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^{-9}seconds)
- 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 $\Rightarrow$ storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
  - hundreds of terabytes (1 terabyte = 10^9 bytes) to even multiple **petabytes** (1 petabyte = 10^{12} 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 ith 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 (10^{15} 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 $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record n to i
  - do not move records, but link all free records on a *free list.*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting Record 3 and Compacting

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting Record 3 and Moving Last Record

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

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.)

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

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

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	

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.

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	
32222	Verdi	Music	48000	

Multitable Clustering File Organization

- Store several relations in one file using a **multitable clustering** file organization

Department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

Instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

Multitable clustering of department and instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

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

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 T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i 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.)

Transaction States in DBMS

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_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

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

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

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

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

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.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- 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 I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_j 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 T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
  2. If in schedule S transaction T_i executes **read**(*Q)*, and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  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.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- 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.

T_1	T_5
read (A)	
A := A – 50	
write (A)	
	read (B)
	B := B - 10
	write (B)
read (B)	
B := B + 50	
write (B)	
	read (A)
	A := A + 10
	write (A)

- 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, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i 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^2 time, 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 $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?

(a)

(b)

(c)

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 T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- 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)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

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 T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- 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

Request

Grant

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.

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.

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

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.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

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.

Lock table.

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.

To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether thesystem has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlocks can be described precisely in terms of a directed graph called a wait for graph.

Figure 15.13 Wait-for graph with no cycle.

Figure 15.14 Wait-for graph with a cycle.

MULTIPLE GRANULARITY

Granularity is the size of the data item allowed to lock.

Multiple Granularity means hierarchically breaking up the database into blocks that can be locked and can be tracked what needs to lock and in what fashion.

Figure 15.15 Granularity hierarchy.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 15.16 Compatibility matrix.

TIMESTAMP-BASED PROTOCOLS

Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp-ordering scheme.

TIMESTAMP

There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

To implement this scheme, we associate with each data item Q two timestamp values:

- W-timestamp(Q)- largest timestamp of any transaction that executed write(Q) successfully.
- R-timestamp(Q)- largest timestamp of any transaction that executed read(Q) successfully.

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction Ti issues read(Q).

- If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
- If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

2. Suppose that transaction Ti issues write(Q).

- If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
- If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.
- Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Thomas' Write Rule

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes thewrite operation and setsW-timestamp(Q) to TS(Ti)

The difference between these rules and those of timestamp ordering lies in the second rule. The timestamp-ordering protocol requires that Ti be rolled back if Ti issues write(Q) and TS(Ti) < W-timestamp(Q). However, here, in those cases where TS(Ti) ≥ R-timestamp(Q), we ignore the obsolete write.

TIMESTAMP ORDERING SCHEME

Transaction T_{26} transfers \$50 from account B to account A, and then displays the contents of both:

T_{26}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Thomas' Write Rule

Figure on the side is not conflict serializable and, thus, is not possible under the two-phase locking protocol, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T27 would be ignored. The result is a schedule that is view equivalent to the serial schedule <T27, T28>

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

ADVANTAGES:

1. The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order (like 2PL Protocols).
2. The protocol ensures freedom from deadlock, since no transaction ever waits.

DISADVANTAGES:

1. There is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction

VALIDATION-BASED PROTOCOLS

A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for monitoring the system. Every Ti has to go through the following phases:-

1. Read Phase: Ti reads all data and store them in temporary variables (Local variables of Ti). After reading all the write operations are made on temporary variables instead of actual database.
2. Validation Phase: In this, validation test is performed to determine whether changes in actual database can be made.
3. Write Phase: If Ti clears the validation test then actual changes are made to database.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

a schedule produced by using validation.

- Validation test ensures the violation free execution of transaction
- Time stamp is used to determine when to start validation test.

Every Ti is associated with 3 time stamps that are:

1. ***Start (Ti)***: It gives time when Ti starts execution.
2. ***Validation(Ti)***: It gives time when Ti finishes its read phase & starts its validation phase
3. ***Finish(Ti)***: It gives time when Ti finished its execution or write phase.

ADVANTAGES:

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed.

Useful as it gives greater degree of concurrency if probability if conflicts is low.

OVERCOMING PROBLEMS

There is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction.

- To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.
- This validation scheme is called the **optimistic concurrency-control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.
- In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

MULTIVERSION SCHEMES

In multiversion concurrency-control schemes, each write(Q) operation creates a new version of Q. When a transaction issues a read(Q) operation, the concurrency-control manager selects one of the versions of Q to be read.

Multiversion Timestamp Ordering

A transaction—say, Ti—creates a new version Qk of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by Ti . The system initializes the W-timestamp and R-timestamp to TS(Ti). It updates the R-timestamp value of Qk whenever a transaction Tj reads the content of Qk , and R-timestamp(Qk) < TS(Tj)

The multiversion timestamp-ordering scheme presented next ensures serializability. The scheme operates as follows: Suppose that transaction Ti issues a read(Q) or write(Q) operation. Let Qk denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(Ti).

ADVANTAGES:

It has the desirable property that a read request never fails and is never made to wait.

It helps prevent deadlocks by allowing transactions to read and write different versions of data items. This flexibility minimizes the chances of circular dependencies leading to deadlocks.

DISADVANTAGES:

The reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one.

The conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive.

Multiversion Two-Phase Locking

The multiversion two-phase locking protocol attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between read-only transactions and update transactions.

- Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order.
- Each version of a data item has a single timestamp.The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.
- The database system assigns read-only transactions a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
- Thus, when a read-only transaction Ti issues a read(Q), the value returned is the contents of the version whose timestamp is the largest timestamp less than or equal to TS(Ti)

ADVANTAGES:

1. Here, read-only transactions never need to wait for locks.
2. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

SNAPSHOT ISOLATION

Snapshot isolation is a particular type of concurrency-control scheme that has gained wide acceptance in commercial and open-source systems, including Oracle, PostgreSQL, and SQL Server.

Conceptually, snapshot isolation involves giving a transaction a “snapshot” of the database at the time when it begins its execution. It then operates on that snap?shot in complete isolation from concurrent transaction

There are two variants of snapshot isolation, both of which prevent lost updates.

Under **first committer** wins, when a transaction T enters the partially committed state, the following actions are taken in an atomic action:

- A test is made to see if any transaction that was concurrent with T has already written an update to the database for some data item that T intends to write.
- If some such transaction is found, then T aborts.
- If no such transaction is found, then T commits and its updates are written to the database.

Under **first updater** wins the system uses a locking mechanism that applies only to updates. The following steps are taken after the lock is acquired:

- If the item has been updated by any concurrent transaction, then Ti aborts.
- Otherwise Ti may proceed with its execution including possibly committing.

ISSUES IN CONCURRENT EXECUTION

- The coordination of the simultaneous execution of transactions in a multiuser database system is known as concurrency control.
- The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment.
- Concurrency control is important because the simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.
- The three main problems are lost updates, uncommitted data, and inconsistent retrievals.

ISSUES IN CONCURRENT EXECUTION

- The five concurrency problems that can occur in the database are:
- Temporary Update Problem
- Incorrect Summary Problem
- Lost Update Problem
- Unrepeatable Read Problem
- Phantom Read Problem

Temporary Update Problem:

- Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.
- **Example:**

Example:

T1	T2
read_item(X)	
X = X - N	
write_item(X)	
	read_item(X)
	X = X + M
	write_item(X)
read_item(Y)	

In the above example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

Incorrect Summary Problem:

- Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records.
- The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:

T1	T2
	sum = 0 read_item(A) sum = sum + A
read_item(X) X = X - N write_item(X)	
	read_item(X) sum = sum + X read_item(Y) sum = sum + Y
read_item(Y) Y = Y + N write_item(Y)	

In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

Lost Update Problem:

- In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

T1	T2
read_item(X) X = X + N	
	X = X + 10 write_item(X)

In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X *(not shown in the image above)*. Therefore, the update done by transaction 2 will be lost. Basically, the write commit done by the **last transaction** will overwrite all previous write commits.

Unrepeatable Read Problem:

- The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

Example:

T1	T2
Read(X)	
	Read(X)
Write(X)	
	Read(X)

In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

Phantom Read Problem:

- The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

Example:

T1	T2
Read(X)	
	Read(X)
Delete(X)	
	Read(X)

In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Solutions

- To prevent concurrency problems in DBMS transactions, several concurrency control techniques can be used, including locking, timestamp ordering, and optimistic concurrency control.
- Locking involves acquiring locks on the data items used by transactions, preventing other transactions from accessing the same data until the lock is released.
- There are different types of locks, such as shared and exclusive locks, and they can be used to prevent Dirty Read and Non-Repeatable Read.

- Timestamp ordering assigns a unique timestamp to each transaction and ensures that transactions execute in timestamp order.
- Timestamp ordering can prevent Non-Repeatable Read and Phantom Read.
- Optimistic concurrency control assumes that conflicts between transactions are rare and allows transactions to proceed without acquiring locks initially.
- If a conflict is detected, the transaction is rolled back, and the conflict is resolved.
- Optimistic concurrency control can prevent Dirty Read, Non-Repeatable Read, and Phantom Read.

Recovery System

Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Remote Backup Systems

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 (e.g., 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 by 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 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
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. 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
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

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):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Stable-Storage Implementation (Cont.)

- 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:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *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
  2. 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.
- **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.

Example of Data Access

Data Access (Cont.)

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

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** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alte
  (brief details in

shadow-copy

(a) Before update (b) After update

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  <T_i **start**>log record
- *Before* T_i executes **write**(X), a log record
  <T_i, X, V_1, V_2>
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commit**> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate Database Modification

- 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* database item is written
  - We assume that the log record is output directly to stable storage
  - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to stable 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

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

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

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

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 T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to 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.

Undo and Redo Operations

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

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>
- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

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

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

Recovery actions in each case above are:

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

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

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

Checkpoints

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

Checkpoints (Cont.)

- 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.
  1. Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  1. 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.

Example of Checkpoints

- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Recovery Algorithm

- **So far:** we covered key concepts
- **Now**: we present the components of the basic recovery algorithm
- **Later**: we present extensions to allow more concurrency

Recovery Algorithm

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - perform the undo by writing V_1 to X_j,
    - write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record <T_i, X_j, V_1, V_2> or <T_i, X_j, V_2> is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. <T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery

Beginning of log

older

<T_0 start>
<T_0, B, 2000, 2050>
<T_1 start>
<checkpoint {T_0, T_1}>
<T_1, C, 700, 600>
<T_1 commit>
<T_2 start>
<T_2, A, 500, 400>
<T_0, B, 2000>
<T_0 abort>
<T_2, A, 500>
<T_2 abort>

newer

End of log at crash!

Log records added during recovery

T_0 rollback (during normal operation) begins

T_0 rollback complete

T_2 is incomplete at crash

T_2 rolled back in undo pass

Start log records found for all transactions in undo list

Redo Pass

Undo list: T_2

Undo Pass

Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**:i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     1. Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use

    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
  - Similar to fuzzy checkpointing

Recovery with Early Lock Release and Logical Undo Operations

Recovery with Early Lock Release

- 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

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 (resp. deletions) are undone by executing a deletion (resp. 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

- Operation logging is done as follows:
  1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
  2. While operation is executing, normal log records with physical redo and physical undo information are logged.
  3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

Example: insert of (key, record-id) pair (K5, RID7) into index I9

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

Operation Logging (Cont.)

- 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.
- <u>Redo of operation (after crash) still uses physical redo information</u>.

Transaction Rollback with Logical Undo

Rollback of transaction T_i is done as follows:

- Scan the log backwards
  1. If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a al <T_i, X, V_1>.
  2. If a <T_i, O_j, **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

         <T_i, O_j, **operation-abort**>.
     - Skip all preceding log records for T_i until the record <T_i, O_j **operation-begin**> is found

Transaction Rollback with Logical Undo (Cont.)

- Transaction rollback, scanning the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a <T_i, O_j, **operation-abort**> record is found:
     - skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
  5. Stop the scan when the record <T_i, start> is found
  6. Add a <T_i, **abort**> record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Transaction Rollback with Logical Undo

- **Transaction rollback during normal operation**

Beginning of log

<T_0 start>
<T_0, B, 2000, 2050>
<T_0, O_1, operation-begin>
<T_0, C, 700, 600>
<T_0, O_1, operation-end, (C, +100)>
<T_1 start>
<T_1, O_2, operation-begin>
<T_1, C, 600, 400>
<T_1, O_2, operation-end, (C, +200)>

<T_0 , C, 400, 500>
<T_0, O_1, operation-abort>
<T_0, B, 2000>
<T_0, abort>
<T_1, commit>

If T_0 aborts before operation O_1 ends, undo of update to C will be physical

T_0 has completed operation O_1 on C, releases lower-level lock; physical undo cannot be done anymore, logical undo will add 100 to C

T_1 can update C since T_0 has released lower-level lock on C

T_1 releases lower-level lock on C

T0 decides to abort

Logical undo of O_1 adds 100 to C

O_1 undo complete

Beginning of log
<T0 start>
<T0, B, 2000, 2050>
<T0 commit>
<T1 start>
<T1, B, 2050, 2100>
<T1, O4, operation-begin>
<checkpoint {T1}>
<T1, C, 700, 400>
<T1, O4, operation-end, (C, +300)>
<T2 start>
<T2, O5, operation-begin>
<T2, C, 400, 300>
End of log at crash!
<T2, C, 400>
<T2 abort>
<T1, C, 400, 700>
<T1, O4, operation-abort>
<T1, B, 2050>
<T1 abort>
Records added during recovery
Start log records found for all transactions in undo list
Redo Pass
Undo list: T1, T2
Undo Pass
Update of C was part of O5, undone physically during recovery since O5 did not complete
Logical undo of O4 adds 300 to C

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 I9, 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

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

Recovery with Logical Undo (Cont.)

Recovery from system crash (cont.)

2. (**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 <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.
   - Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*

- This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

ARIES Recovery Algorithm

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The recovery algorithm we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- Unlike the recovery algorithm described earlier, ARIES
  1. Uses **log sequence number (LSN)** to identify log records
     - Stores LSNs in pages to identify what updates have already been applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time
     - More coming up on each of the above …

ARIES Optimizations

- **Physiological redo**
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be moved to fill hole
        - Physiological redo can log just the record deletion
        - Physical redo would require logging of old and new values for much of the page
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk systems
      - Incomplete page output can be detected by checksum techniques,
        - But extra actions are required for recovery
        - Treated as a media failure

ARIES Data Structures

- ARIES uses several data structures
  - Log sequence number (LSN) identifies each log record
    - Must be sequentially increasing
    - Typically an offset from beginning of log file to allow fast access
      - Easily extended to handle multiple log files
  - Page LSN
  - Log records of several different types
  - Dirty page table

ARIES Data Structures: Page LSN

- Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
  - To update a page:
    - X-latch the page, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page
  - To flush page to disk, must first S-latch page
    - Thus page state on disk is operation consistent
      - Required to support physiological redo
  - PageLSN is used during recovery to prevent repeated redo
    - Thus ensuring idempotence

ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo

  - LSN in log record may be implicit
- Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in earlier recovery algorithm
  - Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone
    - Required to avoid repeated undo of already undone actions

LSN	TransID	UndoNextLSN	RedoInfo

ARIES Data Structures: DirtyPage Table

- **DirtyPageTable**
  - List of pages in the buffer that have been updated
  - Contains, for each such page
    - **PageLSN** of the page
    - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
      - Set to current end of log when a page is inserted into dirty page table (just before being updated)
      - Recorded in checkpoints, helps to minimize redo work

ARIES Data Structures

Database Buffer

PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table

Log Buffer (PrevLSN and UndoNextLSN fields not shown)

Stable data

Stable log

ARIES Data Structures: Checkpoint Log

- **Checkpoint log record**
  - Contains:
    - DirtyPageTable and list of active transactions
    - For each active transaction, LastLSN, the LSN of the last log record written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time
    - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently

ARIES Recovery Algorithm

ARIES recovery involves three passes

- Analysis pass: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- Redo pass:
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- Undo pass:
  - Rolls back all incomplete transactions
    - Transactions whose abort was complete earlier are not undone
      - Key idea: no need to undo these transactions: earlier undo actions were logged, and are redone as required

Aries Recovery: 3 Passes

- Analysis, redo and undo passes
- Analysis determines where redo should start
- Undo has to go back till start of earliest incomplete transaction

ARIES Recovery: Analysis

Analysis pass

- Starts from last complete checkpoint log record
  - Reads DirtyPageTable from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
    - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record
- Scans forward from checkpoint
- .. Cont. on next page …

ARIES Recovery: Analysis (Cont.)

Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in undo-list, adds transaction to undo-list
  - Whenever an update log record is found
    - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
  - If transaction end log record found, delete transaction from undo-list
  - Keeps track of last log record for each transaction in undo-list
    - May be needed for later undo
- At end of analysis pass:
  - RedoLSN determines where to start redo pass
  - RecLSN for each page in DirtyPageTable used to minimize redo work
  - All transactions in undo-list need to be rolled back

ARIES Redo Pass

Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:

- Scans forward from RedoLSN. Whenever an update log record is found:
  1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record
  2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

  NOTE: if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk!

ARIES Undo Actions

- When an undo is performed for an update log record
  - Generate a CLR containing the undo action performed (actions performed during undo are logged physicaly or physiologically).
    - CLR for record *n* noted as *n*' in figure below
  - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
    - Arrows indicate UndoNextLSN value
- ARIES supports partial rollback
  - Used e.g. to handle deadlocks by rolling back just enough to release reqd. locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback

ARIES: Undo Pass

Undo pass:

- Performs backward scan on log undoing all transaction in undo-list
  - Backward scan optimized by skipping unneeded log records as follows:
    - Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
    - At each step pick largest of these LSNs to undo, skip back to it and undo it
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
      - For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record
        - All intervening records are skipped since they would have been undone already
- Undos performed as described earlier

Recovery Actions in ARIES

Other ARIES Features

- Recovery Independence
  - Pages can be recovered independently of others
    - E.g. if some disk pages fail they can be recovered from a backup while other pages are being used
- Savepoints:
  - Transactions can record savepoints and roll back to a savepoint
    - Useful for complex transactions
    - Also used to rollback just enough to release locks on deadlock

Other ARIES Features (Cont.)

- Fine-grained locking:
  - Index concurrency algorithms that permit tuple level locking on indices can be used
    - These require logical undo, rather than physical undo, as in earlier recovery algorithm
- Recovery optimizations: For example:
  - Dirty page table can be used to prefetch pages during redo
  - Out of order redo is possible:
    - redo can be postponed on a page being fetched from disk, and performed when page is fetched.
    - Meanwhile other log records can continue to be processed

Remote Backup Systems

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.

Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19

Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe:** commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

Reference

Abraham Silberschatz, Henry F. Korth, S. Sudharshan, Database System Conceptsll, Seventh Edition, Tata McGraw Hill, 2019

Case study on Ecommerce Database Management System

- Project Description
- Basic structure
    - Functional requirements
    - Entity Relation (ER) diagram and constraints
    - Relational database schema
- Implementation
    - Creating tables
    - Inserting data
- Queries
    - Basic queries
    - PL/SQL function
    - Trigger function
    - Stored procedures
    - Functions
    - Transactions

1. Project Description

In this new modern era of online shopping no seller wants to be left behind and every seller want to the shift from offline selling model to an online selling model for a rampant growth. Therefore, as an software engineer our job is to ease the path of this transition for the seller. Amongst many things that an online site requires the most important is a database system. Hence in this project we are planning to design a database where small sellers can sell their product online.

The Prime Objective of our database project is to design a robust E-commerce database by performing operations such as,

- Viewing orders
- Placing orders
- Updating database
- Reviewing products
- Maintaining data consistency across tables

2. Requirements

- A Customer can see the account details and can update if required.
- Customer can search the products according to the category.
- Customer can add his wish list to the cart and can see the total amount.
- Customer can update the cart whenever required.
- Customer can choose the mode of payment.
- Customer can keep track of the order by seeing order status.
- Customer can review the products which have been purchased.
- Seller can update the stock of a particular product whether it is available or not.
- Seller can keep track of total sales of his products.
- Seller can know the sales on a particular day or month or year.

2.1 Functional Requirements

- A Customer cannot access the Seller details and vice-versa.
- There should not be any inconsistency in the data.
- There should not be any loss of data.

3. Relational Database Schema - e commerce

4. Entities and Attributes

ENTITIES	ATTRIBUTES	ATTRIBUTE TYPE	Entity Type
Customer	Customer_CustomerId Name Email DateOfBirth Phone Age	Simple Composite Simple Simple Multivalued Derived	Strong
Order	OrderId ShippingDate OrderDate OrderAmount Cart_CartID	Simple Simple Simple Simple Simple	Strong
OrderItem	Order_OrderId (PK) Product_ProductId(FK) MRP Quantity	Simple Simple Simple Simple	Weak
Product	productId (PK) ProductName sellerId MRP CategoryID Stock Brand	Simple Simple Simple Simple Simple Simple Simple	Strong
Review	ReviewId(PK) Description Ratings Product_ProductId Customer_CustomerID	Simple Simple Simple Simple	Strong
Cart	cartId (PK) Customer_customerId(FK) GrandTotal ItemsTotal	Simple Simple Derived Derived	Strong
Category	CategoryID(PK) CategoryName DESCRIPTION	Simple Simple Simple	Strong
seller	sellerId (PK) Name Phone Total_Sales	Simple Simple Multivalued Derived	Strong
Payment	payment_id Order_OrderId PaymentMode	Simple Simple Simple	Strong

ENTITIES	ATTRIBUTES	ATTRIBUTE TYPE	Entity Type
	Customer_CustomerId PaymentDate	Simple Simple	

5. Entities and Relations

ENTITIES	RELATION	CARDINALITY	TYPE OF PARTICIPATION
Customer Address	Stays At	OneToOne	Total Partial
Customer Cart	Shops	OneToOne	Partial Total
Customer Order	Places	OneToMany	Partial Total
Customer Payment	Makes	OneToMany	Partial Total
Customer Review	Write	OneToMany	Partial Total
Seller Product	Sells	ManyToMany	Partial Total
Category Product	Categorizes	OneToMany	Partial Total
Cart Product	Contains	ManyToMany	Partial Partial
Product Orderltem	Includes	OneToMany	Partial Total
Order Orderltem	Includes	OneToOne	Partial Total
Payment Order	For	OneToOne	Total Total

6. ER Diagram

QUERIES ON THE ABOVE RELATIONAL SCHEMA

1. Stored procedure for the details of the customer.

2. View for getting sales by category of products.

3. Using triggers to update the no.of products as soon as the payment is made.

4. Trigger to update the total amount of user everytime he adds something to payment table.

5. Stored procedure for getting order history.

6. Processing an order

To process an order, one should check whether those items are in stock.

If items are in stock, they need to be reserved so that they go in hands of those who have expressed them in wishlist/order.

Once ordered the available quantity must be reduced to reflect the correct value in the stock.

Any items not in stock cannot be sanctioned; this requires confirmation from the seller.

The customer needs to be informed as to which items are in stock (and can be shipped immediately) and which are cancelled.

7. Check whether the specified customer exists

IF NOT EXISTS add him/her

COMMIT the info

Fetch the customer id

INSERT a row to Order tables

If unable to do so,ROLLBACK;

Fetch the new orderid in orders table

INSERT row to the order table for every product ordered

If adding tuples to order items fails ROLL BACK all tuples of products added for and the tuple in order row

QUERY 1: Customers to find products with highest ratings for a given category.
QUERY 2: Customers to filter out the products according to their brand and price.
QUERY 3: If a customer want to know the total price for all product present in the cart.
QUERY 4: Customers to find the best seller of a particular product.
QUERY 5: List the orders which are to be delivered at a particular pincode.
QUERY 6: List the product whose sale is the highest on a particular day.
QUERY 7: List the category of product which has been sold the highest on a particular day.
QUERY 8: List the customers who bought products from a particular seller the most.
QUERY 9: List all the orders whose payment mode is not CoD and yet to be delivered.
QUERY 10: List all orders of customers whose total amount is greater than 5000.
QUERY 11: If customer wants to modify the cart that, is he want to delete some products from the cart.
QUERY 12: List the seller who has the highest stock of a particular product.
QUERY 13: Customers to compare the products based on their ratings and reviews.

CASE STUDY ON NoSQL Databases- Document Oriented, Key value pairs, Column Oriented and Graph

NoSQL

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema.

NoSQL databases are non-tabular and handle data storage differently than relational tables.

These databases are classified according to the data model, and popular types include document, graph, column, and key-value.

Non-relational in nature

The core function of NoSQL is to provide a mechanism for storing and retrieving information.

NoSQL

NoSQL database stands for “Not Only SQL” or “Not SQL.”

NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook

SQL Database
Relational
Analytical (OLAP)
NoSQL Database
Column-Family
Graph
Document
Key-Value
key
value
key
value
key
value
key
value

Why NoSQL

Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could “scale up” our systems by upgrading our existing hardware.

But this process is expensive.

Why NoSQL

Alternative for this issue is to distribute database - load on multiple hosts whenever the load increases.

This method is known as "scaling out."

Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates
- Doesn’t require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Features of NoSQL

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain

Features of NoSQL

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Features of NoSQL

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented

Key Value Pair Based

Data is stored in key/value pairs.

It is helpful to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

Column-based

Column-oriented databases work on columns

Every column is treated separately.

Values of single column databases are stored contiguously.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence

HBase, Cassandra, HBase, Hypertable

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Document-Oriented

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document.

Examples of Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB

Document-Oriented

Relational Data Model

Document Store Model

Graph-Based

A graph type database stores entities as well the relations amongst those entities.

The entity is stored as a node with the relationship as edges.

An edge gives a relationship between nodes.

Every node and edge has a unique identifier.

Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature.

Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB

Sales

Customer	Item	Time
0001	1A	20:34
0001	1A	21:15
0003	2A	21:16
0002	1A	21:16
0002	5C	

Inventory

Description	SKU
Pepsi	1A
Club Soda	2A
.	-
.	.
Diet Coke	5C

Customer

Name	CustID
John	0001
Jack	0002
Ted	0003
Ken	0004
Valerie	0005

Traditional database store data to efficiently store facts, but relationships must be rebuilt with JOINs and other inexact techniques.

Graph databases store both facts and the relationships between the facts, making certain types of analysis more intuitive.

Advantages of NoSQL

- Big Data Capability
- No Single Point of Failure
- Easy Replication

Can handle structured, semi-structured, and unstructured data

Offers a flexible schema design

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn’t work as well with relational data

It does not offer any traditional database capabilities, like consistency