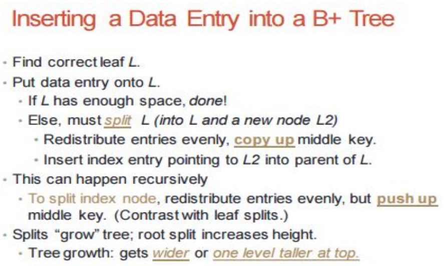
- **Algorithm 15.2. Finding a Minimal Cover F for a Set of Functional Dependencies E**
  - **Input: A set of functional dependencies E.**

```
1. Se tF:=E.
2. Replace each functional dependency X → {A1, A2, ..., An} in F by
   the n functional dependencies X →A1, X →A2, ..., X → An.
3. For each functional dependency X → A in F
       for each attribute B that is an element of X
               if { {F – {X → A} } ∪ { (X – {B} ) → A} } is equivalent to
   F                   then replace X → A with (X – {B} ) → A in F.
       (* The above constitutes a removal of the extraneous
       attribute B from X *)
4. For each remaining functional dependency X → A in F if {F – {X →
   A} } is equivalent to F, then remove X → A from F.
       (* The above constitutes a removal of the redundant dependency
       X → A from F *)
```

4NF
Definition:
• A relation that is in Boyce–Codd normal form and does not contain nontrivial multi-valued dependencies.
• The normalization of BCNF relations to 4NF involves the removal of the MVD from the relation by placing the attribute(s) in a new relation along with a copy of the determinant(s).

Bfr = floor(B/R), B = bytes, R = record size
blocks = ceiling(r/bfr), r is # records, bfr is above

- For each **query** in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each **update** in the workload:
  - Which attributes are involved in selection/join conditions?
  - How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.



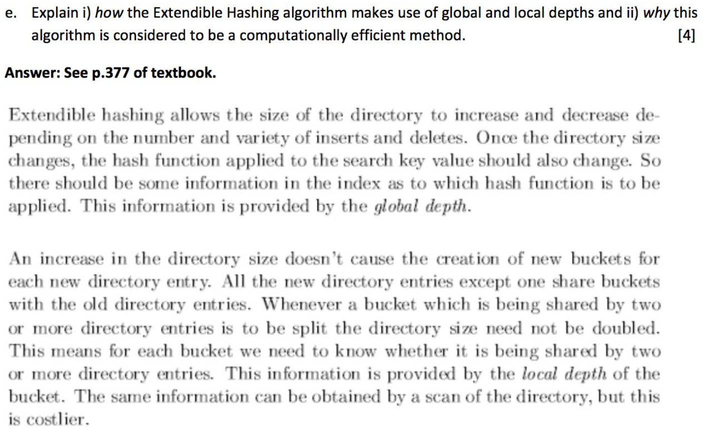
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L.*
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to *L2* into parent of *L.*
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits “grow” tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

• Many alternative file organizations exist, each appropriate in some situation.

• Index is a collection of data entries plus a way to quickly find entries with given key values.

• If selection queries are frequent, sorting the file or building an index is important.
- Hash-based indexes only good for equality search.
- Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)



e. Explain i) *how* the Extendible Hashing algorithm makes use of global and local depths and ii) *why* this algorithm is considered to be a computationally efficient method. [4]

Answer: See p.377 of textbook.

Extendible hashing allows the size of the directory to increase and decrease depending on the number and variety of inserts and deletes. Once the directory size changes, the hash function applied to the search key value should also change. So there should be some information in the index as to which hash function is to be applied. This information is provided by the *global depth*.

An increase in the directory size doesn't cause the creation of new buckets for each new directory entry. All the new directory entries except one share buckets with the old directory entries. Whenever a bucket which is being shared by two or more directory entries is to be split the directory size need not be doubled. This means for each bucket we need to know whether it is being shared by two or more directory entries. This information is provided by the *local depth* of the bucket. The same information can be obtained by a scan of the directory, but this is costlier.

Choice of Indexes: Our Approach
• Consider the most important queries in turn.
• Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
- Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
- For now, we discuss simple 1-table queries.

• Before creating an index, must also consider the impact on updates in the workload!
- Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

• Try to choose indexes that benefit as many queries as possible