DATABASE MANMAGEMENT SYSTEMS : NOTES W11

What is Backup and Recovery?

- 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 backups:
    - 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

Why is backup necessary?

- **Disaster Recovery**
  - Data loss can occur due to various reasons like hardware failures, malware attacks, environmental & physical factors or a simple human error
- **Client-Side Changes**
  - Clients may want to modify the existing application to serve their business's dynamic needs
  - Developers might need to restore a previous version of the database in order to such address such requirements
- **Auditing**
  - From an auditing perspective, you need to know what your data or schema looked like at some point in the past
  - For instance, if your organization happens to get involved in a lawsuit, it may want to have a look at an earlier snapshot of the database.
- **Downtime**
  - Without backup, system failures lead to data loss, which in turn results in application downtime
  - This leads to bad business reputation

Backup Data: 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.

Backup Strategies

Types of Backup Strategies: Full Backup

- **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.
- **Full Backup: Advantages**
    - Recovery from a full backup involves a consolidated read from a single backup
    - Generally, there will not be any dependency between two consecutive backups.
    - Effectively, the loss of a single day's backup does not affect the ability to recover other backups
    - It is relatively easy to setup, configure and maintain
- **Full Backup: Disadvantages**
    - The backup takes largest amount of time among all types of backups
    - This results in longest system downtime during the backup process
    - It uses largest amount of storage media per backup

Types of Backup Strategies: Incremental Backup

- **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.**
- **Incremental Backup: Advantages**
    - Less storage is used per backup
    - The downtime due to backup is minimized
    - It provides considerable cost reductions over full backups
- **Incremental Backup: Disadvantages**
    - It requires more effort and time during recovery

- A complete system recovery needs a full backup to start with
- It cannot be done without the full backups and all incremental backups in between
- If any of the intermediate incremental backups are lost, then the recovery cannot be 100%

Types of Backup Strategies: Differential Backup

- **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

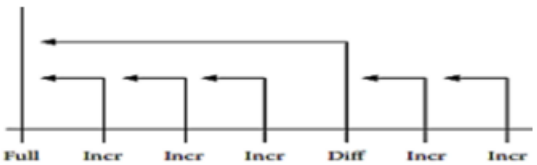
- To evaluate how differential backups might work within an environment, consider the sample backup schedule shown in the figure below.

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



- **Differential Backup: Advantages**
  - Recoveries require fewer backup sets.
  - Provide better recovery options when full backups are run rarely (for example, only monthly)
- **Differential Backup: Disadvantages**
  - Although the number of backup sets required for recovery is less but in differential backups the amount of storage media required may exceed the storage media required for incremental backups
  - If done after quite a long time, differential backups can even reach the size of a full backup

Types of Backup Strategies: Illustrative Example

- 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.

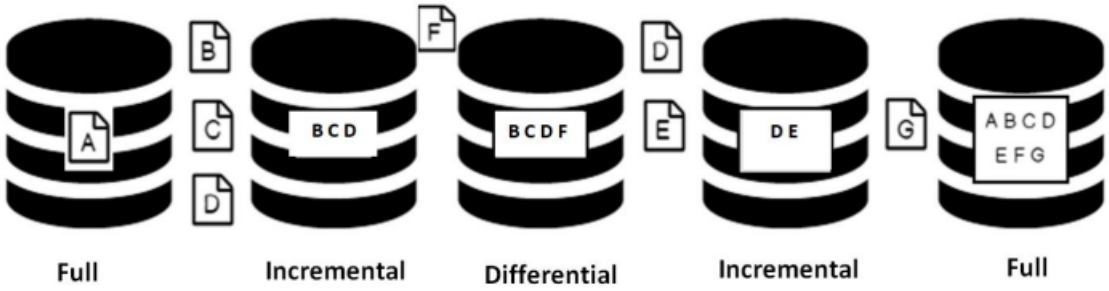


Figure: Backup Types

Case: Monthly Data Backup Schedule

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 cannot 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

- **Hot Backup: Advantages**
  - The database is always available to the end user.
  - Point-in-time recovery is easier to achieve in Hot backup systems.
  - Most efficient while dealing with dynamic and modularized data.

- **Hot Backup: Disadvantages**
  - May not be feasible when the data set is huge and monolithic.
  - Fault tolerance is less. Occurrence of any error on the fly can terminate the whole backup process.
  - Maintenance and setup cost is high

Transactional Logging as 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.**

Failure Classification

Database System Recovery

- All database reads/writes are within a transaction
- Transactions have the "ACID" properties
  - Atomicity - all or nothing
  - Consistency - preserves database integrity
  - Isolation - execute as if they were run alone
  - Durability - results are not lost by a failure
- Concurrency Control guarantees I, contributes to C
- Application program guarantees C
- Recovery subsystem guarantees A & D, contributes to C

Failure Classification

- **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

Recovery Algorithms

- Consider transaction $\mathbf{T_i}$ that transfers $50 from account A to account B

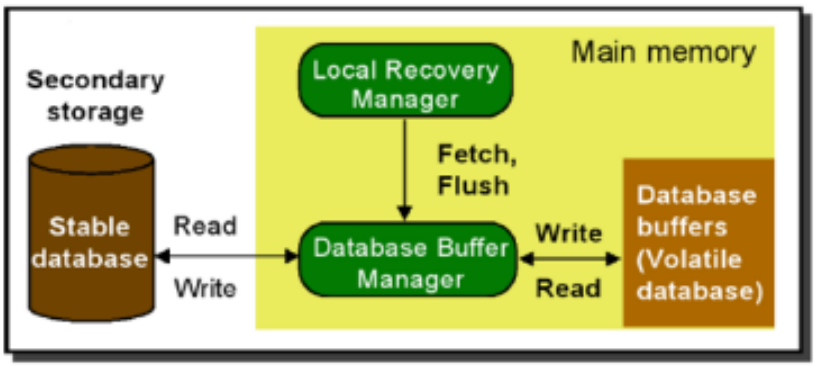
    - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database
    - A failure may occur after one of these modifications have been made but before both of them are made
    - Modifying the database without ensuring that the transaction will 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

Storage Structure

- **Volatile Storage:**
    - does not survive system crashes
    - examples: main memory, cache memory
- **Non-volatile 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 1st physical block
        - When the 1st write is successful, write the same information onto the 2nd physical block
        - The output is completed only after the second write successfully completes

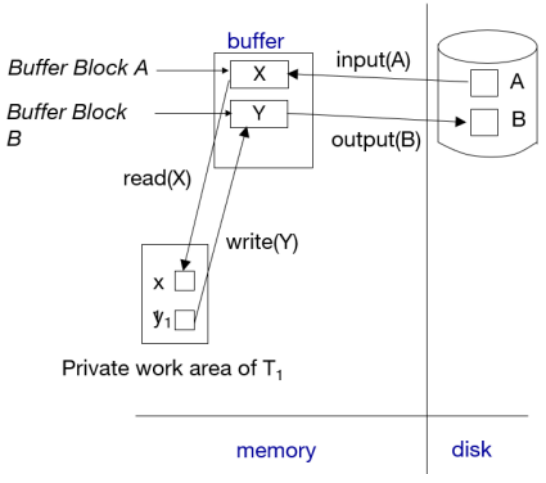


Protecting storage media from failure during data transfer (cont.):

- 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

Data Access

- **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 $\mathbf{T_i}$ has its private work-area in which local copies of all data items accessed and updated by it are kept
  - $\mathbf{T_i}$'s local copy of a data item X is denoted by xi
  - BX 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 xi
  - **write(X)** assigns the value of local variable xi 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

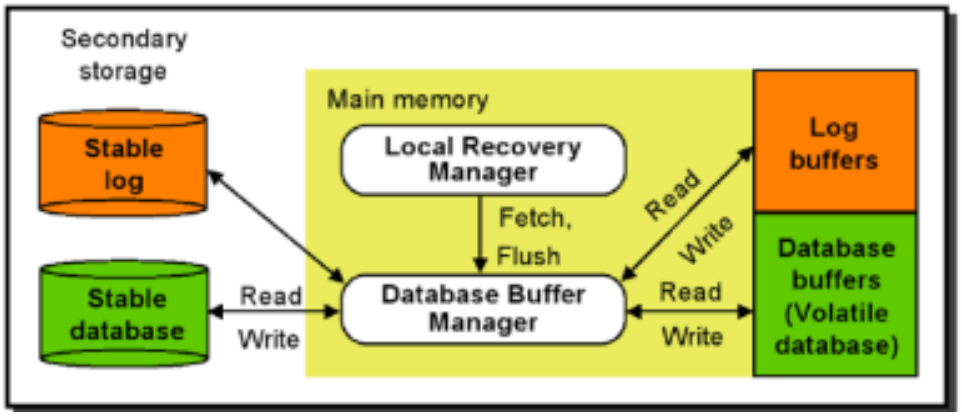


Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself
- We study **Log-based Recovery Mechanisms**
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **Shadow Paging**
- In this Module we assume serial execution of transactions
- In the next Module, we consider the case of concurrent transaction execution

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 $\mathbf{T_i}$ starts, it registers itself by writing a record < $\mathbf{T_i}$ **start** > to the log
- Before $\mathbf{T_i}$ executes **write(X),** a log record < Ti , X, V1, V2 > is written, where V1 is the value of X before the write **(old value)**, and V2 is the value to be written to X (**new value**)
- When Ti finishes its last statement, the log record < $\mathbf{T_i}$ **commit** > is written



Database Modification Schemes

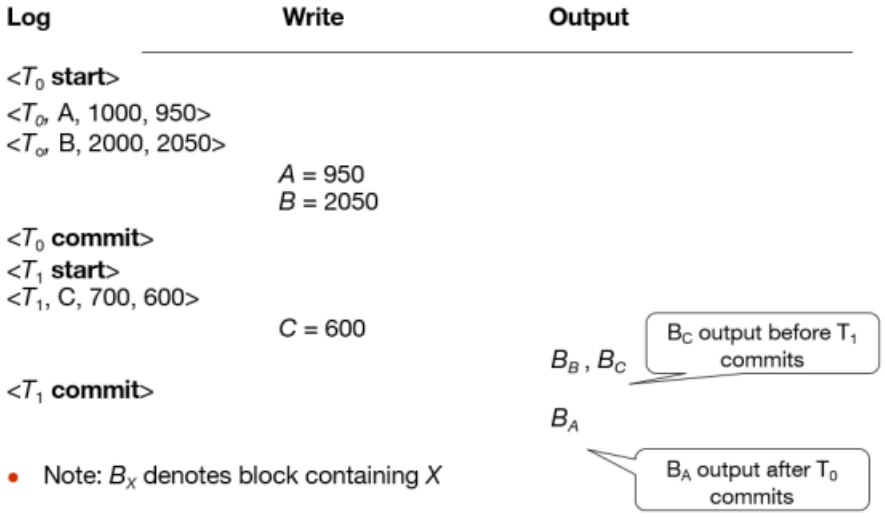
- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
  - Update log record must be written before a database item is written
    - We assume that the log record is output directly to stable storage
  - 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

Transaction Commit

- 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 Database Modification Example



Undo and Redo Operations

- **Undo** of a log record < Ti , X, V1, V2 > writes the **old** value V1 to X
- **Redo** of a log record < Ti , X, V1, V2 > writes the **new** value V2 to X
- **Undo and Redo of Transactions**
  - **undo**(**Ti**) restores the value of all data items updated by **Ti** to their old values, going backwards from the last log record for **Ti**
    - Each time a data item X is restored to its old value V a special log record (called **redo-only**) < Ti , X, V > is written out
    - When undo of a transaction is complete, a log record < Ti **abort**> is written out (to indicate that the undo was completed)
  - **redo**(**Ti**) sets the value of all data items updated by **Ti** to the new values, going forward from the first log record for **Ti**
    - No logging is done in this case

- The **undo** and **redo** operations are used in several different circumstances:
  - The undo is used for transaction rollback during normal operation
    - in case a transaction cannot complete its execution due to some logical error
  - The **undo** and **redo** operations are used during recovery from failure
- We need to deal with the case where during recovery from failure another failure occurs prior to the system having fully recovered

Undo and Redo on Normal Transaction Rollback

- Let Ti be the transaction to be rolled back
- Scan log backwards from the end, and for each log record of Ti of the form < Ti , Xj , V1, V2 >
  - Perform the undo by writing V1 to Xj ,
  - Write a log record < Ti , Xj , V1 >
    - such log records are called **Compensation Log Records**

- Once the record < Ti **start**> is found stop the scan and write the log record < Ti **abort**>

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction Ti needs to be undone if the log
    - contains the record < Ti **start**>,
    - but does not contain either the record < Ti **commit**> or < Ti **abort**>
  - Transaction Ti needs to be redone if the log
    - contains the records < Ti **start**>
    - and contains the record < Ti **commit** > or < Ti **abort** >

  - It may seem strange to redo transaction Ti if the record < Ti abort> record is in the log
    - To see why this works, note that if < Ti **abort**> is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo Ti '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>

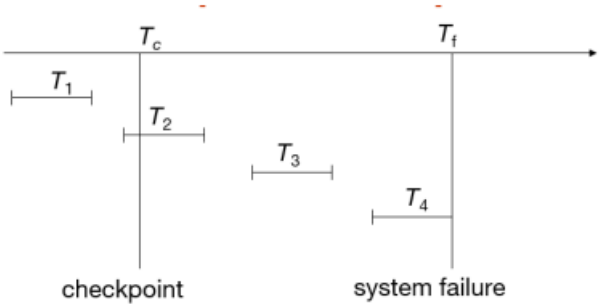
Recovery actions in each case above are:

a) undo (T0): B is restored to 2000 and A to 1000, and log records < T0, B, 2000 >, < T0, A, 1000 >, < T0, **abort**> are written out
b) redo (T0) and undo (T1): A and B are set to 950 and 2050 and C is restored to 700. Log records < T1, C, 700 >, < T1, **abort**> are written out
c) redo (T0) and redo (T1): 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*** >

    - 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
- Some earlier part of the log may be needed for undo operations
    - Continue scanning backwards till a record < T_i **start** > is found for every transaction T_i in L
    - Parts of log prior to earliest < T_i **start** > record above are not needed for recovery, and can be erased whenever desired



- Any transactions that committed before the last checkpoint should be ignored
    - T1 can be ignored (updates already output to disk due to checkpoint)
- Any transactions that committed since the last checkpoint need to be redone
    - T2 and T3 redone
- Any transaction that was running at the time of failure needs to be undone and restarted
    - T4 undone

Transactional Logging

Hot Backup: Recap

- 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

Transactional Logging as 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**

Transactional Logging with Recovery: Example

- To understand how **Transactional Logging** works we consider Figure 1 that represents a chunk of a database just before a backup has been started

- While the backup is in progress, modifications may continue to occur to the database. For example, a request to modify the data at location **"4325"** to '0' arrives.

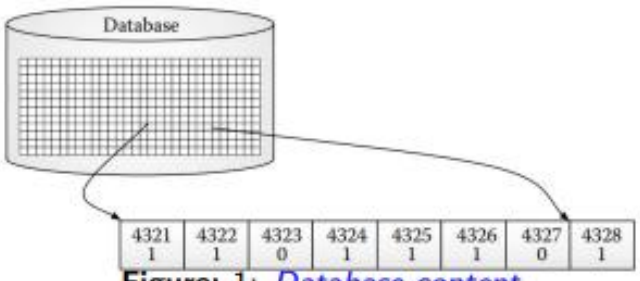


Figure: 1: *Database content*

- When a request comes through to modify a part of the DB, the modifications will be written in the **given order compulsorily** 1 Transaction Log 2 Database (itself) This is depicted in Figure 2

- If a crash occurs **before writing to the database then the inconsistent backed up file is recovered first, and then the pending modifications in the transaction log (backed up*) are applied to re-establish consistency**

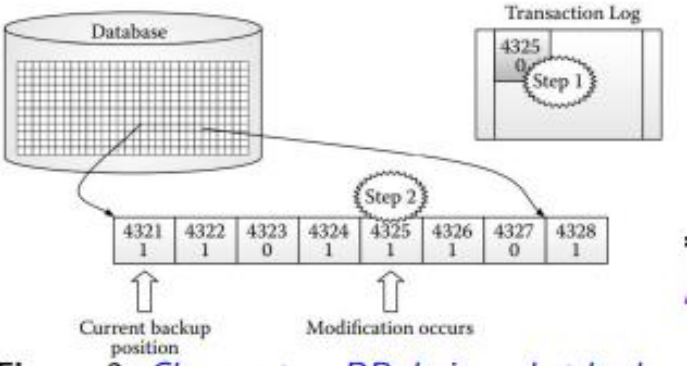


Figure: 2: *Changes to a DB during a hot backup*

***Note:** The Transactional Log itself is backed up using Hot Backup the Data is backed up incrementally*

Consider in the previous scenario before the occurrence of crash, another request modifies the content of location **"4321"** to **'0'**. Incidentally**, this change gets written in the database itself** (recall: Immediate Modification). This is indicated in Figure 3

- Figure 3 is the state of the database after which the system crashes. Note that this part has already been backed up, and hence, the backup is inconsistent with the database.

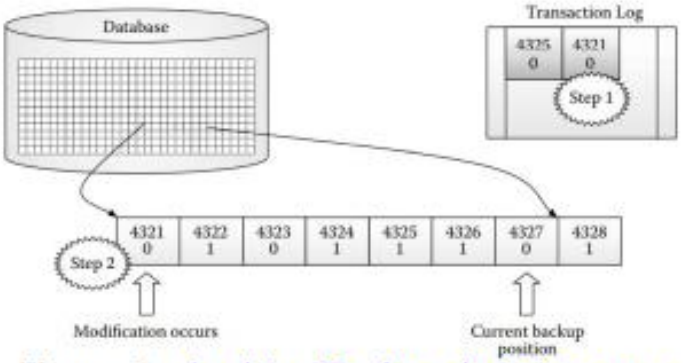


Figure: 3: *Applying Tr. logs during recovery*

- Recovery Phase:
  - Data recovery is done from the last data backup set (Figure 1)
  - Log recovery is done from the Transaction Log backup set. It will be same as the current transaction log because of Hot backup
  - Figure 4 shows the recovered database and log

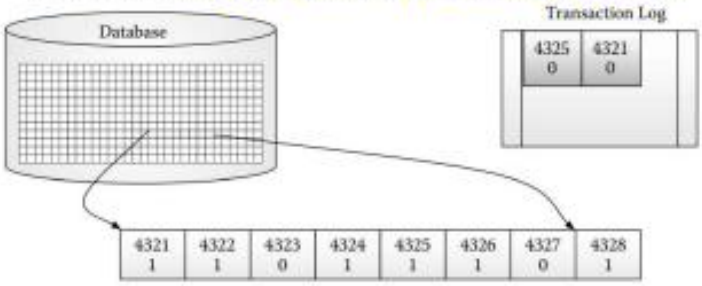


Figure: 4: *Recovered DB files and Tr. logs*

- The recovered database is inconsistent. To re-establish consistency all transaction logs generated between the start of the backup and the end of the backup must be **replayed**

- When using transactional logging we distinguish between recover and restore:
  - **Recover:** retrieve from the backup media the database files and transaction logs, and
  - **Restore:** reapply database consistency based on the transaction logs

- For our restore process, we recover inconsistent database files and completed transaction logs. The recovered files will resemble the configuration shown in Figure 4
- The final database state after replaying log on the recovered database is displayed in Figure 5
- **The state of database is consistent**

- Note that an unnecessary log replay is shown occurring for block 4325. Whether such replays will occur is dependent on the database being used. For instance, a database vendor might choose to replay all logs because it would be faster than first determining whether a particular logged activity needs to be replayed

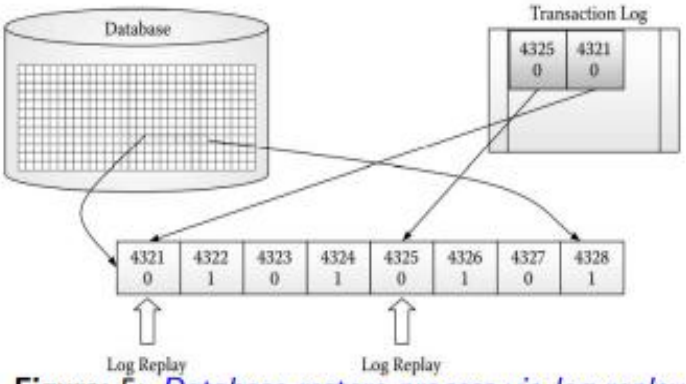


Figure: 5: *Database restore process via log replay*

- Once all transaction logs have been replayed, the database is said to have been restored, that is, it is at a point where it can now be opened for user access

Recovery Algorithm

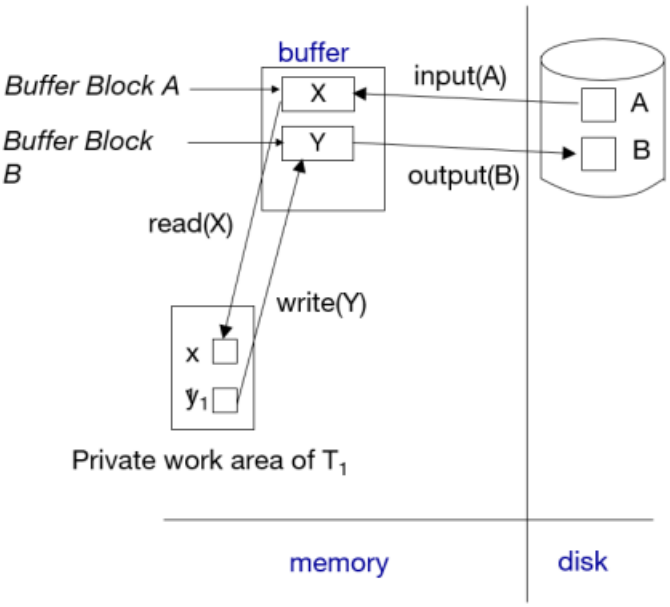
Recovery Schemes

- **So far:**
  - We covered key concepts
  - We assumed serial execution of transactions
- **Now:**
  - We discuss concurrency control issues
  - We present the components of the basic recovery algorithm

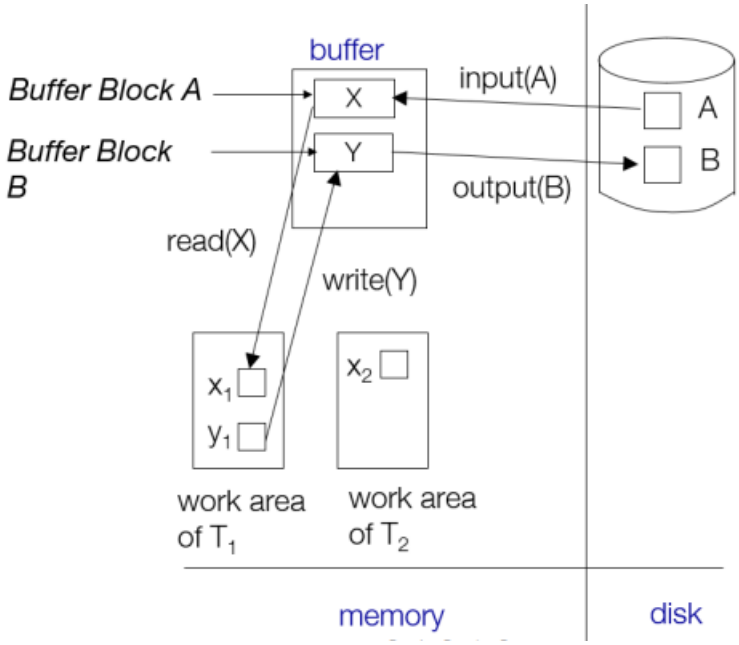
Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that if **a transaction Ti has modified an item, no other transaction can modify the same item until Ti has committed or aborted**
  - That is, the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how do we perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log

Example of Data Access with Serial Transaction



Example of Data Access with Concurrent Transactions



Recovery Algorithm

- Logging (during normal operation):
  - < Ti **start** > at transaction start
  - < Ti , Xj , V1, V2 > for each update, and
  - < Ti **commit**> at transaction end

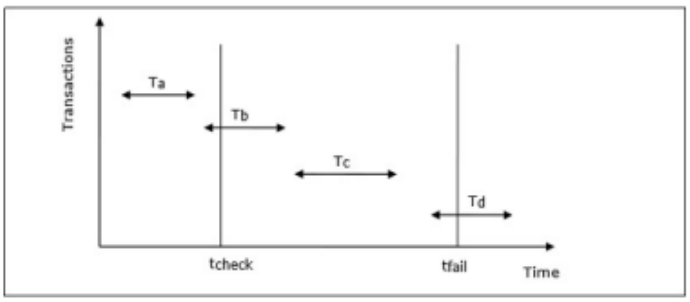
Recovery Algorithm (2)

- **Transaction rollback (during normal operation)**
  - Let Ti be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of Ti of the form < Ti , Xj , V1, V2 >
    - perform the undo by writing V1 to Xj ,
    - write a log record < Ti , Xj , V1 > . . . such log records are called **Compensation Log Records (CLR)**
  - Once the record < Ti **start**> is found stop the scan and write the log record < Ti **abort**>

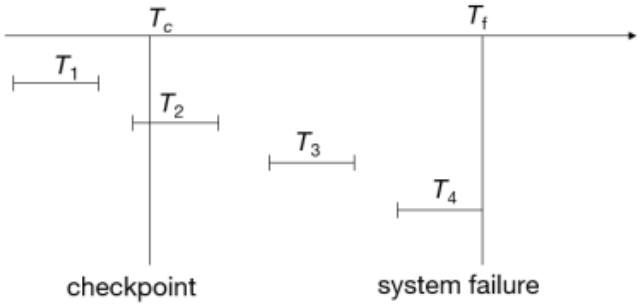
Recovery Algorithm (3): Checkpoints Recap

- Let the time of checkpointing is *tcheck* and the time of system crash is *tfail*
- Let there be four transactions Ta, Tb, Tc and Td such that:
  - Ta commits before checkpoint
  - Tb starts before checkpoint and commits before system crash
  - Tc starts after checkpoint and commits before system crash
  - Td starts after checkpoint and was active at the time of system crash

- The actions that are taken by the recovery manager are:
  - Nothing is done with Ta
  - Transaction redo is performed for Tb and Tc
  - Transaction undo is performed for Td



Recovery Algorithm (4): Checkpoints Recap



- Any transactions that committed before the last checkpoint should be ignored
  - T1 can be ignored (updates already output to disk due to checkpoint)
- Any transactions that committed since the last checkpoint need to be redone
  - T2 and T3 redone
- Any transaction that was running at the time of failure needs to be undone and restarted
  - T4 undone

Recovery Algorithm (5): Redo-Undo Phases

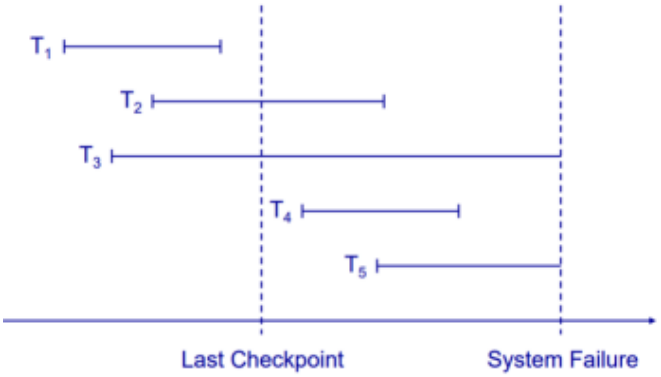
- **Recovery from failure**: Two phases
  - **Redo phase:** Replay updates of all transactions, whether they committed, aborted, or are incomplete
  - **Undo phase:** Undo phase: Undo all incomplete transactions

Requirement:

- Transactions of type T1 need no recovery
- Transactions of type T2 or T4 need to be redone
- Transactions of type T3 or T5 need to be undone and restarted

Strategy:

- Ignore T1
- Redo T2, T3, T4 and T5
- Undo T3 and T



Recovery Algorithm (6): Redo Phase

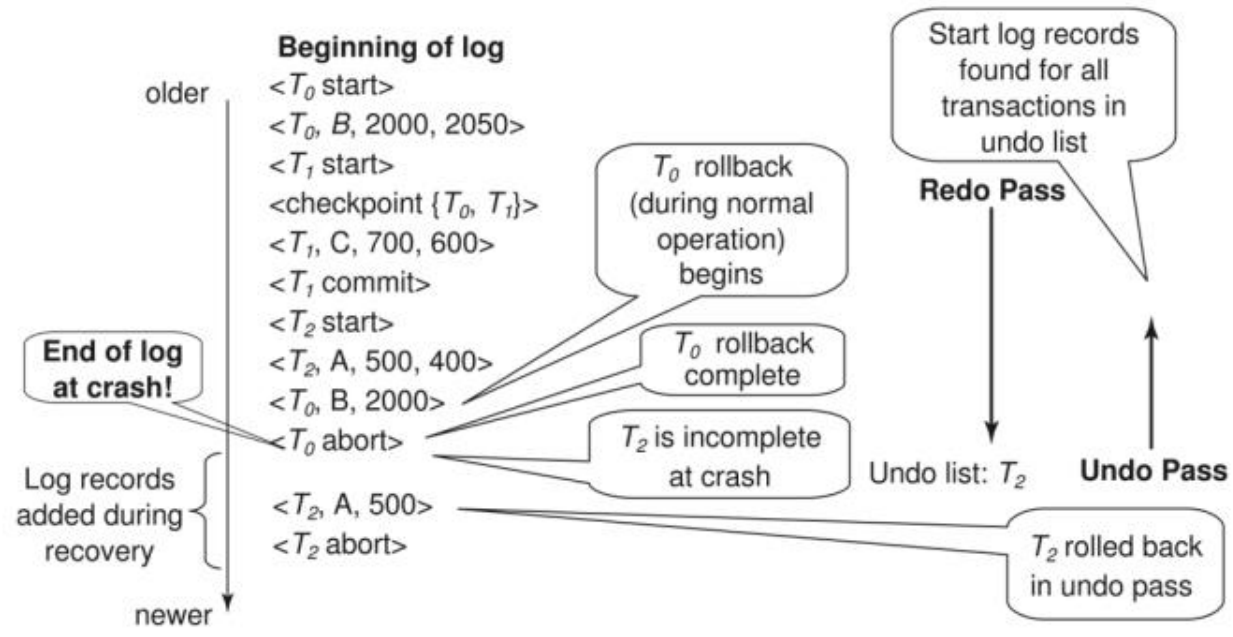
- Find last < **checkpoint** L> record, and set undo-list to L
- Scan forward from above < **checkpoin**t L> record
  - Whenever a record < Ti , Xj , V1, V2 > is found, redo it by writing V2 to Xj
  - Whenever a log record < Ti s**tar**t> is found, add Ti to undo-list
  - Whenever a log record < Ti **commi**t> or < Ti **abort**> is found, remove Ti from undo-list
- Steps for the REDO operation are:
  - If the transaction has done INSERT, the recovery manager generates an insert from the log
  - If the transaction has done DELETE, the recovery manager generates a delete from the log
  - If the transaction has done UPDATE, the recovery manager generates an update from the log.

Recovery Algorithm (7): Undo Phase

- Scan log backwards from end
  - Whenever a log record < Ti , Xj , V1, V2 > is found where Ti is in undo-list perform same actions as for transaction rollback:
    - Perform undo by writing V1 to Xj
    - Write a log record < Ti , Xj , V1 >
  - Whenever a log record < Ti **start**> is found where Ti is in undo-list
    - Write a log record < Ti **abort**>
    - Remove Ti from undo-list
  - Stop when undo-list is empty That is, < Ti start > has been found for every transaction in undo-list

- Steps for the UNDO operation are:
  - If the faulty transaction has done INSERT, the recovery manager deletes the data item(s) inserted
  - If the faulty transaction has done DELETE, the recovery manager inserts the deleted data item(s) from the log
  - If the faulty transaction has done UPDATE, the recovery manager eliminates the value by writing the before-update value from the log
- After undo phase completes, normal transaction processing can commence

Recovery Algorithm (8): Example



Recovery with Early Lock Release

- Any **index used in processing a transaction,** such as a B+-tree, can be treated as normal data
- To increase concurrency, the B+-tree concurrency control algorithm often allow locks to be released early, in a non-two-phase manner
- As a result of early lock release, it is possible that
    - a value in a B+-tree node is updated by one transaction T1, which inserts an entry (V1, R1), and subsequently
    - by another transaction T2, which inserts an entry (V2, R2) in the same node, moving the entry (V1, R1) even before T1 completes execution

- At this point, we cannot undo transaction T1 by replacing the contents of the node with the old value prior to T1 performing its insert, since that would also undo the insert performed by T2; transaction T2 may still commit (or may have already committed)
- Hence, the only way to undo the effect of insertion of (V1, R1) is to execute a corresponding delete operation
- Support for high-concurrency locking techniques, such as those used for B +-tree concurrency control, which release locks early
    - Supports "logical undo"

- Recovery based on **"repeating history",** whereby recovery executes exactly the same actions as normal processing
    - including redo of log records of incomplete transactions, followed by subsequent undo
    - Key benefits
        - supports logical undo
        - easier to understand/show correctness
- Early lock release is important not only for indices, but also for operations on other system data structures that are accessed and updated very frequently like:
    - data structures that track the blocks containing records of a relation
    - the free space in a block
    - the free blocks

Logical Undo Logging

- Operations like B +-tree insertions and deletions release locks early
    - They cannot be undone by restoring old values (physical undo), since once a lock is released, other transactions may have updated the B +-tree
    - Instead, insertions (deletions) are undone by executing a deletion (insertion) operation (known as logical undo)

- For such operations, undo log records should contain the undo operation to be executed
    - Such logging is called logical undo logging, in contrast to physical undo logging
        - Operations are called logical operations
    - Other examples:
        - delete of tuple, to undo insert of tuple
            - allows early lock release on space allocation information
        - subtract amount deposited, to undo deposit
            - allows early lock release on bank balance

Physical Redo

- Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo
    - Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts
    - Physical redo logging does not conflict with early lock release

Operation Logging: Process

- When operation starts, log < Ti , Oj , **operation-begin** >. Here Oj is a unique identifier of the operation instance
- While the system is executing the operation, it creates update log records in the normal fashion for all updates performed by the operation
    - the usual old-value **(physical undo information)** and new-value **(physical redo information)** is written out as usual for each update performed by the operation;
    - the old-value information is required in case the transaction needs to be rolled back before the operation completes
- When operation completes, < Ti , Oj , **operation-end**,U > is logged, where U contains information needed to perform a logical undo information
    - For example, if the operation inserted an entry in a B+-tree, the undo information U would indicate that a deletion operation is to be performed, and would identify the B+-tree and what entry to delete from the tree. This is called **logical logging**
    - In contrast, logging of old-value and new-value information is called **physical logging**, and the corresponding log records are called **physical log records**

Operation Logging (2): Example

- Insert of (key, record-id) pair (K5, RID7) into index I9

<T1, O1, operation-begin>
....
<T1, X, 10, K5>
<T1, Y, 45, RID7>
Physical redo of steps in insert
<T1, O1, operation-end, (delete I9, K5, RID7)>

Operation Logging (3)

- If crash/rollback occurs before operation completes:
    - the **operation-end** log record is not found, and
    - the physical undo information is used to undo operation
- If crash/rollback occurs after the operation completes:
    - the **operation-end** log record is found, and in this case
    - logical undo is performed using U; the physical undo information for the operation is ignored
- **Redo of operation (after crash) still uses physical redo information**

Transaction Rollback with Logical Undo

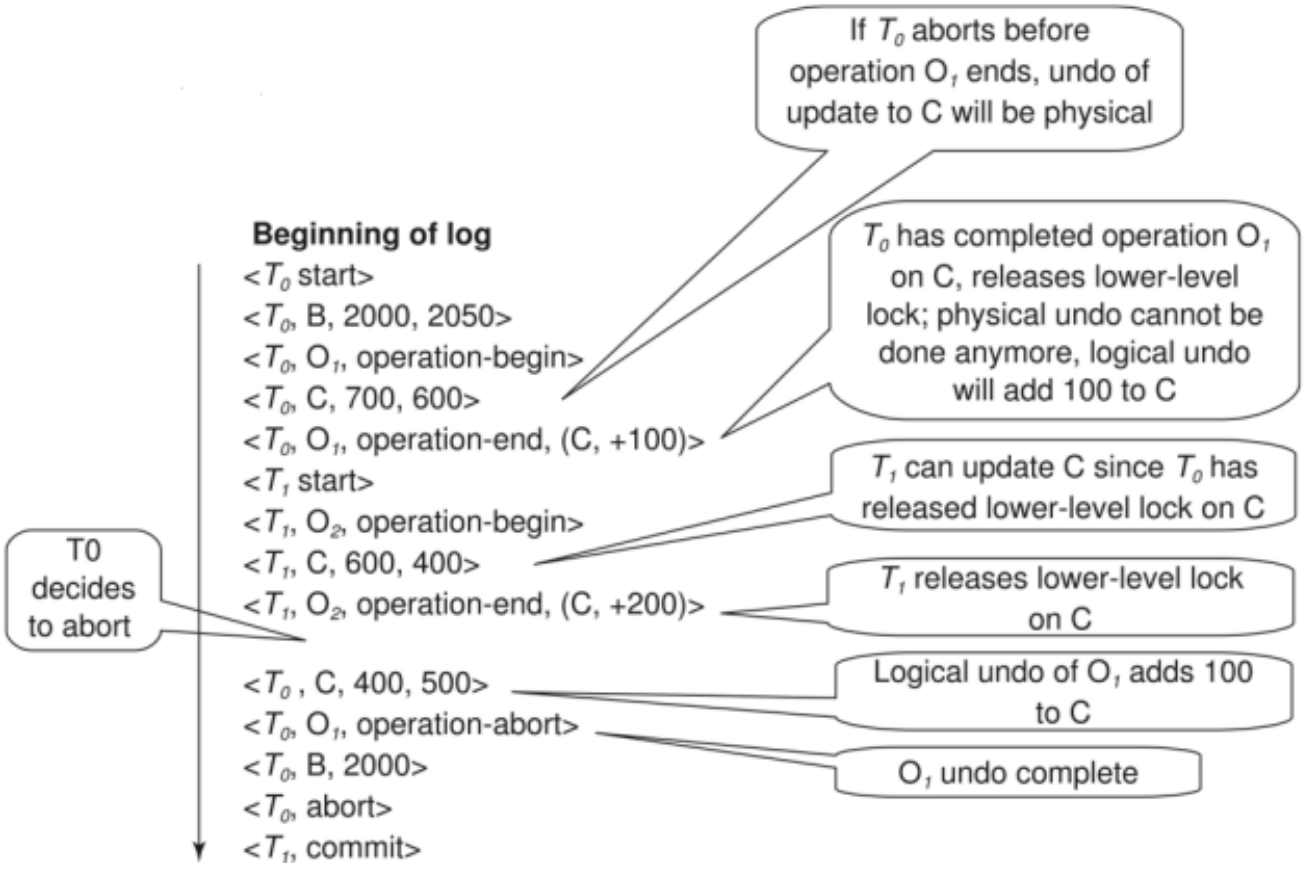
Rollback of transaction Ti , scan the log backwards

a) If a log record < Ti , X, V1, V2 > is found, perform the undo and log < Ti , X, V1 >
b) If a < Ti , Oj , **operation-end,** U > record is found
- Rollback the operation logically using the undo information U
  - Updates performed during roll back are logged just like during normal operation execution
  - At the end of the operation rollback, instead of logging an operation-end record, generate a record < Ti , Oj , operation-abort >
- Skip all preceding log records for Ti until the record < Ti , Oj **operation-begin**> is found

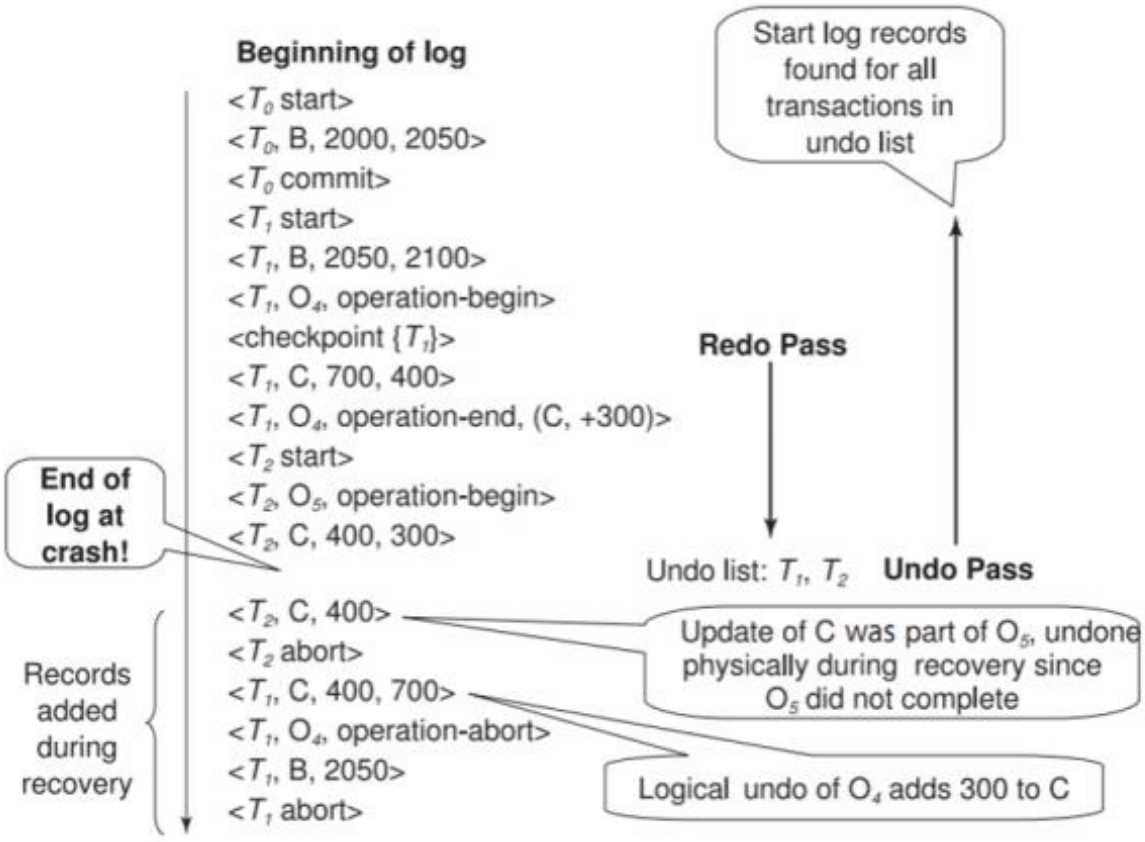
c) If a redo-only record is found ignore it
d) If a < Ti , Oj , **operation-abort** > record is found: skip all preceding log records for Ti until the record < Ti , Oj , **operation-begin** >is found

e) Stop the scan when the record < Ti , **start**> is found
f) Add a < Ti , **abort**> record to the log Note:
- Cases c) and d) above can occur only if the database crashes while a transaction is being rolled back
- Skipping of log records as in case d) is important to prevent multiple rollback of the same operation

Transaction Rollback with Logical Undo



Failure Recovery with Logical Undo



Transaction Rollback: Another Example

• Example with a complete and an incomplete operation

< T1,start >
< T1, O1, operation-begin>
. . .
< T1, X, 10,K5 >
< T1, Y , 45, RID7 >
< T1, O1, operation-end, (delete I 9,K5, RID7) >
< T1, O2, operation-begin>
< T1, Z, 45, 70 >
← T1 Rollback begins here
< T1, Z, 45 > ← Redo-only log record during physical undo (of incomplete O2)
< T1, Y ,. . . ,. . . > ← Normal redo records for logical undo of O1
. . .
< T1, O1, operation-abort> ← What if crash occurred immediately after this?
< T1, abort >

Recovery Algorithm with Logical Undo

Basically same as earlier algorithm, except for changes described earlier for transaction rollback

- (Redo phase): Scan log forward from last < **checkpoint** L > record till end of log
  - Repeat history by physically redoing all updates of all transactions,
  - Create an undo-list during the scan as follows
    - undo-list is set to L initially
    - Whenever < Ti **start**> is found Ti is added to undo-list
    - Whenever < Ti **commit**> or < Ti **abort**> is found, Ti is deleted from undo-list

    - This brings database to state as of crash, with committed as well as uncommitted transactions having been redone
    - Now undo-list contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back

- (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in undo-list
    - Log records of transactions being rolled back are processed as described earlier, as they are found
        - Single shared scan for all transactions being undone
    - When < Ti s**tart**> is found for a transaction Ti in undo-list, write a < Ti **abort**> log record.
    - Stop scan when < Ti **start**> records have been found for all Ti in undo-list
- This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

Plan for Backup and Recovery

Deciding factors for having a Backup & Recovery setup.

- **Data Importance**
    - How important is the information in your database for your company? For business-critical data you will create a plan that involves making extra copies of your database over the same period and ensuring that the copies can be easily restored when required
- **Frequency of Change**
    - How often does your database get updated? For instance, if critical data is modified daily then you should make a daily backup schedule.
- **Speed**
    - How much time do you need to back up or recover your files? Recovery speed is an important factor that determines the maximum possible time period that could be spent on database backup and recovery.
- **Equipment**
    - Do you have necessary equipment to make backups? To perform timely backups and recoveries, you need to have proper software and hardware resources.
- **Employees**
    - Who will be responsible for implementing your database backup and recovery plan? Ideally, one person should be appointed for controlling and supervising the plan, and several IT specialists (e.g. system administrators) should be responsible for performing the actual backup and recovery of data.
- **Storing**
    - Where do you plan to store database duplicates? In case of Online/Offsite storage you can recover your systems in case of a natural disaster. Storing backups on-site is essential to quick restore. But onsite storage has capacity bottlenecks and high maintenance costs.

RAID: Redundant Array of Independent Disks

- Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - high capacity and high speed by using multiple disks in parallel,
    - high reliability by storing data redundantly, so that data can be recovered even if a disk fails

- The chance that some disk out of a set of n disks will fail is much higher than the chance that a specific single disk will fail
    - For example, a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
    - Techniques for using redundancy to avoid data loss are critical with large numbers of disks

- Originally a cost-effective alternative to large, expensive disks
    - "I" in RAID originally stood for inexpensive
    - Today RAIDs are used for their higher reliability and bandwidth
        - The "I" is interpreted as independent

Improvement of Reliability via Redundancy: Mirroring

- **Redundancy**: Store extra information that can be used to rebuild information lost in a disk failure
- **Mean time to data loss** depends on mean time to failure, and **mean time to repair**
    - For example, MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500*106 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

- **Mirroring** (or **shadowing**)
    - Duplicate every disk. Logical disk consists of two physical disks.
    - Every write is carried out on both disks
        - Reads can take place from either disk
    - If one disk in a pair fails, data still available in the other
        - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
            - Probability of combined event is very small
            - Except for dependent failure modes such as fire or building collapse or electrical power surges

Improvement of Reliability via Redundancy (2): Striping

- **Bit-level Striping**: Split the bits of each byte across multiple disks
    - In an array of eight disks, write bit i of each byte to disk i
    - Each access can read data at eight times the rate of a single disk
    - But seek/access time worse than for a single disk
        - Bit level striping is not used much any more
- **Byte-level Striping**: Each file is split up into parts one byte in size. Using n = 4 disk array as an example
    - the 1st byte would be written to the 1st drive
    - the 2nd byte to the 2nd drive and so on, until
    - the 5th byte is then written to the 1st drive again and the whole process starts over
    - the i th byte is then written to the (((i − 1) mod n) + 1)th drive
- **Block-level Striping**: With n disks, block i of a file goes to disk (i mod n) + 1
    - Requests for different blocks can run in parallel if the blocks reside on different disks
    - A request for a long sequence of blocks can utilize all disks in parallel

Improvement of Reliability via Redundancy (3): Parity

- **Bit-Interleaved Parity**: A single parity bit is enough for error correction, not just detection, since we know which disk has failed
    - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
    - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)
- **Block-Interleaved Parity**: Uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from n other disks
    - When writing data block, corresponding block of parity bits must also be computed and written to parity disk
    - To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks

Standard RAID Levels

- A basic set of RAID configurations that employ the techniques of striping, mirroring, or parity to create large reliable data stores from multiple general-purpose HDDs
- The most common types are RAID 0 (striping), RAID 1 (mirroring) and its variants, RAID 5 (distributed parity), and RAID 6 (dual parity)

- Multiple RAID levels can also be combined or nested, for instance RAID 10 (striping of mirrors) or RAID 01 (mirroring stripe sets)
- RAID levels are standardized by the Storage Networking Industry Association (SNIA) in the Common RAID Disk Drive Format (DDF) standard
- The numerical values only serve as identifiers and do not signify any metric

- While most RAID levels can provide good protection against and recovery from hardware defects or defective sectors/read errors (hard errors), they do not provide any protection against data loss due to catastrophic failures (fire, water) or soft errors such as user error, software malfunction, or malware infection
- For valuable data, RAID is only one building block of a larger data loss prevention and recovery scheme – it cannot replace a backup plan

RAID 0: Striping

- RAID level-0 only uses data striping, no redundant information is maintained
- If one disk fails, then all data in the disk array is lost
- Independent of the number of data disks, the effective space utilization for a RAID Level-0 system is always 100 percent
- RAID Level-0 has the best write performance of all RAID levels because the absence of redundant information implies that no redundant information needs to be updated.
- This solution is the least costly
- Reliability is very poor

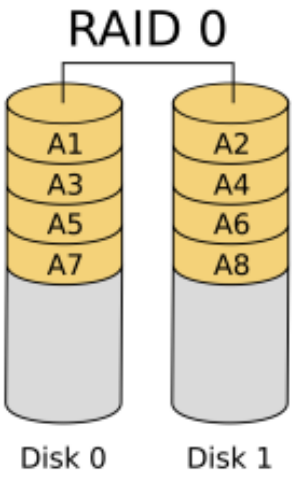


Image source: Standard RAID levels (Accessed 19-Aug-2021)

RAID 1: Mirroring

- RAID 1 employs mirroring, maintaining two identical copies of the data on two different disks • It is the most expensive solution
- It provides excellent fault tolerance
- Every write of a disk block involves a write on both disks
- With two copies of each block exist on different disks, we can distribute reads between the two disks and allow parallel reads
- RAID Level-1 does not stripe the data over different disks. Thus the transfer rate for a single request is com parable to the transfer rate of a single disk
- The effective space utilization is 50 percent, independent of the number of data disks

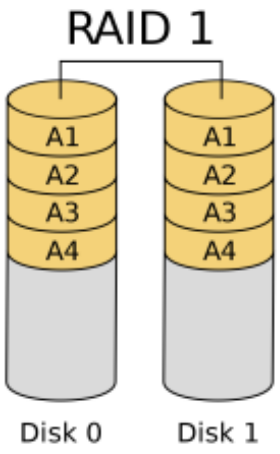


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

RAID 2: Parity

- RAID 2 uses designated drive for parity
- In RAID 2, the striping unit is a single bit
- Hamming Code is used for parity
    - Hamming codes can detect up to two-bit errors or correct one-bit errors
    - For a 4-bit data, 3 bits are added
    - Simple parity code cannot correct errors, and can detect only an odd number of bits in error

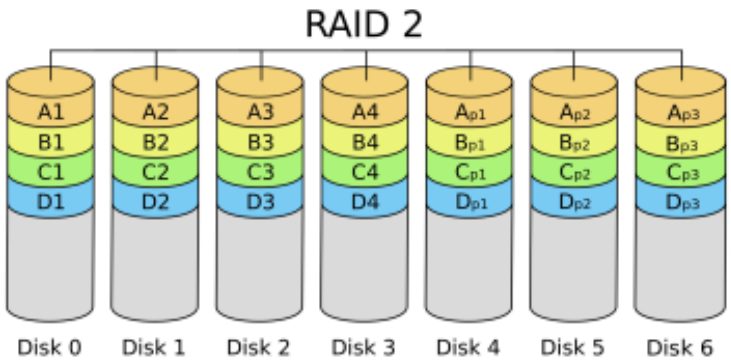


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- In a disk array with D data disks, the smallest unit of transfer for a read is a set of D blocks. It is so because each bit of the data is stored in different blocks of D disks subsequently (Bit-level striping)
- Writing a block involves reading D blocks into main memory, modifying D + C blocks, and writing D + C blocks to disk, where C is the number of check disks. This sequence of steps is called a read-modify-write cycle

RAID 3: Byte Striping + Parity

- RAID 3 has a single check disk with parity information. Thus, the reliability overhead for RAID 3 is a single disk, the lowest overhead possible
- RAID 3 consists of byte-level striping with dedicated parity. Therefore the data transfer rate of this level is high because data can be accessed in parallel

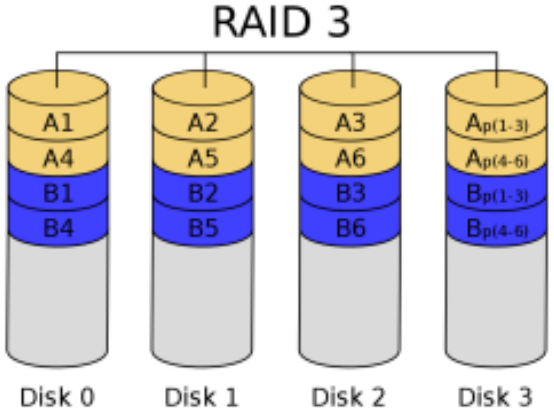


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- RAID-3 cannot service multiple requests simultaneously: This is so because any single block of data will be spread across all members of the set and will reside in the same physical location on each disk and thus every single I/O request has to be addressed by working on every disk in the array

RAID 4: Block Striping + Parity

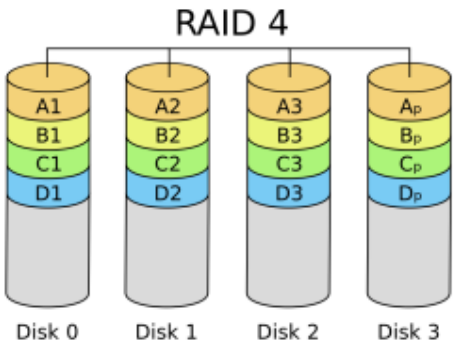


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- RAID 4 has a striping unit of a disk block instead of a single bit, as in RAID 3
- Read requests of the size of a disk block can be served entirely by the disk where the requested block resides therefore RAID 4 provides good performance for data reads
- Provides recovery of corrupted or lost data using XOR recovery mechanism
- If a disk experiences a failure, recovery can be made by simply XORing all the remaining data bits and the parity bit
- Facilitates recovery of at most 1 disk failure. At this level, if more than one disk fails, then there is no way to recover the data
- Write performance is low due to the need to write all parity data to a single disk

RAID 5: Distributed Parity

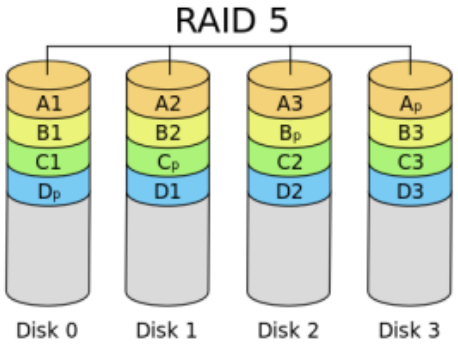


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- RAID 5 improves upon RAID 4 by distributing the parity blocks uniformly over all disks instead of storing them on a single check disk
- Several write requests can potentially be processed in parallel since the bottleneck of a unique check disk has been eliminated
- Read requests have a higher level of parallelism. Since the data is distributed over all disks, read requests involve all disks, whereas, in systems with a dedicated check disk, the check disk never participates in reads
- This level too allows recovery of only 1 disk failure like level 4

RAID 6: Dual Parity

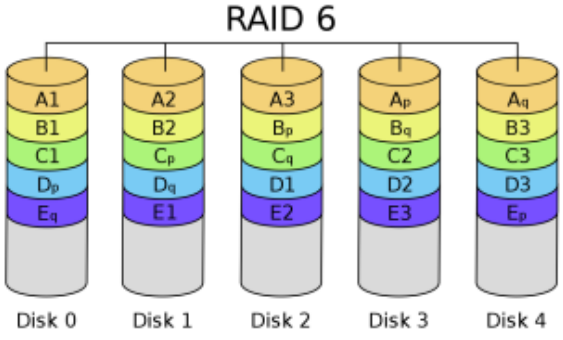


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- RAID 6 extends RAID 5 by adding another parity block, thus it uses block-level striping with two parity blocks distributed across all member disks
- Write performance of RAID 6 is poorer than RAID 5 because of the increased complexity of parity calculation
- RAID 6 use Reed-Solomon Codes to recover from up to two simultaneous disk failures. Therefore it can handle a disk failure during recovery of a failed disk

Hybrid RAID: Nested RAID levels

- Nested RAID levels (Hybrid RAID), combine two or more of the standard RAID levels to gain performance, additional redundancy or both, as a result of combining properties of different standard RAID layouts.
- Nested RAID levels are usually numbered using a series of numbers ◦ The first number in the numeric designation denotes the lowest RAID level in the ”stack”, while ◦ the rightmost one denotes the highest layered RAID level
- For example, RAID 50 layers the data striping of RAID 0 on top of the distributed parity of RAID 5
- Nested RAID levels include RAID 01, RAID 10, RAID 100, RAID 50 and RAID 60, which all combine data striping with other RAID techniques
- As a result of the layering scheme, RAID 01 and RAID 10 represent significantly different nested RAID levels

RAID 01 (RAID 0+1): Mirror of Stripes

- RAID 01 is a mirror of stripes
- It achieves both replication and sharing of data between disks
- The usable capacity of a RAID 01 array is the same as in a RAID 1 array made of the same drives, in which one half of the drives is used to mirror the other half: $(N/2) \cdot S_{min}$, where N is the total number of drives and S_{min} is the capacity of the smallest drive in the array
- At least four disks are required in a standard RAID 01 configuration, but larger arrays are also used

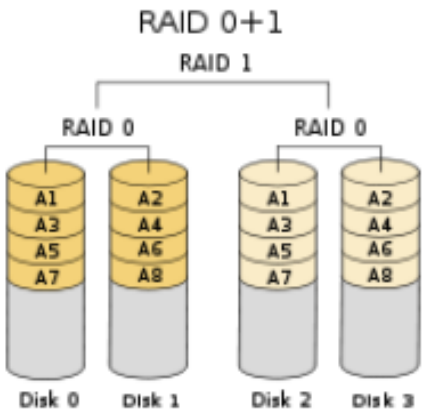


Image source: Nested RAID levels
(Accessed 23-Aug-2021)

RAID 10 (RAID 1+0): Stripe of Mirrors

- RAID 10 is a stripe of mirrors
- RAID 10 is a RAID 0 array of mirrors, which may be two- or three-way mirrors, and requires a minimum of four drives
- RAID 10 provides better throughput and latency than all other RAID levels except RAID 0 (which wins in throughput)
- Thus, it is the preferable RAID level for I/O-intensive applications such as database, email, and web servers, as well as for any other use requiring high disk performance

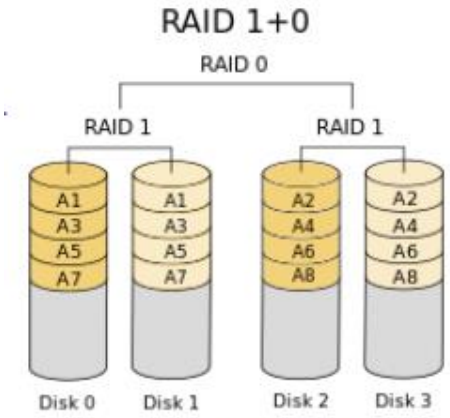


Image source: Nested RAID levels
(Accessed 23-Aug-2021)

Choice of RAID Levels

- Different RAID Levels have different speed and fault tolerance properties
- RAID level 0 is not fault tolerant
- Levels 1, 1E, 5, 50, 6, 60, and 1+0 are fault tolerant to a different degree - should one of the hard drives in the array fail, the data is still reconstructed on the fly and no access interruption occurs

- RAID levels 2, 3, and 4 are theoretically defined but not used in practice
- There are some more complex layouts like RAID 5E/5EE (integrating some spare space) and RAID DP
  - "E" often stands for "Enhanced" or "Extended"
  - Some of them use hot spare drives

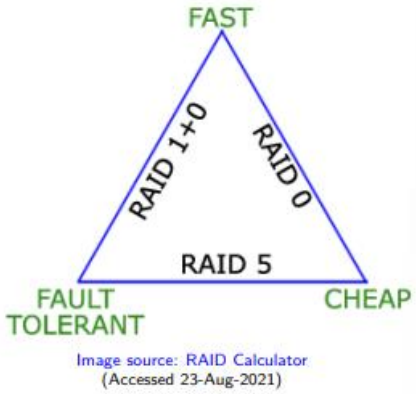


Image source: RAID Calculator (Accessed 23-Aug-2021)

Choice of RAID Levels (2)

- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
  - For example, data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (Level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for most applications

Choice of RAID Levels (3)

- Level 1 provides much better write performance than level 5
  - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
- disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
- I/O requirements have increased greatly, e.g. for Web servers
- When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
  - so there is often no extra monetary cost for Level 1!
- Level 5 is preferred for applications with low update rate, and large amounts of data
- Level 1 is preferred for all other applications

Comparison of RAID: Theoretical

Level	Description	Min.[b] # of drives	Space Efficiency	Fault Tolerance (*Drives*)	Performance (*as factor of single disk*)	
					Read	Write
RAID 0	Block-level striping without parity or mirroring	2	1	None	n	n
RAID 1	Mirroring without parity or striping	2	$\frac{1}{n}$	$n-1$	$n^{[a]}$	$1^{[c]}$
RAID 2	Bit-level striping with Hamming code for error correction	3	$1-\frac{1}{n}\lg(n+1)$	One[d]	Depends	Depends
RAID 3	Byte-level striping with dedicated parity	3	$1-\frac{1}{n}$	One	$n-1$	$n-1^{[e]}$
RAID 4	Block-level striping with dedicated parity	3	$1-\frac{1}{n}$	One	$n-1$	$n-1^{[e]}$
RAID 5	Block-level striping with distributed parity	3	$1-\frac{1}{n}$	One	$n^{[e]}$	single sector: $\frac{1}{4}$ full stripe: $n-1^{[e]}$
RAID 6	Block-level striping with double distributed parity	4	$1-\frac{2}{n}$	Two	$n^{[e]}$	single sector: $\frac{1}{6}$ full stripe: $n-2^{[e]}$

A. Theoretical maximum, as low as single-disk performance in practice
B. Assumes a non-degenerate minimum number of drives
C. If disks with different speeds are used in a RAID 1 array, overall write performance is equal to the speed of the slowest disk
D. RAID 2 can recover from one drive failure or repair corrupt data or parity when a corrupted bit's corresponding data and parity are good
E. Assumes hardware capable of performing associated calculations fast enough

Comparison of RAID: Practical

Features	RAID 0	RAID 1	RAID 5	RAID 6	RAID 10
Minimum # of drives	2	2	3	4	4
Fault tolerance	None	Single-drive failure	Single-drive failure	Two-drive failure	Up to 1 disk failure in each sub-array
Read performance	High	Medium	Low	Low	High
Write Performance	High	Medium	Low	Low	Medium
Capacity utilization	100%	50%	67% – 94%	50% – 88%	50%
Typical applications	*High end workstations, data logging, real-time rendering, very transitory data*	*Operating systems, transaction databases*	*Data warehouse, web servers, archiving*	*Data archive, backup to disk, high availability solutions, servers with large capacity requirements*	*Fast databases, file servers, application servers*

What Does RAID Not Do?

- RAID does not equate to 100% uptime: Nothing can. RAID is another tool on in the toolbox meant to help minimize downtime and availability issues. There is still a risk of a RAID card failure, though that is significantly lower than a HDD failure

- RAID does not replace backups: Nothing can replace a well planned and frequently tested backup implementation!
- RAID does not protect against data corruption, human error, or security issues: While it can protect you against a drive failure, there are innumerable reasons for keeping backups. So RAID is not a replacement for backups
- RAID does not necessarily allow to dynamically increase the size of the array: If you need more disk space, you cannot simply add another drive to the array. You are likely going to have to start from scratch, rebuilding/reformatting the array. Luckily, Steadfast engineers are here to help you architect and execute whatever systems you need to keep your business running.
- RAID isn't always the best option for virtualization and high-availability failover: You will want to look at SAN solutions

BY – ADITYA DHAR DWIVEDI