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

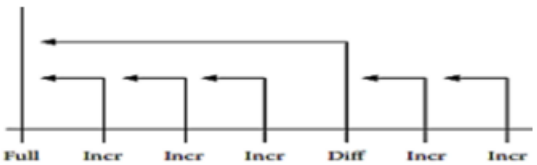
- To evaluate how differential backups might work within an environment, consider the sample backup schedule shown in the figure below.

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.

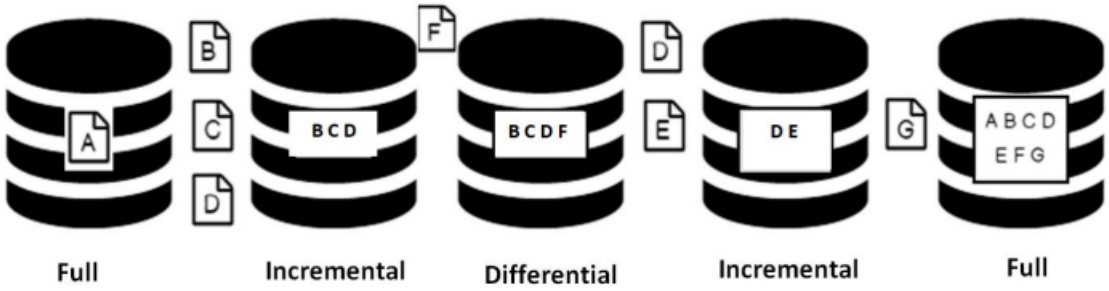


Figure: Backup Types

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

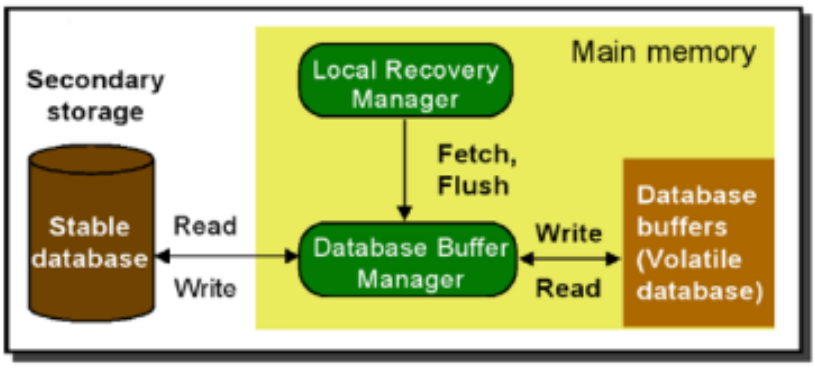
    - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database
    - A failure may occur after one of these modifications have been made but before both of them are made
    - Modifying the database without ensuring that the transaction will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

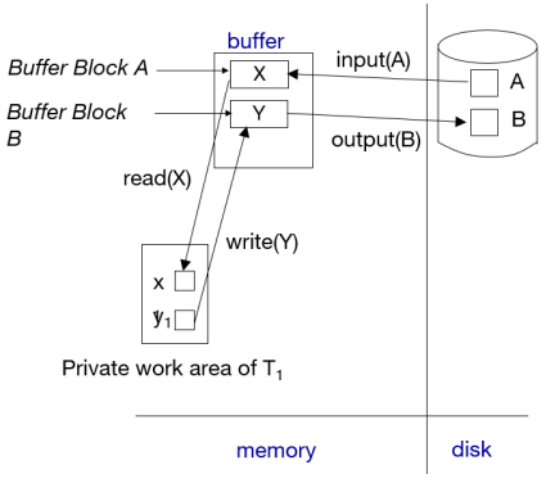


Protecting storage media from failure during data transfer (cont.):

- 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

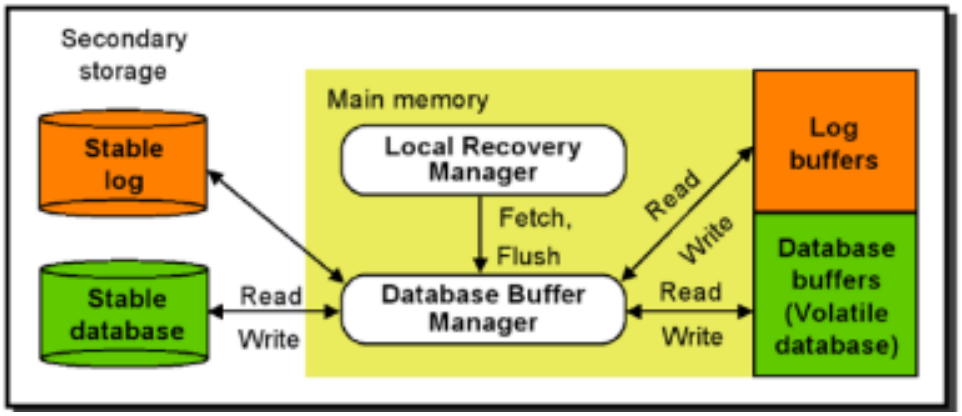


Recovery and Atomicity

- 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

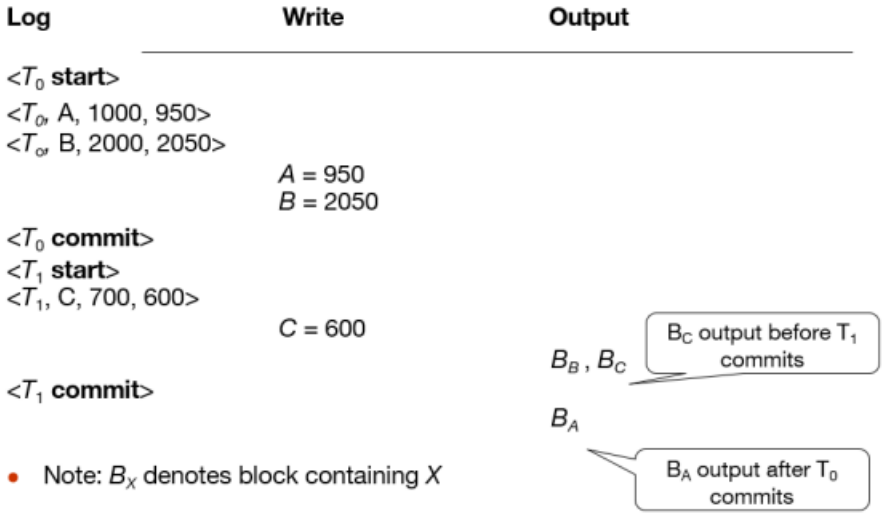
- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
  - Update log record must be written before a database item is written
    - We assume that the log record is output directly to stable storage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

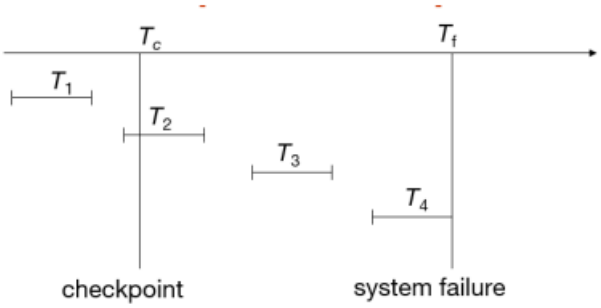
Recovery actions in each case above are:

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.

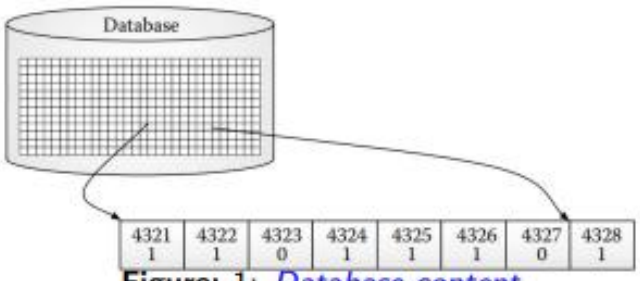


Figure: 1: *Database content*

- 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**

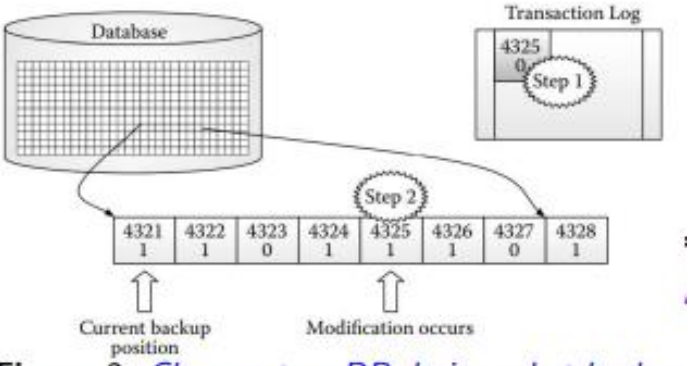


Figure: 2: *Changes to a DB during a hot backup*

***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.

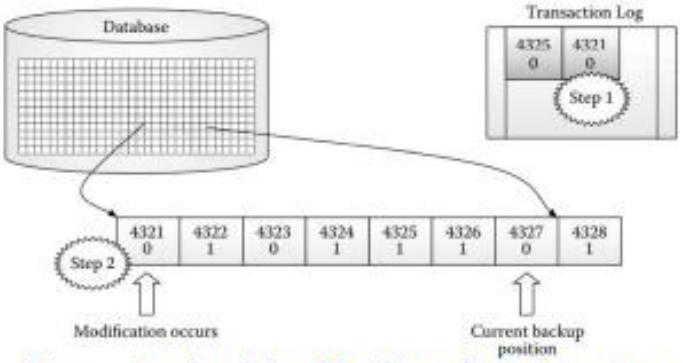


Figure: 3: *Applying Tr. logs during recovery*

- 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

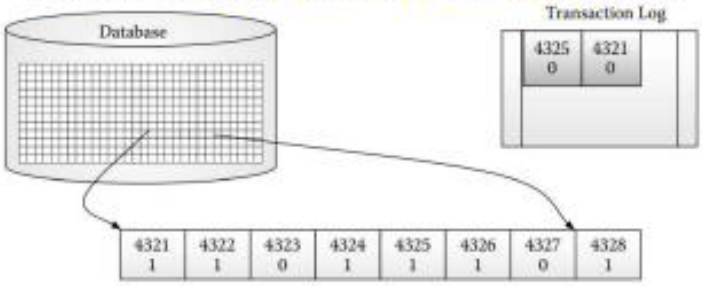


Figure: 4: *Recovered DB files and Tr. logs*

- 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

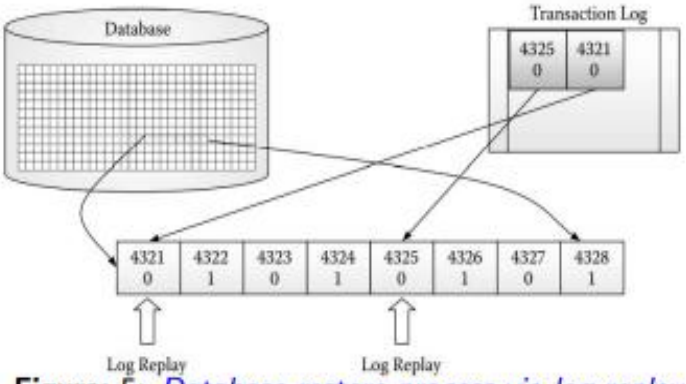


Figure: 5: *Database restore process via log replay*

- 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

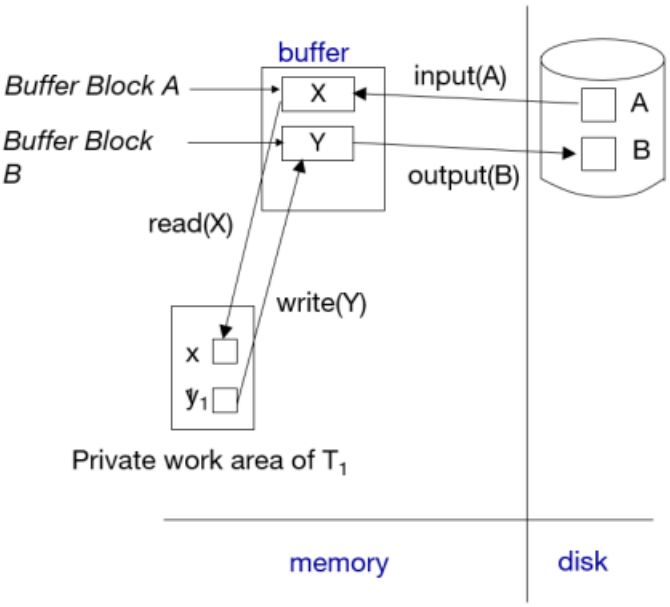
Recovery Schemes

- **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

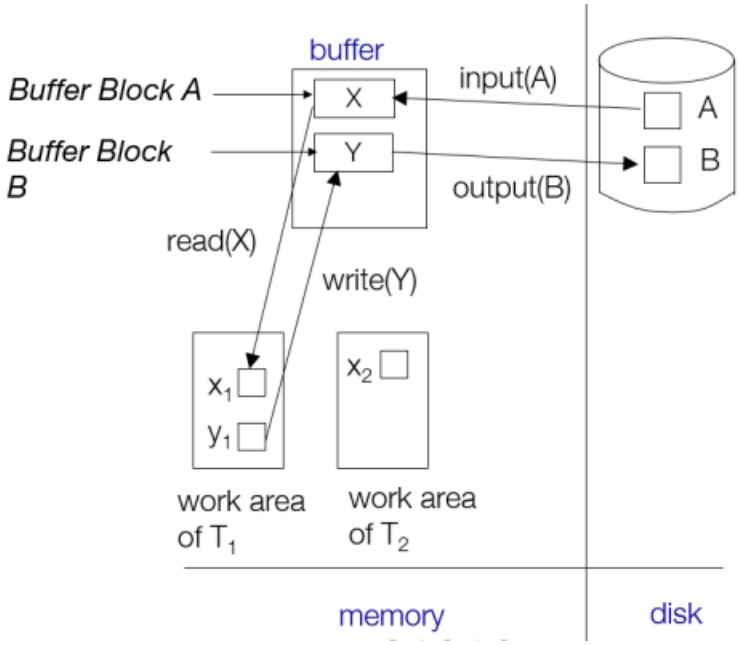
Concurrency Control and Recovery

- 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

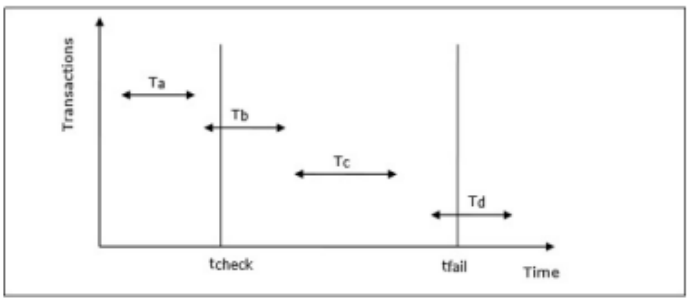
Recovery Algorithm (2)

- **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**>

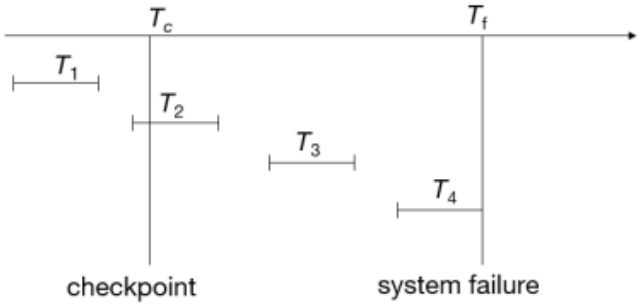
Recovery Algorithm (3): Checkpoints Recap

- 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

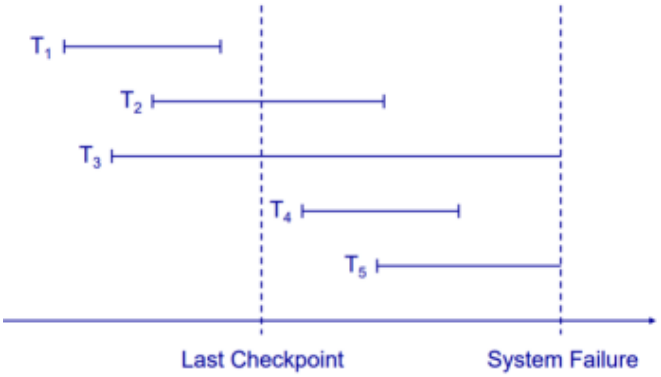
- **Recovery from failure**: Two phases
  - **Redo phase:** Replay updates of all transactions, whether they committed, aborted, or are incomplete
  - **Undo phase:** Undo phase: Undo all incomplete transactions

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

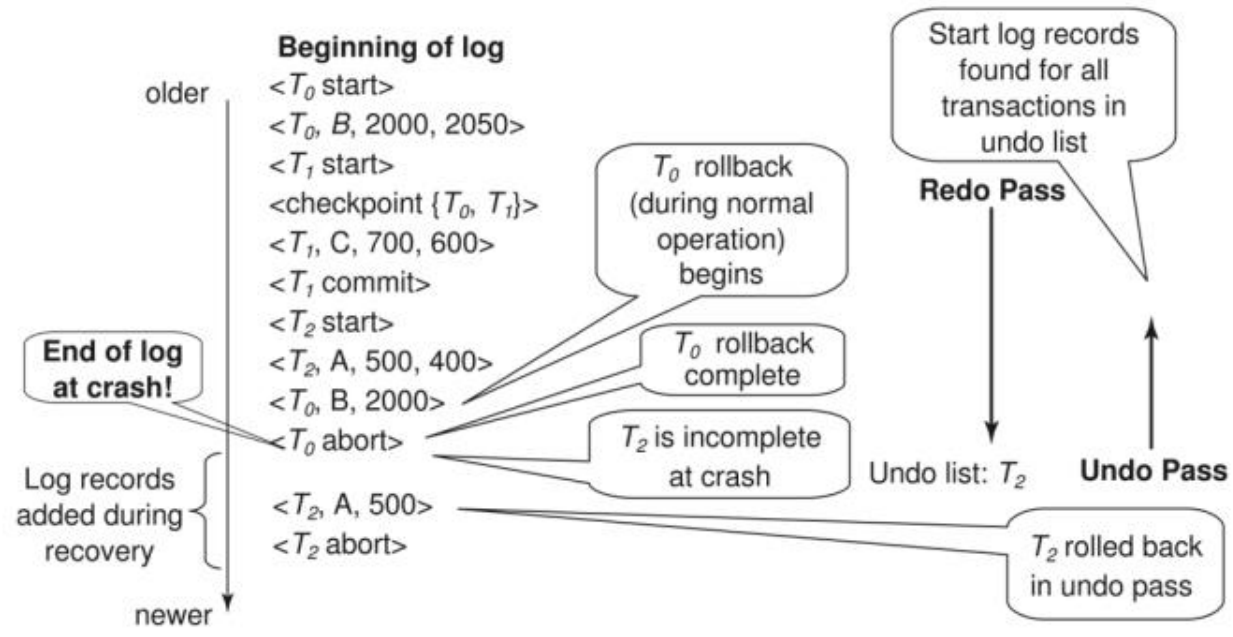
- Find last < **checkpoint** L> record, and set undo-list to L
- Scan forward from above < **checkpoin**t L> record
  - Whenever a record < Ti , Xj , V1, V2 > is found, redo it by writing V2 to Xj
  - Whenever a log record < Ti s**tar**t> is found, add Ti to undo-list
  - Whenever a log record < Ti **commi**t> 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

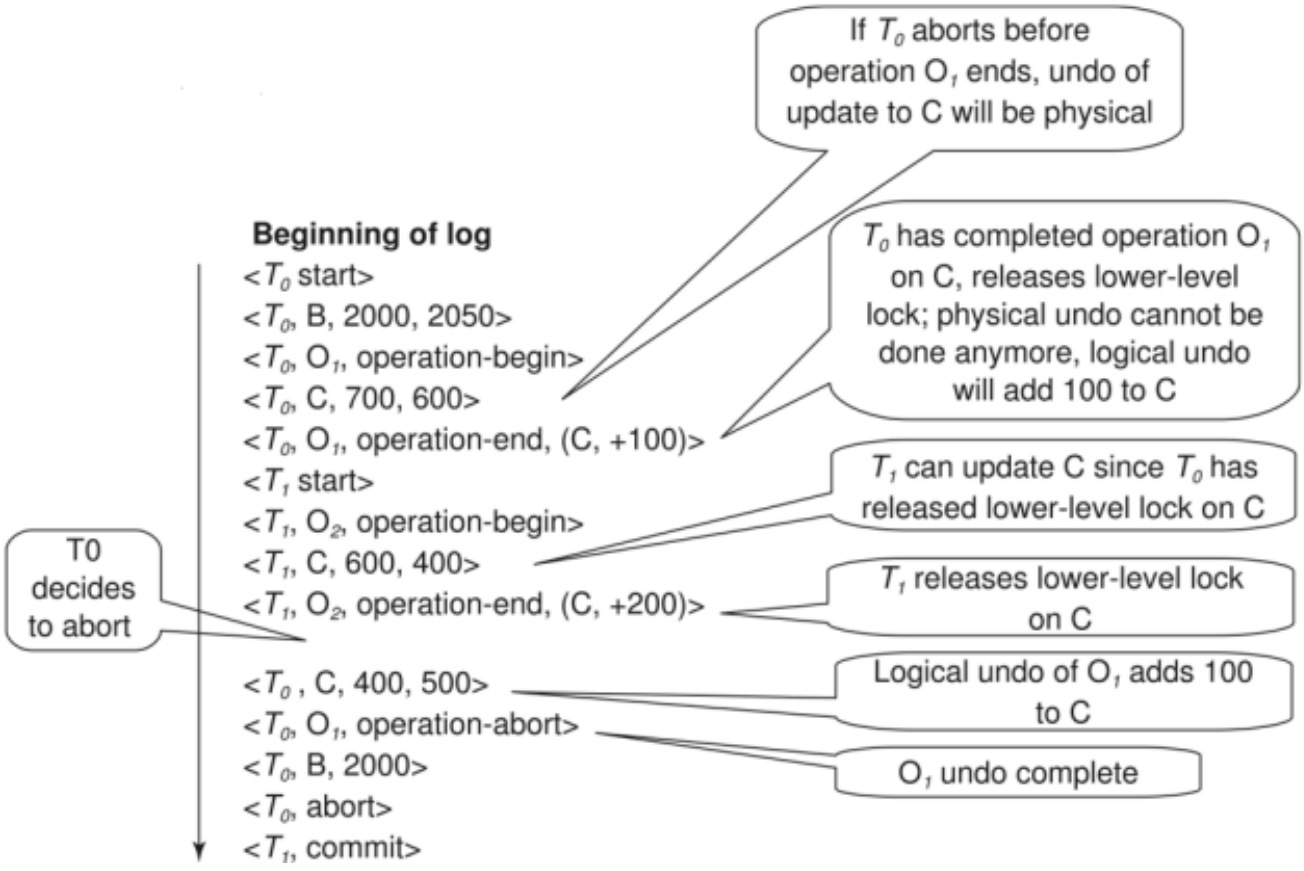
Rollback of transaction Ti , scan the log backwards

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

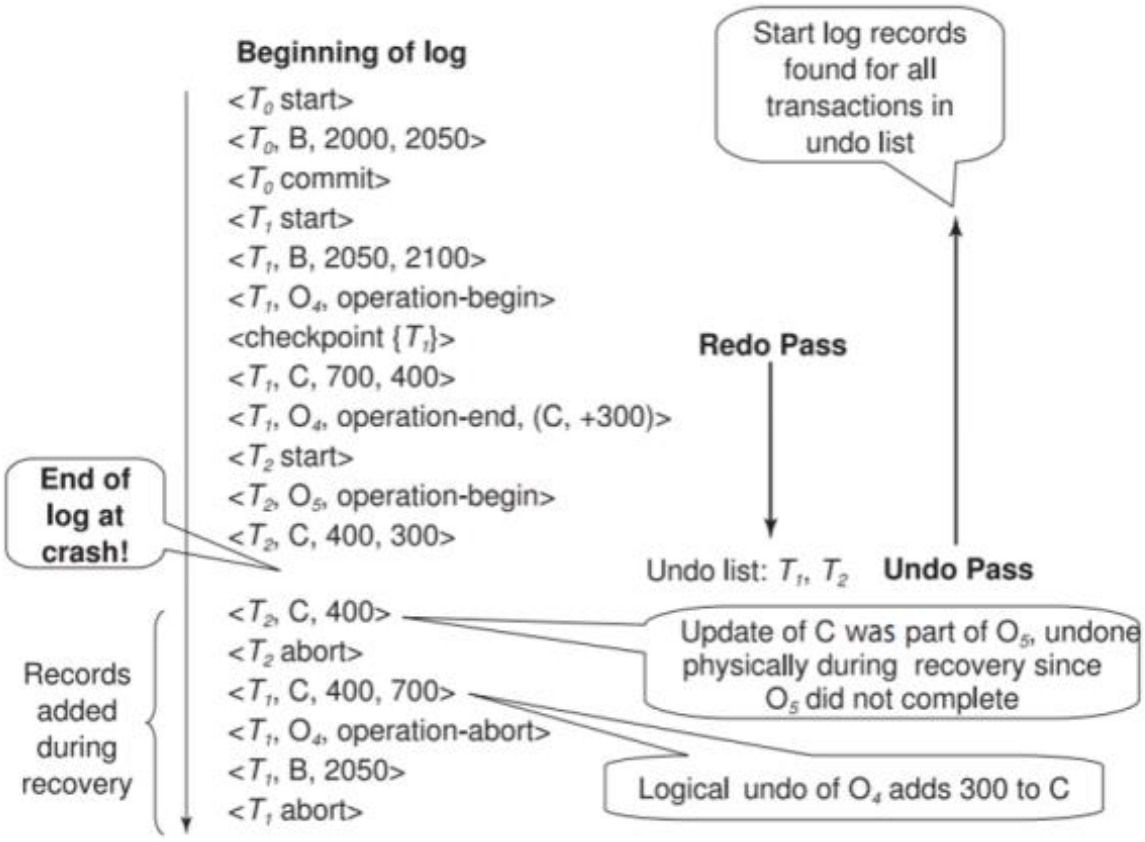
c) If a redo-only record is found ignore it
d) If a < Ti , Oj , **operation-abort** > record is found: skip all preceding log records for Ti until the record < Ti , Oj , **operation-begin** >is found

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

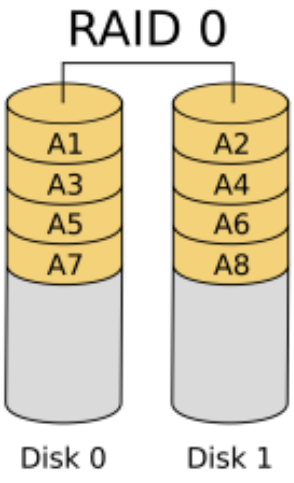


Image source: Standard RAID levels (Accessed 19-Aug-2021)

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

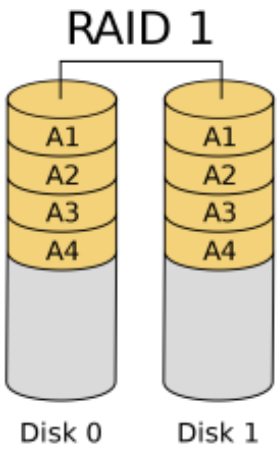


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

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

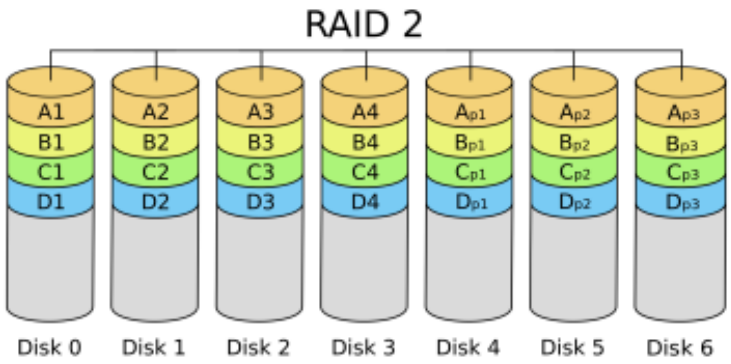


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- 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

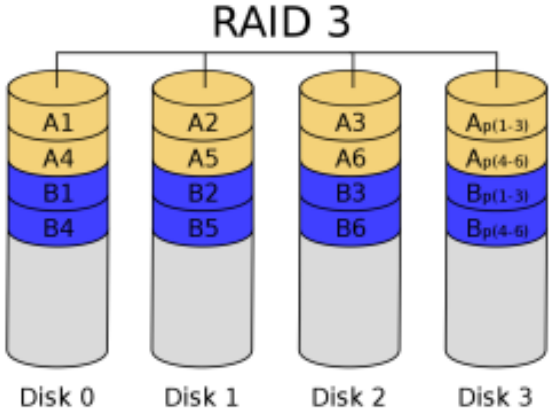


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- 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

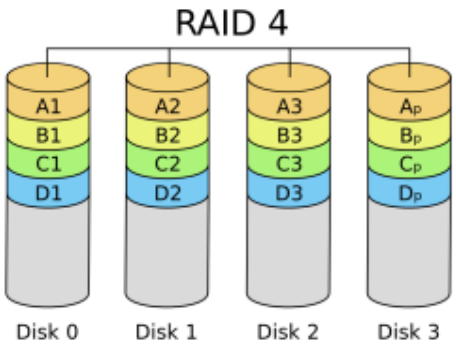


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- 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

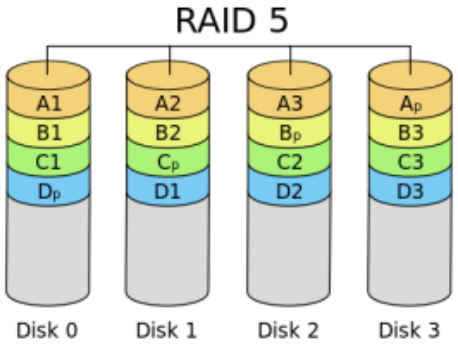


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- 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

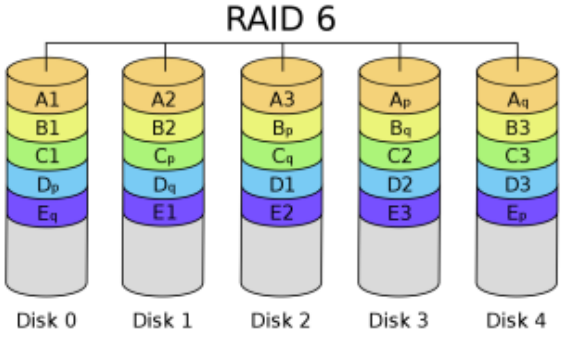


Image source: Standard RAID levels
(Accessed 19-Aug-2021)

- 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

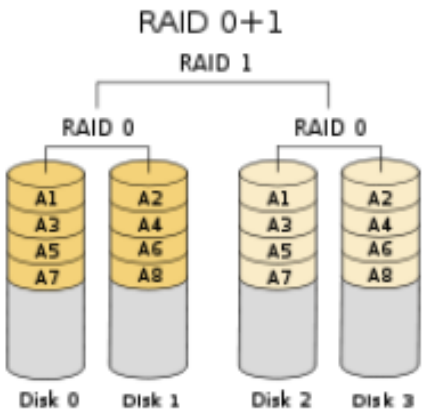


Image source: Nested RAID levels
(Accessed 23-Aug-2021)

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

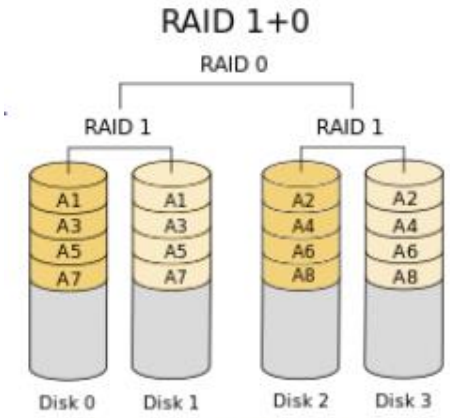


Image source: Nested RAID levels
(Accessed 23-Aug-2021)

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

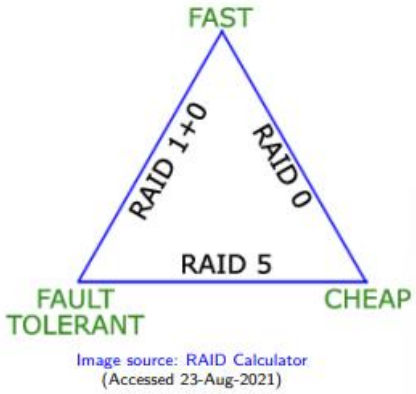


Image source: RAID Calculator (Accessed 23-Aug-2021)

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

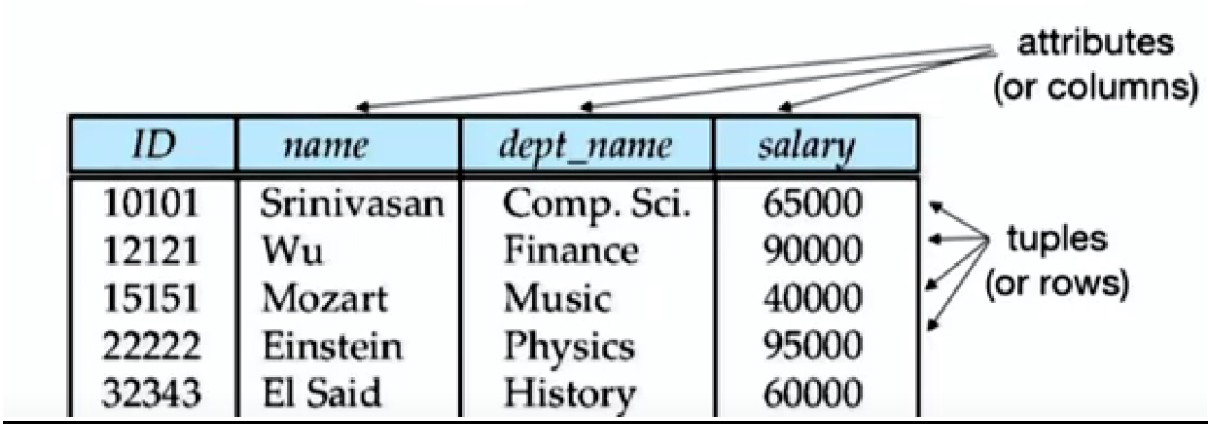
- RAID does not replace backups: Nothing can replace a well planned and frequently tested backup implementation!
- RAID does not protect against data corruption, human error, or security issues: While it can protect you against a drive failure, there are innumerable reasons for keeping backups. So RAID 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

DBMS

WEEK 2

LEC 1: Introduction to Relational Model (Pt.1)



- Attribute
  - Attribute Types:
    - Consider
      Students = Roll #, First Name, Last Name, DoB, Passport #, Aadhaar #, Department relation
    - The set of allowed values for each attribute is called the **domain** of the attribute
      - *Roll #:* Alphanumeric string
      - *First Name, Last Name*: Alpha String
      - *DoB*: Date
      - *Passport #*: String (Letter followed by 7 digits) – **nullable** (optional)
      - *Aadhaar #*: 12-digit number
      - *Department*: Alpha String
    - Attribute values are (normally) required to be **atomic;** that is indivisible
    - The special value **null** is a member of every domain. This indicates that the value is *unknown*
    - The null value may cause complications in the definition of many operations
- Schema and instance
  - Relation Schema and Instance
    - A_1, A_2, …, A_n are attributes
    - R = (A_1, A_2, …, A_n) is a *relation schema.* Example: *instructor* = (*ID, name, dept_name, salary*)

Formally, given sets D_1, D_2, …D_n a **relation** *r* is a subset of D_1 x D_2 x … x D_n. Thus, a relation is a set of *n*-tuples (a_1, a_2, …, a_n) where each $a_i \in D_i$

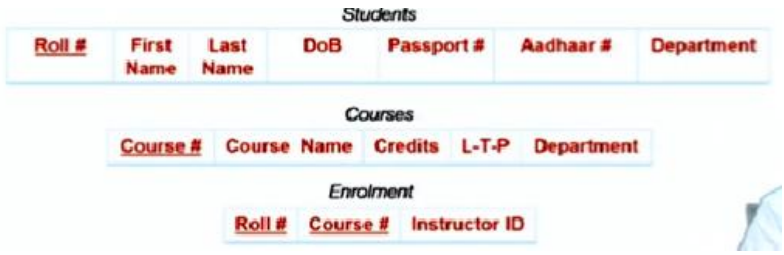
- The current values (**relation instance**) of a relation are specified by a table
- An element *t* of *r* is a tuple, represented by a row in a table

Example: *instructor* ≡ (String (5) x *String* x *String* x *Number*+), where *ID* ∈ *String*, *name* ∈ *String*, *dept_name* ∈ *String*, and *salary* ∈ *Number*+

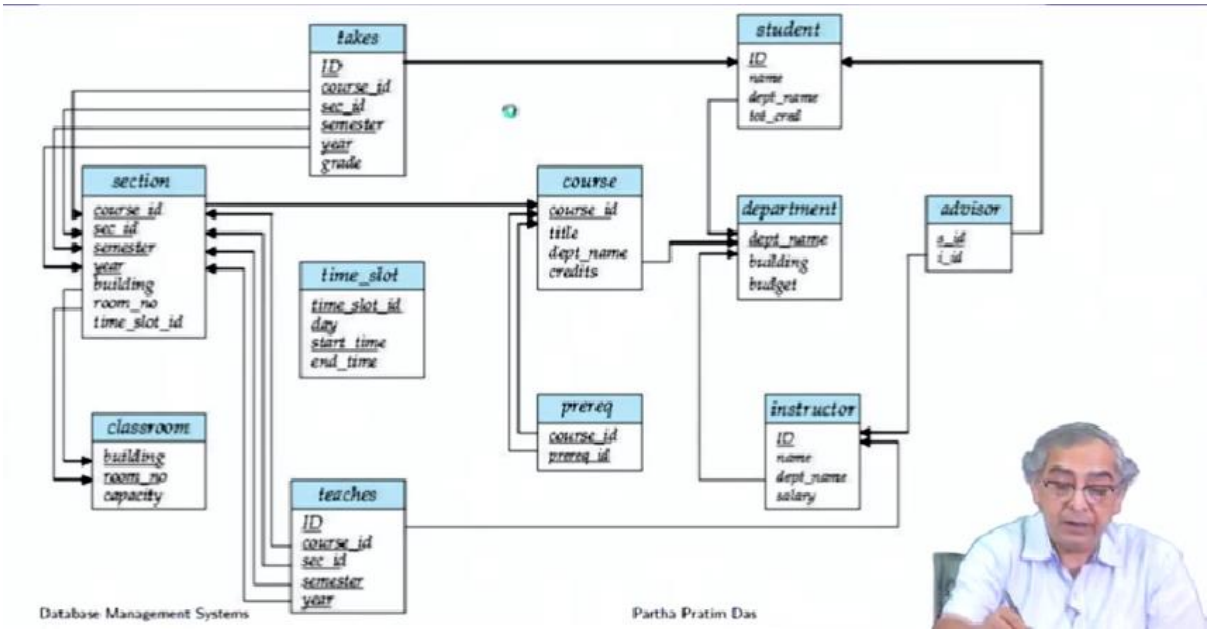
- Relations are unordered with unique tuples
  - Order of tuples/rows is irrelevant (tuples may be stored in arbitrary order)
  - No two tuples/rows may be identical

- Keys
  - Let K ⊆ R, where R is the set of attributes in the relation
  - K is the **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation *r*(R)
    - Example: {*ID*} and {*ID*, *name*} are both superkeys of instructor
  - Superkey K is a **candidate key** if K is minimal
    - Example: {*ID*} is a candidate key for *instructor*
  - One of the candidate keys is selected to be the **primary key**
  - A **surrogate key** (or synthetic key) is a database is a unique identifier for either an *entity* in modeled world or an *object* in the database
    - *Not* derived from application data, unlike a *natural* (or *business*) key which is derived from application data
  - Example:
    - *Students* = *Roll #*, *First Name*, *Last Name*, *DoB*, *Passport #*, *Aadhaar #*, *Department*
    - **Super Key:** Roll #, {Roll #, DoB}
    - **Candidate Keys:** Roll #, {First Name, Last Name}, Aadhaar #
      - Passport # cannot be a key as it is nullable
    - **Primary Key:** Roll #
      - Can Aadhaar # be a primary key? It may suffice for unique identification. But Roll # may have additional useful info
    - **Secondary Key/ Alternate Key:** {First Name, Last Name}, Aadhar #
    - **Simple Key:** Consists of a *single attribute***.**
    - **Composite Key:** {First Name, Last Name}
      - Consists of more than one attribute to uniquely identify an entity occurrence
      - One or more of the attributes, which make up the key are not simple keys in their own right
    - **Foreign Key constraint:** Value in one relation must appear in another
      - **Referencing** relation
        - Enrolment: Foreign keys – Roll #, Course #

- **Referenced** relation
    - Students, Courses
- A **compound key** consists of *more than one attribute* to uniquely identify an entity occurrence
    - Each attribute, which makes up the key, is a simple key in its own right
    - {Roll #, Course #}



- Schema Diagram for university database



- Relational Query Languages
    - Procedural viz-a-viz Non-procedural or Declarative paradigms
        - **Procedural programming** requires that the programmer tell the computer what to do
            - That is, *how* to get the output for the range of required inputs
            - The programmer must know an appropriate algorithm
        - **Declarative programming** requires a more descriptive style
            - The programmer must know *what* relationships hold between various entities
    - "Pure" languages:
        - Relational algebra
        - Tuple relational calculus

        - Domain relational calculus
    - All 3 pure languages are equivalent in computing power

LEC 2: Introduction to Relational Model (Pt.2)

- Relational Operators
    - Basic Properties of Relations
        - A relation is set. Hence,
            - Ordering of rows/tuples is inconsequential

A	B
a1	b1
a1	b2
a2	b1
a2	b2

is same as:

A	B
a1	b1
a2	b1
a2	b2
a1	b2

            - All rows/tuples must be distinct

A	B
a1	b1
a1	b2
a1	b2
a1	b1

is not valid

A	B
a1	b1
a1	b2

is

    - Select Operation – selection of rows (tuples)
        - Relation *r*

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

            - $\sigma_{A=B \wedge D>5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

- Project Operation – selection of columns (Attributes)
  - Relation *r*

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

  - $\pi_{A, C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

- Union of two relations
  - Relation *r*, *s*

A	B
α	1
α	2
β	1

r

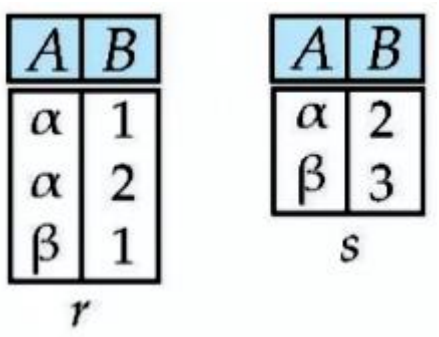
A	B
α	2
β	3

s

  - $r \cup s$

A	B
α	1
α	2
β	1
β	3

- Set difference of two relations
  - Relation *r*, *s*



A	B
α	1
α	2
β	1

r

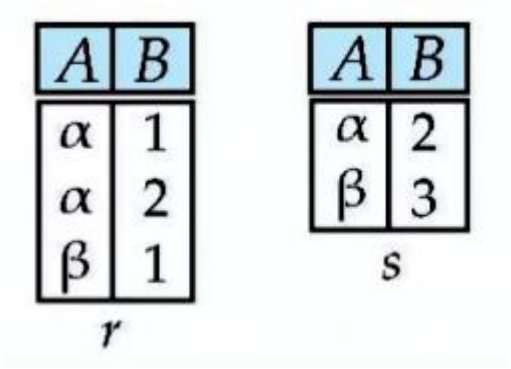
A	B
α	2
β	3

s

  - *r* – *s*

A	B
α	1
β	1

- Set intersection of two relations
  - Relation *r*, *s*



A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

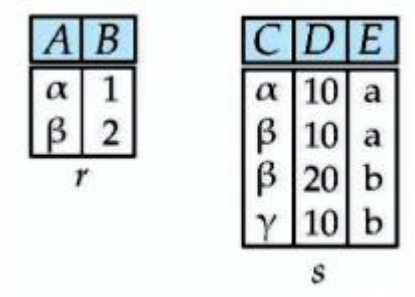
s

  - *r* ∩ *s*

A	B
α	2

- $r \cap s = r - (r - s)$

- Joining 2 relations – Cartesian Product
  - Relation *r*, *s*



A	B
α	1
β	2

r

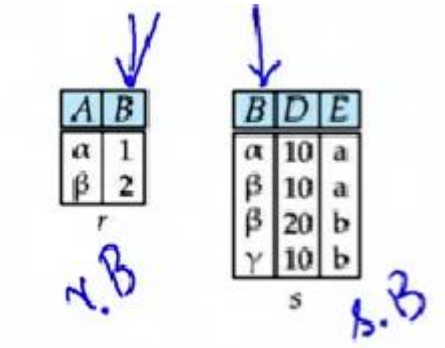
C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

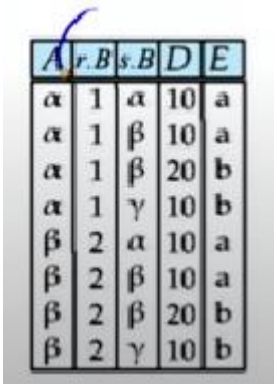
  - *r* x *s*

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

  - $r = \Pi_R (r \times s)$; $s = \Pi_S (r \times s)$ where R = (A, B) and S = (C, D, E)
  - Naming issue: Between the 2 relations, there are one or more columns with the same name
    - Name them r.B and s.B (B is the repeated column)



    - *r* x *s*



A	r.B	s.B	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- Renaming a table
  - Allows us to refer to a relation (say E) by more than one name ρ_X (E) returns the expression *E* under the name *X*
  - Relation *r*

A	B
α	1
β	2

r

  - *r* X ρ_s (*r*)

r.A	r.B	s.A	s.B
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2

- Composition of operations
  - Can build expressions using multiple operations
  - E.g.: $\sigma_{A=C}$ (*r* X *s*)
    - *r* X *s*

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

    - $\sigma_{A=C}$ (*r* X *s*)

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

- Joining two relations – **Natural join**
  - Let *r* and *s* be relations on schemas *R* and *S* respectively. Then, the natural join of relations *R* and *S* is a relation on schema *R* U *S* obtained as follows:
    - Consider each pair of tuples t_r from *r* and t_s from *s*
    - If t_r and t_s have the same value on each of the attributes in R ∩ S, add a tuple *t* to the result, where
      - *t* has the same value as t_r on *r*
      - *t* has the same value as t_s on *s*
  - Example:

- Relations *r*, *s*:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- Natural Join: $r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

$$\Pi_{A,r.B,C,r.D,E} (\sigma_{r.B=s.B \wedge r.D=s.D} (r \times s))$$

- Aggregate Operators
  - Can we compute:
    - SUM
      - E.g.: $SUM_B (\sigma_{B > 2}(r))$
    - AVG
    - MAX
    - MIN
- Notes about Relational Languages
  - Each query input is a table (or set of tables)
  - Each query output is a table
  - All data in the output table appears in one of the input tables
  - Relational algebra is not Turing complete (a system is called Turing complete if this system is able to recognize or decide other data-manipulation rule sets)
- Summary of Relational Operators

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{salary >= 85000}(instructor)$
	Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID, salary}(instructor)$
	Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
x (Cartesian Product)	*instructor* x *department*
	Output all possible combinations of rows in *instructor* and *department*.
∪ (Union)	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$
	Output the union of tuples from the two input relations.
- (Set Difference)	$\Pi_{name}(instructor) - \Pi_{name}(student)$
	Output the set difference of tuples from the two input relations.
⋈ (Natural Join)	*instructor* ⋈ *department*
	Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.

LEC 3: Introduction to SQL (Pt.1)

- History of Query Language
  - IBM developed *Structured English Query Language* (SEQUEL) as a part of System R project. Renamed **Structured Query Language** (SQL)
  - There aren't any alternatives to SQL for speaking to relational databases (that is, SQL as a protocol), but there are many alternatives to writing SQL in the applications
  - These alternatives have been implemented in the form of frontends for working with relational databases. Examples:
    - SchemeQL and CLSQL, which are probably the most flexible, owing to their Lisp heritage, they look a lot more like SQL than other frontends
    - LINQ (in .Net)
    - ScalaQL and ScalaQuery (in Scala)
    - SqlStatement, ActiveRecord and many others in Ruby
    - HaskellDB
    - …
  - There are several query languages that are derived or inspired from SQL. Of them most effective is SPARQL (SPARQL Protocol and RDF Query Language)
    - RDL – Resource Description Language
    - Standardized by W3C Consortium as key technology of semantic web
    - Versions:
      - SPARQL 1.0
      - SPARQL 1.1

        - Used as query languages for several NoSQL systems – particularly the Graph Databases that use RDF as store

- Data Definition Language (DDL)
    - SQL DDL allows the specification of info about relations, including:
        - The *Schema* for each relation
        - The *Domain* of values associated with each attribute
        - *Integrity Constraints*
        - Also,
            - The set of Indices to be maintained for each relation
            - Security and Authorization information for each relation
            - The Physical Storage Structure of each relation on disk
    - Domain Types in SQL
        - **char(*n*) –** Fixed length character string, with user-specified length *n*
        - **varchar(*n*) –** Variable length character strings, with user-specified maximum length *n*
        - **int –** Integer (a finite subset of the integers that is machine-independent)
        - **smallint(*n*) –** Small integer (a machine independent subset of the integer domain type)
        - **numeric(*p,d*) –** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric (**3,1**)** allows 44.5 to be stored exactly, but not 444.5 or 0.32)
        - **real, double precision –** Floating point and double-precision floating point numbers, with machine-dependent precision
        - **float(*n*) –** Floating point number, with user-specified precision of at least *n* digits
    - Create Table Construct
        - An SQL relation is defined using the **create table** command:

            create table *r* (A_1D_1, A_2D_2, …, A_nD_n),
            (integrity-constraint$_1$),
            …
            (integrity-constraint$_k$));

            - *r* is the name of the relation
            - each A_i is an attribute name in the schema of relation *r*
            - D_i is the data type of values in the domain of attribute A_i

            create table *instructor* (

            ID **char(**5**),**

            name **varchar(**20**),**

dept_name **varchar(**20**),**

salary **numeric(**8,2**));**

- Create table construct – Integrity constraints
  - not null
  - primary key (A_1, …, A_m)
  - foreign key (A_m, …, A_n) references *r*

create table *instructor* (

ID **char(**5**),**

name **varchar(**20**) not null,**

dept_name **varchar(**20**),**

salary **numeric(**8,2**),**

primary key (*ID***),**

foreign key (*dept_name***) references** *department*));

  - **primary key** declaration on an attribute automatically ensures **not null**
- Update tables
  - **Insert (**DML command**)**
    - **insert into** *instructor* **values (**'10211', 'Smith', 'Biology', 66000**);**
  - **Delete (**DML command**)**
    - Remove all tuples from the student relation
      - **delete from** *student*
  - **Drop Table (**DDL command**)**
    - **drop table** *r*
  - **Alter (**DDL command**)**
    - **alter table** *r* **add** *A D*
      - Where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*
      - All existing tuples in the relation are assigned *null* as the value for the new attribute
    - **alter table** *r* **drop** *A*
      - Where *A* is the name of an attribute of relation *r*
      - Dropping of attributes not supported by many databases
- Data Manipulation Language (DML): Query Structure
  - Basic Query Structure

- A typical SQL query has the form:

  select A_1, A_2, …, A_n ,
  from r_1, r_2, …, r_m
  where *P*
  - A_i represents an attribute from r_i 's
  - r_i represents a relation
  - *P* is a predicate

The result of an SQL query is a relation

- Select Clause
  - The **select** clause lists the attributes desired in the result of a query
    - Corresponds to the projection operation of the relational algebra
  - Example: find the names of all instructors:

    select *name*,

    from *instructor*

  - NOTE: SQL names are case insensitive
    - *Name* ≡ *NAME* ≡ *name*
  - SQL allows duplicates in relations as well as in query results
  - To force the elimination of duplicates, insert the keyword distinct after select
    - To find the dept names of all instructors and remove duplicates:
      - **select distinct** *dept_name*
        from *instructor*
  - The keyword **all** specifies that duplicates should not be removed
    - **select all** *dept_name*
      from *instructor*
  - An asterisk in the select cause denotes *all attributes*
    - **select ***
      from *instructor*
  - An attribute can be a literal with no **from** clause
    - **select** '437'
    - Results is a table with one column and a single row with value '437'
    - Can give the column a name using:
      - **select** 'A' **as** FOO
  - An attribute can be a literal with **from** clause

- **select** 'A'
  from *instructor*

    - Result is a table with one column and N rows (number of tuples in the instructor table
- The select clause can also contain arithmetic expressions involving the operation, +, -, * and / and operating on constants and attributes of tuples
    - The query:
        - **select** *ID, name, salary/12*
          from *instructor*
    - Would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12 (will be replaced)
    - Can rename "*salary/12*" using the **as** clause:
        - select *ID, name, salary/12* **as** *monthly_salary*

- Where clause
    - Specifies conditions that the result must satisfy
        - Corresponds to the selection predicate of the relational algebra
    - To find all instructors in Comp. Sci. dept
        - **select** *name*
          from *instructor*
          where *dept_name* = 'Comp. Sci.'
    - Comparison results can be combined using the logical connectives **and, or** and **not**
    - Comparisons can be applied to results of arithmetic expressions
- From clause
    - The **from** clause lists the relations involved in the query
        - Corresponds to the Cartesian product operation of the relational algebra
    - Find the Cartesian product *instructor* X *teaches*
        - **select ***
          from *instructor, teaches*
        - Generates every possible instructor-teaches pair, with all attributes from both relations
        - For common attributes (for example, *ID*), the attributes in the resulting table are renamed using the relation's name (for example, *instructor.ID*)
    - Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

LEC 4: Introduction to SQL (Pt.2)

- Additional Basic Operations
  - Cartesian Product
  - Rename AS Operation
    - The SQL allows renaming relations and attributes using the **as** clause:
      old_name **as** *new_name*
    - Keyword as is optional and may be omitted
  - String Operations
    - SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
      - percent (%) The % character matches any substring
      - underscore (_) The _ character matches any character
    - Find the names of all instructors whose name includes the substring "dar"

      select *name*

      from *instructor*

      where *name* **like** '*%dar%*'

    - Match the string "100%"
      like '100%' **escape** '\'

      in that above we use backlash (\) as the escape character

    - Patterns are case sensitive
    - Examples:
      - 'Intro%' matches any string beginning with "Intro"
      - '%Comp%' matches any string containing "Comp" as a substring
      - '_ _ _' matches any string of exactly 3 characters
      - '_ _ _ %' matches any string of at least 3 characters
    - SQL supports a variety of string operations such as
      - Concatenation (using "||")
      - Converting from upper to lower case (and vice versa)
      - Finding string length, extracting substrings, etc
  - Ordering the Display of Tuples
    - We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default
    - Can sort on multiple attributes
  - Selecting Number of tuples in output
    - The **Select Top** clause is used to specify the number of records to return

        - The **Select Top** clause is useful on large tables with thousands of records. Returning a large number of records can impact performance

          select top 10 **distinct** *name*
          from *instructor*
        - Not all database systems support the SELECT TOP clause
            - SQL Server & MS Access support **select top**
            - MySQL supports the **limit** clause
            - Oracle uses **fetch first** *n* **rows only** and **rownum**

              select distinct *name*
              from *instructor*
              order by *name*
              fetch first 10 **rows only**
    - Where Clause Predicates
        - SQL includes a **between** comparison operator
        - Tuple comparison
        - In Operator
            - The **in** operator allows user to specify multiple values in a **where** clause
            - The **in** operator is a shorthand for multiple **or** conditions
    - Duplicates
        - In relations with duplicates, SQL can define how many copies of tuples appear in the result
        - **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:
            - $\sigma_\theta(r_1)$: If there are c_1 copies of tuple t_1 in r_1 and t_1 satisfies selections σ_θ, then there are copies of t_1 in $\sigma_\theta(r_1)$
            - $\Pi_A(r)$: For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1
            - $r_1 \times r_2$: If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of tuple $t_1.t_2$ in $r_1 \times r_2$

LEC 5: Introduction to SQL (Pt.3)

- Set Operations
    - Set operations **union, intersect** and **except**
        - Each of the above operations automatically eliminates duplicates
    - To retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**
    - Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then it occurs:
        - *m* + *n* times in *r* **union all** *s*
        - min (*m, n*) times in *r* **intersect all** *s*

        - max (0, *m* - *n*) times in *r* **except all** *s*
- Null Values
    - It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
    - *null* signifies an unknown value or that a value doesn't exist
    - The result of any arithmetic expressions involving *null* is *null*
    - The predicate **is null** can be used to check for null values
    - It is not possible to test for **null** values with comparison operators, such as =, <, or <>. We need to use the **is null** and **is not null** operators instead
- Three Valued Logic
    - Three values – **true, false, unknown**
    - Any comparison with *null* returns *unknown*
        - E.g: 5 < null or null <> null or null = null return unknown
    - Three-valued logic using the value unknown:
        - **OR**: (unknown or true) = true
          (unknown or false) = unknown
          (unknown or unknown) = unknown
        - **AND**: (true and unknown) = unknown
          (false and unknown) = false
          (unknown and unknown) = unknown
        - **NOT**: (not unknown) = unknown
        - **"***P* is unknown" evaluates to true if predicate *P* evaluates to unknown
    - Result of **where** clause predicate is treated as false if it evaluates to unknown
- Aggregate Functions
    - These functions operate on the multiset of values of a column of a relation and return a value
        - **avg**
        - **min**
        - **max**
        - **sum**
        - **count**
    - Group By
        - Attributes in **select** clause outside of aggregate functions must appear in **group by** list
            - /* erroneous query */
              select dept_name, ID, avg(salary)
              from instructor
              group by dept_name;
    - Having clause
        - Properties are specified using having clause (eg: having avg(salary) > 42000)

        - Predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups
- Null Values and Aggregates
    - Total all salaries

      select sum(salary)
      from instructor;
        - Above statement ignores null amounts
        - Result in *null* if there is no non-null amount
    - All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
    - What if collection has only null values?
        - Count returns 0
        - All other aggregates return null

DBMS

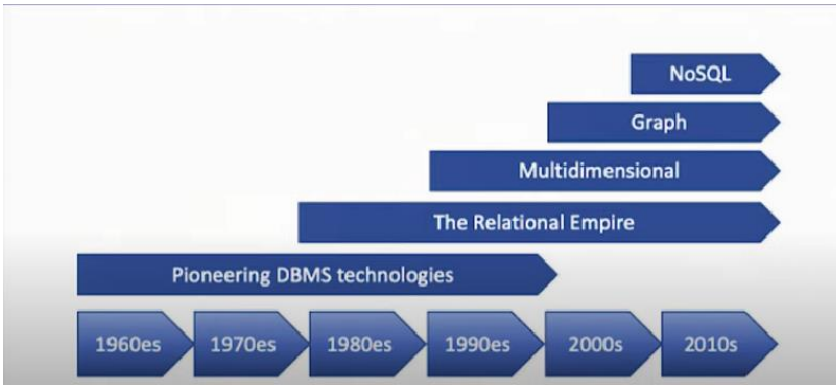
WEEK 1

LEC 1: Course Overview

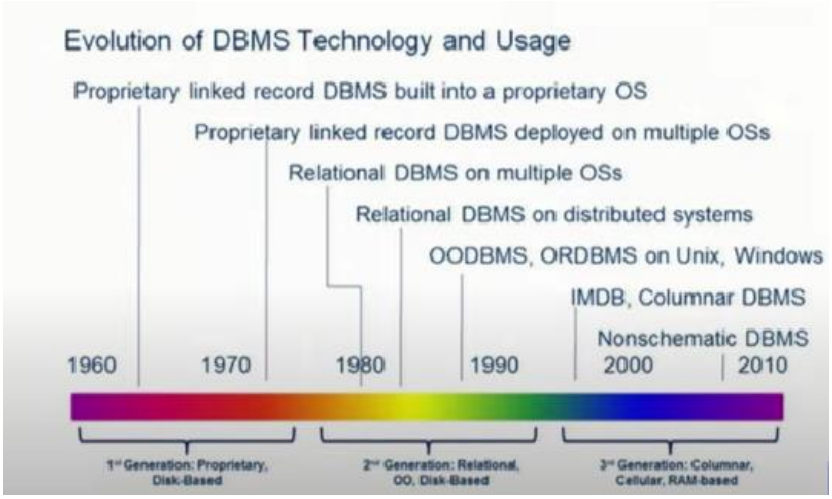
- Database Management System (DBMS)
  - DBMS contains information about a particular enterprise
    - Collection of interrelated data
    - Set of programs to access the data
    - An environment that is both *convenient* and *efficient* to use
  - Database Applications:
    - Banking: transactions
    - Airlines: reservations, schedules
    - Universities: registration, grades
    - Sales: customers, products, purchases
    - Online retailers: order tracking, customized recommendations
    - Manufacturing: production, inventory, orders, supply chain
    - Human resources: employee records, salaries, tax deductions
    - …
  - Databases can be very large
  - Databases touch all aspects of our lives
- University Database Example
  - Application program examples:
    - Add new students, instructors, and courses
    - Register students for courses and generate class rosters
    - Assign grades to students, compute graded point averages (GPA) and generate transcripts
  - In the early days, database applications were built directly on top of file systems
- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
    - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task
  - Data isolation
    - Multiple files and formats
  - Integrity problems
    - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones
  - Atomicity

LEC 2: Why DBMS? (Pt. 1)

- Data Management
  - Management of Data or Records is a basic need for human society:
    - Storage
    - Retrieval
    - Transaction
    - Audit
    - Archival
  - For
    - Individual
    - Small / Big Enterprise
    - Global
  - Two major approaches in this practice:
    - Physical
    - Electronic
- Data Management: Physical
  - Physical data or Records management is more formally known as *Book Keeping* has been physical ledgers and journals for centuries
- Data Management: Electronic
  - Electronic Data or Records management moves with the advances in technology – especially of memory, storage, computing, and networking.
- Electronic Data Management Parameters
  - Durability
  - Scalability
  - Security
  - Retrieval
  - Ease of Use
  - Consistency
  - Efficiency
  - Cost
  - …
- Evolution of Data Models



- Evolution of DB Technology

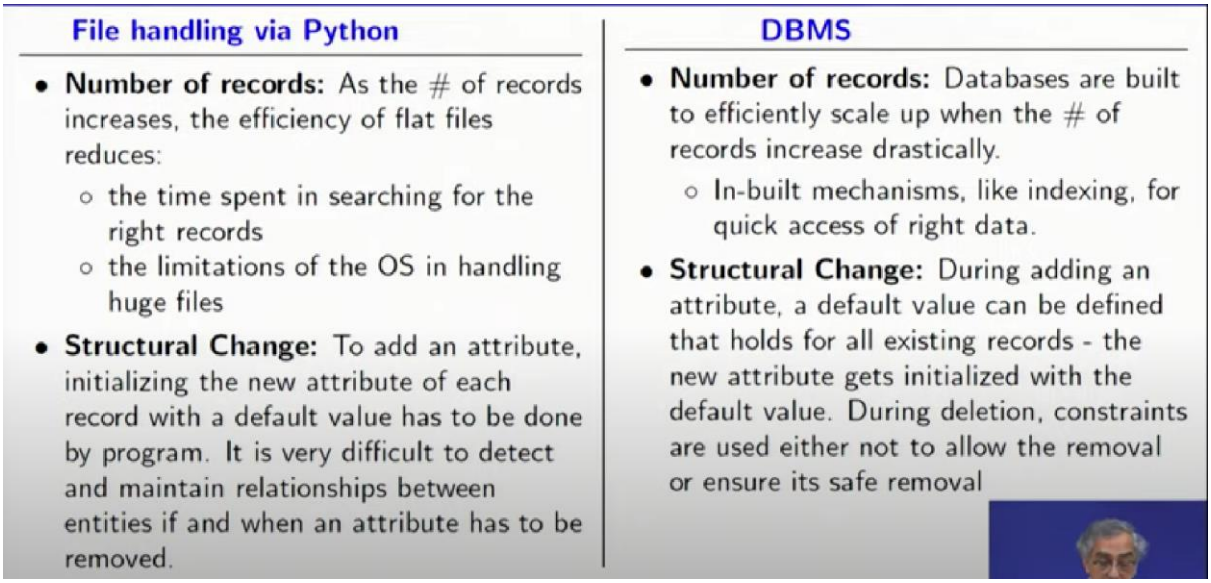


LEC 3: Why DBMS? (Pt. 2)

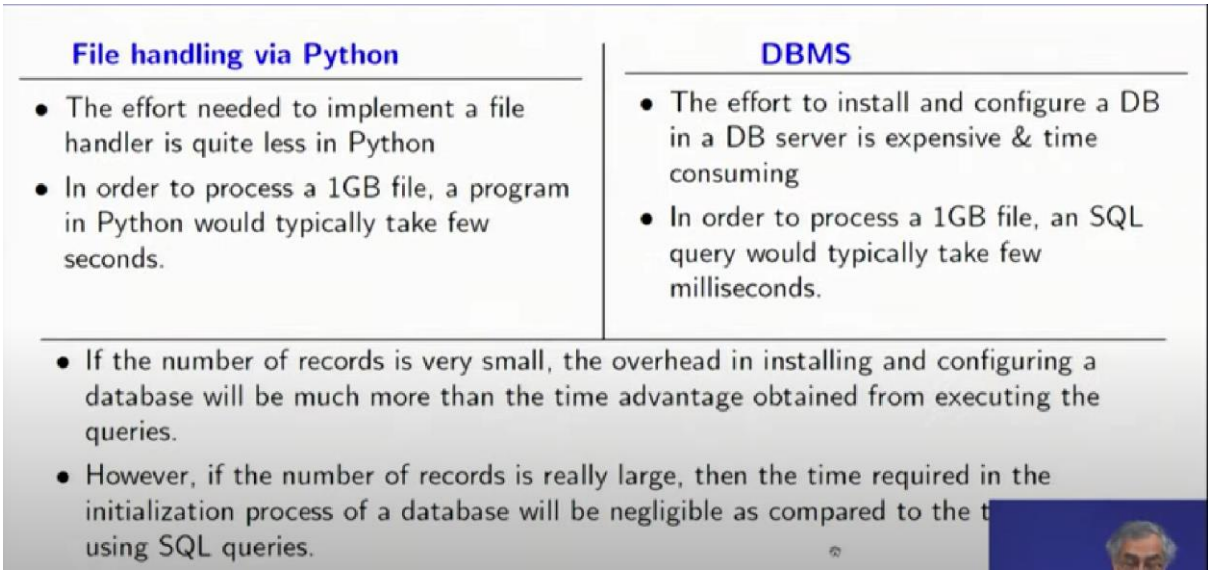
- File handling in Python v/s DBMS

Parameter	File Handling via Python	DBMS
Scalability with respect to amount of data	Very difficult to handle insert, update and querying of records	In-built features to provide high scalability for a large number of records
Scalability with respect to changes in structure	Extremely difficult to change the structure of records as in the case of adding or removing attributes	Adding or removing attributes can be done seamlessly using simple SQL queries
Time of execution	In seconds	In milliseconds
Persistence	Data processed using temporary data structures have to be manually updated to the file	Data persistence is ensured via automatic, system induced mechanisms
Robustness	Ensuring robustness of data has to be done manually	Backup, recovery and restore need minimum manual intervention
Security	Difficult to implement in Python (Security at OS level)	User-specific access at database level
Programmer's productivity	Most file access operations involve extensive coding to ensure persistence, robustness and security of data	Standard and simple built-in queries reduce the effort involved in coding thereby increasing a programmer's throughput
Arithmetic operations	Easy to do arithmetic computations	Limited set of arithmetic operations are available
Costs	Low costs for hardware, software and human resources	High costs for hardware, soft… resources

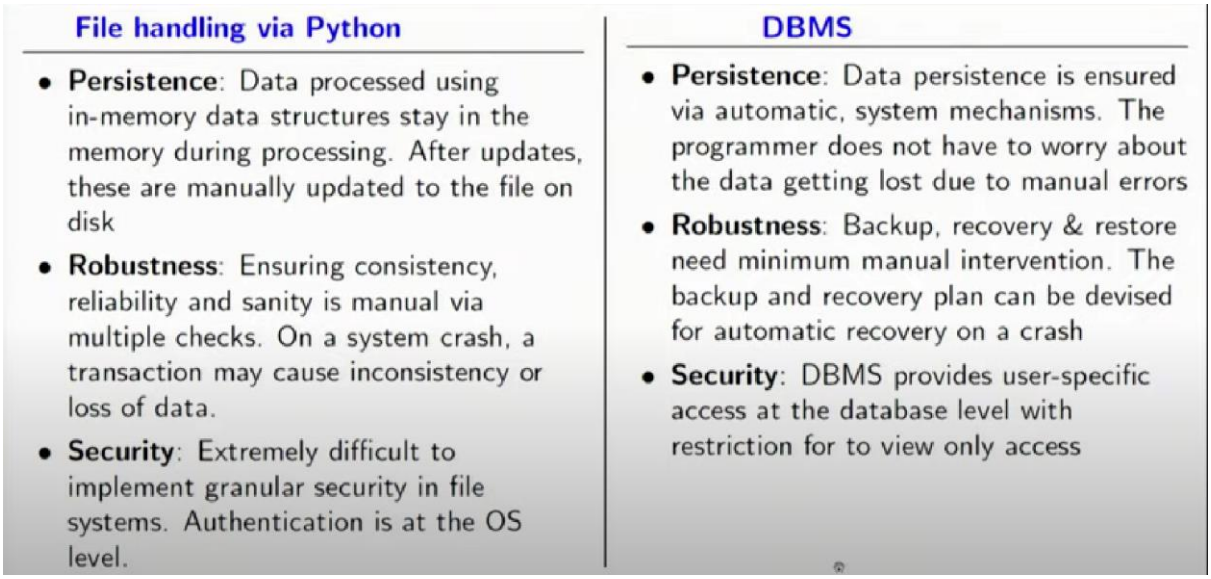
- Scalability



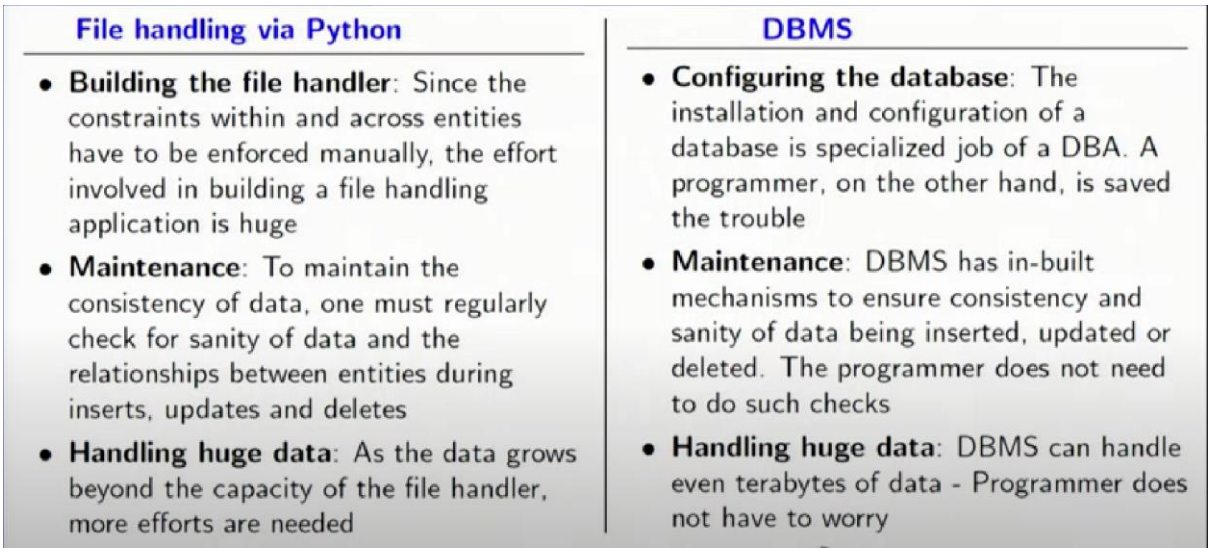
- Time and efficiency



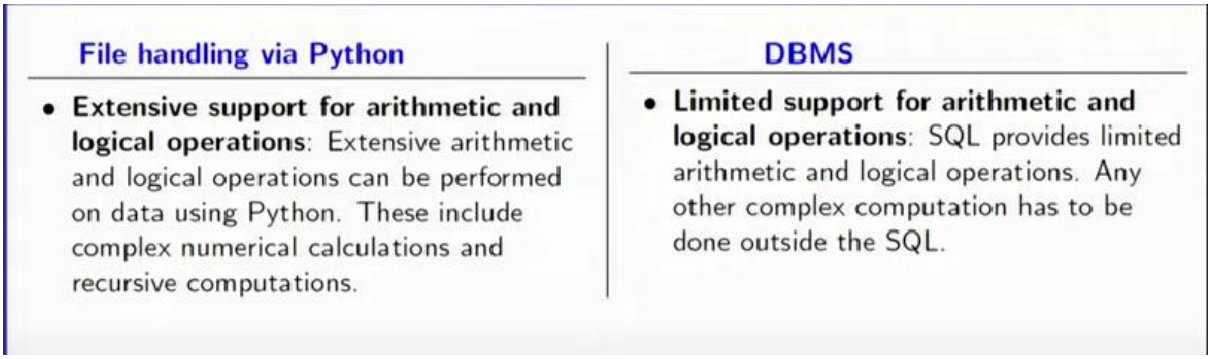
- Persistence, Robustness, Security



- Programmer's Productivity



- Arithmetic Operations



LEC 4: Intro to DBMS (Pt. 1)

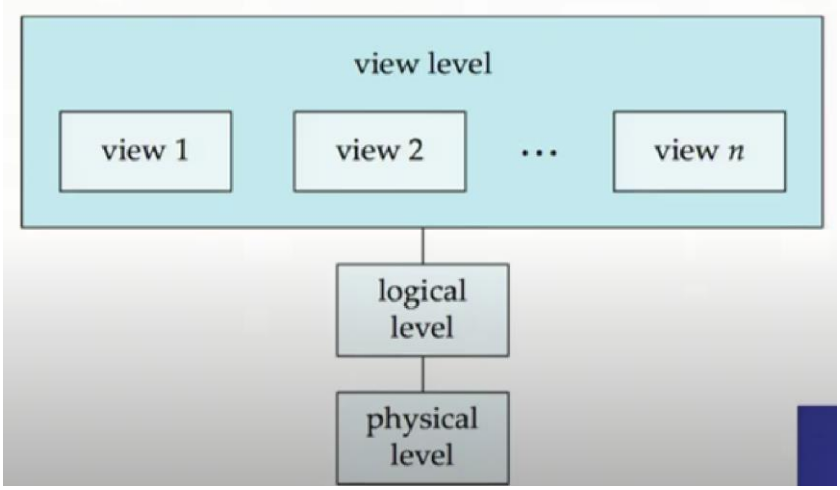
- Levels of Abstraction:

- **Physical Layer:** describes how a record is stored
- **Logical Level:** describes data stored in the database, and the relationships among the data fields
  - **Eg.;**

```
type instructor = record
    ID: string;
    name: string;
    dept_name: string;
    salary: integer;
end;
```

  - **View Level:** application programs hide details of data types
    - Views can also hide info (such as salary) for security purposes

- View of Data



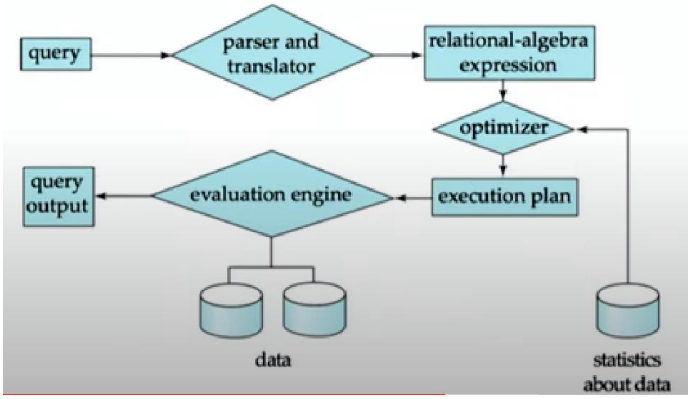
- Each upper level is independent of its lower levels. For e.g.: Logical level is physical level independent and Vie level is logical level independent.

LEC 5: Intro to DBMS (Pt. 2)

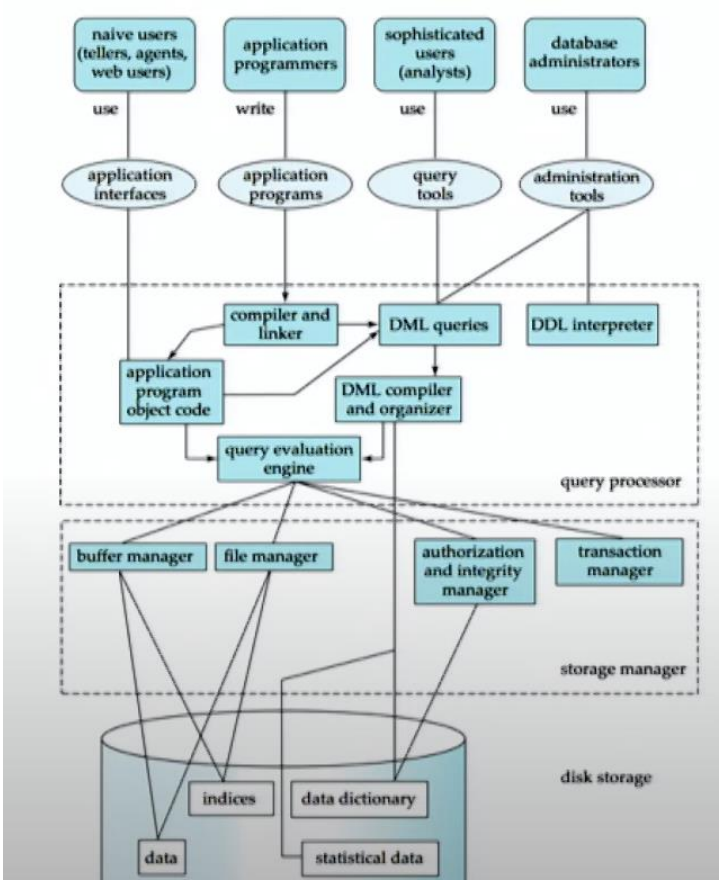
- Database Design
  - The process of designing the general structure of the database:
    - Logical Design
      - Deciding on the database schema.
      - Business decision
        - What attributes should we record in the database?
      - Computer Science decision

        - What relation schema should we have and how should the attributes be distributed among the various relation schemas?
    - Will not change with changes in Physical design
  - Physical Design
    - Deciding on the physical layout of the database
- Design Approaches
  - To ensure that each relation in the database is good
  - 2 ways:
    - Entity-Relationship model
      - Models an enterprise as a collection of entities and relationships
      - Represented diagrammatically by an entity-relationship diagram
    - Normalization Theory
      - Formalize what designs are bad and test for them
- Object-Relational Data Models
  - Relational model: flat, atomic values
  - Extend the relational data model by including object orientation and constructs
  - Allow attributes of tuples to have complex types, including non-atomic values such as nested relations
  - Preserve relational foundations, in particular the declarative access to data, while extending modeling power
  - Provide upward compatibility with existing relational languages
- XML: Extensible Markup Language
  - Defined by the WWW Consortium (W3C)
  - Originally intended as a document markup language not a database language
  - The ability to specify new tags, and to create nested tag structures made XML a great way to exchange data and not just documents
  - XML has become the basis for all new generation **data interchange** formats
  - A wide variety of tools available for parsing, browsing, and querying XML documents/data
- Database Engine
  - Storage manager
    - A program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system
    - Responsible for the following:
      - Interaction with the OS file manager
      - Efficient storing, retrieving, and updating of data
    - Issues:
      - Storage access

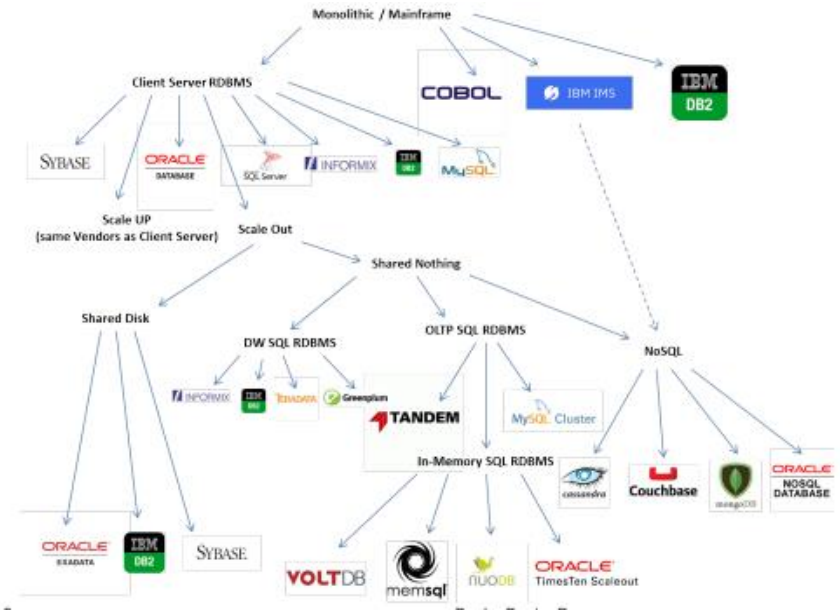
        - File organization
        - Indexing and hashing
    - Query Processing



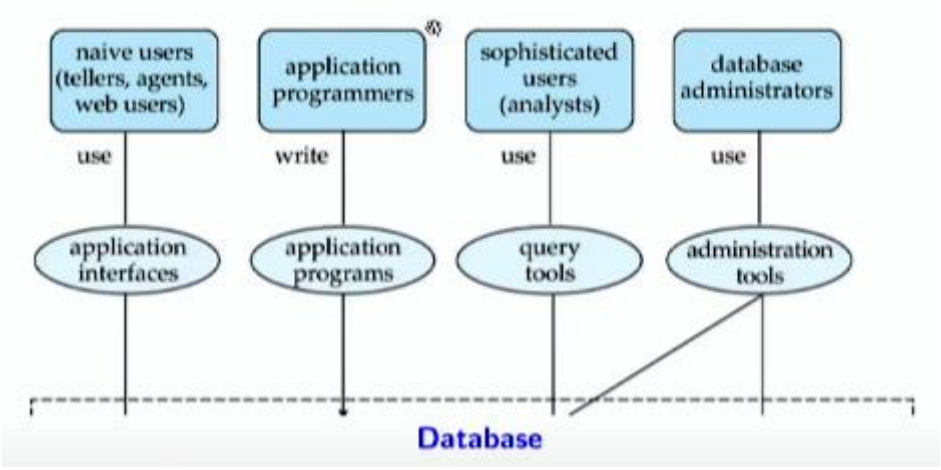
        - 3 major components:
            - Parsing and translation
                - SQL to relational algebra expressions
            - Optimization
            - Evaluation
        - Alternative ways of evaluating a given query
            - Equivalent expressions
            - Different algorithms for each operation
        - Cost difference between a good and a bad way of evaluating a query can be enormous
        - Need to estimate statistics for intermediate results to compute cost of complex expressions
    - Transaction management
        - What if system fails? What if more than one user is concurrently updating the same data?
        - **Transaction** is a collection of operations that performs a single logical function in a database application
        - **Transaction-management** component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures
        - **Concurrency-control manager** controls interaction among the concurrent transactions to ensure the consistency of the database
- Database System Intervals



- Database Architecture
    - The architecture of a database system is greatly influenced by the underlying computer system on which the database is running:
        - Centralized
        - Client-Server
        - Parallel (multi-processor)
        - Distributed
        - Cloud



- Database Users and Administrators



Week 8: Storage Management

L8.1: *Algorithms & Complexity Analysis*

Algorithms and Programs

Algorithm

- A sequence of unambiguous instructions for solving a problem, i.e. a sequence of steps that can be followed to produce a result
- An algorithm is a *recipe* for solving a problem.
- Example: *If you want to make a cake, you need a recipe. The recipe is the algorithm. The cake is the result.*
- An algorithm must terminate.

Program

- A computer program is an algorithm that has been coded into some programming language. It is a sequence of steps that can be followed to produce a result.
- A program may terminate or run forever (e.g. a web server)

Analysis of Algorithms

Why ?

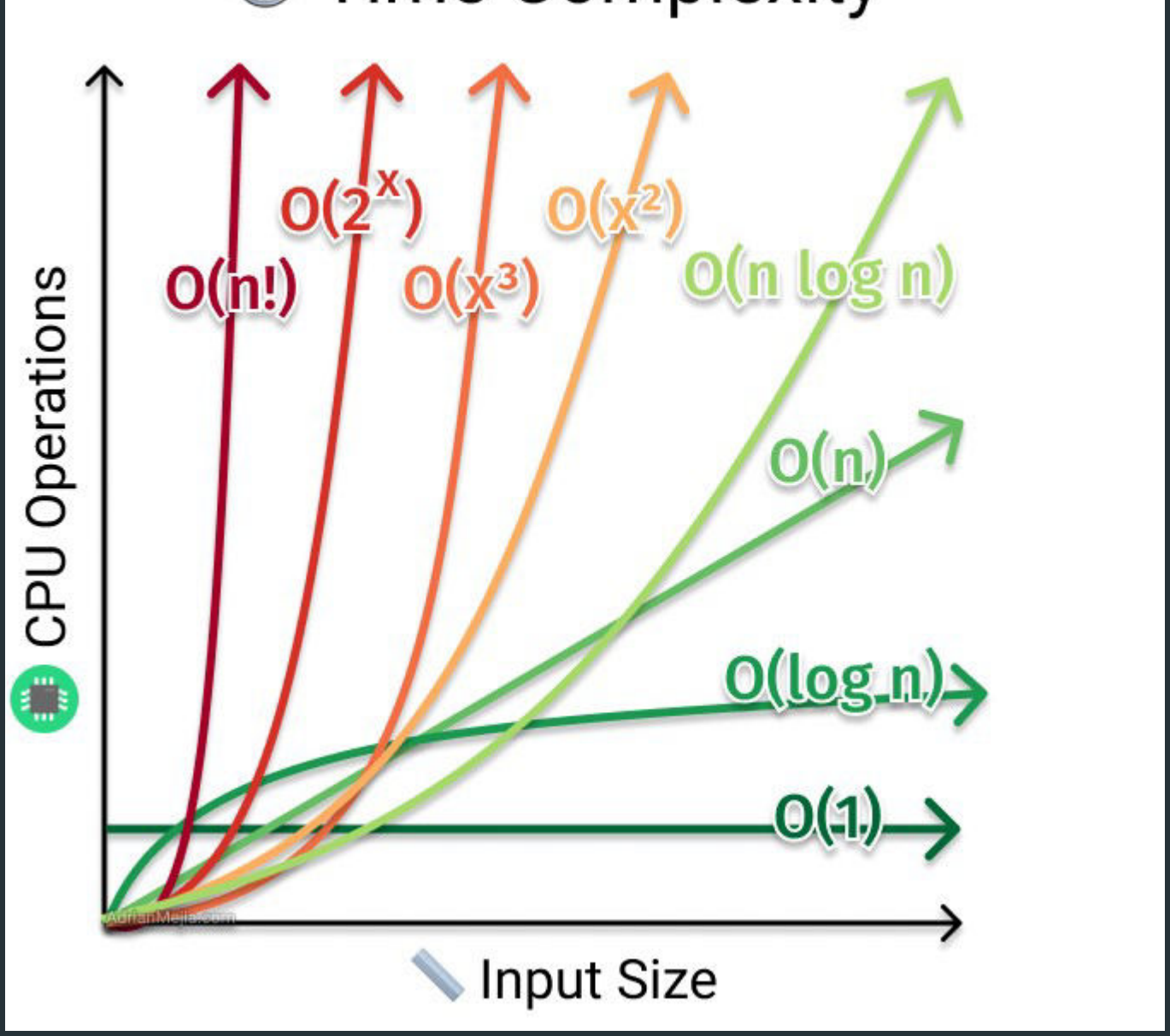
- **Predict performance** of an algorithm
- **Compare** two algorithms for the same problem
- **Understand theoretical basis** for solving problems

What to analyze ?

- **Time** - How long does it take for an algorithm to complete?
- **Space** - How much extra memory does it need?

How to analyze ?

- ***Counting models***
  - `Total running time = Sum of cost x frequency for all operations`
- ***Asymptotic analysis***
  - Cannot compare actual times, hence compare *Growth* or how time increases with input size
  - **Big-O notation** - upper bound on the growth of an algorithm
  - **Big-Omega notation** - lower bound on the growth of an algorithm
  - **Big-Theta notation** - both upper and lower bound on the growth of an algorithm



- ***Generating functions***
- ***Master Theorem***

Where to Analyze ?

- **Worst case** - Maximum number of steps taken on any instance of size n
- **Average case** - Average number of steps taken on any instance of size n
- **Best case** - Minimum number of steps taken on any instance of size n

---

L8.2 & 8.3: *Data Structures*

Data Structure

- A data structure specifies the way of organizing and storing in memory data that enables efficient access and modification of the data.
  - **Linear Data structures**
  - **Non-Linear Data structures**
- For applications related to data management, the key operations are:

- **Create**
- **Insert**
- **Delete**
- **Find / Search**
- **Close**

Linear Data Structures

- A linear data structure is a data structure where data elements are arranged sequentially or linearly, i.e. each element (data) is connected to it's previous and next element.
- Examples:
  - **Arrays**
    - The data elements are stored at contiguous memory locations.
  - **Linked Lists**
    - The data elements are linked using pointers.
  - **Stacks**
    - It follows LIFO (Last In First Out) principle, i.e. the element inserted at the last is the first element to come out.
  - **Queues**
    - It follows FIFO (First In First Out) principle, i.e. the element inserted at the first is the first element to come out.

> For more understanding of linear data structures, check this
> You can learn about searching techniques in linear data structures here

	Array		**Linked List**	
	Unordered	Ordered	Unordered	Ordered
Access	O(1)	O(1)	O(n)	O(n)
Insert	O(n)	O(n)	O(1)	O(1)
Delete	O(n)	O(n)	O(1)	O(1)
Search	O(n)	O(logn)	O(n)	O(n)

- Stacks and Queues are abstract data structures. They can be implemented using arrays or linked lists.
- So, the space complexity of linear data structures is O(n).
- All of them having complexities that are identical for Worst case as well as Average case.

Non-linear data structures can be used to trade-off between extremes and achieve a balance good performance for all.

Non-Linear Data Structures

- A non-linear data structure is a data structure in which data items are not arranged in a sequence and each

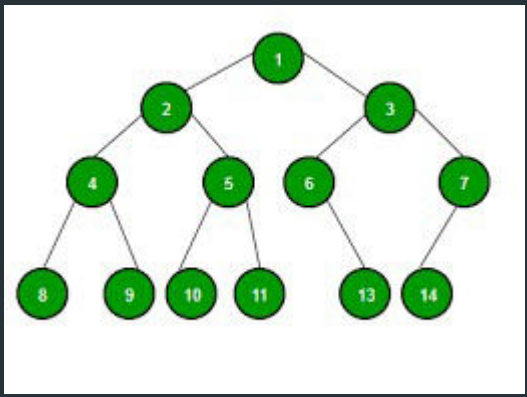
element may have multiple paths to connect to other elements.

- Unlike linear data structures, where an element is at most connected to 2 elements, non-linear data structures can be connected to any number of elements.
- Traversing is not possible in single run as elements are not arranged in sequential manner.
- Example:
  - **Graph**
    - A graph is a non-linear data structure consisting of nodes and edges.
    - The edges are lines or arcs that connect any two nodes in the graph.
    - They can be directed or undirected, and can be weighted or unweighted.
  - **Tree**
    - A tree is a non-linear data structure in which data elements are organized in hierarchical structure. The topmost node is called root of the tree.
    - The elements that are directly under an element are called its children.
    - The element directly above something is called its parent. Finally, elements with no children are called leaves.
    - They can be rooted or unrooted, binary or n-arry, balanced or unbalanced, etc.
  - **Hash Table**
    - In a hash table we store data in linked list which are accessed by a key which refers to the index of the array where the linked list is stored.
    - Each element in an array is a linked list.
  - and there are many more...

> For more understanding of Graph, check this

Tree

- Tree is a connected acyclic graph representing hierarchical structure.



- **Root node**:
  - The topmost node in a tree is called the root node.
  - There is only one root node in a tree.
- **Parent node**:
  - A node that has sub-nodes connected to it is called a parent node.
  - A node can be a parent node to multiple nodes.
  - A node can be a parent node as well as a child node.
- **Child node**:

    - A node that is connected to a parent node is called a child node.
    - A node can be a child node to multiple nodes.
    - A node can be a parent node as well as a child node.
- **Leaf node**:
    - A node that does not have any child node is called a leaf node.
    - A leaf node is also called an external node.
- **Internal node**:
    - A node that has at least one child node is called an internal node.
- **Subtree**:
    - A subtree is a tree that is part of a larger tree.
- **Path**:
    - A path is a sequence of nodes connected by edges of a tree.
- **Sibling**:
    - Nodes that have the same parent node are called siblings.
- **Arity**:
    - Number of children of a node is called its arity.
- **Levels**:
    - Root node has level 0, its children have level 1, and so on.
- **Height**:
    - Maximum level of any node in a tree is called its height.
- **Binary tree**:
    - A tree which arity 2, i.e. each node has at most 2 children.

A tree maybe:

- rooted or unrooted
- Binary or n-ary
- Balanced or unbalanced
- Disconnected (forest) or connected

Examples:

- Composite attributes
- Family Genealogy
- Search trees

> **Facts:**
>
> 1. A tree with n nodes has $n - 1$ edges.
> 2. The maximum number of nodes at level l of a binary tree is 2^l.
> 3. If h is the height of a binary tree with n nodes, then:
> $h + 1 \leq n \leq 2^{h+1} - 1$
> 4. For a k-ary tree with n nodes:
> $log_k(n) \leq h \leq n$

Hash Table

- A hash table is a data structure that maps keys to values for highly efficient lookup.
- It elements an associative array abstract data type, a structure that can map keys to values by using a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

A hash table may be using:

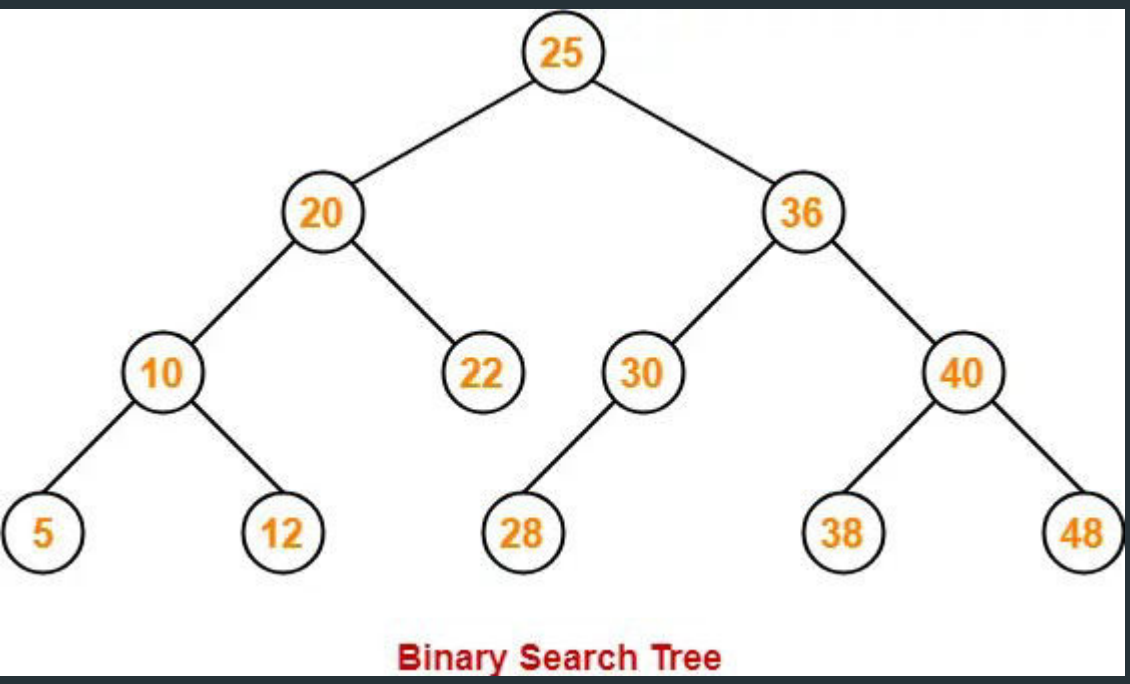
- Static or dynamic schemes
- open addressing
- 2-choice hashing

Examples:

- Associative arrays
- Database indexing
- Caches

Binary Search and Binary Search Tree

- Binary search is efficient in search of a key in a sorted array, i.e. O(log n).
- As the binary search splits the array, we can conceptually consider the Middle element to be the root of a tree and the left and right sub-arrays to be the left and right sub-trees and progress recursively, we get a Binary Search Tree.
- So, A Binary Search Tree is a binary tree in which the root node is greater than all the nodes in the left sub-tree and less than all the nodes in the right sub-tree.



Binary Search Tree

Searching in a Binary Search Tree

- Searching in a Binary Search Tree takes O(h), where h is the height of the tree.
- **Worst Case**: If the BST is skewed, then time complexity is O(n), where n is the number of nodes.
- **Best Case**: If the BST is balanced, then time complexity is O(log n), height (h) becomes log n.

Complexities of Various Data Structures

	Data Structure	Time Complexity								Space Complexity
		Average				Worst				Worst
		Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Linear Data Structures	Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
	Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Non-Linear Data Structures	Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
	Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
	Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
	Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
	B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
	AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

L8.4: *Physical Storage*

Physical Storage

Classification of Physical storage media

- **Speed** - How fast can data be accessed?
- **Cost** - How much does it cost?
- **Reliability** - Data loss on power failure or system crash, physical failure.
- **Volatile storage** - Data is lost when power is turned off.
- **Non-volatile storage** - Data is retained when power is turned off.
- **Cache** - Small, very fast, volatile and most costly memory managed by hardware.
- **Main memory** - Larger, fast, volatile memory managed by hardware.

Flash Memory

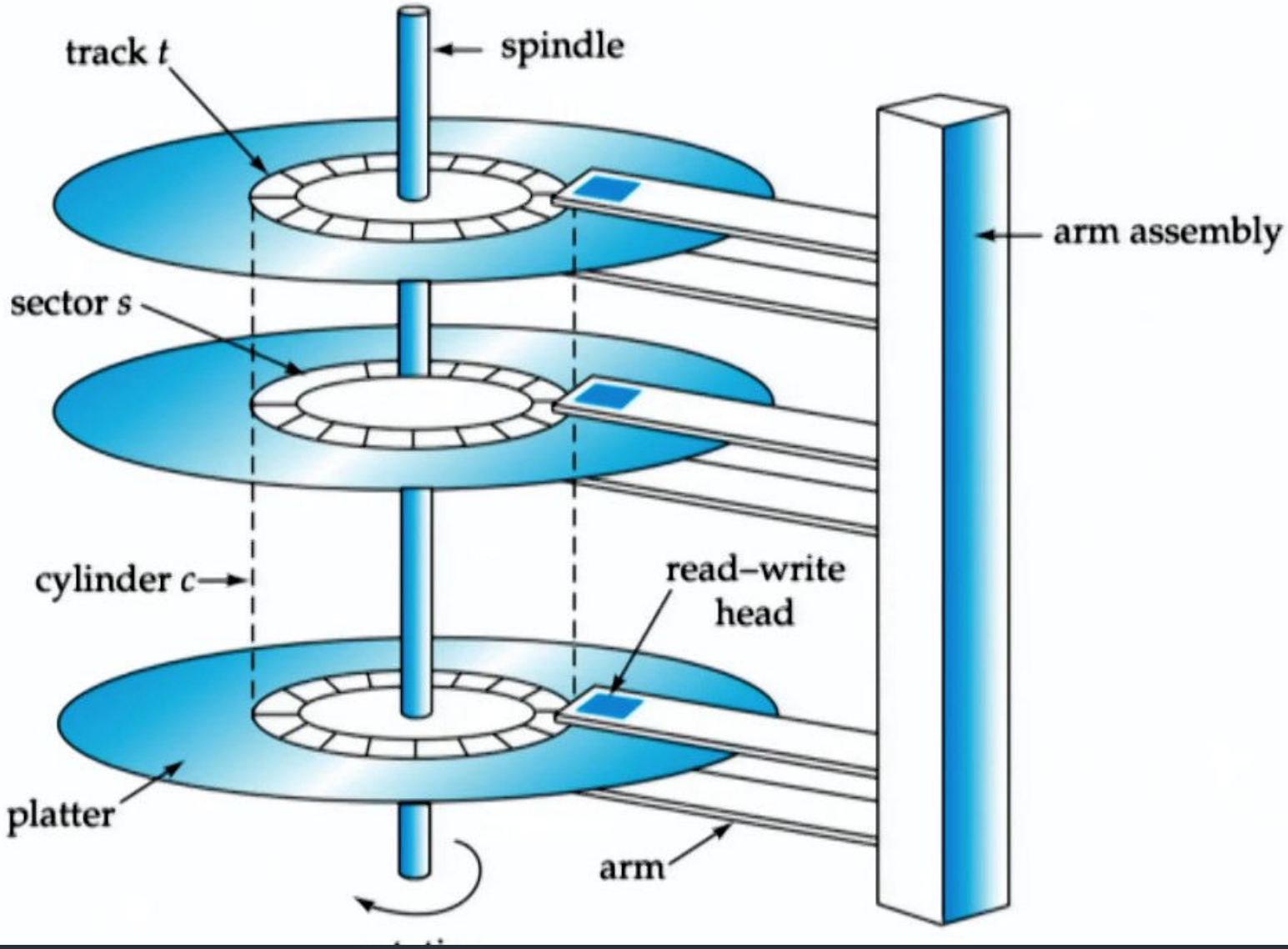
- Flash memory is a non-volatile storage technology that retains data even when power is removed.
- It's commonly used in portable devices, memory cards, and USB drives.
- *Advantages*: Fast read speeds, low power consumption, compact size, durability, and resistance to shocks.
- *Disadvantages*: Limited write endurance (limited number of write cycles per cell).
- Moderately priced, highly reliable for read operations, read speeds are fast, while write speeds are slower.

Magnetic Disk

- Data is stored on spinning disk and read/written magnetically.

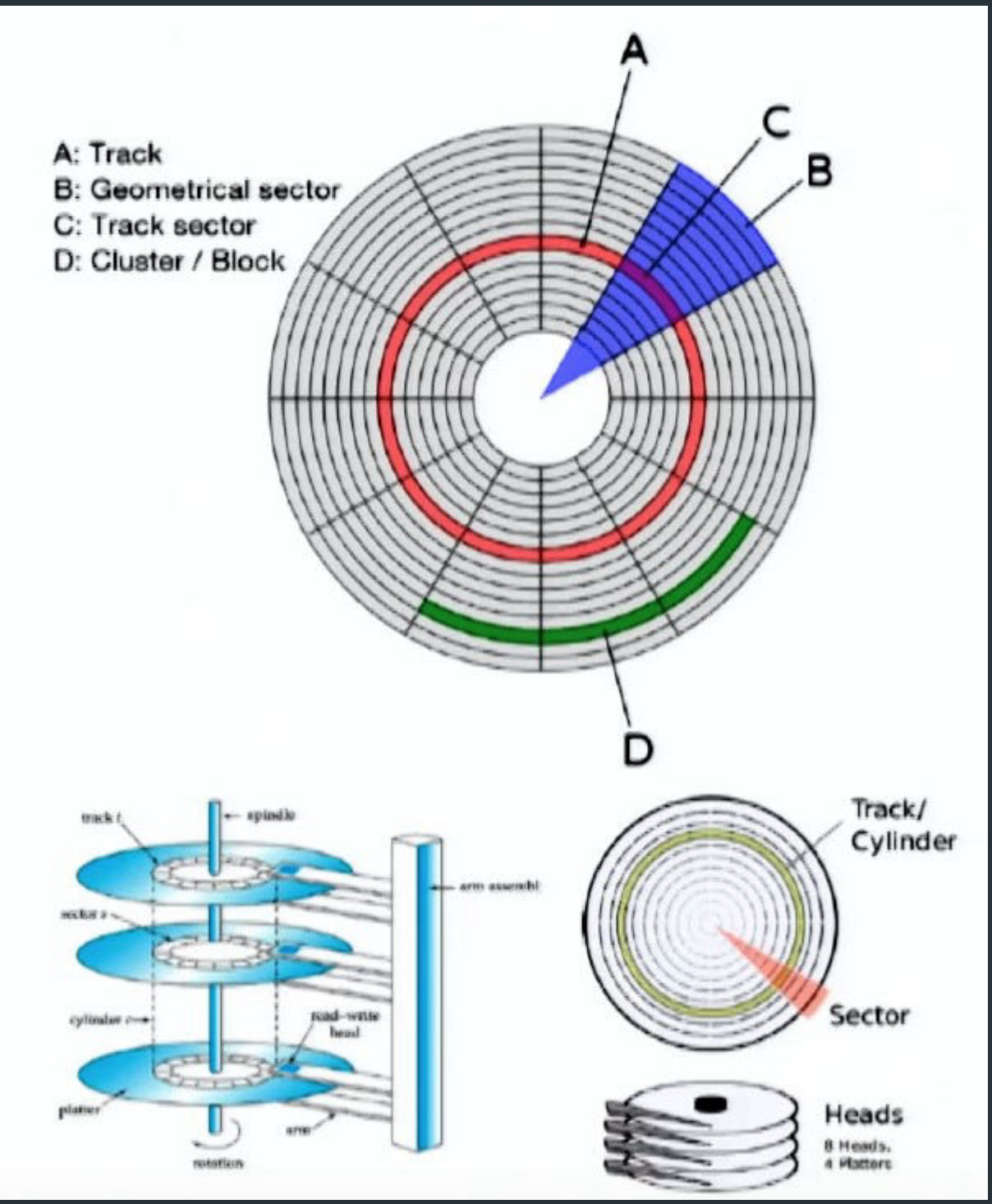
- Primary medium for long-term storage.
- Data must be moved from disk to main memory for access and written ○ack for storage, much slower than main memory.
- **Direct access**: Possible to read data on disk in any order.

Mechanism



- Read-write head
  - Positioned very close to the platter surface
  - Reads or write magnetically encoded information.
- Surface of platter divided into circular **tracks**
  - Over 50k-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**
  - A sector is the smallest unit of data that can be read or written
  - Sector size is typically 512 bytes
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins: read/write as sector passed under head
- Head-disk assemblies

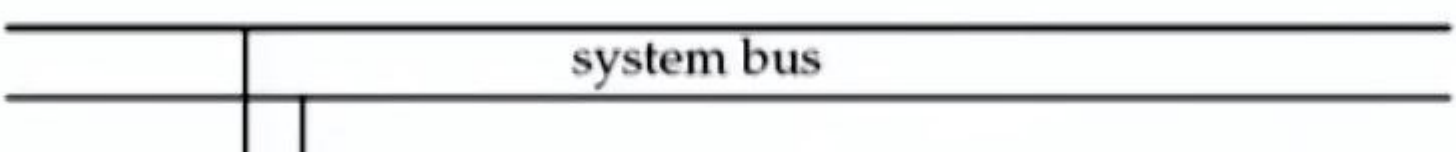
- multiple disk platters on a single spindle (1 to 5 usually)
- one head per platter, mounted on a common arm.

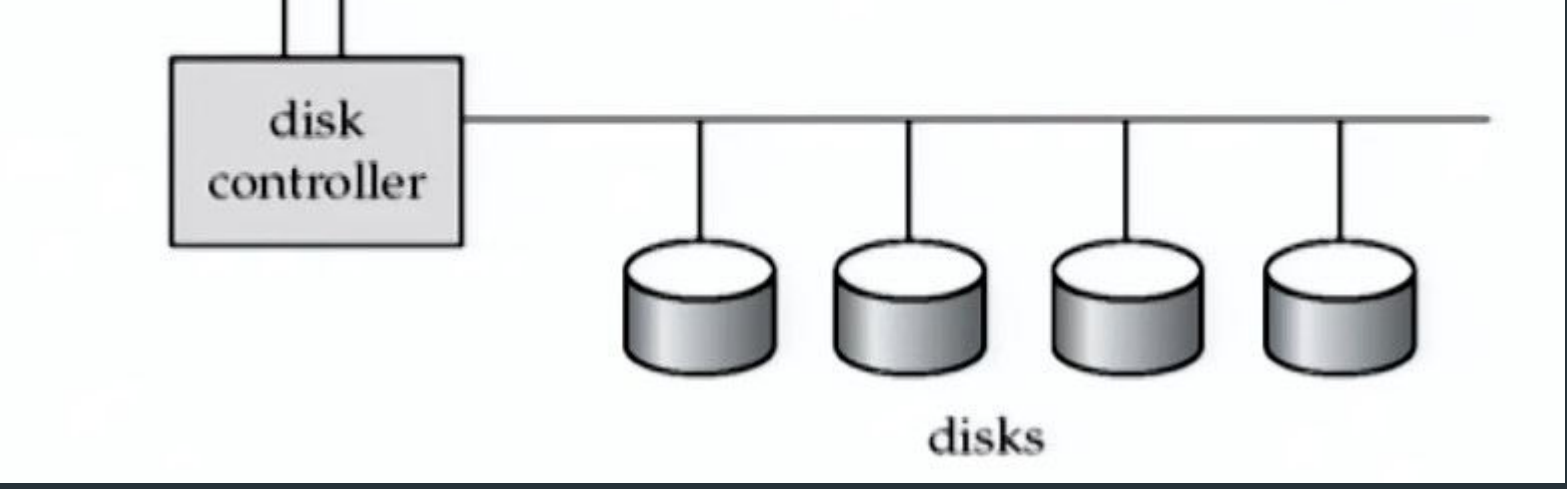


Dis controller

- Interface between the computer system and the disk drive hardware
- Accepts high-level commands to read or write a sector
- Computes and attaches checksums to each sector to verify that correct read back
- Perform remapping of bad sectors

Disk Subsystem





- **Disk Interface Standards Families**: ATA, SATA, SCSI, SAS
- **Storage Area Networks (SAN)** connects disks by a high-speed network to number of servers.
- **Network Attached Storage (NAS)** provides a file system interface using networked file system protocol

Performance Measures

- **Access Time**: Time from a read or write request issue to start of data transfer.
- **Seek Time**: Time to move the disk arm to the desired track.
- **Rotational Latency**: Time for the desired sector to rotate under the disk head.
- **Data transfer Rate**: Rate at which data is transferred to or from the disk.
- **Mean Time to Failure (MTTF)**: Average time the disk is expected to run continuously without any failure.

Optical Storage

- Non-volatile, data is read optically from a spinning disk using a laser.
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) are most popular forms.
- Write once, read many (WORM) optical disks are used for archival storage.
- Reads and write slower than magnetic disks, but cheaper and more durable.

Tape Storage

- Non-volatile, used primarily for backup and for achival data.
- Sequential access, very slow, but very cheap and durable.

Flash Drives

- Flash drives, or USB drives, are portable storage devices using flash memory. They connect via USB ports and come in various sizes.
- *Advantages*: Compact, lightweight, no moving parts, high data transfer rates, and plug-and-play convenience.
- *Disadvantages*: Limited write cycles, smaller capacities compared to traditional HDDs.
- Affordable, reliable for read operations, high read speeds, moderate write speeds.

Flash Storage

- Flash storage refers to non-volatile memory technology used to store data electronically without the need

for power.

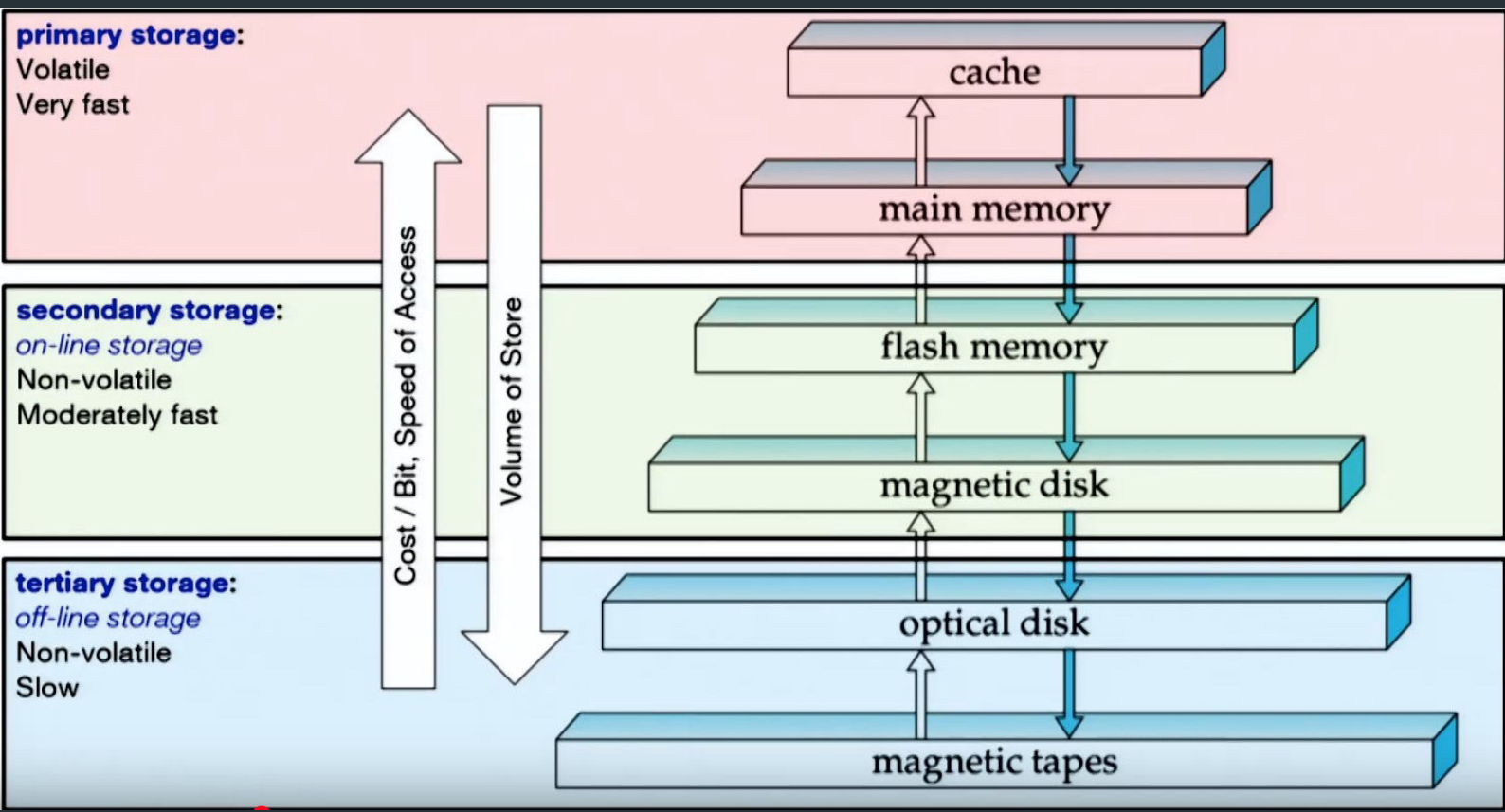
- *Advantages*: Offers high-speed data access, low power consumption, and resistance to physical shocks due to no moving parts.
- *Disadvantages*: Limited write endurance, higher cost compared to traditional HDDs for larger capacities.
- Can be more expensive than HDDs but cost-effective for performance gains, reliable for read operations, and provides faster data access than HDDs.

SSD

- SSDs are storage devices that use flash memory to provide high-speed, non-volatile data storage.
- *Advantages*: Lightning-fast data access, low power usage, silent operation, and longer lifespan than HDDs.
- *Disadvantages*: Generally more expensive, write endurance concerns (though improving), and cost per gigabyte is higher.
- Higher initial cost but becoming more affordable, reliable for both read and write operations, significantly faster read/write speeds than HDDs.

HDD

- HDDs use spinning disks to read/write data magnetically, offering long-term storage solutions.
- *Advantages*: Cost-effective for large storage capacities, improving technology for faster performance, and stability for long-term data storage.
- *Disadvantages*: Slower access times compared to SSDs, mechanical parts can be prone to damage, and consumes more power.
- Generally cheaper per gigabyte, reliable for read operations, slower read/write speeds compared to SSDs.



Cloud Storage

- Cloud storage is a service model in which data is maintained, managed, backed up remotely and made available to users over a network (typically the Internet).
- Applications access cloud storage through traditional storage protocols or directly via an API.
- Example of cloud storage providers: Amazon S3, Google Cloud Storage, Microsoft Azure Storage, Dropbox, etc.

L8.5: *File Structure* 🔗

- You can refer to this video lecture 🔗

Week 6: Functional Dependency and Normal Forms (cont.)

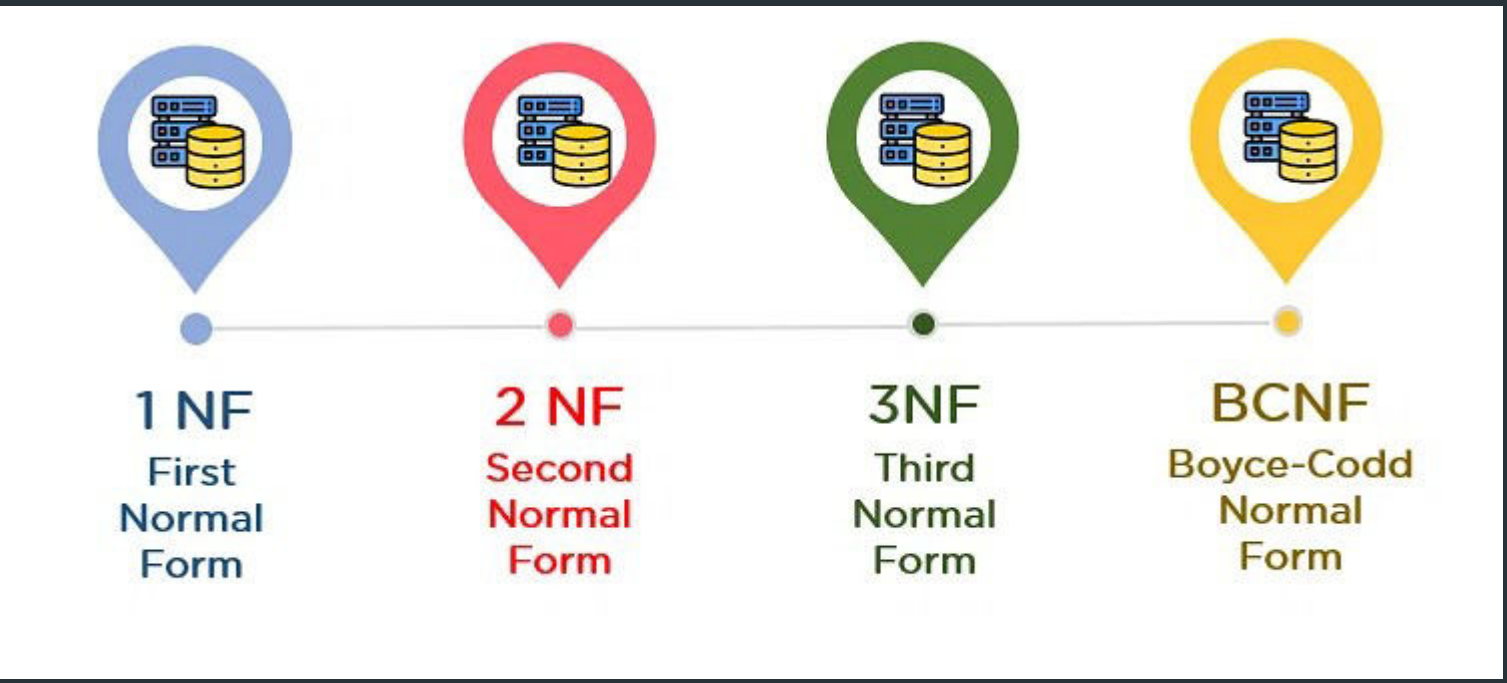
L6.1: Relational Database Design/6: Normal Forms

Normal Forms

- Normalization is used for mainly two purposes:
  - To eliminate redundant data.
  - To ensure data dependencies make sense, i.e. data is logically sorted.

$$\text{Normalization} \implies \text{Good decomposition} \implies \text{Minimization of Dependency} \implies \text{Redundancy} \implies \text{Anomaly}$$

- To know more about normalization, check here



Normalization and Normal Forms

- A normal form specifies a set of conditions that the relational schema must satisfy in terms of its constraints.
- Some common normal forms are:
  - First Normal Form (1NF)
  - Second Normal Form (2NF)
  - Third Normal Form (3NF)
  - Boyce-Codd Normal Form (BCNF)
- A relation is said to be "**normalized**" if it meets *third normal form (3NF)*.

- Most 3NF relations are free of update, insertion, and deletion anomalies.

- We will consider this relation as example in this lecture:

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

First Normal Form (1NF)

- **Rule** - Domain of attribute must include only atomic values.

Relation in 1NF:

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Second Normal Form (2NF)

- **Rule 1** - Relation must be in 1NF.
- **Rule 2** - Relation does not contain any partial dependency.
  - Single column primary key that does not functionally dependent on any subset of candidate key relation.

Relations in 2NF:

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Membership ID	Movies Rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Third Normal Form (3NF)

- **Rule 1** - Relation must be in 2NF.
- **Rule 2** - Relation does not contain any transitive dependency.
    - Non-prime attribute is not functionally dependent on any other non-prime attribute.

Relation in 3NF:

Membership ID	Full Names	Physical Address	Salutation ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

Membership ID	Movies Rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

L6.2: Relational Database Design/7: Normal Forms

3NF Decomposition

Testing

- **Optimization**: Need to check only FDs in F, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, then we have to verify if each attribute in β in a candidate key of R.
    - Decomposition into #NF can be done in polynomial time.

Algorithm

- Eliminate redundant FDs, resulting in a canonical cover F_c of F.
- Create a relation $R_i = XY$ for each FD $X \rightarrow Y$ in F_c.

- If the key K of R does not occur in any relation R_i, create one more relation $R_j = K$.

```
Let Fc be the canonical cover of F;
i=0;
for each FD a->b in Fc do
    If none of the schema Rj, 1 <= j <= i contains ab
        then begin
            i = i + 1;
            Rj = ab;
        end
If none of the schema Rj, 1 <= j <= i contains a candidate key for R
     then begin
          i = i + 1;
          Rj = any candidate key for R;
     end
repeat
If any schema Rj is contained in another schema Rk
     then /* dleete Rj */
          Rj = R;
          i -= 1;
return (R1, R2, ..., Ri)
```

Example

- Relation schema:
  cust_banker_branch = (*customer_id*, *employee_id*, *branch_name*, *type*)
- The functional dependencies for this relation schema are:
  a) *customer_id*, *employee_id* → *branch_name*, *type*
  b) *employee_id* → *branch_name*
  c) *customer_id*, *branch_name* → *employee_id*
- We first compute a canonical cover
  - *branch_name* is extraneous in the RHS of the 1st dependency
  - No other attribute is extraneous, so we get F_c =
    customer_id, *employee_id* → *type*
    employee_id → *branch_name*
    customer_id, *branch_name* → *employee_id*

- The **for** loop generates following 3NF schema:
  (*customer_id*, *employee_id*, *type*)
  (*employee_id*, *branch_name*)
  (*customer_id*, *branch_name*, *employee_id*)
  - Observe that (*customer_id*, *employee_id*, *type*) contains a candidate key of the original schema, so no further relation schema needs be added

- At end of for loop, detect and delete schemas, such as *(employee_id, branch_name)*, which are subsets of other schemas
  - result will not depend on the order in which FDs are considered

- The resultant simplified 3NF schema is:
  (customer_id, employee_id, type)
  (customer_id, branch_name, employee_id)

BCNF Decomposition

- A relation schema R is in BCNF with respect to a set F of Fds if for all FDs in F^+ of the form:
  - $\alpha \rightarrow \beta$ where $a \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is a trivial FD ($\beta \subseteq \alpha$)
  - α is a superkey for schema R.

Testing

Simplified test

- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ causes a violation of BCNF.

Testing for BCNF Decomposition

- To check if a relation R_i in a decomposition of R is in BCNF,
  - Either test R_i for BCNF with respect to the **restriction** of F to R_i (i.e. all FDs in F^+ that contain only attributes of R_i)
  - Or use the original set of dependencies F that hold on R with following test:
    - For every set of attributes $\alpha \subseteq R_i$, check that α^+ either includes no attribute of $R_i - \alpha$ or includes all attributes of R_i.
    - If the condition is violated by some $\alpha \rightarrow \beta$ in F, then the following dependency can be shown to hold on R_i and R_i violates BCNF:

$$\alpha \rightarrow \left(\alpha^+ - \alpha\right) \cap R_i$$

    - We use above dependency to decompose R_i.

Algorithm

1. For all dependencies $A \rightarrow B$ in F^+, check if A is a superkey by using attribute closure.

2. If not, then
    1. Choose a dependency in F^+ that violates BCNF, say $A \rightarrow B$.
    2. Create `R1 = AB`
    3. Create `R2 = (R - (B - A))`
3. Repeat For R1 and R2
    1. By defining $F1^+$ to be all dependencies in F that contain only attributes in R1, similarly $F2^+$.

Example

- **R** (A, B, C, D)
  F $= \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A\}$
- Decomposition: **R1**(A, B) **R2**(B, C) **R3**(C, D)
    - $A \rightarrow B$ is preserved on table R1
    - $B \rightarrow C$ is preserved on table R2
    - $C \rightarrow D$ is preserved on table R3
    - We have to check whether the one remaining FD: **D→A** is preserved or not.

R1	R2	R3
F_1={**A** → AB, **B** → BA}	F_2={**B** → BC, **C** → CB}	F_3={**C** → CD, **D** → DC}

    - $F' = F_1 \cup F_2 \cup F_3$.
    - Checking for: **D** → A in F'^+
        - ▷ $D \rightarrow C$ (from R3), $C \rightarrow B$ (from R2), $B \rightarrow A$ (from R1) : **D→ A** (By Transitivity)
          Hence all dependencies are preserved.

- $R(ABCD)$:. $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A\}$
- $Decomp = \{AB, BC, CD\}$
- On projections:

R1	R2	R3
F1 **A → B**	F2 **B → C**	F3 **C → D**

In this algo F1, F2, F3 are not the closure sets, rather the set of dependencies directly applicable on R1, R2, R3 respectively.

- Need to check for: $A \rightarrow B, B \rightarrow C, C \rightarrow D,$ ***D → A***
- $(D)+/F1 = D$. $(D)+/F2 = D$. $(D)+/F3 = D$. So, ***D → A*** could not be preserved.
- In the previous method we saw the dependency was preserved. In reality also it is preserved. Therefore the polynomial time algorithm may not work in case of all examples. To prove preservation Algo 2 is sufficient but not necessary whereas Algo 1 is both sufficient as well as necessary.

3NF vs BCNF

3NF	BCNF
It concentrates on **Primary Key**.	It concentrates on **Candidate Key**.

3NF	BCNF
Redundancy is high as compared to BCNF.	0% redundancy.
It preserves all the dependencies.	It may not preserve all the dependencies.
A dependency $X \rightarrow Y$ is allowed in 3NF if X is a superkey or Y is a part of some key	A dependency $X \rightarrow Y$ is allowed in BCNF if X is a superkey.

L6.3: Relational Database Design/8: Case Study 🔗

L6.4: Relational Database Design/9: MVD & 4NF

MVD: Multivalued Dependency

- If two or more independent relations are kept in a single relation, then MVD occurs.
- A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes.
- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency** $\alpha \twoheadrightarrow \beta$ holds on R if, in any legal relation r(R), for all pairs of tuples t1 and t2 in r that agree on α, there exist tuples t3 and t4 in r such that:
  - t1[alpha] = t2[alpha] = t3[alpha] = t4[alpha]
  - t3[beta] = t1[beta] and t4[beta] = t2[beta]

Example:

- Suppose there is a bike manufacturer company which produces two colors(white and black) of each model every year.

BIKE_MODEL	MANUF_YEAR	COLOR
M2011	2008	White
M2001	2008	Black
M3001	2013	White
M3001	2013	Black
M4006	2017	White
M4006	2017	White

- Here columns `COLOR` and `MANUF_YEAR` are dependent on `BIKE_MODEL` and independent of each other.
- In this case, these two columns can be called as multivalued dependent on `BIKE_MODEL`. The representation of these dependencies is shown below:
  - `BIKE_MODEL ->> MANUF_YEAR`
  - `BIKE_MODEL ->> COLOR`

	Name	Rule
C-	Complementation	If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow (R - (X \cup Y))$.
A-	Augmentation	If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.
T-	Transitivity	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.
	Replication	If $X \rightarrow Y$, then $X \twoheadrightarrow Y$ but the reverse is not true.
	Coalescence	If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$ and $Y \supseteq Z$, then $X \rightarrow Z$.

- A MVD $\mathbf{X} \twoheadrightarrow \mathbf{Y}$ in $\mathbf{R}$ is called a trivial MVD is
  - $\mathbf{Y}$ is a subset of $\mathbf{X}$ ($\mathbf{X} \supseteq \mathbf{Y}$) or
  - $\mathbf{X} \cup \mathbf{Y} = \mathbf{R}$. Otherwise, it is a non trivial MVD and we have to repeat values redundantly in the tuples.

Decomposition to 4NF

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all MVD in D^+ of the form:
  - $\alpha \twoheadrightarrow \beta$ where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
    - $\alpha \twoheadrightarrow \beta$ is a trivial MVD ($\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
    - α is a superkey for schema R.
- If a relation schema R is in 4NF, then it is also in BCNF.

Restriction of MVD

- The restriction of D to R_i is the set D_i consisting of
  - All FDs in D^+ that include only attributes of R_i
  - All multivalued dependencies of the form
    $\alpha \twoheadrightarrow (\beta \cap R_i)$
    where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+.

Algorithm

1. For all dependencies $A \twoheadrightarrow B$ in D^+, check if A is a superkey by using attribute closure.
2. If not, then:
   1. Choose a dependency in F^+ that violates 4NF, say $A \twoheadrightarrow B$.
   2. Create `R1 = AB`.
   3. Create `R2 = (R - (B - A))`.

3. Repeat for R1 and R2, by defining $D1^+$ to be all dependencies in D that contain only attributes in R1, similarly $D2^+$.

```
result = {R};
done = false;
compute D+;
Let Di denote the restriction of D+ to Ri;
while (not done)
     if (there is a schema Ri in result that is not in 4NF) then
          begin
               let A->>B be a nontrivial MVD that holds on Ri such that
                    A -> Ri is not on Di and a intersection B is not on Di;
               result = (result - Ri) U (Ri - (B - A)) U (A,B);
          end
     else
          done = true;
```

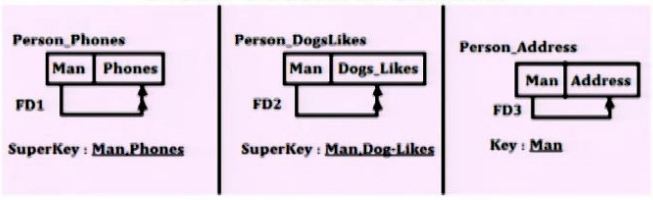
Example

1.

- **Person_Modify(Man(M), Phones(P), Dog_Likes(D), Address(A))**
- FDs:
  - FD1 : Man ↠ Phones
  - FD2 : Man ↠ Dogs_Like
  - FD3 : Man → Address
- Key = MPD
- All dependencies violate 4NF

Man(M)	Phones(P)	Dogs_Likes(D)	Address(A)
M1	P1	D1	49-ABC,Bhiwani(HR.)
M1	P2	D2	49-ABC,Bhiwani(HR.)
M2	P3	D2	36-XYZ,Rohtak(HR.)
M1	P1	D2	49-ABC,Bhiwani(HR.)
M1	P2	D1	49-ABC,Bhiwani(HR.)

Post Normalization



In the above relations for both the MVD's – ***'X' is Man***, which is again not the super key, but as ***X ∪ Y = R*** i.e. (Man & Phones) together make the relation.
So, the above MVD's are trivial and in FD 3, Address is functionally dependent on Man, where **Man** is the key in **Person_Address**, hence all the three relations are in 4NF.

2.

- R =(A, B, C, G, H, I)
  F = A ↠ B
  B ↠ HI
  CG ↠ H
- *R* is not in 4NF since *A* ↠ *B* and *A* is not a superkey for *R*
- Decomposition
  a) R_1 = (A, B) (R_1 is in 4NF)

b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF, decompose into R_3 and R_4)
c) $R_3 = (C, G, H)$ (R_3 is in 4NF)
d) $R_4 = (A, C, G, I)$ (R_4 is not in 4NF, decompose into R_5 and R_6)
- $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI \rightarrow A \twoheadrightarrow HI$, (MVD transitivity), and
- and hence $A \twoheadrightarrow I$ (MVD restriction to R_4)

e) $R_5 = (A, I)$ (R_5 is in 4NF)
f) $R_6 = (A, C, G)$ (R_6 is in 4NF)

L6.5: Relational Database Design/10: Design Summary & Temporal Data

Database Design Process

Design Goals

- Goal for a relational database design is:
  - BCNF / 4NF
  - Lossless Join
  - Dependency Preserving
- If we cannot achieve this, we accept one of:
  - Lack of dependency preserving
  - Redundancy due to use of 3NF

Normal Forms

- Further NFs
  - **Elementary Key Normal Form** (EKNF)
  - **Essential Tuple Normal Form** (ETNF)
  - **Join Dependencies And 5NF** (JD/5NF)
  - **Sixth Normal Form** (6NF)
  - **Domain/Key Normal Form** (DKNF)
- **Join dependencies** generalize multivalued dependencies which lead to *project-join normal form* (PJNF) or 5NF.
- A class of even more general constraints, leads to normal form called **domain-key normal form** (DKNF).

We have assumed schema R is given: - R could have been generated when converting E-R diagram to a set of tables. - R could have been a single relation containing all attributes that are of interest (universal relation). - Normalization breaks R into smaller relations - R could have been the reuslt of ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in real design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity.
- FDs from non-ky attributes of a relationship set possible, but rare, as most relationship sets are binary.

Denormalization for Performance

- We may want to use non-normalized schema for performance
- Example: Displaying `prereq` along with `course_id`, and `title` requires join of course with `prereq`
  - `Course(course_id, title, ...)`
  - `Preqrequisite(course_id, prereq)`
- **Alternative 1**: Use denormalized relation containing attributes of course as well as prereq with all above attributes.
  - `Course(course_id, title, preqreq)`
- **Alternative 2**: Use a materialized view defined as cross join of course and prereq.

Temporal Databases

- A temporal data have an association time interval during which the data are valid.
- A snapshot is the value of the data at a particular time.
- In practice, database designers may add start and end time attributes to relations.
- Example: `course(course_id, course_title)` is replaced by `course(course_id, course_title, start, end)`.
- Constraint: no two tuples can have overlapping valid times and are Hard to enforce efficiently.
- Foreign key references maybe to current version of data, or to data at a point in time.

Temporal Data

- **Model of Temporal Domain** - 1-dimensional linearly ordered which maybe:
  - Discrete or dense
  - Bounded or unbounded
  - Single or multi-dimensional
  - Linear or non-linear
- **Temporal ER model** by adding valid time to *Attributes*, *Entities*, *Relationships*.

Modeling Temporal Data

- **Valid Time** - Time period during which a fact is true in the real world, provided by the application.
- **Transaction Time** - Time period during which a fact is stored in the database, based on transaction serialization order and is the timestamp automatically generated by the system.
- Temporal relation is one where each tuple has associated time; either valid time or transaction time or both.
  - **Uni-Temporal Relations**: Valid time or transaction time.
  - **Bi-Temporal Relations**: Both valid time and transaction time. It includes both start and end time.

Example:

- John's Data In Non-Temporal Database

Date	Real world event	Address
April 3, 1992	John is born	
April 6, 1992	John's father registered his birth	Chennai
June 21, 2015	John gets a job	Chennai
Jan 10, 2016	John registers his new address	Mumbai

In a non-temporal database, John's address is entered as Chennai from 1992. When he registers his new address in 2016, the database gets updated and the address field now shows his Mumbai address. The previous Chennai address details will not be available. So, it will be difficult to find out exactly when he was living in Chennai and when he moved to Mumbai.

- John was born on April 3, 1992 in Chennai.
- His father registered his birth after three days on April 6, 1992.
- John did his entire schooling and college in Chennai.
- He got a job in Mumbai and shifted to Mumbai on June 21, 2015.
- He registered his change of address only on Jan 10, 2016.

- Uni-Temporal Relation (Adding Valid Time To John's Data)

Name	City	Valid From	Valid Till
John	Chennai	April 3, 1992	June 20, 2015
John	Mumbai	June 21, 2015	∞

- Bi-Temporal Relation (John's Data Using Both Valid And Transaction Time)

Name	City	Valid From	Valid Till	Entered	Superseded
John	Chennai	April 3, 1992	June 20, 2015	April 6, 1992	Jan 10, 2016
John	Mumbai	June 21, 2015	∞	Jan 10, 2016	∞

- John was born on April 3, 1992 in Chennai.
- His father registered his birth after three days on April 6, 1992.

Tutorials

6.1 Problem solving on normalization 🔗

6.2 Multivalued Dependency 🔗

❖ Week Recap

- Basic notions of Relational Database Models
  - Attributes and their types
  - Mathematical structure of relational model
  - Schema and Instance
  - Keys, primary as well as foreign
- Relational algebra with operators
- Relational query language
  - DDL (Data Definition)
  - DML (Basic Query Structure)
- Detailed understanding of basic query structure
- Set operations, null values, and aggregation

❖ Select distinct

From the classroom relation in the figure, find the names of buildings in which every individual classroom has capacity less than 100 (removing the duplicates).

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure: classroom relation

➔Query:

select distinct building **from** classroom
where capacity < 100;

➔Output:

building
Painter
Taylor
Watson

❖ Select all

From the classroom relation in the figure, find the names of buildings in which every individual classroom has capacity less than 100 (without removing the duplicates).

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure: classroom relation

➔Query:

select all building **from** classroom
where capacity < 100;

➔Output:

building
Painter
Taylor
Watson
Watson

Cartesian Product

Find the list of all students of departments which have a budget < $0.1million

select name, budget
from student, department
where student.dept name = department.dept name
and budget < 100000;

name	*budget*
Brandt	50000.00
Peltier	70000.00
Levy	70000.00
Sanchez	80000.00
Snow	70000.00
Aoi	85000.00
Bourikas	85000.00
Tanaka	90000.00

- The above query first generates every possible student department pair, which is the Cartesian product of student and department. Then, it filters all the rows with student.dept name = department.dept name **and** budget < 100000.

- The common attribute dept name in the resulting table are renamed using the relation name - student.dept name and department.dept name)

Rename AS Operation

- The same query in the previous slide can be framed by renaming the tables as shown below.

  select S.name **as** studentname, budget **as** deptbudget
  from student **as** S, department **as** D
  where S.dept name = D.dept name **and** budget < 100000;

name	*budget*
Brandt	50000.00
Peltier	70000.00
Levy	70000.00
Sanchez	80000.00
Snow	70000.00
Aoi	85000.00
Bourikas	85000.00
Tanaka	90000.00

- The above query renames the relation student **as** S and the relation department **as** D
- It also displays the attribute name as StudentName and budget as DeptBudget.
- Note that the budget attribute does not have any prefix because it occurs only in the department relation.

Where: AND and OR

- From the instructor and department relations in the figure, find out the names of all instructors whose department is Finance or whose department is in any of the following buildings: Watson, Taylor.

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

department

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

➔ Query:

select name
from instructor I, department D
where D.dept name = I.dept name
and (I.dept name = 'Finance'
or building **in** ('Watson','Taylor'));

➔ Output:

name
Srinivasan
Wu
Einstein
Gold
Katz
Singh
Crick
Brandt
Kim

String Operations

- From the course relation in the figure, find the titles of all courses whose course id has three alphabets indicating the department.

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure: course relation

➔ Query:

```
select title
from course
 where course_id like '___-%';
```

➔ Output:

title
Intro. to Biology
Genetics
Computational Biology
Investment Banking
World History
Physical Principles

- The course id of each department has either 2 or 3 alphabets in the beginning, followed by a hyphen and then followed by a 3-digit number. The above query returns the names of those departments that have 3 alphabets in the beginning.

Order By

- From the student relation in the figure, obtain the list of all students in alphabetic order of departments and within each department, in decreasing order of total credits.

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure: student relation

- The list is first sorted in alphabetic order of dept name.
- Within each dept, it is sorted in decreasing order of total credits.

➔ Query:

```
select name, dept name, tot cred
from student
order by dept name ASC, tot cred DESC;
```

➔ Output:

name	dept_name	tot_cred
Tanaka	Biology	120
Zhang	Comp. Sci.	102
Brown	Comp. Sci.	58
Williams	Comp. Sci.	54
Shankar	Comp. Sci.	32
Bourikas	Elec. Eng.	98
Aoi	Elec. Eng.	60
Chavez	Finance	110
Brandt	History	80
Sanchez	Music	38
Peltier	Physics	56
Levy	Physics	46
Snow	Physics	0

In Operator

- From the teaches relation in the figure, find the IDs of all courses taught in the Fall or Spring of 2018

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure: teaches relation

➔Query:

```
select course id
from teaches
where semester in ('Fall','Spring') and year=2018;
```

➔Output:

course_id
CS-315
FIN-201
MU-199
HIS-351
CS-101
CS-319
CS-319

Note: We can use **distinct** to remove duplicates.

Set Operations: union

- For the same question in the previous slide, we can find the solution using union operator as follows.

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure: teaches relation

➔Query:

```
select course id
from teaches
where semester='Fall'
    and year=2018
union
select course id
from teaches
where semester='Spring'
    and year=2018
```

➔ Output:

course_id
CS-101
CS-315
CS-319
FIN-201
HIS-351
MU-199

- Note that union removes all duplicates. If we use union all instead of union, we get the same set of tuples as in previous slide.

Set Operations (2): intersect

- From the instructor relation in the figure, find the names of all instructors who taught in either the Computer Science department or the Finance department and whose salary is < 80000.

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

➔Query:

```
select name
from instructor
where dept name in ('Comp. Sci.','Finance')
intersect
select name
from instructor
where salary < 80000;
```

⇨ Figure: instructor relation

➔Output:

name
Srinivasan
Katz

- Note that the same can be achieved using the query: select name from instructor where dept name in('Comp. Sci.', 'Finance') and salary < 80000;

Set Operations (3): except

- From the *instructor* relation in the figure, find the names of all instructors who taught in either the Computer Science department or the Finance department and whose salary is either ≥ 90000 or ≤ 70000.

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure: *instructor* relation

  - Query:

```
select name
from instructor
where dept_name in ('Comp. Sci.','Finance')
except
select name
from instructor
where salary < 90000 and salary > 70000;
```

  - Output:

name
Srinivasan
Brandt
Wu

  - Note that the same can be achieved using the query given below:

```
select name from instructor
where dept_name in('Comp. Sci.', 'Finance')
and (salary >= 90000 or salary <= 70000);
```

Aggregate functions: avg

- From the *classroom* relation given in the figure, find the names and the average capacity of each building whose average capacity is greater than 25.

building	room_number	capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure: *classroom* relation

  - Query:

```
select building, avg (capacity)
from classroom
group by building
having avg (capacity) > 25;
```

  - Output:

building	avg
Taylor	70.00
Packard	500.00
Watson	40.00

Aggregate functions (2): min

- From the instructor relation given in the figure, find the least salary drawn by any instructor among all the instructors.

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

➔Figure: instructor relation

➔Query:

```
select min(salary) as least salary
from instructor;
```

➔Output:

least_salary
40000.00

Aggregate functions (3): max

- From the student relation given in the figure, find the maximum credits obtained by any student among all the students.

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

→Figure: student relation

➔Query:

select max(tot cred) as max credits
from student;

➔ Output:

max_credits
120

Aggregate functions (4): count

- From the *section* relation given in the figure, find the number of courses run in each building.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

Figure: section relation

- Query:

select *building*,
count(*course_id*) **as** *course_count*
from *section*
group by *building*;

- Output:

building	*course_count*
Taylor	5
Packard	4
Painter	3
Watson	3

Aggregate functions (5): sum

- From the *course* relation given in the figure, find the total credits offered by each department.

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure: *course* relation

- Query:

select *dept_name*,
sum(*credits*) **as** *sum_credits*
from *course*
group by *dept_name*;

- Output:

dept_name	*sum_credits*
Finance	3
History	3
Physics	4
Music	3
Comp. Sci.	17
Biology	11
Elec. Eng.	3

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries
- A subquery is a select-from-where expression that is nested within another query
- The nesting can be done in the following SQL query

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  as follows:
  - A_i can be replaced by a subquery that generates a single value
  - r_i can be replaced by any valid subquery
  - P can be replaced with an expression of the form:

    B <operation> (subquery)

    where B is an attribute and <operation> to be defined later

Subqueries in the Where Clause

- Typical use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality

Set Membership

- Find courses offered in Fall 2009 and in Spring 2010. (**intersect** example)

```
select distinct course id
from section
where semester =’Fall’ and year = 2009 and
        course id in (select course id
        from section
        where semester =’Spring’ and year = 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010. (**except** example)

```
select distinct course id
from section
where semester =’Fall’ and year = 2009 and
        course id not in (select course id
        from section
        where semester =’Spring’ and year = 2010);
```

Set Membership (2)

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101

```
select count (distinct ID)
from takes
where (course id, sec id, semester, year) in
        (select course id, sec id, semester, year
        from teaches
        where teaches.ID = 10101);
```

- Note: Above query can be written in simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison – "some" Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept name = 'Biology');
```

Definition of "some" Clause

- F <comp> **some** r ⇔ ∃t ∈ r such that (F <comp> t)
  where <comp> can be: <, ≤, >, ≥, =, ≠
- **some** represents existential quantification

(5 < **some** {0, 5, 6}) = true (read: 5 < some tuple in the relation)

(5 < **some** {0, 5}) = false

(5 = **some** {0, 5}) = true

(5 ≠ **some** {0, 5}) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Set Comparison – "all" Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = 'Biology');
```

Definition of "all" Clause

- F <comp> **all** r ⇔ ∀t ∈ r such that (F <comp> t)
  Where <comp> can be: <, ≤, >, ≥, =, ≠
- **all** represents universal quantification

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≢ **in**

Test for Empty Relations: "exists"

- The exists construct returns the value true if the argument subquery is nonempty
  - exists r ⇔ r ≠ ∅
  - not exists r ⇔ r = ∅

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

```
select course id
from section as S
where semester = 'Fall' and year = 2009 and
        exists (select *
                from section as T
                where semester = 'Spring' and year = 2010
                and S.course id = T.course id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course id
                    from course
                    where dept name = 'Biology')
                    except
                    (select T.course id
                        from takes as T
                        where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took

- Note: X − Y = ∅ ⇔ X ⊆ Y
- Note: Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples: "unique"

- The **unique** construct tests whether a subquery has any duplicate tuples in its result
- The **unique** construct evaluates to "true" if a given subquery contains no duplicates
- Find all courses that were offered at most once in 2009

```
select T.course id
from course as T
where unique (select R.course id
                from section as R
                where T.course id = R.course id
                and R.year = 2009);
```

Subqueries in the From Clause

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000

```
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name) as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs
- Find all departments with the maximum budget

```
with max_budget(value) as
      (select max(budget)
      from department)
select department.name
from department, max_budget
where department.budget=max_budget.value;
```

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
            select dept_name, sum(salary)
            from instructor
            group by dept_name,
      dept_total_avg(value) as
            (select avg(value)
            from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

Subqueries in the Select Clause

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

select dept name,

```
(select count(*)
from instructor
where department.dept name = instructor.dept name)
as num instructors
from department;
```

• Runtime error if subquery returns more than one result tuple

Modifications of the Database

Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors
  delete from *instructor*
- Delete all instructors from the Finance department
  delete from *instructor*
  where *dept_name*= 'Finance';
- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building
  delete from *instructor*
  where *dept_name* **in** (**select** *dept_name*
  from *department*
  where *building* = 'Watson');

Deletion (2)

- Delete all instructors whose salary is less than the average salary of instructors
  delete from *instructor*
  where *salary* < (**select avg** (*salary*)
  from *instructor*);
- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
  a) First, compute **avg** (salary) and find all tuples to delete
  b) Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to course
  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
- or equivalently:
  insert into *course (course_id, title, dept_name, credits)*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
- Add a new tuple to student with *tot_creds* set to null
  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

Insertion (2)

- Add all instructors to the *student* relation with *tot_creds* set to 0

```
insert into student
select ID, name, dept_name, 0
    from instructor
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation
- Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem

Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

- The order is important
- Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with **case** statement

```
update instructor
set salary = case
                when salary <= 100000
                then salary * 1.05
                else salary * 1.03
                end
```

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
set tot_creds = (select sum(credits)
                from takes, course
                where takes.course_id = course.course_id and
                S.ID = takes.ID and
                takes.grade <> 'F' and
                takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
when sum(credits) is not null then sum(credits)
else 0
end
```

Join Expressions

Joined Relations

- **Join operations** take two relations and return as a result another relation
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition).
- It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause

Types of Join between Relations

- Cross join
- Inner join
  - Equi-join
    - Natural join
- Outer join
  - Left outer join
  - Right outer join
  - Full outer join
- Self-join

Cross Join

- CROSS JOIN returns the Cartesian product of rows from tables in the join
  - Explicit

```
select *
from employee cross join department;
```

  - Implicit

```
select *
from employee, department;
```

Join operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that
  prereq information is missing for CS-315 and
  course information is missing for CS-347

Inner Join

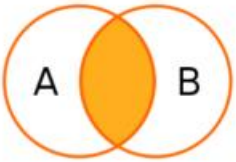
- *course* **inner join** *prereq*

course_id	title	dept_name	credits	prere_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- If specified as **natural**, the 2nd course_id field is skipped

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



Outer Join

- An extension of the join operation that avoids loss of information
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join
- Uses null values

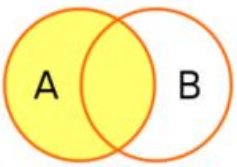
Left Outer Join

- *course* **natural left outer join** *prereq*

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



Right Outer Join

- *course* **natural right outer join** *prereq*

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Joined Relations

- Join operations take two relations and return as a result another relation
- These additional operations are typically used as subquery expressions in the **from** clause
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join
- Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

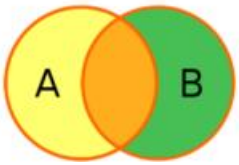
Full Outer Join

- *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



Joined Relations – Examples

- *course* **inner join** *prereq* **on** *course.course_id* = *prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above (equi_join), and a natural join?
- *course* **left outer join** *prereq* **on** *course.course_id* = *prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by
  select *ID, name, dept_name*
  from *instructor*
- A **view** provides a mechanism to hide certain data from the view of certain users
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form
  create view *v* **as** < query expression >
  where < query expression > is any legal SQL expression
- The view name is represented by *v*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view

Example Views

- A view of instructors without their salary
  create view *faculty* **as**
  select *ID, name, dept_name*
  from *instructor*
- Find all instructors in the Biology department
  select *name*
  from *faculty*
  where *dept_name* = 'Biology'
- Create a view of department salary totals
  create view *departments_total_salary(dept_name, total_salary)* **as**
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course.course_id, sec_id, building, room_number*
  from *course, section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2009';

- **create view** *physics_fall_2009_watson* **as**
  select *course_id, room_number*
  from *physics_fall_2009*
  where *building*= 'Watson';

View Expansion

- Expand use of a view in a query/another view

```
create view physics_fall_2009_watson as
    (select course_id, room_number
    from (select course.course_id, building, room_number
          from course, section
          where course.course_id = section.course_id
                and course.dept_name = 'Physics'
                and section.semester = 'Fall'
                and section.year = '2009')
    where building= 'Watson');
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v1 is said to depend directly on a view relation v2 if v2 is used in the expression defining v1
- A view relation v1 is said to depend on view relation v2 if either v1 depends directly on v2 or there is a path of dependencies from v1 to v2
- A view relation v is said to be recursive if it depends on itself

View Expansion*

- A way to define the meaning of views defined in terms of other views
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Update of a View

- Add a new tuple to *faculty* view which we defined earlier
  insert into *faculty* **values** ('30765', 'Green', 'Music');
- This insertion must be represented by the insertion of the tuple
  ('30765', 'Green', 'Music', null)
  into the *instructor* relation

Some Updates cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID, name, building*
  from *instructor, department*
  where *instructor.dept_name*= *department.dept_name*;
- **insert into** *instructor_info* **values** ('69987', 'White', 'Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group by** or **having** clause

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';
- What happens if we insert ('25566', 'Brown', 'Biology', 100000) into history_instructors?

Materialized Views

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated

Transactions

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (for example, using API)
  - In SQL:1999, can use: **begin atomic ... end**
    - Not supported on most databases

Integrity Constraints

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency
  - A checking account must have a balance greater than Rs. 10,000.00
  - A salary of a bank employee must be at least Rs. 250.00 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check**(P), where P is a predicate

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**

    budget **numeric**(12,2) **not null**
- **unique** $(A_1, A_2, \ldots, A_m)$
  - The unique specification states that the attributes $A_1, A_2, \ldots, A_m$ form a candidate key
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- check(P), where P is a predicate
- Ensure that semester is one of fall, winter, spring or summer:

```
create table section (
        course_id varchar(8),
        sec_id varchar(8),
        semester varchar(6),
        year numeric(4,0),
        building varchar(15),
        room_number varchar(7),
        time slot id varchar(4),
        primary key (course_id, sec_id, semester, year),
        check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation
- Example: If “Biology” is a department name appearing in one of the tuples in the instructor relation, then there exists a tuple in the department relation for “Biology”
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S

Cascading Actions in Referential Integrity

- With cascading, you can define the actions that the Database Engine takes when a user tries to delete or update a key to which existing foreign keys point
-
```
create table course (
        course_id char(5) primary key,
        title varchar(20),
        dept_name varchar(20) references department
)
```
-
```
create table course (
        ...
        dept_name varchar(20),
        foreign key (dept_name) references department
                on delete cascade
                on update cascade,
        ...
)
```
- Alternative actions to cascade: **no action, set null, set default**

Integrity Constraint Violation During Transactions

- **create table** *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID*,
  foreign key *father* **references** *person*,
  foreign key *mother* **references** *person*)
- How to insert a tuple without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - OR, Set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR Defer constraint checking (will discuss later)

SQL Data Types and Schemas

Built-in Data Types in SQL

- **date**: Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time**: Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval**: period of time
  - Example: **interval** '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Index Creation

- **create table** *student*
  (*ID* **varchar**(5),
  name **varchar**(20) **not null**,
  dept_name **varchar**(20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student(ID)*
- Indices are data structures used to speed up access to records with specified values for index attributes

  select *
  from *student*
  where *ID* = '12345'
  - Can be executed by using the index to find the required record, without looking at all records of student
  - *More on indices in Chapter 9*

User-Defined Types

- **create type** construct in SQL creates user-defined type (alias, like typedef in C)

  create type *Dollars* **as numeric** (12,2) **final**
  - **create table** *department* (
    dept_name **varchar** (20),
    building **varchar** (15),
    budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types
  create domain *person_name* **char**(20) **not null**
- Types and domains are similar
- Domains can have constraints, such as **not null**, specified on them
  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object – object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object – object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself

Authorization

Authorization

- Forms of authorization on parts of the database:
  - **Read** - allows reading, but not modification of data
  - **Insert** - allows insertion of new data, but not modification of existing data
  - **Update** - allows modification, but not deletion of data
  - **Delete** - allows deletion of data
- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices
  - **Resources** - allows creation of new relations
  - **Alteration** - allows addition or deletion of attributes in a relation
  - **Drop** - allows deletion of relations

Authorization Specification in SQL

- The **grant** statement is used to confer authorization
  grant <privilege list>
  on <relation name or view name> **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator)

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:
    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization
  revoke <privilege list>
  on <relation name or view name> **from** <user list>
- Example:
  revoke select on *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold
- If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation
- All privileges that depend on the privilege being revoked are also revoked

Roles

- **create role** *instructor*;
  grant *instructor* **to** Amit;
- Privileges can be granted to roles:
  grant select on takes **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  create role *teaching_assistant*
  grant *teaching_assistant* **to** *instructor*;
  - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** dean;
  - **grant** *instructor* **to** dean;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

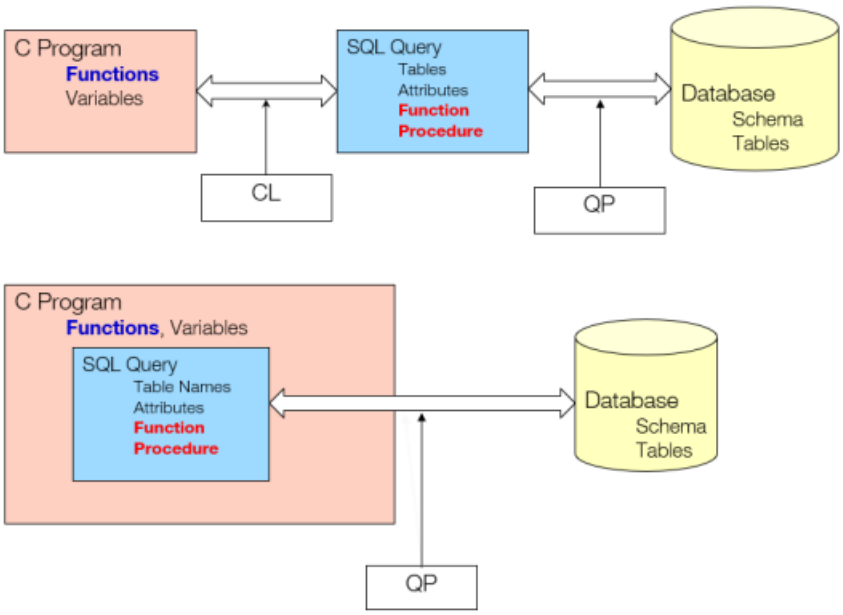
- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
  grant select on *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  select *
  from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor*?
  - creator of view did not have some permissions on *instructor*?

Other Authorization Features

- **references** privilege to create foreign key
  grant reference (*dept_name*) **on** *department* **to** Mariano;
  - why is this required?
- Transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;

Functions and Procedural Constructs

Native Language← →Query Language



Functions and Procedures

- Functions / Procedures and Control Flow Statements were added in SQL:1999
  - **Functions/Procedures** can be written in **SQL itself**, or in an **external programming language** (like C, Java)
  - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects
    - ▷ Example: Functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including **loops**, **if-then-else**, and **assignment**
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department:

```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count integer;
                select count (*) into d_count
                from instructor
                where instructor.dept_name = dept_name
        return d_cont;
        end
```

- The function *dept_count* can be used to find the department names and budget of all departments with more that 12 instructors:

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL functions (2)

- Compound statement: **begin** ... **end**
  May contain multiple SQL statements between **begin** and **end**.
- **returns** – indicates the variable-type that is returned (for example, integer)
- **return** – specifies the values that are to be returned as result of invoking the function
- SQL function are in fact **parameterized views** that generalize the regular notion of views by allowing parameters

Table Functions

- **Functions that return a relation as a result** added in SQL:2003
- Return all instructors in a given department:

```
create function instructor_of (dept_name char(20))
        returns table (
                ID varchar(5),
                name varchar(20),
                dept_name varchar(20)
                salary numeric(8, 2) )
        returns table
                (select ID, name, dept_name, salary
                from instructor
                where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *
from table  (instructor_of ('Music'))
```

SQL Procedures

- The dept_count function could instead be written as procedure:

```
create procedure dept_count_proc (
                in dept_name varchar (20), out d_count integer)
        begin
                select count(*) into d_count
                from instructor
                where instructor.dept_name = dept_count_proc.dept_name
        end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows **overloading** - more than one function/procedure of the same name as long as the number of arguments and / or the types of the arguments differ

Language Constructs for Procedures and Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - *Warning*: **Most database systems implement their own variant of the standard syntax**
- Compound statement: **begin ... end**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements

Language Constructs (2): while and repeat

- **while** loop:

```
while boolean expression do
        sequence of statements;
end while;
```

- **repeat** loop:

```
repeat
        sequence of statements;
 until boolean expression
 end repeat;
```

Language Constructs (3): for

- **for** loop
  - Permits iteration over all results of a query
- Find the budget of all departments:

```
declare n integer default 0;
for r as
    select budget from department
do
    set n = n + r.budget
end for;
```

Language Constructs (4): if-then-else

- Conditional statements
  - **if-then-else**
  - **case**
- **if-then-else** statement

```
if boolean expression then
        sequence of statements;
elseif boolean expression then
        sequence of statements;
 ...
else
        sequence of statements;
end if;
```

- The **if** statement supports the use of optional **elseif** clauses and a default **else** clause.
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book (page 177) for details

Language Constructs (5): Simple case

- Simple **case** statement

```
case variable
        when value1 then
                sequence of statements;
        when value2 then
                sequence of statements;
         ...
        else
                sequence of statements;
end case;
```

- The **when** clause of the **case** statement defines the value that when satisfied determines the flow of control

Language Constructs (6): Searched case

- Searched **case** statement

```
case
        when sql-expression = value1 then
                sequence of statements;
        when sql-expression = value2 then
                sequence of statements;
        ...
        else
                sequence of statements;
end case;
```

- Any supported SQL expression can be used here. These expressions can contain references to variables, parameters, special registers, and more.

Language Constructs (7): Exception

- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
        ...
        signal out_of_classroom_seats
        ...
end
```

  - The handler here is **exit** – causes enclosing **begin** . . . **end** to be terminate and exit
  - Other actions possible on exception

External Language Routines*

- SQL:1999 allows the definition of functions / procedures in an imperative programming language, (Java, C#, C or C++) which can be invoked from SQL queries
- Such functions can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions
- Declaring external language procedures and functions

```
create procedure dept_count_proc(
                in dept_name varchar(20),
                out count integer)
        language C
        external name '/usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
        language C
        external name '/usr/avi/bin/dept_count'
```

External Language Routines (2)*

- Benefits of external language functions/procedures:
  - More efficient for many operations, and more expressive power

External Language Routines (3)*: Security

- To deal with security problems, we can do one of the following:
  - Use **sandbox** techniques
    - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Run external language functions/procedures in a separate process, with no access to the database process' memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space

Triggers

Triggers

- A **trigger** defines a set of actions that are performed in response to an **insert**, **update**, or **delete** operation on a specified table
  - When such an SQL operation is executed, the trigger is said to have been **activated**
  - Triggers are **optional**
  - Triggers are defined using the **create trigger** statement
- Triggers can be used
  - To enforce data integrity rules via referential constraints and check constraints
  - To cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts
- To design a trigger mechanism, we must:
  - Specify the **events** / (like **update**, **insert**, or **delete**) for the trigger to executed
  - Specify the **time** (**BEFORE** or **AFTER**) of execution
  - Specify the **actions** to be taken when the trigger executes
- **Syntax of triggers may vary across systems**

Types of Triggers: BEFORE

- **BEFORE triggers**
  - Run before an **update**, or **insert**
  - Values that are being updated or inserted can be modified before the database is actually modified. You can use triggers that run before an update or insert to:
    - Check or modify values before they are actually updated or inserted in the database
      - Useful if user-view and internal database format differs
    - Run other non-database operations coded in user-defined functions
- **BEFORE DELETE triggers**
  - Run before a **delete**
    - Checks values (a raises an error, if necessary)

Types of Triggers (2): AFTER

- **AFTER triggers**
  - Run after an **update**, **insert**, or **delete**
  - You can use triggers that run after an update or insert to:
    - ▷ Update data in other tables
      - – Useful for maintain relationships between data or keep audit trail
    - ▷ Check against other data in the table or in other tables
      - – Useful to ensure data integrity when referential integrity constraints aren't appropriate, or
      - – when table check constraints limit checking to the current table only
    - ▷ Run non-database operations coded in user-defined functions
      - – Useful when issuing alerts or to update information outside the database

Row Level and Statement Level Triggers

There are two types of triggers based on the level at which the triggers are applied:

- **Row level triggers** are executed whenever a row is affected by the event on which the trigger is defined.
  - Let Employee be a table with 100 rows. Suppose an **update** statement is executed to increase the salary of each employee by 10%. Any row level **update** trigger configured on the table Employee will affect all the 100 rows in the table during this update.
- **Statement level triggers** perform a single action for all rows affected by a statement, instead of executing a separate action for each affected row.
  - Used for each **statement** instead of for each **row**
  - Uses **referencing old table** or **referencing new table** to refer to temporary tables called **transition tables** containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

Triggering Events and Actions in SQL

- Triggering event can be an **insert, delete or update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
    begin atomic
      set nrow.grade = null;
    end;
```

Trigger to Maintain credits earned value

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <>'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
begin atomic
        update student
        set tot_cred= tot_cred +
            (select  credits
            from course
            where  course.course_id=nrow.course_id)
        where student.id = nrow.id;
end;
```

How to use triggers?

- The optimal use of DML triggers is for short, simple, and easy to maintain write operations that act largely independent of an applications business logic.
- Typical and recommended uses of triggers include:
  - Logging changes to a history table
  - Auditing users and their actions against sensitive tables
  - Adding additional values to a table that may not be available to an application (due to security restrictions or other limitations), such as:
    - Login/user name
    - Time an operation occurs
    - Server/database name
  - Simple validation

How not to use triggers?

- Triggers are like Lays: *Once you pop, you can't stop*
- One of the greatest challenges for architects and developers is to ensure that
  - triggers are used only as needed, and
  - to not allow them to become a one-size-fits-all solution for any data needs that happen to come along
- Adding triggers is often seen as faster and easier than adding code to an application, but the cost of doing so is compounded over time with each added line of code

How to use triggers? (2)

- Triggers can become dangerous when:
  - There are too many
  - Trigger code becomes complex
  - Triggers go cross-server - across databases over network
  - Triggers call triggers
  - Recursive triggers are set to ON. This database-level setting is set to off by default
  - Functions, stored procedures, or views are in triggers
  - Iteration occurs