Normalisation :-

Keys

Normalization of data can be considered a process of Analysing the given relation schemas based on their FDs and primary keys to acheive the dpsirable propesties of

(1) Minimizing redundancy and
(2) Minimizing the insertion, deletion and update anomalies

They are three types of Normalization:-

(1) First Normal form
(2) Second Normal form
(3) Third Normal-form
(4) Boyce-codd Normal form

First Normal form:-

* A relation will be 1NF if it contains an atomic value
* It states that an attribute of a table cannot hold multiple values. it must hold only single-valued attribute
* First normal-form disallows the multivalued attributes composite attribute, and their combination

Ex:- Relation EMPLOYEE is not in INF because of multivalued attribute EMP_PHONE

EMPLOYEE Table:-

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Mary	8574783832	Hary
12	sam	7389037239, 8589830302	punjab

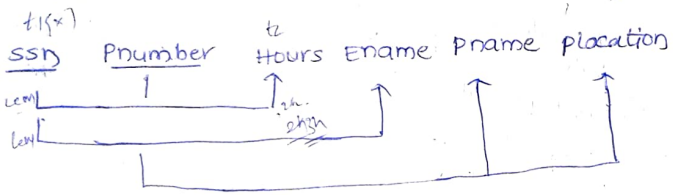
decomposition of the EMPOLEE Table into 1NF has bee ~~Emp~~ below.

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	harry	8574783832	Haryana
12	sam	7390372389	Punjab
12	sam	8589830302	Punjab

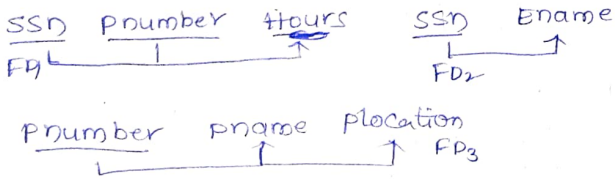
Second Normal form (2NF)

* In the 2NF, relational must be in 1NF
* In the second Normal form, all ~~co~~ non-~~functio~~ keys attributes are fully functional dependent on the primary key.

(a) EMP_PROJ



(b) 2NF Normalization



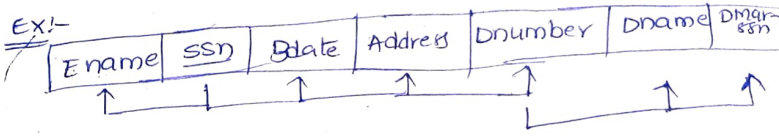
3rd Normal Form:-

Third Normal formal form (3NF) is based on the concept of Transitive dependency.

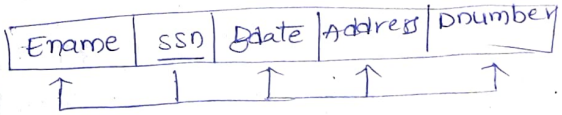
* A functional dependency $X \rightarrow Y$ in a relation schema R is a Transitive dependency
* if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

Def:- A relation schema R is in 3NF if it satisfies 2NF and no non-prime attribute of R is transitively dependent on the primary key.

Ex:-



3rd Normalisation:-



Boyce-Codd Normal form:-

Def:- A relation schema R is in BCNF if whenever a non-trivial functional dependency $X \rightarrow A$ holds in R, then X is a superkey of R.

Non-trivial functional dependency:-

* BCNF is the advance version of 3NF it is stricter than 3NF
* A Table is in BCNF if every function dependency $X \rightarrow Y$
  X is a super key of the table.
* For BCNF, the table should be in 3NF and for every FD, LHS is super key

Ex:- Let's assume there is a company where employee work in more than one department.

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO

FDs are:

EMP_ID → EMP_COUNTRY
EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Super keys are EMP_ID, EMP_DEPT

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose in three tables

EMP_ID	EMP_COUNTRY

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO

EMP_ID Mapping:-

EMP_ID	EMP_DEPT

Candidate Keys:-

1st Table: EMP_ID
2nd Table: EMP_DEPT
3rd Table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both FDs is a keys.

Indexes

Indexing:-

Indexing is a data structure technique which allows you to quickly retrieve records from a database file.

An Index is a small table having only two columns. The

* First column comprises a copy of the primary or candidate key of a table.
* It second column contains set of pointers for holding the Address of the disk block where that specific key value stored

An Index-

* takes a search key as input
* Efficiently returns collection of matching records

They are three types of Indexes:-

(1) primary Indexes — Dense, Sparse

(2) clustering Indexes

(3) secondary Indexes

Indexing in database is defined based on its indexing attributes. Two main types of Indexing method

- primary Indexing
- secondary Indexing

primary Index in DBMS:-

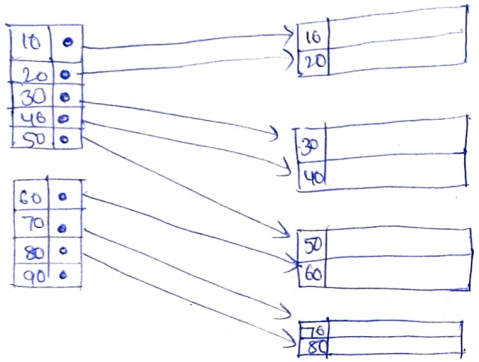
primary index is an ordered file which is fixed length size of two fields. The first field is the same a primary key and second field is pointed to that specific data block. In the primary Index, there is always one to one relationship between the entries in the Index table.

The primary Indexing in DBMS is also further divided into two types:

* Dense Index
* Sparse Index.

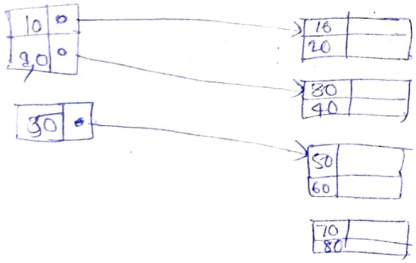
Dense Index:-

In a dense Index, a record is created for every search key valued in the database. this helps you to search faster but needs more space to store index records.
In this Indexing, Method records contain search key value and points to the real record on the disk.



Sparse Index:-

It is an index record that appears for only some of the values in the file. Sparse Index help you to resolve the issues of dense Indexing in DBMS. In this Method of Indexing technique, a range of Index column stores the same data block address, when data needs to be retrieved the block address will be fetched



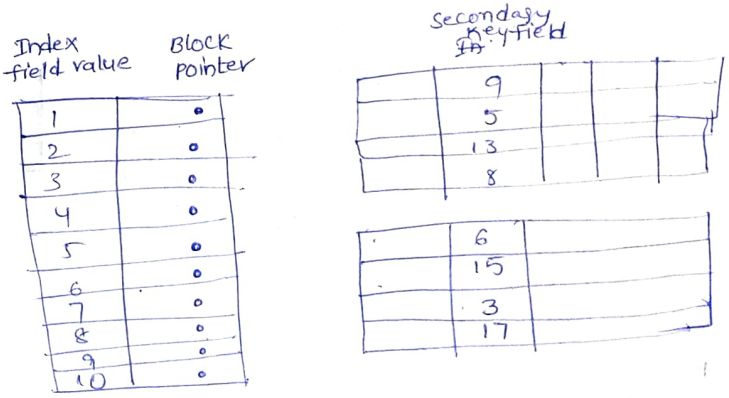
Secondary Index:-

A secondary index provides a secondary means of accessing a datafile for which some primary access already exist.

* The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on non-key field with duplicate values.
* The index again an ordered file with two fields.
* the first field is of the same data type as some non-ordering field of the data file that is an "Indexing field".
* Second field is either a block pointer or a record pointer.
* A secondary index usually needs more storage space and longer search time than does a primary Index because of its larger number of entries

Ex:-

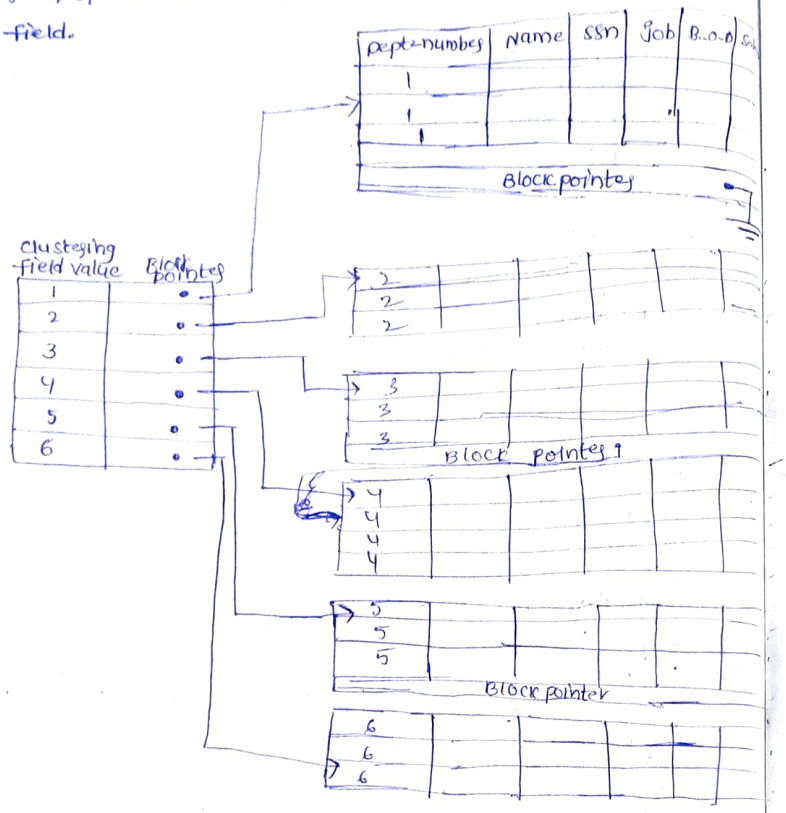
A dense secondary Index on a non-ordering key field of a file.



Clustering Index

In a clustered Index, records themselves are stored in the Index and not pointers. Sometimes the Index is created on non-primary key columns which might not be unique for each record. In such a situation, you can group two or more columns to get the unique values and create an index which is called clustered index. this also helps you to identify the record faster.

Ex:-
clustering index with a seperate block cluster for each group of records that share the same value for the clustering field.



Multilevel Index:-

multilevel Indexing in database is created when a primary Index does not fit in memory. In this type of Indexing method, you can reduce the no. of disk accesses to short any record and kept on a disk as sequential file and create a sparse base on that file.

B+ Tree:-

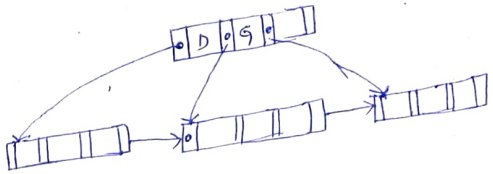
A B+ Tree is primarily utilized for implementating dynamic indexing on multiple level. Compare to B Tree and B+ Tree stores the data pointer only at the leaf nodes of the tree. Which makes search more process more accurate and faster.

search operation:-

Structure of B+ Tree:-

In the B+ Tree, every leaf node is at equal distance from the root node. The B+ Tree of the order n where n is fixed for every B+ Tree

* It contain an internal node and leaf node



Internal nodes:-

* An internal node of the B+ Tree can contain atleast n/2 record pointer exapt the root node.
* At most, an internal node of the tree contains 'n' pointer

Leaf nodes:-

* the leaf node of the B+Tree can contain atleast n/2 record pointers and n/2 key values
* Atmost, leaf node contains n record pointer and n key values
* Every leaf node of the B+ Tree contains one block pointer P to point to next leaf node.

Insertion:-

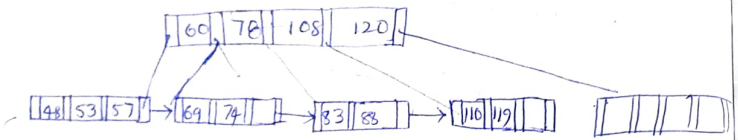
Step-1:- insert the new node as a leaf node

Step-2:- if the leaf doesn't have required space, split the node and copy the middle node to the next index node.

Step 3:- if the index doesn't have required space, split the node and copy the middle element to the next index page

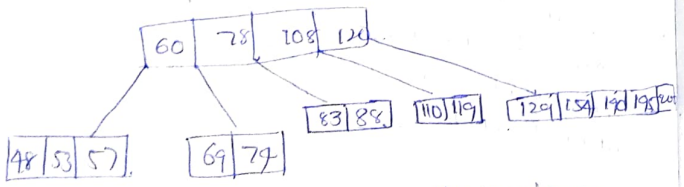
Ex:-

Insert 195 into B+ Tree of order 5

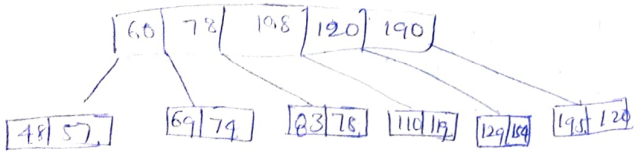


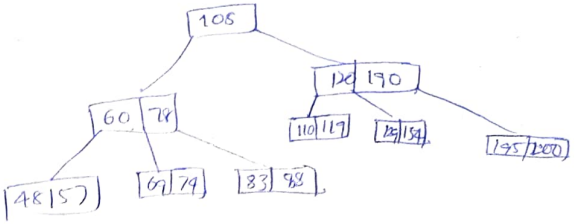
195 will be inserted in the right subtree of 120 after 190

insert it at the desired position



the node contains greater than the maximum number of element i.e 4 split it place median node





Deletion :-

1. Deletion the key and data from the leaves
2. If the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them
3. If the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

decomposition:-

let $R=\{A,B,C,D\}$

$F=\{A\to B, B\to C, C\to D, D\to A\}$

Let decomposed R into

$R_1=AB,\ R_2=BC,\ R_3=CD$

$A\cap R_1 = A\cap AB = A$

$[A]^+ = \{AB\}$

Sol:-

$Z=A$

$Z\cap R_1 = A\cap AB = A$

$[A^+] = ABCD$

$[A^+]\cap R_1 = ABCD \cap AB = AB$

$Z = A\cup AB = AB$

Update $Z=AB$

(ii) $Z=AB$

$Z\cap R_2 = AB\cap BC = B$

$Z\cap R_2 = AB$

$R=(A,B,C,D,B)$

$F=\{AB\to C, C\to E, B\to A, E\to A\}$

$R_1(B,C,D)\quad R_2(A,C,E)$

(i) $Z=AB$

For $Z\cap R_1 = AB\cap BCD = B$

$[B^+] = BD$

$[B^+]\cap R_1 = BD\cap BCD = BD$

Update $Z=AB\cup BD$

$Z=ABD$, Continue

(ii) $Z=ABD$

$Z\cap R_1 = ABD\cap BCD = DB$

$[BD]^+ = BD$

$[BD^+]\cap R_1 = BD\cap BCD = BD$

$Z=ABD\cup BD=ABD$

(iii) Continue

$Z=ABD$

$Z\cap R_2 = ABD\cap ACE = A$

$[A^+] = A$

$[A]^+\cap R_2 = A\cap ACE = A$

$Z=ABD\cup A = ABD$

Functional dependency:-

A functional dependency is a constraint between two sets of attributes from the database.

Def:- A Functional dependency, denoted by $X\to Y$ between two sets of attributes X and Y that are subset of R specifies a constraints on the possible tuples that can form a relation state r of R.

* If the constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X]=t_2[X]$, they must also have $t_1[Y]=t_2[Y]$
* This means that the values of the Y component of a tuple in r depend on, or are determined by, the value of the X component.
* The abbreviation for Functional dependency is FD or f.d. The set of attributes X is called the "left-hand side" of the FD, and Y is called the right hand side
* X-functionally determines Y in a relation schema R if, and only if two tuples of r(R) agree on their X-value, Must necessary agree on their Y-axis.
* X is a candidate key of R- this implies that $X\to Y$ for any subset of attributes Y of R.

Closure :-

Closure of an attribute 'x' is the set of all attributes that are functional dependencies on x with respect to F. It is denoted by x^+. Which means what x can determine

Algorithm:-

let's see the algorithm to compute x^+

- Step-1: $x^+ = x$
- Step-2: repeat until x^+ does not changes
  - For each FD $Y \to Z$ in F
    - If $Y \subseteq x^+$ then $x^+ = x^+ \cup Z$

Example-1:-

Consider a relation $R(A,B,C,D,E,F)$

$F: E \to A, E \to D, A \to C, A \to D, AE \to F, BG \to K$

Find a closure of E or E^+

Sol:-

$E^+ = E$

$= EDA$

$= ACDE$

$= ACDEF$ [For $AG \to B$ don't add k $AG \notin D^+$]

Ex-2:-

Let the relation $R(A,B,C,D,E,F)$

$F: B \to C, ABC \to AD, D \to E, CF \to B.$

Find the closure of B.

Sol:-

$B^+ = B$

$= BC$

$= BCAD$

$= ABCD$

$= ABCDE$ [$CF \to B$ don't

Ex-3:-

$R(A,B,C,D,E)$ AND

$F: A \to B, B \to C, C \to D, A \to E$

Find the closure of F

Sol:-

$A^+ = \{A\}$

$= \{AB\}$

$= \{ABC\}$

$= \{ABCD\}$

$= \{ABCDE\}$

$B^+ = \{B\}$

$= \{BC\}$

$= \{BCD\}$

$=$

$C^+ = \{C\}$

$= \{C,D\}$

$F^+ = \{A \to A, A \to B, A \to C, A \to D, A \to E, B \to B, B \to C, B \to D, C \to C, C \to D\}$

Equivalent closure of functional dependency:-

A set of functional dependencies (FD) F is said to covers another set of functional dependiencies E if every FD in E is also in F closure; that is if every dependency in E can be inferred from F.

We can say E is covered by F. Two sets of functional dependencies E and F are equivalent if $E^+ = F^+$ that is F is equivalent to F if E covers F and F covers E.

To determine whether F covers E we calculate x^+ with respect to F for each FD $x \to y$ in E and then check whether x^+ includes the attributes y.

Ex:-

$R = \{A,B,C,D,E,F\}$

$F_1 = \{A \to BC, B \to CDE, AE \to F\}$

$F_2 = \{A \to BCF, B \to DE, E \to AB\}$

Check whether F_1 and F_2 are equivalent or not.

~~Ex:-~~

Sol:-

To Check F_1 covers F_2

$A = \{A\}$

$= \{ABC\} = \{ABCDE\}$

$= \{ABCDEF\}$ — contains B, C, F

$B^+ = \{B\}$

$= \{BCDE\}$ contain D, E

$= \{\}$

$E^+ = \{E\}$

$= \{E\}$ doesn't covers A, B

So, F_1 does not covers F_2

Hence F_1 and F_2 are not equivalent.

Ex:-2:-

$R = \{A, C, D, E, H\}$

$F_1 = \{A \to C, AC \to D, E \to AD, E \to H\}$

$F_2 = \{A \to CD, E \to AH\}$

Check whether F_1 and F_2 are equivalent or not?

Sol:-

To check F_1 cover F_2:

$A^+ = \{A\}$

$= \{AC\}$

$= \{ACD\}$ contains ED

$E^+ = \{E\}$

$= \{EADH\}$ contain AH

To check F_2 cover F_1:

$A^+ = \{A\}$

$= \{ACD\}$ contain C

$[AC]^+ = \{AC\}$

$= \{ACD\}$ contain D

$E^+ = \{E\}$

$= \{EAH\}$

$= \{ECDAH\}$ contains A, D, H

So. F_2 covers F_1.

Hence F_1 and F_2 are equivalent.

Minimal Cover of FD:-

A minimal cover of set of functional dependencies E is a minimal set of dependencies that is equivalent to E.

A set of FD F to be minimal if it satisfies the following conditions:

1) Every dependency in F has a single attribute for its right hand side

2) We cannot replace any dependency $X \to A$ in F with a dependency $Y \to A$, 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 still have a set of dependencies that are equivalent to F.

Canonical cover is called minimal cover which is called "the minimum set of FD's".

Ex:-

Given FD's are follows

$A \to BC$

$B \to C$

$A \to B$

$AB \to C$

Sol:-

Step-1:- convert RHS Attribute into singleton attribute

$A \to B$

$A \to C$

$B \to C$

$A \to B$

$AB \to C$

Step-2:- Remove the Extra LHS attribute

$A \to B$

$A \to C$

$B \to C$

$A \to B$

$AB \to C$

$A^+ = \{A, B, C\}$ - contains B

$B^+ = \{B, C\}$ - doesn't contains B

$\therefore A \to C$

$A \to B$

$A \to X$

$B \to C$

$A \to B$

$A \to C$

Rem

Step-3:- Remove the redundant FDs and apply also (remove transitive dependency)

$A \to B$
$A \to C$
$B \to C$
$A \to B$
$A \to C$

$\therefore$ $A \to B$, $B \to C$, $A \to C$ $\therefore$ $A \to B$, $B \to C$

EX-2:-

$A \to C, AC \to D, E \to H, E \to AD$

step-1:- Remove the extra LHS attribute convert the RHS attribute into single attribute

$A \to C$
$AC \to D$
$E \to H$
$E \to A$
$E \to D$

step-2:- Remove the extra LHS attribute

$A \to C$
$AC \to D$
$E \to H$
$E \to A$
$E \to D$

$A^+ = \{A, C\}$ Contain 'c'

$C^+ = \{C\}$ doesnt contain

$\therefore A \to D$

Step-3:- Remove the redundants of FDs (remove transitive dependency)

$A \to C$
$A \to D$
$E \to H$
$E \to A$
$E \to D$

EX-3:-

$AB \to C, D \to E, AB \to C, E \to C$

step-1:- convert the RHS attribute into single attributes

$AB \to C$
$D \to E$
$AB \to E$
$E \to C$

step-2:- Remove the extra LHS attribute

Decomposition:-

begin;

for each $X \to Y$ and with $R(R_1, R_2, \dots R_n)$

{

Let $Z = X$

while there are changes in Z

{

from $i=1$ to n

$Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$ wrt to F;

}

y is a proper subset of Z, current fd is presesved

else decomposition not depency preserving

}

this is a dependency preserving decomposition:

end;

Ex:- $R\{A, B, C, D, E\}$

$F = \{A \to BD, B \to E\}$

Decomposition

$R_1\{A, B, C\}$ $R_2\{A, D\}$ $R_3\{B, D, E\}$

is this dependency preserving decomposition.

Sol:- $Z = A$

$Z \cap R_1 = A \cap ABC = A$

$[A]^+ = \{A, B, D, E\}$, $[A^+] \cap R_1 = ABDE \cap ABC = AB$

$Z \cup [AB] = A \cup AB = AB$

$Z = AB$

$Z = AB$

$Z \cap R_2 = AB \cap AD = A$

$[A^+] = \{A, B, D, E\}$

$[A^+] \cap R_2 = ABDE \cap AD = AD$

$Z = AB \cup AD = ABD$

$Z = AB$

$Z \cap R_3 = AB \cap BDE = B$

$[B^+] = BE$

$[B^+] \cap R_3 = BE \cap BDE =$

$Z \cup R_3 = AB \cup BE = AB$

(2) $Z = B$

$Z \cap R_1 = B \cap ABC = B$

$[B^+] = \{BE\}$

$B^+ \cap R_1 = BE \cap \overset{AB}{D} = B$

$Z = B \cup B = B$ still same

$Z = B$ still same

(11) $Z = B$

$Z \cap R_2 = B \cap AD =$ empty set

$Z \cap R_3 = B \cap BDE = B$

$[B^+] = \{BE\}$

$(B^+] \cap R_3 = BE \cap BDE = BE$

$Z = B \cup BE = BE$

thus $B \rightarrow E$ presesred

$R = (city, street, zip)$

$F = (CS \rightarrow Z, Z \rightarrow C$

city street zip