WEEK 11

Database Management Systems

- A **Backup** of a database is a representative copy of data containing all necessary contents of a database such as data files and control files
  - Unexpected database failures, especially those due to factors beyond our control, are unavoidable. Hence, it is important to keep a backup of the entire database
  - There are two major types of backup:
    - *Physical Backup*: A copy of physical database files such as data, control files, log files, and archived redo logs.
    - *Logical Backup*: A copy of logical data that is extracted from a database consisting of tables, procedures, views, functions, etc.
- **Recovery** is the process of restoring the database to its latest known consistent state after a system failure occurs.
  - A *Database Log* records all transactions in a sequence. Recovery using logs is quite popular in databases
  - A typical log file contains information about transactions to execute, transaction states, and modified values

Types of Backup Data

- **Business Data** includes personal information of clients, employees, contractors etc. along with details about places, things, events and rules related to the business.
- **System Data** includes specific environment/configuration of the system used for specialised development purposes, log files, software dependency data, disk images.
- **Media** files like photographs, videos, sounds, graphics etc. need backing up. Media files are typically much larger in size.

Types of Backup Strategies

- **Full Backup** backs up everything. This is a complete copy, which stores all the objects of the database: tables, procedures, functions, views, indexes etc. Full backup can restore all components of the database system as it was at the time of crash.
- *A full backup must be done at least once before any of the other type of backup*
- The frequency of a full backup depends on the type of application. For instance, a full backup is done on a **daily basis** for applications in which one or more of the following is/are true:
  - Either 24/7 availability is not a requirement, or system availability is not affected as a consequence of backups.
  - A complete backup takes a minimum amount of media, i.e. the backup data is not too large.
  - Backup/system administrators may not be available on a daily basis, and therefore a primary goal is to reduce to a bare minimum the amount of media required to complete a restore.

- **Incremental** backup targets only those files or items that have changed since the last backup. This often results in smaller backups and needs shorter duration to complete the backup process.
- For instance, a 2 TB database may only have a 5% change during the day. With incremental database backups, the amount backed up is typically only a little more than the actual amount of **changed data** in the database.
- For most organizations, **a full backup is done once a week, and incremental backups are done for the rest of the time.** This might mean a backup schedule as shown below

Friday	*Saturday*	*Sunday*	*Monday*	*Tuesday*	*Wednesday*	*Thursday*
Full	Incremental	Incremental	Incremental	Incremental	Incremental	Incremental

- *This ensures a minimum backup window during peak activity times, with a longer backup window during non-peak activity times.*

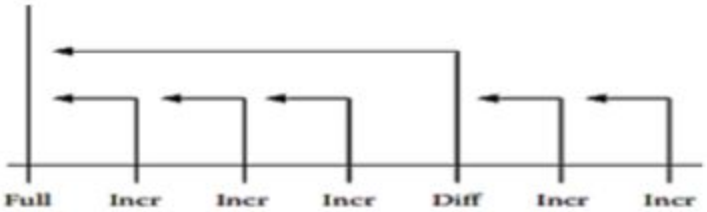
- **Differential** backup backs up all the changes that have occurred since the most recent full backup regardless of what backups have occurred in between
- This "rolls up" multiple changes into a single backup job which sets the basis for the next incremental backup
  - As a differential backup does not back up everything, this backup process usually runs quicker than a full backup
  - The longer the age of a differential backup, the larger the size of its backup window

- To evaluate how differential backups might work within an environment, consider the sample backup schedule shown in the figure below.

Friday	*Saturday*	*Sunday*	*Monday*	*Tuesday*	*Wednesday*	*Thursday*
Full	Incremental	Incremental	Incremental	Differential	Incremental	Incremental

a) The incremental backup on Saturday backs up all files that have changed since the full backup on Friday. Likewise all changes since Saturday and Sunday is backed up on Sunday and Monday's incremental backup respectively.
b) On Tuesday, a differential backup is performed. This backs up all files that have changed since the full backup on Friday. A recovery on Wednesday should only require data from the full and differential backups, **skipping the Saturday/Sunday/Monday incremental backups**.

Recovery on any given day only needs the data from the full backup and the most recent differential backup



- The figure below depicts which of the updated files of the database will be backed up in each respective type of backup throughout a span of 5 days as indicated.

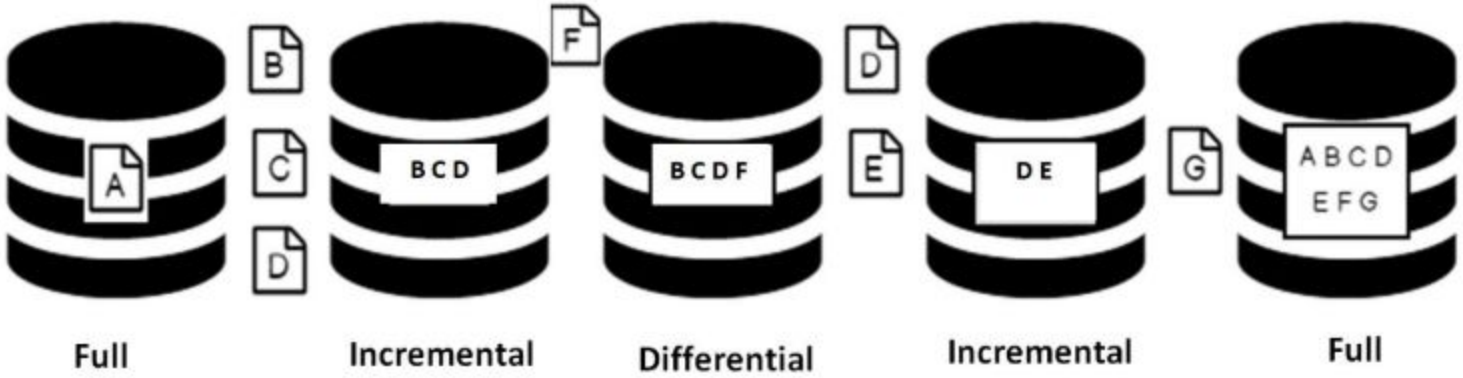


Figure: Backup Types

Consider the following backup schedule for a month:

Sunday	*Monday*	*Tuesday*	*Wednesday*	*Thursday*	*Friday*	*Saturday*
1/Full	2/Incr	3/Incr	4/Incr	5/Incr	6/Incr	7/Incr
8/Diff	9/Incr	10/Incr	11/Incr	12/Incr	13/Incr	14/Incr
15/Diff	16/Incr	17/Incr	18/Incr	19/Incr	20/Incr	21/Incr
22/Diff	23/Incr	24/Incr	25/Incr	26/Incr	27/Incr	28/Incr
29/Diff	30/Incr	31/Incr				

- **Inference**
  - Here full backups are performed once per month, but with differentials being performed weekly, the maximum number of backups required for a **complete system recovery** at any point will be **one full backup, one differential backup, and six incremental backups**
  - A full system recovery will never need more than the full backup from the start of the month, the differential backup at the start of the relevant week, and the incremental backups performed during the week
  - If a policy were used whereby **full backups were done on the first of the month, and incrementals for the rest of the month**, a complete system recovery on last day of month will need as many as **31 backup sets**
  - Thus differential backups can improve efficiency of recovery when planned properly

Hot Backup

- Till now we have learnt about backup strategies which can not happen simultaneously with a running application
- In systems where high availability is a requirement **Hot backup** is preferable **wherever possible**
- **Hot backup** refers to keeping a database up and running while the backup is performed concurrently
  - Such a system usually has a module or plug-in that allows the database to be backed up while staying available to end users
  - Databases which stores transactions of asset management companies, hedge funds, high frequency trading companies etc. try to implement Hot backups as these data are highly dynamic and the operations run 24x7
  - Real time systems like sensor and actuator data in embedded devices, satellite transmissions etc. also use Hot backup

- In regular database systems, hot backup is mainly used for Transaction Log Backup.
- **Cold backup** strategies **like Differential, Incremental** are preferred for Data backup. The reason is evident from the disadvantages of Hot backup.
- **Transactional Logging** is used in circumstances where a possibly inconsistent backup is taken, but another file generated and backed up (after the database file has been fully backed up) can be used to restore consistency.
- The information regarding **data backup versions** while recovery at a **given point** can be inferred from the Transactional Log backup set.
- Thus they play a vital role in **database recovery.**

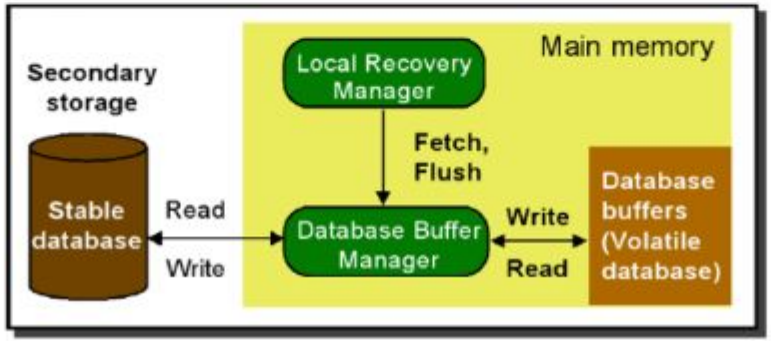
- **Transaction failure**:
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (for example, deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted as result of a system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable
    - Disk drives use checksums to detect failures

- Consider transaction T_i that transfers \$50 from account A to account B
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database
  - A failure may occur after one of these modifications have been made but before both of them are made
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  a) Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  b) Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

- **Volatile Storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile Storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable Storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct non-volatile media

Stable Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding
- Failure during data transfer can still result in inconsistent copies. Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    - ▷ Write the information onto the 1^{st} physical block
    - ▷ When the 1^{st} write is successful, write the same information onto the 2^{nd} physical block
    - ▷ The output is completed only after the second write successfully completes

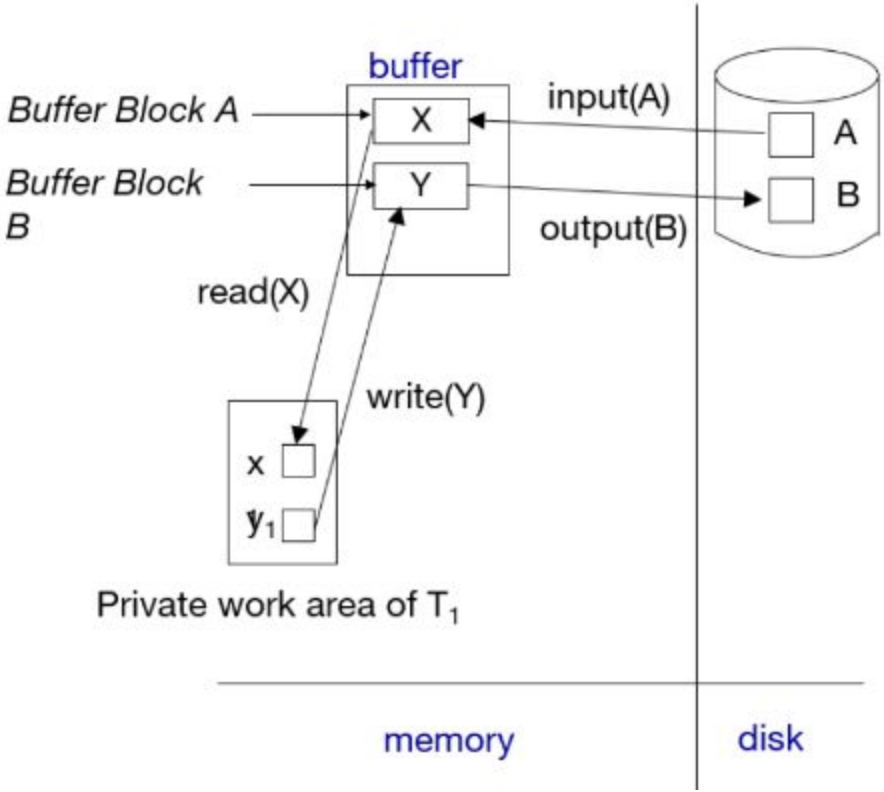


Protecting storage media from failure during data transfer (cont.):

- Copies of a block may differ due to failure during output operation
- To recover from failure:
  - First find inconsistent blocks:
    - *Expensive solution* : Compare the two copies of every disk block
    - *Better solution*:
      - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk)
      - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these
      - Used in hardware RAID systems
  - If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy
  - If both have no error, but are different, overwrite the second block by the first block

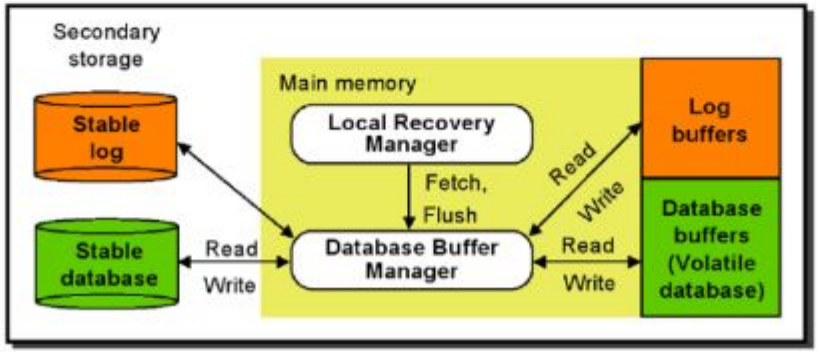
- **Physical Blocks** are those blocks residing on the disk
- **System Buffer Blocks** are the blocks residing temporarily in main memory
- Block movements between disk and main memory are initiated through the following two operations:
  - **input(B)** transfers the physical block B to main memory
  - **output(B)** transfers the buffer block B to the disk, and replaces the appropriate physical block there
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept
  - T_i's local copy of a data item X is denoted by x_i
  - B_X denotes block containing X
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read(X)** assigns the value of data item X to the local variable x_i
  - **write(X)** assigns the value of local variable x_i to data item X in the buffer block
- Transactions
  - Must perform **read(X)** before accessing X for the first time (subsequent reads can be from local copy)
  - The **write(X)** can be executed at any time before the transaction commits
- Note that **output(B_X)** need not immediately follow **write(X)**. System can perform the **output** operation when it deems fit


buffer
Buffer Block A
Buffer Block B
X
Y
input(A)
output(B)
A
B
read(X)
write(Y)
x
y_1
Private work area of T_1
memory
disk

Log Based Recovery

- A **log** is kept on stable storage
  - The log is a sequence of **log records**, which maintains information about update activities on the database
- When transaction Ti starts, it registers itself by writing a record $< T_i$ **start** $>$ to the log
- Before Ti executes **write(X)**, a log record $< T_i, X, V_1, V_2 >$ is written, where V_1 is the value of X before the write (**old value**), and V_2 is the value to be written to X (**new value**)
- When T_i finishes its last statement, the log record $< T_i$ **commit** $>$ is written



Database Modification Schemes

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
  - Update log record must be written *before* a database item is written
    - We assume that the log record is output directly to stable storage
  - Output of updated blocks to disk storage can take place at any time before or after transaction commit
  - Order in which blocks are output can be different from the order in which they are written
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy
- We cover here only the immediate-modification scheme

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate Modification Scheme Example

Log	Write	Output
$<T_0$ **start**$>$		
$<T_0$, A, 1000, 950$>$		
$<T_0$, B, 2000, 2050$>$		
	$A = 950$	
	$B = 2050$	
$<T_0$ **commit**$>$		
$<T_1$ **start**$>$		
$<T_1$, C, 700, 600$>$		
	$C = 600$	
		B_B , B_C
$<T_1$ **commit**$>$		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X

- **Undo** of a log record $< T_i, X, V_1, V_2 >$ writes the **old** value V_1 to X
- **Redo** of a log record $< T_i, X, V_1, V_2 >$ writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - ▷ Each time a data item X is restored to its old value V a special log record (called **redo-only**) $< T_i, X, V >$ is written out
    - ▷ When undo of a transaction is complete, a log record $< T_i$ **abort**$>$ is written out (to indicate that the undo was completed)
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - ▷ No logging is done in this case

- Let T_i be the transaction to be rolled back
- Scan log backwards from the end, and for each log record of T_i of the form $< T_i, X_j, V_1, V_2 >$
  - Perform the undo by writing V_1 to X_j,
  - Write a log record $< T_i, X_j, V_1 >$
    - such log records are called **Compensation Log Records**
- Once the record $< T_i$ **start**$>$ is found stop the scan and write the log record $< T_i$ **abort**$>$

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record $< T_i$ **start**$>$,
    - but does not contain either the record $< T_i$ **commit**$>$ or $< T_i$ **abort**$>$
  - Transaction T_i needs to be redone if the log
    - contains the records $< T_i$ **start**$>$
    - and contains the record $< T_i$ **commit** $>$ or $< T_i$ **abort** $>$
  - It may seem strange to redo transaction T_i if the record $< T_i$ **abort**$>$ record is in the log
    - To see why this works, note that if $< T_i$ **abort**$>$ is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo T_i 's modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time
    - such a redo redoes all the original actions including the steps that restored old value – Known as **Repeating History**

Immediate Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records $< T_0$, B, 2000 $>$, $< T_0$, A, 1000 $>$, $< T_0$, **abort**$>$ are written out

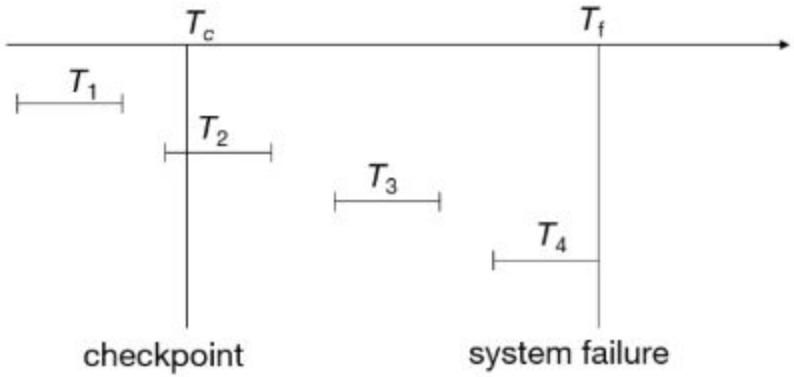
(b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records $< T_1$, C, 700 $>$, $< T_1$, **abort**$>$ are written out

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database
- Streamline recovery procedure by periodically performing **checkpointing**
- All updates are stopped while doing checkpointing
  a) Output all log records currently residing in main memory onto stable storage
  b) Output all modified buffer blocks to the disk
  c) Write a log record $<$ **checkpoint** $L >$ onto stable storage where L is a list of all transactions active at the time of checkpoint

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i
  - Scan backwards from end of log to find the most recent $<$**checkpoint** $L>$ record
  - Only transactions that are in L or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage



- Any transactions that committed before the last checkpoint should be ignored
  - T_1 can be ignored (updates already output to disk due to checkpoint)
- Any transactions that committed since the last checkpoint need to be redone
  - T_2 and T_3 redone
- Any transaction that was running at the time of failure needs to be undone and restarted
  - T_4 undone