Fourteen percent (14%) of the DB2 UDB V8.1 for Linux, UNIX, and Windows Database Administration certification exam (Exam 701) and seventeen percent (17%) of the DB2 UDB V8.1 for Linux, UNIX, and Windows Database Administration certification upgrade exam (Exam 706) exam is designed to evaluate your knowledge of transactions and transaction logging, and to test your ability to backup and restore a database using the various methods of backup and recovery available. The questions that make up this portion of the exam are intended to evaluate the following:

➤ Your knowledge of the various transaction logging features available.
➤ Your knowledge of the types of database recovery available (crash, version, and roll-forward) and your ability to demonstrate when and how each are used.
➤ Your ability to create and use database-level and tablespace-level backup images.
➤ Your ability to create and use full, incremental, and delta backup images.
➤ Your ability to return a damaged or corrupt database to the state it was in at any given point in time.
➤ Your knowledge of how and when invalid indexes are recreated.
➤ Your ability to suspend and resume database I/O and your ability to initialize a split mirror copy of a database.

This chapter is designed to introduce you to the backup and recovery tools that are available with DB2 Universal Database, and to show you how to both backup a database on a regular basis and restore a database if it becomes damaged or corrupted.
Although basic syntax is presented for most of the DB2 UDB commands covered in this chapter, the actual syntax supported may be much more complex. To view the complete syntax for a specific command or to obtain more information about a particular command, refer to the IBM DB2 Universal Database, Version 8 Command Reference product documentation.

Terms you will learn:

- Transaction
- Unit of work
- COMMIT
- ROLLBACK
- Transaction logging
- Log buffer
- Circular logging
- Archival logging
- Infinite logging
- Log mirroring
- Crash recovery
- Version recovery
- Roll-forward recovery
- Transaction failure
- Recoverable database
- Non-recoverable database
- Online backup and recovery
- Offline backup and recovery
- RESTART
- Soft checkpoint
- BACKUP
- RESTORE
- ROLLFORWARD
- Recovery History file
- LIST HISTORY
- PRUNE HISTORY
- Redirected restore
- SET TABLESPACE CONTAINERS
- Invalid index
- Split Mirror
- Mirroring
Techniques you will master:

- Recognizing the types of transaction logging available and understanding when each is to be used.
- Understanding how crash recovery, version recovery, and roll-forward recovery operations are initiated.
- Understanding how database-level and tablespace-level backup images are made.
- Understanding how full, incremental, and delta backup images are created and used.
- Understanding how invalid indexes are recreated.
- Understanding how split mirroring is used to backup a database.
- Understanding how I/O is suspended and resumed so mirrors can be split, as well as knowing how split mirrors are initialized.

Transactions

A transaction (also known as a unit of work) is a sequence of one or more SQL operations grouped together as a single unit, usually within an application process. Such a unit is called “atomic” because, like atoms (before fission and fusion were discovered), it is indivisible—either all of its work is carried out or none of its work is carried out. A given transaction can perform any number of SQL operations—from a single operation to many hundreds or even thousands, depending on what is considered a “single step” within your business logic. (It is important to note that the longer a transaction is, the more database concurrency decreases and the more resource locks are acquired; this is usually considered the sign of a poorly written application.)

The initiation and termination of a single transaction defines points of data consistency within a database; either the effects of all operations performed within a transaction are applied to the database and made permanent (committed), or the effects of all operations performed are backed out (rolled back) and the database is returned to the state it was in before the transaction was initiated.

In most cases, transactions are initiated the first time an executable SQL statement is executed after a connection to a database has been made or immediately after a pre-existing transaction has been terminated. Once initiated, transactions can be implicitly terminated, using a feature known as “automatic
commit” (in which case, each executable SQL statement is treated as a single transaction, and any changes made by that statement are applied to the database if the statement executes successfully or discarded if the statement fails) or they can be explicitly terminated by executing the COMMIT or the ROLLBACK SQL statement. The basic syntax for these two statements is:

\[ \text{COMMIT } \langle \text{WORK} \rangle \]

and

\[ \text{ROLLBACK } \langle \text{WORK} \rangle \]

When the COMMIT statement is used to terminate a transaction, all changes made to the database since the transaction began are made permanent. On the other hand, when the ROLLBACK statement is used, all changes made are backed out and the database is returned to the state it was in just before the transaction began. Figure 7–1 shows the effects of a transaction that was ter-

![Figure 7–1 Terminating a transaction with the COMMIT SQL statement.](image-url)
minated with a **COMMIT** statement; Figure 7–2 shows the effects of a transaction that was terminated with a **ROLLBACK** statement.

It is important to remember that commit and rollback operations only have an effect on changes that have been made within the transaction they terminate. So in order to evaluate the effects of a series of transactions, you must be able to identify where each transaction begins, as well as when and how each transaction is terminated. Figure 7–3 shows how the effects of a series of transactions can be evaluated.

Changes made by a transaction that have not been committed are usually inaccessible to other users and applications (unless another user or application is using the Uncommitted Read isolation level), and can be backed out with a rollback operation. However, once changes made by a transaction have been committed, they become accessible to all other users and/or applications and can only be removed by executing new SQL statements (within a
So what happens if a problem occurs (for example, a power failure occurs or an application abends) before a transaction’s changes can be committed? In order to answer that question, we must first look at how data changes are made and at how transaction activity is logged.

**Figure 7-3** Evaluating the effects of a series of transactions.
Transaction Logging

So just what is transaction logging and how does it work? Transaction logging is simply a process that is used to keep track of changes made to a database (by transactions), as they are made. Each time an update or a delete operation is performed, the page containing the record to be updated/deleted is retrieved from storage and copied to the appropriate buffer pool, where it is then modified by the update/delete operation (if a new record is created by an insert operation, that record is created directly in the appropriate buffer pool). Once the record has been modified (or inserted), a record reflecting the modification/insertion is written to the log buffer, which is simply another designated storage area in memory. (The actual amount of memory that is reserved for the log buffer is controlled by the logbufsize database configuration parameter.) If an insert operation is performed, a record containing the new row is written to the log buffer; if a delete operation is performed, a record containing the row’s original values is written to the log buffer; and if an update operation is performed, a record containing the row’s original values, combined with the row’s new values, is written to the log buffer. These kinds of records, along with records that indicate whether the transactions that were responsible for making changes were committed or rolled back, make up the majority of the records stored in the log buffer.

Whenever buffer pool I/O page cleaners are activated, the log buffer becomes full, or a transaction is terminated (by being committed or rolled back), all records stored in the log buffer are immediately written to one or more log files stored on disk. (This is done to minimize the number of log records that might get lost in the event a system failure occurs.) This process is referred to as write-ahead logging and it ensures that log records are always flushed to log files before data changes are recorded in the database (i.e., copied to the appropriate tablespace containers for permanent storage). Eventually, all changes made in the buffer pool are recorded in the database, but only after the corresponding log records have been externalized to one or more log files. The modified data pages themselves remain in memory, where they can be quickly accessed if necessary; eventually they will be overwritten. The transaction logging process is illustrated in Figure 7–4.

Because multiple transactions may be working with a database at any given point in time, a single log file may contain log records that belong to several different transactions. Therefore, to keep track of which log records belong to which transactions, every log record is assigned a special “transaction identifier” that ties it to the transaction that created it. By using transaction IDs, log records associated with a particular transaction can be written to one or more log files at any time, without impacting data consistency. Eventually,
the execution of the COMMIT or ROLLBACK statement that terminates the transaction will be logged as well.

Since log records are externalized frequently and since changes made by a particular transaction are only externalized to the database after all log records associated with the transaction have been recorded in one or more log files, the ability to return a database to a consistent state after a failure occurs is guaranteed. When the database is restarted, log records are analyzed and each record that has a corresponding COMMIT record is reapplied to the database; every record that does not have a corresponding COMMIT record is either ignored or backed out (which is why “before” and “after” information is recorded for all update operations).

### Logging Strategies

When a database is first created, three log files, known as primary log files, are allocated as part of the creation process. On Linux and UNIX platforms, these log files are 1,000 4K (kilobyte) pages in size; on Windows platforms, these log files are 250 4K pages in size. However, the number of primary log files used, along with the amount of data each is capable of holding, is con-
trolled by the \textit{logprimary} and \textit{logfilsiz} parameters in the database's configuration file. The way in which all primary log files created are used is determined by the logging strategy chosen for the database. Two very different strategies, known as \textit{circular logging} and \textit{archival logging}, are available.

**Circular Logging**

When circular logging is used, records stored in the log buffer are written to primary log files in a circular sequence. Log records are written to the current “active” log file and when that log file becomes full, it is marked as being “unavailable”. At that point, DB2 makes the next log file in the sequence the active log file, and begins writing log records to it. And when that log file becomes full, the process is repeated. In the meantime, as transactions are terminated and their effects are externalized to the database, their corresponding log records are released because they are no longer needed. When all records stored in an individual log file are released, that file is marked as being “reusable” and the next time it becomes the active log file, its contents are overwritten with new log records.

Although primary log files are not marked reusable in any particular order (they are marked reusable when they are no longer needed), they must be written to in sequence. So what happens when the logging cycle gets back to a primary log file that is still marked “unavailable”? When this occurs, the DB2 Database Manager will allocate what is known as a \textit{secondary} log file and begin writing records to it. As soon as this secondary log file becomes full, the DB2 Database Manager will poll the primary log file again and if its status is still “unavailable”, another secondary log file is allocated and filled. This process will continue until either the desired primary log file becomes “reusable” or the number of secondary log files created matches the number of secondary log files allowed. If the former occurs, the DB2 Database Manager will begin writing log records to the appropriate primary log file and logging will pick up where it left off in the logging sequence. In the meantime, the records stored in the secondary log files are eventually released, and when all connections to the database have been terminated and a new connection is established, all secondary log files are destroyed. On the other hand, if the latter happens, all database activity will stop and the following message will be generated:

\texttt{SQL0964C} The transaction log for the database is full.

By default, up to two secondary log files will be created, if necessary, and their size will be the same as that of each primary log file used. However, the total number of secondary log files allowed is controlled by the \textit{logsecond} parameter in the database configuration file. Circular logging is illustrated in Figure 7–5.
Chapter 7

By default, when a new database is first created, circular logging is the logging strategy used.

Archival Logging

Like circular logging, when archival logging (also known as log retention logging) is used, log records stored in the log buffer are written to the primary log files that have been pre-allocated. However, unlike with circular logging, these log files are never reused. Instead, when all records stored in an individual log file are released, that file is marked as being “archived” rather than as being “reusable” and the only time it is used again is if it is...

Figure 7–5  Circular logging.
needed to support a roll-forward recovery operation. Each time a primary log file becomes full, another primary log file is allocated so that the desired number of primary log files (as specified by the `logprimary` database configuration parameter) are always available for use. This process continues as long as there is disk space available.

By default, all log records associated with a single transaction must fit within the active log space available (which is determined by the maximum number of primary and secondary log files allowed and the log file size used). Thus, in the event a long running transaction requires more log space than the primary log files provide, one or more secondary log files may be allocated and filled as well. If such a transaction causes the active log space to become full, all database activity will stop and the `SQL0964C` message we saw earlier will be produced.

Because any number of primary log files can exist when archival logging is used, they are classified according to their current state and location. Log files containing records associated with transactions that have not yet been committed or rolled back that reside in the active log directory (or device) are known as *active log files*; log files containing records associated with completed transactions (i.e., transactions that have been externalized to the database) that reside in the active log directory are known as *online archive log files*; and log files containing records that are associated with completed transactions that have been moved to a storage location other than the active log directory are known as *offline archive log files*. Offline archive files can be moved to their storage location either manually or automatically with a user exit program. Archival logging is illustrated in Figure 7–6.

*Figure 7–6*  Archival logging.
Infinite Active Logging. You would think that you could avoid running out of log space simply by configuring a database to use a large number of primary and/or secondary log files if needed. However, the maximum number of log files allowed (primary and secondary combined) is 256 and if the size of your log files is relatively small, you can still run out of log space quickly when transaction workloads become heavy or when transactions run for an inordinate amount of time. Furthermore, you want to avoid allocating a large number of secondary log files if possible because performance is affected each time a log file has to be allocated. Ideally, you want to allocate enough primary log files to handle most situations and you want to use just enough secondary log files to handle peaks in transaction workloads.

If you are concerned about running out of log space and you want to avoid allocating a large number of secondary log files, you can configure a database to perform what is known as infinite active logging or infinite logging. Infinite active logging allows an active transaction to span all primary logs and one or more archive logs, effectively allowing a transaction to use an infinite number of log files. To enable infinite active logging, you simply set the database configuration parameters `userexit` and `logsecond` to YES and -1, respectively. It is important to note that when the `userexit` database configuration parameter is set to YES, a user-supplied userexit program will be invoked each time a log file is closed and this program can move unneeded log files to another location for permanent storage (thus the risk of running out of log storage space on the server is eliminated).

When the `logsecond` database configuration parameter is set to -1, the `logprimary` and `logfilsiz` configuration parameters are still used to specify how many primary log files DB2 should keep in the active log path as well as with how big each file should be. If DB2 needs to read log data from a log file, but the file is not in the active log path, DB2 will invoke the userexit program provided to retrieve the log file from the archive and copy it to the active log location so that other reads of log data from the same file will be fast. DB2 manages the retrieval, copying, and removal of these log files as required.

Although infinite active logging can be used to support environments with large jobs that require more log space than you would normally allocate to the primary logs, it does have its tradeoffs. Specifically, rollback operations (both at the savepoint level and at the transaction level) could be very slow due to the need to retrieve log files from the archive storage location. Likewise, crash recovery could be very slow for the same reason.

Log mirroring. With DB2 UDB Version 8.1, you have the ability to configure a database such that the DB2 Database Manager will create and update active log files in two different locations (which is sometimes referred to as dual logging). By storing active log files in one location and mirroring them...
in another, separate location, database activity can continue if a disk failure or human error causes log files in one location to be destroyed. (Mirroring log files may also aid in database recovery.) To enable log file mirroring, you simply assign the fully qualified name of the mirror log location to the `mirrorlog-path` database configuration parameter. It is important to note that if log mirroring is used, the primary log file location used must be a directory and not a raw device. And ideally, the mirror log file storage location used should be on a physical disk that is separate from the disk used to store primary log files and that does not have a large amount of I/O.

### Database Recovery Concepts

Over time, a database can encounter any number of problems, including power interruptions, storage media failure, and application abends. All of these can result in database failure and each failure scenario requires a different recovery action.

The concept of backing up a database is the same as that of backing up any other set of data files: you make a copy of the data and store it on a different medium where it can be accessed in the event the original becomes damaged or destroyed. The simplest way to backup a database is to shut it down to ensure that no further transactions are processed, and then back it up using the BACKUP utility provided with DB2 UDB. Once the backup image has been made, you can rebuild the database if for some reason it becomes damaged or corrupted.

The process of rebuilding a database is known as *recovery* and with DB2 UDB, three types of recovery are available. They are:

- Crash recovery
- Version recovery
- Roll-forward recovery

### Crash Recovery

When an event or condition occurs that causes a database and/or the DB2 Database Manager to end abnormally, one or more transaction failures may result. Conditions that can cause transaction failure include:

- A power failure at the workstation where the DB2 Database Manager is running.
- A serious operating system error.
➤ A hardware failure such as memory corruption, disk failure, CPU failure, or network failure.
➤ An application failure.

When a transaction failure takes place, all work done by partially completed transactions that was still in memory is lost. And because some of that work may not have been externalized to the database, the database is left in an inconsistent state (and therefore is unusable). Crash recovery is the process that returns such a database to a consistent and usable state. Crash recovery is performed by using information stored in the transaction log files to roll back all incomplete transactions found and complete any committed transactions that were still in memory (but had not yet been externalized to the database) when the transaction failure occurred. Once a database is returned to a consistent and usable state, it has attained what is known as a "point of consistency."

Version Recovery

*Version recovery* is the process that returns a database to the state it was in at the time a backup image was made. Version recovery is performed by replacing the current version of a database with a previous version, using an image that was created with a backup operation; the entire database is rebuilt using a backup image created earlier. Unfortunately, when a version recovery is performed, all changes that have been made to the database since the backup image was created will be lost. Version recovery can be used to restore an entire database or it can be used to restore individual tablespaces—provided individual tablespace backup images exist.

Roll-Forward Recovery

*Roll-forward recovery* takes version recovery one step farther by replacing a database or individual tablespaces with a backup image and replaying information stored in transaction log files to return the database/tablespaces to the state they were in at an exact point in time. In order to perform a roll-forward recovery operation, you must have archival logging enabled, you must have a full backup image of the database available, and you must have access to all archived log files (or at least the ones you want to use for recovery) that have been created since the backup image was made. Like version recovery, roll-forward recovery can be applied to an entire database or to individual tablespaces.
Recoverable and Non-recoverable Databases

Although any DB2 UDB database can be recovered from transaction log files (crash recovery) or a backup image (version recovery), whether or not a database is considered recoverable is determined by the values of the database's logretain and userexit configuration parameters; a database is recoverable if the logretain parameter is set to RECOVERY and/or the userexit parameter is set to YES. (When both of these configuration parameters are set to NO, which is the default, circular logging is used and the database is considered non-recoverable.) A database is considered recoverable when crash recovery, version recovery, and roll-forward recovery is possible. A database is considered non-recoverable if roll-forward recovery is not supported. Other differences between recoverable and non-recoverable databases are shown in Table 7–1.

<table>
<thead>
<tr>
<th>Recoverable Database</th>
<th>Non-recoverable Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archive logging is used.</td>
<td>Circular logging is used.</td>
</tr>
<tr>
<td>The database can be backed up at any time, regardless of whether or not applications are connected to it and transactions are in progress.</td>
<td>The database can only be backed up when all connections to it have been terminated.</td>
</tr>
<tr>
<td>The entire database can be backed up or individual tablespaces can be backed up. Tablespaces can also be restored independently.</td>
<td>The entire database must be backed up; tablespace level backups are not supported.</td>
</tr>
<tr>
<td>A damaged database can be returned to the state it was in at any point in time; crash recovery, version recovery, and roll-forward recovery are supported.</td>
<td>A damaged database can only be returned to the state it was in at the time the last backup image was taken; only crash recovery and version recovery are supported.</td>
</tr>
</tbody>
</table>

The decision of whether a database should be recoverable or non-recoverable is based on several factors:

➤ If a database is used to support read-only operations, it can be non-recoverable; since no transactions will be logged, roll-forward recovery is not necessary.

➤ If relatively few changes will be made to a database and if all changes made can be easily recreated, it may be desirable to leave the database non-recoverable.
If a large amount of changes will be made to a database or if it would be difficult and time-consuming to recreate all changes made, a recoverable database should be used.

Online versus Offline Backup and Recovery

From a backup and recovery perspective, a database is either online or offline. When a database is offline, other applications and users cannot gain access to it; when a database is online, just the opposite is true. Backup and recovery operations can only be performed against a non-recoverable database after that database has been taken offline. Recoverable databases, on the other hand, can be backed up at any time, regardless of whether the database is offline or online. However, in order to restore a recoverable database (using version recovery and roll-forward recovery), the database must be taken offline. (It is important to note that a recoverable database only has to be taken offline if the entire database is to be restored; individual tablespaces can be restored while the database is online.)

When an online backup operation is performed, roll-forward recovery ensures that all changes made while the backup image is being made are captured and can be recreated with a roll-forward recovery operation. Furthermore, online backup operations can be performed against individual tablespaces as well as entire databases. And, unlike when full database version recovery operations are performed, tablespace version recovery operations and tablespace roll-forward recovery operations can be performed while a database remains online. When a tablespace is backed up online, it remains available for use and all simultaneous modifications to the data stored in that tablespace are recorded in the transaction log files. However, when an online restore or online roll-forward recovery operation is performed against a tablespace, the tablespace itself is not available for use until the operation has completed.

Incremental Backup and Recovery

As the size of a database grows, the time and hardware needed to backup and recover the databases also grows substantially. Furthermore, creating full database and tablespace backup images is not always the best approach when dealing with large databases, because the storage requirements for multiple copies of such databases can be enormous. A better alternative is to create one full backup image and several incremental backup images as changes are made. An incremental backup is a backup image that only contains pages that have been
updated since the previous backup image was made. Along with updated data and index pages, each incremental backup image also contains all of the initial database meta-data (such as database configuration, tablespace definitions, recovery history file, etc.) that is normally found in full backup images.

Two types of incremental backup images are supported: *incremental* and *delta*. An incremental backup image is a copy of all database data that has changed since the most recent, successful, full backup operation has been performed. An incremental backup image is also known as a cumulative backup image, because a series of incremental backups taken over a period of time will have the contents of the previous incremental backup image. The predecessor of an incremental backup image is always the most recent successful full backup of the same object.

A delta backup image is a copy of all database data that has changed since the last successful backup (full, incremental, or delta) of the object in question. A delta backup image is also known as a differential, or noncumulative, backup image; the predecessor of a delta backup image is the most recent successful backup image that contains a copy of each of the objects found in the delta backup image.

The key difference between incremental and delta backup images is their behavior when successive backups are taken of an object that is continually changing over time. Each successive incremental image contains the entire contents of the previous incremental image, plus any data that has changed, or has been added, since the previous full backup image was produced. Delta backup images only contain the pages that have changed since the previous backup image of any type was produced. In either case, database recovery involves restoring the database using the most recent successful full backup image available and applying each incremental backup image produced, in the order in which they were made. (The recovery history file keeps track of which incremental and delta backup images are needed and the order that they were made in.)

**Performing a Crash Recovery Operation**

Earlier, we saw that whenever transaction processing is interrupted by an unexpected event (such as a power failure), the database the transaction was interacting with at the time is placed in an inconsistent state. Such a database will remain in an inconsistent state and will be unusable until a crash recovery operation returns it to some point of consistency. (An inconsistent database
will notify users and applications that it is unusable via a return code and error message that is generated each time an attempt to establish a connection to it is made.)

So just how is a crash recovery operation initiated? One way is by executing the `RESTART` command from the DB2 Command Line Processor (CLP). The basic syntax for this command is:

```
RESTART [DATABASE | DB] [DatabaseName]
<User [UserName] <USING [Password]>>
<DROP PENDING TABLESPACES ( [TS_Name], ... )>
<WRITE RESUME>
```

where:

- **DatabaseName** Identifies the name assigned to the database that is to be returned to a consistent and usable state.
- **UserName** Identifies the name assigned to a specific user whose authority the crash recovery operation is to be performed under.
- **Password** Identifies the password that corresponds to the name of the user that the crash recovery operation is to be performed under.
- **TS_Name** Identifies the name assigned to one or more tablespaces that are to be disabled and placed in “Drop Pending” mode if errors are encountered while trying to return them to a consistent state.

Thus, if you wanted to perform a crash recovery operation on an inconsistent database named SAMPLE, you could do so by executing a `RESTART` command that looks something like this:

```
RESTART DATABASE SAMPLE
```

**NOTE**

If all database I/O was suspended (using the `SET WRITE` command) at the time a crash occurred, I/O must be resumed (using the `WRITE RESUME` option of the `RESTART` command) as part of the crash recovery process. (We'll look at the `SET WRITE` command a little later.)

You can also initiate a crash recovery operation for a particular database by selecting the `Restart` action from the `Databases` menu found in the Control Center. Figure 7–7 shows the Control Center menu items that must be selected in order to perform a crash recovery operation on an unusable database.
It is also possible to configure a database in such a way that crash recovery will automatically be performed, if necessary, when an application or user attempts to establish a connection to it. This is done by assigning the value ON to the `autorestart` database configuration parameter. (The DB2 Database Manager checks the state of a database the first time an attempt to establish a connection to the database is made and if it determines that the database is in an inconsistent state, it executes the `RESTART` command automatically if the `autorestart` database configuration parameter is set to ON.)

Any user can restart an inconsistent database; no authorization checking is performed.

It is important to note that if a crash recovery operation is performed on a recoverable database (i.e., a database that has been configured to support forward recovery operations) and an error occurs during the recovery process that is attributable to an individual tablespace, that tablespace will be taken offline, and will no longer be accessible until it is repaired. This has no effect on crash recovery itself and upon completion of the crash recovery operation, all other tablespaces in the database will be accessible and connections to the
database can be established—provided the tablespace that is taken offline is not the tablespace that contains the system catalogs. If the tablespace containing the system catalogs is taken offline, it must be repaired before any connections to the database will be permitted.

A Word about Soft Checkpoints

It was mentioned earlier that crash recovery is performed by using information stored in the transaction log files to roll back all incomplete transactions found and complete any committed transactions that were still in memory (but had not yet been externalized to storage) when the transaction failure occurred. As you might imagine, if the transaction log files for a database are large, it could take quite a while to scan the entire log and check for corresponding rows in the database. However, it's usually not necessary to scan the entire log since records recorded at the beginning of a log file are usually associated with transactions that have been completed and have already been externalized to the database. Furthermore, if these records can be skipped, the amount of time required to recover a crashed database can be greatly reduced.

That's where a mechanism known as the *soft checkpoint* comes in. The DB2 Database Manager uses a log control file to determine which records from a specific log file need to be applied to the database. This log control file is written to disk periodically, and the frequency at which this file is updated is determined by the value of the *softmax* database configuration parameter. Once the log control file is updated the soft checkpoint information stored in it establishes where in a transaction log file crash recovery should begin; all records in a log file that precede the soft checkpoint are assumed to be associated with transactions that have already been written to the database and are ignored.

Backup and Recovery

Although crash recovery can be used to resolve inconsistency problems that result from power interruptions and/or application failures, it cannot be used to handle problems that arise when the storage media being used to hold a database's files becomes corrupted or fails. In order to handle these types of problems, some kind of backup (and recovery) program must be put in place.

A database recovery strategy should include a regular schedule for making database backup images and, in the case of partitioned database systems, include making backup images whenever the system is scaled (i.e., whenever database partition servers are added or dropped). In addition, the strategy used should ensure that all information needed is available when database
Backup and Recovery

recovery is necessary and it should include procedures for recovering command scripts, applications, user-defined functions (UDFs), stored procedure code in operating system libraries, and load copies as well as database data. To help with such a strategy, DB2 UDB provides three utilities that are used to facilitate backing up and restoring a database. The utilities are:

➤ The BACKUP utility
➤ The RESTORE utility
➤ The ROLLFORWARD utility

The DB2 UDB BACKUP Utility

The single most important item you can possess that will prevent catastrophic data losses in the event storage media becomes corrupted or fails is a database backup image. A database backup image is essentially a copy of an entire database that includes both its objects and its data. Once created, a backup image can be used at any time to return a database to the exact state it was in at the time the backup image was made (version recovery). A good database recovery strategy should ensure that backup images are created on a regular basis, and that backup copies of critical data are retained in a secure location and on different storage media from that used to store the database itself. Depending on the logging method used (circular or archival), database backup images can be made when a database is offline or while other users and applications are connected to it (online). (In order to backup a database while it is online, archival logging must be used.)

A backup image of a DB2 UDB database (or of a tablespace within a DB2 UDB database) can be created by executing the BACKUP command. The basic syntax for this command is:

```
BACKUP [DATABASE | DB] [DatabaseName]
<User [UserName] <USING [Password]>>
<TABLESPACE ( [TS_Name], ... )
<ONLINE>
<INCREMENTAL <DELTAL>
<TO [Location]>
<WITH [NumBuffers] BUFFERS>
<BUFFER [BufferSize]>
<PARALLELISM [ParallelNum]>
<WITHOUT PROMPTING>
```

where:
Chapter 7

*DatabaseName* Identifies the name assigned to the database that a backup image is to be created for.

*UserName* Identifies the name assigned to a specific user whose authority the backup operation is to be performed under.

*Password* Identifies the password that corresponds to the name of the user that the backup operation is to be performed under.

*TS_Name* Identifies the name assigned to one or more specific tablespaces that are to be backed up.

*Location* Identifies the directory or device where the backup image created is to be stored. (If no location is specified, the current location is used as the default.)

*NumBuffers* Identifies the number of buffers that are to be used to perform the backup operation. (By default, two buffers are used if this option is not specified.)

*BufferSize* Identifies the size, in pages, that each buffer used to perform the backup operation will be. (By default, the size of each buffer used by the BACKUP utility is determined by the value of the *backbufsz* DB2 Database Manager configuration parameter.)

*ParallelNum* Identifies the number of tablespaces that can be read in parallel during a backup operation.

If the **INCREMENTAL** option is specified, an incremental backup image will be produced. An incremental backup image is a copy of all data that has changed since the last successful, full backup image was produced. Likewise, if the **DELTA** option is specified, a delta backup image will be produced. A delta backup image is a copy of all data that has changed since the last successful backup image of any type (full, incremental, or delta) was produced.

Thus, if you wanted to create a backup image of a database named SAMPLE and store the image created in a directory named BACKUPS on logical disk drive E:, you could do so by executing a **BACKUP** command that looks something like this:
BACKUP DATABASE SAMPLE
USER DB2ADMIN USING IBMDB2
TO E:\BACKUPS

On the other hand, if you wanted to create an incremental backup image of a
tablespace named TBSP1 and store the image created in a directory named
BACKUPS on logical disk drive E: while the database it is associated with
(named SAMPLE) remains online, you could do so by executing a BACKUP
command that looks something like this:

BACKUP DATABASE SAMPLE
USER DB2ADMIN USING IBMDB2
TABLESPACE (TBSP1) ONLINE INCREMENTAL TO E:\BACKUPS

Keep in mind that tablespace backup images can only be created if archival
logging is being used; if circular logging is used instead, tablespace backups
are not supported.

You can also create a backup image of a database or one or more tablespaces
using the Backup Wizard, which can be activated by selecting the Backup
action from the Databases menu found in the Control Center. Figure 7–8
shows the Control Center menu items that must be selected to activate the
Backup Wizard; Figure 7–9 shows how the first page of the Backup Wizard
might look immediately after it is activated.

![Backup Wizard](image-url)
Chapter 7

The Recovery History File

In Chapter 3—Data Placement, we saw that a special file, known as the recovery history file, is created as part of the database creation process. This file is used to log historical information about specific actions that are performed against the database it is associated with. Specifically, records are written to the recovery history file whenever any of the following are performed:

➤ A backup image of any type is created.
➤ A version recovery operation is performed either on the database or on one of its tablespaces.
➤ A table is loaded using the LOAD utility.
➤ A roll-forward recovery operation is performed either on the database or on one of its tablespaces.
➤ A tablespace is altered.
➤ A tablespace is quiesced.
➤ Data in a table is reorganized using the REORG utility.

NOTE

Only users with System Administrator (SYSADM) authority, System Control (SYSCtrl) authority, or System Maintenance (SYSMAINT) authority are allowed to backup a database or any of its tablespaces.
Statistics for a table are updated using the RUNSTATS utility.

A table is created, renamed, or deleted (dropped).

In addition to identifying the event that was performed, each entry in the recovery history file identifies the date and time the event took place, how the event took place, the tablespaces and tables that were affected, and the location of the backup image created (if the action was a backup operation), along with information on how to access this image. (In a moment, we’ll look at how this information is used when a version recovery operation is performed.)

Because the recovery history file sits quietly in the background and the DB2 Database Manager is responsible for managing its contents, a database administrator rarely has to interact with it. However, two commands are available that provide a way to both view the contents of a database’s recovery history file and to remove one or more entries stored in it. You can view the contents of a database’s recovery history file by executing the LIST HISTORY command from the DB2 Command Line Processor (CLP). The basic syntax for this command is:

```
LIST HISTORY
<BACKUP| ROLLFORWARD| DROPPED TABLE | LOAD | CREATE TABLESPACE | ALTER TABLESPACE |
RENAME TABLESPACE | REORG>
[ALL | SINCE [Timestamp] ] |
CONTAINING <SchemaName.>ObjectName
FOR [DATABASE | DB] [DatabaseName]
```

where:

**Timestamp**
Identifies a timestamp that is to be used as search criteria when retrieving entries from the recovery history file; only entries with timestamps that are greater than or equal to the timestamp provided are retrieved and displayed.

**SchemaName**
Identifies the name assigned to the schema that is to be used as search criteria when retrieving entries from the recovery history file; only entries that are associated with the schema name specified are retrieved and displayed.

**ObjectName**
Identifies the name assigned to an object that is to be used as search criteria when retrieving entries from the recovery history file; only entries that are associated with the object specified are retrieved and displayed.
Chapter 7

DatabaseName  Identifies the name assigned to the database that recovery history file information is to be retrieved and displayed for.

So, if you wanted to display all entries found in the recovery history file for a database named SAMPLE, you could do so by executing a LIST HISTORY command that looks something like this:

LIST HISTORY ALL FOR DATABASE SAMPLE

And when such a LIST HISTORY command is executed, output that looks something like the following might be produced (assuming the SAMPLE database has been backed up):

List History File for SAMPLE
Number of matching file entries = 1

<table>
<thead>
<tr>
<th>Op</th>
<th>Obj</th>
<th>Timestamp+Sequence</th>
<th>Type</th>
<th>Dev</th>
<th>Earliest Log</th>
<th>Current Log</th>
<th>Backup ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>D</td>
<td>20030817204927001</td>
<td>F</td>
<td>D</td>
<td>S00000000.LOG</td>
<td>S00000000.LOG</td>
<td></td>
</tr>
</tbody>
</table>

Contains 2 tablespace(s):

00001 SYSCATSPACE
00002 USERSPACE1

Comment: DB2 BACKUP SAMPLE OFFLINE
Start Time: 20030817204927
End Time: 20030817204957

You can delete a recovery history file entry by executing the PRUNE HISTORY command. The basic syntax for this command is:

PRUNE HISTORY [Timestamp]
<WITH FORCE OPTION>

where:

Timestamp  Identifies a timestamp that is to be used as search criterion when removing entries from the recovery history file; only entries with timestamps that are less than or equal to the timestamp provided are deleted, provided they are not part of the most recent restore set.

If the WITH FORCE OPTION option is specified, entries with timestamps that are less than or equal to the timestamp specified are deleted regardless of whether or not they are part of the most recent restore set.
Thus, if you wanted to remove all recovery history log file entries that were made prior to and including January 1, 2002, regardless of whether or not they are part of the most recent restore set, you could do so by executing a \texttt{PRUNE HISTORY} command that looks something like this:

\begin{verbatim}
PRUNE HISTORY 20020101 WITH FORCE OPTION
\end{verbatim}

It is important to note that where the \texttt{LIST HISTORY} command requires you to provide the name of the database whose recovery history file is to be queried, the \texttt{PRUNE HISTORY} command requires that you establish a connection to the appropriate database before attempting to remove one or more of its recovery history file entries.

**The DB2 UDB RESTORE Utility**

Earlier, we saw that version recovery is the process that returns a database to the state it was in at the time a backup image was made. This means that in order for version recovery to be available, at least one backup image must exist and be available. And since the recovery history file contains image location information for each backup image available, it acts as a tracking and verification mechanism during version recovery operations; each backup image contains special information in its header and this information is compared to the records stored in a database’s recovery history file to determine whether or not a particular backup image is associated with the database that is to be recovered.

So just how is a recovery operation initiated? The most common way is by executing the \texttt{RESTORE} command. The basic syntax for this command is:

\begin{verbatim}
RESTORE [DATABASE | DB] [DatabaseName]
USER [UserName] <USING [Password]>>
<TABLESPACE <ONLINE> | TABLESPACE ( [TS_Name],... ) <ONLINE> |
HISTORY FILE <ONLINE>>
INCREMENTAL <AUTO | AUTOMATIC | ABORT>>
FROM [SourceLocation]>
TO [TargetLocation]>
INTO [TargetAlias] > NEWLOGPATH [LogsLocation]>
WITH [NumBuffers] BUFFERS>
BUFFER [BufferSize]>
REPLACE EXISTING>
REDIRECT>
PARALLELISM [ParallelNum]>
WITHOUT ROLLING FORWARD>
WITHOUT PROMPTING>
\end{verbatim}
or

```
RESTORE [DATABASE | DB] [DatabaseName]
[CONTINUE | ABORT]
```

where:

- **DatabaseName** Identifies the name assigned to the database that is associated with the backup image that is to be used to perform a recovery operation.
- **UserName** Identifies the name assigned to a specific user that the recovery operation is to be performed under.
- **Password** Identifies the password that corresponds to the name of the user that the recovery operation is to be performed under.
- **TS_Name** Identifies the name assigned to one or more specific tablespaces that are to be restored from a backup image.
- **SourceLocation** Identifies the directory or device where the backup image to be used is stored.
- **Timestamp** Identifies a timestamp that is to be used as search criterion when looking for a particular backup image to use for recovery. (If no timestamp is specified there must be only one backup image at the source location specified.)
- **TargetLocation** Identifies the directory where the database that will be created is to be stored, if the backup image is to be used to create a new database.
- **TargetAlias** Identifies the alias to be assigned to the new database to be created.
- **LogsLocation** Identifies the directory or device where log files for the new database are to be stored.
- **NumBuffers** Identifies the number of buffers that are to be used to perform the recovery operation. (By default, two buffers are used if this option is not specified.)
- **BufferSize** Identifies the size, in pages, that each buffer used to perform the backup operation will be. (By default, the size of each buffer used by the
RESTORE utility is determined by the value of the `restbufsz` DB2 Database Manager configuration parameter.

ParallelNum Identifies the number of tablespaces that can be read in parallel during a backup operation.

Thus, if you wanted to restore a database named SAMPLE (which already exists and uses circular logging), using a backup image stored in a directory named BACKUPS on logical disk drive E:, you could do so by executing a RESTORE command that looks something like this:

```
RESTORE DATABASE SAMPLE
  USER DB2ADMIN USING IBMDB2
  FROM E:\BACKUPS
  REPLACE EXISTING
  WITHOUT PROMPTING
```

On the other hand, if you wanted to restore just a tablespace named TBSP1 in a database named SAMPLE from an incremental backup image stored in a directory named BACKUPS on logical disk drive E: while the database is online, you could do so by executing a RESTORE command that looks something like this:

```
RESTORE DATABASE SAMPLE
  USER DB2ADMIN USING IBMDB2
  TABLESPACE (TBSP1) ONLINE
  INCREMENTAL
  FROM E:\BACKUPS
```

Each full database backup image contains, among other things, a copy of the database’s recovery history file. However, when an existing database is restored from a full database backup image, the existing recovery history file is not overwritten. But what if the recovery history file for the database happens to be corrupted? Can the recovery history file be restored as well since a copy exists in the database backup image? The answer is yes. A special form of the RESTORE command can be used to restore just the recovery history file from a database backup image. Such a RESTORE command might look something like this:

```
RESTORE DATABASE SAMPLE
  USER DB2ADMIN USING IBMDB2
  HISTORY FILE
  FROM E:\BACKUPS
```

It is also possible to create an entirely new database from a full database backup image, effectively cloning an existing database. Thus, you could cre-
Create a new database named SAMPLE_2 that is an exact duplicate of a database named SAMPLE, using a backup image stored in a directory named BACK-UPS on logical disk drive E: by executing a RESTORE command that looks something like this:

```
RESTORE DATABASE SAMPLE
    USER DB2ADMIN USING IBMDB2
    FROM E:\BACKUPS
    INTO SAMPLE_2
```

It is important to note that if a backup image is used to create a new database, the recovery history file stored in the backup image will become the recovery history file for the new database.

You can also perform any of the restore/recovery operations just described (along with many others) using the Restore Data Wizard, which can be activated by selecting the Restore action from the Databases menu found in the Control Center. Figure 7–10 shows the Control Center menu items that must be selected to activate the Restore Data Wizard; Figure 7–11 shows how the first page of the Restore Data Wizard might look immediately after it is activated.
Backup and Recovery

Redirected Restore

A full backup image of a database contains, among other things, information about all tablespaces that have been defined for the database, including specific information about each tablespace container being used at the time the backup image was made. During a recovery operation, a check is performed to verify that all tablespace containers referenced by the backup image exist and are accessible. If this check determines that one or more of the tablespace containers needed is no longer available or is no longer accessible, the recovery operation will fail and the database will not be restored. When this happens, any invalid tablespace containers encountered can be redefined at the beginning of the recovery process by performing what is referred to as a redirected restore operation.

Redirected restore operations are performed by executing the RESTORE command with the REDIRECT option specified, followed by one or more SET TABLESPACE CONTAINERS commands, followed by the RESTORE command with the CONTINUE option specified. The basic syntax for the SET TABLESPACE CONTAINERS command is:
SET TABLESPACE CONTAINERS FOR \[TS_ID\] USING
\{( PATH \'\[Container\]\' ,.... ) |
( \[FILE | DEVICE\] \'\[Container\]\' \[ContainerSize\],.... )\}

where:

\(TS_ID\) Identifies the identification number assigned to the tablespace that new storage containers are to be provided for.

\(Container\) Identifies one or more containers that are to be used to store the data associated with the tablespace specified.

\(ContainerSize\) Identifies the number of pages to be stored in the tablespace container specified.

The steps used to perform a redirected restore operation are as follows:

1. Start the redirected restore operation by executing the RESTORE command with the REDIRECT option specified. (When this option is specified, each invalid tablespace container encountered is flagged, and all tablespaces that reference invalid tablespace containers are placed in the “Restore Pending” state. A list of all tablespaces affected can be obtained by executing the LIST TABLESPACES command.) At some point, you should see a message that looks something like this:

   SQL1277N Restore has detected that one or more table space containers are inaccessible, or has set their state to 'storage must be defined'.
   DB20000I The RESTORE DATABASE command completed success fully.

2. Specify new tablespace containers for each tablespace placed in “Restore Pending” state by executing a SET TABLESPACE CONTAINERS for each appropriate tablespace. (Keep in mind that SMS tablespaces can only use PATH containers, while DMS tablespaces can only use FILE or DEVICE containers.)

3. Complete the redirected restore operation by executing the RESTORE command with the CONTINUE option specified.

To simplify things, all of these steps can be coded in a UNIX shell script or a Windows batch file, which can then be used to perform the redirected restore operation. Such a file might look something like this:

```
db2 "RESTORE DATABASE SAMPLE FROM C:\BACKUPS TO D:\DB_DIR INTO SAMPLE_2 REDIRECT"
```
db2 "SET TABLESPACE CONTAINERS FOR 0 USING (PATH 'D:\DB_DIR\SYSTEM')"

db2 "SET TABLESPACE CONTAINERS FOR 1 USING (PATH 'D:\DB_DIR\TEMP')"

db2 "SET TABLESPACE CONTAINERS FOR 2 USING (PATH 'D:\DB_DIR\USER')"

db2 "RESTORE DATABASE SAMPLE CONTINUE"

You can also perform a redirected restore by assigning new tablespace containers to existing tablespaces on the Containers page of the Restore Data Wizard. Figure 7–12 shows how this is used to assist in a redirected restore operation.

You can also perform a redirected restore by assigning new tablespace containers to existing tablespaces on the Containers page of the Restore Data Wizard. Figure 7–12 shows how this is used to assist in a redirected restore operation.

In addition to providing new storage containers for tablespaces when older tablespace containers are inaccessible or are no longer present, a redirected restore can also be used to add new containers to existing SMS tablespaces. (The ALTER TABLESPACE command does not allow you to add new storage containers to existing SMS tablespaces; a redirected restore provides a workaround to this limitation.)

The DB2 UDB ROLLFORWARD Utility

When a backup image is used to restore a damaged or corrupted database, the database can only be returned to the state it was in at the time the backup
image was made. Therefore, all changes that were made to the database after the backup image was created will be lost when a recovery operation is performed. To return a database to the state it was in at any given point in time, roll-forward recovery must be used instead. And in order to perform a roll-forward recovery operation, the database must be recoverable (that is, archival logging must be the logging strategy used), you must have a full backup image of the database available, and you must have access to all archived log files that will be needed to perform the roll-forward recovery operation.

Roll-forward recovery starts out as a recovery operation. However, where a recovery operation will leave a non-recoverable database in a “Normal” state, the same operation will leave a recoverable database in “Roll-forward pending” state. (When a recoverable database is restored from a backup image, it is automatically placed in “Roll-forward pending” state unless the WITHOUT ROLLING FORWARD option is used with the RESTORE command; while a database is in “Roll-forward pending state, it cannot be accessed by users and applications.) At that point, the database can either be taken out of “Roll-forward pending” state (in which case all changes made to the database since the backup image used for recovery was made will be lost), or information stored in the database's transaction log files can be replayed to return the database to the state it was in at any given point in time.

The process of replaying transactions stored in archived log files is known as “rolling the database forward” and one way to roll a database forward is by executing the ROLLFORWARD command. The basic syntax for this command is:

ROLLFORWARD [DATABASE | DB] [DatabaseName]
<USER [UserName] <USING [Password]>>
<TO [PointInTime] <USING LOCAL TIME>
  <AND COMPLETE | AND STOP> |
  TO END OF LOGS <AND COMPLETE | AND STOP> |
  COMPLETE |
  STOP |
  CANCEL |
  QUERY STATUS <USING LOCAL TIME>>
<TABLESPACE ONLINE |
  TABLESPACE <( [TS_Name] ,..., )> <ONLINE>>
<OVERFLOW LOG PATH ( [LogDirectory] ,..., )>
<RECOVER DROPPED TABLE [TableID] TO [Location]>

Once a database is taken out of “Roll-forward pending” state, it cannot be manually returned to that state again. Therefore, if you discover that a database was taken out of “Roll-forward pending” state prematurely and you need to return it to that state, you must restore the database again using an appropriate backup image.
where:

- **DatabaseName** identifies the name assigned to the database that is to be rolled forward.
- **UserName** identifies the name assigned to a specific user that the roll-forward operation is to be performed under.
- **Password** identifies the password that corresponds to the name of the user that the roll-forward operation is to be performed under.
- **PointInTime** identifies a specific point in time, identified by a timestamp value in the form `yyyy-mm-dd-hh:mm:ss.mmmmm` (year, month, day, hour, minutes, seconds, microseconds) that the database is to be rolled forward to. (Only transactions that took place before and up to the time specified will be reapplied to the database.)
- **TS_Name** identifies the name assigned to one or more specific tablespaces that are to be rolled forward.
- **LogDirectory** identifies the directory that contains offline archived log files that are to be used to perform the roll-forward operation.
- **TableID** identifies a specific table (by ID) that was dropped earlier that is to be recovered as part of the roll-forward operation. (The table ID can be obtained by examining the database's recovery history file.)
- **Location** identifies the directory where files containing data that was stored in the table that was dropped are to be written to when the table is recovered as part of the roll-forward operation.

If the **AND COMPLETE, AND STOP, COMPLETE, or STOP** option is specified, the database will be returned to “Normal” state when the roll-forward operation has completed. Otherwise, the database will remain in “Roll-forward pending state”. (When a recoverable database is restored from a backup image, it is automatically placed in “Roll-forward pending” state unless the **WITHOUT ROLLING FORWARD** option is used with the **RESTORE** command; while a database is in “Roll-forward pending state, it cannot be accessed by users and applications.)
Thus, if you wanted to perform a roll-forward recovery operation on a database named SAMPLE and take it out of “Roll-forward pending” state, you could do so by executing a `ROLLFORWARD` command that looks something like this:

```
ROLLFORWARD DATABASE SAMPLE TO END OF LOGS AND STOP
```

On the other hand, if you wanted to perform a roll-forward recovery operation on a database named SAMPLE by reapplying all transactions that were committed at or before 01/01/2003, you could do so by executing a `ROLLFORWARD` command that looks something like this:

```
ROLLFORWARD DATABASE SAMPLE TO 2003-01-01-00.00.00.0000 AND STOP
```

It is important to note that the time value specified is interpreted as a Coordinated Universal Time (UTC), otherwise known as Greenwich Mean Time (GMT), value. If a `ROLLFORWARD` command that looks something like this had been executed instead:

```
ROLLFORWARD DATABASE SAMPLE TO 2003-01-01-00.00.00.0000 USING LOCAL TIME AND STOP
```

The time value specified would have been interpreted as a local time value.

You can also initiate a roll-forward recovery operation using the Rollforward Wizard, which can be activated by selecting the `Roll-forward` action from the `Databases` menu found in the Control Center. Figure 7–13 shows the Control Center menu items that must be selected to activate the Rollforward Wizard; Figure 7–14 shows how the first page of the Rollforward Wizard might look immediately after it is activated.

Because a roll-forward recovery operation is typically performed immediately after a database is restored from a backup image, a roll-forward recovery operation can also be initiated by providing the appropriate information on the Roll forward page of the Restore Data Wizard. Figure 7–15 shows how the Roll forward page of the Restore Data Wizard might look after its input fields have been populated.

**NOTE**

Only users with System Administrator (SYSADM) authority, System Control (SYSCtrl) authority, or System Maintenance (SYSMaint) authority are allowed to perform a roll-forward recovery operation.
Figure 7–13 Invoking the Rollforward Wizard from the Control Center.

Figure 7–14 The first page of the Rollforward Wizard.
Rebuilding Invalid Indexes

So far we have looked at ways to recover data in the event the storage media being used to hold a database’s files becomes corrupted or fails. But what if only indexes are damaged and a database’s data is unaffected (which could be the case if data and indexes are stored in separate DMS tablespaces and only the tablespace container where index data is stored fails)? In this case, the affected indexes are invalidated and can be recovered by being recreated once the faulty media has been replaced.

Whenever the DB2 Database Manager detects that an index is no longer valid, it automatically attempts to rebuild it. However, the point in time at which the DB2 Database Manager attempts to rebuild an invalid index is controlled by the `indexrec` parameter of the database or the DB2 Database Manager configuration file. There are three possible settings for this parameter:

**SYSTEM.** Invalid indexes are to be rebuilt at the time specified in the `indexrec` parameter of the DB2 Database Manager configuration file. (This setting is only valid for database configuration files.)

**RESTART.** Invalid indexes are to be rebuilt when crash recovery is performed on the database (i.e., when the database is restarted).

**ACCESS.** Invalid indexes are to be rebuilt the first time they are accessed after they have been marked as being invalid.
So when is the best time to rebuild invalid indexes? If the time it takes to perform a crash recovery operation on a database is not a concern, it is better to let the DB2 Database Manager rebuild invalid indexes while it is returning a database to a consistent state; the time needed to restart a database will be longer due to the index recreation process, but normal processing will not be affected. On the other hand, if indexes are rebuilt as they are accessed, crash recovery can be performed faster, but users may experience an initial decrease in performance; references made to tables that contain associated invalid indexes will have to wait for the invalid index(es) to be rebuilt. Furthermore, unexpected locks may be acquired and held long after an invalid index has been recreated, especially if the transaction that caused the index recreation to occur is not committed (or rolled back) for quite some time.

**Backing Up a Database with Split Mirroring**

It was mentioned earlier that, as databases increase in size and as heavy usage demands require databases to be available 24 hours a day, seven days a week, the time and hardware needed to backup and restore a database can increase substantially. Backing up an entire database or several table spaces of a large database can put a strain on system resources, require a considerable amount of additional storage space (to hold the backup images), and can reduce the availability of the database system (particularly if the system has to be taken off line in order to be backed up). Therefore, a popular alternative to creating and maintaining backup images of such databases is to use what is known as a *split mirror*.

A split mirror is an “instantaneous” copy of a database that is made by mirroring the disk(s) that contain the database’s data, and splitting the mirror when a backup copy of the database is required. *Mirroring* is the process of writing all database data to two separate disks (or disk subsystems) simultaneously; one disk/subsystem holds the database data while the other holds an exact copy (known as a *mirror*) of the primary disk/subsystem being used. *Splitting* a mirror simply involves separating the primary and secondary copies of the database from each other. Split mirroring provides the following advantages:

➤ The overhead required to create backup images of the database is eliminated
➤ Entire systems can be cloned very quickly
➤ Provides a fast implementation of idle standby failover.
To further enhance split mirroring, DB2 UDB provides a way to temporarily suspend (and later resume) all database I/O so that a mirror can be split without having to take a database offline. The command that provides this functionality is the `SET WRITE` command and the syntax for this command is:

```
SET WRITE [SUSPEND | RESUME] FOR [DATABASE | DB]
```

Thus, if you wanted to temporarily suspend all I/O for a database, you would do so by establishing a connection to that database and executing a `SET WRITE` command that looks like this:

```
SET WRITE SUSPEND FOR DATABASE
```

When executed, the `SET WRITE SUSPEND FOR DATABASE` command causes the DB2 Database Manager to suspend all write operations to tablespace containers and log files that are associated with the current database. (The suspension of writes to tablespaces and log files is intended to prevent partial page writes from occurring until the suspension is removed.) All database operations, apart from online backup and restore operations, will function normally while database writes are suspended. That’s because read-only transactions are not suspended and are able to continue working with the suspended database, provided they do not request a resource that is being held by the suspended I/O process. Furthermore, applications can continue to process insert, update, and delete operations using data that has been cached in the database’s buffer pool(s). However, new pages cannot be read into the buffer pool(s) and no new database connections can be established.

I/O for a suspended database can be resumed at any time by executing a `SET WRITE` command that looks like this:

```
SET WRITE RESUME FOR DATABASE
```

When executed, the `SET WRITE RESUME FOR DATABASE` command causes the DB2 Database Manager to lift all write suspensions and to allow write operations to tablespace containers and log files that are associated with the current database to continue.

Database I/O must be resumed from the same connection from which it was suspended.
Initializing a Split Mirror with db2inidb

Before a split mirror copy of a DB2 UDB database can be used, it must first be initialized; a split mirror database copy is initialized by executing the system command db2inidb. The syntax for this command is:

```
db2inidb [DatabaseAlias]
  AS [SNAPSHOT | MIRROR | STANDBY]
  <RELOCATE USING [ConfigFile]>
```

where:

- **DatabaseAlias** Identifies the alias assigned to the database the split mirror copy that is to be initialized references.
- **ConfigFile** Indicates that the database files contained in the split mirror copy are to be relocated according to information stored in the configuration file specified.

As you can see, a split mirror database copy can be initialized in one of three ways:

- **SNAPSHOT.** The split mirror copy of the database will be initialized as a clone of the primary database. (It will be a working copy that has its own transaction log files.)
- **MIRROR.** The split mirror copy of the database will be initialized as a backup image that can be used to restore the primary database.
- **STANDBY.** The split mirror copy of the database will be initialized and placed in roll-forward pending state so that it can be continuously synchronized with the primary database. (New logs from the primary database can be retrieved and applied to the copy of the database at any time.) The standby copy of the database can then be used in place of the primary database if, for some reason, the primary database goes down.

Thus, if you wanted to initialize a split mirror copy of a database named SAMPLE and make it a backup image that can be used to restore the primary database, you could do so by executing a db2inidb command that looks like this:

```
db2inidb SAMPLE AS MIRROR
```

(The split mirror copy of the SAMPLE database used could have been created while the database was online by temporarily suspending I/O with the `SET WRITE` command, making the split mirror copy of the database using an appropriate non-DB2 UDB utility, and resuming I/O as soon as the split mirror copy was successfully created.)
Practice Questions

Question 1

Which of the following is the primary purpose for using infinite logging?

- A. Eliminate the need to archive log files as they become full.
- B. Eliminate the need to specify the number of primary log files used.
- C. Eliminate the need to specify the size of all log files used.
- D. Support large transactions whose logging activity would normally exceed the space provided by primary and secondary log files.

Question 2

Which of the following is NOT supported when LOGRETAIn is set to RECOVERY and USEREXIT is OFF?

- A. Circular logging
- B. Archival logging
- C. Roll-forward recovery
- D. Crash recovery

Question 3

When attempting to connect to a database named SAMPLE, the following message is displayed:

SQL1015N The database must be restarted because the previous session did not conclude normally.

In order to correct the situation, which of the following should be performed?

- A. RESTORE DATABASE sample
- B. RESTART DATABASE sample
- C. RECOVER DATABASE sample
- D. RESET DATABASE sample
Question 4

Some of a database's indexes need to be rebuilt. To minimize application response time, which of the following is the correct setting of the INDEXREC database configuration parameter?

- A. ACCESS
- B. RESTART
- C. IMMEDIATE
- D. DEFERRED

Question 5

The REDIRECT option of the RESTORE command is used to do which of the following?

- A. Restore a database to a location that is different from its original location.
- B. Restore a database whose tablespace containers reference invalid drives/devices.
- C. Restore a database and at the same time, convert all DMS tablespaces to SMS tablespaces.
- D. Restore a database and at the same time, convert all SMS tablespaces to DMS tablespaces.

Question 6

The following commands have been entered:

UPDATE DB CFG FOR sample USING LOGRETAIN YES IMMEDIATE
BACKUP DB sample TO c:\backups
CONNECT TO sample
INSERT INTO department (deptid, deptname) VALUES ('001', 'RESEARCH')
RESTORE DB sample FROM c:\backups

Which two of the following commands can be issued to make the SAMPLE database usable?

- A. RESTART DB sample
- B. ROLLFORWARD DB sample to END OF LOGS
- C. ROLLFORWARD DB sample COMPLETE
- D. ROLLFORWARD DB sample to END OF LOGS AND STOP
- E. ROLLFORWARD DB sample CONTINUE
Question 7

Given the following command:

```
RESTORE DATABASE sample FROM c:\backups
```

Which of the parameters determines how much buffer space will be used to restore the SAMPLE database?

- A. logbufsz
- B. buffpage
- C. restbufsz
- D. util_heap_sz

Question 8

Given that the USEREXIT configuration parameter for a database is set to YES, which of the following configuration parameters must be set to enable infinite logging?

- A. LOGPRIMARY = -1
- B. LOGSECOND = -1
- C. LOGFILSZ = -1
- D. LOGRETAIN = -1

Question 9

Which combination of database configuration parameters make a database non-recoverable?

- A. LOGRETAIN=RECOVERY, USEREXIT=YES
- B. LOGRETAIN=NO, USEREXIT=YES
- C. LOGRETAIN=RECOVERY, USEREXIT=NO
- D. LOGRETAIN=NO, USEREXIT=NO

Question 10

When does a log file become an off-line archived log file?

- A. When it is moved from the active log directory
- B. When it is no longer needed for crash recovery
- C. When it is full and a new log is being processed
- D. When it becomes full
Question 11

Given the following command:

```
ROLLFORWARD DATABASE sample TO 2003-01-01-00.00.00.0000 AND STOP
```

How is the time 2003-01-01-00.00.00.0000 interpreted by the `ROLLFORWARD` command?

- A. As a local time value
- B. As a coordinated universal time (UTC) value
- C. As a timestamp value
- D. As a world time value

Question 12

Which of the following configuration parameters enables dual logging for a database?

- A. newlogpath
- B. seqlogpath
- C. mirrorlogpath
- D. overflowlogpath

Question 13

Which of the following occurs when the command `SET WRITE SUSPEND FOR DATABASE` is issued?

- A. All tablespace writes are suspended; all log writes are suspended; and all database operations continue to function normally.
- B. All tablespace writes are suspended and all database operations continue to function normally.
- C. All tablespace writes are suspended; all log writes are suspended; and all database operations other than online backup and restore operations continue to function normally.
- D. All tablespace writes are suspended and all database operations other than online backup and restore operations continue to function normally.
Question 14

An incremental backup image contains which of the following?

- A. A copy of all data and index changes made since the last successful backup (full, incremental, or delta) was made.
- B. A copy of all data and index changes made since the last successful full backup image was made.
- C. A copy of all data, index, and database meta-data that has changed since the last successful backup (full, incremental, or delta) was made.
- D. A copy of all data, index, and database meta-data that has changed since the last successful full backup image was made.

Question 15

Which of the following commands will produce just a list of all backup operations performed on a database named SAMPLE?

- A. LIST HISTORY BACKUP ALL FOR sample
- B. LIST BACKUP HISTORY FOR sample
- C. LIST ALL BACKUP HISTORY FOR sample
- D. LIST RECOVERY FILE BACKUP HISTORY FOR sample

Question 16

An offline database backup that was started for a database named SAMPLE at 1:00 AM local time completed at 2:30 AM local time. Assuming the SAMPLE database is a recoverable database, and that it is now 11:30 AM, which of the following commands must be performed in order to restore the SAMPLE database to the state it was in at 2:30 this morning?

- A. RESTORE DATABASE sample FROM c:\backups
- B. RESTORE DATABASE sample FROM c:\backups ROLLFORWARD TO 02:30.00.000000 USING LOCAL TIME
- C. RESTORE DATABASE sample FROM c:\backups WITHOUT ROLLING FORWARD
- D. RESTORE DATABASE sample FROM c:\backups ROLLFORWARD STOP
Answers

Question 1

The correct answer is D. If you are concerned about running out of log space and you want to avoid allocating a large number of secondary log files, you can configure a database to perform infinite logging. To enable infinite logging, you simply set the database configuration parameters userexit and logsecondary to ON and −1, respectively.

Question 2

The correct answer is A. If the logretain configuration parameter for a database is set to RECOVERY and/or the userexit parameter is set to YES, archival logging is used and roll-forward recovery operations can be performed against the database. On the other hand, when both of these configuration parameters are set to NO, which is the default, circular logging is used and roll-forward recovery is not supported. In either case, crash recovery is supported.

Question 3

The correct answer is B. Whenever transaction processing is interrupted by an unexpected event (such as a power failure), the database the transaction was interacting with at the time is placed in an inconsistent state. Such a database will remain in an inconsistent state and will be unusable until a crash recovery operation returns it to some point of consistency; an inconsistent database will notify users and applications that it is unusable via a return code and error message that is generated each time an attempt to establish a connection to it is made. In such a case, crash recovery can be initiated by executing the RESTART command. (Had the autorestart database configuration parameter been set to ON, a crash recovery operation would have been started automatically when the user attempted to connect to the SAMPLE database and the error message would not have been displayed.)

Question 4

The correct answer is B. If the indexrec parameter of a database’s configuration file is set to RESTART, invalid indexes will be rebuilt, either explicitly or implicitly, when the database is restarted (i.e., when crash recovery is per-
formed on the database) and it will take longer to restart the database. On the other hand, if the indexrec parameter is set to ACCESS, invalid indexes will be rebuilt the first time they are accessed (after they have been marked as being invalid) and users may experience a decrease in performance.

Question 5

The correct answer is B. The purpose of a redirected restore (which is initiated by executing the RESTORE command with the REDIRECT option specified) is to redefine any invalid tablespace containers encountered at the beginning of the recovery process. (The TO [Location] option of the RESTORE command is used to restore a database to a location that is different from its original location and it's impossible to change tablespace types with a restore operation, or any other operation for that matter.)

Question 6

The correct answers are C and D. Because the logretain database configuration parameter has been set to YES, roll-forward recovery has been enabled for the SAMPLE database. As a result, when the database is restored using the RESTORE command, it will automatically be placed in “Roll-forward pending” state. While the database is in “Roll-forward pending” state, it cannot be accessed by users and applications. And the only way a database can be taken out of “Roll-forward pending” state is by executing the ROLLFORWARD command with either the STOP or the COMPLETE option specified.

Question 7

The correct answer is C. If the BUFFER option of the RESTORE command is not used to specify how much buffer space should be reserved for the RESTORE utility, the amount of buffer space used is determined by the value of the restbufsz DB2 Database Manager configuration parameter. (Likewise, if the BUFFER option of the BACKUP command is not used to specify how much buffer space should be reserved for the BACKUP utility, the amount of buffer space used is determined by the value of the backbufsz DB2 Database Manager configuration parameter.)
Question 8

The correct answer is B. Infinite logging is enabled by setting the *userexit* database configuration parameter to YES and the *logsecond* database configuration parameter to –1.

Question 9

The correct answer is D. If the *logretain* configuration parameter for a database is set to RECOVERY and/or the *userexit* parameter is set to YES, archival logging is used, roll-forward recovery operations can be performed against the database, and the database is considered to be recoverable. On the other hand, when both of these configuration parameters are set to NO, circular logging is used, roll-forward recovery is not supported, and the database is considered to be non-recoverable.

Question 10

The correct answer is A. An online archive log file is a log file that contains records that are associated with completed transactions that resides in the active log directory. As soon as an online archive log file is moved from the active log directory to another storage location, it becomes an offline archive log file.

Question 11

The correct answer is B. Unless the *ROLLFORWARD* command is executed with the *USING LOCAL TIME* option, all time values are assumed to be Coordinated Universal Time (also referred to as Greenwich Mean Time) values.

Question 12

The correct answer is C. To enable log file mirroring (also referred to as dual logging), you simply assign the fully qualified name of the mirror log location to the *mirrorlogpath* database configuration parameter.

Question 13

The correct answer is C. When executed, the *SET WRITE SUSPEND FOR DATABASE* command causes the DB2 Database Manager to suspend all write
operations to tablespace containers and log files that are associated with the current database. All database operations, apart from online backup and restore operations, will function normally while database writes are suspended. That's because read-only transactions are not suspended and are able to continue working with the suspended database, provided they do not request a resource that is being held by the suspended I/O process. Furthermore, applications can continue to process insert, update, and delete operations using data that has been cached in the database's buffer pool(s). However, new pages cannot be read into the buffer pool(s) and no new database connections can be established.

**Question 14**

The correct answer is D. An incremental backup is a backup image that only contains pages that have been updated since the previous backup image was made. Along with updated data and index pages, each incremental backup image also contains all of the initial database meta-data (such as database configuration, tablespace definitions, recovery history file, etc.) that is normally found in full backup images.

**Question 15**

The correct answer is A. Information about backup operations is recorded in a database's recovery history file and you can view the contents of a recovery history file by executing the `LIST HISTORY` command. The proper syntax for listing just backup operation information with the `LIST HISTORY` command is `LIST HISTORY BACKUP ALL FOR [DatabaseName].`

**Question 16**

The correct answer is C. When a recoverable database is restored from a backup image, it is automatically placed in “Roll-forward pending” state unless the `WITHOUT ROLLING FORWARD` option is used with the `RESTORE` command. Because the backup image was made while the database was offline, no changes were made between 1:00 AM and 2:30 AM. Therefore, by restoring the database from the backup image that was taken at 1:00 AM and by specifying the `WITHOUT ROLLING FORWARD` option, the database will be returned to the state it was in at 2:30 AM and it will be taken out of “Roll-forward pending” state.