Reference Notes – Advanced Database Systems

COMP90050_2022_WIN

Ritwik Giri

University of Melbourne

Table of Contents

Database Systems (DBMS) – Introduction

Effective and efficient manipulation of the stored data is of utmost requirement for most of the companies. Data is growing with the rapid pace and its complexity is also increasing. Now we have multiple data types including images, videos, spatial etc.

The database system provides the ability to retrieve and process the data effectively and efficiently and it also considers the factors of reliability and security according to the needs of the application.

The **performance** of a database system (DBMS) depends on 3 major factors, Hardware, Software, and Crash Recovery Mechanism.

Hardware: Speed of the processors, number of the processors, main memory capacity, or the type of architecture plays a crucial role in retrieval, storage, and manipulation of data. For eg: In Hard Disk Drive, the disk access time depends on the Actuator arm seek and relational time, while in Solid State Drive, these factors are not considered.
Software: The type of technology and algorithms used also impacts the efficiency of the DBMS. The algorithm for query analysis by the optimizer and how the data is manipulated using queries plays a crucial role in the DBMS.
Crash Recovery: It is an important feature that helps the user to retrieve data in case of any crash. It enables the user to retrieve any lost data, if executed properly. Databases uses the concept of ARIES to tackle the crashes.

Hardware of a classical disk (Hard Disk Drive)

All the components are on the rapidly rotating platter.

Track: Circular path on disk surface

Track Sector:

Disk Sector: Tracks are subdivided into disk sectors and each sector can hold equal amount of data (Group of track sectors)

Cluster: Group of sectors within a disk, it can be read sequentially

Actuator Arm: It has magnetic head which reads and write data to platter surfaces.

Head: It is responsible to read and write data into and from the disk

Disk Access Time and SSD

Classical Disk(HDD)

Disk access time = seek time + rotation time + (transfer length/Bandwidth)

Seek time/ seek latency: is the period that the head of the actuator arm moves from a position to a required track

Rotation Time/ Rotation Latency: is caused when the head of the actuator arm needs to get to the required sector of a track.

Eg:

In a hard disk drive (HDD), the average seek time is 12 ms, rotation delay is 4 ms, and transfer rate is 4MB/sec.

Answer:

Seek Time: 12ms
Rotation Time: 4ms
Transfer Length: 8Mb
Bandwidth: 4MB/Sec -> 4Mb/(1000)ms (1sec = 1000ms) -> 4/1000 MB/ms

Disk Access Time = 12 + 4 + (8 * 1000 / (4)) = 2016ms

Seek Time: 12ms
Rotation Time: 4ms
Transfer Length: 8Kb
Bandwidth: 4MB/Sec -> 4Mb/(1000)ms (1sec = 1000ms) -> 4/1000 MB/ms -> 4Kb/ms

Disk Access Time = 12 + 4 + (8 / (4)) = 18ms

Or
12 + 4 + (0.008*1000/4) = 18ms

Comments:

It is efficient to fetch a large amount of data at one go as the rotation and seek time is nullified under the transfer length/bandwidth factor of the formulae. But when the transfer size is less, the seek and rotation time becomes dominant, and most of the time is spent on going towards the data location. Hence, it becomes inefficient.

Transmit Time: distance/c + message_bits/bandwidth

Message length should be large to achieve better utilization.

For Solid State Drive seek time and rotation time doesn't come into the picture. It does not have any rotating part. Hence the formulae become,

Disk access time = (transfer length/Bandwidth) # hence SSD is always faster than the HDD

SSD are generally more expensive and uses silicon instead of magnetic materials. It does not have start-up time like in HDD and runs silently.

Moore’s and Joy’s Law

Moore’s law: memory chip capacity doubles every 18 months since 1970

$$= 2^{\frac{(year-1970)*2}{3}} \ Kb/chip$$

Joy’s law for processors: processor performance doubles every two years since 1984

$$= 2^{(year-1984)/2} \ mips$$

DBMS Design Choices

Initially, the Hard disks were the key factor for the design choices, but eventually with the ever-increasing pace of technology the factors of CPU and networking also plays a crucial role in the DBMS Design Choices. Number of distributed databases, increasing number of users, requirement of reducing latency, and increasing availability all plays a crucial role in the design choices.

Memory Hierarchy

The main components of the memory (from the bottom) are as follows, Hard disks, Main Memory, L2 Cache, L1 Cache, and Registers. The speed of data transfer increase from the top to bottom, e.g., Main Memory will be faster than the Hard Disk. However, the storage capacity decreases from the top to bottom, e.g., Hard Disk can store data in the terms of Terabytes (TBs) while the Main Memory or RAM can only store data in Gigabytes (GBs).

Registers and L1 caches are generally placed on chip, hence can access the processor faster, while the L2 caches are placed off-chip. There is generally 10-100 times drop in access speed from on Chip to off chip.
Hard disks also has a buffer storage. It is a caching provided with HDD for access.

Hit Ratio, Effective Memory Access Time, Disk Buffer Access Time:

Hit Ratio : It is the ratio that measures the references satisfied by a cache successfully from its cache storage, compared to the total references it received.

Hit Ratio(H) = References satisfied by cache/total references

Effective memory access time:

$EA = H*C+(1-H)*M$

where H = hit ratio,
C = cache access time.
M = memory access time

Effective disk buffer access time,

$EA = HB*BC+(1-HB)*D$

where
HB = hit ratio of the disk buffer,
BC = buffer access time.
D = disk access time

Eg:

A has a smaller cache with on average 50% cache hit ratio (H) and the other machine (machine B) has a much larger cache with on average 90% cache hit ratio. However, the memory access time of machine A is 100C and the memory access time of machine B is 400C i.e., memory access in machine A is faster than memory access in machine B), where **C is the cache access time**. Which machine has overall **faster effective memory access time**?

EA (A) = 0.5*C + (1-0.5)100C = 50.5C

EA(B) = 0.9*C + (1-0.9)400C = 40.9C

Even though the memory access time for the Machine B is faster, but the larger cache hit ratio makes the makes faster than the Machine A.

Types of Database Systems

Simple Files:

These are the normal text-based files, which are fast for simple applications. Data is stored as a row of text file separated by delimiters. To perform complex operations, extra features are incorporated, which becomes an overhead.
These files are less reliable and are hard to maintain which can lead to concurrency problems.

Relational DBMS:
In RDBMS the data is stored in a tabular format and with rows representing the individual records and columns representing attributes. It is an application independent optimization and is very reliable and usually very consistent. It is well suited for many applications especially because of the development of fast memory machines and SSDs. It is widely used. It also support Object Oriented model and XML data queries.
It can be slow for simple applications.

Object Oriented Systems:
OODBMS the data is stored as a collection of objects, that encapsulate data as attributes and can have methods associated with it, and not tables. They follow object-oriented programming paradigm.
It is well suited for applications that require complex data. It did not survive because of the boom in the RDBMS technology and hence disappeared from the market.

NoSQL
As the data structure becomes complex and unstructured, a NoSQL-based approach is used widely. It has a dynamic schema and is document-based, unlike table-based RDBMS. It is a practical approach to managing many-to-many relationships and is hence an ideal way to represent complex non-linear networks.
It compromises consistency and allows replications and hence increase the availability. It further provides eventual consistency, which sometimes lead to stale reads.

Types of NoSql

a. Key-value storage: A key-value database stores data as a collection of key-value pairs where a key serves as a unique identifier. All access to the database are done via the key. Both keys and values can be complex.

   Application: if the dataset do not need complex relational table type of structure but can be expressed with simple key-value pairs. The simple structure allows faster insertion and search, and scales quickly. For example, shopping cart database in an e-commerce site

b. Document storage: Flexible for storing different kinds of documents, where they may not all have the same sections. XML, JSON, etc. are subclasses of document-oriented databases.

Application: Well suited when different kinds of documents do not always have the same structure/sections. For example – a database of news articles.

c. Graph storage: Graphs capture connectivity between entities. Searching and traversing by relations are very fast in such structures.

Application: Well suited for storing connection data such as social network connections and spatial data.

d. Column Based: Unlike relational databases, columnar databases store their data by columns, rather than by rows. These columns are gathered to form subgroups. The keys and the column names of this type of database are not fixed.

   Application: Applications for column-based NoSQL databases: They are used in data analysis applications/tasks.

Database Architecture

Centralized:
In centralized system, data is stored in one location and hence the system administration is simple. Optimization process is generally effective. PC/Cluster/Data Centres are examples of this.
It is similar to the Client-server architecture.

Distributed:
In this the data is distributed across several nodes in different locations. The nodes are connected by network. The system provides concurrency, recovery, and transaction processing. But the system administration is hard, and the crash recovery is complicated. Distributed data might lead to data inconsistency.

World Wide Web:
The data is stored in multiple locations and there are several owners of the data. It does not provide certainty data availability or consistency. Optimisation process is generally very ineffective. It is an evolving field and no standards has been fixed as such. This potentially leads to security problems.

Grid Computing and Databases
It is similar to the Distributed databases. Data and processing are shared among a group of computer systems which may be geographically separated. Administration of such systems are done locally by each owner of the system. Reliability and security of such systems are not well developed or studied. It is usually application dependent. The introduction of cloud models outdated the use of grids.

P2P
Data and processing are shared among a group of computer systems which may be geographically separated as in Grid Database systems. But in P2P, Computer nodes can join and leave the network at will. Hence it is harder to design transaction models. Administration of such system is done by the owners of the data. Usually designed for particular usage – e.g. a scientific application

Cloud Computing
Cloud computing offers online computing, storage and a range of new services for data and devices that are accessible through the Internet. User pays for the services just like phone services, electricity, etc. Huge potential for developing applications with minimal infrastructure costs. It offers SaaS, PaaS etc.
A. Centralised – suitable for simple applications, easy to manage; may not scale well
B. Distributed – Scalable, suitable for large applications and applications that need data access from different physical locations; System administration and crash recovery is difficult, usually have some data inconsistency
C. WWW – Very convenient to access and share data; security issues, no guarantee on availability or consistency
D. Grid – Less used now-a-days, very similar to distributed systems with administration done locally by each owner
E. P2P- Suitable when the nodes of the network cannot be planned in advance, or some may leave and join frequently. For example, sensor network
F. Cloud database – on-demand resources, cost-effective, maintenance done externally by the cloud provider; some privacy and confidentially issue – but most trusted providers welladdress
them

Type of Storage Systems

Storage systems can determine the performance and also fault tolerance of a Database. Data is rarely stored in one location. It involves multiple locations and disks accessed over a network.

Storage Area Networks
It is a dedicated network of storage devices. It consists of SAN controller and Switch, that forms an intermediate between the systems and the storage devices. The storage devices are organized as RAID (Redundant array of Independent/Inexpensive disks).

It is used for shared disk file systems and provides automated back up functionality. It has its own networking capabilities. The probability of one disk failure is different for different disks in the network.

Fault Tolerance

The property that enables a system to continue operating properly in the event of the failure of some of its components.

Mean time to event A is, MT(A) = 1/P(A)

Mean time to the first event, m * (1/n) = m/n

Module availability: measures the ratio of service accomplishment to elapsed time i.e. the ratio between mean time to failure to the total elapsed time of failure and repair.

MTTF: the time elapsing before a failure is experienced

Fault Tolerance by RAID

Redundant Array of Independent Disks – different ways to combine multiple disks as a unit for fault tolerance or performance improvement, or both of a database system

From the slides – lecture 1 part 3

Fault Tolerance by Voting

FailVote: Stops if there are no majority agreement. Fails when there is no majority.
FailFast: Similar to failvote except the system senses which modules are available and uses the majority of the available modules. Fails when there is only one disk as there is no majority.

Failfast system has better availability than failvoting (since failvote stops when there is no majority agreement and do not account the available systems)

Supermodule: Naturally, a system with multiple hard disk drives is expected to function with only one working disk. Voting is used when multiple disks are available, but the system still works with one disk**.**

From the slides – lecture 1 part 3

Availability of FailVote:

If there are n events, mean time to the first event = m/n

Device MTTF = 10yrs

With 2 devices = 5 years (system fails with 1 device failure) -> 10/2
With 3 devices = (System fails when 2 devices fail)-> 10/3 + 10/2 = 8.33 yrs
Fault Tolerance with Repair

The faulty equipment is repaired with an average time of MTTR (mean time to repair) as soon as a fault is detected. The MTTF (years) >> MTTR (few hours)
Probability of module not available is:
MTTR/MTTF+MTTR -> MTTR/MTTF

Fault tolerance of a supermodule with repair

From the slides – lecture 1 part 3

$$\left(\frac{n}{MTTF}\right)\left(\frac{MTTR}{MTTF}\right)^{n-1}$$

(n-1) is majority – 1 for fail vote and fail fast

Communication Reliability

From the slides – lecture 1 part 3

Stable storage is used in the nodes in case of any communication failure. It maintains the recorded information of the message passing, and is updated when the output, input or ack is done by or to the nodes. It is checkpointing before sending the message.

Disk Write Consistency

Either entire block is written **correctly** on disk, or the contents of the block is unchanged. It is achieved by

Duplex Write

In duplex write, each block of data is written in two places sequentially. So that, if one of the writes fails the system can issue another write. Each block is associated with a version number. The block with the latest version number contains the most recent data.

While reading the error is determined using CRC. This type of write ensures at least one block has consistent data.

Logged Write

It is similar to duplex write, except one of the writes goes to a log. This method is very efficient if the changes to a block are small.

All the disk writes happens in the main memory. The data block is transferred to the main memory, the operations are performed and then sent to the storage disks.

Cyclic Redundancy Check (CRC) generation

CRC polynomial $\boldsymbol{x^{32} + x^{23} + x^{7} + 1}$

Most of the errors happens on the disks disk happen contiguously, that is in burst in nature.

- The above CRC generator can detect all burst errors with a length less than or equal to 32 bits

Computation in slides Lecture 1 part 4

RAID Questions

Part 1:

1. Which of the following RAID configurations that we saw in class has the lowest disk space utilization? Your answer needs to have explanations with calculations for each case.
   (a) RAID
   (b) 0 with 2 disks

   (b) RAID 1 with 2 disks

   (c) RAID 3 with 3 disks

Where does this lack of utilization of space go, i.e., where we can use such a configuration as it has some benefits gained due to the loss of space utilization?

Solution:

a. In case 1, the space utilization is 100% because the two disks store contiguous blocks of a file in RAID 0.
b. In case 2, the space utilization is 50% because RAID 1 uses mirroring. MTTF increases so for cases where a disk can fail easily this is good. The system operates even when a disk fails.
c. In case 3, the space utilization is (3-1)/3=66.7 because RAID 3 uses one disk for storing parity data.
   Case 2 has the lowest disk space utilization with explanation as given underlined above as a part of the answer.

2. What is the Mean time to failure values of different RAID systems?
   a. RAID 0 with 2 disks
   b. RAID 2 with 2 disks

   c. RAID 1 with 2 disks

   d. RAID 1 with 3 disks

   e. RAID 3 with 3 disks
   f. RAID 4 with 3 disks
   g. RAID 5 with 3 disks

   h. RAID 6 with 5 disks

Let's say the probability of one disk failure is p, and the MTTF of one individual disk is MTTF.

a+b. RAID 0 with 2 disks and RAID 2 with 2 disks – The system fails is one of the disks fails. The probability that one of the two disks fails (disk A or disk B) is p+p = 2p. So, Mean time to failure of the system is $\frac{1}{2p} = \frac{1}{2} * \frac{1}{p} = \frac{1}{2} * MTTF$

c. RAID 1 with 2 disks – The system fails is both of the disks fail at the same time. The probability that both disks fail (disk A and disk B) is p*p = p^2. So, Mean time to failure of the system is $\frac{1}{p^2} = \left(\frac{1}{P}\right)^2 = MTTF^2$

d. RAID 1 with 3 disks – The system fails is three of the disks fail at the same time. The probability that all disks fail (disk A and disk B and disk C) is p*p*p = p^3. So, Mean time to failure of the system is $\frac{1}{p^3} = \left(\frac{1}{P}\right)^3 = MTTF^3$

e+f+g. RAID 3 with 3 disks and RAID 4 with 3 disks and RAID 5 with 3 disks- The system fails if 2 of the 3 disks fail at the same time. The probability that 2 disks fail is p*p = p^2. There are 3 different possible combinations of 2 disks failures (A,B; or A,C; or B,C disks), so the probability that any of the 2 disks out of these 3 disks fail is $3p^2$. Mean time to failure of the system is $\frac{1}{3p^2} = \left(\frac{1}{3}\right) * \left(\frac{1}{P}\right)^2 = \left(\frac{1}{3}\right) * MTTF^2$

h. RAID 6 with 5 disks - The system fails if 3 of the 5 disks fail at the same time. The probability that 3 disks fail is p*p*p = p^3. There are 10 different possible combinations of 3 disks failures out of 5 disks(A,B,C; or A,B,D; or A,B,E; A,C,D; or A,C,E; or A,D,E; or B,C,D; or B,C,E; or B,D,E; or C,D,E disks), so the probability that any of the 3 disks out of these 5 disks fail is $10p^3$. Mean time to failure of the system is $\frac{1}{10p^3} = \left(\frac{1}{10}\right) * \left(\frac{1}{P}\right)^3 = \left(\frac{1}{10}\right) * MTTF^3$

Communication Questions

1. There are two nodes in a network that use stable storage and acknowledgment message passing for reliable communication. The stable storage of Node A contains the following record - Received message (In6); Transmitted message(Out3); **Out:3 Ack:3 In:6**. The stable storage of Node B contains the following record - Received message (In3); Transmitted message(Out6); **Out:6 Ack:6 In:3**.

   Now Node B sends a new message 7 to Node A. What will be in the stable storage of A and B if the message is received correctly, including a correctly received acknowledgement?

   Node A: Received message (In7); Transmitted message(Out3); Out:3 Ack:3 In:7.
   Node B: Received message (In3); Transmitted message(Out7); Out:7 Ack:7 In:3.

2. There are two nodes in a network that use stable storage and acknowledgment message passing for reliable communication. The stable storage of Node A contains the following record - Received message (In6); Transmitted message(Out3); **Out:3 Ack:3 In:6**. The stable storage of Node B contains the following record - Received message (In3); Transmitted message(Out6); **Out:6 Ack:6 In:3**.

Now Node B sends a new message 7 to Node A. **What will change in the stable storage of A and B if the message is received correctly, but the acknowledgement is not received?**

Node A: Received message (In**7**); Transmitted message(Out3); Out:3 Ack:3 In:**7**.
Node B: Received message (In3); Transmitted message(Out**7**); Out:**7** Ack:**6** In:3.

Database Engine

A database engine is the underlying software component that a database management system uses to create, read, update and delete data from a database. It consists of a query processor, storage manager and disk storage.
The query processor consists of the query evaluation engine, responsible to parse both DDL and DML queries. Storage Manager consists of buffer manager, file manager, transaction manager. Disk storage contains the indices, which improves the data access time.

Query Processing

The query is passed into the parser and translator and converted into a relational algebra expression. The algebra expression is then passed to the optimizer, which creates the execution plans. The plan is selected either by enumerating approach or by heuristic approach. The choice depends on the statistics, table sizes, available indices etc.

The plan is then executed, and the relevant actions are performed.

Transaction Processing

A transaction is a collection of operations that need to be performed as a complete unit.

commit_Trans() -> make the changes durable
rollback() -> undo all the changes

Transaction Model (ACID)

Atomicity - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

Example – A transaction that (i) subtracts $100 if balance >100 (ii) deposits $100 to another account
(both actions with either happen together or none will happen)

Consistency - Data is in a 'consistent' state when a transaction starts and when it ends – in other words, any data written to the database must be valid according to all defined rules (e.g., no duplicate student ID, no negative fund transfer, etc.)

What is 'consistent' - depends on the application and context constraints

Isolation- transaction are executed as if it is the only one in the system. Isolation ensures that concurrent transactions leaves the database in the same state as if the transactions were executed separately.

For example, in an application that transfers funds from one account to another, the isolation ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability- the system should tolerate system failures and any committed updates should not be lost.

Features of DB System

- Fast access to large amounts of data
- Provide a secure and stable repository when things fail
- Provides standard interfaces to data definition and manipulation
- Help multiuser accesses are done in an orderly manner
- Allow convenient ways for report production and browsing
- Ease in loading data, archiving, performance tuning
- It should support
  –insert/delete/modify record
  –read a particular record (specified using record id)
  –scan all records (possibly with some conditions on the records to be retrieved), or scan a range of records

Joins

Optimizer makes it decision based on the implemented join algorithm used.

A Simple Nested-Loop Join

```
for each tuple $t_r$ in $r$ do begin
    for each tuple $t_s$ in $s$ do begin
        test pair ($t_r$,$t_s$) to see if they satisfy the join condition theta (θ)
        if they do, add $t_r \bullet t_s$ to the result.
    end
end
```

r is called the **outer relation** and *s* the **inner relation** of the join.

Requires no indices and can be used with any kind of join condition.

Expensive since it examines every pair of tuples in the two relations

Could be cheap if you do it on two small tables where **they fit to main memory**

Building an index would be an overkill

Eg:

- Number of records of *customer*: 10,000 *depositor*: 5000
- Number of blocks of *customer*: 400 *depositor*: 100

$n_r * b_s + b_r$ block transfers, and
$n_r + b_r$ seeks

With *depositor* as the outer relation:

- $5000 * 400 + 100 = 2{,}000{,}100$ block transfers,
- $5000 + 100 = 5100$ seeks

With *customer* as the outer relation:

- $10000 * 100 + 400 = 1{,}000{,}400$ block transfers and 10,400 seeks
-

If seek time = 12ms
10,400 x 12ms seek time which makes about 2 mins to join two tables (just seek time)

Block Nested Loop Join

Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block $B_r$ of $r$ do begin
    for each block $B_s$ of $s$ do begin
        for each tuple $t_r$ in $B_r$ do begin
            for each tuple $t_s$ in $B_s$ do begin
                Check if $(t_r, t_s)$ satisfy the join condition
                if they do, add $t_r \bullet t_s$ to the result.
            end
        end
    end
end
```

Eg:

- Number of records of *customer*: 10,000 *depositor*: 5000
- Number of blocks of *customer*: 400 *depositor*: 100

***b_r * b_s + b_r* block transfers**
2 * *b_r* seeks

(Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation) – hence more efficient

- 400 * 100 + 400 = 40,400 transfers and 800 seeks

If seek time = 12ms

800*12 = 10secs (faster than simple nested)

Choosing the optimized path

Cost-based query optimization

1. Generate logically equivalent expressions of the query
2. Annotate resultant expressions to get alternative query plans
3. Choose the cheapest plan based on estimated cost

Estimation of plan cost based on:
- Statistical information about tables. Example:
  number of distinct values for an attribute
- Statistics estimation for intermediate results to compute cost of complex expressions
- Cost formulae for algorithms, computed using statistics again

The above approach is very expensive in space and time though

Some conditions to keep in mind when selecting the plan:

- Must consider the interaction of evaluation techniques when choosing evaluation plans

- Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm

  E.g., merge-join may be costlier than hash-join, but may provide a sorted output which could be useful later

Heuristic Approach

Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance.
Some of the rules are as follows:
1. Perform selections early (reduces the number of tuples)
2. Perform projections early (reduces the number of attributes)
3. Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations

Two Approaches:

1. Search all the plans and choose the best plan in a cost-based fashion.
2. Uses heuristics to choose a plan.

Equal Expression:

If the queries generate the same set of records.

Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Questions Query Optimizer:

Discuss what a query optimizer does and how it works briefly. Then discuss which approach would be more suitable for the following scenarios:

- Scenario A: Given a table with 10 million tuples, run the following query:

```
SELECT customer
FROM Table
WHERE spend BETWEEN 100 AND 200
AND birth_year> 2000;
```

- Scenario B: Given 5 tables with 1000 tuples in each table, run a query:

```
        SELECT T1.name, T2.salary, T3.qualification, T4.phone, T5.leader
        FROM Table1 T1
           INNER JOIN Table2 T2 ON T2.id = T1.id
           INNER JOIN Table3 T3 ON T3.id = T1.id
           INNER JOIN Table4 T4 ON T4.id = T1.id
           INNER JOIN Table5 T5 ON T5.department = T1.department
WHERE T1.age > 50;
```

Answer:

Queries are generally converted to Relational Algebra expressions internally first. Then the system tries to create alternate plans and pick the best plan to execute in terms of execution time. There are two general approaches for this. One is searching/enumerating all the plans and choose the best one. Another approach is using heuristics to choose a plan. (or a combination of two can be done as well.)

For Scenario A, the heuristic approach would be suitable due to the simplicity of the query.
For Scenario B, the enumerating approach would be more suitable due to the complexity of the query.

Nested Vs Block Nested

Block nested is more efficient as each block in the inner relation is read once for each block in the outer relation (instead of once for each tuple in the outer relation).

Manage Query Costs

SQL server management studio for monitoring - Query Store

Query stores conducts the troubleshooting in order to manage the query costs.

- Identify ‘regressed queries’ - Pinpoint the queries for which execution metrics have recently regressed (for example, changed to worse).

- Track specific queries - Track the execution of the most important queries in real time.

Steps that can be performed when a query with suboptimal performance is identified

1. Force a query plan instead of the plan chosen by the optimizer
2. Sometimes a change done on the query is not compiled in the optimizer, hence a statistic recompilation might help to bring everything to the same state
3. Indices are used to enhance the speed of the data retrieval operations on the database tables.
4. Query can be rewritten in a more efficient way by the user, eg: use of parameterized queries. As the optimizer generates the same plan for the parameterized queries.
5. Frequently used data that is derived can be stored for fast access
6. Tables which joined frequently can be pre-joined. But it needs to be updated regularly for the updates on the original table. It may return some outdated results.

Indices

Indices are used to enhance the speed of the data retrieval operations on the database tables. It allows almost direct access to the individual items.
Using indices is a good choice during the join operations, If it restricts the number of items to be joined in a table.

Search Key
attribute or set of attributes used to look up records/rows in a system like an ID of a person

Index Files

1. An index file consists of records (called index entries) of the form search-key, pointer to where data is. Single seek is required as pointer is present in the index files.
2. Index files are typically much smaller than the original data files and many parts of it are already in memory

Factors to be considered while using indexing

– Insertion time to index is also important

- Deletion time is important as well
- No big index rearrangement after insertion and deletion
- Space overhead needs to be considered for the index itself
- No single indexing technique is the best. Rather, each technique is best suited to particular applications

How a data is stored

Files – A database is mapped into different files. A file is a sequence of records.

Data blocks – Each file is mapped into fixed length storage units, called data blocks (also called logical blocks, or pages)

Types of Indices

Ordered indices: search keys are stored in some order
Hash indices: search keys are distributed hopefully uniformly across "buckets" using a "function"

Ordered Indices

The records in the indexed file may themselves be stored in some sorted order.

Clustering index/ Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file. It is a single seek as the data is in clusters (sequential order). If the query is using the search key, then it is a good option.

Non-clustering index/ Secondary index: an index whose search key specifies an order different from the sequential order of the file
Secondary indices improve the performance of queries that use keys other than the search key of the clustering index.

B+ Trees

The B+ tree is a balanced binary search tree. It follows a multi-level index format. In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height. B+ tree can support random access as well as sequential access. It is a best first search.

Advantages:

1. automatically reorganizes itself with small, local, changes, in the face of insertions and deletions
2. Reorganization of entire file is not required to maintain performance.

Disadvantages

1. Extra insertion and deletion overhead and space overhead

advantages outweigh the disadvantages

B+ tree definition

1. All paths from root to leaf are of the same length (depth)

2. Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children

3. A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values and it stores records instead of pointers

4. Special Case:
    a. If the root is not a leaf, it has at least 2 children
    b. If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.

Most of the higher level nodes of a B+tree would be in main memory already!

B+ Tree Execution

```
1. N=root initially
2. Repeat
    1. Examine N for the smallest search-key value > k.
    2. If such a value exists, assume it is Ki.  Then set N = Pi
    3. Otherwise k ≥ Kn–1. Set N = Pn . Follow pointer.
   Until N is a leaf node
3. If for some i, key Ki = k  follow pointer Pi  to the desired record or bucket.
4. Else no record with search-key value k exists.
```

B+ Tree Metrics

$$\text{Height} = \lceil \log_{\lceil n/2 \rceil}(K) \rceil$$

B+ and Binary Tree

B+ trees are like Binary trees in many aspects, but the fan out is much higher.

The main advantage of a large fanout is the time the reduction in the number of hops required to locate data.

Due to the way the B+ tree is designed the given a fanout n the max height that a tree can achieve is ceil($\log_{ceil(n/2)}(k)$) where k is the number of search keys in the tree.
This implies that we can find any search key, if it is present in a tree, in ceil($\log_{ceil(n/2)}(k)$) hops. Eg: If there are a million search keys and a fanout of 100, the hops required would be ceil($\log_{50}(1,000,000)$) which is 4.
The same number of search keys would have taken approximately 20 hops in a binary tree.

B+ and Hash Indices

B+ tree supports range of queries, so we can retrieve all records >,< or = a search key value. Whereas,Hashing is built for finding exact matches based on a search key. Hash Indexes are much faster for searching a record based on a search key and tries to return in 1 hop (approximately 1.2 hops), whereas B+ tree takes a ceil(logceil(n/2)(k)) hops.

Hash Indices

A hash index organizes the search keys, with their associated record pointers, into a hash file structure. Order is not important. These indices are always secondary.
The aim of the hash indices is to find the related record in one shot. The search key is passed to a function and it returns the location for the record.

An ideal hash function is uniform, so that each bucket is assigned the same number of search key values from the set of all possible values. It should also be random, so that each bucket will have the same number of records assigned to it irrespective of the actual distribution of the search-key values.
Hash functions perform computation on the internal binary representation of the search key.

Function (search key) -> Pointer to the bucket

Bitmap Indices

Records in a relation are assumed to be numbered sequentially from, say, 0. It is applicable on attributes that take relatively small number of distinct values. Eg: gender, country etc. It is simply an array of bits.
It is generally used in business analysis, where how type of one category exists in the data.
Eg:

Name	Gender
R	M
G	F
H	M

M [101] and F [010]

Spatial Indices

Spatial data requires complex computations for accessing data, e.g., finding the intersection of objects in space.

Nearest Neighbour Query

Similar to B+ trees, the 2d space is exponentially reduce the number of calculations by repetitive division of space. It is a best first search.
Class of one such data structure in **QuadTree.**

QuadTree

Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.

Each division happens with respect to a rule based on data type.

Each non-leaf nodes divides its region into four equal sized quadrants
Thus each such node has four child nodes corresponding to the four quadrants and division continues recursively until a stopping condition

Eg of stopping point: Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example below)

R Trees

R-trees are an N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.

Generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N

Bounding boxes of children of a node are allowed to overlap

A **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node

Search in R Trees

To find data items intersecting a given query point/region, do the following, starting from the root node:

- If the node is a leaf node, output the data items whose keys intersect the given query point/region.
- Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child.

Can be very inefficient in worst case since multiple paths may need to be searched due to overlaps, but works acceptably in practice.

Indices in DBMS

- For UNIQUE constraint, DBMS creates a nonclustered index.
- For PRIMARY KEY, DBMS creates a clustered index
(A unique index is one in which no two rows are permitted to have the same index key value)

create index <index-name> **on** <relation-name> (<attribute-list>)
drop index <index-name>

CREATE CLUSTERED INDEX index1 ON table1 (column1);
CREATE UNIQUE INDEX index1 ON table1 (column1 DESC, column2 ASC, column3 DESC);

(A filtered index is an optimized nonclustered index, suited for queries that select a small percentage of rows from a table. It uses a filter predicate to index a portion of the data in the table)

CREATE INDEX index1 ON table1 (column1)
WHERE Year > ‘2010’;

Spatial index
CREATE SPATIAL INDEX index_name ON table_name(Geometry_type_col_name) WITH (BOUNDING_BOX = (0, 0, 500, 200));

Steps to do when uploading in DBMS

- Find the potential query types
- Research what indices that particular DBMS would have for that data type
- Research for what queries you would better do on what index
- Create index if you have large data
- Monitor performance
- Tune or create other indices
- Your DMBS will have a version of the “create index” SQL statement

R tree and Indices Questions

Review the points on indexing with B+ trees. Assume a database table has 10,000,000 records and the index is built with a B+ tree. The maximum number of children of a node, is denoted as n. How many steps are needed to find a record if n=4? How many steps are needed to find a record if n=100?

Solution:

• When n = 4, the maximum height of the tree is

$\lceil \log_{\lceil n/2 \rceil}(K) \rceil = \lceil \log_2(10000000) \rceil = 24$

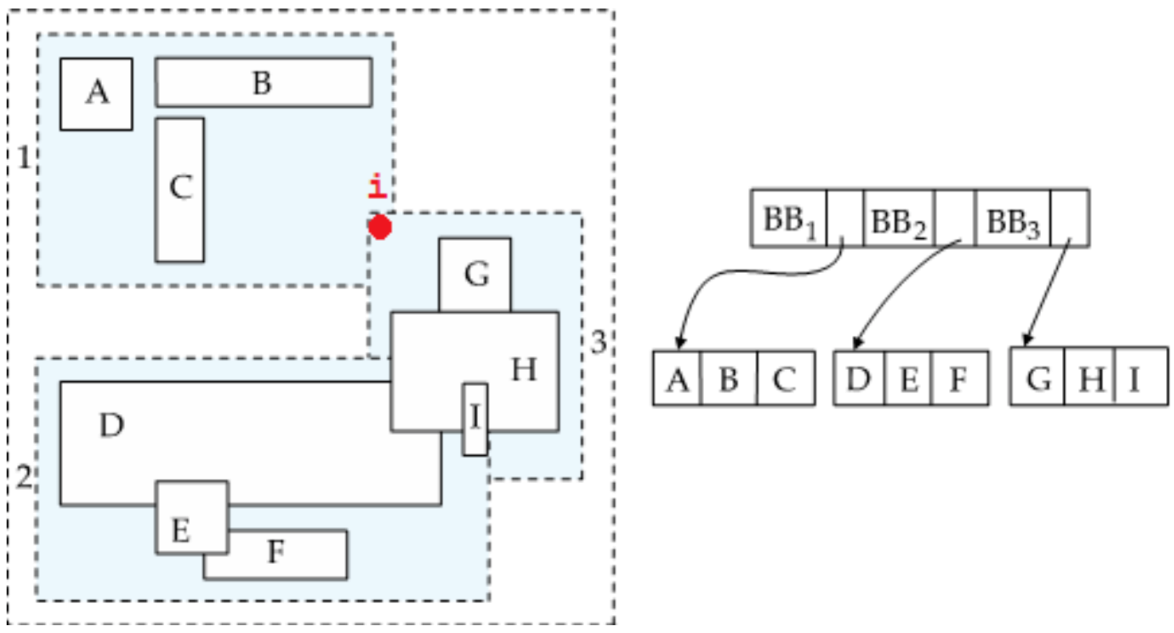
Therefore, 24 steps are needed.

• When n = 100, the maximum height of the tree is

$\lceil \log_{\lceil n/2 \rceil}(K) \rceil = \lceil \log_{50}(10000000) \rceil = 5$

Therefore, 5 steps are needed.

Given the R-tree below please visit the nodes of the R-tree in a best-first manner as discussed in class to find the 1st nearest neighbour of query point “i”. Is there anything peculiar that you notice while traversing an R-tree?



Solution:

In this traversal we first visit node BB_1 as it overlaps with the query point. And find that object B is the closest to query point i.

The issue here is that we cannot stop at this point in the traversal as i overlaps with BB_3 as well so we need to investigate the data there too.

We then figure out that G is the closest object overall. Due to overlaps in R-tree branches two or more branches of an R-tree need to be investigated in many query types.

In addition, as each internal node represents a bounding box, thus we are not sure about the position of objects in a bounding box which may necessitate that we investigate multiple bounding boxes to determine a nearest neighbour in this case.

Database Transactions

A transaction is a unit of work in a database. It can have any number and type of operations in it. In DBMS, a transaction happens as a whole or do not happen at all. It has four properties, ACID (discussed earlier)

Transaction Models Actions

- Unprotected actions - no ACID property

- **Protected actions -** these actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.

- **Real actions -** these are real physical actions once performed cannot be undone. In many situations, atomicity is not possible with real actions (e.g., firing two rockets as a single atomic action)

Types of Transactions

Flat Transactions

All the actions inside BEGIN WORK and COMMIT WORK is at the same level. It means that all the actions survive together (commit) or everything will be rolled back (abort).

Eg: Bank debit and credit

Disadvantages:

The completed and successful actions needs also to be redone, which is an extra overhead. A long running flat transaction when rolled back results in a lot of unnecessary computations.

Eg. Airline booking.

```
BEGIN WORK
                    S1: book flight from Melbourne to Singapore
                    S2: book flight from Singapore to London
                    S3: book flight from London to Dublin
               END WORK
```

Problem: from Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination.
If we roll back we need to re do the booking from Melbourne to Singapore *which is a waste*

Flat Transactions with Save Points

It is a type of flat transactions with the SAVE point. Hence, all the actions need not to be done again.

```
BEGIN WORK
SAVE WORK 1
Action 1
Action 2
SAVE WORK 2
Action 3
Action 4
Action 5
SAVE WORK3
Action 6
Action 7
ROLLBACK WORK(2)
```

Nested Transaction

Transactions are divided into parent-child transactions. Parent can invoke the sub transactions.

Commit rule
- A subtransaction can either commit or abort, however, **commit cannot take place unless the parent itself commits.**
- Subtransactions have A, C, and I properties but not D property unless all its ancestors commit.
- Commit of a sub transaction makes its results available only to its parents.

Roll back Rules
If a subtransaction rolls back, all its children are forced to roll back.

Visibility Rules
Changes made by a subtransaction are visible to the parent only when the subtransaction commits. All objects of parent are visible to its children. Implication of this is that the **parent should not modify objects while children are accessing them.** This is not a problem as parent does not run in parallel with its children.

Transaction Processing Monitor

TP monitor integrates other system components and manage resources. It manages the transfer of data between clients and servers. It breaks down the application or code into transactions and ensure all databases are updated properly. It is also responsible for error handling in case of a system crash.

TP Monitor Services

Heterogeneity: If the application needs access to different DB systems, local ACID properties of individual DB systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure the overall ACID property. A form of 2 phase commit protocol must be employed for this purpose (will be discussed later).
Control communication: If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes to be able to recover from a crash.
Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID property between the client and the server processes.
Presentation service: this is similar to terminal management in the sense it has to deal with different presentation (user interface) software -- e.g. X-windows
Context management: E.g. maintaining the sessions etc.
Start/Restart: There is no difference between start and restart in TP based system.

Concurrency Control

Database concurrency is the ability of a database to allow multiple users to affect multiple transactions. This might result in inconsistent data. Hence, concurrency control is important to resolve conflicts, and preserve database consistency.

Ways of Consistency Control

Slides from lecture 3 part 3

Dekker's algorithm (using code) - needs almost no hardware support, but the code is very complicated to implement for more than two transactions/processes

- **the code is very complicated to implement if more than two transactions/process are involved**
- takes lot of storage space
- uses **busy waiting** (a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution)
- efficient if the lock contention (that is frequency of access to the locks) is low

OS supported primitives (through interruption call) - expensive, independent of number of processes, machine independent
OS supported primitives such as lock and unlock

- through an interrupt call, the lock request is passed to the OS
- need no special hardware
- **are very expensive (several hundreds to thousands of instructions need to be executed to save context of** the requesting process.)

- do not use busy waiting and therefore more effective

Spin locks (using atomic lock/unlock instructions)

Executed using atomic machine instructions such as test and set or swap

- need hardware support – should be able to lock bus (communication channel between CPU and memory + any other devices) for two memory cycles (one for reading and one for writing). During this time no other devices' access is allowed to this memory location.
- use busy waiting
- algorithm does not depend on number of processes
- are very efficient for low lock contentions – all DB systems use them

testAndSet(int *lock) and compare and swap is used in the spin locks.

Semaphores

Computer semaphores have a get() routine that acquires the semaphore (perhaps waiting until it is free)
and a dual give() routine that returns the semaphore to the free state, perhaps signalling (waking up) a waiting process.

Semaphores are very simple locks; indeed, they are used to implement general-purpose locks.

Implementation of Semaphores

It points to the queue of the processes.

If the semaphore is busy but there are no waiters, the pointer is the address of the process that owns the semaphore.

If some processes are waiting, the semaphore points to a linked list of waiting processes. The process owning the semaphore is at the end of this list.

After usage, the owner process wakes up the oldest process in the queue (first in, first out scheduler)

Convoys

Semaphores implementation might result in a long list of waiting processes called convoy.

To avoid convoys, a process may simply free the semaphore (set the queue to null) and then wake up every process in the list after usage.

In that case, each of those processes will have to re-execute the routine for acquiring semaphore.

Deadlocks

In a deadlock, each process in the deadlock is waiting for another member to release the resources it wants.

Deadlock Solution

- Have enough resources so that no waiting occurs – not practical

- Do not allow a process to wait, simply rollback after a certain time. This can create live locks which are worse than deadlocks.

- **Linearly order** the resources and request of resources should follow this order, i.e., a transaction after requesting i^{th} resource can request j^{th} resource if $j > i$ .This type of allocation guarantees no cyclic dependencies among the transactions.

Deadlock Mitigation

- Pre-declare all necessary resources and allocate in a single request

- Periodically check the resource dependency graph for cycles. If a cycle exists - rollback (i.e., terminate) one or more transaction to eliminate cycles (deadlocks). The chosen transactions should be cheap (e.g., they have not consumed too many resources).

- Allow waiting for a maximum time on a lock then force Rollback. Many successful systems (IBM, Tandem) have chosen this approach

- Many distributed database systems maintain only local dependency graphs and use time outs for global deadlocks.

Deadlocks are usually very rare and most companies allow the maximum time on a lock then force Rollback.

Isolation Concepts

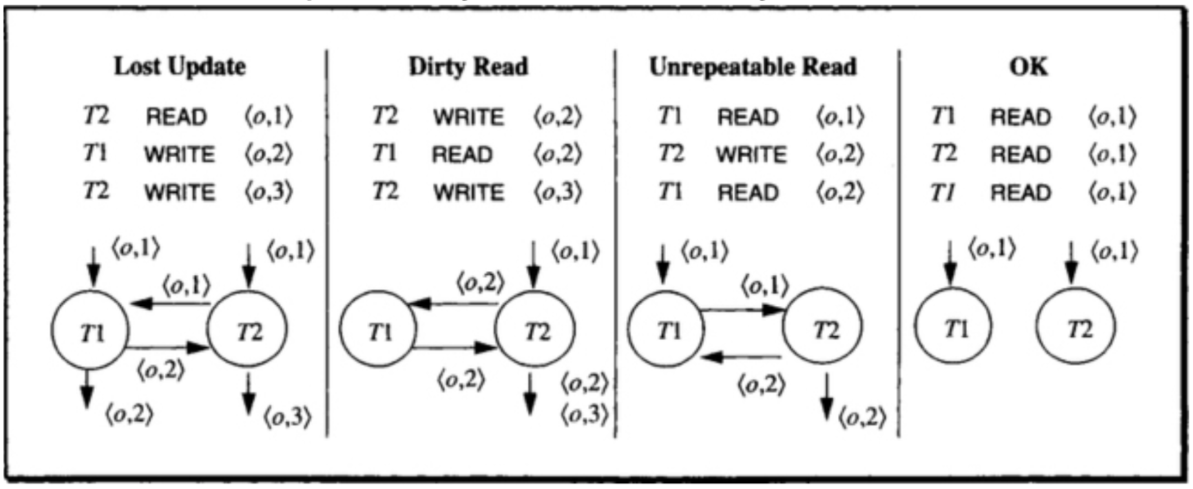
To achieve isolation we need to understand dependency of operations

We need to run transactions concurrently with the following goals:

- concurrent execution should not cause application programs (transactions) to malfunction.
- Concurrent execution should not have lower throughput or bad response times than serial execution.

Possible Dependencies

When dependency graph has cycles then there is a violation of isolation and a possibility of inconsistency.



Lost Update

T2	READ	⟨o,1⟩
T1	WRITE	⟨o,2⟩
T2	WRITE	⟨o,3⟩

Dirty Read

T2	WRITE	⟨o,2⟩
T1	READ	⟨o,2⟩
T2	WRITE	⟨o,3⟩

Unrepeatable Read

T1	READ	⟨o,1⟩
T2	WRITE	⟨o,2⟩
T1	READ	⟨o,2⟩

OK

T1	READ	⟨o,1⟩
T2	READ	⟨o,1⟩
T1	READ	⟨o,1⟩

% T2 did not see the update of T1

What if T2 aborts? T1's read will be invalid

The value of o changes by another transaction T2 while T1 is still running

Dependency Model

I_i : set of inputs (objects that are read) of a transaction T_i
O_i : set of outputs (objects that are modified) of a transaction T_i
Note O_j *and* I_j are not necessarily disjoint that is $O_j \cap I_j \neq \emptyset$
Given a set of transactions τ , Transaction T_j has no dependency on any transaction T_i in τ if-

$$O_i \cap (I_j \cup O_j) = empty\ for\ all\ i \neq j$$

This approach cannot be planed ahead as in many situation inputs and outputs may be state dependant/not known in prior.

Dependency Graph

We mainly focuses on Write-Write, Write-Read, Read-Write dependency.

Given two histories H1 and H2 contain the same tuples, H1 and H2 are **equivalent** if DEP(H1) = DEP(H2)

This implies that a given database will end up in exactly the same final state by executing either of the sequence of operations in H1 or H2

H1 = <(T1,R,O1), (T2, W, O5), (T1,W,O3), (T3,W,O1), (T5,R,O3), (T3,W,O2), (T5,R,O4), (T4,R,O2), (T6,W,O4)>
DEP(H1) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

Isolation History

A history is said to be isolated if it is equivalent to a serial history. A serial history is history that is resulted as a consequence of running transactions sequentially one by one.

A transaction T' is called a wormhole transaction if

$$T' \in Before(T) \cap After(T)$$

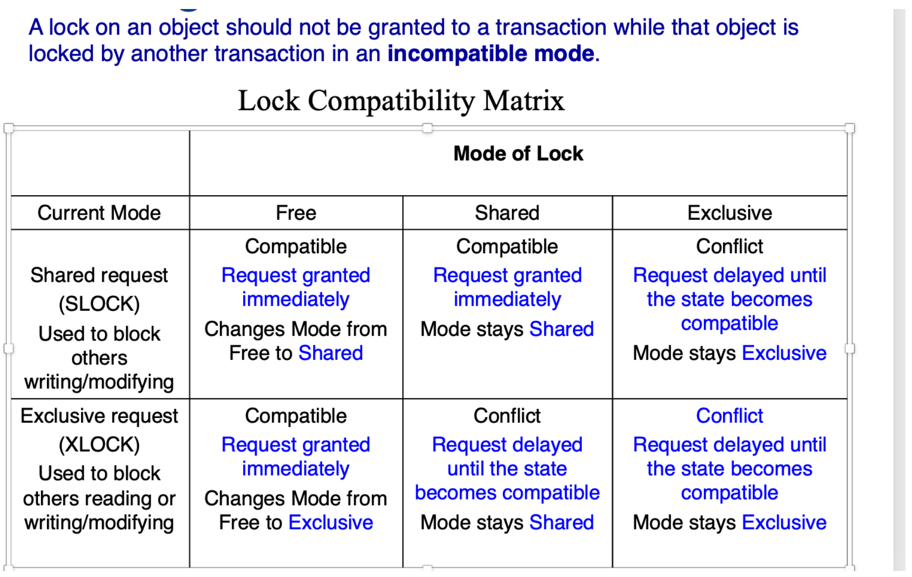
T << T' << T. This implies there is a cycle in the dependency graph of the history.

Presence of a wormhole transaction implies it is not isolated.

Lock Types

SLOCKS

allows other transactions to read, but not write/modify the shared resource



A lock on an object should not be granted to a transaction while that object is locked by another transaction in an **incompatible mode**.

Lock Compatibility Matrix

	Mode of Lock		
Current Mode	Free	Shared	Exclusive
Shared request (SLOCK) Used to block others writing/modifying	Compatible Request granted immediately Changes Mode from Free to Shared	Compatible Request granted immediately Mode stays Shared	Conflict Request delayed until the state becomes compatible Mode stays Exclusive
Exclusive request (XLOCK) Used to block others reading or writing/modifying	Compatible Request granted immediately Changes Mode from Free to Exclusive	Conflict Request delayed until the state becomes compatible Mode stays Shared	Conflict Request delayed until the state becomes compatible Mode stays Exclusive

Isolation Theorems

- **Well-formed transactions**: A transaction is well formed if all READ, WRITE and UNLOCK operations are covered by appropriate LOCK operations
- **Two phase transactions**: A transaction is two phased if all LOCK operations precede all its UNLOCK operations.

A transaction is well formed if each READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation.

A history is legal if does not grant conflicting grants.

A transaction is two phase if its all lock operations precede its unlock operations.

Locking theorem: If all transactions are well formed (READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation) and two-phased (locks are released only at the end), then any legal (does not grant conflicting grants) history will be isolated.

Locking theorem (Converse): If a transaction is not well formed or is not two-phase, then it is possible to write another transaction such that it is a wormhole.

Degree of Isolation

A Three degree isolated Transaction has no lost updates, and has repeatable reads. This is "true" isolation.

Lock protocol is two phase and well formed.

It is sensitive to the following conflicts:

write->write; write ->read; read->write

Degree 3

```
Slock(A)
Xlock(B)
Xlock(C)
Read(A)
Write(B)
Read(A)
Write(C)
Unlock(A)
Unlock(B)
Unlock(C)
```

Degree 2: A Two degree isolated transaction has no lost updates and no dirty reads.

Lock protocol is two phase with respect to exclusive locks and well formed with respect to Reads and writes. (May have Non repeatable reads.)

It is sensitive to the following conflicts:

write->write; write ->read;

Degree 2

```
Slock(A)
Read(A)
Unlock(A)
Xlock(C)
Xlock(B)
Write(B)
Slock(A)
Read(A)
Unlock(A)
Write(C)
Unlock(B)
Unlock(C)
```

Degree 1: A One degree isolation has no lost updates.

Lock protocol is two phase with respect to exclusive locks and well formed with respect to writes.

It is sensitive the following conflicts:
write->write;

```
Degree 1
Read(A)
Xlock(C)
Xlock(B)
Write(B)
Read(A)
Write(C)
Unlock(B)
Unlock(C)
```

Degree 0 : A Zero degree transaction does not overwrite another transactions dirty data if the other transaction is at least One degree.

Lock protocol is well-formed with respect to writes.

It ignores all conflicts.

```
Degree 0
Read(A)
Xlock(B)
Write(B)
Unlock(B)
Read(A)
Xlock(C)
Write(C)
Unlock(C)
```

Granular Locks

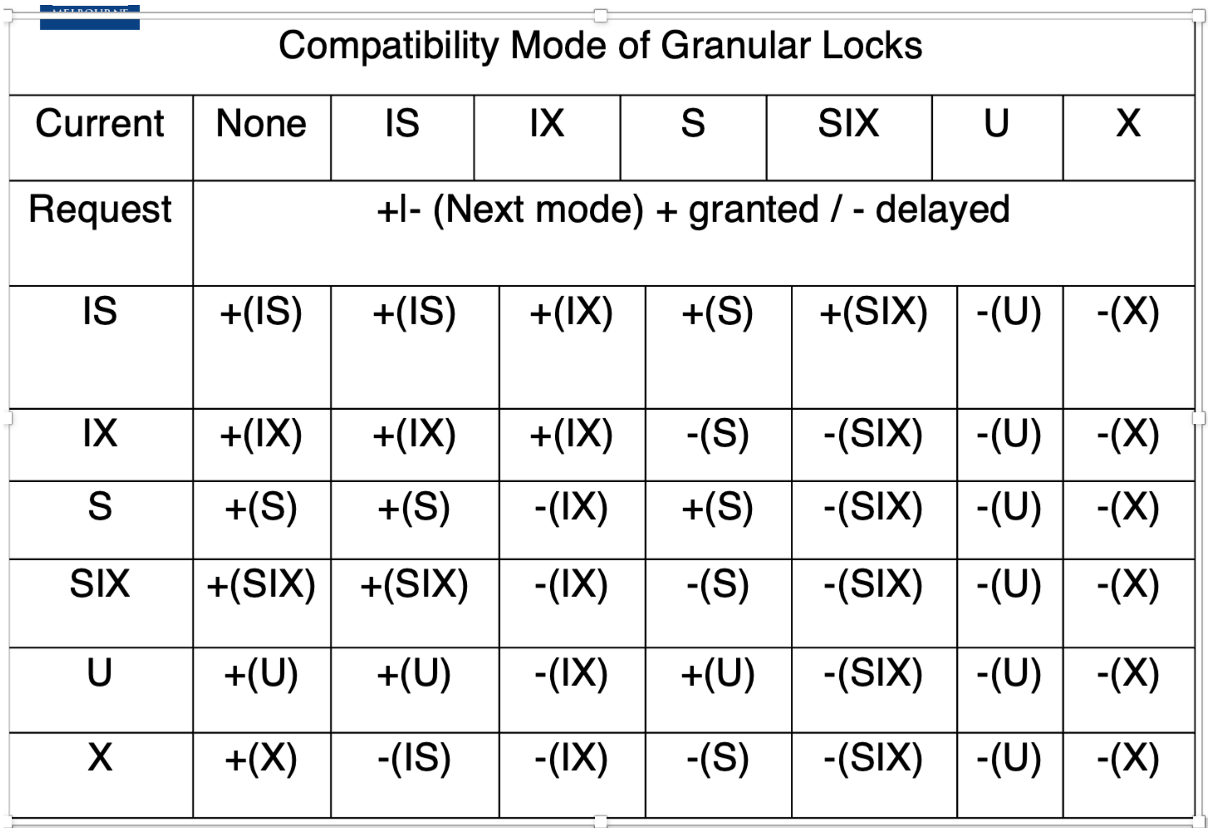
Granular locks can be taken at any level, which automatically grant the locks to the descendants.
Slides from lecture 3 part 5

Advantages:

1. Locking the whole Database will result in less conflict but will also result in less efficiency and poor performance
2. Locking at the individual level will require more locks, but will provide with better performance.

The best of both the worlds is achieved by implementing the Intention mode locks.

Compatibility Matrix



Compatibility Mode of Granular Locks

Current	None	IS	IX	S	SIX	U	X
Request	+I- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)

Optimistic Locking

When conflicts are rare, transactions can execute operations without managing locks and without waiting for locks. This will yield higher throughput.

The data is used without locks and the assumption is made that the different parts of the data is being accessed by the different users.

Before the commit, each transaction verifies that no other transaction has modified the data by taking appropriate locks, but the locks duration is very less.

If the conflict is found then transaction repeats the attempt else make changes and commit.

Snapshot Isolation

In databases, and transaction processing, snapshot isolation is a that all reads made in a transaction will see a consistent snapshot of the database, and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.

Timestamping

These are a special case of optimistic concurrency control. At commit, time stamps are examined. If time stamp is more recent than the transaction read time the transaction is aborted because it has been modified.

Data is never overwritten a new version is created on update.

This model of computation unifies concurrency, recovery and time domain addressing

Dependency graph answer sample

As we can see there are no loops in the dependency graphs and no wormhole transactions (transaction T that occurs both before and after any Transaction J). Hence this transaction can be serialised with the equivalent
serial execution history as follows:
h = < (k,R,a),(k,W,c),(m,W,a),(m,W,b),(n,R,b),(l,W,e),(o,R,c),(o,R,d), (p,W,d)>
with the same dependency graph as seen above.

Transactions Questions

In a nested transaction, a transaction PARENT has three sub-transactions A, B, C. For each of the following scenarios, answer which of these four transactions' commits can be made durable, and which ones has to be forced to rollback.

(i) Scenario 1: Commit by A, B, and C; but PARENT rolls back. – based on the rollback rule, all four transactions rolls back, no durable commit by any transaction.

(ii) Scenario 2: Commit by A, B, C, and PARENT. – all four transactions make durable commits, no roll back.

(iii) Scenario 3: Commit by A, B, and PARENT; but C rolls back. - Durable commits by A,B, and Parent; roll back by C only.

What is the probability that a deadlock situation occurs?

If we use the following comments to lock and unlock access to objects, then which transactions below are in deadlock if they start around the same time?

T1	T2	T3	T4
Begin	Begin	Begin	Begin
LOCK(C)	LOCK(A)	LOCK(C)	LOCK(B)
Write C	Write A	Write C	Write B
UNLOCK(C)	LOCK(B)	UNLOCK(C)	LOCK(A)
End	Write(B)	End	Write A
	UNLOCK(A)		UNLOCK(A)
	UNLOCK(B)		UNLOCK(B)
	End		End

Solution:
T2 and T4 are in a deadlock as each of them will wait for the other to release a lock while holding a lock that the other needs to acquire to complete.

Assume the following two transactions start at nearly the same time and there is no other concurrent transaction. The 2nd operation of both transactions is Xlock(B). Is there a potential problem if Transaction 1 performs the operation first? What if Transaction 2 performs the operation first?

Transaction 1		Transaction 2	
1.1	Slock(A)	2.1	Slock(A)
1.2	Xlock(B)	2.2	Xlock(B)
1.3	Read(A)	2.3	Write(B)
1.4	Read(B)	2.4	Unlock(B)
1.5	Write(B)	2.5	Read(A)
1.6	Unlock(A)	2.6	Xlock(B)
1.7	Unlock(B)	2.7	Write(B)
		2.8	Unlock(A)
		2.9	Unlock(B)

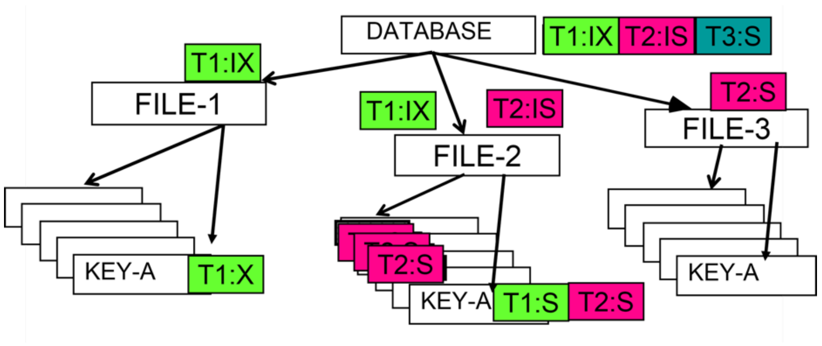
:

If Transaction 1 executes Step 1.2 before Transaction 2 executes Step 2.2, there would be no problem.

This is because T1 releases the lock at the end. T2 has to wait till then.

However, if Transaction 2 executes Step 2.2 first, Transaction 1 may have a dirty read of B if it tries to run Step 1.2 immediately after Transaction 2 releases the Xlock on B. This is because when Transaction 2 releases the lock on B (Step 2.4), Transaction 1 will be granted the Xlock on B (Step 1.2). After that, Transaction 1 will read B (Step 1.4). After Transaction 1 completes, Transaction 2 will acquire another Xlock on B (Step 2.6) then modify the object. In other words, the reading of B by Transaction 1 happens between two writes of B by Transaction 2.

Review the concepts of granular locks then answer the following question. Given the hierarchy of database objects and the corresponding granular locks in the following picture, which transactions can run if the transactions arrive in the order T1-T2-T3? What if the order is T3-T2-T1? Note that locks from the same transaction are in the same colour. We assume that the transactions need to take the locks when they start to run.



Solution:
If the order of the arrival of transactions is T1-T2-T3, then T1 and T2 can run in parallel while T3 waits. This is because

T1's IX lock at the root node is not compatible with T3's S lock at the same node.

If the order is T3-T2-T1, then T3 and T2 can run while T1 waits. This is due to the similar reason as above.

This example shows that the order of transactions can be a deciding factor of the set of parallel-running transactions.

We should also note that granular locks can lead to the delay of transactions at any level in the hierarchy below the root node, e.g., a transaction may need to wait for a lock at the FILE-3 node or a KEY-A node due to lock compatibility issues.

With two-phase locking we have already seen a successful strategy that will solve concurrency problems for DBMSs. Then discuss why someone may want to invent something like Optimistic Concurrency control in addition to that locking mechanism.

Solution:

Two-phase locking or in general locking assumes the worst, i.e. there are many updates in the system and most of them will lead to conflicts in access to objects. This means there is good rationale to pay the overhead of locking and stop problems from occurring in the first place.

But what if the DBMS is one such that people tend to work on different parts of data, or most of the operations are read operations, and as such there aren't many conflicts at all. Then there is no need to pay the overhead of lock management but rather it may be better to allow transactions run freely and have a simple check when they finish whether there was any conflict with concurrent transactions.

Most of the time there will not be so one will get increased concurrency with less overheads. Obviously, the reverse is also true, i.e., if there were really many conflicting writes then doing optimistic concurrency control means many problems would be observed only after running the transactions and a lot of work will need to be wasted to preserve consistency of the data. So there is no clear answer but depending on the situation a strategy may be good or bad

What is cascading aborts? Does two-phase locking address them? If it does, explain how, but if it does not, then give a locking-based strategy that may address them, and then explain briefly how that strategy that you gave will address them.

Cascading aborts occurs when roll back of a transaction requires the rollback of another transaction since it is reading uncommitted rolled back data.

In two phase locking all the locks are released in the shrinking phase that happens before the commit has taken place. Since now the locks are released on the data items, between the shrink phase and commit another transaction can acquire a lock on the data item and access it. In this case if the earlier transaction aborts before commit, then the second transaction should also be rolled back to return the DB to a consistent state since it is reading uncommitted data.

This can be avoided using strict two phase locking. In strict two phase locking, there is no shrinking phase, locks are released only during commits ,i,e, the transaction holds a lock on the data items until it is committed avoiding the issue of cascading aborts, since no other transaction can access the modified information until it is committed.

What is the difference between a classical Optimistic Concurrency control mechanism versus a Snapshot Isolation-based one. What is the implication of this difference? Briefly explain.

In optimistic concurrency control, it is assumed that the transactions can execute without any locks. It checks the conflicts during the commit by either forward or backward validation strategy.

In Snapshot isolation each transaction reads snapshot of data at the start of the transaction and then at the commit stage it is checked if the original snapshot has changed or not.

Snapshot isolation does not guarantee serializability. Hence, another transaction can use and modify the data after the original transaction has started and read the data causing conflicts. It also does not check for reads that happen in between.

Crash Recovery

The recovery management guarantees Atomicity & Durability, assuming that the concurrency is in effect.

Buffer Caches

1. Data is stored on disks
2. Read data is done by reading the whole page (4K or 8K bytes) of data from the disk
3. Writing or modifying a data is done by first reading the whole page to the main memory, performing the operation and then writing the whole page to the disk.
4. Reading and Writing is expensive, it can be reduced by checking the buffer cache of the memory for the desired page
5. When the buffer cache is full, we need to evict the pages
6. Eviction should make sure that no other process is using the page
7. In order to make sure, a locking mechanism is put into place, called latches. These latches are used only for the duration of the operation (e.g. READ/WRITE) and can be released immediately unlike record locks which have to be kept locked until the end of the transaction.
8. fix(pageid)
    1. reads pages from disk into the buffer cache if it is not already in the buffer cache
    2. fixed pages cannot be dropped from the buffer cache as transactions are accessing the contents
9. unfix(pageid)
    1. The page is not in use by the transaction and can be evicted as far as the unfix calling transaction is concerned. (We need to check to see that no one else wants the page before it can be evicted

Buffer Pool

Force:
It is when the pages updated by the transactions are immediately written to disk. Even though, It provides durability but it results in poor response.

No Force:
It leaves page in memory as long as possible even after the commit without modifying the data on the disk. It improves response time and efficiency as many reads can be done in the main memory itself. It might be less durable, as main memory is volatile and might not handle the crash.

Steal
If the recovery protocol allows writing an updated buffer before the transaction commits, it is called steal. Enforcing atomicity is hard. If a page P in frame F is written to disk, and some transaction holds lock on P and then the transaction aborts. This will result in inconsistent state.

Suppose a transaction T1 wants to read a data object X, but the working memory is full with all the other transactions' work. So T1 needs to clear some memory, which it does by kicking some other page in working memory to stable storage. This can be dangerous, because we can't be sure that what T1 is pushing to stable storage has been committed yet. This is known as stealing.

Most crash recovery uses a steal/no-force approach, accepting the risks of writing possibly uncommitted data to memory to gain the speed of not forcing all commit effects to memory.

Logging

Record redo and undo information in the log.

Lazy Writing:
lazy writer is a system process that inspects and removes infrequently used pages in memory with the goal of keeping free buffer (memory) available. After the dirty pages are written to the disk, they are then removed from the memory.

Dirty Page
SQL server stores data in 8 KB pages. When user submits a query asking for data, SQL Server locates the data and retrieves the data pages that contain those information. The process involves retrieving the data pages from the disk into the buffer cache (memory), and returns the result to the user. If user submits a update query on these data, SQL server makes modification on the corresponding data pages in the buffer cache, without updated the data pages on the disk immediately. These modified pages on memory that have not written disk are called dirty pages.