Transaction Management

Recovery

The objectives of this section are that the student will be able to

1. Explain the need for recovery/concurrency in the database environment
2. Describe the main components of a recovery system
3. Explain the term transactions; what are they, how do complicate the work of the DBMS
4. Explain the need for and use of logs, checkpoints and archive databases.
5. Develop/Devise recovery procedures (Business Continuity)

**Introduction.**

Interface User1 User N

SQL reports/queries.

Tables

DBMS

OS

|  |
| --- |
| Cache |
| Processor |
| Buffers |

Files

DATA

Disk

The exact architecture of main memory is database (product) dependent i.e. Oracle, MySql etc. Recall from section on architecture that you could have Data cache for query data, catalog cache for meta data etc ,or it could be local cache to each connection or global/shared cache. The important thing to note from a recovery point of view is that all main memory is volatile. Also, the terms cache/buffer can mean slightly different things but we will use them interchangeably.

**Physical Design:** Tables are logical organisations from the users point of view. The way in which tables end up as files and how they are physically stored on a secondary storage medium is part of the responsibility of the DBA/physical DB designer. The table design must be logically correct (normalisation) but it must also be an efficient physical implementation on the system. (this is **called physical design)**

We should understand from Operating Systems that files are not necessarily contiguous blocks of storage allocated in one block. Assume the existence of a logical page table for a file containing the page addresses of each physical disk page allocated. The logical pages ‘look’ allocated in sequence but the corresponding physical pages are not i.e. the file can be physically spread out over the disk using any available pages, but the file page table maintains the sequence. The unit of I/O from Disk to RAM is the page. The OS usually retains control of the page replacement algorithm for RAM i.e. what is swopped out when a page comes in.

Logical file Physical(DISK file)

TABLE

|  |  |
| --- | --- |
| Page No. | Page address |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |

Aside: Windows Defragmentation utilities. Check this to see how parts of a single file can be spread out over the disk and must be ‘reorganised’ to de-fragment the disk and give better performance.

We should easily understand that data is stored in caches for reuse, or buffers for I/O.

But why does data build up in buffers? Ans: Block I/O aka Asynchronous block I/O.

Question: **Why is block I/O used and why is it asynchronous?**

Ans**: I/O efficiency**. The bottle neck is usually the Disk (2nd storage).

You have 3 Disk delays:

* Seek time is the highest, then
* Disk Rotation, with
* Transmission as the shortest delay.

So for every rotation & seek, it is inefficient to transmit just a little bit of data; so you should try and transmit a larger group of data items (e.g. blocks of data) for efficiency.

Also note the basic unit of I/O transfer is the page. The page size varies from 2K to 64K.

However a DB record can be small e.g. an employee record made up of attributes Empno (Int), Emp\_name(Char(20) = a total of 24 bytes (assuming that int data-type requires 4 bytes). So we actually fit many DB records into one page. Therefore, in effect the system writes in at least one page block anyway so extending the block size to a number of pages should be readily understandable.

So you need to divide the page size in use by the size of each record. This tells you the number of records that will fit in each page.

No. of records per page divide into expected no. of records = total number of pages of storage.

So keep in mind that the buffer DB **normally** writes in groups (not on a record by record basis). This in important as it explains why an end user modification (let’s say an SQL Update of a single record) may NOT be written directly to the disk after the transaction has finished; the record might be in a page that contains other records that are currently being operated on by a concurrent transaction OR it could be in the I/O buffer waiting for a group/block write. The longer it waits to be written the higher the risk of failure.

As a final point on the influence of the memory/cache manager on the recovery system of the DBMS, it must be noted that the RAM/Cache manager works independently of the transaction manager (scheduler), or the recovery manager. Pages are selected to be written to disk by the **RAM/Cache manager** replacement policy e.g. oldest, least frequently used, least recently used etc not by the SQL program. Therefore, pages in cache/buffer may be written out to disk (unknown to the SQL program) before commit (if the buffer manager is independent).

Why might this happen? There is time between an update and an issuing of a commit; there are concurrent programs all accessing the database. If the cache fills, a concurrent program referencing a record that is not in one of the pages in RAM memory will cause the buffer manager to do a page fetch operation. It must also choose one of the existing pages to be removed from the cache i.e. written back to the disk to make room for the incoming page.

**After a transaction finishes, some or all of the records it changed may remain in RAM** because they are

a) In a page not chosen by the Page Replacement Algorithm,

b) In a page used by a concurrent transaction and

c) Held in buffer for block I/O.

**Understanding asynchronous (independent) buffer\cache management and block I/O is essential to understanding recovery procedures in a transaction based DBMS.**

**So for recovery, you need to understand that a page contains a large number of records, and the file is made up of a large number of pages. When a record is updated or page changed (for update, Insert) then that page may be written back to Disk (permanent storage) or not. These two possible states are critical for recovery control.**

**Furthermore, some DBMS implement ‘read ahead’ by reading a number of pages ‘in one go’. A DBMS is not guaranteed that records required for processing will be in those pages.**

A database ‘Select’ is implemented by:

• Read entire page containing item to buffer/cache in DBMS

• Transfer item from buffer to application

A database ‘update’ is implemented by

• Read page containing item to buffer in DBMS (if it is not already there)

• Update item in DBMS buffer/cache

• (Eventually) write buffer page to disk

Record blocks & buffers summary

The system may be designed so that it physically reads a block of data records at a time into a memory buffer or cache for access of individual logical records by the CPU. The block read may take longer than reading individual records (which are usually small amounts of memory)) as they are required, however the benefits of now having a number of logical records (e.g. group of supplier records) available to the user program may be more advantageous in the long run (Note this depends on type of processing by the program. Explain why?).

We now know that a page is the (minimum) unit of I/O between primary and secondary storage. Page sizes vary (and may be configured by the DBA). In normal applications many data records fit into a page; so the idea of blocking is used in page I/O ie. many records read in one physical read.

For certain types of applications (types of processing), this organisation may be so advantageous that the system employs a larger block read i.e. a number of pages are read for any user read(select) where the required record is not in main memory.

**Transactions** :

Example 1 : what if you wanted to move 2000 Eu between two accounts? We have a starting home account balance of 2500 and off shore account balance of 0; a transaction to do this might looks like

Time

1 Update homeaccount set balance = balance – 2000 Home Ac Bal

2 (\* temporary inconsistency at this time \*) 500

3 Update offshoreaccount set balance = balance + 2000 OffS Ac Bal

2000

Save Cancel

Cancel means discard changes. Understand the notion of working on a copy/version of the data item that you can opt not to save ( as in Text Editor/Word Processor). Cancel in a DB is more complicated in that we require different recovery actions if we make changes directly to the data item on the disk as opposed to a copy of the data where we can simply discard changes.

If we introduce a failure at time 2 what happens

Time

1 Update homeaccount set balance = balance – 2000 Home Ac Bal

2 Failure 500

3 OffS Ac Bal

0

A transaction is a **logical unit of work**. A transaction is not necessarily just a single database operation. It may be a sequence of several operations that transforms the database from one consistent (correct!) state to another. Database operations usually consist of operations on records in the database e.g. Insert, delete, update, select/retrieve etc. It is possible that **intermediary points during a transaction leave the database in an inconsistent state but the end state must be correct**. This property of transactions has critical implications for recovery i.e. returning the database to a correct/consistent point (note use interchangeably from now on)

A system that supports transaction processing guarantees that if the transaction executes some updates and then a failure occurs, for whatever reason, before the transaction reaches its planned termination point, then all those updates will be undone.

**Thus the transaction either executes in its entirety or it is totally cancelled i.e. as if it never executed at all (This is called Atomicity)**

In this way a sequence of operations that is fundamentally not atomic can be made to ‘look’ as if it really were atomic from an external and/or system point of view.

These programs introduce three important transaction-processing statements required for recovery/concurrency.

**BEGIN TRANSACTION** defines the start of a unit of work or sequence of operations. The database is consistent at this point in time.(from this transactions point of view)

**COMMIT** signals successful end-of-transaction. All operations within the transaction **are to be made/considered permanent from the user’s point of view. N.B. they may not be stored on a permanent medium yet**. Updates to the database may be held in caches or temporary storage buffers while awaiting transfer to the actual database on disk. COMMIT guarantees that any modifications are **or can be** written to the permanent database. Commit is part of the atomic transaction; a transaction is not successful until the commit process is complete.

**ROLLBACK** signals unsuccessful end-of-transaction. There are two events that trigger Rollback.

* A program controlled error and explicit ROLLBACK is planned and not considered a failure.
* A system detected failure aborts any suspect transactions and effects recovery. i.e. implicit ROLLBACK. Note embedded error handling.

Such problem events must cause the transaction to be aborted, and a rollback issued by the system to undo partial changes made by that transaction. These partial changes must not be visible from outside the transaction.

For a rollback all modifications made by the transaction to date must be undone; back to BEGIN TRANSACTION.

Take the following example where referential integrity requires multiple DB changes for consistency. Recall that in referential integrity, a DBMS may automatically cascade updates on the users behalf, so that a single end user operation becomes a multiple operation system transaction (or multiple related transactions) unknown for the user i.e. to delete Supplier ‘S4’, you can auto cascade a delete for any Shipments by that Supplier to keep the Database integrity correct.

Example: Shipment database with tables Suppliers(S), Parts(P) and shipments(SP)

The shipments table has structure SP( SupplierNo, PartNo, Qty).

TIME OPERATION

1 Begin Transaction

2 Delete Supplier Where Sno = ‘S4’ (\* must cascade multiple deletes to Shipment \*)

3 Commit/Rollback

NB: **The class notes may show detail on planned i.e. intended or possible transactions. It is important to understand that each operation within a transaction is online/interactive. Therefore the system does not have the full program in advance; it only deals with operations as they occur.**

**Classifying Database Systems**:

Database Systems can be classified by their transaction processing capabilities. Smaller desktop databases e.g. MS-Access, My-SQL will have different capabilities in handling transactions to larger systems such as Oracle, Ingres etc. That is, some systems may work on an operation basis with no management of logical sequences of operations as one unit; others differ in the numbers of concurrent transactions or users they can handle. Before you choose a database system you must know (analyse) your requirements and match them to the capabilities of the database system.

This course deals with Multi-user Database systems where the CPU is processing one operation at a time, but is managing many transactions concurrently. There is therefore an interleaved execution of operations from different transaction.

**Concurrency is the management and control of multiple transactions operating simultaneously on a database system**.

AC = Account\_no; Bal = Account Balance; Trans = Transaction#

Time Trans 1 Trans2 TransN

Begin Begin Begin

Insert AC2 ,0 Delete AC40 Update AC2 Bal=400

Update AC2, set Bal=200 Insert AC20 Update AC1 Bal=200

Commit Rollback/Commit? Commit/Rollback?

Below is a **timing diagram** showing the actual processing of each operation by the system(CPU). It shows one concurrent execution of the 3 named transaction. There may be others. We should understand that the sequence of operations in time could be any i.e. random.

This is an online interactive transaction processing database system. The recovery and concurrency system must handle any possible sequence of executing the given operations.

Note that the diagram’s timeline is across the horizontal; earlier we used a vertical arrangement.

Time

Trans1 Insert Update

Trans2 Delete Insert

“

“

TransN Update Update

If we introduce a failure at any time into this timing diagram, we can ask

1. Where is a safe point for the system?
2. How close to the failure can I recover to?
3. Should I recover to that point in all cases?
4. What facilities does the recovery system need?

Transactions are the unit of work and the unit of recovery and concurrency and have the following properties:

**Properties of a transaction** : the ACID properties.

**A**tomicity : all or nothing. A group or sequence taken as 1 unit.

**C**onsistency : a transaction transforms the database from one consistent state to another, intermediate inconsistency allowed.

**I**solation : transaction activities are concealed from other transactions running concurrently until that transaction commits i.e. transactions are independent. This is handled by the concurrency control programs of the DBMS.

**D**urability : once committed modifications made by a transaction survive even if there is a subsequent system crash i.e. assume that non volatile storage is secure.

**The isolation property is normally controlled in DBMS by using locks (see concurrency section).**

**Failures**:

Failures are more serious problems and are of two types:

* **System**: resulting in the entire system stopping or
* **Individual process**: e.g. one user program/process not being able to continue.

Failures include

* Hardware: e.g. Disk head crash; overheating, theft etc
* Software: the user program or the Database System stops functioning. The code may be ok but the problem is ad-hoc e.g. memory address problem. Recall any experience with programs terminating abruptly or freezing requiring Windows task manager to resolve.
* External environment: when the expected operating environment changes causing the system to fail e.g. floods, earthquake, fire, power failure, spilt cups of coffee etc.

Any database recovery system must handle both single process and full system failures.

**The objective of the recovery is to allow database activity to be resumed as soon as possible after a failure with minimum loss of information at an economically justifiable cost.**

Note: the important aspects of this are:

1. the resumption of database activity (core business activity)

2. the timeframe for recovery,

3. possibility of acceptable loss of information and

4. justifiable cost: Financial(purchase of equip e.g. RAID, UPS etc), Operational (time, effort, downtime.

Each of these should be considered but taking costs, we can expand this focusing on Hardware, Time & Effort, Security, Effect on Core Business Activity, System (Storage, CPU time)

**Recovery system planning:**

1. Recovery Time: how much time is acceptable for recovery process and how long will the database be offline?
2. Recovery Point: What point in time is recovery required to. This dictates what data is stored. This relates to the granularity of recovery. Week, Day, Hour, Transaction.
3. Cost-Benefit Analysis: Both of the factors above must be balanced with cost.

**Context**: the next step is to ask the question: in high-level terms how might a recovery system work to realise the overall objective?

**Backward Recovery**: the database is reset to some previous correct state. This may seem straight forward but there are some crucial elements to successful recovery of this type e.g. what previous state, how far back in time, what about changes that had completed since then, how about changes that were taking place at the time of the failure etc. This type of recovery is essential for any application that is subject to accounting or security audit.

So in DBS, we can look at recovery as restoring the database to a state that is correct (assumed) if the current state becomes suspect for some reason. Therefore the

**The underlying principle on which (backward) database recovery is based is data redundancy.**

In addition, we need

* to be able to determine exactly what data is lost.
* to have access to data stored redundantly, and
* a general ‘working’ procedure to recover the data.

**Storage media recovery**: There are two categories of media

**Nonvolatile storage**: secondary storage e.g. magnetic disks & tapes may fail e.g. disk head crash(i.e. R/W heads come in physical contact with the surface of the disk rather than hovering a small distance from it), disk controller failure, natural disaster etc. Part of the database is physically destroyed in this failure. **Archive/Backup copies of important information** is required. We will deal with the later in the notes.

**Volatile storage**: main memory systems may fail thereby losing current operating data, this is crucial data which has not yet been saved on a long term storage medium. The recovery system must be able to deal with this type of problem i.e. loss of data in volatile storage. **Online logs recording current activity are required**. Logs act as the place where redundant copies of data are located that can be used to recover the primary copies.

Note: System crashes due to power failure can be guarded against by using uninterruptible power supplies (UPS). These allow a time window for a controlled shutdown of the DBMS.

Power surges can cause bit errors so power surge limiters can be used to prevent the problem.

**Recovery system (backward recovery)**

a) Logs

We have noted that the system must be able to undo changes made by a transaction. We have also noted that the system must store data redundantly if it needs to reconstruct data that is lost due to some /failure. We now introduce the concept of a system log to deal with the above two points. A log is a record of all transaction activity in the system over time.

Remember actions

Time (going forward): start failure

**Write-ahead log strategy** (WALS):

The WALS states that the log is written and saved to stable 2nd secondary storage before any updates are made to the actual physical database. The actual DB insert can be executed but the WALS is not concerned whether the DB on 2nd storage is written to yet.

This is crucial as log data itself **may be** built up in buffers (or disk caches). If a failure to volatile storage occurs then this log data will be lost i.e. we would not know exactly what operations any particular transaction performed. Partial changes made by any uncommitted transactions (i.e. in progress at crash time) must be undone(transactions in log with BEGIN but no END/COMMIT). Committed transactions that are logged can be redone by the system (without the user knowing)

Recall **Buffer Management is normally asynchronous**; i.e. normally the faster CPU is allowed to work independently of slower buffer writes to disk for max efficiency.

**b)** The **fundamental operations (processes) in a recovery system are UNDO and REDO.**

UNDO: rectifies undesirable changes made by a transaction. Requires Before\_value in log

REDO: ensures that changes made by a transaction are correctly recorded by redoing the action(s). Requires the After\_value in the log.

**A log record must store data at a minimum: TransactionId, Operation, Before\_value, After\_value.**

See Appendix for Log Record Structure check in class if required for exam.

How far back in time (how far back in the log) does the recovery system go when it starts recovery. For example suppose we took a backup last weekend. How about all the transactions that have occurred since then? In a busy system, this could be millions. To handle this, we introduce 2 types of log.

an active or on-line log (2 possible structures)

There are two types of log,

an archive or off-line log. (disaster recovery)

The on-line log is only for recovery for current transactions; the Disk is ok/safe but some or all active transactions have failed.

The Archive log stores all records since the last backup (i.e. no longer needed by the on-line log)

We now need a process to manage the flow of records from the on-line to archive logs. We will call this a checkpoint.

**Checkpoints** :

A checkpoint is a point in recent time where we guarantee that any operation that is in the log is also written out to the physical database. Recovery can safely start from this point in the log/time to recovery the state of the DB to the time just before failure (or as close as possible).

Checkpoints therefore are used to limit the volume of log information that has to be processed in the event of a loss of volatile memory (i.e. including log data in buffers). Note limited redo not undo?

Checkpoints force writes any buffers (both for writes to the database and the log); so you can always come back to this point in time to start recovery in the event of failure. You know that whatever the log tells you is actually written to the database on the disk.

We can also archive any unwanted log records to the archive log.

Backups are done daily/weekly (depending on data importance), in low activity times. Checkpoints are done frequently during processing (every few minutes).

**There are three types of checkpoint**

1. Transaction-consistent checkpoints(TCC) are the most basic,
2. Action Consistent(ACC) improves on this and finally
3. Fuzzy checkpoints for best performance (in a busy online system).

1.Transaction Consistent Checkpoints(TCC). Before a TCC can start, the DB must be in a ‘no activity’ state, so existing transactions must be allowed to complete, and then the checkpoint can begin. No new transaction is allowed to start from the Checkpoint start time until it is finished. Note no recovery work is required back in time beyond the Checkpoint time). Why is that? (in class discussion). This is not good for busy on-line applications or for long transactions.

However it is a simple and effective checkpoint scheme for low volume systems.

User T1 one of many users

TCC Time Begin sharing CPU

T1 insert update Insert

T2 Update Ckp req by system

T3 wait Delete Update

Ckp req start end Failure Commit/Rollback

Ckp start

T1 is in progress; T2 and T3 must wait until CKP end before starting.

What recovery is required? Need to process only transactions that start after the CKP time.

**There are two principles in efficient checkpointing :**

**1. minimise the effect of the Checkpoint down time on the user(denial of service/time wait).**

**2. reduce the number of pages that are written to permanent storage at checkpoint time**

TCC is basic and has problems:

a. how long before existing transactions finish and

b. forced wait on new trans that would like to start.

Action Consistent Checkpoints improves focusing on point a) above; Fuzzy checkpoints further improves again focusing on point b).

2. Action consistent checkpoints ACC improve on TCC by allowing existing transactions to complete the current step/operation before checkpoint starts BUT not finish completely. Once the active transactions finish their next operation the CKP can begin to write the modified pages to the disk. No activity is allowed during the actual checkpoint procedure. ACC prevent new transactions starting from the time the checkpoint is instigated. This limits the REDO to the CKP time however you may need to UNDO past the CKP time back to the start of the DB therefore UNDO data must be retained for those transactions.

Action Consistent Ckp (ACC)

tc (checkpoint) tf (failure)

T1 [----------]

Note WALS

T2 [----------------- ----------------commit] Redo from ckp

Buffer???

T3 [-------- ------------------------------------------------- Undo all

T4 wait [--------------------------commit] ?

T5 [-------------------------- ?

Start end

No processing in ACC/ but as we’ll see processing allowed in fuzzy

NB: a shortcoming of this scheme is if there are a large number of pages to be written to the disk as part of the ckp. This will be slow and therefore this scheme may be unacceptable;

3. Fuzzy checkpoints: See Appendix for details (check in class if details are needed for exam)

**Fuzzy checkpoints concentrate on the 2nd principle of efficient checkpointing. They are programmed to manage the number of pages so that only some are written** (e.g. by tracking time and only writing a page if it has been in RAM for some time)**.** As not all pages are written out, **it means that there needs to be more work if failure occurs figuring out (by processing the log) to find what was written and what was not, but in the normal case that is infrequent. On the positive side, you gain by saving time for every checkpoint.**

See Appendix for Log data structure and management: **(check in class if required for exam)**

**Recovery Procedure (Undo/Redo)** (general recovery procedure; buffers independent, ACC)

Using the above example: at restart time the following recovery procedure is used.

1. retrieve the last checkpoint record

2. create two lists called UNDO and REDO. Set UNDO equal to the list of all transactions given in the checkpoint record.

3. search through the log from the checkpoint time forward.

4. If a BEGIN TRANSACTION log entry is found for transaction T, add T to the UNDO list.

5. If a COMMIT entry is found for transaction T, move T from UNDO to REDO.

Note : ROLLBACK and implications.

6. When the end of the log is reached both list identify the transactions that require partial changes to be undone and unpropagated modifications for committed transactions(i.e. not logged) to be redone.

In the above example T3 and T5 must be undone. T2 and T4 redone.

The system now works backward through the log, undoing the transactions in the UNDO list i.e. replace existing with previous values: note insert and delete, until the start of transaction marker is found. The system then works forward again, redoing the transactions in the REDO list (need only redo from checkpoint time forward). Note possible confusion with backward/forward recovery systems described earlier.

**Recovery is implemented by issuing compensation actions, e.g. to undo an insert operation requires the rec. manager to issue a compensation delete, for an update issue an update of the previous value. To redo you use the same operation with the new values from the log.**

Note failure during the recovery procedure itself may result in reissuing compensation operations. Note to redo an SQL ‘Update salary by 5%’. Log may need to store(have space for) every compensation record for this operation. What happens if recovery process itself fails? It may restart(possibly many times) reissuing compensation operations from the start. So logging the compensation operations is required to sort out exactly what was recovered or not.

**Idempotent (check class if this is on exam)**

The undo and redo operations for a given transaction are **idempotent** i.e. for any transaction performing one of these operations is equivalent to performing it any number of times UNDO( action ) = UNDO ( UNDO ( .......UNDO( action)....)). **This is crucial for recovery as compensation operations can build up and if a failure (re)occurs during recovery itself, then multiple instances of the compensation records can exist. You cannot delete a record that has already been deleted and attempting to do so will result in an error.**

**Alternatives to Undo/Redo**

There are other recovery schemes if the buffer manager program works differently. If it is guaranteed that all writes held until commit: therefore we know that modified data is held in RAM/buffers and not written to permanent storage so we can assume if RAM fails then effectively those changes have been discarded and so we have no Undo work to perform. This scheme is called **No undo/Redo**.

If writes are flushed directly we get **undo/no redo**. In other words we are sure that any modified page has been written to Disk so it’s safe hence no need to Redo.

Another way of thinking about this is that the system can either force the modifications out to storage

1. as they occur or
2. at end of transaction time or
3. leave it to the asynchronous buffer manager or
4. leave force writing to a system wide checkpoint.

Each of these has effects on redo/undo i.e. I/O to disk during the transaction increases the amount of recovery required and the duration of the transaction and therefore the time for possible failure; if forcing is withheld until the end of transaction and a problem occurs then recovery is less i.e. simply discard buffered modifications.

Once a checkpoint is performed we effectively can guarantee that the physical permanent database contains all the transaction modifications up to that point in time. The online log may still be required for recovery depending on the type of checkpoint, however the archive log is completely redundant( in terms of volatile mem failure) as any modifications it recorded is in the permanent DB. The archive logs may be kept for some time to facilitate **non-volatile recovery** (disaster recovery): see below.

**Archival / Backup Databases and Logs** :copies of the database or logs on stable permanent storage.

Database archives or backups must be used in the event of a loss of the physical database itself e.g. disk crash i.e. the most up to date database is gone.

* An archival database is a snapshot of the database in time.
* An archival log is a record of transaction update activity over a period of time between backups.

**What are the basic operations in recovery using Backups?**

* backup/dump and
* restore utilities.

It is crucial that both ordinary data (main working database information) and the redundant data required for reconstruction/recovery are not lost together.

Multiple storage devices and multiple copies of the database can be used to **avoid this possible catastrophic type of failure**. These backup copies may be stored off site to cater for disastrous system failures.

**Backup & Separation (use separate/different Disk Volume, Disk Drive, Office, Site, etc)**

**What mechanism is used to transfer/transmit the copy of the DB**? Manual? Electronic?

Recovery from such a failure involves reloading the database from an archive copy, and then using any existing active and archive logs to redo as many transactions that completed since the time of the backup as possible. **It is crucial that the recovery process is secured and tested.**

**When are backups taken?**

No transactions must be active at dump time (why? It must be a stable snapshot with no partial work by transactions). **Most backups occur at off-peak hours.**

**How long are backups/archives stored for**? May be subject to regulations e.g. DP, SOX.

**Frequency of backup is dictated by the application i.e. importance of the data and the cost of archiving.**

We can control frequency in two ways:

* time interval (e.g. every Sunday at 11pm) or by
* workload (how do we gauge workload? No of users, transactions, Db changes? Use the log?)

Some backup schemes dump sections of the database thereby keeping the database available for some transactions. Other schemes allow dumping of pages updated since the last backup, this is called incremental dumping. **These are important facilities in high demand applications** i.e. reduce inconvenience and denial of service to users.

**Are there any other types of backup?** So far we have been dealing with what’s called static backup copies of a database. They are snapshots of the main operating application in time. No transactions are allowed. These are only recommended in smaller applications where the system downtime incurred by the backup procedure is acceptable.

**In mission critical and highly available applications** e.g. international web ordering, **a complete duel or mirrored system** is maintained. That is each database change is carried out to each mirror. In the event of failure to the main operating system, the mirror can be automatically switched to if the main system fails. This is a real time copy of the database,

Finally, we have concentrated on the data? But programs (and meta data) are also a type of data stored in a system. How do we protect those items?

The recovery process should be checked (**tested and evaluated**) for operational correctness.

Exercise: Explore test & evaluate in more detail e.g. types of test; frequency; How to evaluate?( need to be able to measure success of process).

Note the loss of a complete building in 9/11 prompted the re-evaluation of recovery management particularly for catastrophic failure. In many cases the backups/mirrors were in the same building. But in some, the backups were safe, but all documentation, passwords etc. required to perform the restore operation were lost.

**Documentation of the recovery / continuity process is also an essential item** of data that needed to be covered in any recovery/continuity plan.

**Log management is also required for archive logs**.

The administrator should have a policy for

* How many logs are kept (both active and archive). May be subject to internal policy or external regulation!
* Where they are kept: Location/separation. Ref recovery lab.
* Logs monitoring: should be controlled/managed activity. Note monitor size and content of the log. Critical failure can be caused by log event ‘flood’.
* The recovery process should be checked (tested and evaluated) for operational correctness.
* **Naming standard or convention is critical for backup management and handling.(see lab)**

Extra note on log types:

The log described in the notes is the main log for transaction recovery. There may be other logs in the system for other purposes.

* Error Log
* Audit Log for all activity on the database, not just modifications. This could be used for security audit. It records many event types e.g. access, grants, inserts , creates etc

Note the main recovery log may not be stored in text form. The actual recovery log may be a binary log. Also a log may store the statements e.g. Update Parts where colour = ‘Blue’. But data types such as auto\_increment can cause difficultly for subsequent recovery ie. there is no user supplied data value to log!

Alternatively all records modified may need to be recorded. Why? Recall Referential Int.

Note this has implications for log size but is more exact in its control over exactly what changes were performed.

**Service level agreements**

In modern system architecture, data storage and management may be outsourced or located in a cloud. How this impacts on recovery is important.

“Service level agreements (SLAs) are fundamental to business continuity. The bottom line is that they define your minimum levels of availability from key suppliers, and often determine what actions will be taken in the event of serious disruption. Consequently, they require full consideration and attention and must be constructed extremely carefully.” http://www.disasterrecoveryworld.com/

Each service provider within an organisation whether it is an external vendor or an internal helpdesk or core application provider will develop a SLA so that the organisation/recovery team are aware of response times and level of service that should be provided if an issue arises.

Note, more and more contracted services, many SLA attached to Maintenance/Support contracts. If the machines involved are important or critical to business operation then they must be secured

Virtualisation:

Note that although **virtualisation** is a single point of failure at the physical machine level, it provides benefits for recovery in the form of

* containment of failures at OS, DBMS and Application levels.
* downtime is significantly reduced for maintenance and system admin
* downtime for recovery as the entire system configuration is now in digital form and simply treated as a critical object for protection. This complete running environment can be deployed anywhere quicker than physical install/config/data recovery. In other words the time to manually set up and config (very slow mechanical process) is replaced by a simple electronic copy and run.

**Question: why does the DBMS need any recovery procedures when there are redundancy systems at the OS/Hardware/Network levels?**

We’ll examine two possible non DBMS systems: 1. Mirroring and 2. RAID

1. **Full mirrored/replicated system:**

This is a real time replication system. Cost of a full dual system is a factor. Also, since replication is in real time, accidental and malicious data loss/damage cannot be prevented by this.

1. **RAID** is a storage system that divides and replicates data across multiple disk drives. It uses redundancy (copies) or parity checks to recover lost data. RAID is principally used to increase
2. Availability/reliability, by maintaining data redundancy in case of failure
3. Read/Write performance. By spreading/striping the data across volumes, devices, controllers parallelism in I/O is possible.

**We need to be aware of**

1. What type of access is expected? Read/Write; Random/Sequenced?

2. Hardware or Software RAID implementation.

This leads us to the use of performance Benchmarks:

The benchmark for database RAID includes the following variables:

• Different types of query (select, update, insert, delete); random and sequenced.

• Different amounts of data to be written (small records, big records, group updates); and

• Writing to different sizes of file.

Any benchmark test must examine all combinations of these variables.

Recovery log are processed in sequential read/write. Therefore a sequential data organisation on the disk would seem beneficial. However striping across volumes/disks forces disk head seeks (not good); in addition a full mirror is ok so long as you do not mix multiple log files on it as this again will force disk seeks to read/write the different logs.

For general shared data files with multiple users, some reading, some writing to the database; it would seem beneficial to split the database up into many sub sections to allow concurrent and varied activity on each section. Striping with mirroring on a multiple disks is a method of dividing the database up into sections; (increased concurrent reads and writes). In addition, having a mirrored system allows a choice of read locations; system can choose the fastest; again resulting in increased performance and throughput. But be careful, more replication means more update writes for each copy.

The exact optimum RAID set up may be specific to a particular application; i.e. amount of data; pattern of activity (read/write mix and frequency); amount of data written etc. This being the case, rigorous testing (benchmarking) of your configuration would be highly recommended.

Problems with RAID:

Controller failure

Multiple Drive failure

Not a solution to prevent malicious damage, natural disaster

Cost

Exercise: review N-Tier Enterprise Architecture in light of Recovery requirements (note cloud computing, virtualisation, SLAs, Dual/Mirrors, RAID)

Sample Questions

1. What is meant by reliability?
2. Question: Differentiate using examples between the terms error and failure in the context of database recovery.
3. What is a backward recovery scheme, and in what applications can it be used?
4. State the objective of a recovery system? Briefly discuss the main factors influencing the recovery scheme?
5. Why are transactions the unit of recovery / concurrency and not individual user operations?
6. Write a note on the properties of a transaction?
7. What is the WALS and why is it used.
8. What are checkpoints.
9. What is meant by archive/backup database management?
10. What is meant by archive/backup log management?
11. Explain the concepts of Undo/Redo recovery.
12. Non-volatile memory failure requires archival databases and archival logs. Explain.
13. Besides the transaction recovery log, what other logs are implemented by DBMS? What is a binary log?
14. Explain in detail the use of RAID in recovery (for) a) database data; b) control data; c) log data.
15. Discuss Recovery requirements impact on an N-Tier system architecture
16. Devise/Outline a recovery plan (Business Continuity plan)
17. Explain using an example how the use of transactions complicates the recovery system.
18. Explain using an example how the use of SLAs impacts on the recovery process.
19. Explain how the use of Virtualisation impacts on Recovery/
20. Explain how the use of RAID impacts on Recovery/ Concurrency
21. Storing the query texts in the security log may be sufficient however it is not enough for transaction recovery? Explain.

Appendix 1

**Log record structure/format**: For each transaction the following is kept (at least):

1 Log Sequence Number (LSN)

2 transaction identifier ( multi-user environment)

3. record occurrence involved (possibly tuple or row identifier )

4. operation performed ( Begin trans, insert, update, delete, Rollback, Commit )

5. previous value(s) of the data modified(update,delete); required for undoing changes made by a partially completed transaction. This is called the **before image**.

6. updated values of data modified(update,insert); if failure occurs after a transaction commits then changes may have to be redone. This is called the **after image**.

We can **logically** (NB) think of the log as a normal relational table made up of records of the above structure. There is one record in the table for each operation performed by each transaction. As we are working in a multi-user system, each transaction will be using the same log (NB see log management). Note MS-Word Redo/Undo for action history.

An SQL statement like Select \* from Logtable Might return

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LSN | Trans-ID | Tuple-id | Operation | Before-I | After-I | Timestamp | Prev-transptr |
| 1010 | T45 |  | Begin |  |  | 12:20:Apr-6 |  |
| 1011 | T46 |  | Begin |  |  |  |  |
| 1012 | T46 | 12345 | Delete | S4 |  |  |  |
| 1013 | T45 | 23456 | Update | Red | Blue |  |  |
| 1014 | T46 |  | Commit |  |  |  |  |
| 1015 | T45 |  | Rollback |  |  |  |  |

The Log is a crucial point of failure (duplexed/ multiple copies/) and an important factor in the overall efficiency of the system (bottleneck).

Note : Log tables can easily be processed using SQL e.g.

Select \* from Logtable

where transid = ‘T45’

Select max(lsn) From Logtable or Select min(lsn) From Logtable instead of \* to get last or first log record for transid 45.

**A before /after image may be an entire record (to undo or redo an insert or delete)**

**Log structure & Log management:**

**Context:** Does the log ever stop growing? How big(byte size) is the log? Do we record log events from system start-up to when…forever? Must record forward (i.e. into the future) for the lifetime of the application, but do you need all of the past? Is there a point reached where you no longer need log records? Do we wait until the changes to the database are saved on the Database out on disk? This is a possibility assuming we use the Durability property of transactions and we can assume that the DB is safe (or recoverable). So when is a change saved? Is it when buffer is cleared? Is single/individual operation save enough? What does the log need to have available (stored) to recover a transaction?

When do you process the log? Different types of processing for different types of failure? The processing requirements will dictate a suitable data structure for the log.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

There are two types of log, an active or on-line log (2 possible structures) and an archive or off-line log. (disaster recovery)

The on-line log (for current transaction failure) is usually held on direct access storage device e.g. a disk for 1) speed: low recovery time, and for 2) selective addressing e.g. an individual current transaction may fail and require rollback, we need access to **only those log records pertaining to that single transaction**. This can be done by sequential search but more usually by pointer chains(speed) We introduced the log structure as a table. But in reality a table cannot implement pointers so we need to really see the log as a file out on disk that can implement Prev\_trans\_ptr. Note the internal structure of each log record is described earlier.

LOG ………….Time

Prev\_trans\_ptr

Log size: log stored in a file, can’t have infinite file size; no use of file extends. How can we implement? When is it safe to discard a log record i.e. not required any longer?

**Log Management:** in case of failure, active transactions **must** be capable of being rolled back without human intervention. This means that the automatic recovery system cannot have to manually load/unload archive log or switching from one active log to another. Therefore all log records for all active transactions must be in **a single** on-line log (NB). When the current on-line log is full, the log manager switches to another active log, and dumps the first to archive. Thus the second active log should be opened as soon as the first reaches a set threshold e.g. 80%. Existing transactions use the first log, new transactions use the second. If all transactions using the first log finish then it can be archived, if log overflow occurs the remaining active first log transactions must be aborted (system rollback). The off-line or archive log can be stored on tape(any cheap, high capacity system; direct access not required).

NB, archive log required for disaster recovery. See later section on archive databases/logs.

**Circular log. Alternate log data structure.(used in Ingres) Check in class if this is on exam.**

Front : must not catch up with rear/end of the queue.

Thd1 Thd1 = Threshold 1, stop all new, leave space for active to finish

Thd2 Thd2 = threshold 2, critical, rollback offending long transaction

Rear with room for concurrent to finish + compensation log recs

Rear = Min LSN of all active/recent? Trans (furthest back in time needed) Note Similar to dual log we need some mechanism to archive the data and safely move rear forward in the log toward Front to free up space in the online log.

**Long transactions cause problems for logs, why? Lazy transaction V’s high activity over time transaction. It may not be the transactions fault that it’s long; what approaches could the system take to handle long transactions?**

Note2. time from Thd1 to Rear = undesirable wait for new transactions.

Archiving a circular log works with moving rear forward, you archive each record you pass by until you reach Min(LSN) of active trans? Is this enough? The fact that a transaction has committed, i.e. has a commit record log entry, does not guarantee that the modifications made by that transaction have reached the DB (we may need to Redo). Unless we know DB buffers have been flushed (or in no Redo system) use Min LSN of all active transactions at the last checkpoint as the cut-off for what is required.

Simple variation: checkpoint first before you archive the log. (see next section)

Threshold 2 (Thd2) allows existing trans run concurrently with a system rollback of offending long transaction in the hope that this will free up space.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ End of Circular log \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Appendix : Fuzzy checkpoint

Fuzzy CKP attempt to improve efficiency by reducing the number of pages written as part of the checkpoint. Obviously if you have a certain number, say M, modified pages that would normally be flushed to disk as part of an ACC. In this case however, you are only going to write only some of that set, say M-?, then we need some criterion for deciding on which ones we write and those we do not.

Not writing a modified page as part of a CKP means that recovery must now go back in time before the CKP. (for modified pages not written to safety on the disk). So this is a trade off, you will do less work during checkpoint but a little more recovery (more Redo) in case of failure. As checkpoint is frequent and recovery is not, then you save.

A common Fuzzy Ckp implementation is to view modified pages over a two ckp interval (rather than just one large set as in ACC) and to only write a page as part of the CKP if the modified page has been found in the buffer on TWO successive CKP. This condition reduces the number of pages to be written e.g. as some of the pages have only just been read in and modified and so have not been in the buffer long enough to qualify as part as the CKP write.

Some people remember this as old modified pages are written, newly modified pages are not.

Fuzzy checkpoints flush changes to permanent storage in parallel with normal processing activity (no system/application service stop). The full set of checkpointed pages may not be mutually consistent in a specific time (as updates can continue to be processed as normal, one page of the set can be modified while another is being flushed). The important point is that each page is consistent at the time it is processed as part of the checkpoint. The WALS ensures that the mutual inconsistency can be resolved at restart i.e. by redoing all operations in the log generated during the checkpoint interval the set of pages are converted into a ‘sharp’ picture of the DB. Note idempotence.

TCC : no concurrent trans + Ckp

ACC : concurrent trans but no concurrent actions(ins,update)

Fuzzy: concurrent actions on difference pages of the buffer as ckp proceeds

CPU fast processing (allow T4 process while T1 pages are checkpointed/sequential write to disk).

\_\_\_\_\_\_\_\_\_

Trans4 \_\_\_\_\_\_\_\_\_ DB on Disk

\_\_\_\_5\_\_\_\_\_

\_\_\_4\_\_\_\_\_\_

3 slow ckp output to disk

\_2\_\_\_\_\_\_\_\_

1\_\_\_\_\_\_\_\_\_

Trans1 write page 1

Cache/Buffer pages

Note that there is no question of correctness as there is no concurrent page activity, i.e. DB is page consistent but not necessarily the entire DB consistent at or during the Ckp time.

**Fuzzy Ckp : Min pages written :** P = page (**not** tuple or value), W = write, R = Read

Ckp1 Ckp2 Failure Redo

Insert P1 Commit

T1 REDO

Insert Delete

T2 UNDO

Update P3 buffer write

T3

Read P2 Update P2

T4

P1 x P1 x

P2 P2 x

P3 x buffer write P3

In ACC all pages (P1,P2) are force written. In fuzzy only P1 at ckp2.

In the event of failure and recovery, we must extend the ACC recovery logic to work back past the last CKP BUT it is guaranteed to be limited to two CKP intervals.

Some people confuse the word ‘limit’ in this context. In ACC you only redo from last ckp forward; so there is possibly more redo in fuzzy but limited to 2 ckp intervals back in time i.e. in fuzzy ckp you may need to redo past the last ckp but never beyond the second last ckp.

**Log Sequence Number LSN**  (reference only not on exam)

The Log Sequence Number (LSN) can be used by the recovery procedure to implement idempotence. LSNs are allocated in order. If the LSN is also stored on the DB page, then we can compare the LSN in the log record against the LSN on the DB page during the recovery procedure to determine if that log operation had actually reached the DB before failure. Note that this technique reduces the amount of recovery work. Don’t undo if original operation was lost in buffers, don’t redo if original operation reached permanent DB.

Do we apply all log records in recovery(direct write or overwrite, no DB read, fast but worst case i.e. assumes all recovery work required but note if buffers were in fact written after ckp then no redo is required and if no buffers written then there is no need for undo).

An alternative is to read DB, check state of the data by examining the Max(LSN) and try minimise redo? One read of a DB page header may save on multiple writes to different tuples within the page. LSN’s are used in latter case.

LSNs can be used to gauge how much online log data is required. The log manager needs to store the minimum LSN of all the active transactions in the log at the last checkpoint in Un/Redo or Min(LSN) of active if buffer force-written at commit.

2nd Storage failure recovery needs backup + archive logs (possibly lose latest online). Last LSN on backup = minimum LSNin archive log. System can delete the rest of the archive log.

LSNs are also usually made up of a file# and relative byte address(rba) so they can be used as the pointers in the log records (previous\_rec and prev\_trans\_rec).

<http://dev.mysql.com/doc/refman/5.0/en/innodb-checkpoints.html>

**InnoDB Checkpoints**

**InnoDB** implements a checkpoint mechanism known as “fuzzy” checkpointing. **InnoDB** flushes modified database pages from the buffer pool in small batches. There is no need to flush the buffer pool in one single batch, which would in practice stop processing of user SQL statements during the checkpointing process.

During crash recovery, **InnoDB** looks for a checkpoint label written to the log files. It knows that all modifications to the database before the label are present in the disk image of the database. Then **InnoDB** scans the log files forward from the checkpoint, applying the logged modifications to the database.

**InnoDB** writes to its log files on a rotating basis. It also writes checkpoint information to the first log file at each checkpoint. All committed modifications that make the database pages in the buffer pool different from the images on disk must be available in the log files in case **InnoDB** has to do a recovery. This means that when **InnoDB**starts to reuse a log file, it has to make sure that the database page images on disk contain the modifications logged in the log file that **InnoDB** is going to reuse. In other words, **InnoDB** must create a checkpoint and this often involves flushing of modified database pages to disk.

The preceding description explains why making your log files very large may reduce disk I/O in checkpointing. It often makes sense to set the total size of the log files as large as the buffer pool or even larger. The disadvantage of using large log files is that crash recovery can take longer because there is more logged information to apply to the database.