Unit II
Functions of a DBMS

Module 3
Basic DBMS Functions

Most DBMS functions involve either adding data to the database, updating existing data in the database, or reading useful information off the database. Even the simplest database management systems require all three of these functions in one form or another.

In this module, we discuss building these basic database functions at the physical level, and how other functions, like cataloguing and indexing, can be added to an existing DBMS.

Objectives

At the end of this module, you should be able to:

1. Identify basic functions needed by a DBMS; and
2. Specify how function might be implemented for a particular sample DBMS.

Data Storage and Updating

First of, we'll consider the problem of getting your data into the database. As mentioned in your previous modules, a database is customarily broken down into sets of records, with each record containing several fields. As you may have noticed, this setup is directly analogous to the entities and attributes of the conceptual level-each entity is usually stored in one record and the attributes of that entity are given separate fields for that record.

For example, a programmer told to store a database entity representing a bank customer, with attributes name, phone no. and address, might use a record with fields name, phone number, and address, each of the appropriate data type.

A list of such customers might be stored in an array of records, such that the database would look something like this:

1	Buenaventura, Mari	555-1898	54 Lopez Ave, Batong Malake, Los Baños
2	Lorenzo, Achilles	555-1812	252 Mt. Halcon, Batong Malake, Los Baños
3	Luna, Geronimo	555-1812	Blk. 40, Jubileeville, Los Baños
4	Realuyo, Dennis	555-1896	725 Bangkal St., San Antonio, Los Baños

Here, records are stored one after the other in memory, and new records are added to the end of the array. One disadvantage of this approach is the fixed size of an array, which imposes a limit on the number of records one can store in the database. This can be solved by using a linked list, instead of an array, with the resulting increase in program complexity.

Unfortunately, we cannot rely on mere records stored in memory for any database meant for actual use, due to the following limitations:

- Current memory technology is volatile. This means once you turn off the power to the computer, all data in memory is lost.

- Memory size cannot hold a database large enough for most applications. Databases comprising more than a few thousand records are already too large to fit in most computer systems' memory spaces.

For this reason, we have to store records on some non-volatile media, like magnetic disks or tape. To prevent loss of large amounts of data, typically only one or two records are kept in memory while editing or adding new records, and the rest are kept on disk. Updated records are written to disk as soon as possible.

Updating records on the on-disk database typically consists of three functions: adding new records, modifying existing records, and deleting or purging records. We'll take a look at each in turn.

Adding Records

A database programmed in a non-DBMS specific programming language would require a function that:

- Accepts data to be stored in the new record from the user. This is accomplished typically by providing one or more "template" screens containing the fields of the record, and having the user fill these fields in.

The program should check for correct input-i.e., numerical fields should have numbers, data fields should make sense,etc. One way of checking for correct input is to have the program accept all keyboard input as strings, and convert from string to the appropriate data type.

An error in the conversion routine will mean that the user has typed an invalid value.

- Checks for dependencies in other, related databases. Some fields may be shared between different relational tables, for instance. The function must find these other tables and modify the appropriate tables.

- Updates all tables affected by the new record, typically saving to disk.

In most database programming languages, these functions are performed automatically by an update function. For example, in dBase, the APPEND command produces an interface which executes the three routines given above.

Modifying Records

Modifying an existing record first requires that the user identify the particular record to be updated. The user typically provides a primary key to search on. The record is loaded, and a template screen similar to the one used for adding records is shown, allowing the user to change a particular field. An example of this is the dBase EDIT command.

An alternate method can be provided by showing a list of records to the user, and allowing him/her to pick the record to be edited as in the dBase BROWSE command. A table of records is shown, and the user can choose one record by selecting it with a cursor.

As with adding new records, care must be taken that relational or functional dependencies are also updated by the modify routine.

Deleting Records

Here, we have to identify the particular record or records to delete, and search for them, either by using a primary key, or allowing the user to pick records off a list. The records to be deleted are typically only "marked" as deleted until a "purge" command (the dBase PACK command is an example) is issued. This allows a user to "undelete" a record if one was marked to be deleted by mistake.

The deleted records may also have functional or relational dependencies in other tables. Two methods can be used in such an eventuality:

- The affected records may also be deleted, if they are dependent on the original record (weak entities). For example, deleting the record for a bank customer may also delete the records for the accounts that customer holds, since an account must have an owner.

- The affected records may have "null" or "nonexistent" values entered in the affected fields. Using the same example, the account might have "nonexistent" entered as its owner, in which case there must also be a function that finds all such "floating" accounts and either assign them a new owner, or delete them later.

The method used for a particular database depends on the nature of the relationship or functional dependency.

Searching for a Particular Record

This function is used in both modifying and deleting functions given above. Here we have to identify and return the location of a particular record. This typically requires the user to provide a key to search. The provided search key is then compared to the primary key field of the records in the database until a match is found.

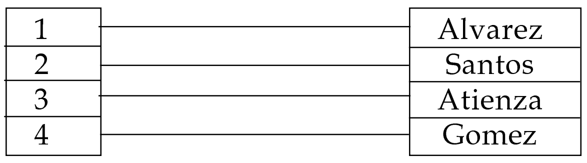
For small databases, a linear search (check records from the start of the file until one is found) is usually simple enough to implement and does not take too long to execute. The file may also be sorted or indexed to facilitate searching.

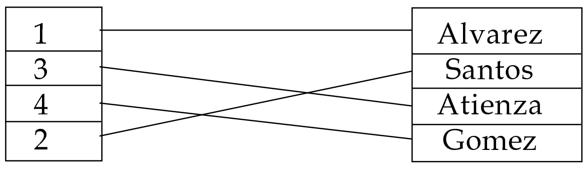
DBMS languages typically implement proprietary search algorithms in an effort to quickly search large databases - these search functions may require that the database is indexed or catalogued in some way.

Indexing and Cataloguing

To facilitate quicker access to the database (for easier sorting or searching) DBMS languages provide indexing functions. An index file is maintained, separate to the database file, containing keys to the contents of the file. A typical index file is small enough to fit in memory (allowing quicker sorting or searching) but contains enough information to identify every record in the database.

For example:





In the top diagram, the index file contains the record location of the records in the file. The data file is unsorted. In the second, the index file has been sorted by alphabetical order of the last name field, but the database file is still unsorted. We can therefore produce a printout of the records of the database in sorted order simply by following the order in the index file, without having to swap around the actual records in the file.

In actual use, the index file would typically also contain the primary key field, which can also be used for quicker searching (since the entire index file is in memory, and does not have to be loaded record by record from disk).

We can also maintain multiple indexes, one for each key. For example, in one index, we can sort the database by last name, in another, by address or by account number. We can therefore access the database as if it were sorted in several different ways at once.

Module 4
Transaction Processing and Concurrency Control

So far, we've dealt with **single-user** databases, or database systems that assume only one user is accessing our data at any given time. In most real-world database applications, this is not the case–a single computer system may service several users at once, ranging from a few clerks in a small office LAN, to hundreds of customer transactions per second on the largest mainframes.

In this module we discuss functions essential to maintaining database coherence in such situations. In **transaction processing**, we will show how multiple updates can be executed safely, and how these updates can be properly implemented, even in the event of unforeseen accidents, outages or mistakes on the part of one or more of the clients. **Concurrency control** deals with general-purpose techniques that are used in any multi-user system, not just in multi-user databases.

Objectives

Athe the end of this module, you should be able to:

1. Understand the differences between single-user and multi-user database;
2. Understand how transaction processing works in a multi-user database; and
3. Understand how concurrency control is used in multi-user systems.

Multi-user Database Systems

Several issues appear in a multi-user system that are not found in systems designed for single users. One such issue is **data consistency**, where we have to make sure that any data on the system is kept intact, even when two or more users are inputting data into the system at the same time.

Multi-user operating systems, like UNIX and Windows NT, have had to develop features to insure data coherency on their file systems. This is usually accomplished by **file locking**, where a file on disk can be "locked" or "reserved" for writing by an application. During the period of time that the file is "locked", no other application is allowed to write data to the file, to insure that two users aren't overwriting each other's updates at the same time.

Multi-user database management systems, on the other hand, often require that multiple users be allowed access to the same files at the same time; in order to read, edit and modify records. A simple solution would be to leave data coherency up to the operating system, and merely disallow access if a file is locked. A user (let's call him User A) trying to edit records at the same time as User B would find that he has to wait for User B to finish editing, before he is allowed to open the file.

This is a suboptimal solution (but one which is in use in many smaller database systems), as a user could conceivably have to wait a long time before getting access to a particular record, one which may not have been modified at all by the other user. Also, this is not workable for databases with more than a few simultaneous users, as our hapless User A may have to wait a long time before all the other users are finished with that particular file.

Transaction Processing

The concept of transaction processing involves separating individual database operations into transactions, where each transaction consists of a single update to the database system. For example the operation:

T_0: Transfer P50 from Account A to Account B

can be considered one transaction. Each transaction can consist of several database functions (reading or writing data into the database, or modifying data previously read from the database). Transaction T0 may be defined as consisting of the following functions:

T_0: Transfer P50 from Account A to Account B

```
seek(A)              #seek to position of Account A
read(balA);          #read current balance of account
balA := balA - 50;   #subtract 50 from this value;
seek(A);
write(balA);         #write new balance back into Account A
seek(B);             #seek to position of Account B
read(balB);          #read current balance of account
balB := balB+50;     #add 50 to this value;
seek(B);
write(balB);         #write new balance back into Account B
```

Now assume we have a different transaction T_1, (possibly being entered at the same time by some other clerk) such that:

T_1: Transfer P100 from Account B to Account A

```
seek(B)              #seek to position of Account A
read(balB);          #read current balance of account
balB := balB - 100;  #subtract 100 from this value;
seek(B);
write(balB);         #write new balance back into Account A
seek(A);             #seek to position of Account B
read(balA);          #read current balance of account
balA := balA+100;    #add 100 to this value;
seek(A);
write(balA);         #write new balance back into Account B
```

We now have two transactions, one moving P50 from A to B, and the other moving P100 from B to A. At the end, Account A should be P50 larger, and Account B should be P50 less.

If the two transactions are done one after the other, we have no problem with the end result.

Transaction T_0	**Transaction T_0**
seek(A)	
read(ba1A)	
ba1A = ba1a-50	
seek(a)	
write(ba1A)	
seek(B)	
read(ba1B)	
ba1B = ba1B + 50	
seek()	
write(ba1B)	
	seek(B)
	read(ba1B)
	ba1B = ba1B - 100
	seek(B)
	write(ba1B)
	seek(A)
	read(ba1A)
	ba1A = ba1A + 100
	seek(A)
	write(ba1A)

The end result would be the same if T_1 was executed before T_0.

Unfortunately, the same is not true in the rare case where the transactions are run almost simultaneously. Of course, in a single-processor system, the computer can only execute one command at a time, but even if we assume that the computer can only do one database function at any given time, we might end up with something like this:

(For this example, let us assume A contains P5000 and B contains P4000 at the start.)

Transaction T_0	**Transaction T_1**	**A**	**B**
seek(A)		5000	4000
read(ba1A)		5000	4000
ba1A= ba1A - 50			
	seek(B)		
	read(ba1B)	5000	4000
	ba1B = ba1B - 100		
	seek(B)		
	write(ba1B)	5000	3900
	seek(A)		
	read(ba1A)	5000	3900
	ba1A = ba1A + 100		
	seek(A)		
	write(ba1A)	5100	3900
seek(A)			
write(ba1A)		4950*	3900
seek(B)			
read(ba1B)		4950	3900
ba1B = ba1B + 50			
seek(B)			
write(ba1B)		4950	3950

Here, in T_0, P50 is deducted from memory variable balA. But before the updated quantity can be written back to disk, T_1 has already updated account A (to 5100). Unfortunately, since T_0 does not know that T_1 has changed the disk data, it overwrites the disk data with its own result, (marked by the *) losing whatever change T_1 made to account A. Instead of A = 5050 and B = 3950 (as what should have been if T_0 and T_1 were not run simultaneously), we have the erroneous value A = 4950. P100 is now missing from account A.

You can imagine how many errors would now appear in a system that handles dozens or even hundreds of simultaneous transactions.

Critical Sections

The gray area between "read(A)" and "write(A)" in the table is what we call a critical section. This is a portion of a transaction which we should not interrupt - in the table, if we had allowed T_0 to write the new value to disk before allowing T_1 to continue, we would have gotten the correct answer.

Therefore, we need to identify and preserve critical sections in our code. We need to make sure that inside this critical section, only that transaction is allowed to manipulate that particular data item. We can use file/ record locking in order to implement critical sections properly.

For our example, we can define a function lock(A) such that when the function is called, the file containing (A) is locked from access by any transaction other than the one that called the lock() function. If a transaction attempts to lock() some file already locked by another transaction, it will have to wait for that transaction to unlock() the file. The unlock(A) function, on the other hand, frees up the file for use by other transactions, and allows any waiting lock() by other transactions to proceed.

Our new transactions, using the lock() and unlock() functions, should now look like:

Transaction T_0	**Transaction T_1**	**A**	**B**
seek(A)		5000	4000
lock(A)			
read(ba1A)		5000	4000
ba1A= ba1A - 50			
	seek(B)		
	lock(B)		
	read(ba1B)	5000	4000
	ba1B = ba1B - 100		
	seek(B)		
	write(ba1B)	5000	3900
	unlock(B)		
	seek(A)		
	lock(A)*		
seek(A)		5000	3900
write(ba1A)		4950	3900
unlock(A)	*		
	read(ba1A)	4950	3900
	ba1A = ba1A + 100		
	seek(A)		
	write(ba1A)	5050	3900
	unlock(A)		
seek(B)			
lock(B)			
read(ba1B)		5050	3900
ba1B = ba1B + 50			
seek(B)			
write(ba1B)		5050	3950
unlock(B)			

The asterisks (*) mark the point where T_1 attempts to get a lock on A, but it is already locked by T_0. Therefore, T_1 has to pause until T_0 unlocks A, at the second asterisk. This prevents T_1 from changing the value of A within T_0's critical section (the gray area).

Summary

Multi-user multiprocessing database systems are systems that allow more than one user to access the database at any given time. While this makes a database system more useful, it also introduces problems with regards to ensuring data integrity and coherence. Simultaneous updates, in particular, have a chance of corrupting the data if executed in the incorrect order.

We can identify critical sections in the code, portions where another transaction must not be allowed to update a particular on-disk record or data item. We can use locking in order to ensure that no other simultaneous transaction can edit a particular variable, where the other transaction is paused or stopped if it attempts such an action (by trying to initiate its own lock).

Module 5
Database Integrity and Crash Recovery

In this module, you will be studying what the integrity of a database is and how to recover from database crashes. Integrity of a database refers to the accuracy or correctness of the data in the database. Accuracy or correctness of data is a critical aspect in database management specifically databases that are multi-user. However, systems today are yet weak on integrity that most integrity checking is still done by writing user-procedural code. Although most relational database management systems have begun providing a reasonable level of declarative support for primary and foreign keys, most of them are still quite weak in the area of database-specific integrity. In this module, you will be seeing a definition of a hypothetical integrity language that most systems might be able to use. A set of examples will also be presented for you to better understand the topic.

Objectives

At the end of this module, you should be able to:

1. Understand database integrity and how to manage it using a hypothetical language; and
2. Understand crash recovery and how to implement it in a multi-user database system.

Bound up with a multi-user (concurrent) database are the problems of recovery and the notion of transaction processing. This module will explain to you what a transaction is and what the term transaction processing (or transaction *management*) means, and introduce the functions COMMIT and ROLLBACK. Then the problems of recovery and concurrency that the transaction concept is intended to solve will be discussed. The examples and discussions will be based specifically on an SQL system. However, the ideas that will be presented are very general and will apply to numerous other systems, relational or otherwise, with comparatively little change.

Database Integrity

As mentioned in the introduction, it will be a good idea to specify the constraints in database integrity in some more declarative fashion and let the system do the checking instead of the user writing long procedural codes. Almost all database designers would agree that the specification for integrity constraints could account for as much as 90% of a typical database definition. In this scenario, a system that will support these specifications would ease out for application programmers the burden of writing codes for integrity checking. At the same time, it will enable those programmers to become significantly more productive as they can channel their efforts to other programming and data management tasks.

What is an integrity constraint?

An integrity constraint can be regarded as a condition that all correct states of the database are required to satisfy. A simple example of such a condition might be

```
FORALL STUDENTS (STUDENTS.GRADE>0)
```

(for all STUDENTS, the GRADE must be positive). If the user attempts to execute an operation that would violate the constraint, the system must then either reject the operation or possibly (in more complicated situations) perform some compensating action on some other part of the database to ensure that the overall result is still a correct state. Thus, any language for specifying integrity constraints should include, not only the ability to specify arbitrary conditions, but also facilities for specifying such compensating actions when appropriate.

A Hypothetical database integrity language

Your hypothetical integrity language consists of two statements, CREATE INTEGRITY RULE and DROP INTEGRITY RULE. A detailed definition of these statements will not be given here (because in reality, further definitional work is still needed before these statements can be considered for implementation) but instead some examples will be used to show the features these statements will provide. You will be using three tables STUDENT, TEACHER, and ST, where ST is a table that relates to primary tables STUDENT and TEACHER.

Example 1. Attribute values (in this example, the attribute STATUS) must be positive:

```
CREATE INTEGRITY RULE R1
        ON      INSERT STUDENT.AGE
                UPDATE STUDENT.AGE
        CHECK           FORALL STUDENT (STUDENT.AGE > 0)
        ELSE    REJECT;
```

In general, the CREATE INTEGRITY RULE statement must specify:

- The name of the rule (R1 in the example);
- One or more checking times (ON INSERT STUDENT.AGE, UPDATE STUDENT.AGE in the example) telling the system when to perform the check in the CHECK clause;
- A constraint which must be satisfied by all legal states of the database (FORALL STUDENT (STUDENT.AGE > 0) in the example);
- A violation response, indicating what to do if the check fails (REJECT in the example, meaning that the offending INSERT or UPDATE is to be rejected with an appropriate return code).

The example above illustrates that integrity rules must include all four components (name, checking time, constraint, and violation response). However, some obvious simplifications to these rules will be applied in the subsequent examples. These simplifications are:

- The checking time(s) will usually be obvious. Thus, it is better to assume that the system is capable of determining the checking time(s) for itself if there is no explicit specification of any such time(s).
- The constraint in the CHECK clause will almost always begin with a universal quantifier. Thus, a variable without a quantifier is automatically quantified by the FORALL quantifier.
- Assume also that if the ELSE clause is omitted, the default violation response *REJECT the update operation* (*with a suitable return code*) is implied.

Applying the above simplifications, the example above can be reduced to just

```
CREATE INTEGRITY RULE R1
      CHECK STUDENT.AGE > 0;
```

When the CREATE INTEGRITY RULE statement is executed, the system first checks to see whether the current state of the database satisfies the specified constraint. If it does not, the new rule is rejected; otherwise it is accepted and enforced from that time on. Enforcement in the example requires the system to monitor all operations that would insert a value into, or update a value in, column AGE of table STUDENT.

Example 2. A constraint that is complex. In this example, assume that the table STUDENT includes an additional set of attributes MONTH, DAY, and YEAR, each of type character of widths 2, 2 and 4, respectively, representing a date (also assuming that the system does not support a data type for dates):

```
CREATE INTEGRITY RULE R2
        CHECK        IS_INTEGER(STUDENT.YEAR)
        AND   IS_INTEGER(STUDENT.MONTH)
        AND   IS_INTEGER(STUDENT.DAY)
        AND   NUM(STUDENT.YEAR) BETWEEN 1900 AND 2051
        AND   NUM(STUDENT.MONTH) BETWEEN 1 AND 12
        AND   NUM(STUDENT.DAY) > 0
        AND   IF NUM(STUDENT.MONTH) IN (1,3,5,7,8,10,12)
              THEN NUM(STUDENT.DAY) < 32
        AND   IF NUM(STUDENT.MONTH)=12
              THEN NUM(STUDENT.DAY)<30
        AND   IF NUM(STUDENT.MONTH)=2 AND
                      NUM(STUDENT.YEAR)<>0 AND
                      MOD(NUM(STUDENT.YEAR), 4)<>0
              THEN NUM(STUDENT.DAY)<29;
```

In the example above (example 2), the existence of two built-in functions: IS_INTEGER that tests a character string to see if it represents a legal decimal integer value; and NUM that converts a character string that represents a decimal value to internal numeric form; are assumed.

Example 3. Attribute values must never decrease.

```
CREATE INTEGRITY RULE R3
        BEFORE UPDATE OF STUDENT.ID FROM NEW_ID:
                CHECK NEW_ID > S.ID;
```

Rule R3 applies to the transition between two states rather than to database states. Note that this rule requires an explicit checking time (BEFORE clause) to be specified to indicate to the system when the checking is to be done. In this example, if a student is to be given a new ID, the new ID must be greater than the old.

Example 4. Attribute ID is a foreign key in table ST, matching the primary key of table STUDENT.

```
 CREATE INTEGRITY RULE R4A
     AFTER INSERT OF ST, UPDATE OF ST.ID:
     CHECK EXISTS STUDENT (STUDENT.ID=ST.ID);
CREATE INTEGRITY RULE R4B
     BEFORE DELETE OF STUDENT.ID, UPDATE OF STUDENT.ID:
     CHECK NOT EXISTS ST (ST.ID=STUDENT.ID);
```

Rule R4A says that it is illegal to insert an ST record or change the ID value in an ST record if no corresponding STUDENT record exists after the operation. Rule R4B says that it is illegal to delete a STUDENT record or change the ID value of a STUDENT record if any corresponding ST record currently exists.

Example 5. An example of an integrity rule that implements the cascading of a DELETE action from the STUDENT table to ST table, and an UPDATE action from ST.

```
CREATE INTEGRITY RULE R5A
     BEFORE DELETE OF STUDENT:
     CHECK NOT EXISTS ST (ST.ID=STUDENT.ID)
     ELSE DELETE ST WHERE ST.ID=STUDENT.ID;
CREATE INTEGRITY RULE R5B
     BEFORE UPDATE OF STUDENT.ID FROM NEW_STUDENT.ID:
     CHECK NOT EXISTS ST (ST.ID=STUDENT.ID)
     ELSE UPDATE ST.ID FROM NEW_STUDENT.ID
              WHERE ST.ID=STUDENT.ID;
```

Crash Recovery

Crash recovery is synonymous to transaction recovery. Let us begin our discussions by introducing the fundamental notion of a *transaction.* A transaction is a *logical unit of work.* Consider the following example. Suppose that table TEACHER, includes the fields EMP_ID and TOT_STUDENTS while the table ST includes the fields ID, and EMP_ID. Here the TOT_STUDENTS represents the total number of students a certain teacher handles; in other words, the value of TOT_STUDENTS for any given teacher is equal to the count of all ST.ID values, taken over all ST records for that teacher. Now consider the following sequence of operations, the intent of which is to add a student S5 to teacher T1 to the database:

```
            ON SQLERROR GO TO UNDO;
            INSERT INTOST (ID, EMP_ID)
                  VALUES ('S5','T1');
            UPDATE TEACHER
                  SET    TOT_STUDENTS = TOT_STUDENTS + 1
                  WHERE EMP_ID = 'T1';
            COMMIT;
            GO TO FINISH;
UNDO:
            ROLLBACK;
FINISH:     RETURN;
```

The INSERT adds the new student (with student ID equal to 'S5') to the ST table, the UPDATE updates the TOT_STUDENTS field for teacher T1 appropriately.

The point of this example is that what is presumably intended to be a single atomic operation-"Add a new student"-in fact involves *two* updates to the database. What is more, the database is not even consistent between those two updates; it temporarily violates the requirement that the value of TOT_STUDENT for teacher T1 is supposed to be equal to the count of all ST.ID values for teacher T1. Thus, a logical unit of work (i.e., a transaction) is not necessarily just a single database operation; rather, it is a *sequence* of several such operations, in general, that transforms a consistent state of the database into another consistent state, without really being consistent at all intermediate points.

Now it is clear that what must not be allowed to happen in the example is for one of the two updates to be executed and the other not (because that would leave the database in an inconsistent state). What is needed ideally, of course, is a guarantee that both updates will be executed. Unfortunately, it is impossible to provide any such guarantee–there is always a chance that things will go wrong. For example, a system crash might occur between the two updates, or an arithmetic overflow might occur on the second of them. But a system that supports *transactions processing* provides the guarantee. Specifically, it guarantees that if the transaction executes some updates and then a failure occurs (for whatever reason) before the transaction reaches its normal termination, *then those updates will be undone.* Thus, the transaction either executes in its entirety or is totally canceled (as if it never executed at all). In this way a sequence of operations that is fundamentally not atomic can be made to look as if it really were atomic from afar.

The system component that provides this atomicity (or semblance of atomicity) is known as the *transaction manager*, and the COMMIT and ROLLBACK operations are the key to the way it works:

- The COMMIT operation signals *successful* end-of-transaction: It tells the transaction manager that a logical unit of work has been successfully completed, the database is (or should be) in a consistent state again, and all of the updates made by that unit of work can now be "committed" or made permanent.
- The ROLLBACK operation, by contrast, signals *unsuccessful* end-of-transaction: It tells the transaction manager that something has gone wrong, the database might be in an inconsistent state, and all of the updates made by the logical unit of work so far must be "rolled back" or undone.

In the example, therefore, a COMMIT statement is issued if the two updates were run successfully, which will commit the changes in the database and make them permanent. If anything goes wrong, however i.e., if either update statement raises the SQLERROR condition then a ROLLBACK statement is issued instead, to undo any changes made so far.

Note 1: This module has shown the COMMIT and ROLLBACK operation explicitly, for the sake of the example. However, some systems will automatically issue a COMMIT for any program that reaches normal termination, and will automatically issue a ROLLBACK for any program that does not (regardless of the reason; in particular, if a program terminates abnormally because of a *system* failure, a ROLLBACK will be issued on its behalf when the system is restarted). In the example, therefore, the explicit COMMIT could have been omitted, but not the explicit ROLLBACK.

Note 2: A realistic application program should not only update the database (or attempt to) but should also send some kind of message back to the end user indicating what has happened. In the example, the message "Student added" could have been sent if the COMMIT is reached, or the message "Error - student not added" otherwise. Message-handling, in turn, has additional implications for recovery.

At this time, you might be wondering how it is possible to undo an update. The answer is that the system maintains a *log* or *journal* on disk, on which details of all update operations are recorded. Thus, if it becomes necessary to undo some particular update, the system can use the corresponding log entry to restore the updated object to its previous value.

One further point: In a relational system, data manipulation statements are set-level and typically operate on multiple records at a time. What then if something goes wrong in the middle of such a statement? For example, it is possible that a multiple-record UPDATE could update some of its target records and then fail before updating the rest? The answer is no, it is not. SQL statements are required to be individually atomic in so far as their effect on the database is concerned. If an error occurs in the middle of such a statement, then the database will remain totally unchanged.

Synchronization points

Executing either a COMMIT or a ROLLBACK operation establishes what is called a *synchronization point.* A synchronization point represents the boundary between two consecutive transactions; it thus corresponds to the end of a logical unit of work, and hence to a point at which the database is in a state of consistency. The only operations that establish a synchronization point are COMMIT, ROLLBACK, and program initiation. (Remember, however, that COMMIT and ROLLBACK may often be implicit.) When a synchronization point is established:

- All updates made by the program since the previous synchronization point are committed (COMMIT) or undone (ROLLBACK).
- All open cursors are closed and all database positioning is lost (this is true in most systems, but not to all).
- All record locks are released.

Note carefully that COMMIT and ROLLBACK terminate the transaction, not the program. In general, a single program execution will consist of a sequence of several transactions, running one after another, with each COMMIT or ROLLBACK operation terminating one transaction and starting the next. However, it is true that very often one program execution will correspond to just one transaction; and if it does, then it will frequently be possible to code that program without any explicit COMMIT or ROLLBACK statements at all.

In conclusion, you must now be able to see that transactions are not only the unit of work but also the unit of *recovery*. For if a transaction successfully COMMITs, then the system must guarantee that its updates will be permanently established in the database, even if the system crashes the very next moment. It is quite possible, for instance, that the system might crash after the COMMIT has been honored but before the updates have been physically written to the database (they could still be waiting in a main storage buffer and so be lost at the time of the crash for example). Even if that happens, the system's restart procedure will still install those updates in the database; it is able to discover the values to be written by examining the relevant entries in the log. (It follows then that the log must be physically written before COMMIT processing can complete. This important rule is known as the *Write-Ahead Log Protocol.)* Thus, the restart procedure will recover any units of work (transactions) that completed successfully but did not manage to get their updates physically written prior to the crash; hence, as stated earlier, transactions are indeed the unit of recovery.

System and media recovery

The system must be prepared to recover, not only from purely local failures such as the occurrence of an overflow condition within a single transaction, but also from "global" failures such as a power failure on the CPU. A local failure, by definition, affects only the transaction in which the failure has actually occurred; such failures have already been adequately discussed above. A global failure, by contrast, affects several of the transactions in progress at the time of the failure, and hence has significant system-wide implications. In this section, what is involved in recovering from a global failure will be discussed.

Such failure fall into two broad categories:

- *System failures* (e.g., power failure) affect all transactions currently in progress but do not physically damage the database. A system failure is also known as a *soft crash.*
- *Media failures* (e.g., head crash on the disk), which do cause damage to the database, or to some portion of it, and affect at least those transactions currently using that portion. A media failure is sometimes called a *hard crash.*

Let us now proceed to discuss each case in somewhat more detail.

System failure

The critical point about system failure is that *the contents of main storage are lost* (strictly speaking, it is the contents of database's buffer that are lost). At the time of the failure, the precise state of any transaction that was in progress is no longer known. When the system restarts, this transaction can never be completed and so must be *undone* (or rolled back). Furthermore, it is necessary at restart time to *redo* certain transactions that were successfully completed prior to the crash but did not manage to get their updates transferred from the database buffers to the physical database.

Now, how does the system know at restart time which transactions to undo and which to redo? The answer is as follows. At certain prescribed intervals, the system automatically *takes a checkpoint.* Taking a checkpoint involves (a) physically writing the contents of the database buffers to the physical database, and (b) physically writing a special *checkpoint record* to the physical log. The checkpoint record lists all transactions that were in progress

- A system failure has occurred at time t_{fail}.
- The most recent checkpoint prior to time t_{fail}. was taken at time t_{check}.
- Transaction T_1 was completed prior to time t_{check}.
- Transaction T_2 started prior to time tcheck and completed after time t_{check} and before time t_{fail}.
- Transaction T_3 also started prior to time tcheck but did not complete by time $t_{fail.}$.
- Transaction T_4 started after time t_{check} and completed before time t_{fail}.
- Finally, transaction T_5 also started after time t_{check} but did not complete by time t_{fail}.

When the system is restarted, transactions T_3 and T_5 must be undone, and transactions of types T_2 and T_4 must be redone. Note, however, that transactions of type T_1 do not enter into the restart process at all, because their updates were physically written to the database at time *tcheck* as part of the checkpoint process.

At restart time, therefore, the system goes through the following procedure in order to identify all transactions T_2 through T_5:

1. Start with two lists of transactions, the UNDO list and the REDO list. Set the UNDO list equal to the list of all transactions given in the checkpoint record; set the REDO list to empty.
2. Search forward through the log, starting from the checkpoint record.
3. If a "begin transaction" log entry is found for transaction T, add T to the UNDO list.
4. If a "commit" log entry is found for transaction T, move T from the UNDO list to the REDO list.
5. When the end of the log is reached, the UNDO list identifies transactions T_3 and T_5 while the REDO list identifies transactions T_2 and T_4.

The system now works backward through the log, undoing the transactions in the UNDO list. Then it works forward again, redoing the transactions in the REDO list. Finally, when all such recovery activity is complete, then (and only then) the system is ready to accept new work.

Media failure

A media failure is a failure (such as a disk head crash or a disk controller failure) in which some portion of the database has been physically destroyed. Recovery from such a failure involves reloading (or *restoring)* the database from a backup copy (or *dump)*, and then using the log, both active and archive portions, to redo all transactions that completed since that backup copy was taken. There is no need to undo transactions that were still in progress at the time of the failure, since by definition all updates of such transactions have been "undone" (destroyed) anyway.

Note: The recovery implies the need for a *dump/restore* (or *unload/reload)* utility. The dump portion of that utility is used to make backup copies of the database on demand (backups can be kept on an archive disk or tape). The restore portion of the utility is then used to recreate the database after a media failure from a specified backup copy.

Summary

In this module, database integrity and crash recovery were discussed. Hypothetical language was defined, albeit not in detail, and examples of the use of this language were presented for better understanding of the integrity of databases. Simplifications of the hypothetical language were also discussed and used in the set of examples presented.

Transaction recovery is defined to be synonymous to crash recovery. The module also defines in detail what transactions are. A single transaction is seen to be a series of atomic operations in the database that may result in inconsistency in the database if not completed in its entirety. Synchronization of operations can be implemented via the COMMIT statement while recovery from failure during the middle of a transaction can be done via the ROLLBACK statement. The database system keeps logs of the transactions during predetermined time intervals called checkpoints. These logs enable the system to either REDO or UNDO operations affected by a system crash.