Infosys Technologies Limited

Education and Research Department

DB2

November, 1995

Document No.	Authorized by	Ver. Revision	Signature / Date
ER/CORP/CRS/DB01/002	Ganesh M. Baliga	1.2	

Modification Log

Ver. Revision	Date	Author(s)	Description
1.2	2/11/95	Achin Sahay	

TABLE OF CONTENTS

1

DB2 & DB2' Key Component

1.1. Objectives

This chapter introduces you to

- DB2
- DB2's attachment facilities
- DBRM
- Bind process
- DB2 objects
- Catalog tables
- Unit of work
- Referential integrity

1.2. Introduction

DB2 Is a Relational DBMS developed by IBM for computers running under MVS, its most advanced operating system for large computers. DB2 supports SQL (structured query language), which has been standardized by ANSI (American National Standards Institutes) and ISO (International Standards Organization) and has become the standard for all relational DBMSs.

DB2 co-operates with attaches to is the technical term-any of three MVS subsystems environments : IMS, CICS, and TSO. These subsystems cooperate with DB2 facilities to provide such services as data communications and control of transactions, which are group of database operations that must be coordinated to avoid the introduction of errors. CICS is a teleprocessing monitor, a program for controlling online transactions those that execute as they are entered from a terminal allowing users to interact with the computer. IMS/DB/DC is a well established nonrelational DBMS, which includes a teleprocessing monitor. TSO also contains a teleprocessing monitor that can be used by DB2. DB2 applications running under TSO may be online or batch.

CICS attachment facility ADB2 subcomponent that uses the MVS subsystem interface (SSI) and cross storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

IMS attachment facility A DB2 subcomponent that uses MVS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

TSO attachment facility A DB2 facility consisting of the DSN command processor and DB21. Applications that are not written for the CICS or IMS environments can run under the TSO attachment facility.

1.3. Database Request Module

The precompiler output include a database request module (DBRM) which contains SQL statements extracted from the source program. The SQL statements in a DBRM are those executable statements that must be bound before they can be executed. The DBRM is kept as a member of a partitioned data set (library) and is given the name of the program. It also contains a consistency token to distinguish it from other DBRMs derived from other versions of the program.

1.4. The Bind Process

The bind process establishes a relationship between an application program and its relational data. This step is necessary before you can execute your program. Currently, DB2 allows you two basic ways of binding a program : to a package, or directly to an application plan. If your application is to make use of remote units of work, then you must use packages.

During the precompilation process, the DB2 precompiler produces both modified source code and a database request module (DBRM) for each application program. The modified source code must be compiled and link edited the application program can be run. DBRMs must go through the bind process.

When determining the maximum size of a plan, several physical limitations must be considered, including the time required to bind the plan, the size of the EDM (environmental descriptor manager, which manages application plans and packages) pool, and fragmentation. There are no restrictions to the number of DBRMs that can be included in a plan. However, packages provide a more flexible method for handling large numbers of DBRMs within a plan. As a general rule, it is suggested that the EDM pool be at least 10 times the size of the largest DBD or plan, whichever is greater.

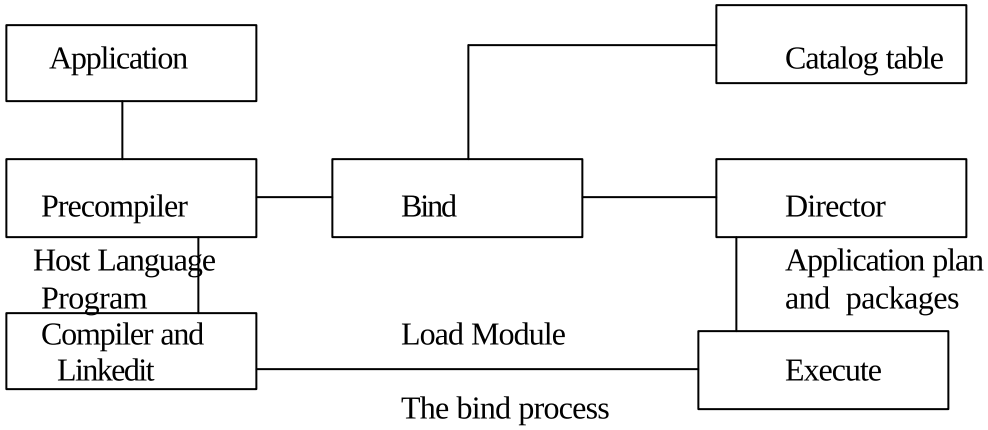
The BIND PACKAGE subcommand allows you to bind DBRMs individually. It gives you the ability to test different versions of an application without extensive rebinding. Package binding is also the only method for binding applications at remote sites.

Even when they are bound into packages, all programs must be designated in an application plan. BIND PLAN establishes the relationship between DB2 and all DBRMs in that plan. Plans can specify explicitly named DBRMs, packages, collections of packages, or a combination of these elements. The plan will contain information about the designated DBRMs and about the data the application program intends to use. It is stored in the DB2 system catalog.

In addition to building packages and plans, the bind process :
Validates the SQL statements using the DB2 catalog. During the bind process, DB2 checks your SQL statements for valid table, view and column names. Because the bind process occurs as a separate step before program execution, errors are detected and can be corrected before the program is executed.

Verifies that the process binding the program is authorized to perform the data accessing operations requested by your program’s SQL statements. When you issue BIND, you can specify an authorization ID as the owner of the plan or package. The owner can be any one of the authorization Ids of the process performing the bind. The bind process determines whether or not the owner of the plan or package is authorized to access the data the program requests.

Selects the access paths needed to access the DB2 data your program wants to process. In selecting an access path, DB2 considers indexes, table sizes, and other factors. DB2 considers all indexes available to access the data and decides which ones (if any) to use when selecting a path to the data.



The bind process

1.5. DB2 Objects

A DB2 object is something that can be defined using an SQL CREATE statement The objects are classified as follows :

Tables Indexes
Views
Table spaces
Storage groups
Data bases
Synonyms
Aliases

1.5. The Catalog

Each DB2 maintains a set of tables containing information about the data it manages. These tables are collectively known as the catalog. The catalog tables contain information about DB2 objects such as tables, views, and indexes.

With appropriate authorization, you can retrieve data from catalog tables by using SQL statements, just as you would with any other table. Each DB2 ensures that at all times its catalog contains accurate descriptions of the objects that the DB2 controls.

1.7. Unit of work

A unit of work is a logically distinct procedure containing one or more steps that change one or more pieces of data. If all the steps complete successfully, you want the data changes made to become permanent. But, if any of the steps fail to complete successfully, you want all modified data to be returned to the value before the procedure began.

For example, suppose two employees in the sample table DSN8230.EMP exchange offices. Their office phone numbers need to be exchanged in the PHONENO column.

You would use two UPDATE statements to make each phone number current. Both statements, taken together, are a unit of work. You want both statements to complete successfully ; if say, only one statement was successful, then you would want both phone numbers rolled back to their original value before attempting another update.

When a unit of work completes, all locks implicitly acquired by that unit of work after it begins are released, allowing a new unit of work to begin.

The amount of processing time used by a unit of work in your program determines the length of time DB2 prevents other users from accessing that locked data. Several programs trying to use the same data concurrently require each program's unit of work to be kept as short as possible in order to minimize the interference between the programs.

1.8. Referential integrity

The condition that exists when all intended references from data in one column of a table to data in another column of the same or a different table are valid. Maintaining referential integrity requires enforcing referential constraints on all LOAD, RECOVER, INSERT, UPDATE and DELETE operations.

1.9. Review Questions

- What are the MVS subsystems environment DB2 attaches to ?
- What services these subsystems provide co-operating with DB2 ?
- What is CICS attachment facility ?
- What is IMS attachment facility ?
- What IS TSO attachment facility ?
- What is a DBRM ?
- What is the BIND process ?
- What are the DB2 objects ?
- What are the catalog tables ?
- What is an unit of work ?
- What do you understand by referential integrity ?

2

DB2 System Architecture

2.1. Objectives

This chapter briefly discusses the concept of

- Threads
- AUTHID
- Address Spaces
- Table naming

2.2. Threads

A thread is a control structure used by DB2 to communicate with an application program. The thread is used to send request to DB2 to send data from DB2 to the program, and to communicate (through the SQLCA) the status of each SQL statement after it is executed. Every program must communicate with DB2 by means of a thread.

2.3. AUTHID

Allows you to specify the primary authorization ID of the owner of the new package. That ID is the name owning the package, and the name associated with all accounting and trace records produced by the package.

2.4. Address Spaces

Each DB2 subsystem consists of three or four tasks started from the operator console. As shown in the figure below. Each of these started tasks run in a portion of the CPU called an address space.

DBAS	SSAS	IRLM	DDF
Database functions	Logging	Locking	Distributing requests
Buffering	Attachment Coordination		
DSNDBM1	DSNMSTR	IRLMPROC	DSNDDF

The Database Services Address Space (DBAS), provides the facility for the manipulation of DB2 data structures. The default name for this address space is DSNDBM1. This component is responsible for the execution of SQL and the management of buffers, and contains the core logic of the DBMS. The DBAS consists of three components, each of which performs specific tasks : the Relational Data System, the Data Manager, and the Buffer Manager.

The System Services Address Space (SSAS), coordinates the attachments of DB2 to other subsystems (CICS, IMS/DC, or TSO). SSAS is also responsible for all logging activities.
DSNMSTR is the default name for this address space.

Intersystem Resource Lock Manager (IRLM) is responsible for the management of all DB2 locks. The default name of this address space is IRLMPROC.

Distributed Data Facility (DDF) is optional. This is required only when distributed database functionality is needed.

2.5 Using Three-Part Table and View Names

A three part table or view name consists of three identifiers separated by periods :

The first identifier is the location name for the object.
The second identifier is the owning authorization ID.
The third identifier is the actual table name.

For example, the name DALLAS.DSN8230.EMP could represent a table at the DALLAS location. The owning authorization ID is DSN8230., and the table name is EMP. The location name could be the name of your local subsystem, instead of a remote location.

Suppose that you want the name, employee number, and department ID of every employee whose last name ends in “son” in table EMP at location DALLAS. If you have the appropriate authority, you could run the following query :

```
SELECT LAST NAME, MIDINIT, FIRSTNME, EMPNO, WORKDEPT FROM
DALLAS.DSN8230.EMP
WHERE LASTNAME LIKE ‘%S ON’;
```

2.6. Review Questions

- What is a thread ?
- What is a AUTHID ?
- What are the address spaces ?
- What is a DBRM ?
- What is DBAS ?
- What is IRLM ?
- What is DDF ?

3

Object

3.1. Objectives

This chapter discusses in detail

- Physical objects
  - Database
    - DBD
  - Tablespace
    - Simple tablespace
    - Segmented tablespace
    - Partitioned tablespace
    - Storage groups
    - Tablespace parameters
  - Indexspace
- Logical objects
  - Table
    - Data types
    - Concept of NULL
    - Synonyms and Aliases
    - OMMENT ON
    - LABEL ON
- Index
- View
    - View merge
    - View materialization
- Catalog tables
- DB2 Directory

3.2. Objects

DB2 manages data through a system of logical and physical entities called objects. For example, tables, indexes and databases are objects. Objects that describes the data in the way users and developers think about it, are called logical objects ; for example, tables, and views are logical objects. Objects that refer to the way data are actually stored in the system, are called physical objects ; for example, database and tablespace are physical objects (which will be discussed in detail in the coming sections).

3.2.1. Physical objects

3.2.1.1. Database

Database is a collection of logically related objects. Stored data is split into disjoint databases.

A DB2 database is a set of table spaces and index spaces which are related in that the index spaces contain indexes on the tables in the table spaces. Databases are used primarily for administration : whenever a table space is created, it is assigned, explicity or implicitly to an existing database. That table space is then part of the database, along with any tables defined for the table space and any indexes defined for the tables. All these objects are under the control of any one that has been grated certain authority over the database. An agent with that authority can take appropriate administrative actions for the objects. Such actions include dropping existing objects in the database, creating new ones, and examining the data in the tables.

A database is defined using the CREATE DATABASE statement. One DB2 system can manage upto 65,279 separate database.

Although several application systems can operate on the same database and each program can access many databases transparently, there are number of advantages to defining a separate database for each subject area.

The advantages area,

- From administrative point of view, database can be controlled very easily as it is accessed by only one application.
- When any object is created, DB2 writes descriptive and control information in an area called Database Descriptor (DBD) and also locks DBD. When SQL statements are bound, then also DB2 locks DBD. So if many applications operate on the same database, a situation called lock contention may arise. The developer can avoid this problem by having one database for one application.

3.2.1.1.1. DBD

Database descriptor (DBD). An internal representation of DB2 database definition which reflects the data definition found in the DB2 catalog. The objects defined in a database descriptor are table spaces, tables, indexes, index spaces, and relationships.

3.2.1.2.Tablespace

A page set used to store the records of one or more tables. Space is dynamically extendable collection of pages.

A table space is a storage structure. Depending on its nature, a table space can hold one or more tables. All tables are kept in table spaces. There are three types of tablesapces.

1. **Segmented tablespace**
2. **Simple tablespace,**
3. **Partitioned tablespace**

3.2.1.3. Storage groups

Each space has an associated Storage Group.

Maintenance for the data sets of a storage structure can be left to DB2. If it is left to DB2, the storage structure has an associated storage group. The storage group is essentially a list of DASD volumes on which DB2 can allocate data sets for associated storage structures. The association between a storage structure and its storage group is made, explicity or implicitly, by the statement that creates the storage structure.

Spaces in a given database do not all have to have same storage group. All spaces sharing a storage group need not be from the same database. For convenience, defaults for spaces and storage groups are defined, so that a naïve user can ignore these aspects.

<u>Creation of a storage groups</u>

```
CREATE STOGROUP DASPJSTG
        VOLUMES     (VOL1,  VOL4)
        VCAT        VCATID
        PASSWORD    SESAME ;
```

<u>Example for creating a database :</u>

```
CREATE DATABASE ERDB
        STOGROUP ERSTG
        BUFFERPOOL  BPO
```

Storage group and bufferpool in the definition will be defaults if no storage group and bufferpool are specified during creation of tablespace.

CREATION OF A TABLESPACE

```
CREATE TABLESPACE PTSP IN DASPJDB USING DASPJSTG
        PRIQTY          1000
        SECQTY          4
        ERASE           NO
        LOCKSIZE        PAGE
        BUFFERPOOL      BP0
        CLOSE           NO
        PCTFREE         10
        FREEPAGE        20
        SEGSIZE         64
        DSETPASSE       SESAME ;
```

3.2.1.4. Tablespace parameters

PCTFREE OPTION

PCTFREE specifies the percentage of free space to be left free on each page.

PCTFREE free space allows for insertion of new records and for expansion of variable length fields.

PCTFREE parameters will be 0 for read only tablespace and indexspace.

FREEPAGE OPTION

FREEPAGE specifies the number of pages to be loaded between each page left free. It is useful for finding a near page for inserting a new record.

ESTIMATING FREE SPACE NEEDS

Properties to be considered during estimation

(1) Insertion of new rows,
(2) Adding / deleting a new column
(3) When a VARCHAR field is involved, change in the number of bytes.

Free Space Adjustments

Estimates of free space requirements need not be exact. The developer can monitor the number and frequency of relocated rows and the loss of clustering and adjust free space before re-organisation to achieve the desired re-organisation schedule.

- Add free space if re-organisation is needed more frequently than desired.

- Subtract unnecessary free space, if re-orgnisation is not needed for the scheduled re-organisation. Because free space reduces the number of rows that can fit in a page and thereby increases I/O.

Adding columns

While adding new columns, more space may be required. Additional freespace may be allocated by specifying it with the PCTFREE parameter in the ALTER TABLESPACE command. This can be done before ore after the values for the new column are entered. Done before – no row relocation. After – row relocation may be required (a row is never broken up between pages). The correct choice depends on table size and frequency of update. Large table with update values known : create space first. Small table, with infrequent update : keep adding, do re-organisation on schedule.

Change in length in VARCHAR field

The amount by which a VARCHAR field can expand can be estimated roughly. The possible amount of expansion of each row is the difference between the minimum and maximum description contained in the VARCHAR column. The probable amount of expansion depends on a number of factors related to update patterns. For example, the percentage of rows changed and the average size of the changes are some of the factors.

CLOSE OPTION

The CLOSE parameter is used during tablespace creation to tell DB2 How to handle opening and closing of tablespace's VSAM datasets. CLOSE = YES is the default and causes the tablespace to be opened each time a plan is executed and closed after execution. Closing also takes place when work is committed. CLOSE = NO will open the tablespace when it is first accessed, and will keep it open till DB2 is shut down or the tablespace is stopped.

BUFFERPOOL OPTION

There are 4 bufferpools BP0, BP1, BP2 and BP32 which DB2 provides. It can analyse usage patterns and decide what data to keep in buffers to minimize secondary storage access. The administrator has to allocate memory to these bufferpools. BP0 is compulsory as DB2 uses it for catalog tables, joins, utilises and sorts. BP32 is also compulsory if any table has rows longer than 4 k bytes (or if any joins result in rows longer than 32K bytes) thereby needing 32K byte pages. An access to a table having 32K byte pages is equivalent to 8 physical accesses and every effort should be made to keep the need for this as low as possible.

Other than BP0 and BP32 the administrator has several choices for the remaining bufferpools.

1. Allocate all tablespace in one pool and all indexspaces in another
2. One bufferpool for ordinary applications (tables as well as indexes) and another for critical ones.
3. Allocate all bufferpool space to BP0.

Under 1, the principle is that most applications will open a table and its associated index. Thus this approach will prevent the tables and indexes of an application from contending for bufferspace.

The idea in 2 is to avoid non-priority applications from blocking out buffer space from priority applications. This has drawbacks. When priority applications are not active, bufferpool space is wasted. Also as the number of priority applications builds up, they will contend with each other. Further the priority of an application changes over time and hence the assignment of applications to bufferpools needs to be managed.

The third approach leaves the problem of bufferpool management to DB2. Its algorithms will allocate bufferspace according to actual use. In general this is a good strategy because DB2's algorithm are time tested. Anotehr good reason for leaving this department to DB2 is that when 97.5% of a bufferpools' pages are in use, then DB2 automatically switches off many of its sophisticated buffer management techniques. The chance of this happening is significantly less with one large bufferpool than with a few smaller ones.

In general, large bufferpools are good performance wise. As the bufferpool fills up, sophisticated features are shutoff one by one. 50% full – buffered writing of updated pages to disk is shut off 90% synchronous prefetch (getting 32 pages at a time rather than 8) will be shut off 95% - one I/O is performed for each page. These can cause significant performance degradation.

LOCKSIZE

It is the size of the physical object to be locked during concurrent access to the database. There are three lock sizes. They are page, table and tablespace locks. Table locksize is available only in SEGMENTED tablespaces. The tablespace creator can leave the decision of the size of the lock to DB2 by specifying a LOCKSIZE of ANY. DB2 usually locks by PAGE. This is alos the default if nothing is specified. Choice of locking unit involves a concurrency/efficiency trade-off. Locking at lower levels of granularity allow high degree of concurrency, but involve more CPU time for checking the presence of locks.

When application plans are bound, DB2 chooses the locksize, types and duration of each lock depending on the LOCKSIZE specified in the CREATE TABLESPACE, type of SQL statement being executed and the BIND parameters chosen.

For read-only tables, tablespace locksize is best. For multiple concurrent update transactions against a table, page locksize is best from the viewpoint of concurrency. When ANY has been specified, if more than a certain specified number of page locks are taken against a table, then DB2 will release all page locks and lock the entire tablespace.

For tablespace that usually have moderate updates, but occasionally have heavy updates, PAGE locksize can be specified in the CREATE TABLESPACE. Any program that wishes to perform extensive updates would lock the tablespace before doing its updates.

ERASE OPTION

DB2 provides two methods for deleting the data when the tablespace is dropped. First, it will merely drop the database from the system, leaving the data in DASD until it is written over. For this, the option is 'ERASE NO'. The second delete option fills the data of the tablespace with zeros. The option for this is 'ERASE YES'.

SEGSIZE

SEGSIZE keyword in the CREATE TABLESPZE specifies the segment size in multiples of 4 pages ranging from 4 to 64 pages.

PRIQTY and SECQTY

It specifies the size of the VSAM datasets that DB2 will create for the tablespace. DB2 assigns the space to primary area and a secondary area which it uses when the primary area is full. The key for selecting the most efficient primary and secondary quantity sizes is determining a primary amount that is certain to hold the tablespace but not so large that it wastes storage.

3.2.1.5. Segmented tablespace

In this, tablespace is split up into segments of equal size. Segments are of 4 to 64 pages size in increments of 4 pages.

1. It can house one or more table ; there is no limit on the number of tables,
2. Each segment is dedicated to a table ; a table can occupy multiple segments,
3. All segments in a table space must be of same size,
4. A segment size of 64 or 32 pages maximises the benefits of prefetch,
5. To avoid wasting storage space for tables smaller than 64 pages, table and segment sizes should be approximately the same.

Because DB2 treats each segment separately when allocating free pages, FREEPAGE should be set with segment

Blocks , with each block containing information about a segment. Each segment block includes a pointer to the next segment block that applies to the same table, 4 bits of information are kept per segment. This encodes information such as whether there is enough space in the segment for inserting a row in sequence, for increasing a the length of a variable length row, etc, basically this information could help DB2 in avoiding the data page in many situations thus increasing efficiency.

The segmented tablespace allows for the mass delete algorithm which reads and updates only the space map table and avoids altogether the data pages (this however has the disadvantage that it cannot return the number of rows deleted). Mass delete is possible only when the delete statement has no predicate.

Segmented tablespaces also allow skipping of pages that do not hold date of a table. This is not possible in a non-segmented that contains multiple tables.

Freespace from dropped tables becomes immediately reusable. Concurrence control locks can be managed at the table level instead of the tablespace level.

Utilities for copying and recovering data operate at the level of a tablespace and not table level. Thus if related tables are put into one segmented tablespace then such operations can be done in one shot on them. This can of course be a disadvantage when the operation needs to be performed on only one of the tables.

The REORG utility which, among other things, reallocates free space and returns rows which are out of sequence to their proper place works better for multiple tables in a segmented tablespace than for multiple tables in a non-segmented tablespace. It will not recluster a a clustering index if more than one table one table occupies a non-segmented tablespace.

Opening a tablespace requires about 1.2K of virtual space. Thus if several tables are put in single tablespace then the same 1.2 K will serve to access all the tables in the tablespace.

3.2.1.6 Simple tablespace

Both simple and segmented types can provide storage space for one or more tables, but segmented tablespaces are better for managing multiple tables. Simple tablespace should almost never be used for managing multiple tables.

1. Simple tablespace can house one or more table; there is no limit on the number of table.
2. Rows from multiple tables can be interleaved on a page under the developers control and maintenance.

One advantage of such interleaving is, if table S and SP were assigned to the same table space, them it would be possible to store all the shipments for supplier S1 close to the stored record for supplier S1, all the shipments for supplier S2 close to the stored record for supplier S2, and so on. Queries such as "get details of supplier S1 and all corresponding shipments' – in particular, certain join queries can be handled very efficiently, since the number of I/O operation would be reduced. But neither the optimizer nor the REORG utility will understand such a clustering.

3.2.1.7 partitioned tablespace

A partitioned tablespace allows a table to be divided by rows into partitions that the system can manage separately. Each partition can be placed on different storage devices, and freespace can be allocated by partition. A partitioned tablespace must have a clustering index. DB2 allows up to 64 one GB partitions, each in its own VSAM dataset, partitions can be larger-up to 4 GB each, but the total tablespace size connot exceed 64 GB.

Partitioning and performance

Large table (more than 1 million rows) should generally be kept partitioned. Frequently accessed portions can be kept on fast media. DB2's utilities lock tablespace when they work. The utilities can work at the partition level. Thus the locks will be in force for a shorter duration.

As the number of indexes grows, the benefits of partitioning reduce as the effort in reorganising indexes will be prohibitive. The clustering index is kept partitioned and needs little maintenance. The first 40 bytes of a partitioning index identifies the partition number. All other indexes cover the entire tablespace. Re-organisation of a partition requires reorganiation of all of these large indexes and may cost almost equal to the cost of reorganising the entire database.

Creating Partitioning Tablespace

```
CREATE TABLESPACE ZIPTBS IN DAZIPDB USING STOGROUP
                        DAZPOSTG
    PRIQTY 7200
    SECQTY 0
    ERASE NO
    NUMPART44
    (PART 1 USING STOGROUP DAZPISTG
    PRIQTY14400
    SECQTY 720
    ERASE MP
        PCTREE 20
        FREEPAGE7,
        PART 2 USING STOGROUP DAZP2STG
        PRIQTY 14400
        SECQTY720
                EARASE NO
                PCTFREE15
                FREEPAGE 7)
PCTFREE 0


FREEPAGE 0
LOCKSIZE PAGE
BUFFERPOOL BP 0
CLOSE  NO;
```

Each partition has its own VSAM dataset.

3.2.1.8 Indexspace

A page set used to store the entries of one index. Indexes are also kept in storage structures. Unlike table spaces, each index space holds one and only one index. DB2 storage structures—table or index spaces—are managed by access method services.

1. There is exactly one index per indexspace. Table and its associated indexes must be stored in a single datebase.
2. Indexspace is created when the associated index is creatd.
3. Indexpages are 4 K pages. A portion of the index page can be locked.
4. Indexspace for a partitioned tablespace is considered to be partitioned. Individual partitions can be assigned to diiferent storage groups.
5. Index is a B-tree.

Create index

Only one index is allowed per indexspace. Thus CREATE INDEX also creates the

INDEXSPACE.

Index is created as follows.

```
CRETE UNIQUE INDEX DAPSJDB. SPJX
        ON BSPJ (SN,PN,JN)
                USING STOGROUP DASPJSTO
                        PRIQTY 5000
                        SECQTO 4
                        ERASE  NO
                CLUSTER
                SUBPAGES 4
                BUFFERPOOL BP 0
                CLOSE NO
        DSETPASS SESAME;
```

CLUSTER and SUBPAGE options cannot be used with an ALTER INDEX statement. USING, PCTFREE, FREE[AGE, BUFFERPOOL, CLOSE and DSETPASS
Partmeters are as in CREATE TABLESPACE, These can be changed with ALTER INDEX, clustering may be ASC (default) or DES.
An index node contains values from one or more of table's column and an RID to the corresponding row of the table. A table can contain only one clustering index.

Subpage locks

Index pages are 4K bytes, Index pages can be locked by subpage increment (1,2/2,1/4 1/8 and 1/16). A SUBPAGE value of 1 indicates that the entire page will be locked, 4 is the default which is used if no value is specified. The subpage value must be large enough to contain multiple key values, their RID's and some free space. Poor response may sometimes be because of index contention rather than table contention. In these cases response can be improved by increasing the value of the SUBPAGE parameter. This cannot be done by an ALTER and needs dropping and recreation of the index.

The index for the partitioned tablespace given above is crated as shown below.

```
CREATE INDEX ZIPINDEX ON ZIPTB (ZIPCODE)
    CLUSTER
        (PART 1 VALUES (500000)
            USING STOGROUP DAZ 01 STG
                PRIQTY 7200
                SECQTY 720
                PCTFREE 20

    PART 2 VALUVES (1000000)
        USING STOGROUP DAZ 02STG
            PRIQTY 7200
            SECQTY 720
            PCTFREE 15,

    PART 44 VALUES (22000000)
        USING STOGROUP DAZ 44 STG)
    PCTGREE 5
    FREEPAGE 0
    SUBPAGES 4

BUFFERPOOL BP 0
CLOSE NO;
```

3.2.2.Logical objects

3.2.2.1 Table

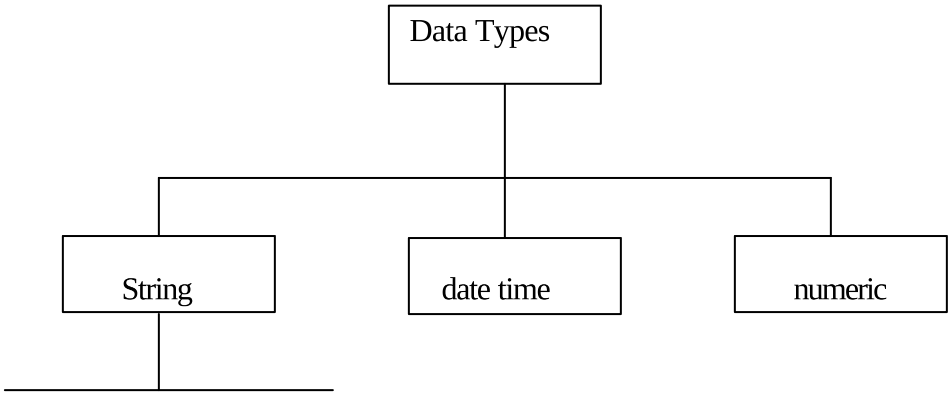
Users perceive a table as a rectangular array of values. Typically, a table represents some class of entities, and each row represents a member of that class, while each column represents an attribute of the members. The rows of table have no intrinsic order, but the columns do. Moreover, each column has a name, by which it can be referred to in SQL statements.

3.2.2.1.1 data types

Each value in a table has a data type. The data type indicates how value is represented in storage . All the values in a given column have the same data types , which is the data type of the column.

The data type of a column determines what you can and cannot do with it. For example, you cannot insert character data, like a last name, into a column whose data type is numeric.

The data types are divided into three general categories: string, numeric, and datetime.



Data Types

String

date time

numeric

DB 2 Data Types	Description
String Data	

CHAR(n)
CHARACTER(n)
	Fixed-length character string n is the number of characters in the string.

VARCHAR(n)	Variable- length character string; n is the maximum number of characters in a string.
GRANPHI(n)	Fixed-length graphic string; n is the number of characters in string.
VARGRAPHIC(n)	Variable-length graphic string ; n is the maximum number of characters in a string
Numeric Data	
SMALLINT	Halfword binary integer. Values an range from –32768 to +32767
INTERER INT	Fullword binary integer, values can range from-214748348 to +2147483647.

DEC(p.s)
DECIMAL(p.s.)
NUMERIC(p.s)
	Decimal value, precision(p) is the total number of digits, and scale (s) is the number of digits to the right of the implied decimal point.

FLOAT (n) REAL (n)	Single precision floating-point number. The value of a determines the representation ; n is between 1 and 21 for single precision floating-point numbers.
DOUBLE PRECISION9(n) FLOAT FLOAT(n)	Double precision floating –point number. The value of n determines the representation ; n is between 22 and 53 for double precision floating –point numbers.
Datetime Data	
DATE	Designates a point in time according to the Gregorian calendar.
TIME	Designates a time of day according to a 24 – hours clock.
TIMES STAMP	Designates a date and time as previously defined by the DATE and TIME data types. The time includes a fractional part in microseconds.
Note: Character strings can contain both single-byte and double-byte characters. Such strings are commonly referred to as mixed data strings.	

The basic operation of SQL are assignment and comparison. Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, and SELECT INTO statements, comparison operations are performed during the execution of statements that include predicates and other languages elements such as MAX,MINDISTINCT, GROUPBY, and ORDER BY.

The basic rule for operations is that the data types of the operands involved must be compatible. The compatibility rule also applies to other operations such as UNION and concatenation. The compatibility matrix for data types is shown in table 3.

Table 3 : Compatibility of Data Types

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-Stamp
Binary Integer	Yes	Yes	Yes	No	No	No	No	No
Decimal Number	Yes	Yes	Yes	No	No	No	No	No
Floating Point	Yes	Yes	Yes	No	No	No	NO	No
Character String	No	No	No	Yes	No	*	*	*
Graphic String	NO	No	No	No	Yes	No	No	No
Date	No	No	No	*	No	Yes	No	No
Time	No	No	No	*	No	No	Yes	No
Time-Stamp	No	No	No	*	No	No	No	Yes

Note :- * The compatibility of date time values is limited to assignment and comparison
Datetime values can be assigned to character string columns and to character string variables. See Chapter 3 of SQL Reference for more information about datetime assignments.

A valid string representation of a date can be assigned to a date column or compared to a date.
A valid string representation of a time can be assigned to a time column or compared to a time.
A valid string representation of a timestamp can be assigned to a timestamp column or compared to a timestamp.

A basic rule for assignment operations is that a null value cannot be assigned to a column that cannot contain null values, or to a host variable that does not have an associated indicator variable. For a host variable that does have an associated indicator variable, a null value is assigned by setting the indicator variable to a negative value.

3.2.2.1.2 concept of NULL

a null value indicates the absence of a column value in a row . A null value is not the same as zero or all blanks.

When a table is created in DB2, each column is defined to have a specific data type. Depending on how it is defined, a column may allow null values. A null value in a column tells DB2that the actual value is unknown. Because of this, logical expressions, such as those as that compare a column value to a constant, could evaluate to true, false, or unknown.

AWHERE clause can specify a column that, for some rows, contains a null value. Normally, values from such a row are not retrieve, because a null value is neither less than, equal to, nor greater than the value specified in the condition. To select values from rows that contain null values, specify;

WHERE COLUMN –name IS NULL

You can also use a predicate to screen out null values, specify:

WHERE column-name IS NOT NULL

When a null value is encountered, its value is treated as if it were higher than all other values. Therefore, a null value appears last in an ascending sort and first in a descending sort. If there are null values in the column you specify in the GROUP BY clause, DB2 considers those null values in the GROUPS BY column to be equal, and returns a single-row result summarizing the data in those rows with null values.

All data types include the null value. The null value is a special value that is distinct from all nonnull values and thereby denotes the absence of a (nonnull) value. Although all data types include the null value, columns defined as NOT NULL cannot contain null values.

3.2..2.1.3. synonyms and Aliases

these are alternate names for tables and views. Synonyms can only used to refer to objects that exist locally, but aliases can be used to refer to remote objects as well as local objects.

Synonym. In SQL, an alternative name for a table or view. Synonyms can only be used to refer to objects at the substem in which the synonmy is defined.

Alias. An alternate name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem.

An alias, like a synonym, is a DB2 object that represents a table or a view. Unlike a synonym, an alias can represent remote tables and views, and it can be used by anyone, not just its creator. In addition, you do not need DB2 authority to use it. However, you must have authority to use the table or view that it represents.

A reference to an alias could a one-two, or three-part name. The rules are basically the same as those used to refer to a table or view.

A one-part name refers to a local alias. If the statement being executed is dynamic, the owner of the alias is your current SQL authorization ID. Otherwise, it is the value specified on the AQUALIFIER bind option. If a value is not specified on the QUALIFIER bind option, then the owner of your package or plan is the qualifier of the alias.

Example : A reference to EMP in a n interactively executed query could refer to the alias SMITH.EMP if your current SQL authorization ID is SMITH.

A two-part name also refers to a local alias. As is true for a table or view, the first qualifier identifies the owner.

Example : JONES.NEWTAB could refer to an alias named NEWTAB and owned by JONES.

A three-part name could refer to either a local or a remote alias. As is true for a table or view, the first qualifier specifies the location, and the second qualifier identifies the owner. If the alias is remote, it must represent a table or view at its own location.

Example : A statement issued at the SAN_FRANCISCO subsystem refers to an alias at DALLAS. The alias referred to must represent a table or view at DALLAS and nowhere else.

Assume now that the alias SMITH.DALEMP has been defined at your local subsystem for the table DALLAS.DSN 8230.EMP. You could then substitute the alias for this table name. The result would look like this :

```
SELECT LASTNAME, MIDINIT, FIRSTNME, EMPNO, WORKDEPT FROM
SMITH.DALEMP
WHERE LASTNAME LIKE '%SON';
```

An advantage to using a locally defined alias is that the SQL statements in which it appears need not be changed if the table or view for the alias is either moved to another location or renamed. To make these statements valid, drop the original alias and create it again, and, for embedded SQL, rebind the program in which it appears.

The option of referencing a table or view by an alias or a synonym is not explicitly shown in the syntax diagrams or mentioned in the description of SQL statements. Nevertheless, an alias or a synonym can be used wherever a table or view can be referred to in an SQL statement, with two exceptions: an alias cannot be used in CREATE ALIAS, and a synonym cannot be used in CREATE SYNONYM. If an alias is used in CREATE SYNONYM, it must identify a table or view at the current server. The synonym is defined on the name of that table or view. If a synonym is used in CREATE ALIAS, the alias is defined on the name of the table or view identified by the synonym.

The effect of using an alias or a synonym in an SQL statement is that of text substitution. For example, if A is an alias or synonym for table Q.T, one of the steps involved in the preparation of SELECT *FROM A is the replacement of 'A' by 'A.T' .

The differences between aliases and synonyms are a s follows :

SYSADM or SYSCTRL authority or the CREATE ALIAS privilege is required to define an alias. No authorization is required to define a synonym.

An alias can be defined on the name of a table or view, including tables and views that are not at the current server. A synonym can only be defined on the name of a table or view at the current server.

An alias can be defined on an undefined name. A synonym can only be defined on the name of an existing table or view.

Dropping a table or view has no effect on its aliases. But dropping a table or view does drop its synonyms.

An alias is a qualified name that can be used by any authorization ID. A synonym is an unqualified name that can only be used by the authorization ID that created it.

An alias defined at one DB2 subsystem can be used at another DB2 subsystem. A synonym can only be used at the DB2 subsystem where it is defined.

When an alias is used, an error occurs if the name that it designates is undefined or is the name of an alias at the current server. (Note, however, that the name can designate an alias defined at another server if the alias represents a table or view at this other server), When a synonym is used, this error cannot occur.

3.2.2.1.4. COMMENT ON

The COMMENT ON statement adds or replaces comments in the descriptions of tables, views, aliases, or columns in the DB2 catalog at the current server.

Example 1 : Enter a comment on table DSN8230.EMP.

```
COMMENT ON TABLE DSN8230.EMP
IS ‘REFLECTS IST QTR 81 REORG’
```

Example 2 : Enter a comment on view DSN 8230, VDEPT.

```
COMMENT ON TABLE DSN8230.VDEPT
IS  ‘VIEW OF TABLE DSN8230.DEPT’.
```

Example 3 : Enter a comment on the DEPTNO column of table DSN8230.DEPT.

```
COMMENT ON COLUMN DSN8230.DEPT.DEPT NO
IS  ‘DEPARTMENT ID-UNIQUE’.
```

Example 4 : Enter comments on two columns in table DSN8230.DEPT.

```
COMMENT ON DSN8230 DEPT
(MGRNO IS ‘EMPLOYEE NUMBER OF DEPARTMENT MANAGER’
ADMRDEPT IS ‘DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT);
```

3.2.2.1.5. LABEL ON

The LABEL ON statement adds or replaces labels in the descriptions of tables, views aliases, or columns in the catalog at the current server.

Example 1: Enter a label on the DEPPNO column of table DSN8230.DEPT.

```
        LABEL ON COLUNMN  DSN 8230.DEPT.DEPTNO
                IS ‘DEPARTMENT NUMBER’;
```

Example 2 : Enter labels on two columns in table DSN8230.DEPT.

```
        LABEL ON DSN8230.DEPT
        (MGRNO IS ‘MANAGER’S  EMPLOYEE NUMBER’,
        ADMRDEPT IS ADMINISTERING DEPARTMENT);
```

3.2.2.2. Index

An index is an ordered set of pointers to rows of a base table. When you request the creation of an index, DB2 builds, this structure and maintains it automatically.

When you request the creation of an index, you specify a key. The key consists of one or more columns of the table. In the CREATE statement, you identify these columns and their order of appearance in the key.

The key has a value for each row in the table for which the index is defined. The value is based on the values for that row of the columns in the key. The value of a key with two or more columns is the concatenation of the column values in the specified order.

3.2.2.2 View

Views provide an alternative way of looking at the data in one or more tables. Like tables, views have rows and columns with no inherent order of rows. You specify view names in the FROM clause of the SELECT statement in the same way that you specify table names. You can create views and authorize their use for many table-like operations.

When you create a view, you specify a query in the CREATE VIEW statement. The result table defined by this query is then the table the view represents in operations involving the view. This is sometimes termed the view table to distinguish it from the view's definition. Looked at this way, the view table might change as changes are made to the base tables for the view, but the view definition remains the same.

Consider, for example, the following SQL statement

```
CREATE VIEW XYZ (EMPLOYEE, WHEN_HIRED)
AS SELECT EMPNO, HIREDATE
FROM DSN 8230, EMP
WHERE WORKDEPT IN ('A00' 'D1 1)
```

The view is named XYZ. The view table consists of certain values in the table DSN8230. The values are those in the EMPNO and HIREDATE columns for the rows representing employees in departments A00 and D11. For the view table, these columns have the names EMPLOYEE and WHEN_HIRED.

A table has a storage representation, but a view does not. After an operation involving a view is ended, the view table disappears. A view's definition is stored in the catalog. No data is stored and, therefore, no index can be created for a view. However, an index created for a table on which a view is based can improve the efficiency of operations involving the view.

Views can be used for the following :

Control access to a table
Access to a view of a table can be granted on the view without granting access to the table itself. The view could be defined to show only portions of data in the table, thereby screening out sensitive data.

Make data easier to use.

For example, a view can show summary data for a given table, combine two or more tables in meaningful ways, or show only rows that are pertinent to the process using the view.

Some valid operations on tables are not valid on views or the definition of views, for example, read-only views cannot be used to insert, update, or delete rows in their base tables. Also, the keyword, UNION, Cannot be used in the definition of a view. There might be times when no single table contains all of the data that you need. Instead, the data is scattered among several tables. DB2 provides views so you can look at parts of a table or at data from several combined tables.

A view does not contain data ; it is a stored definition of a set of rows and columns. A view scan present any or all of the data in one or more tables, and in most cases, can be used interchangeably with tables.

When a program accesses the data defined by a view, DB2 processes the view's definition. This results in a set of rows the program can access with SQL statements. Views that can be updated are subject to the same referential constraints as the tables upon which they are defined.

Use the CREATE VIEW statement to define a view and given the view a name, just as you do for a table.

```
CREATE VIEW VDEPTM AS
SELECT DEPTNO, MGRNO, LASTNAME,  ADMRDEPT
FROM DSN8230,  DEPT, DSN8230,  EMP
WHEREDSN 8230.EMP.EMP NO =  DSN 8230.DEPT.MGRNO.
```

This view adds each department manager's name to the department data in the DSN8230. DEPT table. Now that the view VDEPTM exists, you can manipulate the data in it just a s you would manipulate the data in DSN8230. DPET and DSN8230.EMP. However a view defined on more than one table cannot be used for update operations. It is considered a read-only view. To see all of the data in the view, you can execute the following statement.

```
SELECT
FROM VDEPTM;
```

When you define a view, DB2 stores the definition of the view in the DB2 catalog but does not store any data for the view since the data already exists nit the source tables or tables. Views are kept up to date automatically as the tables on which they are based are updated. You can use views to limit access to certain kinds of data, such as salary information. Views can also be used to the following :

Make a subset of a table's data available to an application. For example, a view based on the employee table might contain rows for a particular department only.

Combine data from two or more tables and make the combined data available to an application. By using a SELECT statement that matches values in one table with those in another table, you can create a view that presents data from both tables. However, data defined by this type of view can only be selected. You cannot update, delete, or insert data into a view that joins two tables.

Perform functions or operations on data in a table, and make the resulting data available to an application. For example, the resulting data computed by DB2 can be ;

- The sum of the values in a column
- The maximum value in a column
- The average of the values in a column
- The length of a value in a column
- The value in a column converted to another data type
- The result of an arithmetic expression applied to one or more Columns, such as (COLB + COLA)/COLC.

When using views in an application program, whether to simplify a table, to get more or different data than a single table contains, ore prevent a user from seeing classified columns, you must be aware of the restrictions that apply to some views.

Because a view does not contain data, special rules apply for inserting, deleting, or updating rows in a view. If the view is merely a subset (or a rearrangement) of a base table's rows or columns :

The owner of the plan or package that contains the program must be authorized to update, delete, or insert rows into the view. You are so authorized if you have created that view and have privileges for the table on which the view is based. Otherwise, to bind the program you have to obtain authorization via a GRANT statement.

When inserting a row into a table (via a view), the row must have a value for each column of the table that does not have a default value. If a column in the table on which the view is based is not specified in the view's definition, and if the column does not have a default value, you cannot insert rows into the table via the view.

3.2.2.3.1. View Merge

In the view merge process, the statement that references the view is combined with the subselect that defined the view. This combination creates a logically equivalent statement. This equivalent statement is executed against the database. Consider the following statements.

View defining statement :

```
CREATE  VIEW VIEW1  (VC1,  VC21,  VC32)  AS
SELECT  C1,  C2,  C3 FROM  T1
WHERE C1 >  C3;
```

View referencing statement :

```
SELECT  VC1,  VC2
FROM  VIEW1
WHERE VC1 IN   (A,B,C) ;
```

The subselect of the view defining statement can be merged with the view referencing statement to yield the following logically equivalent statement.

Merged statement :

```
SELECT  C1,  C2  FROM  T1
WHERE  C1 > C3  AND C1  IN (A,B,C) :
```

The merged statement is executed to achieve the intended result :

3.2.2.3.2. View Materialization

It is not always possible to merge. In the following statements :

View defining statement :   View referencing statement :

```
CREATE VIEW VIEW1  (VC1,  VC2)  AS          SELECT  MAX  (VC1)
SELECT  SUM (C1),  C2 FROM T1               FROM  VIEW1 ;
GROUP  BY  C2:
```

Column VC1 occurs as the argument of a column function in the view referencing statement. The values of VC1, as defined by the view defining subselect, are the result of applying the column function SUM (C1) to groups after grouping the base table T1 by column C2. There is no equivalent single SQL SELECT statement which can be executed against the base table T1 to achieve the intended result. There is no way to specify that column functions should be applied successively.

In the previous example, DB2 performs view materialization, which is a two step process.

1. The view's defining subselect is executed against the database and the results are placed in a temporary table.
2. The view's referencing statement is then executed against the temporary table to obtain the intended result.

Whether a view needs to be materialized depends upon the attributes of the view referencing statement, or logically equivalent referencing statement from a prior view merge, and the view's defining subselect.

3.2.2.4. Catalog tables

Each DB2 catalog table maintains data about an aspect of the DB2 environment.

The catalog tables describe such things as table spaces, tables, columns, indexes, privileges, application plans, and packages. Data in the catalog tables is available to authorized users of DB2 through normal SQL query facilities.

The DB2 catalog is composed of 9 tablespaces and 39 tables all in single database, DSNDB06. This records all the information required by DB2 for the following functional areas :

Objects	STOGROUPS, databases, tablespaces, partitions, tables, columns, views, synonyms, aliases, indexes, index keys, relationships, plans, packages, and DBRMs.
Security	Database privileges, plan privileges, system privileges, table Privileges, and use privileges.
Utility	Image copy data sets, REORG executions, LOAD executions, and object organization efficiency information.
DB2 Catalog	Links and relationships between the DB2 Catalog tables.

The catalog tables are updated by DB2 during normal operations in response to SQL data definition statements, SQL control statements, and certain commands and utilities.

Some of the DB2 catalog tables are briefly described here :

The SYSIBM.SYSCOLUMNS table contains one row for every column of each table and view.

The SYSIBM.SYSCOPY table contains information needed for recovery.

The SYSIBM.SYSDATABASE table contains one row for each database, except for database DSNDB01.

The SYSIBM.SYSDBAUTH table records the privileges held by users over databases.

The SYSIBM.SYSDBRAM table contains one row for each DBRM (database request module) of each application plan.

The SYSIBM.SYSFIELDS table contains one row for every column that has a field procedure. It can also contain up to ten additional rows for every column that serves as the first column in an index key. The additional rows, which contain statistics, can be added when the index is scanned by the RUNSTATS utility. The column for the added rows need not have a fieldprocedure.

The SYSIBM.SYSFOREIGNKEYS table contains one row for every column or every foreign key.

The SYSIBM.SYSINDEXES table contains one row for every index.

The SYSIBM.SYSINDEXPART table contains one row for each nonpartitioned index and one row for each partition of a partitioned index.

The SYSIBM.SYSKEYS table contains one row for each column of an index key.

The SYSIBM.SYSLINKS table contains information about the links between DB2 catalog tables.

The SYSIBM.SYSPACKAGE table contains a row for every package bound at the local DBMS, regardless of the location of the binder.

The SYSIBM.SYSPACKAUTH table records the privileges held by users over packages bound at the local DBMS.

The SYSIBM.SYSPACKDEPT table records the dependencies of packages bound at the local DBMS on local tables, views, synonyms, table spaces, indexes, and aliases. Each row represents a package and an object on which the package depends.

The SYSIBM.SYSPACKLIST table contains one or more rows for every local application plan bound with a package list. Each row represents a unique entry in the plan’s package list. The entry could choose an individual package or all the packages in a given collection.

The SYSIBM.SYSPLAN table contains one row for each application plan.

The SYSIBM.SYSPLANAUTH table records the privileges held by users over application plans.

The SYSIBM.SYSPLANDEP table records the dependencies of plans on tables, views, aliases, synonyms, table spaces, and indexes.

The SYSIBM.SYSRELS table contains one row for every referential constraint.

The SYSIBM.SYSSTMT table contains one or more rows for each SQL statement of each DBRM.

The SYSIBM.SYSSTOGROUP table contains one row for each storage group.

The SYSIBM.SYSSNONYMS table contains one row for each synonym of a table or view.

The SYSIBM.SYSTABAUTH table records the privileges held by users on table nad views.

The SYSIBM.SYSTABAUTH table records the priviliges held by users on tables and views.

The SYSIBM.SYSTABLES table contains one row for each table, view, or alias.

The SYSIBM.SYSTABLESPACE table contains one row for each table space.

The SYSIBM.SYSVIEWDEPT table records the dependencies of views on tables and other views.

The SYSIBM.SYSVIEWS table contains one or more rows for each view.

3.2.2.5 The DB2 Directory

The DB2 directory is composed of five “tables”. These tables, however, are not true DB2 tables because they are not addressable using SQL. Let us call these “tables” structures. These structures control DB2 housekeeping tasks and house complex control structures used by DB2. Brief description of these five structures are given below :

SCT02 The SCT02 structure holds the skeleton cursor tables (SKCTs) for DB2 For DB2 application plans. These skeleton cursor tables contain the instructions for implementing the access path logic determined by the DB2 optimizer. The BIND PLAN command causes skeleton cursor tables to be created in the SCT02 structure.

SPT01 This structure houses skeleton package tables, which contain the access Path information for DB2 packages. The BIND PACKAGE command causes this table to be created in this structure.

DBD01 Database descriptors, or DBDs, are stored in the DBD01 DB2 Directory Structure.

SYSUTIL This structure monitors the execution of all on-line DB2 utilities.

SYSLGRNG Information from the DB2 logs are recorded on SYSLGRNG for tablespace updates.

3.3. Review Questions

- What do you understand by logical objects ?
- What do you understand by physical objects ?
- What is DBD ?
- What is a STORAGE GROUP ?
- What is a TABLE SPACE ?
- What are the different kind of Table Spaces ?
- What is a buffer pool ?
- What is a prefetch ?
- What are the four buffer pools DB2 supports ?
- What does PRIQTY 1000 means in Table Space definition ?
- How does SECQTY work ?
- Explain each of the terms ERASE, LOCKSIZE, PCTFREE, FREEPAGE.
- What PCTFREE and FREEPAGE values will you keep for read-only database ?
- Can one segment have more than one table in a segmented table space ?

- Can one table occupy more than one segment in a segmented table space ?
- When does a partitioned table space is required ?
- What is a SPACE MAP PAGE ?
- Can we have a segment size 6 pages in a segmented table space ?
- What is an index space ?
- When does a partitioned Table Space is required ?
- What is a SPACE MAP PAGE ?
- Can we have a segment size 6 pages in a segmented table space ?
- What is an index space ?
- Can we create an index space ? Justify
- What does the SUBPAGE option do ?
- What is a NULL value ?
- Where will a NULL value appear in an ascending and descending sorts ?
- What is synonyms and Aliases ?
- What are the differences between a Synonyms and Aliases ?
- What is COMMENT ON statement ?
- What is LABEL ON statement ?
- What is an Index ?
- What are view ?
- What do you understanding by view merge ?
- What do you understanding by view materialization ?
- What are the functional areas by which DB2 catalog tables are divided ?
- What is DB2 Directory ?

4
Referential Integrity

4.1. Objectives

In this chapter we will discuss

- Referential integrity
- DB2's support of referential integrity
- Constraint when RI is in effect
- Self-referencing tables
- Cycles
- Delete-connected tables

Consider a system that has a Supplier table (SN, SNAME, CITY, STATUS), a Parts table (PN, PNAME, PCITY, WEIGHT, COLOR) and a shipments table (SN, PN, QTY). with SN, PN and SN + PN respectively as the keys to the three tables. Clearly it does not make sense for a row in the shipments table to refer to an SN that does not exist in the supplier table. SN is a primary key in the supplier table, and a foreign key in the shipments table.

Referential integrity rules could be of three types, restrict (action rejected), cascade rule (deletion will be cascade) and neutralise rules (NULLs substituted approximately).

4.2. DB2's Support

While creating or altering tables, columns can be designated as primary or foreign key. Then DB2 automatically disallows changes to foreign key columns that would violate referential integrity. It also enforces rules governing deletions and updates of primary key columns. The developer can, while creating the foreign key column, specify whether DB2 should employ restrict, cascade or neutrialise when deletions are done on primary key columns. Update of a primary key column is not allowed if the old value exists in the referenced foreign key column.

RI checking requires a lot of resources, but in general, DB2's enforcement of RI will be more efficient than enforcement by applications themselves. This is because it can make proper use of available indexes, which may not always be the case when the application has coded the logic.

In some situations DB2's enforcement may not be suitable to the organisation's policies. Also in some kinds of batch updates, it may be better for an application to do RI checking. For example, when multiple insertions on a single foreign key value are to be done then the RI check need be done only once, if it is done in the application.

```
CREATE TABLES
(SN  CHAR(6)          NOT  NULL,
SNAME   CHAR(20)  NOT FULL WITH DEFAULT,
STATUS  SMALLINT  NOT NULL WITH DEFAULT,
CITY CHAR (15)  NOT NULL WITH DEFAULT,
PRIMARY KEY  (SN)
IN DAGKWB.STSP ;

CREATE  UNIQUE INDEX SNX  ON  S  (SN);

CREATE TABLE SPJ
(SN   CHAR(6)  NOT NULL,
PN  CHAR(6)  NOT NULL,
JN   CHAR (6)  NOT NULL,
QTY.  INTEGER NOT FULL WITH DEFAULT,
PRIMARY  KEY  (SN, PN, JN),
FOREIGN KEY SNFK  (SN) REFERENCES ON  DELETE RESTRICT.


FOREIGN KEY PNFK  (PN) REFERENCES P ON DELETE CASCADE,
FOREIGN KEY JNFK  (JN)  REFERENCES J ON  DELETE SET NULL)
IN  DAGKWDB.SPJTSP;

CREATE UNIQUE INDEX SPJX
ON  SPJ (SN, PN, PN);

CREATE INDEX SPJPNX
ON SPJ (PN);

CREATE INDEX SPJJNX
ON SPJ  (JN);
```

SNFK, PNFK and JNFK are names of referential constraints (max 8 characters). The same foreign key can appear in multiple tables, but the referential constraint's name must be unique to each table. Foreign and primary keys should be of the same data type and length. This applies to simple and composite keys. The sequence of their indexes however need not be the same. Foreign keys need not have indexes, but it is better to create indexes on foreign keys. Notice that the FOREIGN KEY clause does not specify the primary key. This is implicit. Also a foreign key can reference only one table. (This is not what Codd has laid out).

4.3. Foreign Keys and NULLs

Primary and Foreign key columns need not have the same NULL defaults. Whether foreign keys should be allowed to be null or not depends on the situation. In the S, P, SP situation it would not make sense. But in an employee table, the department number could conceivably be null if the employee is yet to be assigned.

4.3.1. Adding Keys

A primary key can be added to an existing table by an ALTER TABLE statement. There must first be a unique index on that column.

```
ALTER TABLES
PRIMARY KEY (SN)
```

A foreign key can also be added to a table.

```
ALTER TABLE SPJ
FOREIGN KEY SNFK  (SN) REFERENCES
ON DELETE RESTRICT ;
```

After this is done, DB2 does not allow any access to the table till it checks out that no violations of the foreign key already exist in the table. Any violations found are written to an exception table. A foreign key and the associated constraints can be dropped for batch updates. After this, when the foreign key is redefined, the CHECK DATA utility should be used to verify that all is well.

4.4. Constraints when Integrity Checking is in Effect

Primary key update

Consider a table TEST with TK (integer) as key

```
UPDATE TEST SET TK = TK + 1A;
```

This can possibly cause a violation of entity integrity. At bind time DB2 recognises this and flags it an error.

Delete with subselect

Consider the tables

S

S#	SNAME	STATUS	CITY	S_OWNER
S1	SMITH	20	LONDON	-
S2	JONES	10	PARIS	S1
S3	BLAKE	30	PARIS	S2

JSUP

J#	PRIMARY_S	SECONDARY_S
J1	S1	S2
J2	S1	S2
J3	S2	S3

Both PRIMARY_S and SECONDARY_S are foreign keys.

```
DELETE FROM S
WHERE S NOT IN
(SELECT SECONDARY_S
  FROM JSUP
WHERE JSUP.SECONDARY_S = S.S#);
```

The results of this statement, however, will be different depending on the order in which the rows are processed. If S# = S1 is evaluated in the subselect first, the NOT IN clause evaluates as true because S1 is not a secondary supplier. Therefore, S1 is deleted from the S table. Since it is a primary key value and a cascade rule is in effect, the S1 rows in

JSUB table also deleted. When S# = S2 is evaluated in the subselect, the NOT IN clause also returns true, since the previous cascade delete had eliminated the two jobs in which S2 had been secondary supplier. S2, therfore, is also deleted and its deletion cascades to eliminate the J3 row. Now, when S# = S3 is evaluated in the subselect, the NOT IN clause returns true and S3 is deleted.

Suppose, however, that S3 were the first value from the S table to be evaluated in the sublselect. In that case, the NOT IN clause would return false because, at this point in the processing, the J3 row will exists, with S3 as its secondary supplier, and S3 row will not be deleted.

To avoid these anomalies, DB2 will not allow deletions based on a subselect that references the table from which the deletions are being made.

Self reference

A single table can include both a primary key and a related foreign key. This is called self reference.

Consider the following table in which S_OWNER is foreign key and S# is primary key.

S_O

S#	SNAME	STATUS	CITY	S_OWNER
S1	SMITH	20	LONDON	NULL
S2	JONES	10	PARIS	S1
S3	BLAKE	30	PARIS	S2

DELETE RESTRICT is of little practical use. The DELETE SET NULL rule is not suitable for a self referencing constraint because it holds the potential for unpredictable results.

Consider the following query.

```
DELETE FROM S_O WHERE_OWNER IS NULL
```

If DELETE SET NULL were in effect, when the S# =S1 row was deleted, S1 in S_OWNER column would be set to NULL. If the rows were processed in order, when the S# = S2 row was deleted, S2 in S_OWNER column would be set to NULL. Eventually, all rows in the table would be deleted. If the rows are not processed in order, then less number of rows will be deleted.

Therefore by definition, the foreign key in a self referencing table must specify DELETE CASCADE rule.

A self referencing constraint must be created with an ALTER TABLE statement. This is because the primary key table and key must exist before the foreign key can be defined.

In a self referencing table, more than one row cannot be inserted with a subselect. Because, depending upon the order of insertion some rows from subquery may get inserted or not get inserted.

4.5 CYCLES

One table is a primary key table to other and the second is a primary key table to the first. Cycles could involve more than tow tables also. DELETE CASCADE may not be used in a tow table cycle. In a cycle involving more tables, at least one of the foreign keys must employ the RESTRICT or SET NULL rule.

4.6 DLELTE-CONNECTED TABLES

tables related by foreign keys are delete connected because the presence or absence of values in one table effects delete processing in the other. When transitive dependencies occur, there can be problems. For example A may be delete connected with B which is delete connected with C, thereby implying that C is delete connected with A. through another path C may be directly or indirectly delete connected with A. C is thus delete connected with A in more than one way. If the nature of delete connection is different then different orders of processing will yield different results. Thus DB2 requires that the foreign keys that establishes table’s multiple delete connections must all be defined to have the same delete rule and the rule may not be the SET NULL rule. Attempts to violate this rule will be prohibited.

4.7 Review Questions

- What is referential intergrity?
- What is the maximum number of foreign keys that can be there for a table?
- Can a primary key be a foreign key?
- What does DLELETE RESTIRCT do?
- What does DELETE CASCADE do?
- What does DELETE SET NULL do?
- What is self referencing tables?
- What delete rule does DB2 supports on a self referencing table?
- How can we create a foreign key on a self referencing table?
- Why DB2 does not allow multiple primary key updation if that is referenced?
- Why DB2 does not allow deletions based on a sub select on a primary key table?
- Why is it necessary to have both PK and FK fields to be of same data type & length?

5

Indexing

5.1 Objectives

This chapter deals with indexing and performance. In this chapter we will discuss about.

- Matching and non-matching scan
- Structure of a B-Tree node
- Selecting columns to index
- Columns to avoid indexing
- Multiple indexes
- Value of clustering
- Index cardinality and composite indexes
- Impact of column ordering in indexes
- Intersection tables
- High rebundacy indexes
- Maintaining clustering
- Index monitoring

DB2 uses a B-tree index. It can be used-for matching scans when looking for specific value. Pointers chain the leaf nodes in the sequence of key values. Thus the index can be used to find a specific value from where sequential operations may then be performed. This is a non-matching scan. This can be used to locate all customers having Ids between 200 and 500, say (indexed on ID). First a matching scan will find the key 200. Then on sequential traversal begins similarly requests for all values below a given value will first fix on the start of the chain and go on till the specified end point is reached.

Sometimes DB2 can satisfy the query by doing a scan of the index only without going to the data pages at all (matching or non-matching scans without data reference).

5.2 STRUCTURE OF THE BTREE NODE

the leaf page level must hold enough index entries and associated pointers to point to point to all rows in a table. For unique indexes, the leaf node holds one entry and one pointer. The pointer is a 4 byte RID indicating the row's position in the data pages. The leaf page in a non-unique index has space for a value and up to 255 pointers. If more than 255 occurrences of a particular value exist then a new leaf node with the value and 255 more pointer is created, and so on. A single index page contains 3660 bytes of usable space. The remaining space may be used for control Information.

5.3 SLECTING COLUMNS TO INDEX

Updating a value on which a table is indexed may cause it to be moved from one page to another. When a table is loaded or recovered each index to it must be built or rebuilt. When tablespaces are reorganised indexes must be rebuilt.

If a table has less than 7 pages the optimizer would generally not use an index even if one exists, because a complete scan will be faster. Possible exceptions to this arise when joins or referential intergrity checking are involved.

Updates represent most of the cost of indexes. If a table experiences bulk periodic updates, affecting more than about 10 of its rows, the dropping and recreating the indexes may be better.

Therefore, we should avoid indexing columns

- that have redundant values, unless a clustering index is used
- that have a skewed distribution of data
- that are frequently updated
- that are longer than 30 character long

nonclustering indexes are beneficial on columns that are searched often, or used injoins of small percentages of their rows. Foreign keys are good candidates. Indexing columns on which aggregates are often performed (SUM, COUNT, etc.), is also beneficial as DB2 can satisfy the query without accessing the data pages. The same reasoning applies to MAX, MIN, etc.

Example:

Consider the following query,

```
SELECT SUM (QTY) FROM SP;
```

If there is an index on QTY, then only leaf nodes of the index pages need to be accessed. From that, we can find out the SUM (QTY) without accessing data pages at all.

Clustering indexes should be selected with care since only one is allowed per table. God for columns with high rebundancy that are searched over ranges of values. Hence clustered index should not be created on primary key or unique keys. Foreign key columns are usually highly rebundant, and are used in joins. They are good candidates for clustering indexes.

MAX and MIN can be satisfied in one index access of descending or ascending index repectively. If a predicate is predicate is specified in a query with an aggregate function, a matching scan can be used to narrow the search, before the aggregrate is applied. Processing all leaf pages will require much fewer I/Os than scanning all data pages since many more leaf nodes than data records will fit into a page. Clustering is useful when all the values need to be processed in sequence. Without a clustering index, DB2 will need to sort to satisfy queries involving GROUP BY, ORDER BY or DISTINCT. While inserting, if a clustering index exists, it will be used. If no clustering index exists, a non-clustering index may be chosen by the optimizer. This will be the first index created for the table. Initially values are inserted as if this were the chosen clustering index. Re-organisation will however not preserve this clustering.

Columns used freqently together can be considered for composite indexing. Querying on trailing portions of a composite index is inefficient.

5.4 Columns to Avoid Indexing

high rebundancy columns are not good for nonclustering indexes. This is because of high cost of updating, as well as because they are of little value for retrieval. If a high percentage of data rows contain a particular value, a full scan will be more effiecient than going through an index. In general if distinct values will, on the average, appear more than once on each data page, then nonclustering index is of little use.

Example:

Consider a table A with cardinality of 20,000 and assume these 20,000 rows are uniformly distributed among 100 pages. Consider a column of the table A with cardinality of 100.

Number of rows for a value of the column = 20,000/100 = 200

These 200 rows may uniformly get distributed among 100 pages. A page will have 2 rows for a given value of the column. The, for a equality condition, at the worst case, we need to access each page twice. In this condition, using index is of little use. Suppose if the number of distinct values is 400, then number of rows slected for a value will be 50 and a page has only 0.5 row for a particular equality condition.

Sometimes indexes will be needed for locating infreqently occuring data in skewed tables.

5.5 Multiple Indexes

Useful for satisfying queries involving multiple predicates jointed by AND and OR. The qualifying RIDs obtained through the individual indexes are sorted an merged. The resulting RIDs will be in order and hence reduce the number of page accessed (list prefetch)

5.6 Value of Clustering

If a non-clustering index is used for accessing data, then a given data page may be accessed multiple times. Clustering indexes allow for merge join without sorting the inner and outer tables. Indexes will be used in joins only if the joining columns are of the same data type and length. This kind of mismatch is sometimes the cause of poor performance. Use of functions to achieve compatibility also will not help as this will negate the use of indexes.

Example:
Consider an employee table with 10,000 rows located on 200 data pages, Employees distributed uniformly across 20 offices. Consider locating all employees in a particular office (500 rows will be returned). With a non-clustering index the leaf pages will point to the 500 RIDs. Thus 500 I/Os will be required, with some of the 200 data pages being read more than once. In some cases list prefetch will be employed. In this the RIDs will be sorted and sequential prefetch will be used, in which 32 pages will be obtained in one I/O. each page need be read only once. Thus the number of I/Os will be 200/32.

With a clustering index a sort of the RIDs can be avoided as once the first record in the office is located (with a matching index scan), a sequential scan will give the rest. Thus about 10 I/Os will suffice (over and above those for the index scan). If the distribution were not uniform and say 70-80% of the employees were located in the said office, then use of the index will not be efficient and tablespace scan will be preferred. With static SQL where a host language variable gave a relevant variable, the optimizer will not be able to decide not to use the index. With dynamic SQL this would be possible.

5.7 Index Cardinality and Composite Indexes

DB2 analyses the extent of rebundancy and decides whether a tablespace scan or index scan is preferred. It estimates what percentage of the total number of row will be returned by the SELECT. This is the FILTER FACTOR. With composite indexes the individual filter factors are multiplied. There may not always be complete information available to DB2. The following information is kept by DB2.

FIRSTKEYCARD – Number of distinct values in a single indexed column or in the leading column of a composite index. SYSIBM. SYSINDEXES keeps this information. DB2 uses this to determine rebundancy for single columan indexes or when only the leading column of a multiple column index is specified.

FULLKEYCARD – Number of distinct values in a single column. Kept in SYSIBM.SYSCOLUMNS. If the column is indexed this will be the same as FIRSTKEYCARD. This can be entered by the user (estimate) or RUNSTATS can be invoked to update it. The optimizer uses it to determine rebundancy for composite indexes in some cases.

5.8 Impact of Column Ordering in Indexes

Ordering of columns in a composite index can be important. Consider the following query:

```
SELECT PN, PNAME
FROM P
WHERE CITY =’XYZ’
AND WEIGHT > 500;
```

Consider keeping a composite index on city and weight. If there were far fewer entries with CITY = ‘XYZ’ than parts with weight > 500, then putting city before part will be useful because a matching scan for XYZ, 500 followed by a sequential scan will read about half of all the XYZ entries to satisfy the query. If weight were put first, then a matching scan for 500, XYZ followed by a sequential search of all parts with weight greater than 500 may be needed. Columns with high filtering capability should be put first.

5.9 Intersection Tables

intersection tables are tables made up of a number of foreign key columns that participate in a composite primary key (example SPJ table). Foreign keys are good candidates for indexing. In the SPJ case do we need indexes on all three? A unique index on SN, PN, JN will anyway exist. Thus this itself can be used for referential integrity checking and for joins on the first column. Thus one index can be eliminated. About the others, the kind of activity is the guide. If heavy update will take place on the primary key then referential integrity checking will be facilitated if the foreign key index exists. Without it, tablespace scan will be necessary.

5.10 High Redundancy Indexes

Updating these requires heavy I/O. this is because the RID's have to be added to the list, which may cause page splitting and consequently locking of index and loss of concurrency. If several rebundant columns are indexed in a table, then update and delete processing can be very slow. Delete processing is even slower as it involves deleting and shifting of RIDs to the left from each rebundant index. Low cardinality columns should ideally not be indexed. If at all needed, some other column can be added to increase the cardinality to improve performance.

5.11 Maintaining Clustering

DB2 keeps track of which columns are clustered. After RUNSTATS the percentage of clustering is determined. If it is above 95%, then the column is considered to be clustered. Clustering is maintained as long as free pages and freespace in pages will allow it. After that clustering will be disturbed, till reorganisation is done again. The percentage of clustering is used by DB2 to determine if an index should be used. The optimizer will go by the value recorded in SYSIBM.SYSINDEXES for CLUSTERRATIO. If RUNSTATS is not executed, then the optimizer can be misled to make a wrong choice. Periodically re-organisation should be done. To identify the tables which are marked as clustered but are in fact below 80% clustering the administrator my use SQL:

```
SELECT NAME, CREATOR, TBNAME, CLUSTERRATIO, CLUSTERING,
CLUSTERED
        FROM SYSIBM.SYSINDEXES
        WHERE DBNAME ='ABCDEF' AND CLUSTERING ='Y'
                AND CLUSTERRATIO < 0.80;
```

REOGR will not recluster an index when multiple tables share a simple tablespace. Running REORG does not cause automatic rebind. Typcially, after a REORG, RUNSTATS should be used so that the optimizer can use current information on future bind operations.

5.12 Index Monitoring

tow utilities are provided for index monitoring, viz., SYSPLANDEP and EXPLAIN. SYSPLANDEP can be used to find the programs using an index. If very little use is being made the index can be dropped. EXPLAIN command can be used to determine what access path is chosen by the optimizer for a plan. The user can then explore alternate ways of stating quering to make best use of available indexes, or to create new indexes.

5.13 Review Questions

- What is a clustered index?
- What does an unique index do?
- When to create an index?
- Are data pages accessed for all queries?
- What is RID?
- What is the pointer sturcture of an index?
- How many pointers can there be for a data value? What happens if the number crosses the limit?
- Why for rebundant value columns, it is not good to have non-clustered index?
- Is it necessary to have an index on primary key? If so, which index?
- Is it necessary to have an index on foreign key? If so which index?
- How many clustered index per table is allowed?
- Is it good to have an index on frequently updated column?

6

Councurrency Control

6.1 Objectives

Issues to be discussed in this chapter are:

- -Locking levels in DB2
- -Lock escalation limits
- -Share, Exclusive, and Update locks
- -Isolation Level
    - -Repeatable Road
    - -Cursor Stability
- -Intent Locks
- -Acquire and Release parameters
- -When should you commit
- -Explicit Locking in DB2
- -Deadlocks

Concurrency is one of the major area to be handled by a database management system. Only in multi-user environment, the system need to give concurrency support. Various types of locks are supported by DB2. In this chapter, We will see various principles involved in DB2 concurrency control.

6.2 Locks

Page, table and tablespace level locking is provided. Index can be locked at page or subpage levels. While creating tablespaces the level of locking can be specified. Locksize of ANY will allow DB2 to determine lock size. For indexes also, the locksize can be specified at creation time. However if tablespace lock is in effect, then the index level locks will not be used.

6.2.1 Lock Excalation Limits

NUMLKTS – causes a page locksize to be promoted to a tablespace locksize when any one program or user request has taken more than a specified number of page locks on a single tablspace (This will happen only when LOCKSIZE is defined with ANY).

NUMLKUS – issues an error message when a user or program exceeds a predetermined number of locks on all tablespaces.

SHARE AND EXCLUSIVE LOCKS

DB2 uses share locks while responding to SLELCT statements. Exclusive locks are used for INSERT, UPDATE and DELETE. When DB2 attempts to insert a row and finds the corresponding page locked, it will put it on the next available unlocked page. This avoids lock delays but contributes to unclustering. DB2 user X, S and U locks, and the corresponding protocols. Exclusive lock bars any access to the locked resource. It is the corresponding protocols. Exclusive lock bars any access to the locked resource. It is necessary to read a row before it can be updated or deleted, and during this read, the update lock allows other to read the page and acquire a share lock. An update lock must wait for any share lock to be released so that an exclusive lock can be taken before the actual update or delete. The update lock is queued with share locks to get an exclusive lock.

6.2 Isolation Level

Isolation level means the extent to which the transaction islolates the data is accessed.

CURSOR STABILTIY or REPEATABLE READ (default) isolation level can be specified at bind time. Under cursor stability, DB2 takes a lock on the page the cursor is accessing, and releases it when the cursor moves on. Under repeatable read, it holds all page locks as the cursor moves on, and relases them only at commit point. Cursor stability provides better concurrency, but data read by one program can be changed by other programs before the first one has finished processing. These isolation levels apply only to data locked at page locksize.

<u>Problems with Cursor Stabilty</u>

* Consider a program that is scanning a file and preparing a list of parts available before reserving those parts. It might assume that some part was available but this situation may change (because of these update of other transaction) before the program finishes, thereby causing errors in its output.

* Another possibility with cursor stability is that row may be processed twice or not at all. This can happen when program B changes the value of a field in the range already processed by program A into one that is yet to be processed by A. Or B may change a value from a range yet to be processed into one already processed thus causing A to miss the concerned record.

Repeatable read avoids these problems, but affects concurrency and runs the risk of lock promotion to table or tablespace level. If a subfile needs to be processed then the locks will be in place for a long time. Often cursor stability is used in conjunction with scheduling constrints that minimise the problem of double or missed processing.

How cncurrency is handled in production?

Every table is created with a timestamp field. Whenever a record is read, the timestamp is read into the host variable and before actual updation of the record, the record is read again and new timestamp value is compared with the old timestamp value. If they don't match (It is due to the updation of the field by other transaction.), the transaction will be rolled back and started again. By this mechanism, lost update problem is avoided to a greater percentage.

6.4 Intent Locks

Because DB2 supports locking at multiple levels of granularity, it uses the concept of intent locking at the tablespace level to reduce the amount of processing to manage locks.

Intent Share – The transaction intends to read but not update data pages and therefore take S locks against them. It will tolerate concurrent transactions taking S, IS, SIX, IX or U locks.

Intent Exclusive – The transaction intends to change data. It will tolerate other transactions taking an IS lock on the tablespace, which allows them to read data by taking S page locks at the page level.

Program A Locks	X	U	SIX	IX	S	IS
Program B Locks						
X	-	-	-	-	-	
U	-	-	-	-	Y	Y
SIX	-	-	-	-	-	Y
IX	-	-	-	Y	-	Y
S	-	Y	-	-	Y	Y
IS	-	Y	Y	Y	Y	Y
Page	-	-	S	S/X	S	S/X

Fig. Table and tablspace lock compatibility

6.5 Acquire and Release Parameteres

Page locks are always taken when they are needed, and released when commit or rollback takes place. At the tablspace level developers have some control (specified at bind time) over when locks are acquired or released.

If ACQUIRE parameter is specified with USE then the tablespace lock is imposed at the time the concerned SQL statement is executed. If the value is ALLOCATE then DB2 locks all the tablspaces when the plan is allocated.

RELASE value of COMMIT causes the locks to be released at COMMIT/ROLLBACK time. DEALLOCATE value for RELEASE keeps the locks till the thread is deallocated. Except ACQUIRE (ALLOCATE) and RELEASE (COMMIT), all other combinations are permitted.

ACQUIRE (USE) provides higher concurrency.

ACQUIRE (ALLOCATE) reduces concurrency. Also locks not needed may be taken. Program execution cannot begin till all tablespaces referenced by program are available. Reduces deadlocks.

ACQUIRE (USE) is the default. Also used by DB2 for referential intergrity checks, irrespective of user's choice. DB2 always follows the user's choice of RELEASE parameter.

ACQUIRE(USE), RELEASE(COMMIT) provides maximum concurrency. However, under thread reuse, when multiple users will execute a plan without having to restablish the thread, ACQUIRE(ALLOCATE), RELEASE (DEALLOCCATE), saves on processing.

WHEN TO COMMIT ?

Generally a COMMIT should be done before requesting terminal input. If this is not done then locks will be in force during operator delays. In some cases when the input data will be used in update, the old value may be part of the screen, and therefore should not be allowed to be changed. Therefore the locks should stay in place till the update is done. Another way to handle this is to commit before terminal input, but reread the row again after the input to test that no update has taken place. This reread can sometimes be avoided, by specifying WHERE clauses in the UPDATE that specify all the old values. If any change has taken place, the update will fail.

Generally COMMITS should be done after heavy update. Alternately, all updates should be places just before commit points.

Whenever an object is created or dropped, catalog tables and the DBD (Database Descriptor, which contains descriptive information about objects) are locked. This will prevent other users from creating objects or binding plans. Thus applications which are frequently executed should not generally create or drop objects dynamically. If this is done, a COMMIT should be done after object creation so that the locks can be released.

IF RELEASE (COMMIT) has been specified then after commit all locks will be released. Cursors which were in effect should be reopened, if they are to be reused. If the cursor has not fetched all its rows, then the last key value should be saved and used to set the correct restart point.

6.6. Explicit Lock Statements

LOCK TABLE statement within a program overrides all lock sizes and types that would otherwise be in effect for the specified tables. This can be used for programs that do heavy update activity. It is a good alternative to RR isolation level when a high percentage of rows will be updated.

LOCK TABLE (name) IN SHARE/EXCLUSIVE MODE

Table name is required. In nonsegmented tablespaces, the entire tablespace will be locked. In segmented tablespaces, only the specified table will be locked. The lock comes into effect only when the statement executes, even if ACQUIRE (ALLOCATE) has been specified. It will be released as specified in the RELEASE parameter at bind time.

LOCK TABLE is not permitted against views. It will give runtime error.

6.7. Deadlocks

Deadlocks are resolved by rolling back a transaction with fewest log records. Usually this is the one for which the lock was most recently taken. Inserts usually cause deadlocks because they involve locking of multiple indexes. More indexes imply more deadlocks. By default DB2 checks for deadlocks every 15 seconds. When a deadlock is encountered and resolved, SQL return code is sent -91 if ROLLBACK has been done, or –913 if the program is requested to issue the rollback. IRLM (Intersystem Resource Lock Manager) controls all locks. If a lock request cannot be serviced within 60 seconds a negative SQL code is returned.

6.8. Review Questions

- What are different level of locking supported in DB2 ?
- What does locksize option ANY do in DB2 ?
- What are NUMLKTS and NUMLKUS ?
- What are Share(S), Update(U), and Exclusive(X) locks ?
- What is an isolation level ?
- What is Cursor stability ?
- What is Repeatable Read ?
- What are the advantages and disadvantages of CS ?
- What are intent locks ?
- What is IS, IX, and SIX ?
- What are the parameters available with ACQUIRE and RELEASE options ?
- Can all combination of parameters be used with these options ?
- How can you explicitly lock a table in DB2 ?
- How does a Deadlock solved in DB2 ?

7

The Optimizer

7.1. Objectives

In this chapter, we will learn about

- The DB2 optimizer
- The EXPLAIN command
- How does EXPLAIN works

7.2. The Optimizer

The DB2 optimizer is integral to the operation of SQL statements. The optimizer, as its name implies, determines the optimal method of satisfying a SQL request. For example, consider the following statement.

```
SELECT      EMPNO, WORKDEPT, DEPTNAME
FROM        DSN8230,  EMP, DSN8230.DEPT
WHERE       DEPTNO = WORKDEPT :
```

This statement, whether embedded statically in an application program or executed dynamically, must be passed through the DB2 optimizer before execution. The optimizer parses the statement and determines the following :

- Which table must be accessed
- Which columns from those tables need to be returned
- Which columns participate in the SQL statement's predicates
- Whether there are any indexes for this combination of tables and columns
- What statistics are available in the DB2 catalog

Based on this information, the optimizer analyzes the possible access paths and chooses the best one for the given query. An access path is the navigation logic used by DB2 to access the requisite data. A tablespace scan using sequential prefetch is an example of a DB2 access path.

Based on models developed by IBM for estimating the cost of CPU and I/O time, the impact of uniform and nonuniform data distribution, and the state of tablespace and indexes, the optimizer usually arrives at a good estimate of the optimal access path.

Note : Many factors can cause the DB2 optimizer to choose the wrong access path, such as incorrect or outdated statistics in the DB2 catalog, an improper physical or logical database design, an improper use of SQL (for example, record-at-a-time processing), or bugs in the logic of the optimizer (occurs infrequently). In addition, the optimizer does not contain optimization logic for every combination and permutation of SQL statements.

7.3. The EXPLAIN command

The EXPLAIN command in DB2 describes the access path selected by the DB2 optimizer for SQL query. The information provided by EXPLAIN is invaluable for determining the following :

- The work DB2 does "behind the scenes" to satisfy a single SQL statement
- Whether DB2 is using available indexes, and, if indexes are used, how DB2 is using them
- The order in which DB2 tables are accessed to satisfy join criteria
- Whether a sort is required for the SQL statement

- Tablespace locking requirements for a statement

The performance of a SQL statement based on the access paths chosen

7.4. How Does EXPLAIN work

A single SQL statement, or a series of SQL statements in a package or plan, can be the subject of an EXPLAIN. When EXPLAIN is requested, the SQL statements are passed through the DB2 optimizer, and the access paths that DB2 chooses are externalized, in coded format, into a PLAN_TABLE.

The PLAN_TABLE is nothing more than a standard DB2 table that must be defined with predetermined columns, data types, and lengths.

Refer to your DB2 manual for all the columns and their datatypes in the PLAN_TABLE. You can create PLAN_TABLE as you create any other table in DB2. Note that the PLAN_TABLE will be created in the default database (DSNDB04) and STOGROUP(SYNDEFLT) in a DB2 generated tablespace, unless specified otherwise.

If a PLAN_TABLE already exists, the LIKE clause of CREATE TABLE can be used to create PLAN_TABLEs for individual users based on a master PLAN_TABLE. It is a good idea to have a PLAN_TABLE for every application programmer, for every individual owner of every production DB2 plan and every DBA and system programmer.

To EXPLAIN a single SQL statement, precede the SQL statement with the EXPLAIN command as follows :

```
EXPLAIN  ALL SET QUERYNO  =  integer  FOR
SQL  statement ;
```

It can be executed in the same way as any other SQL statement. QUERY NO, which can be set to any integer, is used for identification in the PLAN_TABLE. The other method of issuing an EXPLAIN as a part of the BIND command. By indicating EXPLAIN (YES) when building a package or a plan, DB2 externalizes the access paths chosen for all SQL statements in that DBRM to the PLAN_TABLE.

After issuing the EXPLAIN command on your statements, you can inspect the result in PLAN_TABLE. You can use a simple SQL query to retrieve this information. For example.

```
SELECT      QUERYNO,   QBLOCKNO,  APPLNAME, PROGNAME, PLANNO,
            METHOD, CREATOR, TNAME, TABNO ………………..
FROM        Ownerid. PLAN_TABLE
ORDER BY    APPLNAME,  COLLID ……………..
```

A common method of retrieving access path data from the PLAN_TABLE is to use QMF to format the results of a simple SELECT statement.

7.5. Review Questions

- What are the functions of an optimizer ?
- What does an EXPLAIN command do ?
- What is a PLAN_TABLE ?
- How does EXPLAIN command work ?

8

BINDING and Program Preparation

8.1. Objectives

The purpose of this chapter is to make you aware of

- The DB2 program preparation process
- BINDing
- DBRM
- PLANs
- Packages
- Collections
- Versions
- ReBIND

To prepare a DB2 program following tasks are performed

- Precompile the program
- Issue the BIND command
- Compile the program
- Link the program
- Run the program

8.2. Precompilation of the program

DB2 programs must be parsed and modified before normal compilation. This is accomplished by the DB2 precompiler. The precompiler performs the following functions :

- Searches for and expands DB2-related INCLUDE members
- Searches for SQL statements in the body of the program's source code
- Creates a modified version of the source program in which every SQL statement in the program is commented out and a CALL to the DB2 runtime interface module, along with applicable parameters, replaces each SQL statement
- Extracts all SQL statements from the program and places them in a database request module (DBRM)

  A DBRM is nothing more than a module containing SQL statements extracted from a source program by the DB2 precompiler. It is stored as a member of partitioned data set. It is not stored in the DB2 catalog or DB2 directory. Although there is a DB2 catalog table named SYSIBM.SYSDBRM, it does not contain the DBRMs. It consists of information about DBRMs that have been bound into application plans and packages. When a DBRM is bound into a plan, all of its SQL statements are placed into the SYSIBM.SYSSTMT DB2 catalog table. When a DBRM is bound into a package, all of its SQL statements are placed into the SYSIBM.SYSPACKSTMT table.

- Places a timestamp token in the modified source and the DBRM to ensure that these two items are inextricably tied
- Reports on the success or failure of the precompile process

8.3. Issue the BIND command

To draw an analogy, BIND command is a type of compiler for SQL statements. In general, BIND reads SQL statements from DBRMs and produces a mechanism to access data as directed by SQL statements being bound.

There are two types of BINDs: BIND PLAN and BIND PACKAGE.

BIND PLAN accepts as input one or more DBRMs produced from previous DB2 program precompilations, one or more packages produced from previous BIND PACKAGE commands, or a combination of DBRMs and package lists. The output of the BIND PLAN command is an application plan containing executable logic representing optimized access paths to the DB2 data. An application plan is executable only with a corresponding load module. Before you can run a DB2 program, regardless of environment, an application plan name must be specified.

The BIND PACKAGE command accepts as input a DBRM and produces a single package containing optimized access path logic. There can be only one DBRM per package. You can then bind packages into an application plan using the BIND PLAN command. A package is not executable, and can not be specified when a DB2 program is being run. You must bind a package into a plan before using it.

The BIND command :

- Reads the SQL statements in the DBRM and checks the syntax of those statements
- Checks that the DB2 tables and columns being accessed conform to the corresponding DB2 catalog information
- Performs authorization validation
- Optimizes the SQL statements into efficient accesspaths

DBRM/PLAN/PACKAGE/COLLECTION/VERSIONS

Previously, the two principal activities performed by BIND ("package bind" and "plan bind") were collapsed into a single process, in which all of the DBRMs for a given application were simultaneously compiled into optimized code and bound together into the required application plan. This scheme, however, suffered from a number of disadvantages.

- If an individual DBRM needed to be recompiled for any reason (e.g. because some index had been dropped), the entire plan had to be recompiled and rebound in their entirety.
- If multiple plans involved the same DBRM, that same DBRM had to be compiled multiple times-and, if that DBRM ever needed to be recompiled, then all relevant plans had to be recompiled and rebound in their entirety.
- Adding a new DBRM to an existing plan required(again) a recompilation and rebind of the entire plan.
- Partly as a consequence of the foregoing points, bind and (especially) rebind times were becoming unacceptably high in some DB2 installations, and availability was suffering as a result.

The package concept was introduced to remedy such deficiencies. A package is the compiled form of a DBRM, and a plan is essentially just a list of packages (for compatibility DBRMs are also allowed as input for the PLAN BIND) therefore ;

- If a given DBRM needs to be recompiled, all that has to be done is an appropriate package bind-it is not necessary to recompile the entire plan. Indeed, it may not be necessary to do a new plan bind to incorporate the new package neither.
- If multiple plans involve the same DBRM, that DBRM can now be compiled once, and the corresponding package referenced multiple times in multiple plan binds.

It is possible to add a new package to an existing plan without having to do a new plan bind, because of the concept of package collection. Each package belongs to exactly one collection. When a plan is bound, the input to the bind operation can be specified as any combination of packages and / or collections (and/or DBRMs). And specifying a collection is equivalent to specifying all of the packages in that collection including packages that may be added to the collection after the plan bind is done.

Collections also provide a means whereby a given plan can access different (but similar i.e. same name, same column name, number, and data types) tables at different times. This can be achieved by putting the same DBRM in two distinct packages and the packages into distinct collections, and if those two collections are then bound together into the same plan, then it is possible-by selecting the appropriate collection at run-time for the plan to access whichever of the two tables is desired.

Before the availability of packages, when programmers wanted to use an old version of a program, they were forced to rebind the program's plan using the correct DBRM. If the DBRM was unavailable, they had to repeat the entire program preparation process. When using packages, you can keep multiple versions of a single package that refer to different versions of the corresponding application program. This enables the programmer to use a previous incarnation of a program without rebinding.

You can specify a version identifier as a parameter to the DB2 precompiler. Other than this, versioning is automatic and requires no programmer or operator intervention.

- When a package is bound into a plan, all versions of that package are bound into the plan
- When a program is executed specifying that plan, DB2 checks the versions identifier of the link that is running and finds the appropriate package version in the plan
- If that version does not exist in the plan, the program will not run
- To use a previous version of the program, simply restore and run the load module.

8.4. Compile the program

The modified source data set produced by the DB2 precompiler must then be compiled. DB2 does not need to be operational in order to compile the program.

8.5. Link the program

The compiled source is then link-edited to an executable load module. The appropriate DB2 host language interface module must also be included by the link edit step. This interface module is based on the environment (TSO, CICS, or IMS/DC) in which the program will execute.

The output of the link edit step is an executable load module, which can then be run with a plan containing the program's DBRM or package. The link edit procedure does not require the services of DB2. Therefore, the DB2 subsystem can be inactive when your program is being link edited.

8.6. Run the program

After a program has been prepared, two separate physical components are produced a DB2 plan and a link edited load module. Neither is executable without the other. The plan contains the access path specifications for the SQL statements in the program. The load module contains the executable machine instructions for the host language statement sin the program.

When you run an application program containing SQL statements, you must specify the name of the plan that will be used. The plan name must include the DBRM that was produced by the precompile process that created the load module being run. It must be understood that a load module is forever tied to a specific DBRM. This is enforced by a timestamp token placed into both the DBRM and the modified source by the DB2 precompiler.

At execution time, DB2 checks that the tokens indicate the compatibility of the plan and the load module. If they do not match, DB2 does not allow the SQL statements in the program to be run. A-818 SQL code is returned for each SQL call attempted by the program.

If a load module is run outside the control of DB2, the program abends at the first SQL statement.

8.7 Re*BIND*ing

There can't be two types of rebinding of your plans and packages:

- Automatic
- Forced

Rebinding of plans and packages will be needed if DB2 has marked them as "invalid" or the user decides the BIND his package again.

If an object is dropped, DB2 examines the catalog to see which packages (if any) are dependent on that object. Finding such packages it marks them as "invalid". When the Runtime Supervisor retrieves such as package for execution, it sees the "invalid" marker, and therefore invokes BIND to reproduce a new package-I.e., to choose some different access strategy and to recompile the original SQL statements in accordance with that new strategy. Assuming the recompilation is successful, the new package effectively replaces the old one, and Runtime Supervisor continues with that new package.
For example, if an index has been dropped then the package using this index will be marked "invalid" and DB2 will perform BIND again on this package at runtime. This will be transparent to the user. If a table has been dropped then (re) BIND on the affected package will be successful only if another table with the same name and functionality has been recreated.

An user may opt for BINDing a package again. This can happen, for example, if a new index has been created and the user wants the optimizer to consider this index also while deciding about the access path. Creation of new index does not lead to automatic reBIND.

8.8 Review Questions

- What is the need of precompilation for a DB2 program?
- Why does a timestamp token is placed on modified source code and DBRM at precompilation time?
- What is the difference between BIND PLAN and BIND PACKAGE?
- What is a pacakage?
- What is collection?
- What is a version?
- When will DB2 go for automatic reBIND?

9

Security and Authorization

9.1 Objectives

This chapter discusses:

- Security features provided by DB2
- User indentification in DB2
- GRANT and REVOKE
- Bundled Privileges

Security means the protection of the data in the data base against unauthorised disclosure, alteration or destruction. DB2 provides a good degree of security. The specific data that can be protected in DB2 ranges from an entire table to a data value at a specific row-and-column position within such a table. A given user can have different access privileges on different objects (For example the privilege may be one of SELECT, UPDATE, USAGE etc.). Also different users can have different privileges on the same object.

Two independent features exist in DB2 to provide security.

1. The view mechanism, which can be used to hide sensitive data from unauthorised users.

2. The authorisation subsystem, which allows users having specific privileges selectively and dynamically to grant those privileges to other users, and subsequently revoke those privileges if desired.

In order to enforce security, DB2 performs the following,

(a) When GRANT or REVOKE statement is executed, it stores the privilege type, object, grantor and grantee in the catalog.

(b) Whenever user does operation for which security is executed, it stores the privilege type, object, grantor and grantee in the catalog.

9.2 USER IDENTIFICATION

Users are known in DB2 by an "authorisation identifier". It is the user's responsibility to identify himself by supplying the authorisation ID to log into the system. A group of related users is called a *user functional area.* In DB2, each functional area can also be given a authorisation ID. Therefore DB2 installation operates as follows.

1. Each individual user is assigned an authorisation ID to log in to the system and that is the *primary* ID for the user in question. All objects created by a particular user will belong to that user.

2. Each functional area in the organisation is also assigned an authorisation ID (called *secondary* ID). However, that ID is typically not given for login. User login using the *primary* ID for the user in question. All objects created by a particular user will belong to that user.

3. The SET CURRENT SQLID has the format

```
SET CURRENT SQLID = sqlid;
```

sqlid can be a string, host variable or USER.

9.3 GRANT and REVOKE

The view mechanism allows the database to be conceptually divided in various ways. Users can be granted privileges on these views using *GRANT* SQUL Statement. Thus view and grant mechanism together provides security on sensitive information. Granted privileges can revoked from the user by using *REVOKE* SQL statement.

In order to be able to perform any operation on any object, the user must hold appropriate privilege for the operation and the object in question. Otherwise, the operation will be rejected with an appropriate error message.

DB2 recognises a wide of privileges. Every privilege falls into one of the following classes:

- Table privileges: Privileges that apply to tables and views
- *Plan* and *Package* privileges: to use a given plan, package, or collection of packages
- *Database* privileges: Privilege to operations such as creation of table within a particular database
- *Use* privileges: Privileges for doing certain system resources, namely storage groups, tablespaces, and buffer pools, etc.,
- *System* privileges: Privileges for doing certain system wide operations such as the creation of a new database

There are certain *bundled* privileges, which serve as shorthand for collections of other privileges. For example, the *system administration* privilege (SYSSADM) is shorthand for the collection of all other privileges in the system.

Security mechanism works in DB2 as follows:

When DB2 is first installed, part of the installation process involves the designation of a user as the *system administrator* for DB2 system. So initially only one user exists who can do all jobs; and later on, he can grant privileges to other users.

Next, a user who creates an object is automatically given full privileges on that object.

9.3 GRANT

The general format of the statement is,

GRANT *privileges* [ON [type] objects] TO *users*;

Where *privileges* is a list of one or more privileges, separated by commas, or the phrase ALL PRIVILEGES; *users* is either a list of one or more authorisation Ids separated by commas or the special key word PUBLIC (meaning all users); *objects* is a list of names. Of one or more objects (all of the same type) separated by commas; and *type* indicates the type of the objects (if type is omitted, it is assumed to be TABLE).

Here are some examples:

Table privileges:

```
GRANT SECLECT ON TABLE SUPPLIER TO PRMOD;

GRANT SELECT, UPDATE (STATUS, CITY) ON TABLE SUPPLIER TO
SUBHA;

GRANT ALL PRIVILEGES ON TABLE S, P, SP TO GANESH, JOTHI;

GRANT SELECT ON TABLE P TO PUBLIC;
```

Package and Plan privileges:

```
GRANT EXECUTE ON PLAN PLAN_A TO NAMRATA;
```

Database privileges:

```
GRANT CREATETAB ON DATABASE FINANCE TO G.R.NAYAK;
```

Use privileges:

```
GRANT USE OF TABLSPACE FINANCE. TS1 TO RUCHI;
```

System privileges:

```
GRANT CREATEDBC TO DINESH;
```

9.3.2 REVOKE

If user U1 grants some privilege to some other user U2, user U1 can subsequently revoke the privilege from U2. This is done by REVOKE statement. The general format of REVOKE statement is:

REVOKE *privileges* [ON [type] objects] FROM *users;*

Revoking a given privilege form a given user causes all packages dependent on that privilege to be marked invalid and causes automatic rebind on the next invocation of each such package. This is similar to what happens when an object such as an index is dropped.

Here are some examples of the REVOKE statement:

REVOKE SELECT ON TABLE FROM SUBHASRI;

REVOKE UPDATE ON TABLES FROM PRAMOD;

REVOKE SYSADM FROM SANDEEP;

9.4 The GRANT Option

If user U1 has the authority to grant a privilege P to another user U2, then user U1 also has the authority to grant the privilege P to the user U2 "with the GRANT option". This means user U2 in turn has the authority to grant the privilege P to another user. For example:

User U1:

GRANT SELECT ON TABLE S TO U2 WITH GRANT OPTION;

USER U2:

GRANT SELECT ON TABLE TO U3 WITH GRANT OPTION;

And so on. If user U1 now issues

REVOKE SELECT ON TABLE S FROM U2;

Then the revocation will cascade (that is, U2's GRANT to U3, etc., will also be revoked automatically).

9.5 BUNDLED PRIVILEGES

- SYSDM

  SYSADM authority allows the holder to execute any operation that the system supports.

- SYSCTRL

  SYSCTRL (system control) authority allows the holder to execute any operation that the system supports, except for operations that access database contents (e.g., operations such as "create storage group" is allowed, but SQL data manipulations are not).

- DBADM

  DBADM (database administration) authority on a specific database allows the holder to execute any operation that the system supports on that database.

- DBCTRL

  DBCTRL (database control) authority on a specific database allows the holder to execute any operation that the system supports on the database, except for operations that access the database contents of the database (e.g., utility operations such as “recover database” are allowed, but allowed, but SQL DML is not allowed).

- DBMAINT

  DBMAINT (database maintenance) authority on the specific database allows the holder to execute read-only maintenance functions (such as the utility operation “image copy”) on that database.

- SYSOPR

  SYSOPR (system operator) authority allows the holder to carry out console operator functions on the system (such as starting and stopping system trace activities).

9.6 Review Questions

- What features are provided in DB2 for security?
- How is the user identification done in DB2?
- What are GRANT and REVOKE statements?
- What are the different types of privileges?
- What is bundled privileges?
- What is SA privileges?

10

DB2 Utilities

10.1 Objective

This chapter discuss some of the important DB2 utilities like

-CHECK
-COPY
-DIAGNOSE
-LOAD
-MERGECOPY
-MODIFY
-QUIESCE
-RECOVER
-REORG
-REPAIR
-REPORT
-RUNSTATS
-STOSPACE

<u>Utility</u>

CHECK The CHECK INDEX utility tests whether indexes are consistent with the data they index, and issues warning messages when an inconsistency is found.

The CHECK DATA utility checks table spaces for violations of referential constraints, and reports information about each violation it detects.

COPY Creates upto four image copies of a tablespace or a data set within a table space. There are two types of image copies.

a full image copy is a copy of all pages in a table space or data set.

an incremental image copy is a copy only of pages that have been modified since the last use of the COPY utility.

DIAGNOSE Generates information useful in diagnosing problems. DIAGNOSE is intented foruse under the direction of your IBM support center.

LOAD The LOAD utility loads data into one or more tables in a table sapce, replaces the contents of a single partition, or replaces the contents of an entire tablespace. The LOAD DATA statement describes the data to be loaded and provides information needed forallocating resources. The loaded data isprocessed by any edit or validation routine associated with the table, and any field procedure associated with any column of the table.

MERGECOPY Merges image copies produced by the COPY utility. The utility can merge several incremental copies of a table space to make one incremental copy, or it can merge incremental copies with a full image copy to make a new full image copy.

MODIFY The MODIFY RECOVERY utility deletes records ofunwanted copies from the SYSIBM.SYSCOPY catalog table and related records from the SYSIBM.SYSLGRNG directory table. You can remove records that were written before a specific date or you can remove records of a specific age. You can delete records for an entire table space or data set.

The MODIFY STATISTICS utility removes non-uniform distribution statistics from the SYSIBM.SYSFIELDS catalog table.

QUIESCE Establishes a quiesce point (the current log RBA) for a table space, or list of table space, and records itin the SYSIBM.SYSCOPY catalog table.

RECOVER Recovers data to the current state. You can also RECOVER to a prior copy or a specified log RBA. The largest unit of data recovery is the table space ; the smallest is the page. You can recover one table space, a list of table spaces, a data set, pages within an error range, ora single page. Data is recovered from image copies of a t able space and databae log change records. If the most recent full image copy data set isunusable, and there are previous image copy data set existing in the system, RECOVER uses the previous image copy data sets.

If the RECOVER utility cannot use the latest primary copied data set as a starting point for recovery, it attempts to use the backup copied data set, if one is available. If neither image copy is usable, it attemps to fall back to a previous recoverable point.

RECOVER also recovers multiple indexes over tables that reside in a common table space. Inexes are recovered in their entirety with a single invocation of RECOVER. Indexes can be recovered concurrently on the same table space.

REORG Reorganized a table space to improve access performance and reclaim fragmented space. In addition, the utility can reorganize a single partition of either a partitioned index ora partitioned table space. If you specify REORG UNLOAD ONLY or REORG UNLOAD PAUSE, the REORG utility unloads data in a format acceptable to the LOAD utility of the same DB2 table space.

REPAIR Repairs data. The data can be your own data, or data you should not normally access, such as space map pages and index entries.

Be extremely careful in using REPAIR. Improper use can damage the data even further.

REPORT Reaports the following information necessary for recovering a table space.

Recovery history from the SYSIBM.SYSCOPY catalog table Log ranges from SYSIBM. SYSLGRNG
Volume serial numbers where archive log data sets from
the BSDS reside
Volume serial numbers where the image copy data sets from
SYSCOPY reside
Names of all tablespace and tables in a table space set.

RUNSTATS Gathers summary information about the characteristics of the data in table spaces and indexes. This information is recorded in the DB2 catalog, and is used by DB2 to select access paths to data during the bind process. It is available to the database administrator for evaluating database design, and determining when table spaces or indexes should be reorganized.

STOSPACE Updates DB2 catalog columns that indicate how much space is allocated for storage groups and related table spaces and indexes.

10.2. Review Questions

- Which utility in DB2 gathers summary information about the characteristics of the data in table spaces and indexes ?
- Which utility gives you necessary information for recovering a table space ?
- Which utility reorganizes the table space ?
- Which utility tells you how much space is allocated for storage groups ?
- Which utility loads data into one or more tables in a table spaces ?
- Which utility recovers data to the current state ?
- Which utility creates image copies of a table space ?
- Which utility tests consistency of data and index ?
- Which utility generates information for diagnosing a problem ?
-

11
Facilities of DB2

11.1. Objectives

Objectives of this chapter is to briefly introduce you to the facilities of DB2 like

- Query Management Facility (QMF)
- Query By Example (QBE)
- QMF PROC
- Data Extract Facility (DXT)
- SPUFI - SQL Processor Using File Input
- DB21
- DCLGEN
- Defaults

DB2 provides several facilities suitable for different categories of end-users.

QMF	Query Management Facility
DXT	Data Extract Facility
SPUFI	SQL Process using file input
DB21	DB2 Interactive

11.2. Query Management Facility (QMF)

- a query tool of DB2
- allows end users to enter queries in either SQL or QBE (query-by-example)
- can produce a variety of reports and graphs from the results of these queries.
- QMF operates under TSO/ISPF control

A typical QMF session might go as follows

- The user constructs the query (SQL) in a QMF work area called QUERY
- The user issues RUN QUERY to execute the query stored in QUERY. The result is stored in another work area called DATA.
- QMF creates a default from for the QUERY which is kept in the work area called FORM
- The user views the report with DISPLAY REPORT
- The user may issue DISPLAY FORM to edit the form of the report in FORM
- The user then issues DISPLAY REPORT to prouce a revised report corresponding to the revised form.
- The Query need not be run again. The result has been kept in DATA and DISPLAY REPORT uses the current FORM to format and display the current DATA.
- The user can issue PRINT REPORT to obtain a hard copy of the report.
- The user can also issue DISPLAY CHART to display a chart or graph using the results of the query.
- The current query and associated FORM, DATA can be saved using SAVE QUERY command.

11.2.1. Query By Example

- A relational query language that is more user friendly
- All query operations are formulated by making appropriate entries in empty tables on the screen.
- Very much useful for the end users who have little or no data processing training
- The user issues DRAW < table name > command to display blank version of the table.

  the user constructs the query by selecting the desired columns and specifying the required conditions on the columns of this table.

11.2.2. QMF PROC

- QMF commands are grouped together to form a QMF procedure
- the set of QMF commands can be executed by a single command
- PROC can contain variable, values for which must be supplied when the procedure is executed.

  the procedure can be saved for later use using the SAVE PROC command and executed later using the RUN PROC command

11.3. Data Extract Facility (DXT)

- a product used to extract data from existing system files and convert it in DB2 compatible data.
- based on the information provided through a set of SQL like statements. DXT extracts and converts the original data formats into standard relations formats.
- the DXT scenario involves

  1 . The creation of DXT views of the data,
  2. The selection of the target table,
  3. The extract request which specifies WHERE search criteria for the data extrated.
  4. The data extract manager execution of the eligible extract requests.

11.4. SPUFI - SQL Processor Using File Input

- a convenient way of testing SQL statements before embedding them into the application program.
- is used mostly by application developers and database or system administrators
- the user can create a text file containing via SPUFI is

1. use SPUFI option on DB21 screen
2. on SPUFI menu, the user enters the input dataset name, output dataset, whether autocommit is YES/NO etc.
3. edits the SQL statement in the input file and submits
4. after execution of the SQL statement, the user can BROWSE the output

11.5. DB2 INTERACTIVE : DB21

Menu based interface to all of DB2's facilities.

<u>MENU</u>

SPUFI SQL Processing Using File Input, Interactive SQL
DCLGEN Host language data declaration generator
PROGRAM PREPARATION
PRECOMPILE
BIND/REBIND/FREE
RUN
DB2 COMMANDS
UTILITIES
DB21 DEFAULTS

11.5.1. DCLGEN

- this option of DB21 invokes the Declarations GENERATOR program.
- creates embeded SQL DECLARE TABLE statements and corresponding PL/1 or COBOL structure declarations from table descriptions in the catalog.
- the output from DCLGEN is stored as a member of a partitioned data set under user specified name
- the output of DCLGEN can be include into a host program by the following statement

```
EXEC SQL INCLUDE member
```

11.5.2. Defaults

Under defaults the user may choose things like language, lines per page of listing, message level, string delimiter, type of decimal point to be displayed and JCL card for the JCL generated by DB21.

12
Program Development

12.1. Objectives

This entire chapter is taken from the book 'DB2 for Application Programmers' by Pacifico Amarga Lim from page 60 to 132. Please read this chapter keeping in mind that no changes have been made to the original text.

This chapter will discuss

- Coding the COBOL program
- The program preparation process
- Select operations
- Update/Delete using the current cursor
- Update/Delete/Insert without using the current cursor

Part III SQL in Cobol Programs

10

Coding the Cobol Program

(Author's note : Wewill deal only with batch Cobol programs)

1. THE COBOL PROGRAM

The Cobol programs are written in the usual way, except that access to DB2 resoruces are done via SQL statements, not READ file-name, WRITE record-name, and so on - batch Cobol) or READ DATASET, WRITE DATASET, and so on (Cobol running under CICS/VS). The programmer therefore uses the same SELECT, INSERT, UPDATE, and DELETE statements we have previously discussed. In addition, the FETCH, OPEN, and CLOSE statements are also used.

1.1. Commit/Rollback for Cobol Programs

The program generally executes the COMMIT or ROLLBACK statement to commit or roll back information issued to update rows (INSERT, UPDATE, DELETE). Otherwise, all uncommitted updated are committed only at normal end of job-batch program) or normal task termination (CIS application programs).

See Section 3.1. for a complete discussion on commit / roll back.

1.2. The Cobol SQL Statement Delimiter

One minor difference between SQL statements in Cobol and those used in SPUFI is that the former are delimited by the "EXEC SQL" and "END-EXEC" statements. The semicolon (;) delimiter is not used. Thus:

```
E-EC      SQL
          SELECT  ENPDEPT,          SU-CEMPS4LARY
END-EXEC
```

Instead of

```
SELECT  EMPDEPT,  SUMCEMPS4LARY................'
```

2 CODING THE WORKING-STORAGE SECTION

In addition to the usual data areas defined in the WORKING-STORAGE section, the programmer includes specific data areas (see Section 2.1 and others following). The SQL statements to execute the logic (INSERT, UPDATE, FETCH and so on) are naturally coded in the Procedure division.

2.1. The SQL Communication Area (SQLCA)

The SQLCA is a data area that must be present in the WORKING-STORAGE section. On each execution of an SQL statement, DB2 returns certain information to the program concerning the success or failure of the statement. The programhas the option of using or disregarding any information returned.

There are three fields that are very important. They are :

1. SQLCODE. The return code. The values are :

   a. Zeroes. The SQL statement executed without any error, or with a minor error such as truncation (see SQL WARNO as follows).

   b. A positive value. The statement executed but with an exceptional condition. An example is + 100 for the various "no rows foun" conditions such as "end of table" when processing multiple rows via the FETCH statement, "no rows found" on an UPDATE or DELETE operation, and so on.

   c. A negative value. The statement executed but with an exceptional condition. An exampleis + 100 for the various "no rows found" conditions such as "end of table" when processing multiple rows via the FETCH statement, "no rows found" on an UPDATE or DELETE operation, and so on.

2. **SQLERRD (3)** The number of rows updated, inserted, or deleted by DB2.

3. SQLWARNO. If equal to "W", then at least one of the warning flag fields (SQL WARN1 to SQLWARNA) is set to a value. This field may contain a "W" even if SQLCODE is zeroes. See section 2.1.3.3.

2.1.1. Define SQLCA.

There are two ways to do this. The easiest way is to allow the precompiler to generate this area with the following statement in the WORKING-STORAGE section.

```
EXEC  SQL
      INCLUDE     SQLCA
END-EXEC.
```

Optionally, the user may use the Cobol COPY statement to copy the area from a source statement library.

2.1.2. The precompiler-generated SQLCA.

Figure 10.1 shows the typical SQLCA as generated by the precompiler.

```
01.     SQLCA
        05      SQLCAID                 PIC X  (8)
        05      SQLCABC                 PIC  S9 (9)  COMP.
        05      SQLCODE                 PIC  S9 (9)  COMP.
        05      SQLERRM.
                49      SQLERRML                PIC  S9 (4)         COMP.
                49      SQLERRMC                PIC X (70).
        05      SQLERRP                 PIC X (8)
        05      SQLERRD                 OCCURS  6 TIMES
                                                PIC  S9  (9)        COMP.

        05      SQLWARN.
                10      SQLWARN0                PIC X (1)
                10      SQLWARN1                PIC X (1)
                10      SQLWARN2                PIC X (1)
                10      SQLWARN3                PIC X (1)
                10      SQLWARN4                PIC X (1)
                10      SQLWARN5                PIC X (1)
                10      SQLWARN6                PIC X (1)
                10      SQLWARN7                PIC X (1)
                10      SQLWARN8                PIC X (1)
                10      SQLWARN9                PIC X (1)
                10      SQLWARNA                PIC X (1)
```

Figure 10.1 The SQLCA as generated by the precompiler

2.1.3. Using the SQLCA.

After the execution of each SQL statement, the programmer may investigate any field in the SQLCA. The most important fields are discussed in the following sections.

2.1.3.1. Using the SQLCODE field

There are two common uses of the SQLCODE field. First, it is used to determine if any error occured on the execution of an SQL statement ; checking is therefore usually done right after the execution of the SQL statement. For instance ;

```
EXEC.   SQL
        SELECT
END-EXEC
IF  SQLCODE EQUAL TO ZEROES
                OK,  Ccontinue
esle  execute  error routine
```

A second use is when processing the output table row by row via a cursor (FETCH statement) ; see the next chapter). A value of 100 signifies the "no record found" condition, which for the FETCH statement, means "end of rows".

2.1.3.2. The WHENEVER exceptional condition

The method of using SQL CODE as previously explained is the only way to process exceptional conditions if we prefer to treat each condition differently from the others. For most programs, this is the safest and best method, not subject to errors when a program is modified. In addition, the programmer may take any action (say, doing a CALL, which is not allowed in the following alternate method).

However, if we prefer to treat each condition the same way anywhere in the program, the programmer may actually just use the SQLCODE indirectly by simply specifying the WHENEVER exceptional condition, which may becoded anywhere in the Procedure Division but must execute ahead of the SQL statement we want to control.

The format is :

```
EXEC  SQL
        WHENEVER   (NOT FOUND / SQLWARNING / SQLERROR
        (GO TO  label / CONTINUE)

END-EXEC
```

The following apply :

1. The only actions allowed are GO TO label or CONTINUE (Continue execution)

2. Each NOT FOUND, SQL WARNING, or SQLERROR condition is paired with either a GO TO label or CONTINUE. The programmer may specify all three conditions in a single statement.

   a. NOT FOUND is true if SQLCODE is 100.

   b. SQLWARNING is true if SQLWARNO = ‘W’ or SQLCODE is positive, but not 100.

   c. SQLERROR is true if SQLCODE is negative

3. Multiple WHENEVER statements may be coded in the program. The latest one executed will then take effect.

4. After executing each SQL statement (except the WHENEVER statement), DB2 checks whether any WHENEVER statement is active ; if so, the action (GO TO label or CONTINUE) is done if the condition is true.

   An example is :

```
EXEC    SQL
        WHENEVER  NOT  FQUN
                GO TO Error-routine
END-EXEC
.       .       .       .
.       .       .       .
EXEC  SQL
        SELECT          .       .       .       .
END-EXEC
```

If the “NOT FOUND” condition is true for the SELECT statement (SQL-CODE = 100), the program then automatically executes “Error-routine”.

2.1.3.3. The meaning of SQL fields.

1. SQLCAID : Set to ‘SQLCA’ to identify the field in a dump.
2. SQLCABC : Length of SQLCA, set by DB2 when your program first uses the tructure.
3. SQLCODE : The SQL return code.

   a. A zero value denotes successful execution.

   b. A positive value denotes normal conditions but with a warning. This is stly the “no record foun” condition (SQLCOE value of 100).

   c. A negative value denotes an abnormal condition such that the statement was unsuccessful. For example, a value of 803 means that on an insert operation, the value of a column already exists fora column that was defined with an index defined as UNIQUE.

4. SQLERRC : Error message that usually goes with the SQLCODE.

5. SQLERRP : If the SQLCODE is negative (abnormal condition) this contains the name of the DB2 routine that discovered the error. This field is used with SQLERRD in debugging.

6. SQLERRD : This describes the current internal state of DB2 SQLERRD # 3 is significant for it specifies the number of rows processed on an INSERT, UPDATE, and DELETE operation.

7. SQLWARN : Characters that warn of various conditions encountered during the processing on your statement. Alternately, specific warnings may be indicated by positive value in the SQLCODE field.

8. SQLWARNO : Has the value 'W' if one of the following warning character is set to 'W' and thus serves as a quick test for the existence of any warning. It has a value of 'S' (severe) if SQLWARN6 is let to 'S'.

9. SQLWARN1 : Has the value of 'W' if at least one column's value was truncated in the host variable. This always happens when truncating character data items but may or may not happen for numeric items.

10. SQLWARN 2 : Has the value of 'W' if null values were ignored in the computation of a built-in function (AVG, SUM, and so on. This value is set only during preprocessing, never at run time.

11. SQLWARN 3 : Has the value of 'W' if the number of items in the SELECT list is not equal to the number of target variables in the INTO clause.

12. SQLWARN 4 : Has the value of 'W' if an UPATE or DELETE statement has been used without a WHERE clause. You should verify that the update or delete was intended unconditionally on the entire table. This value is set only during preprocessing, never at run time.

13. SQLWARN 5 : Has the value of 'W' if a WHERE clause associated with a SELECT statement has exceeded a DB2 internal limitation.

14. SQLWARN 6 : Has the value of 'W' if the last SQL statement executed caused DB2 to terminate a logical unit of work. This is set to 'S' when DB2 issues an SQLCODE that is severe (-805, - 86, and so on).

15. SQL WARN 7 : Reserved for DB2 use.

16. SQLWARN 8 : Has the value of 'W' if a statement has been disqualified for blocking for reasons other than storage.

17. SQLWARN 9 : Has the value of 'W' if blocking was cancelled for a cursor because of insufficient storage in the user partition.

18. SQLWARNA : Has the value of 'W' if blocking was cancelled for a cursor because a blocking factor of atleast two rows could not be maintained.

19. SQLTEXT : Reserved for DB2 use.

2.2. Define the Tables / Views to be Used

Although optional, the programmer should define the table(s) or views to be used. It both serves as a documentation, as well as provides an extra measure of control since during the precompile step (see section 4,) DB2 will verify it against the names used in SQL statements.

There are two options in defining tables.

2.2.1. Define the tables/views via DCLGEN.

The DCLGEN function is option 2 of the DB2 interactive (DB21) Primary Option Menu (see Fig 4.2). It may also be invoked directly in TSO via the TSO command "DSN" with the subcommand "DCLGEN". Lastly, it may be run in batch.

The two advantages of using DCLGEN over the DECLARE TABLE statement (see section 2.2.2) is that it uses the table definition in the DB2 catalog and there are therefore no possible coding errors. In addition, the corresponding Cobol record description (see host variable, section 2.3), is also generated.

Note the following :

1. The SOURCE TABLE NAME entry is the table or view from which we generate the DECLARE statement and Cobol data-names.

2. DATA SET NAME contains the output of DCLGEN. The library name is used in the Precompile panel (Fig. 1.1.6) and Compile / Link / Run panel (Fig.11.8) of Chapter 11. The member name is the one specified in the EXEC SQL INCLUDE statement section 2.2.2) that brings the declaration and Cobol data-names into the Working-storage section.

3. The DATA SET PASSWORD entry is used if the output DCLGEN is password protected.

4. ACTION is ignored for sequential data sets. For partitioned data sets, REPLACE will replace an old version or create a new one ; ADD creates a new version.

```
=====>
Enter table name for which declarations are required :
1.   SOURCE TABLE NAME   ------  'XLIM.EMPTABLE'

Enter destination data set   :                    (Can be sequential or
partitioned

2.    DATA  SET  NAME   :              'XLIM.DB2.COBOL  (EMPTABDL)
3.    DATA SET PASSWORD:                             (if  password pretected)

Enter options as desired :
4.    ACTION   ...................  ADD           (ADD new or  REPLACE old declaration)

5.    COLUMN  LABEL                       NO
(enter YES for column label)
```

Figure 10.2

The DCLGEN panel

5. COLUMN LABEL is normally NO.

6. STRUCTURE NAME (up to 31 characters) specifies the generated data structure name (01-level group item) ; if missing, DB2 will generate the table or view name prefixed with "DCL". In this case, our output would become "01 DECLEMPTABLE".

7. FIELD NAME PREFIX (up to 28 characters) specifies the prefix for the fields in the generated output (which correspond to table or view columns). The fields will then be generated as prefix 001, prefix 002, and so on. The prefix is usually not specified so the field names will be instead identical to the column names in the table or view.

2.2.2. DECLARE TABLE Statement generated by DCLGEN.

2.2.3. Including the DCLGEN output.

To bring the DECLARE TABLE output into the appliction program, the programmer codes in the working-storage section :

```
EXEC     SQL
         INCLUDE   EMPTABDL
END-EXEC
```

```
********************************************************************************
***
*       DCLGEN   TABLE   (XLIM.EMPTABLE)

*                          LIBRARY (XLIM.DB2.COBOL  (EMPTABDL)

*                          ACTION  (REPLACE)

*                        APOST

*           IS  THE  DCLGEN   COMMAND  THAT  MADE  THE   FOLLOWING
STATEMENTS.
```

Figure 10.2

The DCLGEN panel

2.2.4. Coding the DECLARE TABLE statement

As we have seen, the DECLARE statement is one of the outputs of the DCLGEN procedure (the other is the Cobol record description). If DCLGEN is not used, the DECLARE statement may be coded by the programmer and the format is similar to one for creating the tables or views. The format is :

```
EXEC   SQL    DECL4RE           table-name  TABLE
       (column -name1           data-type
       column -name 2           data-type
```

END-EXEC Note that the literal "TABLE" is always used, whether we are defining a table or view. Unlike using DCLGEN, this method does not generate the record definition of the host variables.

2.3. Define Host Variables

Host variables are those that are usedin the application program. Naturally, they are defined according to the rules of the programming language (Cobol, for instance). Those for the tables or views are either automatically generated in DCLGEN or specifically defined by the programmer.

2.3.1. Table row description generated by DCLGEN.

When DCLGEN is notused, the programmer must code the host varibles for the tables and views, a processthat is error prone. Using DCLGEN is thus better since the DB2 catalog is used.

2.3.2. Using host variables in SQL Statements.

When used in SQL statements, Cobol data-names must be prefixed with a colon (:). It goes without saying that they do not have the colon prefix when used in regular Cobol statements (meaning non-SQL statements.

```
*************************************************************************
***
               COBOL  DECLARATION FOR TABLE XLIM EMPTABLE
*************************************************************************
***
            01             DCLEMPTABLE
                         10             EMP NO
             PIC X (5)
                         10             EMPNAME
          PIC X (30)
                         10            EMPDEPT
        PIC XXX           10             EMPSALARY
```

Figure 10.2

Examples (as used in SQL clauses) are :

```
1)      WHERE  EMPNO      =        EMP NO
2)      SET  EMPSAL4RY    =        EMPSALARY
3)      VALUES  C:EMPNQ =          :EMPNAME)
```

Note that in the three previous examples, the ones with the colon prefix are the Cobol data-names generated by DCLGEN and those without prefixes are the original table column names.

2.4. The Indicator Variables

An indicator variable is defined in working storage for a column that can have a NULL value. On a SELECT or FETCH statement, DB2 will place a negative value in the variable if the column has a NULL value. The corresponding Cobol data-name is not changed and therefore retains whatever value it received from a previous SELECT or FETCH statement. On an UPDATE or INSERT statement, the programmer places a negative value in the variable to indicate to DB2 that a NULL value is to be used for the column. In this case, the column cannot be defined in the table as NOT NULL.

Unless an indicator variable is used for a field, on an input operation (SELECT, FETCH) there is an SQL error if the field does contain NULLs; on an output operation (INSERT, UPDATE). DB2 will not be able to insert or update NULLs into the field.

2.4.1. Defining the indicator variable

Each indicator variable is a half-word integer (PIC S9(4) COMP). If defined as an array (OCCURS clause), then it may be used for a list of columns with the first occurrence of the indicator variable corresponding to the first column in that list. (For an example, see Section 2.4.2, item 2).

Examples of indicator variables follow :

```
01      INDICATOR-VARIABLES          PIC  S9CA         COMP.
        05      EMPSALARY-IND                 PIC  S9C4         COMP OCCURS 3.
        05      EMPTABLE-IND
```

2.4.2. Using indicator variables on “read” (SELECT, FETCH)

An example of using a nonarray indicator variable is:

```
        EXEC    SQL
                SELECT  EMPNAME,  EMPSALARY
                        INTO : EMPNAME,  : EMPSALARY:EMPSALARY-IND
                        FROM  XLIM.EMPTABLE
```

```
                    WHERE           EMP NO   =       EMP NO
        END-EXEC
```

Note the following :

1. The indicator variable (EMPSALARY-IND) is immediately placed (prefixed with a colon) after the corresponding variable (here the Cobol data-name EMPSALARY) that corresponds to the column.

   a. EMPSALARY-IN will have a negative value if the column EMPSALARY
   has a NULL value. The programmer may or may not use this fact (depending on the contents of other fields) to process the row.

2. Other fields may or may not also have inicator variables. An example of using an array indicator variable is :

```
EXEC    SQL
        SEELECT  EMP NO,  EMPNAME,  EMPDEPT,  EMPSALARY
                INTO : DCLEMPTABLE  :  EMPTABLE-IND
                FROM XLIM.EMPTABLE
        WHERE  EMPNO  =  :SEARCH-DEPT
END - EXEC
```

Note that a negative value in EMPTABLE-IN(1) means a NULL value in the first data-name under DCLEMPTABLE ; a negative value in EMPTABLE-IND(2) means a NULL value in the second data-name under DCLEMPTABLE, and so on.

2.4.3. Using indicator variables on UPDATE and INSERT.

On an UPDATE or INSERT statement, it is the programmer who indicates (by placing a negative value in the appropriate indicator variable) that we will use nulls for a column used in the UPDATE or INSERT statement. Naturally, such a column must not be defined with the NOT NULL attribute.

An example of using an indicator variable in an insert operation is :

```
***** set EMPNO,  EMPNAME,  EMPSAL4RY  to the correct values.
        IF  condition-1
                MOVE 0  TO  EMPSALARY - IND
        ELSE
        MOVE -   1   TO  EMPSALARY-IND
        EXEC  SQL
                INSERT  INTO  XLIM.EMPTABLE
                (:EMPNO,  EMPNAME,  EMPSALARY)
                VALUES   (: EMPNO,   :EMPNAME,   EMPSALARY:   EMPSALARY-IND)
END - EXEC.
```

If condition-1 is true, we set EMPSALARY-IND to 0 and the value in the EMPSALARY data-name (which the programmer should previously set to the correct numeric value) will be used. If condition-1 is false, we set EMPSALARY-IND to 1 and DB2 will insert nulls for EMPSALARY. Note that in the latter case, if EMPSALARY is defined with NOT NULL WITH DEFAULT, the salary value as finally inserted in the rowis zeroes.

3 CODING THE PROCEDURE DIVISION

The same SELECT, UPDATE, INSERT and DELETE statements we have seen before are coded where needed right in the procedure division, and interspersed with regular Cobol statements. In addition, we will learn the use of the FETCH statement when processing rows in an application program.

3.1. COMMIT / ROLL BACK and the Unit of Recovery

A unit of recovery is that portion of processing where the program tells DB2 that all updates (successful INSERT, UPDATE and DELETE statements) in that unit should be either committed (to be written to the physical device) or rolled back (disregarded). Because DB2 takes log records for a unit of recovery, a user's decision to commit or roll back will be done, even if the action is not completed because of abnormal termination of processing (say, the DB2 subsystem crashes or your program abends). In this case, it is eventually done either by DB2 (if only the program bombs or during emergency restart of DB2 or the system.

A unit of recovery has both a start and an end. At end, DB2 will do a physical write of updated rows if the end is triggered with the SQL COMMIT statement or normal end of job (for a batch program or CICS SYNCPOINT command or normal task termination (for a CICS appliction program).

3.1.1. Choosing the unit of recovery

In some cases, choosing the unit of recovery is a user judgement call. For instance, one may choose to wait for every 1 rows updated before they are committed as final if the user judges it to offer the best tradeoff between efficiency and allowing greater concurrency. More commits incur a greater overhead processing, but allow more users to share the data.

However, certain processing requires that a series of updates be within a unit of recovery. For instance, in an order entry application, a single customer order will insert one row in the Purchase order table, insert as many line-item rows as there are in the order, then update the same number of rows (the inventory items) in the Inventory table.

The update of these 3 tables must be synchronized as one unit of recovery since if one is not completed (say, due to a program bomb or power failure), we want all inserts and updated to be rolled back (not committed.

3.1.2. Start of unit of recovery

The unit of recovery starts on the following :

1. At program start (batch program) or task start (CICS application program).

2. On the SQL COMMIT statement or CICS SYNCPOINT command. This starts another unit of recovery while also ending the current one.

3. On the SQL ROLLBACK statement or CICS SYNCPOINT ROLLBACK command. This starts another unit of recovery while also ending the current one.

3.1.3. End of unit of recovery

The unit of recovery ends on the following :

1. On the SQL COMMIT statement or CICS SYNCPOINT command. This requests B2 to physically write all changes to data done since the start of this unit of recovery. This also starts a newunit of recovery.

2. On the SQL ROLLBACK statement or CICS SYNCPOINT ROLLBACK command. This requests DB2 to roll back (therefore do not physically write, that is, ignore) all changes to data done since the start of this unit of recovery. This also starts a new unit of recovery.

3. When the program normally terminates (batch program) or task normally terminates (CICS application program). All uncommitted changes within this unit of recovery are also committed (not rolled back.

3.1.4. Incomplete unit of recovery

By definition, an incomplete unit of recovery (that is, with no end) is not really a unit of recovery. Thus, if the program bombs, all uncommitted updates are lost (automatically rolled back).

3.2. The Commit Process

We have learned that under SPUR, the SPUR main panel will allow the user to select an automatic commit or roll back or efer that decision to a later time (at which point he or she may commit or roll back the changes).

For application programs, if the program doesnot so specify, DB2 will not commit any update, unless the batch application program normally ends or the CICS task normally terminates.

Note that only successful INSERT, UPDATE, and DELETE statements (SQL-CODE zeroes) will have the corresponding rows included in theunit of recovery.

3.2.1. The format of the COMMIT statement

The format of the statement is :

```
EXEC    SQL
        COMMIT
END-EXEC
```

3.2.2. When to commit

Efficiency, the reduction of deadlocks, and data integrity are three reasons for the timing of the commit process.

3.2.2.1. Commit for greater efficiency

Since DB2 implicitly secures a lock on uncommitted updates (this preventing other programs from accessing data in those pages), prolonging the commit process results in other programs having to wait longer to access DB2 data. For a heavily-used table, this promotes lower concurrency (lower degree of sharing among many users), hence less overall efficiency.

It is therefore important that the programmer does the commit or roll back as soon as feasible, especially for heavily-used tables, to allow a larger level of concurrency. DB2's data "blocks" are called pages and when updating single tables (especially if very large), we may issue the commit for each row if we are processing randomly; we may wait for several rows if we are processing in sequence. The physical rewrite of pages (which results from the commit) incurs some overhead and doing a single rewrite for multiple rows that are upated gains some efficiency.

If we are updating more than one table, and they must be synchronized, we may do the commit when we have reached a consistent unit of recovery. For instnce, in an Order Entry application, once we have updated both the Order table and ordered-item table (to enter the order) and the Inventory table (to reduce inventory by the items being ordered), we may issue the SQL COMMIT statement.

3.2.2.2. Commit to reduce deadlocks

The problem of deadlocks is discussed in more detail on page 106. For now, let it suffice to say that the more pages users hold on to, the greater the chance that two or more users cannot continue processing because they need data on pages held by somebody else. The timely commit of updates therefore reduces this problem.

3.2.2.3. Commit for data integrity

If the system goes down or the program bombs, any uncommitted update is lost. Doing a commit therefore makes sure that updated rows are made final on physical device.

3.2.3. DB2 action on commit

Any user request for commit only requests DB2 to commit all changes within that unit of recovery. DB2 writes log records to guarantee that even if the system crashes or the program bombs before any physical write is done, the updates are eventually written out to the physical device.

DB2 does not immediately do a physical write since pages of updated data in the buffers are written out only at checkpoint intervals, which are specified at installation time. An exception is if the buffer pool is full and a user needs additional data to be brought to the buffer, DB2 will use the least frequently used page, writing it out first if it contains uncommitted updates. This actually allows more pages to be already in the buffer when users request data.

3.3. The ROLLBACK Statement

The SQL ROLLBACK statement rolls back all changes to rows, instead of committing them. In the same manner as the SQL COMMIT statement, it tells DB2 that the program is finished with the uncommitted rows and that changes in the current unit of recovery should be ignored. As far as improving concurrency, it has the same effect as the COMMIT statement.

3.3.1. The format of the ROLLBACK Statement

The format of the statement is :

```
EXEC  SQL
        ROLLBACK
END-EXEC
```

3.4. CICS COMMIT/ROLLBACK

The SQL COMMIT and ROLLBACK statements are not valid in CICS. The programmer instead uses the CICS SYNCPOINT and SYNCPOINT ROLLBACK commands.

4. BASIC PROGRAM SKELETON

Actual program examples are shown in Chapters 12, 13 and 14 for the moment, we will just show how they will look like in Figures 10.5 and 10.6.

The basic program skeleton is :
The items, in the following order, are :

1. The INCLUDE for the DCLGEN output
2. All needed indicator variables
3. The INCLUDE for the SQLCA data block

```
WORKING-STORAGE  SECTION
        ......
        EXEC  SQL
                INCLUDE  member-name  (output  of DCLGEN)
        END-EXEC
        ****    Code  indicate  variables  (if  any)
        EXEC  SQL
                INCLUDE  SQLCA
                END-EXEC.

        PROCEDURE  DIVISION

****    Here set the EMPNO  dataname to the correct value.
****    For instance,  the value may come from a file.

        EXEC    SQL
                SELECT    EMPNAME,  EMPSALARY
                        I  NTO :  EMPNAME     :  EMP SALARY
                                FROM  XLIM.  EMPTABLE
                                        WHERE  EMP NO  =    EMP NO

        END-EXEC.
        IF  SQLCODE  EQUAL TO ZEROS
                MOVE       EMPNAME  TO  .......
        ELSE  error  routine.
```

Figure 10.5 Basic program skeleton

4. The setting of the EMP NO data-name to the correct value.

5. The SELECT statement to select a specific row and bring it to the Cobol datanames EMPNAME and EMPSALARY.

6. The processing of the data if SQLCODE is zeroes.

5. PROGRAM EXAMPLE (AS CODED)

Figure 10.6 is an example of a coded program (with the CLGEN output printed out and with certainCobol program headers omitted). Here we display the employee number and name of employees in the Employee table based on employee numbers keyed in an "in-line" card file (//CARDIN D* JCL entry).

The items, in the following order are :

1. The SELECT statement for the CARDIN file
2. The DCLGEN output, both the DECLARE TABLE statement and the table row descriptions.
3. The initial house-keeping routines such as the opening of the card file and the code for the main loop.
4. The SELECT statement to bring the data of one row (assuming EMPNO is define with a unique index) into the EMPNAME data-name.

5. The processing of the data if SQLCODE is zeroes.

```
        ........
        SELECT EMPLOYEE-CARDIN  ASSIGN TO CARDIN
WORKING-STORAGESECTION
**********************************************************************
*       DCLGEN TABLE  (XLIN,EMPTABLE)
*       LIBRARY  (XLIN.DB2.COBOL(EMPTABDL)
*       ACTION  (REPLACE)
*       APOST
*
*       IS THE DCLGEN CONMAND THAT MADE THE FOLLOWING STATEMENTS
********************************************************************************
        EXEC SQL DECLARE  XLIM.EMPTABLE  TABLE
        (EMP  NO                    CHAR (5)  NOT NULL
        EMPNAME                     CHAR (30) NOT NULL WITH DEFAULT
        EMPDEPT                     CHAR  (3)  NUT NULL WITH DEFAULT
        EMP SALARY                  DECIMAL  7,0)  NULL
        ) END-EXEC.
**********************************************************************************
* COBOL  DECLARATION FOR TABLE XLIM.EMPTABLE
**********************************************************************************
01.     DCLEMPTABLE
        10      EMPNO               PIC  X (5)
        10      EMPNAME             PIC X (30)
        10      EMPDEPT             PIC XXX
***********************************************************************************
*  THE NUMBER OF COLUMNS  DESCRIBEDBY THJS DECLARATIONIS  4
***********************************************************************************
        01      W005 - EMPLOYEE  - NUMBER           PIC  X  (5)
        01      W005 - END -OF -FILE-SE             PIC X.
                88 W005 - CARDIAN-HAS -ENDED        VALUE 'Y'
EXEC SQL
        INCLUDE SQLCA
END -EXEC.
PROCEDURE  DIVISION
```

```
    OPEN  INPUT EMPLOYEE -CARDIN
    MOVE ‘N’ TO W005 - END - OF - FILE - SW.
    PERFORM C060 - READ-EMPLOYEE-CARDIN.
    PERFORMN  C040-PROCESS-ONE -EMPLOYEE
                UNTL W005-CARDIN - HAS-ENDED.
    CLOSE EMPLOYEEE- CARDIN.
    GOBACK.

C040-PROCESS-ONE-EMPLOYEE.
        EXEC SQL
            SELECT EMPNAME
                INTO :EMPNAME
                FROM XLIMEMPTABLE
                WHERE EMPO = :W005- EMPLOYEE-NUMBER
        END-EXEC.
        IF SQLCODE EQUAL TO ZEROES
            DISPLAY W005 -EMPLOYEE -NUMBER  ‘EMPNAME’
        ELSE DISPLAY W005-EMPLOYEE -NUMBER ‘ MAJOR ERROR’
        PERFORMC060 -READ-EMPLOYEE - CARDIAN.

C060 - READ - EMPLIYEE - CARDIN
        READ EMPLOYEE-CARDIN INTO W005- EMPLOYEE - NUMBER
            AT END ,MOE ‘Y’ TO W005 - END OF -FILE SW
```

FIGURE 10.6 pROGRAM EXAMPLE (AS CODED)

THE PROGRAM PREPARATION PROCESS

1 STEPS IN PROGRAM PREPARATION

Figure 11.1 showsthe five steps in preparing anapplcation program.

The five teps acomplish the following:

1. The CICS tanslaton step is the only optional step. It is only done for command-level CICS application programs and it generates two outputs.

    a. the translation listing in userid.temp.exlist.

    b. the translated source program in userlid.temp.cicsm.

2. The PRECOMPILE stepchecks that the SQL statements are free of syntax errors. in addition, if there is a DECLARE TABLE statement (automatically generated by DCLGEN), the table nzmes and column names in the statement are verified. PRECOMPILE generates three outputs.

    a. Theprecompile listing in useri.temp.plist.

    b. The database request moule (DBRM), which contains a parse tree version fthe SQL atements in a program. this will be the input tothe BIND process Step 3.

Figure 11.1 Preparation an application programm

c. The modified source program in userid.temp.cobol, but with each SQL statement replaced mostly by a series of Call Statements.
Useri.temp.cobol is the input to the compile in Step 4.

3. The BIND process is DB2's version of a link-et. It readsone or more database request modules (see Section 1.2) and "binds" the,m togethere into one applcation plan. This process accesses the DB2 catalog to verify table information, access authorization, and so on. Specifically, it does the following:

a. It catches the SQL statements for valid table, view, and column names 78.

b. Unless this is postponed until program execution (VALIDATE(RUN) option), it checks if the person doingthe bing is authorized to usethe resources (tables, and so on) named in the SQL statements.

c. It determined via the Optimizer module how the data will be accessed, including whether top use any existing index or not.

d. It converts the DBRMs into one application plan. These are control structures used by DB2 during program execution. The application plan is stored in the DB2 catalog. In DB2, it is the plan, not the corresponding program(s) that a user is authorized to execute.

4. The compile process, which generates the usual output, including the listing in userid.temp.list.

5. The link-edit process.

1.1 Significance of the BIND Process.

The BIND process is done before the execution of the corresponding application. Since data access os resolved during BIND, not at program execution, this results in greater efficiency. Note that in SPUFI, data access is resolved during the execution of SQL statements, thus making it inherently less efficient than the corresponding application program.

In addition, checking for authorization during the BIND process, as opposed to postponding it until program execution, enhances efficiency because this overhead is not one during the program run.

1.2 How Many Programs (DBRM's) per plan?

At PRECOMPILE, each program generates a single database request module (DBRM). In turn, at BIND (as we said, DB2, version of link-edit), one or more DBRMs generate a single plan.

Unless otherwise required, (such as when a main program an several subprograms - if the latter have SQLstatements - form one run unit), the user should use only one program per plan. This allows for maximum flexibility. Unnecessarily putting mulitple programs (DBRMs) in a single planmakes that plan more prone to being invalidated, since a modification to a single program (change to tables, hence to DCLGEN, output, delection of indexes,and so on) may invalidte the corresponding DBRM. This in turn invalidates the wjole paln. When this happens, all programs within that plan cannot be executed, including those that were not modified in the first place. However, asexplained later, DB2 tries to help the user with this problem.

1.3. DB2's Automatic Rebind Of Invalidated Plan

The user may do the rebind of any previous plan, incluing an invalidated plan. If not, DB2 will attempt an automatic rebnd of aninvaidated plan without any user intervention, at the time of the plan is next run. however, if the attempt fails (for instance, a program accesses a column that since the last bind has been deleted fromthe table), DB2, will inform the user and change the plan statusfrom invalidated to deactivated. The plan cannot then be executed until the user makes the corrections to programs, tables, and so on.

2. USING B21 TO PREPARE ANO TEST PROGRAMS

Once the programmer has coded the program (using TSO/ISPF), he or she may then execute the steps in Figure, 11.1. There are two ways to do this:

1. One way is to directly submit JCL statementsin TSO background. These statements reside in a data set previously saved by the programmer. See Section 8 later in this letter in this chapter.

2. Another way is to use the DB2 Interactive (DB21) facility, which later automatically executes the program in TSO foreground or generates the JCL statements that are submitted to run in TSO background. Figure 11.2 shows that PROGRAM PREPARATION is one of the DB21 functions.

Note that the interactive panels (for PRECOMPILE, BIND, RUN an others) *we will learn to use not implement Figure 11.1 nteractively. They merely generate control information that implements Figure 11.1 when later run in* TSO background or foreground.

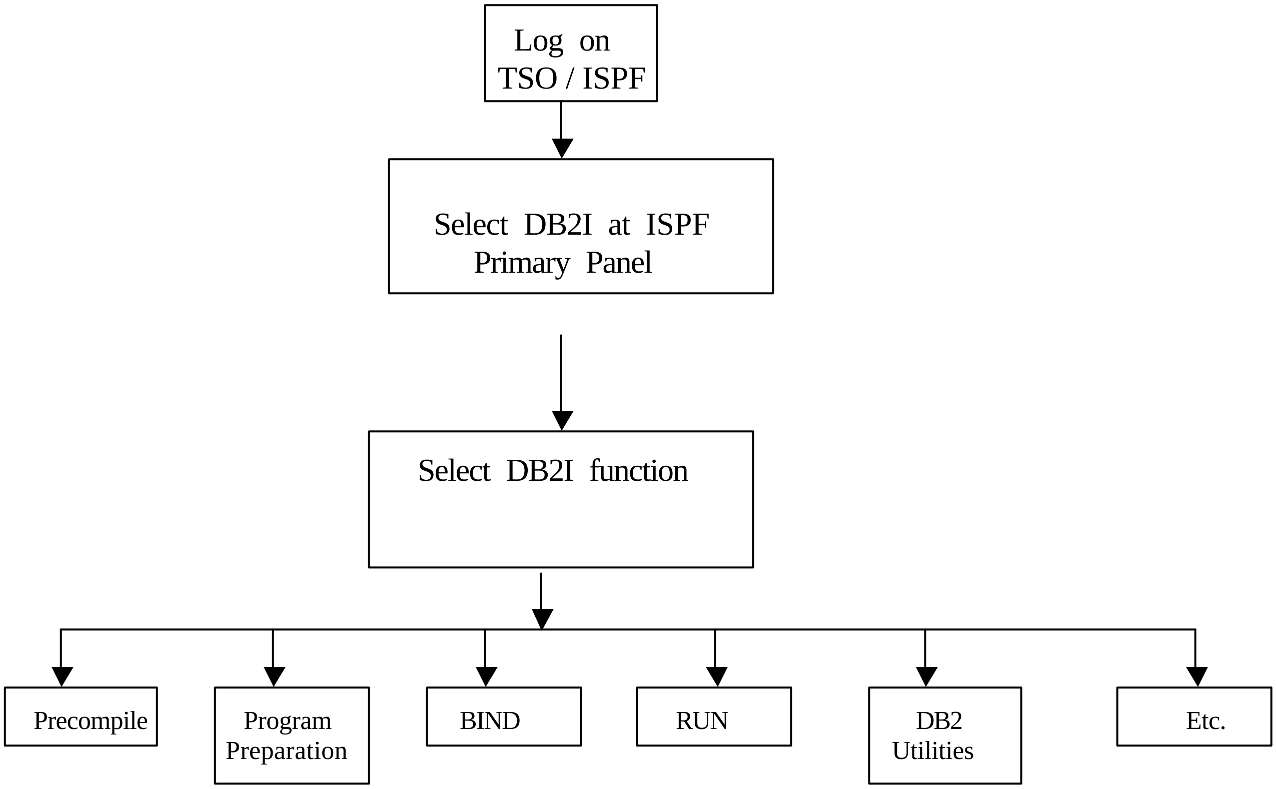


Figure 11.2 The various DB21 functions

2.1. Saving the Generated JCL Statements

If the programmer chooses "BACKGROUND" or "EDITJCL" in entry 3 of Figure 11.5, DB2 generates JCL statements in userid.temp.cntl, which will remain until TSO logoff. For option "EDITJCL", the programmer may include additional JCL statements before submitting it for execution (needed if the program has batch files). DB2 automatically saves userid.temp.cntl when the programmer leaves the panel.

An even better option is to copy userid.temp.cntl into a separate data set, since the former is lost at TSO logoff. This is shown in Section 8 later in this chapter. This saves time and effort since there is now no need to use the DB21 program preparation steps. The programmer simply recalls the saved JCL, generally changes a few JCL statements (such as program name), then resubmits it to run in TSO batch.

Note however that each combination of functions (precompile only, bind only, precompile-bind-run, etc.) generates a different procedure. Each one has to be saved separately.

2.2. Invoke DB21 in TSO

Once in TSO, the programmer selects DB21, possibly via the ISPF Primary Option Menu.

The programmer selects option 5.

```
--------------------------------- ISPF  PRIMARY OPTION MENU ---------------------------------------------------
              SELECT  OPTION  ====>

          0           ISPF  PARMS                   -             SPECIFY  TERMINAL  AND ISPF
PARAMETERS
          1           BROWSE                        -             DISPLAY  SOURCE DATA / COMPUTER
LISTING
          2           EDIT                                        -       CREATE  OR CHANGE SOURCE
DATA
          3           UTILITIES                     -             PERFORM ISPF UTILITY FUNCTIONS
          4            . . . . . .
          5            DB21                                       -       DATABASE  INTERACTIVE
          6           . . . . . .
          7
```

Figure 11.3 The ISPF primary option menu.

2.3. Invoke Function in DB21

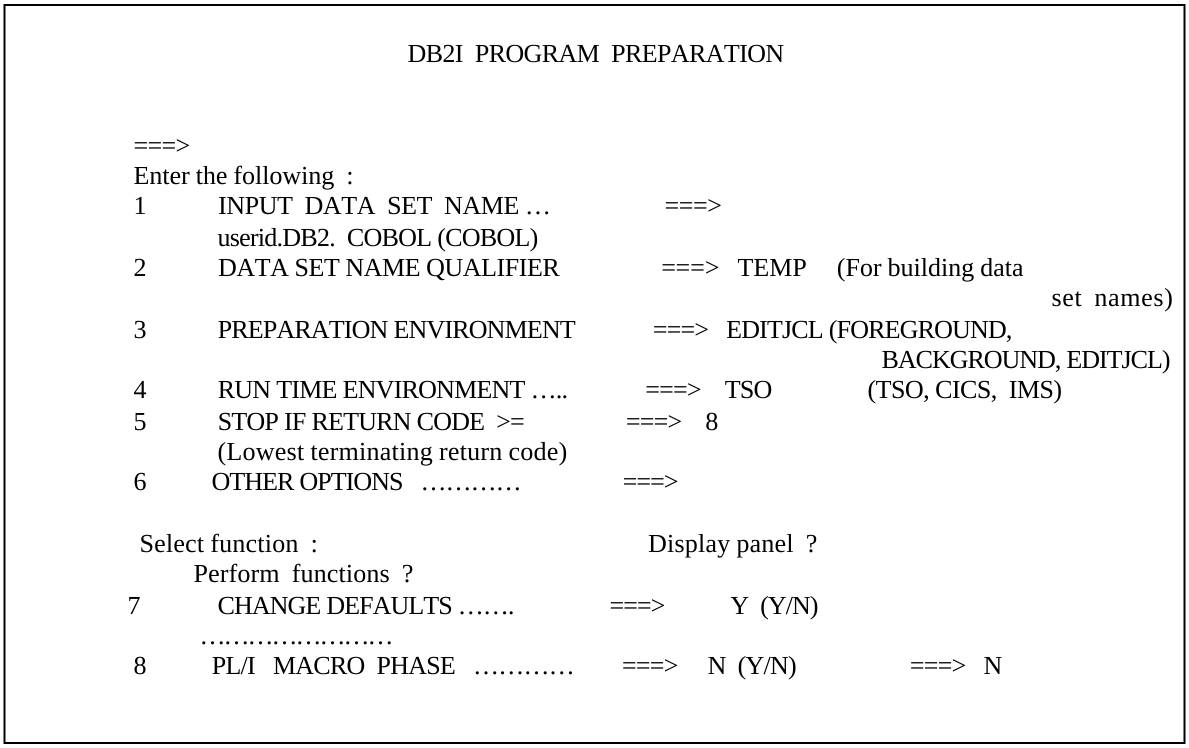
Figure 11.4 is the DB2I primary option menu.

2.4. The Program Preparation Panel

The programmer selects option 3, which will then generate the program preparation panel in Figure 11.5.

```
                                DB2I  PRIMARY  OPTION  MENU

          ==>
          Select one of the following and press ENTER

          1         SPUFI                                  (process SQL statements)

           2         DCLGEN                                (Generate SQL and source language
                                                            declarations)

          3         PROGRAM PREPARATION                      (prepare a DB2 application program
                                                           to run)

          4         PRECOMPILE                             (Invoke  DB2)

          5         BIND/REBIND                           (BIND/REBIND/FREE
                                                         application  plans)
```

Figure 11.4 The DB2I primary optin menu



```
                        DB2I  PROGRAM  PREPARATION


===>
Enter the following  :
1       INPUT  DATA  SET  NAME ...              ===>
        userid.DB2.  COBOL (COBOL)
2       DATA SET NAME QUALIFIER                 ===>  TEMP    (For building data
                                                                          set  names)
3       PREPARATION ENVIRONMENT                 ===>  EDITJCL (FOREGROUND,
                                                                 BACKGROUND, EDITJCL)
4       RUN TIME ENVIRONMENT .....              ===>  TSO          (TSO, CICS,  IMS)
5       STOP IF RETURN CODE  >=                 ===>  8
        (Lowest terminating return code)
6       OTHER OPTIONS  ...........              ===>

 Select function  :                             Display panel  ?
      Perform  functions  ?
7       CHANGE DEFAULTS .......                 ===>       Y  (Y/N)
       ......................
8       PL/I   MACRO  PHASE  ...........        ===>   N  (Y/N)            ===>  N
```

Figure 11.5 The Program preparation panel

Note the following :

1. The INPUT DATA SET NAME is that of the program

2. The DATA SET NAME QUALIFIER is TEMP. This is used in the BIND process.

3. The PREPARATION ENVIRONMENT is usually EDITJCL for batch programs, which allows the generated JCL statements in userid.temp.cntl to be edited. This is needed to add the JCL statements for batch files (say, the printer). The programmer can then later SUBMIT userid.temp.cntl to run in TSO batch. If FOREGROUND, the steps is Figure 11.1 are executed in foreground, one step at a time, which gives the programmer an instantaneous feedback on the execution of each step.

4. The RUN TIME ENVIRONMENT is TSO, unless we access the DB2 resources via CICS or IMS.

5. The STOP IF RETURN CODE value is 8.

6. The OTHER OPTIONS entry is a list of parameters you want included in this program preparation process ; for instance, options to the CICS/VS command language translator.

7. The CHANGE DEFAULT entry is usually Y the very first time so the programmer can see what the defaults are. In addition, the programmer usually codes the JCL Job statement that will be used instead of the default when the job is later submitted.

8. The PL/I macro phase is naturally only used in PL/I programs.

9. The PRECOMPILE options are both Y. before it is executed, the corresponding panel is displayed.

10. The CICS COMMAND TRANSLATION entry is for CICS programs.

11. Like those of PRECOMPILE, the BIND options are both Y.

12. In like manner, the COMPILE OR ASSEMBLE options are both Y. Note that COMPILE, LINK (item 13), and RUN (Item 14) use the same panel.

13. The LINK option has N for display since we will enter the necessary information along with the previous COMPILE OR ASSEMBLE step.

14. The RUN option has N for display since we will enter the necessary information along with the previous COMPILE OR ASSEMBLE step.

3. THE DB21 DEFAULT PANEL

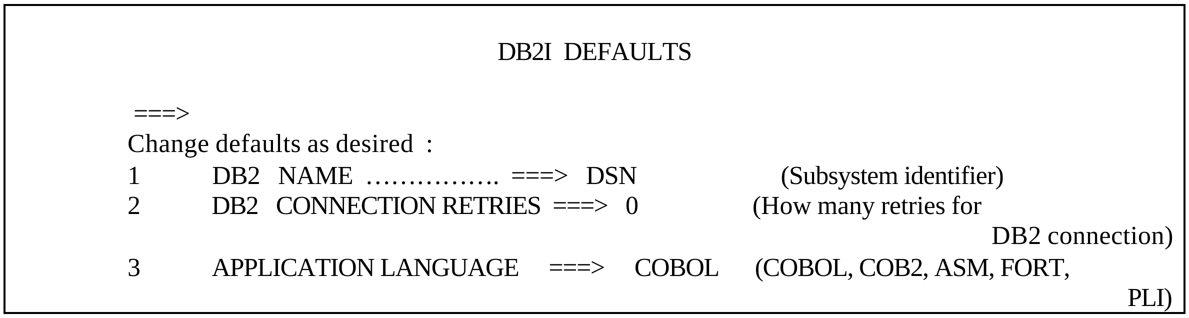
Note the following :

1. This panel appears if we choose "Y" for the Display panel column of the CHANGE DEFAULTS.

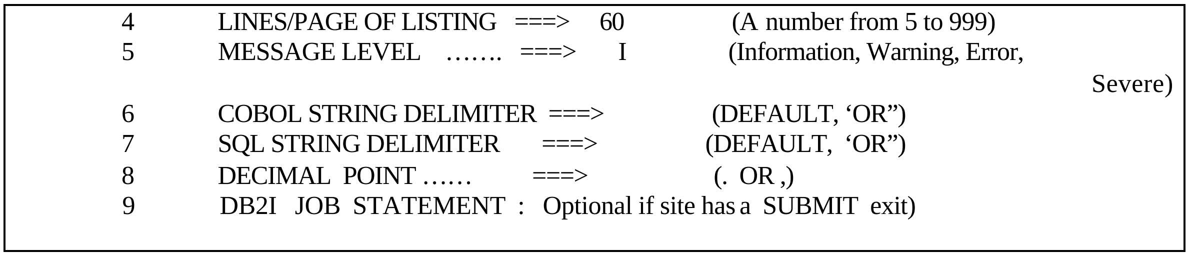
2. The programmer may change the system defaults.

3. Entry 9 is the Job statement coded by the programmer and will be used in place of the defaults. This will appear in the generated JCL that is submitted to run (Fig. 1.10).

Figure 11.6 The DB21 defaults panel



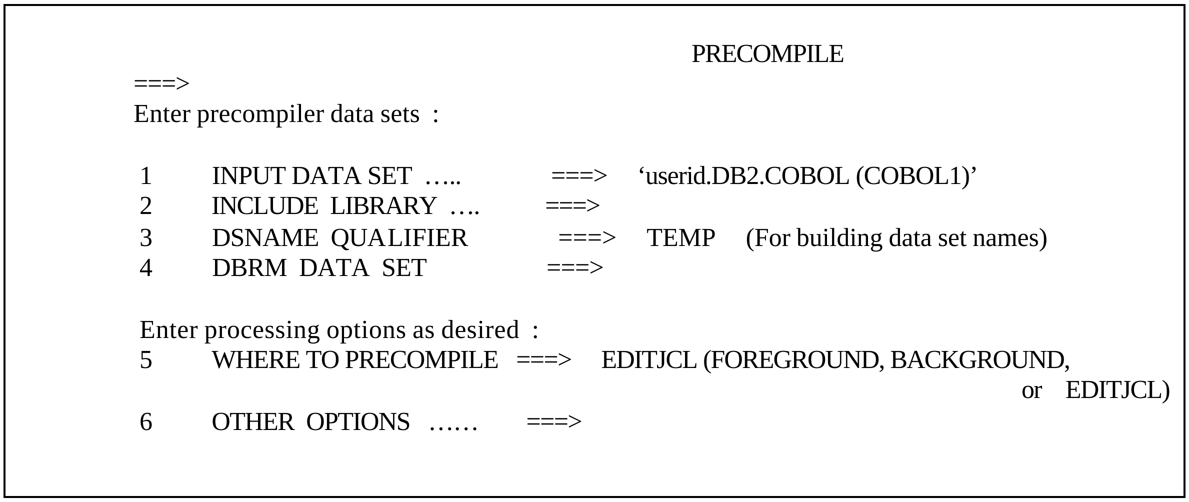
```
                          DB2I  DEFAULTS

 ===>
Change defaults as desired  :
1      DB2  NAME ……………. ===>  DSN           (Subsystem identifier)
2      DB2  CONNECTION RETRIES ===>  0       (How many retries for
                                                      DB2 connection)
3      APPLICATION LANGUAGE   ===>   COBOL   (COBOL, COB2, ASM, FORT,
                                                               PLI)
```



```
     4          LINES/PAGE OF LISTING   ===>    60             (A number from 5 to 999)
     5          MESSAGE LEVEL    …….  ===>     I             (Information, Warning, Error,
                                                                                    Severe)
     6          COBOL STRING DELIMITER  ===>                 (DEFAULT, ‘OR”)
     7          SQL STRING DELIMITER      ===>                (DEFAULT,  ‘OR”)
     8          DECIMAL  POINT ……        ===>                 (.  OR ,)
     9          DB2I   JOB  STATEMENT  :   Optional if site has a  SUBMIT  exit)
```

4. THE PRECOMPILE PANEL

Figure 11.7 is the Precompile panel



```
                                                         PRECOMPILE
      ===>
      Enter precompiler data sets  :

       1       INPUT DATA SET …..            ===>    ‘userid.DB2.COBOL (COBOL1)’
       2       INCLUDE  LIBRARY ….         ===>
       3       DSNAME  QUALIFIER            ===>    TEMP    (For building data set names)
       4       DBRM  DATA  SET               ===>

       Enter processing options as desired  :
       5       WHERE TO PRECOMPILE   ===>    EDITJCL (FOREGROUND, BACKGROUND,
                                                                           or    EDITJCL)
       6       OTHER  OPTIONS   ……       ===>
```

Figure 11.7 The PRECOMPILE panel

Note the following :

1. This panel appears if we choose "Y" for the Display panel column of the PRECOMPILE entry of Figure 11.5.

2. If this panel is accessed via the Progra-n Preparation panel (Fig. 11.5), the INPUT DATA SET NAME contains the value specified in that panel.

3. The INCLUDE LIBRARY entry specifies the library containing members to be included by the precompiler, for instance, the library containing the output of the DCLGEN operation.

4. If this panel is accessed via the Program Preparation panel (Fig. 11.5), the DSNAME QUALIFIER entry contains the value specified in that panel.

5. The DBRM DATA SET entry is the DBRM library for the precompiler output (which becomes input to BIND). If this panel is accessed via the Program preparation panel (Fig. 11.5) it is initially blanks. When you press ENTER, the value of DATA SET NAME QUALIFIER (here equal to "TEMP") is concatenated with "DBRM" and becomes the DBRM DATA SET.

6. If this panel is accessed via the Program Preparation panel, the WHERE TO COMPILE entry is the same as the PREPARATION ENVIRONMENT of that panel.

7. The OTHER OPTIONS entry are precompiler options that will override the installation standards. Many of these are identical to the options for compile.

Examples are APOST to specify the single quotes as the literal delimiter, FLAG (x,y) to specify the level of diagnostic messages, and so on.

4.1. Output of CICS Command Translation

1. If there is a command language translation step (CICS application program only, the translated program is in userid.temp.cicsin.

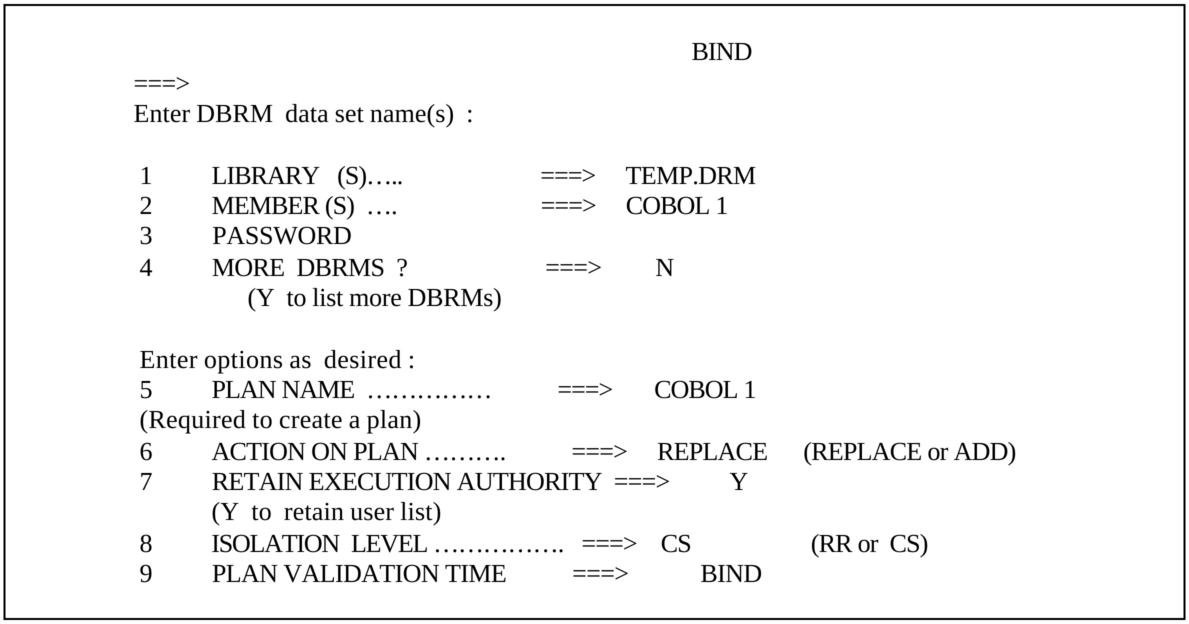
2. The output listing is in userid.temp.cxlist.

4.2. Output of Precompile

1. The modified source module is in userid.temp.cobol.
2. The precompile listing is in userid.temp.pclist.

5. THE BIND PANEL

Figure 11.8 is the BIND panel



```
                                                BIND
===>
Enter DBRM  data set name(s)  :

 1      LIBRARY   (S)…..              ===>    TEMP.DRM
 2      MEMBER (S)  ….                ===>    COBOL 1
 3      PASSWORD
 4      MORE  DBRMS  ?                 ===>      N
           (Y  to list more DBRMs)

 Enter options as  desired :
 5      PLAN NAME  ……………          ===>     COBOL 1
 (Required to create a plan)
 6      ACTION ON PLAN ……….         ===>    REPLACE     (REPLACE or ADD)
 7      RETAIN EXECUTION AUTHORITY  ===>        Y
        (Y  to  retain user list)
 8      ISOLATION  LEVEL …………….  ===>   CS             (RR or  CS)
 9      PLAN VALIDATION TIME         ===>         BIND
```

Figure 11.8 The BIND panel

The program Preparation Process

1. This panel appears if we choose "Y" for the Display panel column of the BIND entry of Figure 11.5.

2. If this panel is accessed via the program preparation steps, the LIBRARY entry contains the value specified in the DBRM DATA SET entry of the Precompile step.

3. The MEMBER entry is the DBRM itself. For a partitioned data set, its default value is the member name of the INPUT DATA SET NAME entry of the program preparation panel. For a sequential data set, it is the second qualifier of the data set.

4. The PASSWORD entry is used for a password.

5. The MORE DBRMS entry allows the programmer to enter more DBRMs via another panel.

6. The PLAN NAME entry is the name of the application plan created during BIND. The default value for a partitioned data set is the member name of the INPUT DATA SET NAME of the program preparation panel. For a sequential file, it is the second qualifier.

7. The ACTION OF PLAN entry is REPLACE or ADD. REPLACE will also add a new plan.

8. The RETAIN EXECUTION AUTHORITY entry is valid only if we are modifying an old plan. If Y, those authorized to bind or execute the old plan are to retain the authority for the modified plan.

9. The ISOLATION LEVEL entries are explained in Chapter 20. CS promotes greater concurrency ; RR allows "repeatable read" by a single program.

10. The PLAN VALIDATION TIME specifies when full validity checking is to be done. BIND means do it during the BIND process, which is the efficient way. RUN means do it during program execution, which has to be used if authorization is not granted by the time BIND is done.

11. The RESOURCE ACQUISITION TIME entry determines when you want the system to acquire resources for your program. USE means acquire it when first used in the program ; LLOCATE means acquire it when first used in the plan is allocated (the program starts). Specifying USE promotes greater concurrency.

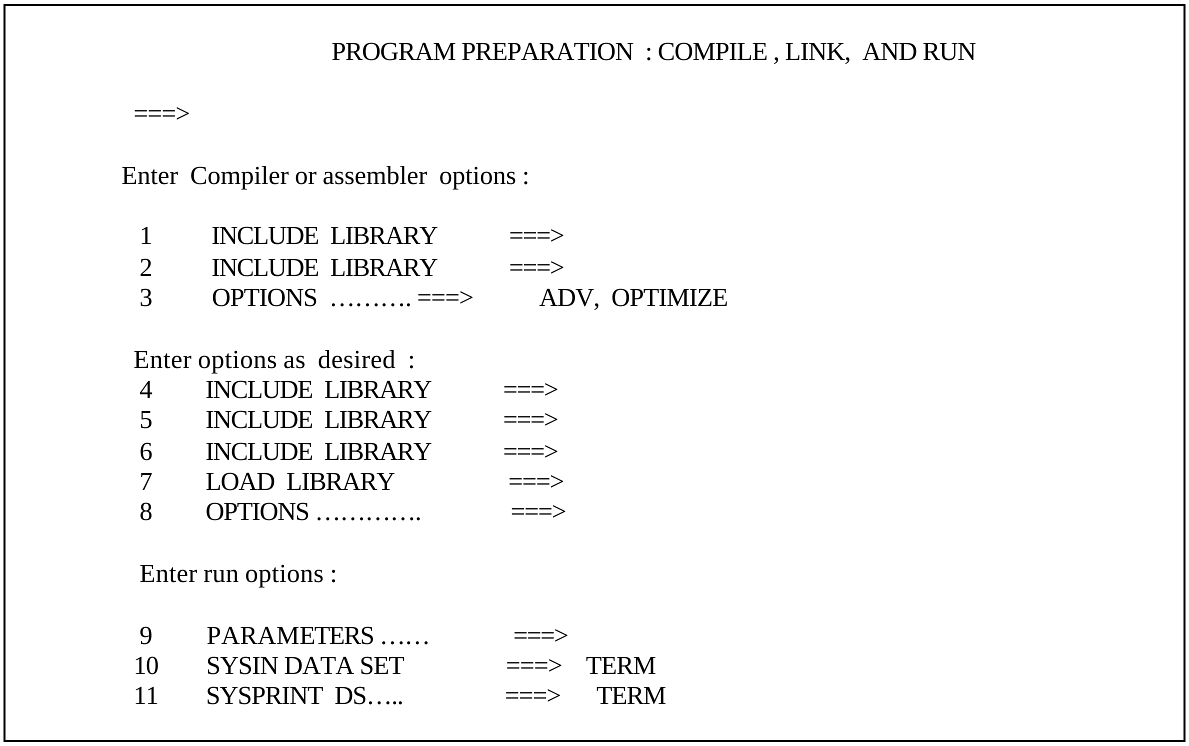
12. The RESOURCE RELEASE TIME entry determines when you want the system to release resources taken by your program. COMMIT means release them when committed (SQL COMMIT or SQL ROLL BACK statements).

DEALLOCATE means wait until the program terminates. COMMIT promotes greater concurrency.

13. The EXPLAIN PATH SELECTION entry allows the user to query DB2 about how it navigates through the table to access the data.

6. THE COMPILE / LIN / RUN PANEL

Figure 11.9 is the COMPILE / LINK / RUN panel



```
                    PROGRAM PREPARATION  : COMPILE , LINK,  AND RUN

   ===>

  Enter  Compiler or assembler  options :

   1      INCLUDE  LIBRARY        ===>
   2      INCLUDE  LIBRARY        ===>
   3      OPTIONS  ………. ===>        ADV,  OPTIMIZE

   Enter options as  desired  :
   4      INCLUDE  LIBRARY        ===>
   5      INCLUDE  LIBRARY        ===>
   6      INCLUDE  LIBRARY        ===>
   7      LOAD  LIBRARY           ===>
   8      OPTIONS …………..         ===>

   Enter run options :

   9      PARAMETERS ……          ===>
  10      SYSIN DATA SET          ===>   TERM
  11      SYSPRINT  DS…..         ===>    TERM
```

Figure 11.9 The COMPILE/LINK/RUN panel

Note the following :

1. This panel appears if we choose "Y" for the Display panel column of the COMPILE OR ASSEMBLE entry of Figure 11.5.

2. The programmer may include upto two libraries for the compile phase.

3. The OPTIONS entry are compiler options that will override the installation standards, for instance, ADV.

4. Entries 4 to 6 allow upto three libraries containing members to be included in the linkage editor run.

5. The LOAD LIBRARY entry has the default of RUNLIB.LOAD.

6. The PARAMETERS entry is a list of parameters you want passed to the runtime process or the program.

7. By default, the SYSIN DATA SET and SYSPRINT DS entries are TERM.

7 EXECUTING THE PROGRAM PREPARATION STEPS

With typical batch programs, "EDITJCL" is entered in the PREPARATION ENVIRONMENT entry of the DB2 program preparation panel. Before the run is done, DB2 displays a panel that show the JCL statements it generated, and if the programmer entered a Job statement in the DB21 DEFAULT panel, it will be used in place of the standard Job statement. The programmer may then insert JCL statements for batch files (say, printers).

Figure 11.10 shows this DB2-generated JCL.

```
000001//jobname  JOB ……..              from Job statement
000002// Go EXEC PGM ……..
000003 //  ……..
…………
000009 // LSTOUT  DD  SYSOUT=A      DB2-generated statements
000010 // CARDIN  DD  *
000011   00005
000012   00010                         user-added statements
000013   00015
000014  /*
……….
000020    //  SYSTSIN  DD   *
………
000030    INPUT (DD ‘’’ userid.DB2.COBOL (PROG1202) ‘’’)
……..
000036    // PLAN (plan name)          DB2-generated statements
……..
000039          RUN   (TSO)
…….
000045    /*
000046    //
```

Figure 11.10 The DB2-generated JCL.

1. DB2 saves the JCL statements in userid.temp.cntl. This data set will be retained until TSO logoff. As suggested in Section 8 later in this chapter, this is best copied into a programmer-owned data set.

2. The programmer adds JCL statements for the batch files. The programmer may now submit the job with the SUB command.

8. EXECUTING THE PROGRAM PREPARATION STEPS USING JCL

The DB2-generated JCL statements (in userid.temp.cntl) are lost at TSO logoff. Therefore, it is generally best for the programmer to copy it into his or her own data set. As mentioned before, note that each combination of functions selected (precompile only, bind only, precompile-bind-run, etc.) generates a different procedure.

Each must be saved separately. The programmer may conveniently just resubmit one of the saved versions of the JCL, after making minor changes.

One change is the JCL statements for batch files. Others are the "INPUT" entry for the program data set name (line 00030 of Fig. 11.10) and the "PLAN" entry for the plan name (line 000036).

SELECT OPERATIONS

1 PROCESSING ONE ROW FROM SOURCE TABLE (S) VIEW

The programmer may select one row from a source table(s) or view and bring it immediately into the application program. This option is used only if there is at most only one row selected because the column(s) used in the selection criteria (WHERE clause) only contains unique values (the column was defined with UNIQUE INDEX). An example is a selection based on employee number (which is unique).

If this processing style is used where the selection criteria uses a column that may contain non unique values, an error will result if DB2 finds more than one row.

1.1. SQLCODE for Select (Read-Only) Operations

For SELECT operations, there are only two values of SQLCODE that are of interest to the user: a value of 0 (no error) or a value of 100 (“no record found”) on either the SELECT statement itself or the FETCH statement (if processing multiple rows). On a negative value, the row is not read into the program.

The use of SQLCODE is shown in the discussion of program logic and the program examples.

1.2 DCLGEN Output for XLIM.EMPTABLE

Figure 12.1 is the DCLGEN output for the XLIM.EMPTABLE Employee table. We will use it in all our program examples. Note that we have both the DECLARE TABLE statement and the table row description.

```
************************************************************************
*       DCLGEN

*       TABLE (XLIM.EMPTABLE)

*       LIBRARY (XLIM.DB2 COBOL (EXMPTABDL)

*       ACTION (REPLACE)

*       APOST

*       IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS.
*************************************************************************

        EXEC SQL DECLA RE XLIM.EMPTABLE TABLE
        (       EMPNO                         CHAR (5) NOT NULL,
                EMPNAME               CHAR (30) NOT NULL WITH DEFAULT,
                EMPDEPT               CHAR (3) NOT NULL WITH DEFAULT,
                EMP SALARY                    DECIMAL (7,0) NULL
*************************************************************************
*       COBOL DECLARATION FOR TABLE XLIM.EMPTABLE
*************************************************************************
01      DCLEMPTABLE
```

```
        10 EMPNO                                PIC x (5)
        10 EMPNO                                PIC x (30)
        10 EMMPDEPT                             PIC xxx.
        10 EMPSALARY                    PIC S9 (7) COMP-3
****************************************************************************
*       THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 4
****************************************************************************
```

Figure 12.1 DCLGEN output for the XLIM.EMPTABLE employee table.

1.3 Current Data on XLIIVI.EMPTABLE

In this chapter, we will use Figure 12.2 as the input to our program. It contains the current data on the XLIM.EMPTABLE Employee table.

	Employee Number (EMPNO)	Name (EMPNAME)	Department Number (EMPDEPT)	Salary EMPSALARY
1.	00005	BAKER, C.	003	0030000
2.	00008	RANDOLPH, R.	005	0029000
3.	00010	RICHARDS, M.	002	Nulls
4.	00015	DAVIS, L.	006	0030000
5.	00150	ELLIOT, T.	005	0031000
6.	00170	RABINOVITE, M.	003	Nulls
7.	00190	LEE, R.	005	0030000

Figure 12.2 Current data on the XLIM.EMPTABLE employee table.

1.4 Program Logic: One-row Selection

If the selection criteria (WHERE clause) specifies a column (s) defined with a unique index, then at most only one row can be selected. In our example, we assume this case for the employee number column. The general program logic is shown in Figure 12.3.

```
WORKING-STORAGE SECTION


        . . . . .
        EXEC SQL
                INCLUDE member-name (from DLGEN output)
        END-EXEC.
        EXEC SQL
                INCLUDE SQLCA
        END EXEC.
PROCEDURE DIVISION.
        . . . . . .
***     set EMPNO to the correct value ***
        EXEC SQL
                SELECT EMPNAME,
                        EMPSALARY
```

```
            INTO     : EMPNAME,
                     : EMPSALARY

            FROM XLIM – EMPTABLE
            WHERE EMPNO = :EMPNO
END-EXEC.
IF SQLCODE EQUAL TO ZEROES
            OK continue
ELSE error routine.
```

Figure 12.3 Program logic: one – row selection.

The following apply:

1. At the beginning, we have to set the Cobol data-name EMPNO to the correct value. It will be used as the search criteria.

2. The column names in the SELECT statement (EMPNAME, EMPSALARY) are those of the table.

3. The INTO clause brings the data for the selected columns into the Cobol data-names specified (note the colon prefix). Thus,:EMPNAME and:EMPSALARY are the Cobol data-names generated by DCLGEN (see Fig. 12.1).

4. We check the SQLCODE field and if zero, we know the select was successful and we may continue or else we did not get any row.

1.5 Program Listing: One-row Selection

We will now print selected rows (one row per selection) from XLIM.EMPTABLE (Fig. 12.2), Selection is based on employee numbers coded in an in-line "card file", unless the salary value of the employee is nulls, where we bypass the row. Figure 12.4 is the program listing.

The following apply:

1. Lines 001400 to 002600 are the in-line card file that contains the employee numbers, and print file.

2. Lines 002800 to 006800 are the various data-names such as counters and the print lines.

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID. PROG1204.
000300 ******************************************************************
000400 *
000500 *  1.  THIS PROGRAM PRINTS SELECTED ROWS IN THE EMPLOYEE TABLE.
000600 *
000700 *  2.  SELECTION IS BASED ON THE EMPLOYEE NUMBER.  IT IS THE
000800 *       PRIMARY KEY (DEFINED WITH UNIQUE, CLUSTERED INDEX).
***************************************S*****************************
001100   ENVIRONMENT DIVISION
001200   CONFIGURATION SECTION
001300   INPUT-OUTPUT SECTION
001400   FILE-CONTROL
001500            SELECT SEARCH-FILE  ASSIGN TO CARDIN
001600            SELECT PRINT-OUTPUT ASSIGN TO LSTOUT
001700   DATA  DIVISION
001800   FILE  SECTION
001900   FD                 SEARCH-FILE
002000                      BLOCK CONTAINS  0  RECORDS
002100                      LABEL  RECORDS OMITTED
002200   01        SEARCH-RECORD
002300   FD                 PRINT-OUTPUT
002400                      BLOCK CONTAINS  0  RECORDS
002500                      LABEL  RECORDS OMITTED
002600   01                 PRINT-RECORD                           PIC X  (133)
002700   WORKING-STORAGE SECTION
002800   01                 W005-LINE-COUNT                        PIC S9 (8) COMP VALUE +99
002900   01                 W005-LINE-LIMIT                        PIC S9 (8)  COMP VALUE +55
003000   01                 W005-LINE-SKIP                         PIC 99
003100   01                 W005-SEARCH-EMPNO                      PIC X (5)
003200   01                 W005-SALARY-INDV                       PIC S9 (4)  COMP
003300   01                 W005-END-OF-SEARCH-FILE                PIC X VALUE  'N'
003400                      88        W005-NO-MORE-SEARCH-CARDS PIC X VALUE 'Y'

003500   01                 W005-DETAIL-LINE

003600                      05                FILLER               PIC X (4)
003700                      05                W005-DETAIL-EMPNO    PIC X (5)
003800                      05                FILLER               PIC X (4)
003900                      05                W005-DETAIL-EMPNO    PIC X (30)
004000                      05                FILLER               PIC X (2)
004100                      05                W005-DETAIL-EMPDEPT           PIC XXX
004200                      05                FILLER               PIC X (3)
004300                      05                W005-DETAIL-EMPSALARY         PIC  Z, ZZZ,  ZZ9
004400                      05                W005-DETAIL-EMPSALARY –  ALPHA
004500                      05                REDEFINES  W005-DETAIL EMPSALARY
004600                                                             PIC X (9)
004700                      05                FILLER               PIC X
004800                      05                W005-DETAIL-SQLCODE           PIC **********
004900                      05                FILLER               PIC X
005000                      05                W005-DETAIL-SQLCODE-NOTE   PIC X (3)
005100   01                 W005-HEADER-LINE2
005200                      05                FILLER         PIC X (3) VALUE SPACES
005300                      05                FILLER         PIC X (8) VALUE 'EMPLOYEE'
005400                      05                FILLER         PIC X (34)  VALUE SPACES
005500                      05                FILLER         PIC X (5) VALUE 'DEPT'
005600                      05                FILLER         PIC X (18) VALUE SPACES
005700                      05                FILLER         PIC X (6)  VA LUE 'RETURN'
```

```
005800  01          W005-HEADER-LINE2
005900              05          FILLER        PIC X (4) VALUE SPACES
006000              05          FILLER        PIC X (6)  VALUE ‘NUMBER’
006100              05          FILLER        PIC X (8) VALUE SPACES
006200              05          FILLER        PIC X (4) VALUE ‘NAME’
006300              05          FILLER        PIC X (22) VALUE SPACES
006400              05          FILLER        PIC X (6)  VALUE ‘NUMBER’
006500              05          FILLER        PIC X (4)  VALUE ‘SPACES’
006600              05          FILLER        PIC X (6)  VALUE ‘SALARY’
006700              05          FILLER        PIC X (8)  VALUE ‘SPACES’
006800              05          FILLER        PIC X (4)  VALUE ‘CODE’
006900              EXEC SQL
007000              INCLUDE  SQLCA
007100              END-EXEC
007200              EXEC SQL
007300                    INCLUDE  EMPTABDL
007400              END-EXEC
007500  PROCEDURE DIVISION
007600              OPEN INPUT SEARCH-FILE
007700                    OUTPUT PRINT-OUTPUT
007800              MOVE  SPACES TO  W005-DETAIL-LINE
007900              PERFORM C120-READ-ONE-DETAIL-LINE
008000              PERFORM  C020-PROCESS-ALL-ROWS
008100                    UNTIL  W005-NO-MORE-SEARCH-CARDS
008200              CLOSE SEARCH-FILE PRINT-OUTPUT
008300              DISPLAY ***  END OF JOB PROG 1204  UPON SYSOUT
008400              GOBACK
008500  C020-PROCESS-ALL-ROWS
008600              PERFORM  C100-FETCH-ONE-ROW
008700              IF SQLCODE EQUAL TO ZERO
008800                    PERFORM  C040-LAYOUT-ROW-COLUMNS
008900              ELSE MOVE  W005-SEARCH-EMPNO  TO W005-DETAIL-EMPNO
009000              MOVE SQLCODE          TO      W005-DETAIL-SQLCODE-NOTE
009100              IF SQLCODE NOTG EQUAL  TO ZERO
009200                    MOVE  ‘***’             TO W005-DETAIL-EMPNO
009300              PERFORM  C060-PRINT-DETAIL-LINE
009400              MOVE SPACES TO  W005-DETAIL-LINE
009500              PERFORM  C120-READ-ONE-SEARCH-CARD
009600  C040-LAYOUT-ROW-COLUMNS
009700              MOVE  EMP NO          TO      W005-DETAIL-EMPNO
009800              MOVE  EMPNAME         TO      W005-DETAIL-EMPNAMES
009900              MOVE EMPDEPT          TO      W005-DETAIL-EMPDEPT.
010000              IF W005-SALARY-INDV  LESS THAN  ZERO
010100              MOVE  ‘NULLS ***’     TO      W005-DETAIL-EMPSALARY-ALPHA
010200              ELSE MOVE EMPSALARY  TO       W005-DETAIL-EMPSALARY
010300  C060-PRINT-DETAIL-LINE
010400              MOVE EMP NO           TO      W005-DETAIL-EMPNO
010500              PERFORM  C080-PRINT-HEADER-LINES
010600              WRITE PRINT-RECORD FROM  W005-DETAIL-LINE
010700                          AFTER ADVANCING  W005-LINE-SKIP LINES
010800              ADD      1   TO     W005-LINE-COUNT
010900              MOVE    1    TO     W005-LINE-SKIP
011000  C080-PRINT-HEADER-LINES
011100              WRITE PRINT-RECORD FROM  W005-HEADER-LINE1
011200                          AFTER ADVANCING PAGE
011300              WRITE PRINT-RECORD FROM  W005-HEADER-LINE2
011400                          AFTER ADVANCING 2 LINES
011500              MOVE ZEROS  TO  W-005-LINE-COUNT
```

```
011600                MOVE   2                        TO   W-005-LINE-SKIP
011700    C100-FETCH-ONE-ROW
011800                EXEC SQL
011900                          SELECT EMP NO
012000                                              EMP NAME,
012100                                              EMPDEPT,
012200                                              EMPSALARY
012300                INTO  :  EMPNO
012400                                    :         EMPNAME
012500                                    :         EMPDEPT,
012600                                    :         EMPSALARY : W005-SALARY-INDV
012700                FROM XLIM.EMPTABLE
012800                          WHERE EMPNO  =  :W005-SEARCH-EMPNO
012900                END-EXEC
013000    C120-READ-ONE-SEARCH-CARD
013100                READ  SEARCH-FILE INTO  W005-SEARCH-EMPNO
013200                          AT END, MOVE  'Y'  TO  W005-END-OF-SEARCH-FILE
```

3. Lines 006900 to 007100 generate the communication area data block shown in Figure 10.1

4. Lines 007200 to 007400 include the DCLGEN output shown in Figure 12.1

5. Lines 007600 to 008200 are the house-keeping routines and the main-loop control.

6. Lines 008500 to 009500 control the processing of rows selected from the employee number entered in the in-line card file. Since the employee number is a unique index, we actually select at most only one row.

   a. Line 008600 performs the attempt to select a single row.

   b. Lines 008700 to 008900 perform the laying out of the columns if there was a row selected; or else we only lay out the employee number from the original card file (so we can later print an error message).

   c. Lines 009000 to 009200 lay out the SQULCODE, plus the literal *** if it is not zero. Note that an SQLCODE value of zeros will print as blanks because of line 004800.

   d. Lines 009600 to 010200 lay out the detail line, then blank it out for the next row.

7. Lines 009600 to 010200 lay out the row columns (from lines 008700 to 008800) only if a row was selected.

   a. Lines 009700 to 009900 simply lay out the columns that will never contain nulls (see Fig. 12.1).

b. Lines 010000 to 010200 check the W005-SALARY-INDV indicator variable to see if DB2 detected nulls (a negative value for W005-SALARY-INDV) for the EMPSALARY column (see line 012600). If so, we display the 'NULLS **' literal, instead of the actual salary value.

8. Lines 010300 to 011600 are the routines to print the detail and header lines.

9. Lines 011700 to 012900 is the statement to select one row. Note that we use the INTO clause, which brings that single row selected row selected into the data-names specified in the clause.

   a. Lines 011900 to 012200 specify the columns to be selected. Note that it is easy to code one row per line since the programmer can just copy the row definitions from the DCLGEN output (see Fig. 12.1). in addition, program maintenance becomes easier.

   b. Lines 012300 to 012600 specify the corresponding Cobol data-names that will get the value from the row. Note that each is prefixed with a colon.

   c. Line 012600 shows the use of the indicator variable for a column that may contain nulls: if not specified, there is an SQLCODE error if the column does contain nulls.

   d. Line 012800 is the selection criteria. We know that there will be at most only one row selected since EMPNO is defined with a unique index.

10. Lines 013000 to 013200 read the in-line card file to get the employee numbers used in the selection process.

1.6 Additional JCL Statements and In-line Card File

Assuming that we use "EDITJCL" in entry 3 of Figure 11.5 in Chapter 11, DB2 will generate most of the JCL statements for us. However, we still have to include those for the batch files.

For our program example, these are for the display output, the in-line "card file," and the print file. Thus:

```
/ / SYSOUT   DD     SYSOUT=*
/ / LSTOUT   DD     SYSOUT=*
/ / CARD IN  DD     *
             0015
             00010
             00600
             00008.
             /*
```

1.6.1 The list of selected rows.

Note that from the previous additional JCL statements, the employee numbers are not in sequence. The program will just do a random read of the input table as each input card is read. From the current data in XLIM.EMPTABLE (Fig. 12.2), we can see that we process in sequence, employee numbers ‘00015’, ‘0010’, and ‘0008’. Employee number 00600 does not exist in the table.

1.7 Output Listing: One-row Selection

Figure 12.5 is the output of the program in Figure 12.4.

```
EMPLOYE                       DEPT.                  RETURN
NUMBER      NAME              NUMBER     SALARY      CODE

00015       DAVIS, L.         006        30,000

00010       RICHARDS, M       002        NULLS ***

00600                                                + 100 ***

00008       RANDOLPH,R        005        29,000
```

Figure 12.5 Output listing: one-row selection.

Note the following:

1. Employee number ‘00015’ is printed out. SQLCODE is zero, hence the “RETURN CODE” column in the detail line shows spaces.

2. Employee number ‘00010’ is printed out. Since the salary value is nulls, we print our ‘NULLS***’ in the salary column.

3. Employee number ‘00600’ is not in the table. The SQLCODE is 100, which we print out as +100.

4. Employee number ‘00008’ is printed out. SQLCODE is zero.

7. PROCESSING MULTIPLE ROWS (BY USING A CURSOR)

In many cases, there are multiple rows that satisfy a particular selection criteria (that is, the column used in the **WHERE** clause is nonunique). *Since Cobol does not allow the programmer to bring all such rows”en masse” to program, the only way to process this condition is by using a cursor fetch the data, one row at a time. The style shown in Section I will not work.*

2.1 Defining the Cursor

The cursor name, which is not defined as a Cobol data-name, is defined in a **DECLARE** statement that has the **SELECT** statement subordinated to it. The **DECLARE** statement is nonexecutable code, and is coded either in the **PRCEDURE DIVISION** (preferably, for better documentation) or in the **WORKING –STORAGE** section. The subordinate SELECT statement (which is coded just like any other **SELECT** statement) identifies the subset of the input table(s) that will serve as the "temporary cursor table" (see Section 2.1.1) from which the program fetches data, one row at a time. Processing is then similar to that of a sequential file.
The format of the **DECLARE** statement is:

```
EXEC SQL
        DECLARE cursor-name CURSOR FOR
                        SELECT EMPNAME, EMPSALARY
                                FROM XLIM.EMPTABLE
                        WHERE EXMPDEPT=:SEARCH-DEPT
END-EXEC
```

The following apply:

1. Cursor-name is the programmer-defined cursor name. It is not a Cobol dataname.

2. The SELECT operand specifies the rows in the "temporary cursor table".

3. The WHERE operand specifies the rows in the "temporary cursor table". In our example, they are all rows where the value of the EMPDEPT column is equal to that of the Cobol SEARCH-DEPT data-name. If this operand is missing, all rows are used.

2.1.1 The "temporary cursor table".

The **SELECT** Operand of the **DECLARE** cursor-name statement defines the group that serves as the "temporary cursor table" from which individual rows are later fetched. For instance, if we use Figure 12.2 as the original table and code the following:

```
EXEC SQL
        DECLARE EMPCSR CURSOR FOR
        SELECT EMPNO, EMPNAME, EMPSALARY
        FROM XLIM.EMPTABLE
        WHERE EMPDEPT = ‘005’
END-EXEC.
```

	Employee		
	Number	Name	Salary
	(EMPNO)	(EMPNAME)	(EMPSALARY)
1.	00150	ELLIOT, T.	0031000
2.	00190	LEE, R.	0030000
3.	00008	RANDOLPH,’R.	0029000

Figure 12.7 The physical “Temporary Cursor Table”.

Note that Figure 12.7 shows physical entity, implemented by DB2 in a temporary file.

2.2 Opening the Cursor to Start Processing

Before cursor processing starts, the program must execute the SQL statement OPEN cursor-name. This also brings the cursor to the first row the “temporary cursor table”. In figure 12.6,this is the row belonging to employee number ‘00008’; in Figure 12.7, this is the row belonging to employee number ‘00150’.

The format is:

```
EXEC SQL
            OPEN Cursor-name
END-EXEC
```

2.3 FETCH Statement to Read One Row

For an open cursor, the FETCH statement gets one row at a time to the program. The format is:

```
EXEC.SQL
            FETCH cursor-name
            INTO:data-name,:data-name2.
END-EXEC.
```

Note that SQLCODE is equal to 100 if there are not more rows (“no record found” condition).

2.4 Closing the Cursor at End of Processing

Once the program is finished with processing the “temporary cursor table” (or at any time), the CLOSE cursor-name statement is executed. The format is:

```
EXEC.SQL
            CLOSE cursor-name
END-EXEC.
```

2.5 Basic Program Logic: Multiple Rows

The basic program logic is Figure 12.8.

The following apply:

1. The DECLARE cursor-name statement identifies the cursor, the columns selected, and the search criteria.
   a. The column-names in the SELECT clause (EMPNAME, EMPSALARY) are those of the table.
   b. Note that if the WHERE clause is missing (as we will show in the program example in Fig. 12.9), we are reading the whole table.

2. We set the Cobol data-name SEARCH-DEPT to the correct value since it is used as the search criteria.

```
        WORKING-STORAGE SECTION
        …..
        EXECSQUL
                        INCLUDE member-name (from DCLGEN output)
        END EXEC.
        …..
        END SQL
                        INCLUDE SQLCA
PROCEDURE DIVISION.
        …..
        EXEC.SQL
                        DECLARE cursor-name CURSOR FOR
                                SELECT EMPNAME.EMPSALARY
                        FROM XLIM.EMPTABLE
                                WHERE EMPDEPT =:SEARCH-DEPT

        END-EXEC.
***     set SEARCH-DEPT to the correct value ***
        EXECSQL
                        OPEN cu-sor-name
        END.EXEC.
        PERFORM C200-PROCESS-EMPLOYEE-ROW UNTIL SQLCLDE = 100
        EXECSQL
                        CLOSE cursor-name
        END-EXEC.
C200-PROCESS-EMPLOYEE-ROW
                        FETCH cursor-name
                                INTO:EMPNAME
                                        :EMPSALARY
        END.EXEC.
        IF SQLCODE TO EQUAL TO ZEROS
                        OK continue.
```

Figure 12.8 Program logic: multiple-row selection.

3. As part of the Main-loop routine, we open the cursor, perform the processing of all rows, then close the cursor.

	Employee Number (EMP NO)	NAME (EMP NAME)	Salary (EMPSALARY)	
1.	00005	BAKER, C.	0030000	
2.	00008	RANDOLPH, R.	0029000	◄
3.	00010	RICHARDS, M.	Nulls	◄
4.	00015	DAVIS, L.	0030000	3 rows in the "temporary" cursor table"
5.	00150	ELLIOT, T.	0031000	◄
6.	00170	RABINOVITZ, M.	Nulls	
7.	00190	LEE, R.	0030000	

Figure 12.6 "Temporary Cursor Table"

We get Figure 12.6 as the "temporary cursor table".
Note that there are only three rows in this "temporary cursor table".

2.1.2. The "Conceptual temporary cursor table"

If the program can use the original table (for instance, it does not care about the sequence of the rows being fetched), then this "temporary cursor table" is just a concept. DB2 will simply use the original table and point the cursor to the current row being processed.
This is in fact the case in Figure 12.6 As we said, only three rows can be fetched from the table.

2.1.3. The "Physical temporary cursor table"

If the program requires rows fetched to be in a specific sequence, the ORDER BY clause is specified. If DB2 decides that using an existing index is more efficient than doing a sort, then DB2 will still use the original table as the "temporary cursor table", otherwise, a sort is done. In the latter, DB2 first selects the needed rows, then sorts them as the "temporary cursor table" in a temporary file. In short, it exists as a physical entity, independent of the original table. For instance, if we use Figure 12.2 as the input the code the following (we assume there is no index by EMPNAME).

```
EXEC SQL
        DECLARE EMPCSR CURSOR FOR
        SELECT EMPNQ, EMPNAME, EMPSALARY
        FROM XLIM.EMPTABLE
        WHERE EMPDEPT = '005'
        ORDER BY EMPNAME
END-EXEC.
```

We get Figure 12.7 as the "temporary cursor table".

	Employee Number EMP NO.	Name (EMPNAME)	Salary (EMPSALARY)
1.	00150	ELLIOT, T.	0031000
2.	00190	LEE, R.	0030000
3.	00008	RANDOLPH ; R	0029000

Figure 12.7 The physical "Temporary Cursor Table".

Note that Figure 12.7 shows a physical entity, implemented by DB2 in a temporary file.

2.2. Opening the Cursor to Start Processing

Before cursor processing starts, the program must execute the SQL statement OPEN cursor-name. This also brings the cursor to the first row of the "temporary cursor table". In figure 12.6, this is the row belonging to employee number '00008', in Figure 12.7, this is the row belonging to employee number '00150'.

The format is :

```
EXEC  SQL
                Open  cursor-name
END-EXEC.
```

2.3. FETCH Statement to Read one Row

For an open cursor, the FETCH statement gets one row at a time to the program. The format is :

```
EXEC SQL
        FETCH  cursor-name
        INTO:data-name1:data-name2
END-EXEC
```

Note that SQL CODE is equal to 100 if there are no more rows ("no record found" condition).

2.4. Closing the Cursor at End of Processing

Once the program is finished with processing the "temporary cursor table" (or at any time), the CLOSE cursor-name statement is executed. The format is :

```
EXEC  SQL
                CLOSE Cursor-name
END-EXEC
```

2.5. Basic Program Logic : Multiple Rows

The basic program logic is figure 12.8

The following apply :

1. The DECLARE cursor-name statement identifies the cursor, the columns selected, and the search criteria.

   a) The column-names in the SELECT clause (EMPNAME, EMPSALARY) are those of the table.

   b) Note that if the WHERE clause is missing (as we will show in the program example in Fig. 12.9), we are reading the whole table.

2. We set the Cobol data-name SEARCH-=DEPT to the correct value since it is used as the search criteria.

```
WORKING STORAGE SECTION
                ….
                EXEC SQL
                        INCLUDE  member-name  (from DCLGEN output)
                END EXEC.
                …..
                EXEC   SQL
                        INCLUDE  SQLCA
                END-EXEC.
                …..
                PROCEDURE DIVISION
                ….
                EXEC SQL
                        DECLARE Cursor-name  CURSOR FOR
                                SELECT EMPNAME.EMPSALARY
                        FROM XLIM. EMPTABLE
                                WHERE EMPDETP = SEARCH-DEPT.

                END-EXEC
        ***     Set SEARCH-DEPT to the correct value  ***
                EXEC SQL
                                OPEN cursor-name
                END-EXEC.
        PERFORM  C200-PROCESS-EMPLOYEE-ROW UNTIL SQLCODE=100
                EXEC SQL
                                CLOSE  cursor-name
                END-EXEC.
        C200-PROCESS-EMPLOYEE-ROW
                EXEC SQL
                        FETCH  cursor-name
                                INTO:EMPNAME
                END-EXEC.
                IF  SQLCODE TO EQUAL TO ZEROS
                                OK  Contine
```

Figure 12.8 Program logic : multiple-row selection

3. As part of the Main-loop routine, we open the cursor, perform the processing of all rows, then close the cursor.

4. The FETCH statement has the INTO clause, which brings the data for the selected columns into the Cobol data-names specified. Thus, :EMPNAME and EMPSALARY are the Cobol data-names generated by DCLGEN

5. We check the SQLCODE field and if zero, we know the FETCH was successful and we may continue, if the value is 100, then this is the "end of file" condition.

2.6. Program Example : Multiple-row selection

We will use Figure 12.2. as the input.

2.6.1. Program listing : multiple-row selection.

We will now print all rows in the XLIM.EMPTABLE Employee table. Figure 12.9 is the program listing. The following apply :

1. Lines 001100 to 001800 define the print file.

2. Lines 002000 to 005100 are the various data-names such as counters and the print lines.

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAN-ID. PROG1207.
000300 *********************************************************************
000400 *                                                                   *
000500 *  1.  THIS PROGRAN PRINTS ALL ROWS IN THE EMPLOYEE TABLE.           *
000600 *                                                                   *
000700 * *******************************************************************
000800   ENVIRONMENT DIVISION
000900   CONFIGURATION SECTION
001000   INPUT-OUTPUT SECTION
001100   FILE-CONTROL
001200            SELECT PRINT-OUTPUT          ASSIGN TO       LSTOUT.
001300   DATA DIVISION
001400   FILE SECTION
001500   FD                 PRINT-OUTPUT
001600                      BLOCK CONTAINS  0 RECORDS
001700                      LABEL  RECORDS OMITTED
001800   01                 PRINT-RECORD                       PIC X  (133)
001900   WORKING-STORAGE  SECTION
002000   01                 W005-LINE-COUNT                    PIC S9 (8) COMP VA LUE +99
002100   01                 W005-LINE-LIMIT                    PIC S9 (8) COMP VALUE +55
002200   01                 W005-LINE-SKIP                     PIC  99
002300   01                 W005-SALARY-INDV                   PIC S9 (4)  COMP.
002400   01                 W005-END-OF-ROWS                   PIC X VALUE  'N'
002500                      88                 W005-NO-MORE-ROWS
002600   01                 W005-DETAIL-LINE
002700                      05       FILLER                    PIC X (4)
```

```
002800              05     W005-DETAIL-EMPNO               PIC X (5)
002900              05     FILLER                          PIC X (4)
003000              05     W005-DETAIL-EMPNAME             PIC X (30)
003100              05     FILLER                          PIC X  (2)
003200              05     W005-DETAIL-EMPDEPT             PIC XXX
003300              05     FILLER                          PIC X (3)
003400              05     W-005-DETAIL-EMPSALARY          PIC Z, ZZZ, ZZ9.
003500              05     W-005-DETAIL-EMPSALARY-ALPHA
003600                     REDEFINES W005-DETAIL-EMPSALARY
003700                                           PIC X (9)
003800   01         05     W005-HEADER-LINE1
003900              05     FILLER         PIC X (3)        VALUE SPACES
004000              05     FILLER         PIC X (8)        VALUE 'EMPLOYEE'
004100              05     FILLER         PIC X (34)       VALUE SPACES
004200              05     FILLER         PIC X (5)        VALUE
004300   01         W005-HEADER-LINE2
004400              05     FILLER         PIC X (4)        VALUE SPACES
004500              05     FILLER         PIC X (6)        VALUE 'NUMBER'
004600              05     FILLER         PIC X (8)        VALUE SPACES
004700              05     FILLER         PIC X (4)        VALUE  'NAME'
004800              05     FILLER         PIC X (22)       VALUE SPACES
004900              05     FILLER         PIC X (6)        VALUE 'NUMBER'
005000              05     FILLER         PIC X  (4)       VALUE SPACES
005100              05     FILLER         PIC X (6)        VALUE 'SALARY'
005200              05     EXEC SQL
005300              05            INCLUDE  SQLCA
005400              05     END-EXEC.
005500              05     EXEC SQL
005600              05            INCLUDE  EMPTABDL
005700              05     END-EXEC.
```

Figure 12.9 Program listing : multiple-row selection.

3. Lines 005200 to 005400 generate the communication area data block shown in Figure 10.1 1 on page 62.

4. Lines 005-00 to 005700 include the DCLGEN output shown in Figure 12.1.

5. Lines 005900 to 006600 implement the DECLARE statement that specifies the cursor name (EMPCSR), the columns selected, and usually also the WHERE

```
005800         PROCEDURE DIVISION
005900              EXEC  SQL
006000                     DECLARE EMPCSR  CURSOR  FOR
006100                                   SELECT  EMPNO,
006200                                                     EMPNAME,
006300                                                     EMPDEPT,
006400                                                     EMPSALARY
006500                                         FROM XLIM.EMPTABLE
006600              END-EXEC.
006700              EXEC  SQL
006800                     OPEN  EMPCSR
006900              END-EXEC.
007000              OPEN OUTPUT  PRINT-OUTPUT
007100              MOVE SPACES TO  W005-DETAIL-LINE
```

```
007200            PERFORM  C100-FETCH-ONE-ROW
007300            PERFORM  C020-PROCESS-ALL-ROWS
007400                 UNTIL  W005-NO-MORE-ROWS
007500            EXEC  SQL
007600                 CLOSE  EMPCSR
007700            END-EXEC
007800            CLOSE PRINT-OUTPRINT
007900            DISPLAY  ***  END OF JOB PROG 1207'  UPON SYSOUT
008000            GOBACK
008100  C020-PROCESS-ALL-ROWS
008200            IF  SQLCODE EQUAL TO ZERO
008300                 PERFORM  C040-LAYOUT-ROW-COLUMNS
008400                 PERFORM  C060-PRINT-DETAIL-LINE
008500            PERFORM  C100-FETCH-ONE-ROW
008600  C040-LAYOUT-ROW-COLUMNS.
008700            MOVE  EMPNO         TO             W005-DETAIL-EMPNO.
008800            MOVE  EMPNAME       TO        W005-DETAIL-EMPNAME
008900            MOVE EMPDEPT        TO        W005-DETAIL-EMPDEPT.
009000            IF W005-SALARY-INDV           LESS THAN ZERO
009100  MOVE  'NULLS  ***'            TO        W005-DETAIL-EMPSALARY-ALPHA
009200  ELSE MOVE EMPSALARY  TO    W005-DETAIL-EMPSALARY
009300  C060-PRINT-DETAIL-LINE.
009400            IF  W005-LINE-COUNT GREATER THAN W005-LINE-LIMIT
009500                      PERFORM  C080-PRINT-HEADER-LINES.
009600       WRITE PRINT-RECORD FROM W          W005-DETAIL-LINE
009700                 AFTER  ADVANCING 2005-LINE-SKIP LINES.
009800       ADD      1 TO  W005-LINE-COUNT
009900       MOVE   1  TO  W005-LINE-SKIP
010000  C080-PRINT-HEADER-LINES
010100       WRITE PRINT-RECORD FROM  W005-HEADER-LINE1
010200                                AFTER  ADVANCING PAGE
010300       WRITE PRINT-RECORD FROM W005-HEADER-LINE2
010400                      AFTER ADVANCING  2 LINES
010500       MOVE ZEROES            TO W005-LINE-COUNT
010600       MOVE   2               TO    W005-LINE-SKIP
010700  C100-FETCH-ONE-ROW
010800       EXEC  SQL
010900                 FETCH  EMPCSR
011000                      INTO     :EMPNO,
011100                                :EMPNAME,
011200                                : EMPDEPT,
011300                                : EMPSALARY:W005-SALARY-INDV
011400            END-EXEC.
011500            IF SQLCODE = 100
011600                 MOVE  'Y'  TO  W005-END-OF-ROWS
```

Figure 12.9 (contuned)

Clause as the row selection criteria, however there is no WHERE clause and we are in fact selecting all rows.

6. lines(x) 7600to 006900 open the cursor.
7. Lines 007000 to 007400 are the housekeeping routines and the Main – loop control.

8. Lines 007500 to 007700 close the cursor.
9. Lines 008100 to 008500 control the processing of each row as we select each one is sequence.
   a. Lines 008200 to 008400 perform the laying out and printing of the columns if there was a row selected; if not (negative SQULCODE value), there is nothing to process.
   b. Line 008500 fetches another row from the table.

10. Lines 008600 to 009200 lay out the row columns (from line 008200, only if a row was selected.
    a. Lines 008700 to 008900 simple layout the columns that will never contain nulls (seefig.12.1).
    b. Lines 009000 to 009200 check the w005-SALARY –INDV indicator variable to see if DB2 detected nulls (a negative value for W005-SALARY-INDV) for the EMPSALARY column (see line 011300). If so, we display the 'NULLS' literal instead of the actual salary value.

11. Lines 009300 to 010600 are the routines to print the details and header lines.

12. Lines 0101800 to 011400 implement the statement to fetch one row. Note that we use the INTO clause, which brings the row selected into the date names specified in the SELECT clause of theDECLARE cursor-name statement (lines 005900 to 006600).
    a. lines 011000 to 011300 specify the Cobol date-names that will get the value from the row. Note that each is prefixed with a colon.
    b. Line 011300 shows the use of the indicator variable for a column that may contain nulls; if not specified, there is an SQULCODE error if the column does contain nulls.

13. Lines 011500 to 011600 set the "end of row" switch once SQLCODE becomes 100.

2.6.2. Additional JCL statement to be included.

Assuming that we use "EDITJCL" in entry 3 of Figure 11.5 in Chapter 11,DB2 will generate most of the JCL statement for us. However, we still have to include those for the batch files.

For our program example, these are for the display output and print file. Thus:

```
//SYSOUT    DD    SYSQUT=*
//LSTOUT    DD    SYSOUT=*
```

2.6.3 Out put listing: multiple-row selection.

Figure 12,10 is the out put of the program in figure 12.9.

Note the following:

1. All seven rows are printed out.
2. For employee numbers’00010’ and ‘00170’, we print out ‘NULLS’ in the salary column.

```
Employee                                DEPT
Number          NAME                    NUMBER          Salary

00005           BAKER,C.                003             30,000
00008           RANDOLPH,R              005             29,000
00010           RICHARDS,M.             002             NULLS***
00015           DAVIS,L.                006             30,000
00150           ELLIOT,L.               005             31,000
00170           RABINOVITZM.            003             NULLS***
00190           LEE,R.                  005             30,000
```

Figure 12.10 output listing: multiple-row selection

1 UPDATE USING THE CURRENT CURSOR

In most types of update, the programmer first selects a row before it is updated. This is because the row is printed out, verified, and so on before the update is made. This type of processing uses many of the same cursor processing techniques shown in the previous chapter.

1.1 the DECLARE statement

Since we are processing using a cursor, we must also code the DECLARE statement in the same manner as in Chapter 12. The format is:

```
        EXEC SQL
                DECLARE CURSOR-NAME CURSOR FOR
                                SELECT EMPNAME, EMPSALARY
                                FROM XLIM, EMPTABLE
                                WHERE EMPDEPT=: SEARCH-DEPT
                                FOR UPDATE OF COLUMN-NAMEL,
                                                                COLUMN-NAME2,
END-EXEC.
```

The “FOR UPDATE OF …” clause is an additional clause that specifies the column(s)to be updated. Note also that this statement merely defines the criteria for update; the actual update is done only with the UPDATE statement.

1.2 The OPEN/FETCH/ CLOSE Cursor-name Statements

the UPDATE or DELETE statement, as with any other update or delete statement, may update or delete multiple rows if we use the "WHERE column-name=date-name" clause and there are multiple rows with the value name of data-name. We have seen this using SPUFI and we explain its use in programs in Chapter 14.

In this chapter, we deal with the common situation where we update / delete the same row we have just fetched, without updating/deleting any other row. This is done by coding the 'WHERE CURRENT OF cursor-name" clause instead of the "WHERE column-name=; date-name' clause in the UPDATED/DELETE statement that updates or deletes the row brought into the program by the last FETCH statement.

1.3.1 Format of UPDATE using the current cursor.

The format of the statement is:

```
EXEC SQL
        UPDATE XLIM. EMPTABLE
                SET column 1= . . . . .
                        Column2= ……….
        WHERE CURRENT OF CURSOR-names
END-EXEC.
```

The "WHERE CURRENT OF cursor-name" clause cause the update of the very same row brought into the program by the last FETCH statement.

1.3.2 UPDATE/ DELETE USING CURSOR AND temporary cursor table

In chapter 12 we explained the difference between the conceptual and physical temporary cursor tables. When using the "WHERE CURRENT OF cursor-name " clause, only the conceptual temporary cursor table may be used. This means that the program must use the original table.

If DB2 creates a physical temporary cursor table then the "WHERE CURRENT OF cursor-name " clause cannot be used, since the cursor would then be pointing to the former and not the original table. In this case, to update/delete rows in the original table, we have to use instead the 'WHERE column-name =; data-name" clause. However, we cannot then guarantee that we are updating /deleting only one row(as we could if we Update/delete using the current cursor), unless the value of column-name is unique.

1.4 SQLCODE of 0 or 100.

As before, if SQLCODE is 0, then the statement had no error.

In an update/ delete not using the WHERE CURRENT OF CLUASE, this mean there was now row updated or deleted, In an update using the WHERE CURRENT OF clause, this means that the corresponding FETCH statement resulted in the "end of rows" condition.

1.4.2. SQLCODE on too many pages for user.

Many installations limit the number of pages that a given user can own (lock) at any given time. This is the NUMLKTS parameter specified when DB2 is installed.

1.4.2.1 SQLOCODE if LOCKSIZE PAGE.

If the user has chosen LOCK-SIZE PAGE and DB2 has determined that a user already owns lock on the maximum number of pages, it will not execute any further statement that will cause another page lock and instead generates an SQLCODE value of –904 . The user should then either issue a COMMIT statements (SYNCPOINT command in CICS/VS) or ROLLBACK statement (SYNCPOINT ROLLBACK command in CICS/VS) before continuing.

In most cases, the programmer should just commit, then continue, using the current input that caused the SQLCODE value of –904.

Note that this problem occurs only for page locking, if we choose LOCKSIZE PAGE not LOCKSIZE ANY, in most situations, LOCKSIZE ANY is actually preferred since we then allow DB2 to choose the lock size, usually starting with page lock, but with the possibility it to table space lock.

1.4.2.2 SQLCODE if LOCK SIZE ANY.

On the other hand, if the user has chosen LOCKSIZE ANY and DB2 has determined that a user already owned lock on the maximum number of pages, on the next statement that would otherwise cause another a the page lock , it will simply promote the page lock to table space lock, with the user not even aware of it. Processing continues as usual.

1.4.3. SQLCODE on dead lock condition.

The dead lock conditions is very well known, even in nondatabase applications. In DB2 it can occur even more frequently, because there are so many users, each one having locks on pages scattered all over the table space and needing other pages that other users may have locks on.

Figure13.1 shows the deadlock condition

Depending on certain conditions, DB2 will choose which user will be sacrificed, and which one allowed to continue. For the program to be sacrificed, it will either automatically have its pages rolled back, with an SQLCODE of –911, or have an SQLCODE of –913, with the program itself doing the roll back.

PROGRAMM ACTION	DB2 ACTION
1. user A executes update on pagel	Update is allowed and DB2 notes that user A now has EXCLUSIVE lock on page1.
2……(other activities	
3. User B executes up9date of page3.	Update is allowed and DB2 notes that user B now has EXCLUSIVE lock on page 3.
4………(other activities)	
5.user A executes update on pages 6.	Update is allowed and DB2 notes that user A now has EXCLUSIVE lock on page6.
6……..(other activities)	
7. user B executes update on page9.	Update is allowed and DB2 notes that user B now has EXCLUSIVE lock on page9.
8…….(other activities)	
9. user A executes update on page3	Update is disallowed since DB2 note that user B has EXCLUSIVE lock on page3. User a waits for the lock to be released.
10……..(other activities)	
11. user B executes update on pagel.	Updaste is disallowed since DB2 notes that user A has EXCLUSIVE lock on pegal. User B waits for the lock to be released.
12. this is a deadlock conditions	Both user A and B cannot continue until locks are released .DB2 sacrifices either user A or user B.

Figure 13.1 the deadlock condition

1.5 Program logic Update the Current Row

The logic is figure 13.2.

Note the following.

1. The DECLARE cursor-name statement identifies the cursor, the columns read into the program, the search criteria, and the FOR UPDATE OF clause that specifies the column(s) that are updated.

a. The column names in the SELECT clause (EMPNAME, EMPSALARY) ARE those of the table. These are the columns we want read into the program.

b. Any column name in FOR UPDATE OF clause does not have to appear as a column name in the SELECT clause. Those in the former are simply the columns we want update.

1) If the column to be updated is simply replaced with new values independent of the current value, then there is no need to bring it first into the program for the update to be successful (although we may still have to do so for some other reasons, such as wanting to print the original value).

```
WORKING STORAGE SECTION.
            ……….…
    EXEC SQL
            INCLUDE member-name (from DCLGEN out put)
    END-EXEC
    EXEC SQL
            INCLUDE SQLCA
    END-EXEC.
            ………..
PROCEDURE DIVISION.
    EXEC SQL
            DECLARE CURSOR-NAME CURSOR FOR
                    SELECT EMPNAME, EMPSALARY
                            FROM XLIM. EMPTABLE
                    WHERE EMPDENT =; SEARCH-DEPT
                    FOR UPDATE OF EMPSALARY

    END-EXEC.
*** set SEARCH-DEPT to the CORRECT VALUE***
    EXEC SQL
            OPEN CURSOR-NAME
END-EXEC.
PERFORM C200-PROCESS-EMPLOYEE-ROW UNTIL SQLCODE=100.
EXCE SQL
    CLOSE Acursor-name
END-EXEC,.
```

Figure 13.2 Program logic; update the current row.

2) If the original value of the column to be updated is needed in the program, then we have to bring it to the program before the update is made.

2. We set the Cobol data –name SEARCH –DEPT to the correct value. It will be used as the search criteria.

3. As part of the Main- loop routine , we open the cursor, process all rows, then close the cursor.

4. The FETCH statement has the INTO clause, which brings the data for the selected columns into the Cobol date-names specified. Thus, ;EMPNAME and ;EMPASALARY are the coblol data-names generated by DCLGEN (see fig. 12.1)

1.6. Program Example: Update the Current Row

We not run a program that reads “card images” from an in-line card files, which contain department numbers and salary updating information. For each department number read, we process corresponding departments in the XLIM,EMPTABLE Employee table, For each row belonging to a department, we increase the salary column by the salary undating information in the “card”. Except when the original salary value is NULLS, where no update is done.

	Employees		Department	
	Number	**Name**	**Number**	**Salary**
	(EMPNO)	(EMPNAME)	(EMPDEPT)	(EMPSALARY)
1.	00005	BAKER,C.	003	0030000
2.	00008	RANDOLPH,R.	005	0029000
3.	00010	RICHARDS,M.	002	NULLS
4.	00015	DAVIS,L.	006	0030000
5.	00150	ELLIOT,T.	005	0031000
6.	00170	RABINVOVITZ,M.	003	NULLS
7.	00190	LEE,R.	005	003000

Figure13.3 Current data on the XLIM, EMPTABLE employee table.

Figure 13.3. is the data used as input to the update.

1.6.1 Program listing; undate the current row.

The program shown in figure 13.4 shows that for the SQLCODE, we only check for the “end of table” (SQLCODE 100) and deadlock conditions (sqlcode-911 or-913). We are not concerned with the condition of number of pages going beyond the maximum allowed (SQLCODE-904) since we assume LOCKSIZE ANY.

The following apply:

1. Lines 001200 to 001800 pertain to the input file containing the department number and salary updating information.

b. Lines OOZOOO to 002600 are the various data-names s-ch as counters and switches.

3. Lines 002700 to 002900 generate the communication area data block shown in Figure 10.1.

4. Lines 003000 to 003200 include the DCLGEN out put shown in Figure12.1.

5. Lines 003400 to 004000 implement the DECLARE statement that specifies the cursor name (EMPCSR), the columns selected, the row selection criteria, and the clause'FOR UPDATE OF "column-name".

   a. Line 003600 specifies the column we want read into the program, Note that in general, this column(s) is naturally the one(s) we want to process(say to print) in the program. It does not have to be the same columns (s) that is updated (FOR UPDATE OF clause), although in our example it just happens to be the same.
   b. Line 003900 specifies the column to be updated.

6. Lines 004100 to 004700 are the housekeeping routines and the Main –loop control.

7. Lines 004800 to 007100 process each department read in the in-line card file.

   a. Lines 004900 to 005100 open the cursor.

   b. Line 005200 fetches the first row belonging to the department.

   c. Lines 005300 to 005500 display an error message if the department does not have a row in the table.

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID. PROG1304.
000300 ************************************************************************
000400 *  1.  THIS PROGRAM UPDATES ROWS VIA A CURSOR                          *
000500 *                                                                      *
000600 *                                                                      *
000700 * **********************************************************************
000800    ENVIRONMENT DIVISION
000900    CONFIGURATION SECTION
001000    INPUT-OUTPUT SECTION
001100    FILE-CONTROL
001200              SELECT UPDATE-INPUT            ASSIGN TO CARDIN
```

```
001300                    DATA  DIVISION
001400            FILE SECTION
001500  FD        UPDATE-INPUT
001600            BLOCK CONTAINS      0         RECORDS
001700            LABEL RECORDS OMITTED
001800  01                UPDATE-RECORD                   PIC X (80)
001900  WORKING-STORAGE-SECTION
002000  01                W005-SALARY-INDV                PIC S9 (4)  COMP.
002100  01                W005-UPDATE-RECORD.
002200                    05          W005-EMPDATE        PIC X (3)
002300                    05          FILLER              PIC XX..
002400                    05          W005-SALARY-INCREASE PIC S9 (5)
002500                    W005-END-OF-UPDATE-RECORDS      PIC X VALUE 'N'
002600                    88          W005-NO-MORE-UPDATES        VALUE  'Y'
002700                    EXEC  SQL
002800                            INCLUDE  SQLCA
002900                    END-EXEC.
003000                    EXEC  SQL
003100                            INCLUDE  EMPTABDL
003200                    END-EXEC.
003300  PROCEDURE DIVISION.
003400                    EXEC SQL
003500                                DECLARE EMPCSR  CURSOR  FOR
003600                                        SELECT EMPSALARY
003700                                                FROM XLIM.EMPTABLE
003800                                WHERE EMPDPT = W005-NO-MORE-UPDATES.
003900                                FOR UPDATE OF EMPSALARY
004000                    END-EXEC.
004100                    OPEN INPUT  UPDATE
004200                    PERFORM  C120-READ-UPDATE-INPUT
004300                    PERFORM C020-PROCESS-EACH-DEPT
004400                                UNTIL  W005-NO-NORE UPDATES.
004500                    CLOSE UPDATE-INPUT
004600                    DISPLAY '*** END OF JOB  PROG1304'  UPON SYSOUT.
004700                    GOACK.
```

Figure 13.4 Program listing : update the current row.

d. Line 005600 processes all rows otherwise.

e. Lines 005700 to oo6400 display an error message if the department processing is interrupted by DB2 due to a deadlock conditions. If SQLCODE is –913, 23 do a rollback.

f. Lines 006500 to 007000 execute aftrer processing all rows for a department (SQLCODE noW 100). It close the clusor, then does a commit, here, we commit for every department updated (that is, multiple rows belonging to the department.)

8. lines 007200 to 007500 process departments with at least one row until there are no more rows or DB2 determine a deadlock.

9. Lines 007700 to 007800 show that if the salary value contains NULLs (from line 009100), we don’t do any update.

```
004800 020-PROCESS-EACH-DEPT
004900     EXE  SQL
005000         OPEN EMMPCSR
005100     END-EXEC
005200     PERFORM   C-100  FETCH-ONE-ROW
005300     IF  SQLCODE = 100
005400             DISPLAY  ‘PROG1304  --  DEPT  ‘W005-EMPDEPT
005500                 ‘HAS NO TABLE DATE’  UPON SYSOUT
005600     ELSE PERFORM  C040-PROCESS-GOOD-DEPT.
005700     IF  SQLCODE   =  (-911  OR  -913)
005800             DISPLAY  ‘PROG1304  -  DE (T  ‘W005-DMPDEPT
005900                 ‘BYPASSED DUE TO DEADLOCK’  UPON SYSOUT
006000             IF  SQLCODE =  -913
006100                     EXEC  SQL
006200                         ROLLBACK
006300                     END-EXEC
006400             ELSE NEXT SENTENCE
006500             ELSE  EXEC  SQL
006600                 CLOSE  EMPCSR
006700             END-EXEC
006800             EXEC  SQL
006900                     COMMIT
007000             END-EXEC.
007100     PERFORM  C-120-READ-UPDATE-INPUT
007200 C040-PROCESS-GOOD-DEPT.
007300     PERFORM C060-PROCESS-EACH-DEPT-ROW UNTIL SQLCODE = 100
007400                             OR  SQLCODE =  -911
007500                             OR  SQLCODE = -913
007600 C060-PROCESS-EACH-DEPT-ROW
007700     IF  W005-SALARY-INDV LESS THAN  ZERO
007800             NEXT  SENTENCE
007900     ELSE PERFORM  C080-UPDATE-THIS-ROW
008000     PERFORM C100-FETCH-ONE-ROW
008100 C080-UPDATE-THIS-ROW
008200     ADD W005-SALARY-INCREASE TO  EMPSALARY.
008300     EXEC SQL
008400             UPDATE  XLIM.EMPTABLE
008500                 SET EMPSALARY = :EMPSALARY
008600                 WHERE CURRENT OF EMPCSR
008700     END-EXEC.
008800 C100-FETCH-ONE-ROW
008900     EXEC  SQL
009000         FETCH  EMPCSR
009100                     INTO :EMPSALARY:W005-SALARY-INDV
009200     END-EXEC
009300 C120-READ-UPDATE-INPUT
009400     READ UPDATE-INPUT-INTO   W005-UPDATE-RECORD
009500             AT  END, MOVE   ‘Y’  TO    W005-END-OF-UPDATE-RECORDS
```

Figure 13.4 continued)

10. Lines 008100 to 008700 execute the update.

    a. Lines 008200 adds the salary updating data from the in-line card file to the original salary data from the row(from line 009100).

    b. Lines 008300 to 008700 update the current row (from line 008600), the columns updated is EMPSALARY, which picks up the data in the data-name EMPSALARY (now containing the correct value.)

11. Lines 008800 to 009200 fetch on row from the table. The EMPSALARY data from the row (from line 003600) are brought to the data-name EMPSALARY.

12. LINES 009300 TO 009500 read the in-line card file for the department number and the corresponding salary updating information.

1.6.2 Additional JCL statements to be included.

Assuming that we use "EDITJCL" in entry 3 of firure 11.5 in chapter 11,DB2will generate most of the JCL statements-for us. However, we still have to include those for the batch files.

In this program example, these are;

```
/ / SYSOUT DD SYSOUT =        *
/ / CARD  I  N DD
0 0 5  00500
0 0 3  00 700
```

1.6.3 Updated table: update the current row,

Figure 13.5 is the updated table from the program in Figure 13.4

	Emplyee Number (EMPNO)	Name (EMPNAME)	Department Number (EMPDEPT)	Salary (EMPSALARY)
1.	00005	BAKER,C.	003	0030700
2.	00008	RANDOLPH, R.	005	0029000
3.	00010	RICHARDS, M.	002	NULLS
4.	00015	DAVIS, L.	006	0030000
5.	00150	ELLIOT, T.	005	0031500
6.	00170	RABINOVITZ, M.	003	NULLS
7.	00190	LEE, R.	005	0030500

Figure 13.5 updated table; update the current row.

Note that for department numbers '005' and '003'. We update all rows, except those with salary value of NULLS, all rows for department number '005' and '003' had their salary value updated by $ 500 and $700 respectively.

1.7. COMMIT Using cursor processing

Lines 006800 to 007000 in Figure 13.4 committed update changes, in our example, we did this for all rows belonging to a department. The programmer should issue the COMMIT statement when feasible to optimize concurrency. However, when processing using cursors, any commit or rollback automatically closes all open cursors.

1.7.1 COMMIT with multiple open/close cursor.

If there are too many updates between the OPEN/CLOSE of single cursor, the program may be holding on to too many pages, with the ensuring problems of lower concurrence and deadlock potential. To avoid this problem, the programmer may be forced to commit changes even before he or she is finished with the temporary cursor table. This of course automatically closes the cursor.

If the program now opens another cursor, the DECLARE statement would still generate the same temporary cursor table and we would be processing what we processed before. However, the programmer may actually be able to continue normal processing (that is, access the correct rows when we continue) by using the “trick shown in the nest section.
This ‘trick’ requires the rows to be in sequence. If the search criteria (WHERE clause) specifies a clustered index, then the rows are always in sequence; otherwise the programmers has to specify the ORDER BY clause, However, in the latter case DB2 creates a physical temporary cursor table, in which case the ’WHERE CURRENT OF cursor-name clause cannot be used, since the cursor would then be pointing to the former and not the original table. To update/ delete rows in the original table, we have to use instead the ‘WHERE column-name=; data-name” clause. However, we cannot then guarantee that we are updating /deleting only one row(as we could if we update/delete using the current cursor), unless the value of column-name is unique.

1.7.1.1. program logic; COMMIT with multiple open .close cursor.

Figure 13.6 shows the program logic.

```
WORKING-STORAGE SECTION
        ……..
        EXEC  SQL
                INCLUDE  SQLCA
        END-EXEC.
        EXEC  SQL
                INCLUDE  EMPTABDL
        END-EXEC.
        EXEC  SQL
                INCLUDE  EMPTABDL
        END-EXEC.
PROCEDURE  DIVISION
        ……..
EXEC  SQL
                DECLARE  cursor-name  CURSOR  FOR
                        SELECT  EMPNO,  EMPSALARY
                                        FROM  XLIM.EMPTABLE
                                        WHERE  EMPNO  >  :W005-EMPNO-UPDATED
                                        FOR  UPDATE OF EMPSALARY
        END-EXEC.
        MOVE  ‘00000’  TO  W005-EMPNO-UPDATED
        PERFORM  C020-PROCESS-ALL-ROWS  UNTIL  W005-NO-NORE-ROWS.
        GOBACK.
C020-PROCESS-ALL-ROWS
        MOVE  ZEROS  TO  W005-NUMBER-ROWS-UPDATED
        EXEC  SQL
                OPEN  cursor-name
        END-EXEC.
        PERFORM  C080-FETCH-ONE-ROW
        PERFORM  C040-PROCESS-GROUP-OF-ROWS  UNTIL  W005-NO-NORE-ROWS

                                        OR   W005-NUMBER-ROWS-UPDATED  =  3.
                EXEC  SQL
                        COMMIT
                END-EXEC
                EXEC  SQL
                        CLOSE  cursor-name
                END-EXEC.
C040-PROCESS-GROUP-OF-ROWS.
                ADD     W005-SALARY-INCREASE  TO  EMPSALARY.
                EXEC  SQL
                        UPDATE  XLIN.EMPTABLE
                        SET  EMPSALARY  =  :EMPSALARY
                        WHERE  CURRENT OF cursor-name

                END-EXEC
                ADD  1  TO  W005-NUMBER-ROWS-UPDATED
C080-FETCH-ONE-ROW
                EXEC  SQL
                        FETCH  cursor-name
                        INTO   :W005-EMPNO-UPDATED
                                        :EMPSALARY:005-SALARY-INDV

                END-EXEC.
                IF  SQLCODE  =  100
                        MOVE    'Y'   TO  W005-END-OF-ROWS.
```

Figure 13.6. Program logic : COMMIT with multiple open/close cursor.

The following apply:

1. The DECLARE cursor-name statement identifies the cursor, the columns(s) read into the program, the search criteria, and the FOR UP DATE OF clause that specifies the columns(s) that are updated.

    a. The column-names in the SELECT clause (EMPNAME, EMPSALARY) are those of the table.

    b. The columns-names in the FOR UPDATE OF clause may be different from those in the SSELECT clause.

    c. The WHERE clause makes sure that any OPEN cursor-name statement gets the row where the value of EMPNO is greater than the value of the data-name W005-EMPNO-UPDATED (originally set to '00000') by putting the value of the last row fetches into WOOS-EMPNO-UPDATED, we can process the whole table correctly even with repeated open and close of the cursor.

2. The PROCESS-ALL-ROWS and PROCESS –GROUP-OF-ROWS paragraphs count the number of rows actually updated, then commit and close the cursor when we have reached the number we want (in this example, three rows).

3. The FETCH statement has the INTO clause, which brings the data for the selected columns into the Cobol data-names specified. The use of w005-EMPNO-

UPDATED brings the employee number of the latest row fetched into this data-name.

1.7.1.2 program listing; COOMIT with multiple openiclose cursor.

We will run a program that adds $ 500 to all salary columns, except when the original value is NULLS. We use figure 13.5 as the input. The program listing in figure 13.7.

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID. PROG1307.
000300 *********************************************************************
000400 *                                                                   *
000500 *  1.  THIS PROGRAM UPDATES ROWS VIA A CURSOR                       *
000600 *                                                                   *
000700 *  2.  UPDATE IS VIA MULTIPLE OPEN / CLOSE OF CURSOR.               *
000800 *                                                                    *
000900   *********************************************************************
001000   ENVIRONMENT DIVISION
001100   CONFIGURATION SECTION
001200   INPUT-OUTPUT SECTION
001300   DATA DIVISION
001400   FILE SECTION
001500   WORKING-STORAGE SECTION
001600   01           W005-SALARY-INDV                PIC  S9 (4)  COMP.
001700   01           W005-EMPNO-UPDATED              PIC X (5)
001800   01           W005-EMPNO-LOW RANGE            PIC X (5)
001900   01           W005-NUMBER-ROWS-UPDATED        PIC S9 (8)  COMP.
002000   01           W005-SALARY-INCREASE            PIC S9 (5)  COMP-3 VALUE+500
002100   01           W005-END-OF-ROWS                PIC X VALUE ‘N’
002200                88            W005-NO-MORE-ROWS
002300                EXEC SQL
002400                              INCLUDE  SQLCA
002500                END-EXEC.
002600                EXEC  SQL
002700                              INCLUDE  EMPTABDL
002800                END-EXEC.
002900   PROCEDURE  DIVISION.
003000                EXEC  SQL
003100                              DECLARE  EMPCSR  CURSOR FOR
003200                              SELECT EMPNO,  EMPSALARY
003300                                     FROM XLIM.EMPTABLE
003400                                     WHERE EMPNO > :W005-EMPNO-UPDATE
003500                              FOR UPDATE OF EMPSALARY
003600                END-EXEC.
003700                MOVE  ‘00000’  TO W005-EMPNO-UPDATED
003800                PERFORM  C020-PROCESS-ALL-ROWS
003900                              UNTIL W005-NO-MORE-ROWS
004000          DISPLAY  ‘**** END OF JOB PROG1307’  UPON SYSOUT.
004100          GOBACK
004200   C020-PROCESS-ALL-ROWS.
004300          MOVE  ZEROS TO W005-NUMBER-ROWS-UPDATED
004400          EXEC SQL
004500                        OPEN EMPCSR
004600          END-EXEC.
004700          PERFORM  C040-PROCESS-GROUP-OF-ROWS
004800                        UNTIL 2005-NO-MORE-ROWS
004900                        OR SQLCODE  =  (-911 OR  -913)
005000                              OR W005-NUMBER-ROWS-UPDATED = 3)
```

```
005100      IF SQLCODE =  (-911 OR –913)
005200          DISPLAY ‘PROG1307 --- EMPLOYEE ‘W005-EMPNO-LOW-RANGE
005300          ‘TO’   W005-EMPNO-UPDATED
005400              ‘BYPASSED DUE TO DEADLOCK’ UPON SYSOUT
005500          IF SQLCODE =  -913
005600              EXEC  SQL
005700                          ROLLBACK
005800                          END-EXEC
005900          ELSE NEXT SENTENCE
006000      ELSE EXEC SQL
006100                  CLOSE EMPCSR
006200          END-EXEC
006300          EXEC SQL
006400              COMMIT
006500          END-EXEC.
006600  C040-PROCESS-GROUP-OF-ROWS
006700          PERFORM  C080-FETCH-ONE-ROW
006800          IF  SQLCODE     EQUAL TO            ZERO
006900              AND  W005-SALARY-INDV      NOT LESS THAN    ZERO
007000                  PERFORM  C060-UPDATE-THIS-ROW.
007100  C060-UPDATE-THIS-ROW
007200      IF W005-NUMBER-ROWS-UPDATED  EQUAL TO ZERO
007300          MOVE  W005-EMPNO-UPDATED  TO  W005-EMPNO-LOW-RANGE
007400      ADD  W005-SALARY-INCREASE TO EMPSALARY
007500      EXEC  SQL
007600              UPDATE XLIM.EMPTABLE
007700                  SET EMPSALARY  =  :EMPSALARY
007800                  WERE CURRENT OF EMPCSR
007900      END-EXEC
008000      ADD     1   TO  W005-NUMBER-ROWS-UPDATED
008100  C080-FETCH-ONE-ROW.
008200      EXEC  SQL
008300              FETCH  EMPCSR
008400                  INTO :  W005-EMPNO-UPDATED
008500                      :EMPSALARY:W005-SALARY INDV
008600          END-EXEC
008700          IF  SQLCODE   = 100
008800              MOVE  ‘Y’  TO  W005-END-OF-ROWS.
```

The following apply:

1. Lines 001600 to 002200 are the various data-names such as counters and switches.

a. Line 002000 shows the value of +500 which we will add to the salary column of all rows, except those with NULLS for values.

2. Lines 002300 to 002500 generate the communication area data block show in Figure10.1

3. Lines 002600 to 002800 include the DCLGEN output shown in Figure.12.1

4. Lines 003000 to 003600 implement the DECLARE statement that specifies the cursor name (EMPSCR), the columns selected, the row selection criteria, and the clause”FOR UPDATE OF “ column-name”.

a. Line 003200 specirfies the columns (s) we want read into the program. They do not have to be same columns that are updated 9FOR UPDATE OF clause).

b. Line 003500 specifies the column (s) to be updated.

5. Line 003700 sets the proper value to W005-EMPNO-UPDATED. This makes sure that we initially fetch the first row in the table (from line 003400)

6. Line 004300 zeros out the update counter. We do a commit after updating three rows.

7. Lines 004700 to 005000 process each groups for the three possible conditions.

8. Lines 005100 to 005900 display the range of bypassed rows if DB2 detected a deadlock.

9. Lines 006000 to 006500 do the close and commit if we have reached the end of rows or have updated three rows already.

10. Lines 006700 to 007000 show that we update only those rows where salary is not NULLS.

11. Lines 007200 to 008000 do the update.

    a. line 007300 saves the first employee number in the group. It is used to display the range of by passed rows if DB2 detects a deadlock from lines 005100 to 005900).

    b. Line 007400 adds the value of +500 to the salary column,

    c. Line 008000 adds 1 to the count of updated rows.

12. Lines 008100 to 008800 fetch the row.

    a. line 008400 is important since it places the employee number of the current row into W005-EMPNO-UPDATED. When we reopen the cursor the next times to process the next group because of line 003400, we start at the first employee number after the last one fetches.

1.7.1.3. Updated table: COMMIT with multiple open close cursor.

Figure 13.8 is the group process on the first of the cursor. We process up to the fourth row(employee number 00015) since we do not include the row with NULLS value for salary.

	Employee Number (EMPNO)	Name (EMPNAME)	Department Number (EMPDEPT)	Salary (EMPSALARYY
1.	00005	BAKER,C.	003	0031200
2.	00008	RANDOLPH,R.	005	0030000
3.	00010	RICHARDS,M.	002	NULLS
4.	00015	DAVIS,L.	006	0030500
5.	00150	ELLIOT,T.	005	0032000
6.	00170	RABINOVITIA,M.	003	NULLS
7.	00190	LEE,R.	005	0031000

Figure 13.8 updated table: COMMIT with multiple open/close cursor.

	Employee Number (EMPNO)		Salary (EMPSALARY)
1.	00005	0030700	
2.	00008	0029500	
3.	00010	Nulls Table	Temporary cursor
4.	00015	0030000	
5.	00150	0031500	
6.	00170	Nulls	
7.	00190	0030500	

Figure 13.9 Temporary cursor table first open

1.7.1.5. The second open : Temporary cursor table

Figure 13.10 is the group processed on the second open of the cursor. We process until the end of the table.

	Employee Number (EMPNO)		Salary (EMPSALARY)
1.	00005	0030700	
2.	00008	0029500	
3.	00010	Nulls	
4.	00015	0030000	
5.	00150	0031500	
6.	00170	Nulls table	Temporary cursor
7.	00190	0030500	

Figure 13.10 Temporary cursor table. Second open.

2. DELETE USING THE CURRENT CURSOR

Just like that of update, in most types of delete, the programmer first selects the row before it is deleted. This is because the row is printed out, verified, and so on before the delete is made. This type of processing uses the same cursor processing technique we saw in Section 1 of this chapter. The procedure is :

1. Code the usual DECLARE cursor statement in either the Procedure Division (preferred for better documentation), or working-storage section. Thus :

```
EXEC SQL
        DECLARE cursor-name CURSOR FOR
                SELECT EMPNAME, EMPSALARY
                FROM XLIM.EMPTABLE
                WHERE EMPDEPT = :SEARCH-DEPT
END-EXEC.
```

2. In the procedure Division, code the DELETE statement after the corresponding FETCH statement. The format of the DELETE statement is :

```
EXEC SQL
                DELETE FROM table-name
WHERE CURRENT QF cursor-name
END-EXEC.
```

3. As before, also code the OPEN, FETCH, and CLOSE cursor-name statements.

2.1. DELETE USING Cursor and Temporary Cursor Table

We explained earlier in this chapter (Section 1.3.2) that when using the "WHERE CURRENT OF current-name" clause, only the conceptual temporary cursor table may be used. This means that the program must use the original table. If DB2 creates a physical temporary cursor table, then the "WHERE CURRENT OF cursor-name" clause cannot be used, since the cursor would then be pointing to the former and not the original table. In this case, to delete rows in the original table, we have to use instead the "WHERE column-name = :data-name" clause. However, we cannot then guarantee that we are deleting only one row (as we could if we delete using the current cursor), unless the value of column-name is unique.

2.2. Program Logic : Delete the Current Row

The program logic is Figure 13.11

The follow apply :

1. The DECLARE cursor-name statement identifies the cursor, the columns read into the program, and the search criteria.

    a) The column-names in the SELECT clause (EMPNAME, EMPSALARY) are those of the table.

    b) The WHERE clause makes sure that any OPEN cursor-name statement gets the row where the value of EMPNO is greater than the value of the data name W005-EMPNO-DELETED (originally set to ‘00000’). By putting the value of the last row fetched into W00S-EMPNO-DELETED, we can process the whole table correctly even with reputed open and close of the cursor.

2. The PROCESS-ALL-ROWS and PROCESS-GROUP-OF-ROWS paragraphs count the number of rows actually deleted, then commit and close the cursor when we have reached the number we want (in this example, ten rows).

```
WORKING-STORAGE SECTION
        ……
        EXEC SQL
                INCLUDE SQLCA
        END-EXEC
        EXEC SQL
                INCLUDE  EMPTABDL
        END-EXEC
PROCEDURE DIVISION
        ……..
        EXEC SQL
                DECLARE cursor-name  CURSOR FOR
                                SELECT EMPNO,   EMPSALARY
                                FROM  XLIM.EMPTABLE
                WHERE  EMPNO >  :W005-EMPNO-DELETED
        END-EXEC.
        MOVE  ‘00000’  TO  W005-EMPNO-DELETED
        PERFORM  C020-PROCESS-ALL-ROWS UNTIL   W005-NO-MORE-ROWS
        GOBACK
C020-PROCESS-ALL-ROWS
        MOVE ZEROS TO W005-NUMBER-ROWS-DELETED
        EXEC  SQL
                OPEN  cursor-name
        END-EXEC.
        PERFORM  C080-FETCH-ONE-ROW
        PERFORM C040-PROCESS-GROUP-OF-ROWS  UNTIL  W005-NO-MORE-ROWS

                                OR  W005-NUMBER-ROWS-DELETED  =  10
        EXEC  SQL
                COMMIT
        END-EXEC
        EXEC  SQL
                CLOSE  cursor-name
        END EXEC
```

```
C040-PROCESS-GROUP-OF-ROWS
          EXEC  SQL
                    DELETE XLIM – EMPTABLE
                              WHERE CURRENT OF cursor-name
          END-EXEC.
          ADD   1  TO  W005-NUMBER-ROWS-DELETED
C080-FETCH-ONE-ROW
          EXEC  SQL
                    FETCH  cursor-name
                              INTO : W005-EMPNO-DELETED
                                        :EMPSALARY:005-SALARY-INDV
          END EXEC.
          IF SQLCODE  =  100
                              MOVE ‘Y’  TO  W005-END-OF-ROWS.
```

Figure 13.11 Program logic : delete the current row.

3. The FETCH statement has the INTO clause, which brings the data for the selected columns into the Cobol data-names specified. The use of W005-EMPNO-DELETED brings the employee number of the latest row fetched into this dataname.

2.3. Program Example : Delete the Current Row

We use the same data in Figure 13.8 as the input data for the delete. 2.3.1. Program listing : Delete the Current Row. We now delete all rows where the salary value is NULLs. Figure 13.12 is the program listing.

2.3.1. Program Listing : Delete the Current Row

We now delete all rows where the salary value is. NULLS. Figure 13.12 is the program listing.

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID. PROG1307.
000300 **********************************************************************
000400 *                                                                    *
000500 *  1.  THIS PROGRAM UPDATES ROWS VIA A CURSOR                        *
000600 *                                                                    *
000700 *  2.  UPDATE IS VIA MULTIPLE OPEN / CLOSE OF CURSOR.                *
000800 *                                                                    *
000900    **********************************************************************
001000    ENVIRONMENT DIVISION
001100    CONFIGURATION SECTION
001200    INPUT-OUTPUT SECTION
001300    DATA DIVISION
001400    FILE SECTION
001500    WORKING-STORAGE SECTION
001600    01             W005-SALARY-INDV                PIC  S9 (4)  COMP.
001700    01             W005-EMPNO-DELETED              PIC X (5)
001800    01             W005-EMPNO-LOW RANGE            PIC X (5)
```

```
001900  01      W005-NUMBER-ROWS-DELETED            PIC S9 (4)  COMP.
002000  01      W005-END-OF-ROWS                    PIC X VALUE ‘N’
002100  01      88          W-005-NO-MORE-ROWS                  VALUE ‘Y’
002200          EXEC SQL
002300              INCLUDE SQLCA
002400          END-EXEC.
002500          EXEC SQL
002600                  INCLUDE EMPTABDL
002700          END-EXEC.
002800          PROCEDURE DIVISION.
002900          EXEC SQL
003000              DECLARE EMPCSR  CURSOR FOR
003100                  SELECT EMPNO,  EMPSALARY
003200                      FROM  XLIM.EMPTABLE
003300                      WHERE EMPNO > :W005-EMPNO-DELETED
003400      .   END-EXEC
003500          MOVE  ‘00000’  TO W005-EMPNO-DELETED
003600          PERFORM  C020-PROCESS-ALL-ROWS
003700                  UNTIL  W005-NO-MORE-ROWS
003800          DISPLAY  ‘*** END OF JOB PROG 1312’  UPON SYSOUT.
003900          GOBACK
004000  C020-PROCESS-ALL-ROWS
004100          MOVE ZEROS TO  W005-NUMBER-ROWS DELETED
004200          EXEC SQL
004300          OPEN EMPCSR
004400          END-EXEC.
004500          PERFORM  C040-PROCESS-GROUP-OF-ROWS
004600              UNTIL W005-NO-MORE-ROWS
004700                  OR  SQLCODE  =  (-911  OR  -913)
004800                  OR W005-NUMBER-ROWS-DELETED = 10.
004900          IF  SQLCODE =  (-911  OR  -913)
005000              DISPLAY  ‘PROG 1312 --  EMPLOYEE  ‘  W005-EMPNO-LOW-RANGE
005100                              TO ‘  W005-EMPNO-DELETED
005200                  BYPASSED DUE TO DEADLOCK’  UPON SYSOUT
005300          IF  SQLCODE  =  -913
005400              EXEC  SQL
005500                  ROLLBACK
005600              END-EXEC
005700          ELSE NEXT SENTENCE
005800          ELSE EXEC SQL
005900                      CLOSE EMPCSR
006000          END-EXEC
006100          EXEC SQL
006200              COMMIT
006300          END-EXEC
006400  C040-PROCESS-GROUP-OF-ROWS
006500          PERFORM C080-FETCH-ONE-ROW.
006600          IF      SQLCODE             EQUAL TO ZERO
006700              AND     W005-SALARY-INDV LESS THAN ZERO
006800              PERFORM  C060-DELETE THIS-ROW
006900  C060-DELETE-THIS-ROW.
007000          IF W005-NUMBER-ROWS-DELETED EQUAL TO ZERO
007100              MOVE W005-EMPNO-DELETED TO W005-EMPNO-LOW-RANGE
007200          EXE SQL
007300              DELETE  XLIM. EMPTABLE
007400                  WHERE CURRENT OF EMPCSR
007500          END-EXEC.
007600          ADD     1       TO      W005-NUMBER-ROWS-DELETED
```

```
007700  C080-FETCH-ONE-ROW
007800                  EXEC SQL
007900                          FETCH EMPCSR
008000                                  INTO    :W005-EMPNO-DELETED
008100                                          :EMPSALARY:W005-SALARY-INDV
008200          END-EXEC.
008300          IF SQLCODE  =  100
008400                          MOVE  'Y'       TO      W005-END-OF-ROWS.
```

Figure 13.12. Program listing : delete the current row.

1. Lines 001600 t0 002100 are the various data-names such as counters and switches.

2. Lines 002200 to 002400 generate the communication area data block shown in Figure 10.1

3. Lines 002500 to 002700 include the DCLGEN output shown in Figure 12.1.

4. Lines 002900 to 003400 implement the DECLARE statement that specifies the cursor name (EMPCSR), the columns selected, and the row selection criteria.

    a) Line 003100 specifies the column(s) we want read into the program.

5. Line 003500 sets the proper value of W005-EMPNO-DELETED. This makes sure that we initially fetch the first row in the table (from line 003300).

6. Line 004100 zeros out the delete counter. We do a commit after deleting ten rows.

7. Lines 004500 to 004800 process each group for the three possible conditions.

8. Lines 004900 to 005700 display the range of bypassed rows if DB2 detected a deadlock.

9. Lines 005800 to 006300 do the close and commit if we have reached the end of rows or have deleted ten rows already.

10. Lines 006600 to 007600 show that we delete only those rows where salary is NULLs.

    a) Line 007100 saves the first employee number in the group. It is used to display the range of bypassed rows if DB2 detects a deadlock (from lines 004900 to 005700).

    b) Line 007600 to 008400 fetch the row.

11. Lines 007700 to 008400 fetch the row.

a) Line 008000 is important since it places the employee number of the current row into W005-EMPNO-DELTED. When we reopen the cursor the next time to process the next group, because of line 003300, we start at the first employee number after the last one fetched.

2.3.2. Additional JCL statements to be included. Assuming that we use

"EDITJCL" in entry 3 of Figure 11.5 in Chapter 11, DB2 will generate most of the JCL Statements for us. However, we still have to include additional JCL statements. In this example, this is the display output. Thus :

```
                                        / /   SYSOUT  DD     SYSOUT  =

*
```

2.3.3. Updated table : delete the current row.

Figure 13.13. is the updated table from the program in Figure 13.6.

	Employee Number (EMPNO)	Name (EMPNAME)	Department Number (EMPDEPT)	Salary (EMPSALARY)
1.	00005	BAKER, C.	003	0031200
2.	00008	RANDOLPH, R	005	0030000
3.	00015	DAVIS, L.	006	0030500
4.	00050	ELLIOT, T.	005	0032000
5.	00190	LEE, R.	005	0031000

Figure 13.13 Updated table : delete the current row.

Note that we have deleted the two rows in Figure 13.8, where the salary is NULLs.

14. UPDATE / DELETE / INSERT
Without Using Current Cursor

1. UPDATE / DELETE WITHOUT USING CURRENT CURSOR

We mentioned in the previous that most update and delete operations use a cursor. However, a program may still do a noncursor update or delete in the same manner we did it in SPUFI in Chapter 9. We remember that in this case, all rows that meet the criteria (based on the WHERE clause) are updated. This is different from an update using the current cursor, where only the row we are currently processing is updated.

2. NON CURSOR UPDATE OF ONE OR MORE ROWS

The noncursor update using SPUFI and Cobol program are basically identical. The statement is issued at the point in the program we want the update done. The format is :

```
EXEC  SQL
         UPDATE  table-name
         SET  column-name 1  =  expression1,
         Column-name2  =  expression 2 ……….)
END-EXEC
```

The SET operand specifies the column(s) and the corresponding value(s) for the update ; the WHERE clause specifies the criteria for selecting the rows to be updated and is very important because all rows will be updated if is missing.

2.1. Current XLIM.EMPTABLE:Noncursor Update

Figure 14.1. is the input to the noncursor delete program.

	Employee		Department	
	Number	Name	Number	Salary
	(EMPNO)	(EMPNAME)	(EMPDEPT)	(EMPSALARY)
1.	00005	BAKER, C.	003	0031200
2.	00008	RANDOLPH, R	005	0030000
3.	00015	DAVIS, L.	006	0030500
4.	00050	ELLIOT, T.	005	0032000
5.	00190	LEE, R.	005	0031000

Figure 14.1 Current XLIM.EMPTABLE:noncursor update.

2.2. Program Listing : Noncursor Update

For our program example, we read update cards containing both old and new department numbers. Rows which match the old department number are updated with the new department numbers. The program listing is Figure 14.2

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. PROG1402.
000300 ********************************************************************
000400 *                                                                  *
000500 *  1.  THIS PROGRAM DOES A NONCURSOR UPDATE                        *
000600 *                                                                  *
000700 ********************************************************************
000800   ENVIRONMENT  DIVISION
000900   CONFIGURATION SECTION
001000   INPUT-OUTPUT SECTION
001100   FILE-CONTROL
001200                 SELECT UPDATE-INPUT               ASSIGN TO CARDIN
001300   DATA DIVISION
001400   FILE SECTION
001500   FD            UPDATE-INPUT
001600                 UPDATE CONTAINS  0  RECORDS
001700                 LABEL RECORDS OMITTED
001800   01            UPDATE-RECORDS                    PIC X  (8)
001900   WORKING-STORAGE    SECTION
002000   01            W005-SALARY-INDV                  PIC S9 (4) COMP.
002100   01            W005-UPDATE-RECORD
002200                 05           W005-EMPDEPT         PIC X (3)
002300                 05           FILLER               PIC XX
002400                 05           W005-NEW-EMPDEPT          PIC X (3)
002500   01            W005-END-OF-UPDATE-RECORDS             PIC X VALUE 'N'
002600                 88           W005-NO-MORE-UPDATES           VALUE 'Y'
002700        EXEC SQL
002800                      INCLUDE  SQLCA
002900        END-EXEC.
003000        EXEC SQL
003100                      INCLUDE EMPTABDL
003200        END-EXEC.
003300   PROCEDURE DIVISION
003400        OPEN INPUT  UPDATE-INPUT
003500        PERFORM C040-READ-UPDATE-INPUT
003600        PERFORM C020-PROCESS-EACH-DEPT
003700                                UNTIL W005-NO-MORE  UPDATES
003800        CLOSE UPDATE-INPUT
003900        DISPLAY '*** END OF JOB PROG 1402'  UPON SYSOUT
004000        GOBACK
004100   C020-PROCESS-EACH-DEPT.
004200        EXEC SQL
004300                      UPDATE XLIM.EMPTABLE
004400                           SET EMPEPT      =      :W005-NEW-EMPDEPT
004500                           WHERE EMPDEPT  =       :W005-EMPDEPT
004600   END-EXEC.
004700        IF SQLCODE = 0
004800                      EXEC SQL
004900                           COMMIT
005000                      END-EXEC
```

```
005100       ELSE IF SQLCODE = 100
005200           DISPLAY ‘PROG1402’  --  DEPT ‘W005-EMPDEPT
005300               ‘HAS NO TABLE DATA’ UPON SYSOUT
005400       ELSE IF SQLCODE =  (-911  OR –913)
005500           DISPLAY  ‘PROG 1402  -- DEPT  ‘W005-EMMPDEPT
005600               ‘BYPASSED DUE TO DEADLOCK’ UPON SYSOUT
005700           IF SQLCODE =  -91.3
005800               EXEC SQL
005900                       ROLLBACK
006000               END-EXEC
006100       PERFORM C040-READ-UPDATE-INPUT
006200   C040-READ-UPDATE-INPUT
006300       READ UPDATE-INPUT INTO W005-UPDATE-RECORD
006400               AT END,  MOVE  ‘Y’  TO W005-END-OF-UPDATE-RECORDS
```

figure 14.2 Program listing : noncursor update

The following apply :

1. Lines 0011000 to 001800 describe the input file containing the old and new department numbers.

2. Lines 002000 to 002600 are the various data-names such as counters and switches.

3. Lines 002700 to 002900 generate the communication area data block shown in Figure 0.1.

4. Lines 003000 to 003200 include the DCLGEN output shown in Figure 12.1.

5. Lines 003400 to 004000 are the housekeeping routines, including the Mainloop routine.

6. Lines 004100 to 006000 process each department read from the in-line card file.

   a) Lines 004200 to 004600 update all rows with the same department number as that read from the card.

   b) Lines 004700 to 005000 commit the deletes.

   c) Lines 005100 to 005300 display an error message if the department does not have a row in the table.

   d) Lines 005400 to 006000 display an error messages if the department processing is interrupted by DB2 due to a deadlock condition. IF SQLCODE is-913, we do a rollback.

7. Lines 006200 to 006400 read the in-line card file for the department number of departments to be deleted.

2.3. Additional JCL Statement to be included

Assuming that we use "EDITJCL" in entry 3 of Figure 11.5 in Chapter 11, DB2 will generate most of the JCL statements for us. However, we still have to include those for batch files. In this example, these are for the display output and the card file. Thus :

```
//      SYSOUT  DD  SYSQUT  =  *
//      CARDIN  DD   *
003     045
005     046
/*
```

2.4. Output Table : Noncursor Update

Using Figure 14.1 as input, Figure 14.3 is the output of the program. Note that the original department numbers '003' and '005' have been changed to '045' and '046' respectively.

	Employee Number (EMPNO)	Name (EMPNAME)	Department Number (EMPDEPT)	Salary (EMPSALARY)
1.	00005	BAKER, C.	003	0031200
2.	00008	RANDOLPH, R.	005	0030000
3.	00015	DAVIS, L.	006	0030500
4.	00150	ELLIOT, T.	005	0032000
5.	00190	LEE, R.	005	0031000

Figure 14.3 Output table : noncursor update

3. NONCURSOR DELETE

The style is similar for that of noncursor update. The format is :

```
EXEC SQL
        DELETE FROM table-name
                        (WHERE clause)
END-EXEC
```

The WHERE clause specifies the criteria for selecting the row(s) to be deleted. Just like that of UPDATE, this clause is very important because all rows will be deleted if it is missing.

3.1. Program Listing : Noncursor Delete

For our program example, we read delete cards containing department numbers. Rows which match the department number are deleted. The input is Figure 14.3. The program listing is Figure 14.4.

The following apply :

1. Lines 001100 to 001800 describe the input file containing the department number of departments to be deleted.

2. Lines 002000 to 002400 are the various data-names such as counters and switches.

3. Lines 002500 to 002700 generate the communication area data block shown in Figure 10.1

4. Lines 002800 to 003000 include the DCLGEN output shown in Figure 12.1.

5. Lines 003200 to 003800 are the housekeeping routines, including the Mainloop routine.

6. Lines 003900 to 005800 process each department read in the in-line card file.

   a) Lines 004000 to 004300 delete all rows with the same department number as that read from the card.
   b) Lines 004400 to 004700 commit the deletes.
   c) Lines 004800 to 005700 display an error message if the department processing is interrupted by DB2 due to deadlock condition. If SQLCODE is 913, we do a roll back.

7. Lines 005900 to 006100 read in-line card file for the department number of departments to be deleted.

```
000100  IDENTIFICATION DIVISION
000200  PROGRAM-ID. PROG 1404
000300  *****************************************************************
000400  *                                                      *
000500  *     1.     THIS PROGRAM DOES A NON CURSOR DELETE     *
000600  *                                                      *
000700  *****************************************************************
000800  ENVIRONMENT  DIVISION
000900  CONFIGURATION  SECTION
001000  INPUT-OUTPUT  SECTION
001100  FILE-CONTROL
001200              SELECT DELETE – INPUT ASSIGN TO CARDIN
001300  DATA DIVISION
001400  FILE SECTION
001500  FD          DELETE-INPUT
001600              BLOCK CONTAINS 0  RECORDS
001700              LABEL RECORDS OMITTED
001800  01          DELETE-RECORD                 PIC X (80)
001900  WORKING-STORAGE SECTION
002000  01          W005-SALARY-INDV              PIC S9 (4)  COMP.
002100  01          W005-DELETE-RECORD.
002200              05     W005-EMPDATE           PIC X (3)
002300  01          W005-END-OR-DELETE-RECORDS    PIC X VALUE ‘N’
002400              88           W005-NO-MORE-DELETES     VALUE ‘Y’
```

```
        EXEC  SQL
                INCLUDE  SQLCA
        END-EXEC
        EXEC SQL
                INCLUDE  EMPTABDL
        END-EXEC
PROCEDURE DIVISION
        OPEN INPUT DELETE-INPUT
        PERFORM  C040-READ-DELETE-INPUT


    PERFORM  C020-PROCESS-EACH-DEPT
            UNTIL   W005-NO-MORE-DELETES
    CLOSE DELETE-INPUT
    DISPLAY  *** END OF JOB PROG1404'   UPON SYSOUT
    GOBACK
C020-PROCESS-EACH-DEPT.
    EXEC  SQL
            DELETE XLIM.EMPTABLE
                WHERE  EMPDEPT  =  W005-NO-MORE DELETES
    END-EXEC
    IF  SQLCODE  =  0
            EXEC  SQL
                    COMMIT
    END-EXEC
    ELSE IF SQLCODE  =  100
                DISPLAY 'PROG1404 – DEPT'  W005-EMPDPT
                ' HAS NO TABLE DATA '  UPON SYSOUT
    ELSE IF  SQLCODE  =  (-911  OR  -913)
                DISPLAY 'PROG1404 – DE (T 'W005-EMPDEPT
                    'BYPASSED DUE TO DEADLOCK' UPON SYSOUT
                IF  SQLCODE =  913
                    EXEC  SQL
                            ROLLBACK
                END-EXEC
        PERFORM C040-READ-DELETE-INPUT
C040-READ-DELETE-INPUT
    READ DELETE-INPUT INTO  W005-DELETE-RECORD
            AT END,  MOVE  'Y'  TO  W005-END-OF-DELETE-RECORDS.
```

Figure 14.4 Program listing : noncursor delete

3.2. Additional JCL Statements to be included

Assuming that we use "EDITJCL" in entry 3 of Figure 11.5 in Chapter 11, DB2 will generate most of the JCL statements for us. However, we still have to include those for batch files. In this example, these are for the display for the display output and the deleting card file. Thus :

```
/ /     SYSOUT  DD  SYS OUT  =  *
/ /     CARDIN  DD *
006
045
/ *
```

3.3. Output Table : Noncursor Delete

Using Figure 14.3 as the input, Figure 14.5 is the output of the program. Note that department numbers '006' and '045' from Figure 14.3 have been deleted.

	Employee		Department	
	Number	Name	Number	Salary
	(EMPNO)	(EMPNAME)	(EMPDEPT)	(EMPSALARY)
1.	00008	RANDOLPH, R.	046	0030000
2.	00150	ELLIOT, T.	046	0032000
3.	00190	LEE, R.	046	0031000

Figure 14.5 **O**utput table : noncursor delete.

4. INSERT

The style is similar to those in Chapter 9. We can either insert a single row (as in Chapter, Section 1.5) or do a mass insert (as in Chapter 9, Section 1.7). The format for a single row insert is :

```
EXEC SQL
        INSERT INTO table-name
                (column-name1, column name2……..)
                VALUES string1, string 2………,)
```

The following apply :

1. The column list (column names) is optional. If it is not specified, the sequence of string values corresponds to the sequence of the columns of the tables as created. If it is specified, columns and strings have a one-to-one correspondence and do not have to match the original sequence of the columns in the table.

2. If a column is missing in the column list and it was defined as "NOT NULL WITH DEFAULT" when the table was created, DB2 will generate the default value (zeroes for numeric columns, spaces for nonnumeric columns).

3. If a column is missing and it was defined as "NOT NULL" there is an error and the INSERT operation is not done.

4.1. Insert : No Cursor Required

Note that while there is a cursor version of update and delete, all insert operations are done without a cursor. Just like in SPUFI, we either insert row or do a mass insert of multiple rows, where the input rows would then generally come from another table.

4.2. Program Listing : Insert One Row

For our program example, we insert rows based on data cards. The input is Figure 14.5. The program listing is Figure 14.6.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. PROG1406.
*******************************************************************
*                                                                 *
*  1.  THIS PROGRAM DOES A SINGLE ROW INSERT                      *
*                                                                 *
* *****************************************************************
  ENVIRONMENT  DIVISION
  CONFIGURATION SECTION
  INPUT-OUTPUT SECTION
  FILE-CONTROL
         SELECT INSERT-INPUT  ASSIGN TO CARDIN
  DATA DIVISION
  FILE SELECTION
  FD              INSERT-INPUT
                  BLOCK CONTAINS  0  RECORDS
                  LABEL  RECORDS OMITTED
  01              INSERT-RECORD                          PIC X (80)
  WORKING-STORAGE SECTION.
  01              W005-SALARY-INDV                       PIC S9 (4)  COMP.
  01              W005-NUMBER-ROWS-INSERTED              PIC S (8)  COMP.
  01              W005-INSERT-RECORD
                  05          W005-EMPNO                 PIC X (5)
                  05          FILLER                     PIC XX
                  05          W005-EMPNAME               PIC X  (30)
                  05          FILLER                     PIC XX
                  05          W005-EMPDEPT               PIC X (3)
                  05          FILLER                     PIC XX
                  05          W005-NO-MORE-INSERTS       PIC S9 (7)
  01              W005-END-OF-INSERT-RECORDS             PIC X VALUE ‘N’
                  88          W005-NO-MORE-INSERTS       VALUE ‘Y’
  01              W005-FIRST-RECORD-POINTER              PIC S9 (8)  COMP.
  01              W005-LAST-RECORD-POINTER               PIC S9 (8)  COMP.
  01              W005-RECORD-COUNT                 PIC S9 (8) COMP VALUE ZERO.
                  EXEC SQL
                              INCLUDE SQLCA
                  END-EXEC.
                  EXEX SQL
                              INCLUDE EMPTABDL
                  END-EXEC
  PROCEDURE DIVISION
                  OPEN INPUT INSERT-INPUT
                  PERFORM C060-READ-INSERT-INPUT
                  PERFORM C020-PROCESS-ALL-ROWS
                              UNTIL W005-NO-MORE-INSERTS
                  CLOSE INSERT-INPUT
                  DISPLAY *** END OF JOB PROG 1406’  UPON SYSOUT.
```

```
004800                  GOBACK.
```

The following apply :

1. Lines 001100 to 001800 are the input file containing the data to be inserted.

2. Lines 002000 to 003400 are the various data-names such as counters and switches.

3. Lines 003500 to 003700 generate the communication area data block shown in Figure 10.1.

4. Lines 003800 to 004000 include the DCLGEN output shown in Figure 12.1.

5. Lines 004200 to 004800 are the housekeeping routines.

```
004900  C020-PROCESS-ALL-ROWS
005000                  MOVE ZEROES             TO      W005-NUMBER-ROWS-INSERTED
005100                  MOVE W005-RECORD-COUNT          TO      W005-FIRST-RECORD-POINTER
005200                  PERFORM C040-PROCESS-ROW-GROUP
005300                          UNTIL W005-NO-MORE-INSERTS
005400                                  OR W005-NUMBER-ROWS-INSERTED = 10
005500                                          OR  SQLCODE = (-911 OR –913)
005600                  IF SQLCODE =  0
005700                          EXEC SQL
005800                                          COMMIT
005900                          END-EXEC
006000                  ELSE IF SQLCODE  =  (-911  OR  -913)
006100                          COMPUTE W005-LAST-RECORD-POINTER
006200                                          =  W005-RECORD-COUNT - 1
006300                  DISPLAY ‘PROG1406 --  INPUT ‘W005-FIRST-RECORD-POINTER
006400                          ‘ TO ’  W005-LAST-RECORD-POINTER
006500                                  BYPASSED DUE TO DEADLOCK   UPON SYSOUT
006600                          IF SQLCODE =  -913
006700                                          EXEC SQL
006800                                                          ROLLBACK
006900                                          END-EXEC.
007000  C040-PROCESS-ROW-GROUP
007100                  MOVE W005-EMPNO                 TO              EMPNO.
007200                  MOVE W005-EMPNAME               TO              EMPNAME
007300                  MOVE W005-EMPDEPT               TO      EMPDEPT
007400                  IF  W005-EMPSALARY NUMERIC
007500                  MOVE  0         TO      W005-EMPSALARY TO EMPSALARY
007600                  MOVE  W005-EMPSALARY            TO              EMPSALARY
007700                  ELSE  MOVE – 1          TO      W005-SALARY-INDV.
007800                  EXEC  SQL
007900                          INSERT INTO XLIM.EMPTABLE
008000                                                  (EMPNO,
008100                                                  EMPNAME,
008200                                                  EMPDEPT,
008300                                                  EMPSALARY)
008400                          VALUES  (:EMPNO,
008500                                  :EMPNAME,
008600                                  : EMPDEPT,
008700                                  : EMPSALARY : W005-SALARY-INDV)
008800          END-EXEC.
```

```
009000           IF SQLCODE = 0
009100                ADD  1  TO  PERFORM  C060-READ-INSERT-INPUT
009200      C060-READ-INSERT-INPUT
009300           READ INSERT-INPUT INTO  W005-INSERT-RECORD
009400                AT END,  MOVE  ‘Y’  TO  W005-END-OF-INSERT-RECORDS
009500           IF  NOT W005-NO-MORE-INSERTS
009600                ADD  1  TO  W005-RECORD-COUNT.
009700
```

Figure 14.6 (continued)

6. Lines 004900 to 009100 process each input card.

   a) Line 005100 marks the first record number in the group.

   b) Lines 005200 to 005500 process the input under three conditions.

   c) Lines 005600 to 005900 commit the insertions.

   d) Lines 006000 to 006900 display an error message if the processing is interrupted by DB2 due to a deadlock condition. If SQLCODE is –913, we do a rollback.

   e) Lines 007100 to 007300 lay out columns which will never have null values.

   f) Lines 007400 to 007700 lay out the salary column. We indicate via an indicator variable whether or not we want DB2 to use nulls (if the input is not numeric).

   g) Lines 007800 to 008800 do the insert of one row.

   h) Lines 008900 to 009000 add 1 to the count on a successful insertion.

7. Lines 009200 to 009600 read the in-line card file for the input data.

4.3. Additional JCL Statements to be included

Assuming that we use “EDITJCL” in entry 3 of Figure 11.5 in Chapter 11, DB2 will generate most of the JCL for us. However, we still have to include those for batch files. In this example, these are for the display output and the input card file. Thus :

```
/ /     SYSQUT   DD  SVSOUT  = *
/ /     CARDIN   DD  *
00600    LIM,  P.                 005     0034000
00763    WARNER,  P.              006     0031500
```

4.4. Output Table : Insert One Row.

Using Figure 14.5 as the input, Figure 14.7 is the output of the program in Figure 14.6.

4.5 Mass insert from a Table.

	Employee Number (EMPNO)	Name (EMPNAME)	Department Number (EMPDEPT)	Salary (EMPSALARY)
1.	00008	RANDOLPH, R.	046	0030000
2.	00150	ELLIOT, T.	046	0032000
3.	00190	LEE, R.	046	0031000
4.	00600	LIM, P.	005	0034000
5.	00763	WARNER, P.	006	0031500

Note that we have added the data for employee numbers ‘00600’ and ‘00763’.

4.4. Mass Insert from a Table

The programmer may do a mass insert of rows from current tables. The style is similar to that of Chapter 9, Sections 1.7 and 1.8.

References

IBM Manuals

DB2 Application Programming and SQL guide
DB2 Messages and Codes
DB2 SQL Reference
DB2 Command and Utility Reference

(CSM) Craig S. Mullins, "DB2 Developer's Guide", Sams Publishing 1992.

(W&K) Gabrielle Wiorkowski & David Kull, "DB2 Design & Development Guide"
ADDISON-WESLEY Publishing 1992.

(D&W) C.J. Date & Colin J. White, "A Guide to DB2", ADDISON-WESLEY Publishing.