f

**21CSC205P-Database Management Systems**

UNIT-V

**TOPICS**

• Storage Structure

• Transaction control

• Concurrency control algorithms and Graph

• Issues in Concurrent execution

• Failures and Recovery algorithms

• Case Study: Demonstration of Entire project by applying all the concepts learned with minimum Front-End requirements, NoSQL Database, Document Oriented, Key Value pairs, Column Oriented

**Storage Structure**

• Overview of Physical Storage Media • Magnetic Disks

• RAID

• Tertiary Storage

• Storage Access

• File Organization

• Organization of Records in Files • Data-Dictionary Storage

**Physical Storage Media**

**Classification of Physical Storage Media**

• Speed with which data can be accessed

• Cost per unit of data

• Reliability

o data loss on power failure or system crash

o physical failure of the storage device

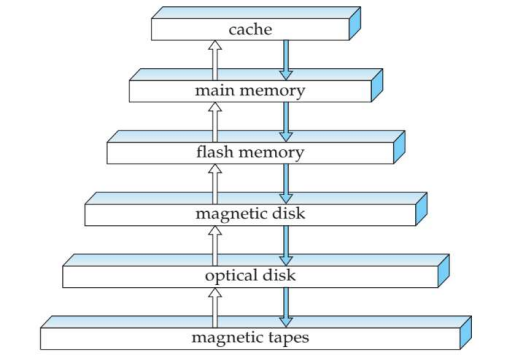
• Can differentiate storage into:

o **Volatile storage:** loses contents when power is switched off

o **Non-volatile storage:**

▪ Contents persist even when power is switched off.

▪ Includes secondary and tertiary storage, as well as batter- backed up main-memory.

**Storage Device Hierarchy**

**Physical Storage Media**

**1. Cache:** fastest and most costly form of storage; volatile; managed by the computer system hardware.

**2. Main memory:**

• fast access (10s to 100s of nanoseconds; 1 nanosecond = 10–9seconds) • generally too small (or too expensive) to store the entire database • capacities of up to a few Gigabytes widely used currently

• Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)

• Volatile: contents of main memory are usually lost if a power failure or system crash occurs.

**3. Flash Memory**

• Data survives power failure

• Data can be written at a location only once, but location can be erased and written to again

▪ Can support only a limited number (10K – 1M) of write/erase cycles. ▪ Erasing of memory has to be done to an entire bank of memory • Reads are roughly as fast as main memory

• But writes are slow (few microseconds), erase is slower

• Cost per unit of storage roughly similar to main memory

• Widely used in embedded devices such as digital cameras, phones, and USB keys

• Is a type of EEPROM (Electrically Erasable Programmable Read-Only Memory)

**4. Magnetic-disk Storage**

• Data is stored on spinning disk, and read/written magnetically

• Primary medium for the long-term storage of data; typically stores entire database. • Data must be moved from disk to main memory for access, and written back for storage • Much slower access than main memory

• direct-access – possible to read data on disk in any order, unlike magnetic tape • Capacities range up to roughly 400 GB currently

• Much larger capacity and cost/byte than main memory/flash memory

• Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years) • Survives power failures and system crashes

• disk failure can destroy data, but is rare

**5. 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) most popular forms • Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)

• Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD RAM) • Reads and writes are slower than with magnetic disk

• Juke-box systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data.

**6. Tape Storage**

• Non-volatile, used primarily for backup (to recover from disk failure), and for archival data

• **Sequential-access** – much slower than disk

• Very high capacity (40 to 300 GB tapes available)

• Tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive

• Tape jukeboxes available for storing massive amounts of data • hundreds of terabytes (1 terabyte = 109 bytes) to even multiple **petabytes** (1 petabyte = 1012 bytes)

**Storage Hierarchy (Cont.)**

• **Primary Storage:** Fastest media but volatile (cache, main memory).

• **Secondary Storage:** next level in hierarchy, non-volatile, moderately fast access time

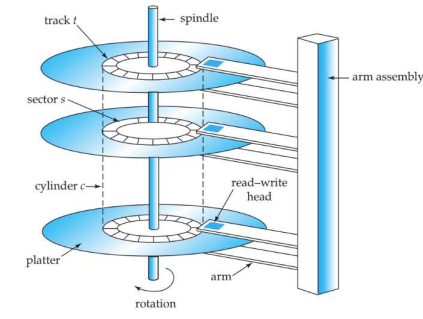
• also called **on-line storage**

• E.g. flash memory, magnetic disks

• **Tertiary Storage:** lowest level in hierarchy, non-volatile, slow access time

• also called **off-line storage**

• E.g. magnetic tape, optical storage

**Magnetic Disk and Flash Storage **

**Moving Head Disk Mechanism**

**Magnetic Disks**

**Physical Characteristics of Disks**

• **Read-write head**

• Positioned very close to the platter surface

• Reads or writes 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 typically 512 bytes

• Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks) • To read/write a sector

• disk arm swings to position head on right track

• platter spins continually; data is read/written as sector passes 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.

• **Cylinder** *i* consists of *i*th track of all the platters

**Magnetic Disks (Cont.)**

• Earlier generation disks were susceptible to head-crashes

• Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk

• Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted

• **Disk controller** – interfaces between the computer system and the disk drive hardware. • accepts high-level commands to read or write a sector

• initiates actions such as moving the disk arm to the right track and actually reading or writing the data

• Computes and attaches **checksums** to each sector to verify that data is read back correctly

• If data is corrupted, with very high probability stored checksum won’t match recomputed checksum

• Ensures successful writing by reading back sector after writing it

• Performs remapping of bad sectors

**Disk Subsystem**

• Multiple disks connected to a computer system through a controller

• Controllers functionality (checksum, bad sector remapping) often carried out by individual disks; reduces load on controller

• Disk interface standards families

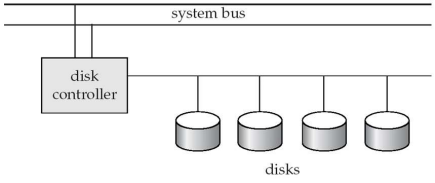
• ATA (AT adaptor) range of standards

• SATA (Serial ATA)

• SCSI (Small Computer System Interconnect) range of standards

• SAS (Serial Attached SCSI)

• Several variants of each standard (different speeds and capabilities)



**Disk Subsystem (cont.)**

• Disks usually connected directly to computer system

• In **Storage Area Networks (SAN)**, a large number of disks are connected by a high-speed network to a number of servers

• In **Network Attached Storage (NAS)** networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface

**Performance Measures of Disks**

• **Access time** – the time it takes from when a read or write request is issued to when data transfer begins.

• **Seek time** – time it takes to reposition the arm over the correct track.

• Average seek time is 1/2 the worst case seek time.

• Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement

• 4 to 10 milliseconds on typical disks

• **Rotational latency** – time it takes for the sector to be accessed to appear under the head. • Average latency is 1/2 of the worst case latency.

• 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)

• **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk. • 25 to 100 MB per second max rate, lower for inner tracks

• Multiple disks may share a controller, so rate that controller can handle is also important • E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)

• Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)

• Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s

• **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.

**Flash Storage**

• NOR flash vs NAND flash

• NAND flash

• used widely for storage, since it is much cheaper than NOR flash

• requires page-at-a-time read (page: 512 bytes to 4 KB)

• transfer rate around 20 MB/sec

• **solid state disks**: use multiple flash storage devices to provide higher transfer rate of 100 to 200 MB/sec

• erase is very slow (1 to 2 millisecs)

• erase block contains multiple pages

• **remapp**ing of logical page addresses to physical page addresses avoids waiting for erase • **translation table** tracks mapping

• also stored in a label field of flash page

• remapping carried out by **flash translation layer**

• after 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used • **wear leveling**

**Redundant Array of Independent Disks (RAID)**

• RAID is a technology that uses multiple physical disk drives to protect data from a single disk failure.

• The purpose of RAID is to ensure that at the time of failure, there should be one copy of data which should be available for immediate use.

• RAID levels define the use of disk arrays.

**RAID levels**

• RAID 0

• RAID 1

• RAID 2

• RAID 3

• RAID 4

• RAID 5

• RAID 6

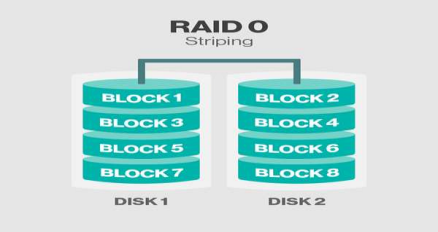
**RAID 0**

- RAID 0 consists of striping, but no mirroring or parity, but no redundancy of data. It offers the best performance, but no fault tolerance.

- In this level, a striped array of disks is implemented. The data is broken down into blocks and the blocks are distributed among disks.

- Block “1, 2” forms a stripe.

- Each disk receives a block of data to write/read in parallel.

- Reliability: there is no duplication of data. Hence, a block once lost cannot be recovered.

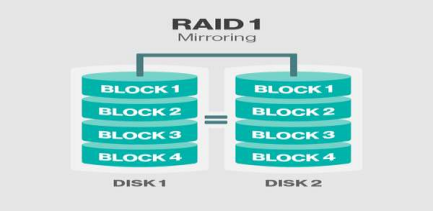
**RAID 1**

• RAID 1 is also known as *disk mirroring*, this configuration consists of at least two drives that duplicate the storage of data.

• There is no striping. When data is sent to a RAID controller, it sends a copy of data to all the disks in the array.

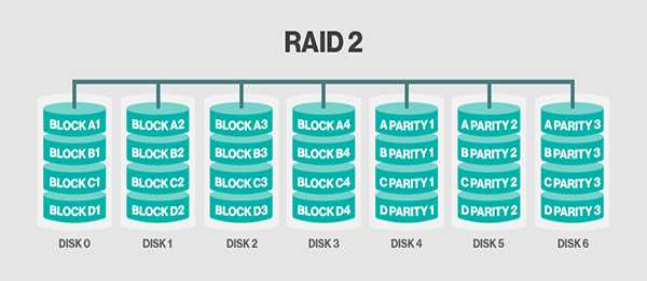
• Read performance is improved since either disk can be read at the same time.

• Write performance is the same as for single disk storage. (This level performs mirroring of data in drive 1 to drive 2. It offers 100% redundancy as array will continue to work even if either disk fails.)



**RAID 2**

• RAID2 uses striping across disks, with some disks storing error checking and correcting (ECC) information.

• This level uses bit-level data stripping rather than block level. • It uses an extra disk for storing all the parity information.

**RAID 3**

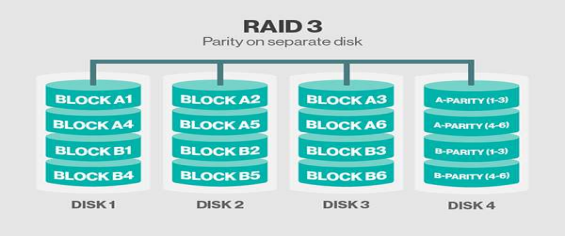
• This technique uses striping and dedicates one drive to storing parity information. So RAID 3 stripes the data onto multiple disks.

• The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failures.

• The ECC information is used to detect errors.

• This level uses byte level stripping along with parity.

• One dedicated drive is used to store the parity information and in case of any drive failure the parity is restored using this extra drive.

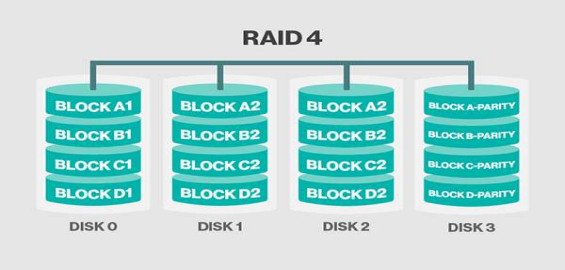
• But in case the parity drive crashes then the redundancy gets affected again so not much considered in organizations.

**RAID 4**

• In this level, an entire block of data is written onto data disks and then the parity is generated and stored on a different disk. Note that level 3 uses byte-level striping, whereas level 4 uses block level striping. Both level 3 and level 4 require at least three disks to implement RAID.

• This level uses large stripes, which means you can read records from any single drive.

• This level is very much similar to RAID 3 apart from the feature where RAID 4 uses block level stripping rather than byte level.



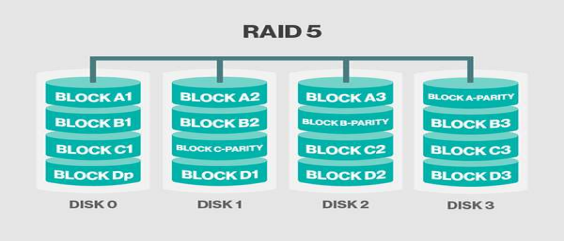
**RAID 5**

• This level is based on block level striping with distributed parity.

• The parity information is striped across each drive.

• RAID 5 requires at least three disks, but it is often recommended to use at least five disks for performance reasons.

• Parity information is written to a different disk in the array for each stripe.

• In case of single disk failure data can be recovered with the help of distributed parity. • The parity bit rotates among the drives to make the random write performance better.

**RAID 6 (P+Q Redundancy Scheme)**

• RAID 6 is an extension of level 5. In this level, two independent parities are generated and stored in distributed fashion among multiple disks. Two parities provide additional fault tolerance. This level requires at least four disk drives to implement RAID.

• The use of additional parity allows the array to continue to function even if two disks fail simultaneously. However, this extra protection have a higher cost per gigabyte(GB).

• This level is an enhanced version of RAID 5 adding extra benefit of dual parity (2 parity blocks are created.)

• This level uses block level stripping with DUAL distributed parity and can survive concurrent 2 drive failures in an array which leads to extra fault tolerance and

redundancy.



**Tertiary Storage**

• In a large database system, some of the data may have to reside on tertiary storage. • The two most common tertiary storage media are optical disks and magnetic tapes. 1. Optical Disks

2. Magnetic Tapes

**1. Optical Disks**

• Compact disk-read only memory (CD-ROM)

• Removable disks, 640 MB per disk

• Seek time about 100 msec (optical read head is heavier and slower)

• Higher latency (3000 RPM) and lower data-transfer rates (3-6 MB/s) compared to magnetic disks

• Digital Video Disk (DVD)

• DVD-5 holds 4.7 GB , and DVD-9 holds 8.5 GB

• DVD-10 and DVD-18 are double sided formats with capacities of 9.4 GB and 17 GB • Blu-ray DVD: 27 GB (54 GB for double sided disk)

• Slow seek time, for same reasons as CD-ROM

• Record once versions (CD-R and DVD-R) are popular

• data can only be written once, and cannot be erased.

• high capacity and long lifetime; used for archival storage

• Multi-write versions (CD-RW, DVD-RW, DVD+RW and DVD-RAM) also available

**2. Magnetic Tapes**

• Hold large volumes of data and provide high transfer rates

• Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format • Transfer rates from few to 10s of MB/s

• Tapes are cheap, but cost of drives is very high

• Very slow access time in comparison to magnetic and optical disks

• limited to sequential access.

• Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity

• Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.

• Tape jukeboxes used for very large capacity storage

• Multiple petabyes (1015 bytes)

**File Organization**

• The database is stored as a collection of files. Each file is a sequence of *records.* A record is a sequence of fields.

• One approach:

• assume record size is fixed

• each file has records of one particular type only

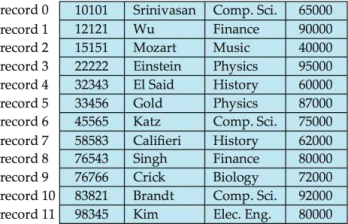
• different files are used for different relations

**1. Fixed Length Records**

• Simple approach:

• Store record *i* starting from byte *n* \* *(i –* 1), where *n* is the size of each record. • Record access is simple but records may cross blocks

• Modification: do not allow records to cross block boundaries

• Deletion of record *i:* 

alternatives*:*

• move records *i* + 1, . . ., *n*

to *i, . . . , n –* 1

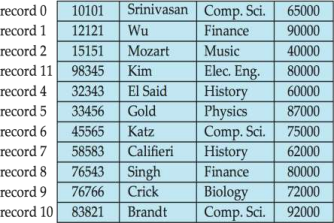
• move record *n* to *i*

• do not move records, but

link all free records on a

*free list.*

**Deleting Record 3 and Compacting Deleting Record 3 and Moving Last Record**

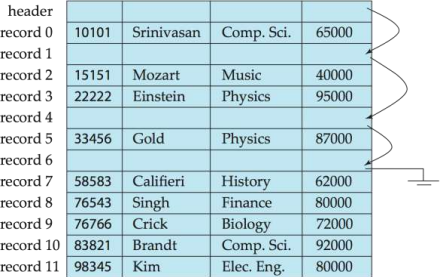
****

**Free Lists**

• Store the address of the first deleted record in the file header.

• Use this first record to store the address of the second deleted record, and so on • Can think of these stored addresses as pointers since they “point” to the location of a record.

• More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



**2. Variable-Length Records**

• Variable-length records arise in database systems in several ways:

• Storage of multiple record types in a file.

• Record types that allow variable lengths for one or more fields such as strings (**varchar**) • Record types that allow repeating fields (used in some older data models). • Attributes are stored in order

• Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes

• Null values represented by null-value bitmap



**Variable-Length Records: Slotted Page Structure**

• **Slotted page** header contains: 

• number of record entries

• end of free space in the block

• location and size of each record

• Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.

• Pointers should not point directly to record — instead they should point to the entry for the record in header.

**Organization of Records in Files**

• **Heap** – a record can be placed anywhere in the file where there is space

• **Sequential** – store records in sequential order, based on the value of the search key of each record

• **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

• Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file

• store related records on the same block to minimize I/O

**Sequential File Organization**

• Suitable for applications that require sequential processing of the entire file. • The records in the file are ordered by a search-key



**Sequential File Organization (Cont.)**

• Deletion – use pointer chains

• Insertion –locate the position where the record is to be inserted • if there is free space insert there

• if no free space, insert the record in an overflow block • In either case, pointer chain must be updated 

• Need to reorganize the file

from time to time to restore

sequential order.

**Multitable Clustering File Organization**

• Store several relations in one file using a **multitable clustering** file organization Department Instructor 

Multitable clustering 

of department and

instructor

**Data Dictionary Storage**

• The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

• Information about relations

• names of relations

• names, types and lengths of attributes of each relation

• names and definitions of views

• integrity constraints

• User and accounting information, including passwords

• Statistical and descriptive data

• number of tuples in each relation

• Physical file organization information

• How relation is stored (sequential/hash/…)

• Physical location of relation

• Information about indices

**Relational Representation of System Metadata**

• Relational representation on disk

• Specialized data structures designed for efficient access, in memory

**Storage Access**

• A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.

• Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

• **Buffer** – portion of main memory available to store copies of disk blocks.

• **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

**Buffer Manager**

• Programs call on the buffer manager when they need a block from disk. 1. If the block is already in the buffer, buffer manager returns the address of the block in main memory

2. If the block is not in the buffer, the buffer manager

1. Allocates space in the buffer for the block

▪ Replacing (throwing out) some other block, if required, to make space for the new block.

▪ Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.

2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



Transaction control

**Database System Concepts - 7 17.1 ©Silberschatz, Korth and Sudarshan th Edition**

**Transaction control**

▪ Transaction Concept

▪ Transaction State

▪ Concurrent Executions

▪ Serializability

▪ Testing for conflict and View Serializability. ▪ Recoverability

✔ Cascading rollback

✔ Cascade less

**Transaction Concept**

▪ A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

▪ E.g., transaction to transfer $50 from account A to account B: 1. **read**(*A*)

2. *A* := *A –* 50

3. **write**(*A*)

4. **read**(*B*)

5. *B* := *B +* 50

6. **write**(*B)*

▪ Two main issues to deal with:

• Failures of various kinds, such as hardware failures and system crashes

• Concurrent execution of multiple transactions

**Example of Fund Transfer**

▪ Transaction to transfer $50 from account A to account B:

1. **read**(*A*)

2. *A* := *A –* 50

3. **write**(*A*)

4. **read**(*B*)

5. *B* := *B +* 50

6. **write**(*B)*

▪ **Atomicity requirement**

• If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state

▪ Failure could be due to software or hardware

• The system should ensure that updates of a partially executed transaction are not reflected in the database

▪ **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

**Example of Fund Transfer (Cont.)**

▪ **Consistency requirement** in above example:

• The sum of A and B is unchanged by the execution of the transaction ▪ In general, consistency requirements include

• Explicitly specified integrity constraints such as primary keys and foreign keys

• Implicit integrity constraints

▪ e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

• A transaction must see a consistent database.

• During transaction execution the database may be temporarily inconsistent.

• When the transaction completes successfully the database must be consistent

▪ Erroneous transaction logic can lead to inconsistency

**Example of Fund Transfer (Cont.)**

▪ **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A + B* will be less than it should be).

**T1 T2**

1. **read**(*A*)

2. *A* := *A –* 50

3. **write**(*A*)

read(A), read(B), print(A+B)

4. **read**(*B*)

5. *B* := *B +* 50

6. **write**(*B*

▪ Isolation can be ensured trivially by running transactions **serially** • That is, one after the other.

▪ However, executing multiple transactions concurrently has significant benefits, as we will see later.

**ACID Properties**

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

▪ **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.

▪ **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

▪ **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

• That is, for every pair of transactions *Ti* and *Tj,* it appears to *Ti*that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti*finished.

▪ **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

**Transaction State**

▪ **Active** – the initial state; the transaction stays in this state while it is executing

▪ **Partially committed** – after the final statement has been executed. ▪ **Failed --** after the discovery that normal execution can no longer proceed. ▪ **Aborted** – after the transaction has been rolled back and the database

restored to its state prior to the start of the transaction. Two options after it has been aborted:

• Restart the transaction

▪ Can be done only if no internal logical error

• Kill the transaction

▪ **Committed** – after successful completion.

**Transaction State (Cont.)**

****

****

**Concurrent Executions**

▪ Multiple transactions are allowed to run concurrently in the system.

Advantages are:

• **Increased processor and disk utilization**, leading to better

transaction *throughput*

▪ E.g., one transaction can be using the CPU while another is

reading from or writing to the disk

• **Reduced average response time** for transactions: short transactions need not wait behind long ones.

▪ **Concurrency control schemes** – mechanisms to achieve isolation

• That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

▪ Will study in Chapter 15, after studying notion of correctness of

concurrent executions.



**Schedules**

▪ **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

• A schedule for a set of transactions must consist of all instructions of those transactions

• Must preserve the order in which the instructions appear in each

individual transaction.

▪ A transaction that successfully completes its execution will have a commit instructions as the last statement

• By default transaction assumed to execute commit instruction as its last step

▪ A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



**Schedule 1**

▪ Let *T*1transfer $50 from *A* to *B*, and *T*2transfer 10% of the balance from *A* to *B.*

▪ A serial schedule in which *T*1is followed by *T*2:



**Schedule 2**

▪ A serial schedule where *T2*is followed by *T*1



**Schedule 3**

▪ Let *T*1 and *T*2 be the transactions defined previously*.* The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

▪ In Schedules 1, 2 and 3, the sum A + B is preserved.

**Database System Concepts - 7 17.14 ©Silberschatz, Korth and Sudarshan th Edition**

**Schedule 4** ▪ The following concurrent schedule does not preserve the value of (*A* + *B )*.

**Database System Concepts - 7 17.15 ©Silberschatz, Korth and Sudarshan th Edition**

**Serializability**

▪ **Basic Assumption** – Each transaction preserves database consistency. ▪ Thus, serial execution of a set of transactions preserves database

consistency.

▪ A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **Conflict serializability**

2. **View serializability**

****

**Conflict Serializability**

▪ If a schedule *S* can be transformed into a schedule *S’* by a series of swaps of non-conflicting instructions, we say that *S* and *S’* are **conflict**

**equivalent***.*

▪ We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule



**Conflict Serializability (Cont.)**

▪ Schedule 3 can be transformed into Schedule 6, a serial schedule where *T*2 follows *T*1, by series of swaps of non-conflicting instructions. Therefore

Schedule 3 is conflict serializable.

Schedule 3 Schedule 6



**Conflict Serializability (Cont.)**

▪ Example of a schedule that is not conflict serializable:

▪ We are unable to swap instructions in the above schedule to obtain either the serial schedule < *T*3, *T*4 >, or the serial schedule < *T*4, *T*3 >.



***Simplified view of transactions***

▪ We ignore operations other than **read** and **write** instructions

▪ We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

▪ Our simplified schedules consist of only **read** and **write** instructions.



**Conflicting Instructions**

▪ Instructions *li* and *lj* of transactions *Ti* and *Tj*respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q.*

1. *li* = **read**(*Q), lj =* **read**(*Q*). *li* and *lj* don’t conflict.

2. *li* = **read**(*Q), lj =* **write**(*Q*). They conflict.

3. *li* = **write**(*Q), lj =* **read**(*Q*). They conflict

4. *li* = **write**(*Q), lj =* **write**(*Q*). They conflict

▪ Intuitively, a conflict between *li* and *lj*forces a (logical) temporal order between them.

▪ If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

**View Serializability**

▪ Let *S* and *S’* be two schedules with the same set of transactions. *S* and *S’* are **view equivalent** if the following three conditions are met, for each data item *Q,*

1. If in schedule S, transaction *Ti*reads the initial value of *Q*, then in

schedule *S’* also transaction *Ti* must read the initial value of *Q.*

2. If in schedule S transaction *Ti* executes **read**(*Q)*, and that value was produced by transaction *Tj*(if any), then in schedule *S’* also

transaction *Ti* must read the value of *Q* that was produced by the

same **write**(Q) operation of transaction *Tj*.

3. The transaction (if any) that performs the final **write**(*Q*) operation in

schedule *S* must also perform the final **write**(*Q*) operation in schedule *S’.* ▪ As can be seen, view equivalence is also based purely on **reads** and **writes** alone.



**View Serializability (Cont.)**

▪ A schedule *S* is **view serializable** if it is view equivalent to a serial

schedule.

▪ Every conflict serializable schedule is also view serializable.

▪ Below is a schedule which is view-serializable but *not* conflict serializable.

▪ What serial schedule is above equivalent to?

▪ Every view serializable schedule that is not conflict serializable has **blind writes.**

****

**Other Notions of Serializability**

▪ The schedule below produces same outcome as the serial schedule < *T*1, *T*5 >, yet is not conflict equivalent or view equivalent to it.

▪ Determining such equivalence requires analysis of operations other

than read and write.



**Testing for Serializability**

▪ Consider some schedule of a set of transactions *T*1, *T*2, ..., *Tn*

▪ **Precedence graph** — a direct graph where the vertices are the

transactions (names).

▪ We draw an arc from *Ti*to *Tj*if the two transaction conflict, and *Ti*

accessed the data item on which the conflict arose earlier.

▪ We may label the arc by the item that was accessed.

▪ Example of a precedence graph



**Test for Conflict Serializability**

▪ A schedule is conflict serializable if and only if

its precedence graph is acyclic.

▪ Cycle-detection algorithms exist which take

order *n*2time, where *n* is the number of

vertices in the graph.

• (Better algorithms take order *n* + *e* where

*e* is the number of edges.)

▪ If precedence graph is acyclic, the

serializability order can be obtained by a

*topological sorting* of the graph.

• This is a linear order consistent with the

partial order of the graph.

• For example, a serializability order for

Schedule A would be

*T*5 → *T*1 → *T*3 → *T*2 → *T*4

▪ Are there others?



**Test for View Serializability**

▪ The precedence graph test for conflict serializability cannot be used

directly to test for view serializability.

• Extension to test for view serializability has cost exponential in the

size of the precedence graph.

▪ The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.

• Thus, existence of an efficient algorithm is *extremely* unlikely.

▪ However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.



**Recoverable Schedules**

Need to address the effect of transaction failures on concurrently

running transactions.

▪ **Recoverable schedule** — if a transaction *Tj*reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti* appears before the commit operation of *Tj.*

▪ The following schedule (Schedule 11) is not recoverable

▪ If *T*8 should abort, *T*9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.



**Cascading Rollbacks**

▪ **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

If *T*10 fails, *T*11 and *T*12 must also be rolled back.

▪ Can lead to the undoing of a significant amount of work



**Cascadeless Schedules**

▪ **Cascadeless schedules** — cascading rollbacks cannot occur;

• For each pair of transactions *Ti* and *Tj* such that *Tj*reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the

read operation of *Tj*.

▪ Every Cascadeless schedule is also recoverable

▪ It is desirable to restrict the schedules to those that are cascadeless



CONCURRENCY CONTROL

CONCURRENCY CONTROL

When several transactions execute concurrently in the database, the consis?tency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called concurrency-control schemes.

There are a variety of concurrency-control schemes. No one scheme is clearly the best; each one has advantages. Some of the protocols used are :

1.Lock Based Protocols

2. Deadlock Handling

3.Multiple Granularity

4. Timestamp-Based Protocols

5. Validation-Based Protocols

6. Multiversion Schemes

LOCK BASED PROTOCOLS

Shared-exclusive

protocol

• Shared Locks (S) : If transaction locked data item in shared mode then allowed to read only.

• Exclusive locks (X) : If transaction locked data item in exclusive mode then allowed to read and write both.

If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function can be represented conveniently by a matrix.

An element comp(A, B) of the matrix has the value true if and only if mode A is compatible with mode B

Reques 

t



DRAWBACKS OF SHARED EXCLUSIVE

LOCKING

• The protocol may not be sufficient to produce serializable schedule only, which means it cannot provide consistent data at times. 

• If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states.

• If we do not unlock an item before requesting a lock on another item, deadlocks may occur.

• In case of sequence of request for shared locks, each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock. Hence , the transaction is starved.

2 PHASED LOCKING(2PL)

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase : A transaction may obtain locks, but may not release any lock. 2. Shrinking phase : A transaction may release locks, but may not obtain any new locks. 

Modifications of the 2PL protocol include :-

• Strict two-phase locking protocol- This protocol

requires not only that locking be two phase, but also

that all exclusive-mode locks taken by a transaction be

held until that transaction commits

• Rigorous two-phase locking protocol -which requires

that all locks be held until the transaction commits.

DRAWBACKS OF 2PL PROTOCOL

1.The protocol may not be free from irrecoverability. 2.It may not be free from deadlocks.

3.It may not be free from starvation

4.It may not be free from cascading rollback.

The lock manager uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the lock table. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.



DEADLOCK

HANDLING

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.

We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state.

Two different deadlock-prevention schemes using timestamps have been proposed: 1. The wait–die scheme is a non preemptive technique.

2. The wound–wait scheme is a preemptive technique.

Another simple approach to deadlock prevention is based on lock timeouts.

Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery scheme.