DEPARTMENT OF COMMERCE (CA)
DATABASE MANAGEMENT SYSTEM (Semester-III)
II B.COM(CA) Sub Code-18BCA32C
UNIT - III

Embedded SQL: Introduction – Operations not involving cursors, involving cursors – Dynamic statements, Query by Example – Retrieval operations, Built-in Functions, update operations - QBE Dictionary. Normalization: Functional dependency, First, Second, Third normal forms, Relations with more than one candidate key, Good and bad decomposition.

Embedded SQL

SQL provides a powerful declarative query language; writing queries in SQL are typically much easier than is coding the same queries in a general-purpose programming language. To access a database from a general-purpose programming language is for the following two reasons.

1.Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exists queries that can be expressed in a language such as Pascal, C, COBOL or FORTRAN that cannot be expressed in SQL write queries, we can embed SQL within a more powerful language

2.Nondeclarative actions-such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface-cannot be done from within SQL.

A language in which SQL queries are embedded is referred to as host language, and the SQL structures permitted I the host language constitute embedded SQL.
Languages such as PL/I however are not well equipped to handle more that one record at a time. It is therefore necessary t provide some form of bridge between the two functional levels and embedded SQL provides such a bridge by means of a new type of object called a cursor.

Operations not involving cursors

The DML statements that do not need cursors are as follows:

- "Singleton SELECT"
- UPDATE
- INSERT
- DELETE

Singleton SELECT

We use the term "singleton SELECT " to mean statement for which the retrieved table contains at most one row.
Example: SELECT statement

UPDATE

This statement can be executed to have changes in the databases designed.
Example: UPDATE, statement of SQL.

INSERT

This statement is used to include new row or information.
Example: INSERT, statement of SQL.

DELETE

This is used to delete information from the database.
Example: DELETE, statement of SQL.

Operations involving cursors

Consider the case of a SELECT that selects a whole set of records, not just one. What is needed is a mechanism for accessing the records in the set one by one; and cursors provide such a mechanism. Explicitly defined cursors are constructs that enable the user to name an area of memory to hold a specific statement for access at a later time.

The programmer to process a multiple-row active set one record at a time defines explicit cursors. The following are steps for using explicitly defined cursors within PL/SQL.

1. Declare the cursor

* Name the cursor
* Each cursor associates a query with cursor

Syntax

Declare cursor-name is *select statement*

Example

Declare c_names is select branch_name from branch where branch_city='Brooklyn';

2. Open the cursor

Opening the cursor activates the query and identifies the active set. Open also initializes the cursor pointer to just before the first row of the active set.

Syntax

Open cursor-name;

3. Fetching the cursor

Getting data into the cursor is accompolished with the fetch command.The fetch command retrieves the rows in the cursor set one row at a time.

Syntax

```
Fetch cursor-name into record-list;
```

4. Closing the cursor

The close statement closes or deactivates the previously opened cursor and makes the active set undefined oracle will implicitly close a cursor when the user's program or session is terminated. After a cursor is closed ,we cannot perform any operation on it.

Syntax

```
Close cursor-name;
```

Attributes involved in cursors

- %ISOPEN returns TRUE if the cursor is already OPEN
- %FOUND returns TRUE if the last FETCH returned a row, and returns FALSE if the last FETCH failed to return a row.
- %NOTFOUND is the logical opposite of %FOUND.
- %ROWCOUNT yields the number of rows fetched.

Example to illustrate cursor

1)Declare

```
        Cursor c4 is select salary,job from emp where job='CLERK';
    Begin
        if  c4%isopen then
        dbms.output.put_line('This message will not be displayed');
        else
            open c4;
            dbms.output.put_line('Cursor not found');
        end if;
            close c4;
    end;
```

2) The procedure to update students information by finding the total and average.

```
Declare
    st stu%rowtype;
    cursor c1 is select * from stu;
Begin
    Open c1;
    loop;
        fetch c1 into st;
        exit when c1%notfound;
        st.tot1l:=st.m1+st.m2+st.m3;
        st.average:=st.total/3;
```

```
                if st.m1>=50 and st.m2>=50 and st.m3>=50 then
                        st.result:='PASS';
                else
                        st.result:='FAIL';
                end if;
                        update stu set total=st.total,average=st.average,result=st.result
 where regno=st.regno;
        end loop;
        commit;
end;
```

Dynamic Statements

Embedded SQL provides certain features to facilitate the writing of on-line application programs that is programs to support on-line access to the database from an end-user at the terminal. Steps involved are

1.accept a command from the terminal
2.analyze the command
3.issue appropriate SQL statements
4.return a message and/or results to the terminal

The precompiler is a compiler for the SQL language. Suppose the application programs have written a program P that includes some embedded SQL statements.

Pre-compilation proceeds as follows.

- The precompiler scans the source program P and locates the embedded SQL statements.
- For each statement it finds the precompiler decides on a strategy for implementing that statements in terms of RSI operations. This process is referred to as optimization
- The precompiler replaces each of the original embedded SQL statements by an ordinary PL/I statement

The dynamic SQL component of SQL-92 allows programs to construct and submit SQL queries at run-time. In case of embedded SQL, each statement must be completely present at compile time, and are compiled by the embedded SQL preprocessor.
Using dynamic SQL, programs can create SQL queries as strings at run-time (based on i/p from the user) and can either have them executed immediately, or have them prepared for subsequent use.
The two principal dynamic statements are PREPARE and EXECUTE.

```
DCL SQLSOURCE   CHAR (256);

SQLSOUCE   ='DELETE   FROM   BRANCH   WHERE
BRANCH_CITY='PERRYRIDGE';
$PREPARE   SQLOBJ   FROM   SQLSOURCE:
$EXECUTE SQLOBJ:
```

The PREPARE statement passes the SQLSOURCE string to the RDS precompiler which goes through its normal process of parsing, optimization, code generation and builds a machine language versions of the statement called SQLOBJ.EXECUTE statement causes this machine language routine to be executed and thus causes the actual deletions to occur.
Once PREPARE, a given dynamically generated SQL statement can be EXECUTED many times. The generated statement can be replaced by another by issuing PREPARE again with the same target and a different source.

QUERY-BY-EXAMPLE

Query-by-example (QBE) is the name of both a data-manipulation language and the database system that included this language. The QBE database system was developed at IBM T.J.Watson Research center in the early 1970s.Today,some-database systems for personal computers support variants of QBE languages. It has two distinctive features:
1.Unlike most query languages and programming languages, QBE has a two-dimensional syntax: Queries look like tables. A query in one-dimensional language can be written in a one line. A two-dimensional language requires two dimensions for its expression.
2.QBE queries are expressed "by example". Instead of giving a procedure for obtaining the desired answer, the user gives an example of what is desired. The system generalizes this example to compute the answer to the query.

We express queries in QBE using skeleton tables. These tables show the relation schema as shown below.

Example the representation of branch relation

	Branch Branch name	Branch city	assets

Retreival operations

Queries on One relation

Examples:

1:Find all loan numbers at the Perryridge branch

Loan	Branch-name	Loan-number	Amount
	Perryridge	P._x	

The proceeding query causes the system to look for tuples in loan that have "perryridge" as the value for the branch-name attribute. For each such tuple the value of the loan-number

attribute is assigned to the variable x. The value of the variable x is “printed”, because the command P. appears in the loan-number coloumn next to the variable x.QBE assumes that a blank position in a row contains unique variable.As a result,if a variable does not appear more than once in a query,it may be omitted.

Thus the previous query can be re-written as

Loan	branch-name	loan-number	amount
	Perryridge	P.	

QBE performs duplicate elimination automatically.*To suppress the duplicate* elimination,we insert the command ALL. After the P. command:

Loan	branch-name	loan-number	amount
	Perryridge	P.ALL	

To display the entire loan relation ,we can create a single row consisting of P. in every field.

Loan	branch-name	loan-number	amount
	P.		

QBE allows queries that involve arithmetic comparisons

Example

1.Find the loan numbers of all loans with a loan amount of more than $700.

Loan	Branch-name	Loan-no.	Amount
			P.>700

The arithmetic operations that QBE supports are =,<,≤,≥ and ¬

2.Find the names of all branches that are not located in Brooklyn.

Branch	Branch-name	Branch-city	Assets
	P.	¬Brooklyn	

3.Find the loan-no. of all loans made jointly to Smith and Jones.

Borrower	Customer-name	Loan-no.
	‘Smith’	P._x
	‘Jones’	_x

4.Find the loan numbers of all loans made to smith ,to Jones or to both jointly.

Borrower	customer-name	loan-no.
‘Smith’	P._x	
‘Jones’	P._y	

5.Find all customers who live in the same city as Jones.

Customer	Customer-name	Customer-street	Customer-city
	P._x		_y
	Jones		_y

Queries on several relations

QBE allows queries that span several different relations. The connections among the various relations are achieved through variables that force certain tuples to have the same value on certain attributes.

Example

1.Find the names of all customers who have a loan from the ‘perryridge’ branch..

loan	branch_name	loan_no.	amount
	perryridge	_x	

borrower	cust_name	loan_no.
	P._x	_x

2.Find the names of all customers who have both an account and a loan at the bank.

Depositor	customer-name	account-no.
	P._x	

Borrower	customer-name	account-no.
	_x	

3.Find the names of all customers who have an account at the bank ,but who have a loan from the bank.

Depositor	customer-name	account-no.
	P._x	

Borrower	customer-name	loan-no.
		_x

4.Find all customers who have atleast two account.

Depositor	customer-name	account-no.
	P._x	_y
	x	¬*y*

<u>The condition box</u>

It is not convenient to express all the constraints on the domain variables within the skeleton tables. To overcome this QBE includes a condition box feature that allows the expression of general constraints over any of the domain variables.

Example:

1:Find all customers who are not named ‘Jones’ and who atleast two account.

Depositor	customer-name	account-no.
	P._x	_y
	x	¬*y*

Conditions
-*Y*>_z

2.Find all account-no. with a balance between $1300 and $1500 ,we write

acc-no.	branch-name	acc-no.	balance
		P.	_x

Conditions
_x.≥1300 _x≤1500

3.Find all branches that have assests greater than those of atleast one branch loacated in ‘Brooklyn’.

Branch	branch-name	branch-city	assets
	P._x		_y
		Brooklyn	_x

Conditions
_*Y* >_z

Options available with condition Box

1.QBE allows complex arithmetic expressions to appear in a condition box.
Example:
Find all branches that have assets that are atleast twice as large as the assets of one of the branches located in Brooklyn.

Branch	branch-name	branch-city	assets
	P._x		_y
		Brooklyn	_x

2.QBE allows logical expressions to appear in condition box.Operators used are and(&),or(|)

Example

Find all account numbers with a balance between $1300 and $2000 but not exactly $1500.

Account	branch-name	account-no.	balance
		P.	_x

Conditions
_x=(≥1300 and ≤2000 and ┐1500)

The result relation

If the result of a query includes attributes from several relation schemas, we need a mechanism to display the desired result in a single table.
Example
1.Find the customer-name, account-no. and balance for all accounts at the perryridge branch

In relational algebra
1.Join depositor and account relation
2.project customer-name, account-no. and balance

QBE related with this.

1.Create a skeleton table called result with attributes customer-name, account-no. and balance.

Account	branch-name	account-no.	Balance
	Perryridge	_y	_z

Depositor	customer-name	account-no.
	_x	_y

Result	customer-name	account-no.	Balance
P.	_x	_y	_z

Ordering of the display of tuples

By using the command AO. And DO. we can order the contents.

Example

1.List all customers in descending alphabetical order.

Depositor	customer-name	account-no.
	P.DO.	

Aggregate functions[Built-in functions]

QBE includes the aggregate operators AVG, MAX, MIN, SUM and CNT.we must postfix these operators with ALL. to create a multiset on which the aggregate operation is evaluated.

Example

1.Find the total balance of all the account maintained at the perryridge branch.

Account	branch-name	account-no.	balance
	Perryridge		P.SUM ALL.

2.Find the total no. of customers who have an account at the bank.

Depositor	customer-name	account-no.
	P.CNT.UNQ.ALL.	

3.Find the name,street and city of all customers who have more than one account at the bank.

Customer	cust-name	cust-street	cust-city
P.	_x		

Depositor	Cust-name	Account-No.
	G._x	CNT.ALL._y

Conditions
CNT.ALL._y > 1

Update operations/Modification of the database

This section deals with the options how to add, remove or change information using QBE.

Deletion

Deletion of tuples from a relation is expressed in much the same way as a query. The major difference is the use of D. in the place of P..In QBE we can delete whole tuples, as well as values in selected coloumns. To delete information in only some of the columns, null values, specified by-are inserted.

D. Operates on only one relation. To delete tuples from several relations, we must use one D. operator for each relation.

*Delete customer smith

customer	cust_name	cust_street	cust_city
D.	Smith		

*Delete the branch-city value of the branch whose name is “Perryridge”.

Branch	branch-name	branch-city	asstes
	Perryridge	D.	

*Delete all loans with a loan amount between $1300 and $1500

Loan	Branch-name	loan-no.	amount
D.		_y	_x

Borrower	cust_name	loan_no.
D.		_y

Condition
_x=(>=1300 and <= 1500)

*Delete all accounts at all branches located in Brooklyn.

Account	branch_name	account_no.	balance
D.	_x	_y	

Depositor	cust_name	acc_no.
D.		_y

branch	branch_name	branch_city	assets
	_x	Brooklyn	

Insertion

We do the insertion by placing the I. Operator in the query expression.The attribute values for inserted tuplles must be members of the attributes domain

Example

*To insert into the branch relation information about a new branch with name "Capital" and city "Queens",but with a null asset value,we write

branch	branch_name	branch_city	assets
I.	Capital	Queens	

*To insert the account A-9732 at the Perryridge branch has a balance of $700.

Account	branch-name	account_no.	balance
I.	Perryridge	A-9732	700

Updates

If we want to changeone value in a tuple withput changing all values in the tuple we use the update facility and the operartor used is U. .QBE allows users to update the primary key fields.

- Update the asset value of the Perryridge branch to $10,000,000

Branch	branch-name	branch-city	assets
	Perryridge		U. 100000000

The query updates the assets of the Perryrigde branch to $10,000,000 regardless of the old values.If we want to update a value using the previous vaulue ,we must express a request using two rows:One specifying the old tuples that need to be updated,and the other indicating the new updated tuples to be inserted in the database

- The interesty payments are being made,and all branches are to be increased by 5%.

Account	branch-name	account-no.	balance
U.			_x * 1.05
			_x.

QBE Dictionary

QBE has a built-in dictionary that is represented to the user as a collection of tables. The dictionary includes for example, a TABLE and a DOMAIN table, giving details of all tables and all domains currently known to the system. The dictionary tables can be interrogated using the ordinary retrieval operations of the DML.

Retrieval of table-names

Get the names of all tables known to the system.

P.			

Instead of having to build a skeleton for the TABLE table and entering “P.” in the NAME column of that skeleton, the user can formulate this query by simply entering the “P.” in the table-name position of the blank table.

Retrieval of column-name for a given table

Get names of all columns in table S.

S	P.	

User enters the table-name (S) followed by “P.” against the row of (blank) column-names.

Creation of a new table

1.Create table branch

I. branch I.	Branch name	branch city	branch street

The first I. Creates a dictionary entry for table branch; the 2nd I. Creates dictionary entries for the four columns of the table branch. Also the information for each column must be specified .The information includes the name of the underlying domain; the data-type of the domain; if that domain is not already known to QBE.

Dropping a table

Drop table branch.

A table can be dropped only if it is currently empty.

1) Delete all branch details

branch	branch name	branch city	branch street
D.			

2) Drop the table

D. Branch	branch name	branch city	branch street

Expanding a table

Add a asset coloumn to the table branch.

QBE does not directly support the dynamic addition of a new column to an existing table is currently empty.

So the following steps should be followed.

1) Define a new table the same shape as the existing table plus the new column.
2) Load the new table from the old using a multiple-record insert.
3) Delete all data from the old table.
4) Drop the old table.
5) Change the name of the new table to that of the old table.

Normalization

Introduction

Normalization theory is build around the concept of normal forms. A relation is said to be in a particular normal form if it satisfies a certain specified set of constraints. For example, a relation is said to be in first normal form if and only if it satisfies the constraint that it contains atomic values only. Various normal forms are First Normal Form, Second Normal Form, Third Normal Form, DKNF, and BCNF etc. Concept of normalization arises in the case to design a relational-database without unnecessary redundancy, easy way of retrieval etc…So if we want to design such a database we go for normalization.

For the description of normalization, we shall consider the supplier-and-parts database. The database or relation is as follows:

PART---P

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

SP------

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

FIG:1

Functional Dependency

Definition:

Given a relation R, attribute Y of R is functionally dependent on attribute X of R if and only if each X-value in R has associated with it precisely one Y-value in R.

In the supplier-and-parts database the attributes SNAME, STATUS and CITY of a relation S are each functionally dependent on attribute S#. For a particular value for S# there exists precisely one corresponding value for each of SNAME, STATUS and CITY.

S.S# → S.SNAME
S.S# → S.STATUS
S.S# →S.CITY

Or we can say represent as

S.S#→S. (SNAME, STATUS, CITY)

The statement S.S#→S.CITY is read as "attribute S.CITY is functionally dependent on attribute S.S#", or "attribute S.S# functionally determines attribute S.CITY".

Alternate definition for functional dependence

Given a relation R, attribute Y of R is functionally dependent on attribute X of R if and only if, whenever two tuples of R agree on their X-value, they also agree on their Y-value.

S#	P#	Qty	Status
S1	P1	300	20
S1	P2	200	20
S1	P3	400	20
S1	P4	100	20

Fig: Partial tabulation of relation SP'.

For example in this relation SP'

SP'.S#→SP'.STATUS

A functional dependence is a special form of integrity constraint. For example, if a relation S satisfies the FD S.S#→S.CITY then we say that every legal extension of that relation satisfies that constraint.
It is convenient to represent the FDs in a given set of relations by means of a functional dependency diagram.

Example:

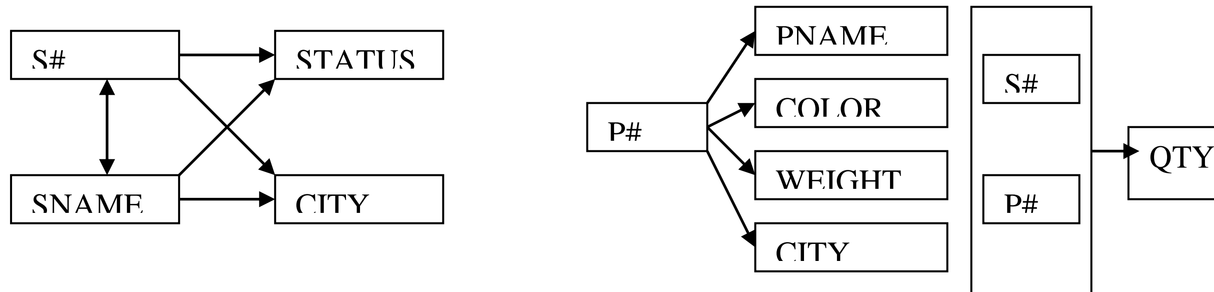


Fig: Functional dependencies in relations S, P, SP.

Various Normal Forms

Brief description of Normal forms

First Normal Form

- Eliminates repetition of data that is converts each data value to its atomic form

- No two rows should be identical
- Each table entry should be single valued
- Every table has a primary key, which is a unique label or identifier for each row

<u>Second Normal Form</u>

- Requires taking out data that is only dependent on a part of the key
- Each non-key attribute is functionally dependent on the entire key

<u>Third Normal form</u>

- Involves getting rid of anything in the tables that does not depend solely on the primary key
- 3NF is sometimes characterized as “the key, the whole key, and nothing but the key”

FIRST NORMAL FORM

Definition:

<u>A relation R is in first normal form(1NF) if and only if all underlying domain contain atomic values only.</u>

A relation that is only in first normal form has a structure that is undesirable for a number of reasons.

For example:

Let us assume that information concerning suppliers and shipments, rather than being split into two separate relations (S and SP) is combined into a single relation and let the name be FIRST with fields (S#, STATUS, CITY, P#, QTY).

Where S# represents the supplier number, STATUS represents the supply details, CITY represents the city where the supply has been made P# represents the Part number, QTY represents the quantity of supply.

Here the constraint is STATUS is functionally dependent on CITY. That is the meaning of this constraint is that a supplier’s status is determined by the corresponding location: e.g., all LONDON suppliers must have a status of 20.Also we ignore the attribute SNAME for simplicity The primary key of FIRST is the combination of (S#, P#). The following is the functional dependency diagram for this relation

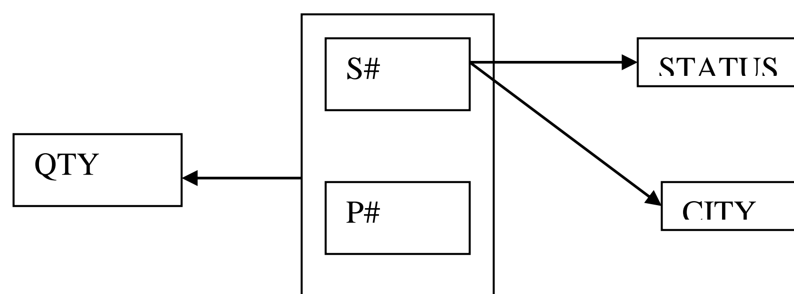


Fig: Functional dependencies in the relation FIRST

In the diagram

i) STATUS and CITY are not functionally dependent on the primary key.
ii) STATUS and CITY are not mutually dependent.
Certain difficulties of the FIRST relation occurs while UPDATION.They are explained as

<u>Insert:</u> We cannot enter the fact that a particular supplier is located in a particular city until that supplier supplies at least one part. The following is the tabulation of FIRST.

S#	STATUS	CITY	P#	QTY
S1	20	London	P1	300
S1	20	London	P2	200
S1	202	London	P3	400
S1	20	London	P4	200
S1	20	London	P5	100
S1	20	London	P6	100
S2	10	Paris	P1	300
S2	10	Paris	P2	400
S3	10	Paris	P2	200
S4	20	London	P2	200
S4	20	London	P4	300
S4	20	London	P5	400

Table: FIRST NORMAL FORM

The FIRST relation does not show that supplier S% is located in ATHENS. Because until S5 supplies some part, we have not appropriate primary key value.
<u>Deletion:</u> If we delete the only FIRST tuple for a particular supplier, we destroy not only the shipment connecting that supplier to some part but also the information that the supplier is located in a particular city.
For example if we delete the FIRST tuple with S# value S# and P# value P2, we lose the information that S3 is located in Paris.

<u>Updation:</u> the city value for a given supplier appears in FIRST many times, this redundancy causes update problems.

For example, if supplier S1 moves from London to Amsterdam then the two difficulties occurs. They are
Searching the FIRST relation to find every tuple connecting S1 and London and this produces an inconsistent result. The solution to these problems is to replace the relation FIRST by the two relations SECOND (S#, STATUS, CITY) and SP (S#, P#, QTY). The functional dependency diagrams for these two relations are as shown here.

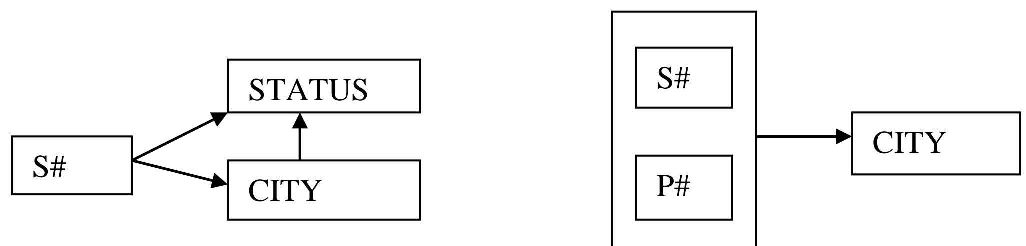


Fig: Functional dependencies in the relation SECOND and SP.

The following tables shows the sample tabulations corresponding to the data values of FIG:1 except the information for supplier S5 has been included in SECOND and not in SP.

SECOND NORMAL FORM

S#	Status	City
S1	20	London
S2	10	Paris
S3	10	Paris
S4	20	London
S5	30	Athens

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Fig: Sample tabulations of SECOND and SP.

After building the tables as shown we overcome the difficulties of FIRST relation. Now we can easily do the operations on the tables. This is about first normal form.

SECOND NORMAL FORM:

DEFINITION: A relation R is in second normal form (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key.

Relations SECOND and SP are both 2NF (the primary keys are S# and the combination (S#,P#), respectively). Relation FIRST is not in 2NF. A relation that is in first normal form and not in second can always be reduced to an equivalent collection of 2NF relations. The reduction consists of replacing the relations by suitable projections; the collections of these projections is equivalent to the original relations, in the sense that the original relation can always be recovered by taking the natural join of these projections, so no information is lost in the process. In other words, the process is reversible.

In our example: SECOND and SP relations are projections of FIRST, and FIRST is the natural join of SECOND and SP over S#.

The reduction of FIRST to the pair (SECOND, SP) is an example of nonloss decomposition. In general, given a relation R with possibly composite attributes A, B, C satisfying the FD R.A→ R.B, R can always be “nonloss-decomposed” into its projections R1 (A, B) and R2

(A, C).Since no information is lost in the reduction process, any information that can be derived from the original structure can also be derived from the new structure. The converse is not true, however: The new structure may contain information (such as the fact that S5 is located in Athens) that could not be represented in the original. In the sense the new structure is a slightly more faithful reflection of the real world.

The SECOND /SP structure still causes problems, however. Relation SP is satisfactory; as a matter of fact, relation SP is now in the normal form, and we shall ignore it for the reminder of this section. Relation SECOND, on the other hand, still suffers from a lack of mutual independence among its nonkey attributes. The dependence diagram for SECOND is still more complex than a 3NF diagram. To be specific, the dependency of the STATUS on S#, thought it is functional, is transitive (via CITY): Each S# value determines a CITY value, and this in returns determines the STATUS value. This transitivity leads, once again, to difficulties over update operations. (We now concentrate on the association between cities and status values-ie.,on the functional dependency of STATUS on CITY .)

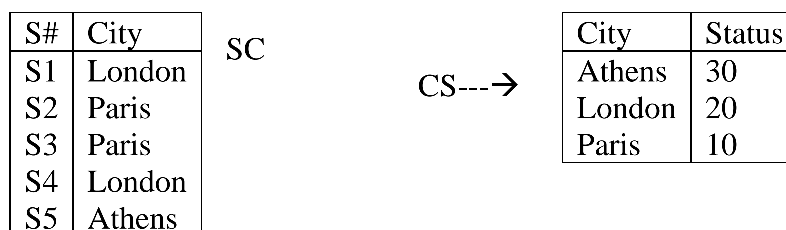
INSERTING: We cannot enter the fact that a particular city has a particular status value-for example, we cannot state that any supplier in Rome must have a status of 50-until we have some supplier located in that city. The reason is, again, that until such a supplier exists we have no appropriate primary key value.

DELETING: If we delete the only SECOND tuple for a particular city, we destroy not only the information for the supplier concerned but also the information that that the city has that particular status value. For example, if we delete the SECOND tuple for S5, we lose the information that the status for the Athens is 30.

UPDATING: The status value for a given city appears in SECOND many times. Thus, if we need to change the status value for London from 20 to 30 we are faced with either the problem of searching the SECOND relation to find every tuple for London or the possibility of producing an inconsistent result.

The solution to the problems is to replace the original relation (SECOND) by two projections SC(S#,CITY) and CS(CITY,STATUS).And the corresponding functional dependency diagram is shown here.

The tabulations corresponding to these is



SC

S#	City
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

CS---→

City	Status
Athens	30
London	20
Paris	10

Fig:2 Sample tabulations of SC and CS.

It should be clear that this new structure overcomes all the problems over update operations concerning the CITY-STATUS association.

Third Normal Form

Definition: A relation R is in third normal form (3NF) if and only if is in 2NF and every non-key attribute is non-transitively dependent on the primary key.

Relations SC and CS (shown in Fig:2)are both 3NF;relation SECOND (shown in page 20)is not in 3NF.A relation that is not in second normal form and not in third can always be reduced to an equivalent collection of 3NF relations.

Relations with more than one candidate key or BCNF (Boyce-codd normal form)

Definition:

A relation R is in BCNF if and only if every determinant is a candidate key.

The objective of BCNF is to handle a relation having two or more composite and overlapping candidate keys. Although BCNF is stronger than 3NF,it is still true that any relation can be decomposed in a non-less way into an equivalent collection of BCNF relations.

Relation FIRST consists of three determinants: S#, CITY and the combination (S#, P#). Among these (S#, P#) alone is a candidate key; hence FIRST is not in BCNF.

Relation SECOND is also not in BCNF because the determinant CITY is not a candidate key.

Relations SP, SC and CS are in BCNF because in each case the primary key is the only determinant in the relation.

Example: involving two disjoint (non-overlapping) candidate keys. Let us consider relation S (S#, SNAME, STATUS, CITY) .the relation S is BCNF.However, it is desirable to specify both keys in the definition of the relation:

a) To inform the DBMS, so that it may enforce the constraints implied by the two-way dependency between the two keys-namely, that corresponding to each supplier number there exists a unique supplier name, and conversely

b) To inform the users, since of course the uniqueness of the two attributes is an aspect of the semantics of the relation and is therefore of interest to people using it.

Example -where the candidate keys overlap.

Two candidate keys overlap if they involve two or more attributes each and have an attribute in common.

1) We suppose that the supplier names are unique, and we consider the relation SSP (S#, SNAME, P#, QTY). The keys are (S#, P#) and (SNAME, P#). This is relation is not in BCNF because we have two determinants# and SNAME, which are not keys for the relation (S# determines SNAME, and conversely). But the relation is in 3NF if we consider the definition----A relation R is in 3NF if and only if it is in 2NF and every non-key attribute is non-transitively dependent on the primary key. Here in this definition it does not require an attribute to be fully dependent on the primary key if it was itself a component of some other key in the relation, and so the fact that SNAME is not fully dependent on (S#, P#). But this fact leads to redundancy and hence to update problems in the relation SSP.If we go for updating the name of supplier S from Smith to Robinson leads either to search problems or to possibly inconsistent results. The solution to the problems as usual is to decompose the relation SSP into two projections, in this case SS (S#, SNAME) and SP (S#, P#, QTY) for SP (SNAME,P#,QTY).These projections are both BCNF.
2) Second example;
   Consider the relation SJT with attributes S(student),J(subject) and T(teacher).The meaning of an SJT tuple is that the specified student is taught the specified subject by the specified teacher. The semantic rules follow:
   1.Only one teacher teaches each student of thet subject
   2.Each teacher teaches only one subject
   3.Several tachers teach each subject.

   The sample tabulation of this relation is as follows
   SJT

S	J	T
Smith	Math	Prof.white
Smith	Physics	Prof.Green
Jones	Math	Prof.White
Jones	Physics	Prof.Brown

The functional dependencies of SJT are:
From the first semantic rule we have functional dependency of T on the composite attributes (S, J).
Form the second semantic rule we have a functional dependency of J on T.
From the third semantic rule it is understood that there is no functional dependency of T on J.
So the diagram is as follows

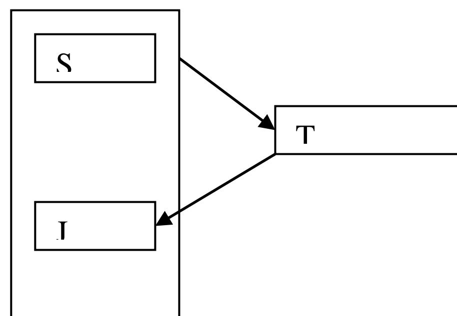


Fig: Functional dependencies in the relation SJT.

Here again we are having two overlapping candidate keys: the combination (S, J) and the combination (S, T). Once again the relation is 3NF and not BCNF; and once again the relation suffers from certain anomalies in connection with update operations. For example, if we wish to delete the information that Jones is studying physics, we cannot do so without at the same time losing information that professor Brown teaches physics.
The difficulties are caused by the fact that T is determinant but not a candidate key. Again we can get over the problem by replacing the original relation by two BCNF projections, in this case ST (S, T) and T, J (T, J).
Finally we say that the concept of BCNF eliminates certain problem cases that could occur under the old definition of 3NF.Moreover,BCNF is conceptually simpler than 3NF,in that it involves no reference to the concepts of primary key, transitive dependence and full dependence. The reference of candidate keys can also be replaced by a reference to the more fundamental notion of functional dependence. The reference to candidate keys can also be replaced by a reference to the more fundamental notion of functional dependence.

<u>Good and Bad decompositions</u>

During the reduction process it is frequently the case that a given relation can be decomposed in a variety of different ways. Consider the relation SECOND (S#, STATUS, CITY) with functional dependencies (FDs).

SECOND.S#→SECOND.CITY
SECOND.CITY→SECOND.STATUS

And therefore by transitivity

SECOND.S#→SECOND.STATUS

The representation of SECOND relation is

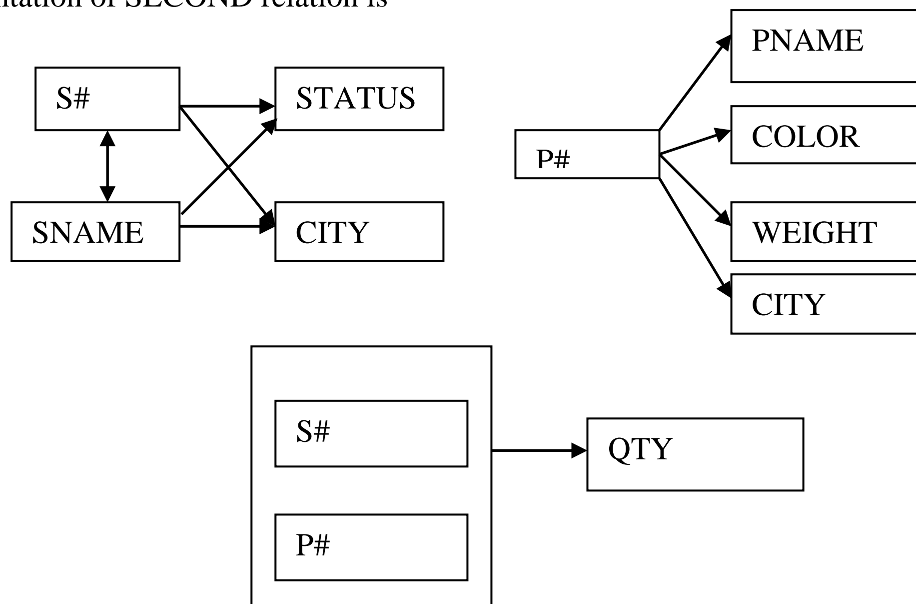


Fig: Functional dependencies in relations S, P, SP

The above diagram clearly states that the update problems encountered with SECOND could be overcome by replacing it by its decomposition into the two 3NF projections

SC (S#, CITY) and CS (CITY, STATUS)------------------→A

Let this composition be A.

An alternative decomposition is

SC (S#, CITY) and SS (S#, STATUS)---------------------------→B

Decomposition B is also nonloss, and the two projections are again BCNF.But decomposition B is less satisfactory than decomposition A.

For example, it is still not possible (in B) to insert the fact that a particular city has a particular status value unless supplier is located in that city. The explanation of this example is as follows:

In decomposition A the two projections are independent of each other, in the sense that updates can be made to either one without regard for the other; So joining them will not violate the FD constraints on SECOND.

In decomposition B updates to either of the two projections must be monitored to ensure that the FD SECOND.CITY→SECOND.STATUS is not violated. Thus projections SC and SS are not independent of each other.

A relation that cannot be decomposed into independent component is said to be atomic.

BOOKS FOR REFERENCE:

1. Database systems concepts by Abraham Silberschatz, Henry F. Korth.
2. An Introduction to Database System – C. Dsai.
3. An introduction to Database Systems (Seventh Edition) – C.J. Date

Prepared by Dr. N. SHANMUGAVADIVU