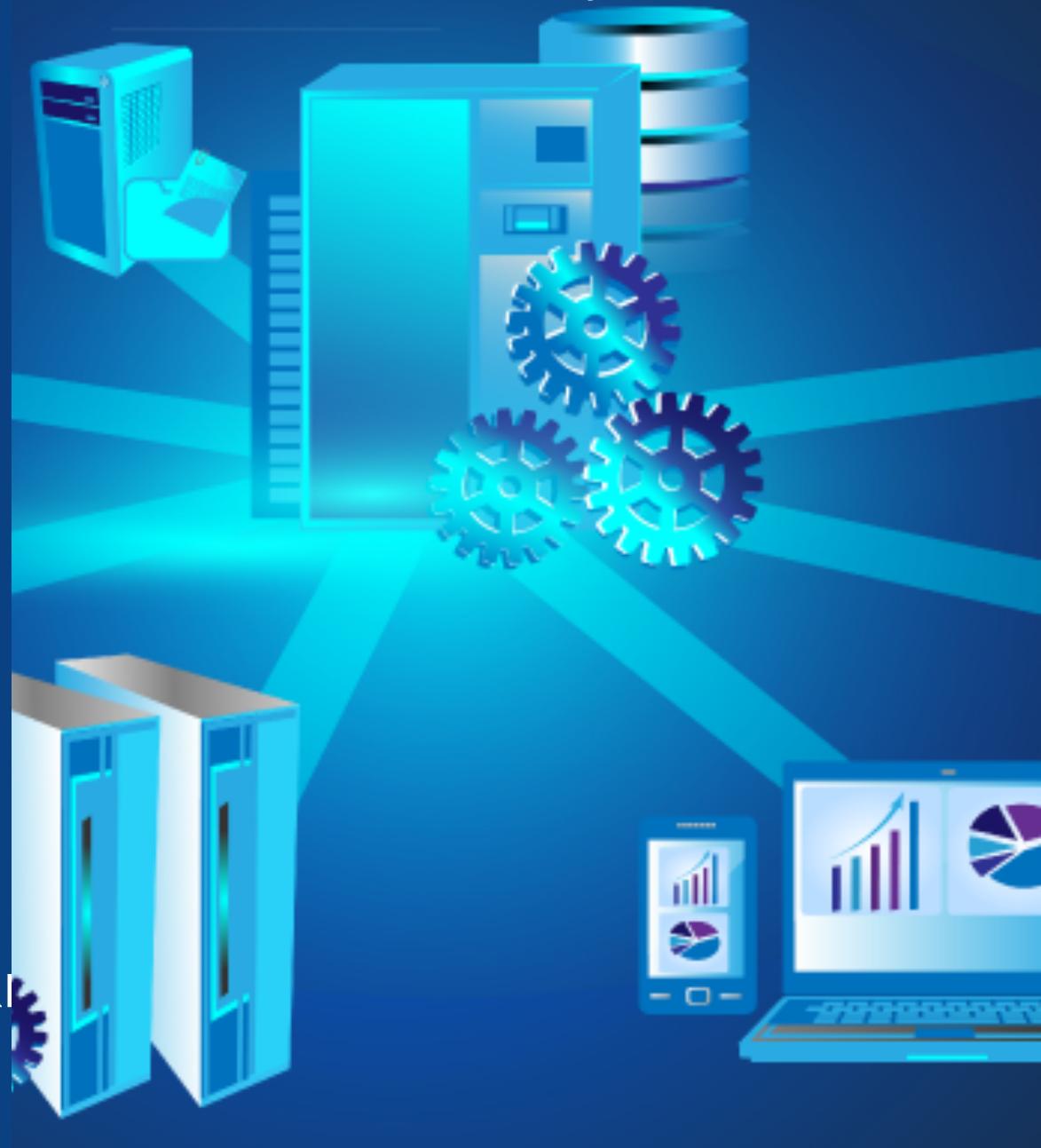




THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

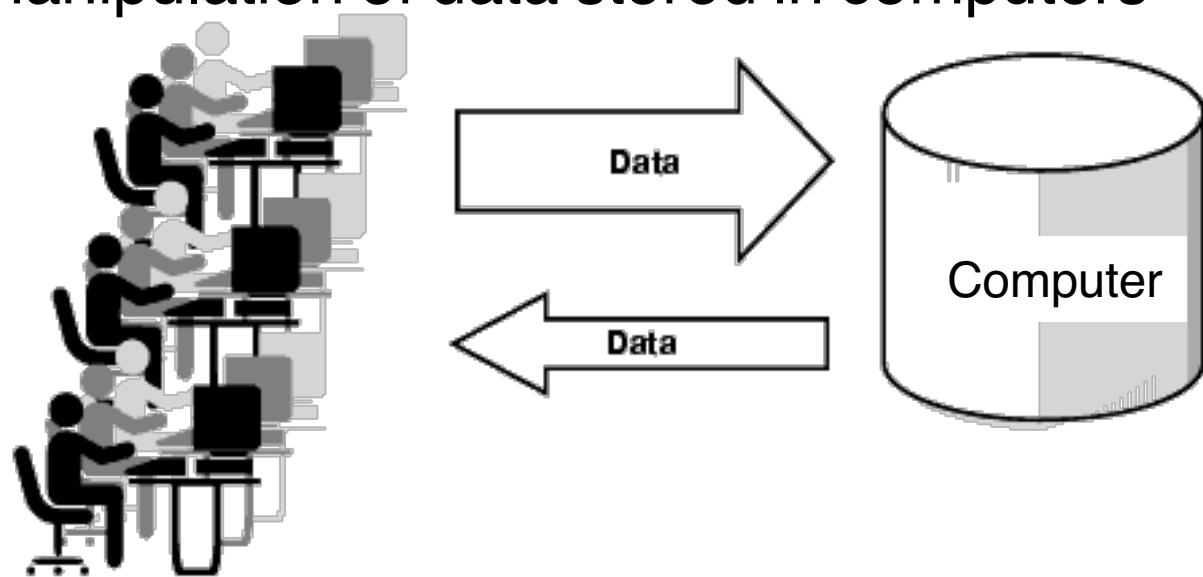
COMP90050 Advanced Database Systems



Lecturer: Prof Egemen Tan

Before All: Why study this subject?

All successful companies and organizations rely on the effective and efficient manipulation of data stored in computers

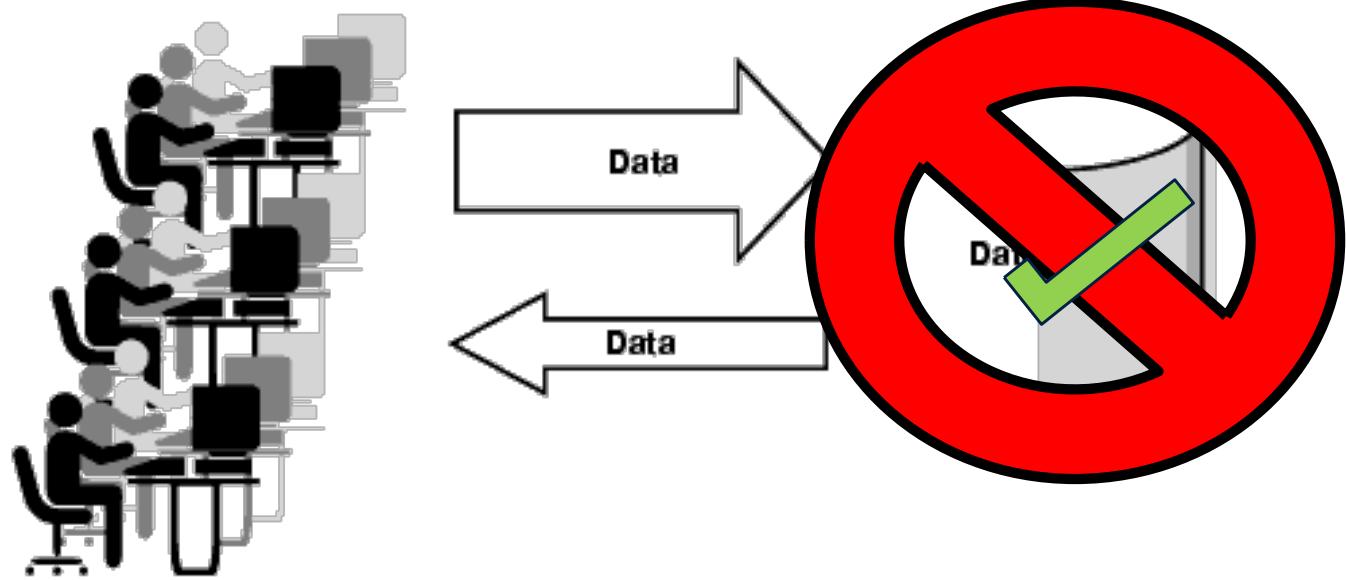


Increasingly this data:

- Includes more and **more aspects** of businesses
- **Getting larger** and larger
- **More complex** and includes novel data types such as images, etc
- **Stored in various sites** and **accessed by many users**

Motivation Contd.

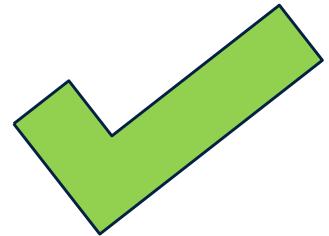
Historically, companies stored their **data across a few files**



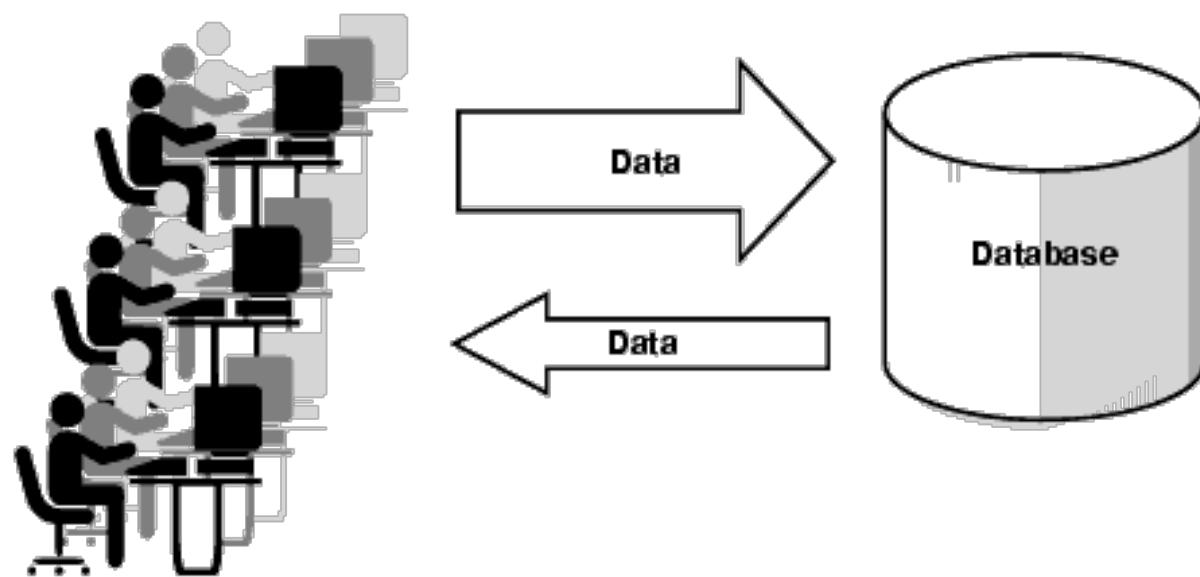
Why: This eventually became inadequate:

- As data grows one or **many files gets cumbersome** to store data
- As users who access it grow it becomes **hard to coordinate access**
- As systems grow, it is **hard to guard against loss, format differences, breach or damage to data...**

Thus: Came the Database Era



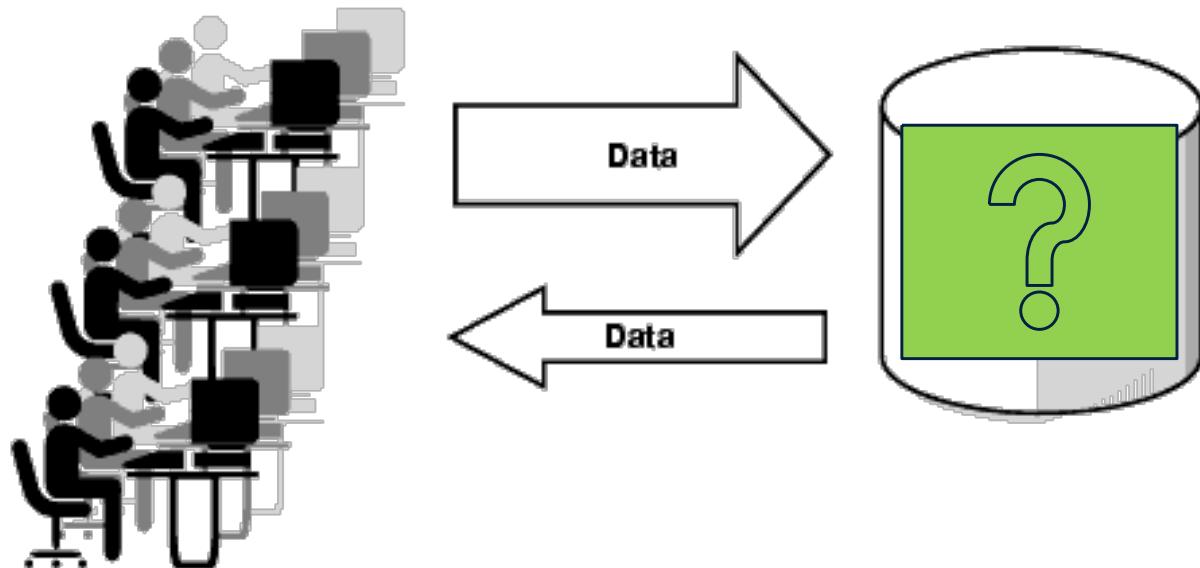
Eventually people developed the concept of Databases which is at the base of many systems that store data today



A database system thus provides

- **Secure and reliable storage of data**
- Ability to **retrieve and process the data effectively and efficiently**

You learned how to use them but...



- What is in a **Database Management System** and also:
 - Can you tune them
 - Do you know how to reduce chances of failure
 - Alternatives to some default strategies, e.g., going beyond simple thoughts of locks for concurrency control...

Hence, we will look **under the hood of Database Systems**

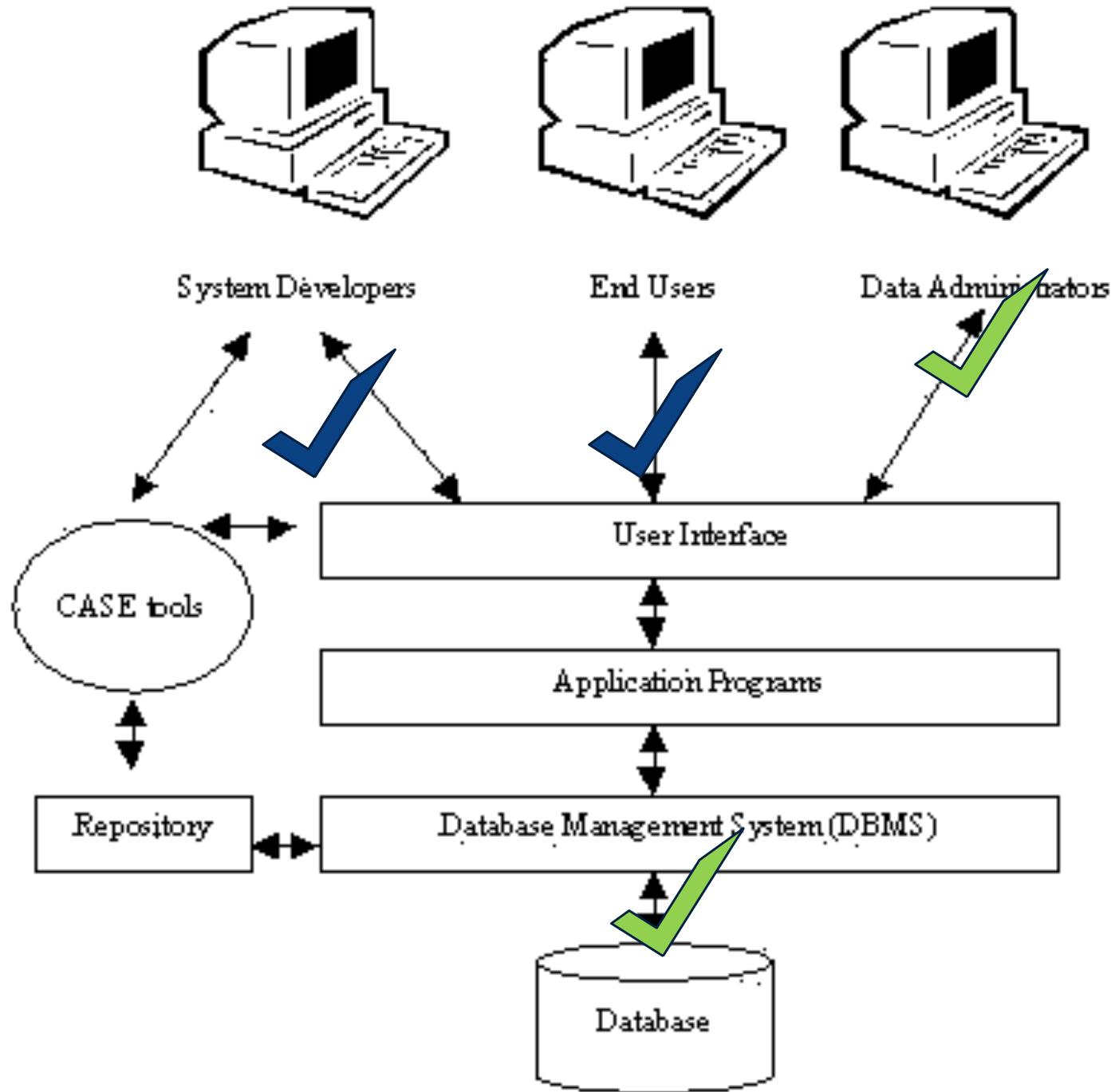


Subject Content Ideas

We will cover:

- Knowledge of how DB systems work at the architectural level
- Essentials to achieve correct behaviour and the best possible performance
- Mechanisms used by current systems to provide many useful features
- Techniques for achieving reliability and good concurrency among other things

Visually We will look at ?





But logistics of the subject first...

Semester 1, 2023

In class delivery with interactive questions/answers during lectures as well as...

live streaming of lectures + online recordings (forward questions to LMS forums if not in class during lectures)

Most tutorials are allocated to be face to face and if you are onshore please go to those

Some tutorials are available for offshore students via zoom

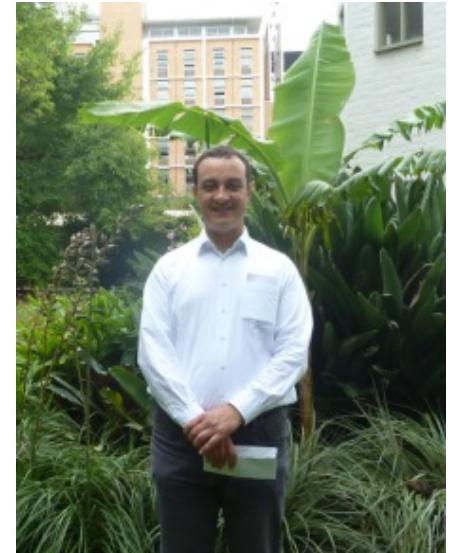


Who we are and Contact Info/Mode

Lecturer and subject coordinator

Prof Egemen Tanin (etanin@unimelb.edu.au)

(tutors will introduce themselves to you in the tutorials...)



**Note: DISCUSSION
FORUMS and LMS
ANNOUNCEMENTS
BEFORE EMAILS!**



...logistics of the subject first...

Lectures

2 x 1 hour lectures per week for 12 weeks , except holidays and the non-teaching break which is in the middle of the semester (University calendar for the break and holiday dates can be found from the uni website)

Weekly location and times of lectures are available from **LMS as a link to the timetable**



Logistic Contd.

Tutorials

One 1-hour tutorial/workshop per week per student

- Each tutorial will be **lead by a tutor**
- There are many to choose from per week, if some you check is full, choose another but make sure to enrol to your tutes
- They **start in week 2**
- Tutorials are all **active sessions**, face to face or zoom sessions, you should speak and interact with tutors frequently
- They are discussion places and will not be recorded
- Tute enrolment data is crucial for us in marking, management, etc
- Each student has to **attend the same tute every week**



Books

Main reference: Database System Concepts, 7th edition, by Silberschatz et al.

We will also reference other books, e.g. Transaction Processing, Jim Gray and Andreas Reuter, Morgan Kaufmann, 1992

I will highlight sources per topic that we cover...

Note 1: Please try and get a copy of the main reference, it is a good addition to your library,

Note 2: We will not read any book cover to cover as the size of these books are beyond one subject, so slides are the key in this subject...



Logistic Information Contd.

By the way, all the key information is available on LMS :

- You need to check it a few times a week for being up to date on discussions and announcements
- It has also other basic info such as contact info for us
- Find project specs and other assessment info
- Where submissions are made
- Where you can find lecture capture and other materials
-
- Lets visit it for a minute now actually!**



Assessments

- **Online quizzes** – 5 quizzes worth 2% each (spread evenly through out the semester, will be announced one week before the quiz, i.e., if/when we cover enough new materials to have questions from), expect the first quiz by week 3...
- A genuine effort to attend the classes and tutorials should be adequate to have a good mark in quizzes, only briefly prepare before each quiz using tutorial questions (but mainly expect multiple choice questions)
- **Project** – A hot Database topic survey report and oral presentation (40%) and done as a group of 4 students (will be discussed in detail shortly...)
- **Final examination** (50%), similar to tutorial questions (but less multiple choice and mainly classical questions)



Project

A survey report and oral presentation on a database research topic (40%) (presentation 15% + Report 25%)

The project is group based. A group has **4 members:**

- As some are all around the world we will have time to group formation, no reason to panic
- You can choose your own group but adhere to announcements, deadlines, and rules
- No need for all team members to be on the same tutorial due to dual delivery nature of the subject
- But we can say that many teams are commonly formed in the tutes



Project Info Contd

Project Specs is already on LMS! Read this carefully asap by yourself.

All members of a group should contribute to all aspects of the project. If there is significant difference among group member contributions, we reserve the right to differentiate marks among members and take other misconduct action accordingly...

Lets briefly visit the project specs online now!



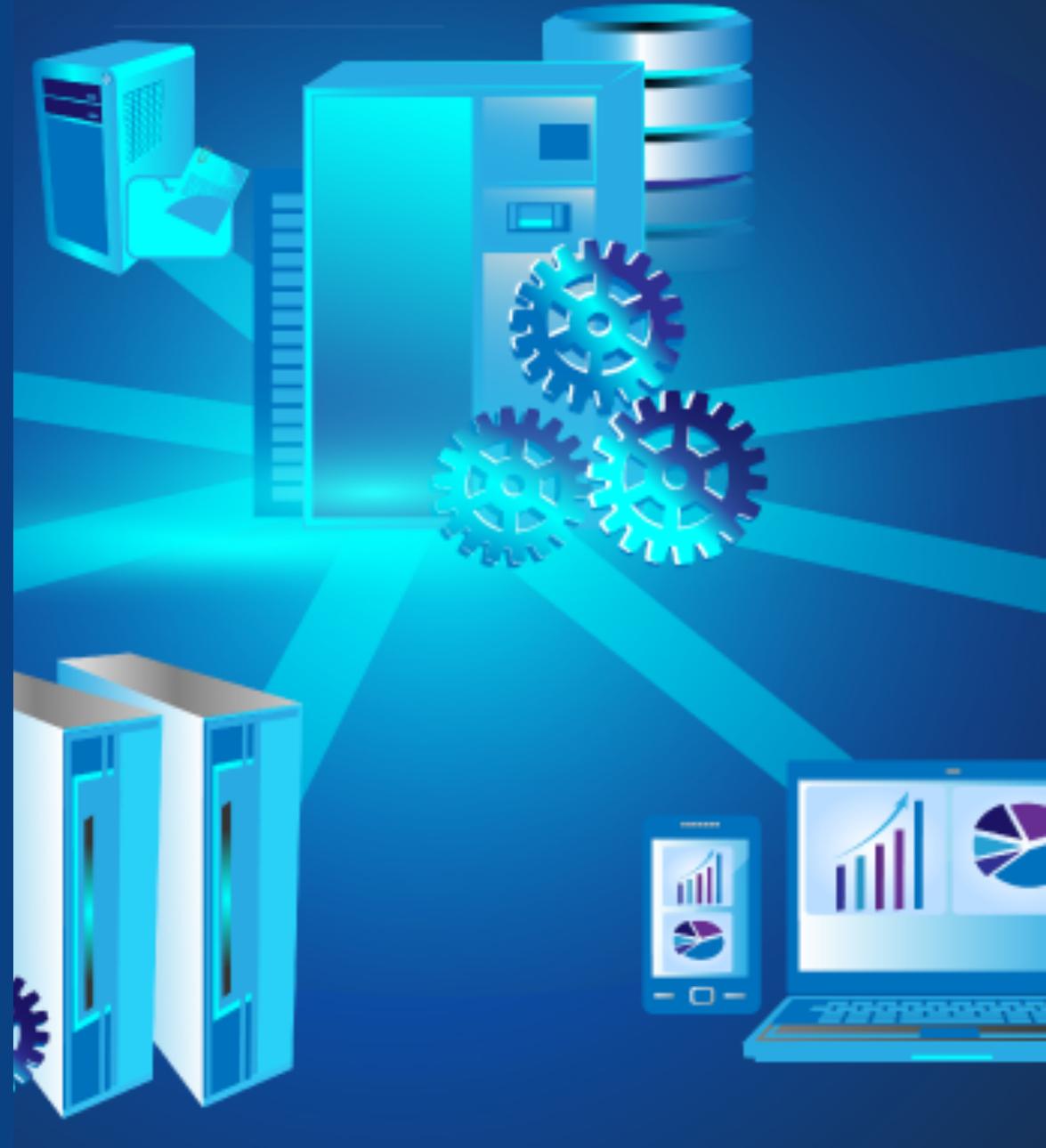
THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

Getting
Started

Lecturer: Prof Egemen Tanin

COMP90050 Advanced Database Systems





Where to start studying Advanced Databases

- ?] Recall what you already know first:
 - ?] You have seen how to organize data for the most popular DBMS type
 - ?] Relational Databases: for these you know creation of tables
 - ?] You have run queries and populated data into tables as well
 - ?] Thus you know how to run SQL queries
 - ?] Databases basically created a nice and tidy way to deal with data for you already



Where to start studying Advanced Databases

- ? Now it is time to see under the hood:
- ? The cost of structure and tidiness came with a complexity associated with accessing data efficiently, this is under the hood, e.g., not putting everything in one file means a lot of join operations
- ? If one wants to create a generic system to store and query large data in a tidy way, then one needs to forget about specialized fast programs and files but still one needs to compete in terms of efficiency by inventing novel methods
- ? This means database researchers have invested a lot of time creating under the hood techniques for performance
- ? Understanding the key costs and considerations in the whole process is a good point to start looking at advanced topics

Performance of a database system comes from

Hardware

- The speed of the processor
- Number of processors
- Number of drives and I/O bandwidth
- Size of main memory
- Communication network
- Type of architecture

Software

- Type and details of database technology used for a given application

Database tuning, crash recovery

- Indexing parameters
- Data duplication
- Sharing data
- ...



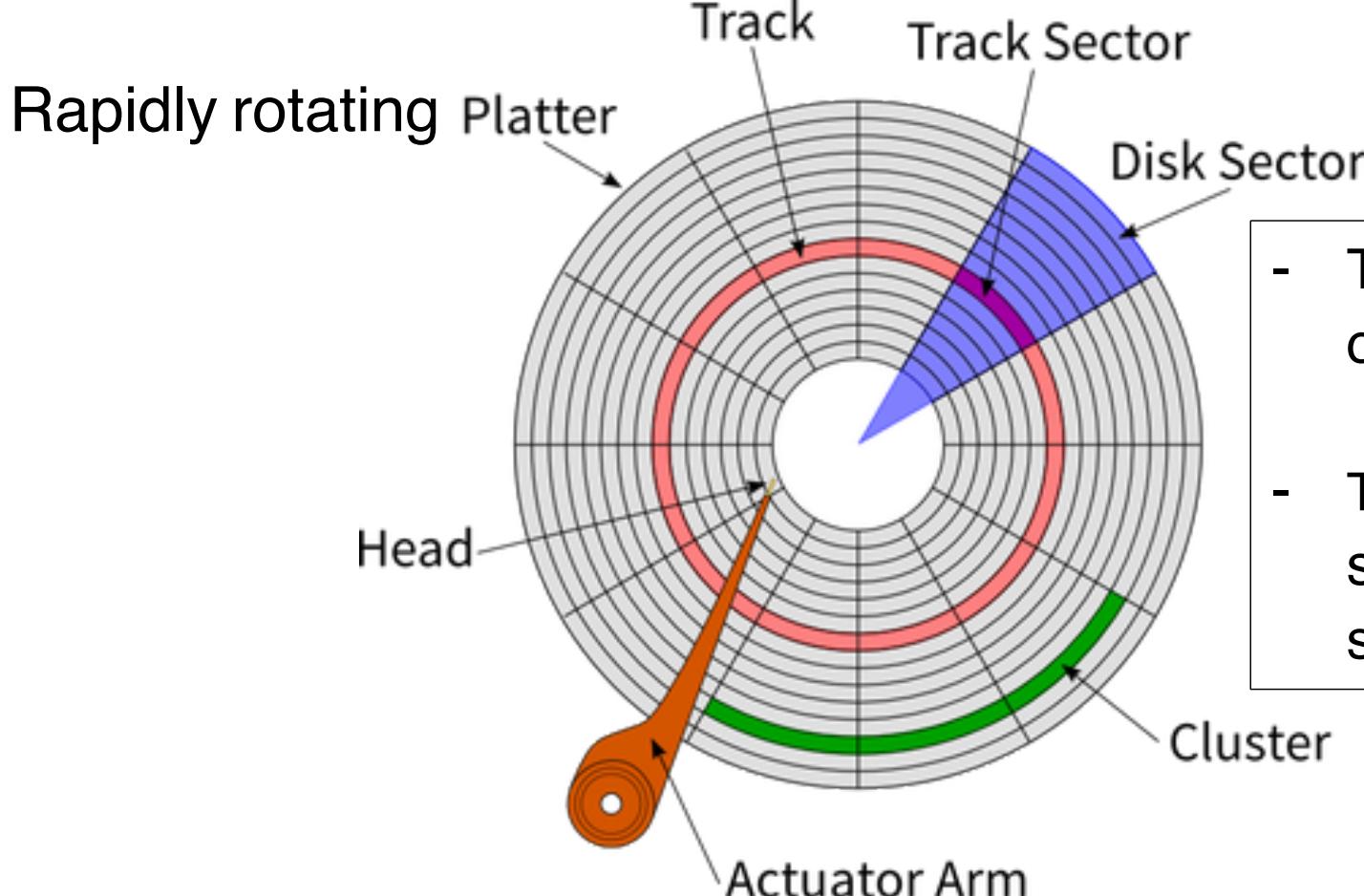
Among these, one item became the dominant one, effecting most design decisions till today

How to access large stored data fast?

You will find
this picture
even on the
cover of DB
books!



Disks and how they work became fundamental to DBMS design



- Tracks: circular path on disk surface
- Tracks are subdivided into disk sectors

...with magnetic head, which reads and writes data to the platter surfaces...



Modelling Disk Access was the Key

What is the Disk access time for a transfer size of 4KB, when average seek time is 12 ms, rotation delay 4 ms, transfer rate 4MB/sec?

... and **seek time here became the unwanted component** that many algorithms tried to minimize...

E.g., if joining two tables, **minimize no of disk seeks** and pick a join algorithm accordingly...

New disks...new challenges: **SSD (Solid-State Drive/Solid-State Disk)**

- **No moving parts** like Hard Disk Drive (HDD)
- Silicon rather than magnetic materials
- No seek/rotational latency
- No start-up times like HDD
- Runs silently
- Random access of typically under 100 micro-seconds compared 2000 - 3000 micro-seconds for HDD
- Relatively very expensive, thus did not dominate at all fronts yet... but soon will be... nevertheless did not effect DBMS when they were born
- Certain read/write limitations plagued it for years

Disk access time =



SSD specification example with realistic figures...

Samsung 860 PRO SATA III 2.5-inch

Capacity: 4TB SSD

Price: Many hundreds of dollars

Weight: < 62 grams

Bandwidth Performance (SATA Standard Serial)

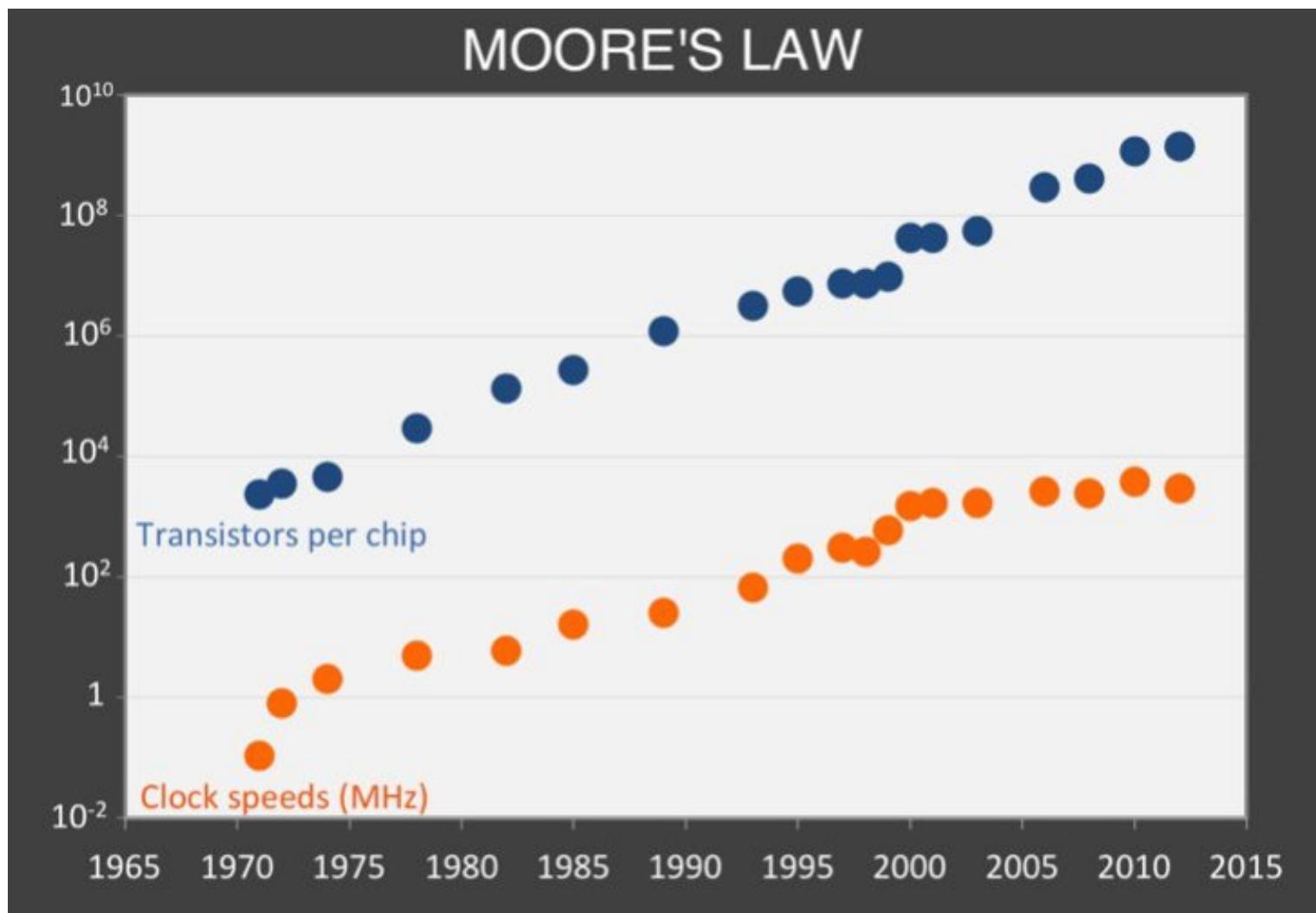
- Sustained Sequential Read: up to 560 MB/s
- Sustained Sequential Write: up to 530MB/s

Read and Write IOPS (Input/Output Per Second) – QD32

- Random 4 KB Reads: Up to 100,000 IOPS
- Random 4 KB Writes: Up to 90,000 IOPS

Other Hardware Considerations came about in recent years...

Observations on historical trends on chips and clock speeds:





Basically: are we going into the age of CPU clocks dominating design decisions?

- Moore's law: memory chip capacity doubles every **18 months** since 1970
- Joy's law for processors: processor performance doubles **every two years** since 1984
- In general, in database system implementations, hardware limitations effect how things are done and this changes over decades, slowly but surely....



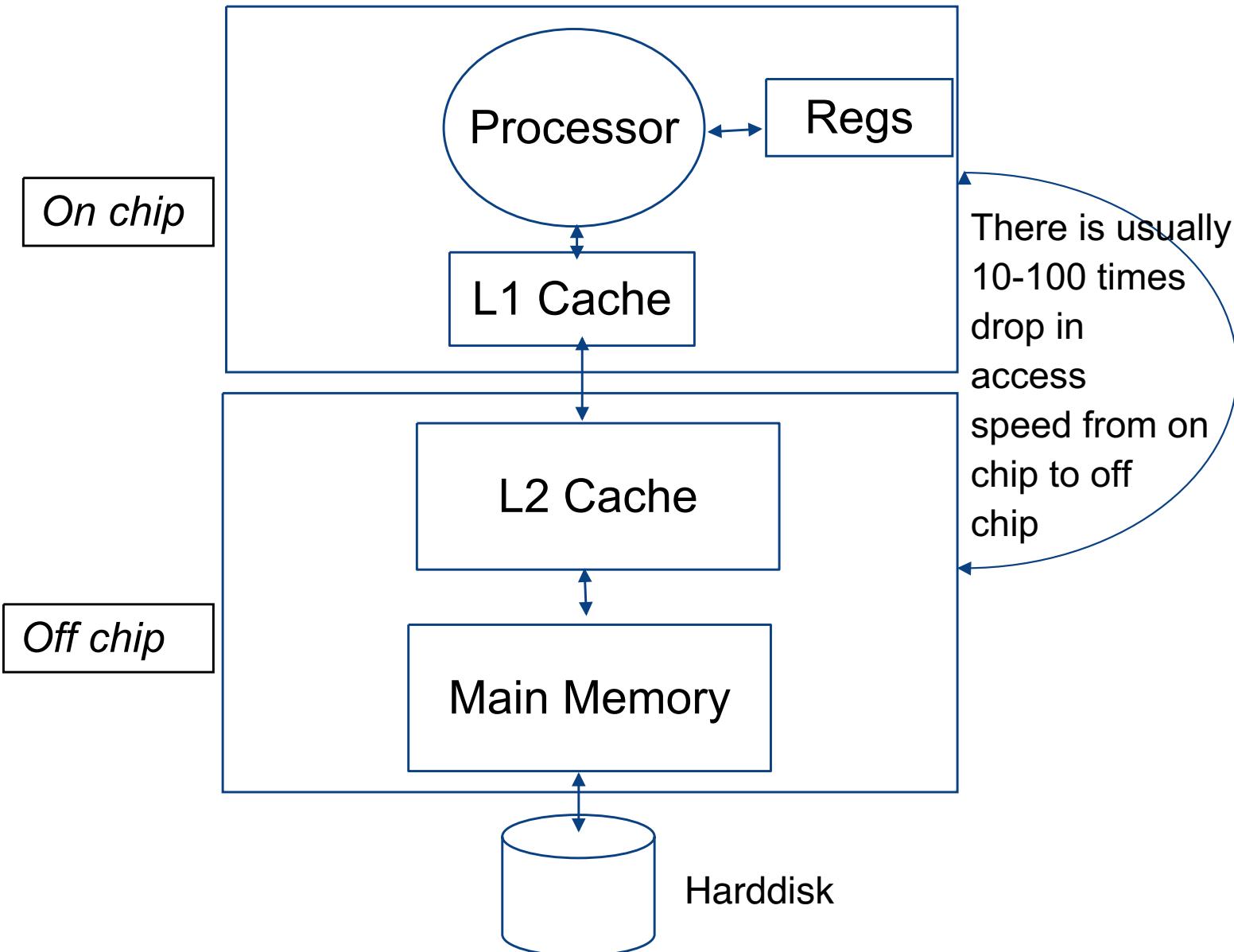
Some more hardware spec examples

- **IBM Summit** (2019) performs 200 Petaflops (200,000 trillion calculations/second).
- Summit approximately doubled the top speeds of **TaihuLight Supercomputer** (2018) in about a year.
- But **Blue Gene/P** performed 1 Petaflops (2^{50})/s using ~300,000 CPUs, about decade ago.
- Very soon we will be measuring the **performance by number of cores** as individual CPU is reaching its maximum clock speeds.
- Intel's Xeon Cascade Lake series can have up to 48 cores...

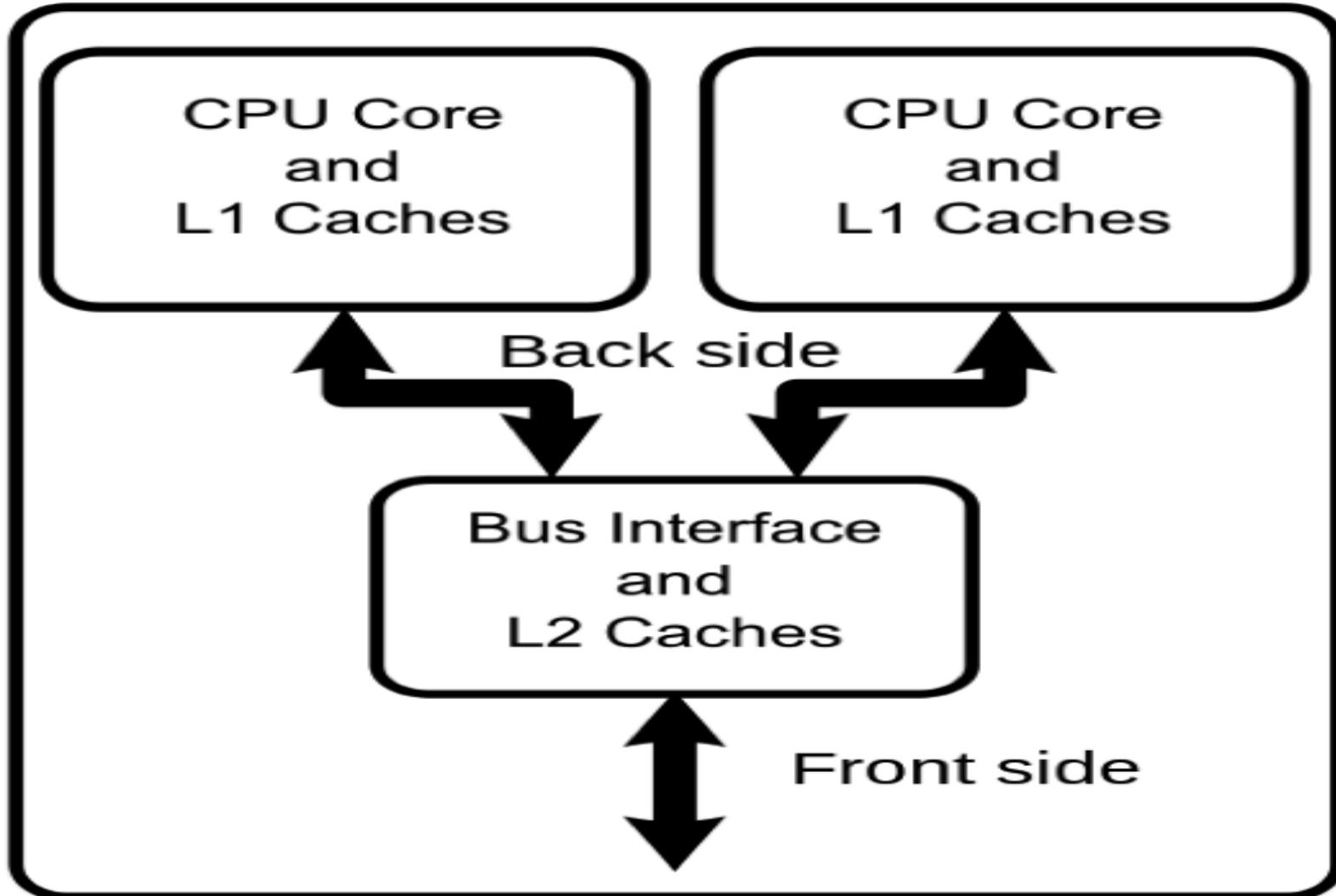
What does this all mean?

- **Harddisk was the foundation stone for many DBMS design choices** under the hood, and this applies even today
- In future more designs choices for databases will be made based on **SSDs and CPU problems** and these will likely dominate the arena eventually
- Another recent player is networking:
 - **More and more databases are distributed**
 - The network hardware speeds are at the speed of light already
 - The network router firmware/software forms the bottleneck: will they be upgraded every year?
 - Can this be another determining front for DBMS design choices
 - Some of these are already at play in various new systems!

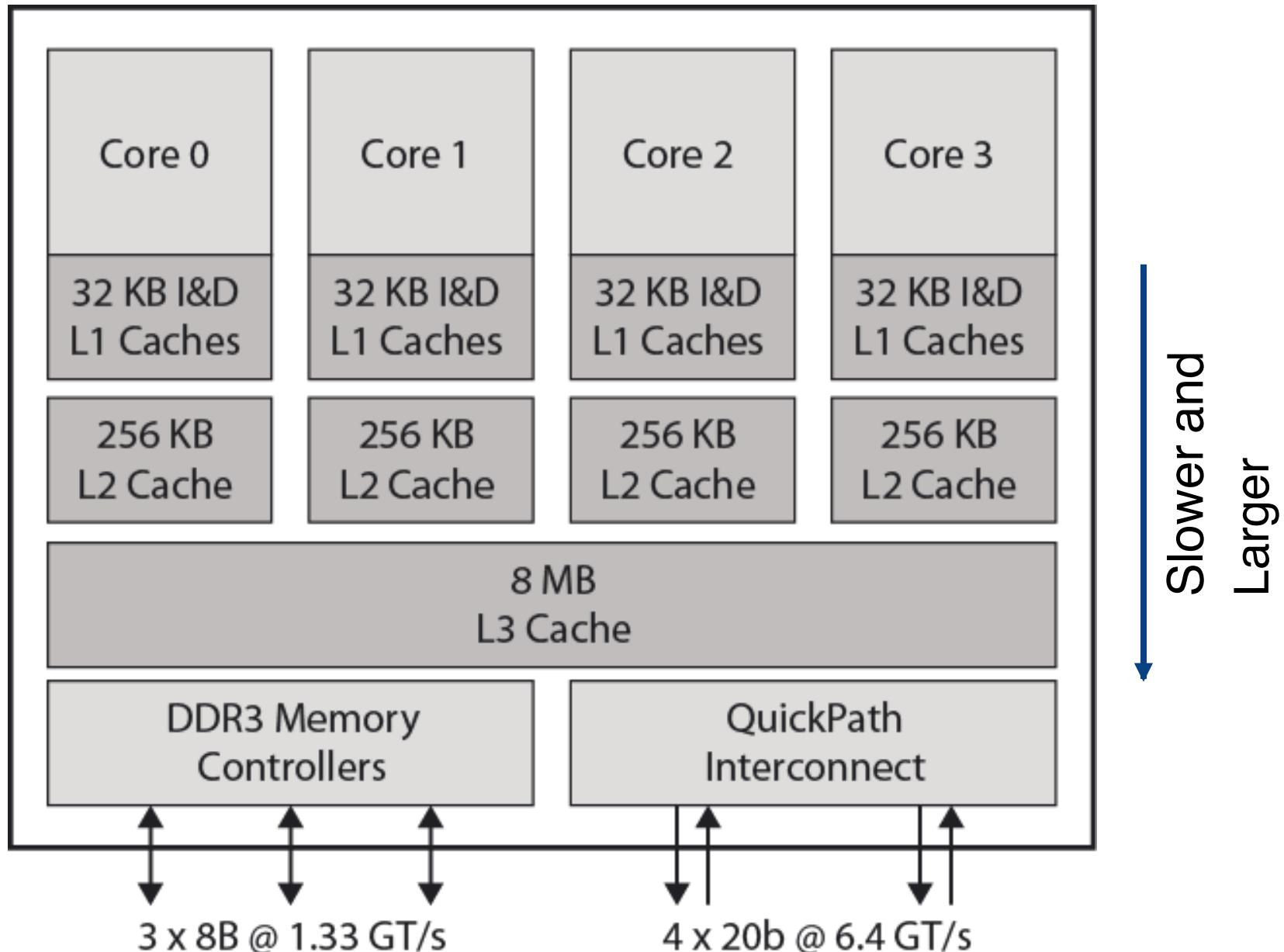
But where is data really: Just to complicate things it is in a Memory Hierarchy



Also There are Multi-Core Systems



Increasingly **L1, L2 and L3** caches are on the chip too!



Memory hierarchy leads to a more complicated model

Effective memory access time,

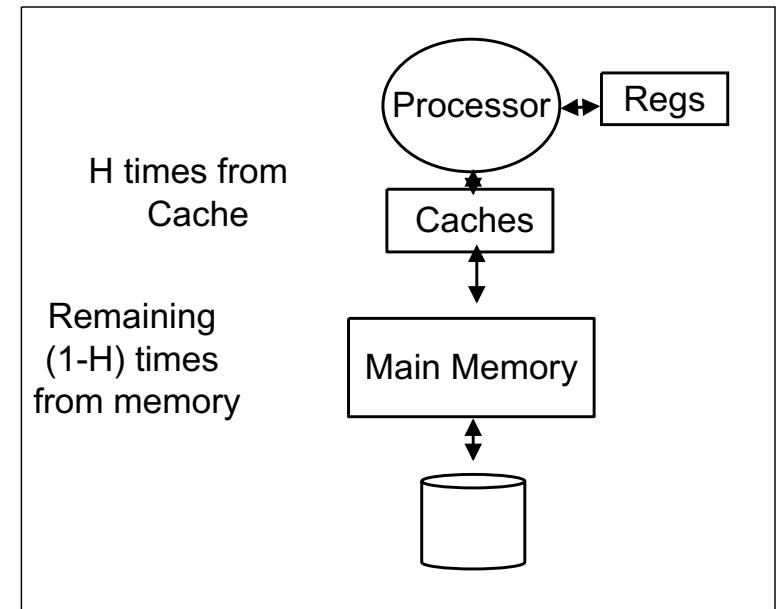
$$EA = H*C + (1-H)*M$$

where H = hit ratio,

C = cache access time;

M = memory access time

Hit ratio	Effective access time as multiple of C , $M = 100 C$
50.00%	50.5
90.00%	10.9
99.90%	1.1



Further more if we access disks...

Caching provided with HDD for access

Effective disk buffer access time,

$$EA = HB \cdot BC + (1 - HB) \cdot D \text{ where}$$

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

Hit ratio	
50.00%	500.5
99.00%	100.9
99.90%	1.999
99.99%	1.099

Memory hierarchy example



By Solomon203 - File:WD5000AAKX-221CA1_2012-07-07_2.jpg, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=30484513>



Note: some numbers to recall before doing storage computations

Metric	Value		Bytes
Byte (B)	1	2^0	1
Kilobyte (KB)	$1,024^1$	2^{10}	1,024
Megabyte (MB)	$1,024^2$	2^{20}	1,048,576
Gigabyte (GB)	$1,024^3$	2^{30}	1,073,741,824
Terabyte (TB)	$1,024^4$	2^{40}	1,099,511,627,776
Petabyte (PB)	$1,024^5$	2^{50}	1,125,899,906,842,624
Exabyte (EB)	$1,024^6$	2^{60}	1,152,921,504,606,846,976
Zettabyte (ZB)	$1,024^7$	2^{70}	1,180,591,620,717,411,303,424
Yottabyte (YB)	$1,024^8$	2^{80}	1,208,925,819,614,629,174,706,176

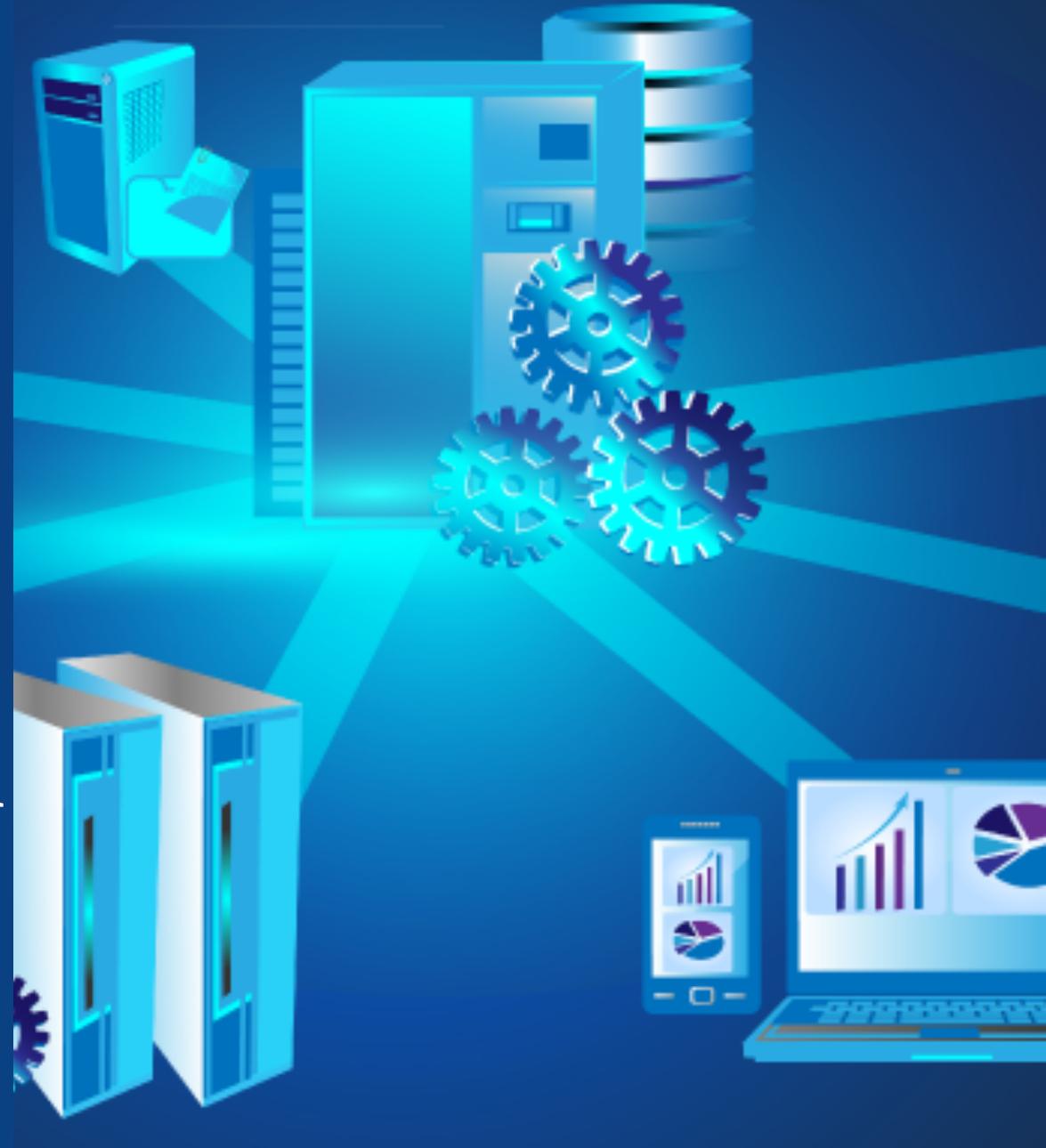


THE UNIVERSITY OF
MELBOURNE

COMP90050 Advanced Database Systems

Semester 1, 2023

Introduction Continues...

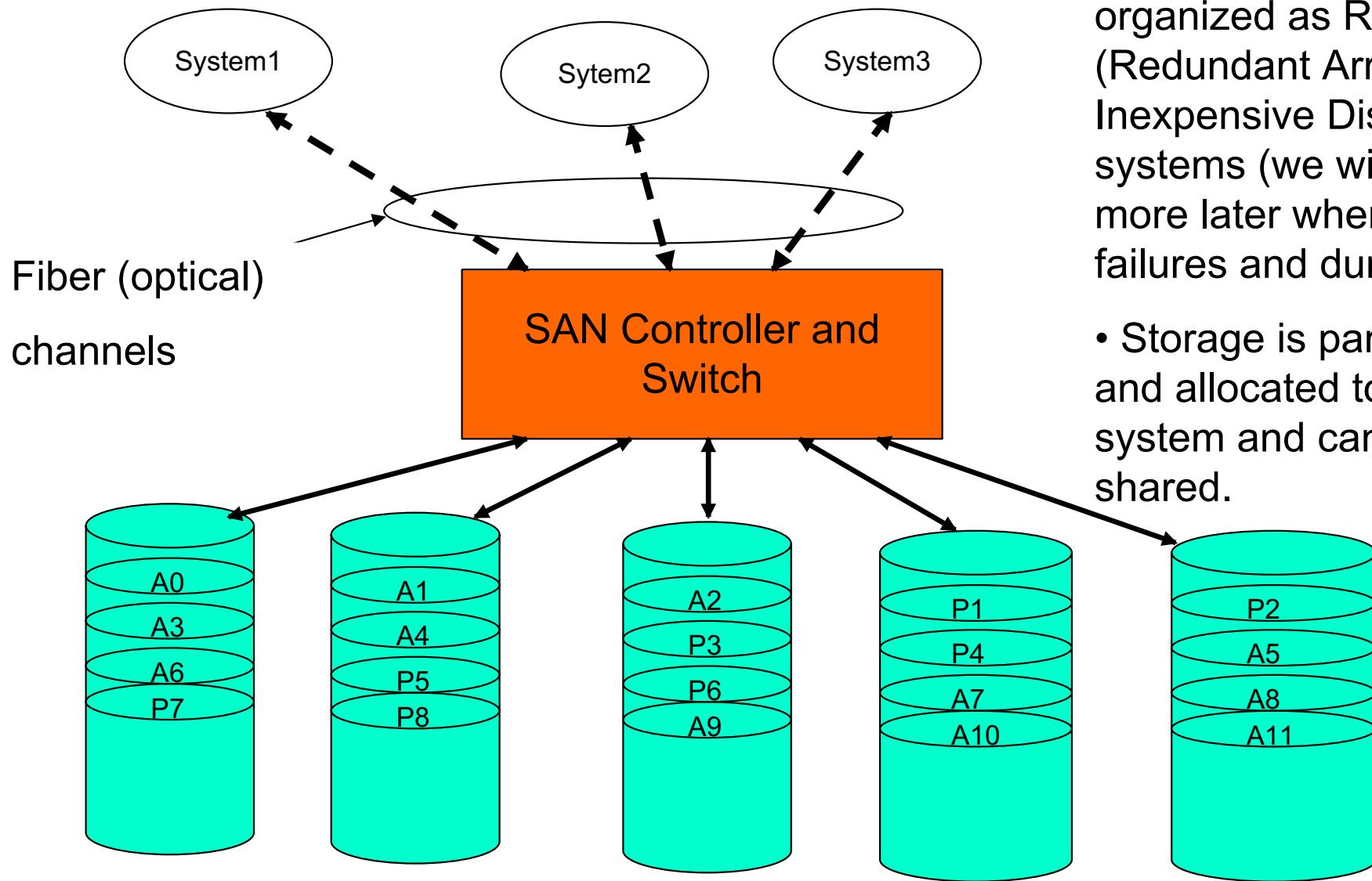


More on Storage

- Storage today **does not come as one disk** as well
- They come as a system and increasingly complex
- **Storage systems** can **determine the performance** and also **fault tolerance** of a Database
- In Database Management Systems (DBMSs), rarely data is stored in one/local location and rather available **over a network**
- Storage systems are now much larger, involves many disks, and accessed over a network, and at many sites

Storage Area Networks (SAN)

A dedicated network of storage devices



- Storage can be organized as RAID (Redundant Array of Inexpensive Disks) systems (we will see more later when we see failures and durability).
- Storage is partitioned and allocated to each system and can also be shared.



More on SANs

- They are used for shared-disk file systems
- They regularly also **allow for automated back up** functionality
- It **was the fundamental storage for data center** type systems with mainframes for decades
- Different versions evolved over time to allow for more data but fundamentals are the same even today
- They came with their own **networking capabilities**
- We visit a version of these when we see what can be done about failures
- In a nutshell **failure probability of one disk is different to 100s of disks** which requires design choices



There is also communication costs..

When things are stored afar then increasingly, another item to model is the cost of **data transfer/communication**

$$\text{transmit time} = \text{distance}/c + \text{message_bits}/\text{bandwidth}$$

c = speed of light (200 million meters/sec) with fibre optics

Today it is **hard to reduce hardware related latency** on contemporary systems further

The key cost in latency comes from software in networking

The DBMS has little control over that

So each **message length should be large** to achieve better link utilization while **keep the overall need to transfer data to a minimum**

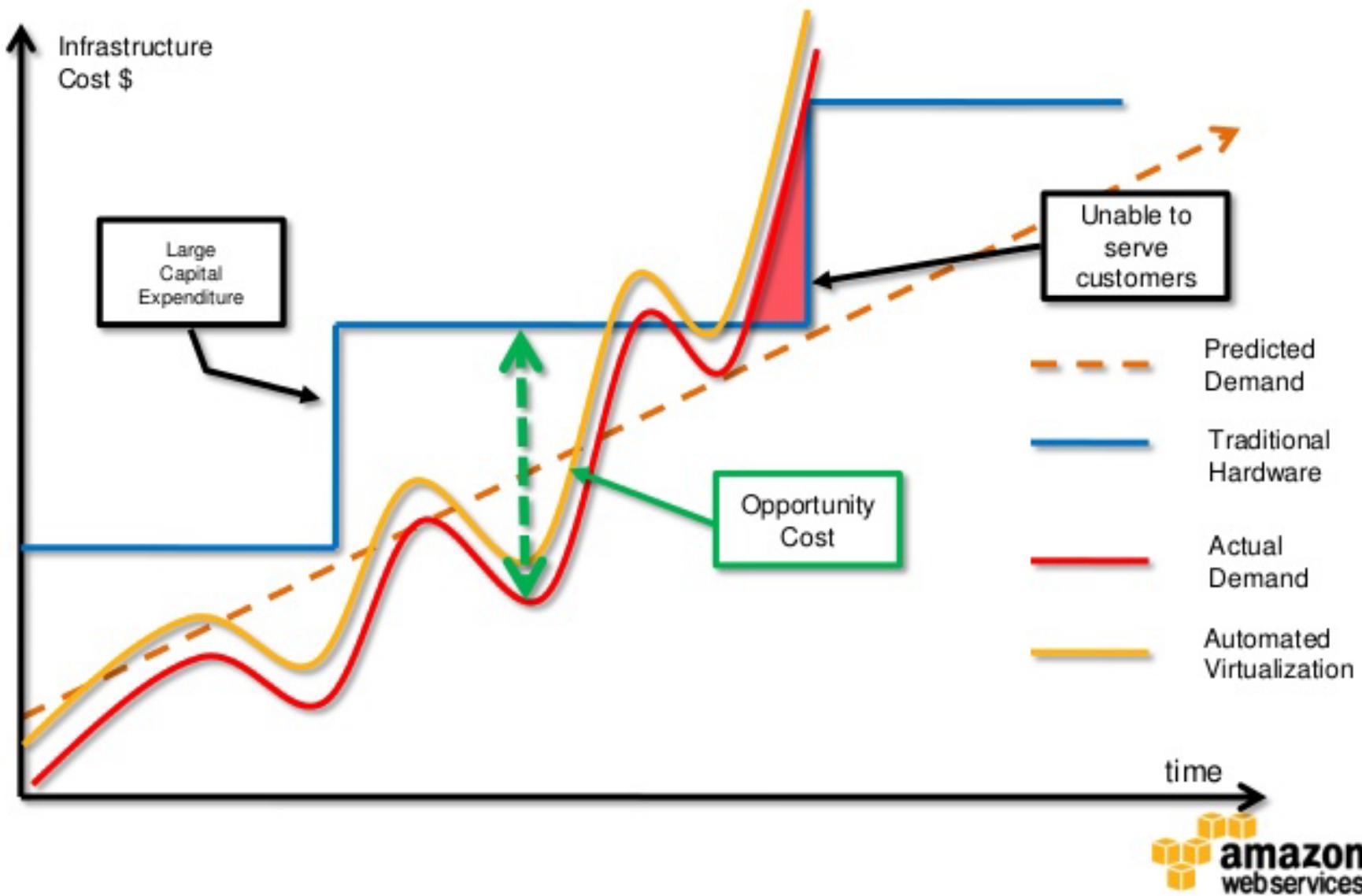
Another Cost to Consider As Well: \$\$

Attributes of Cloud Computing

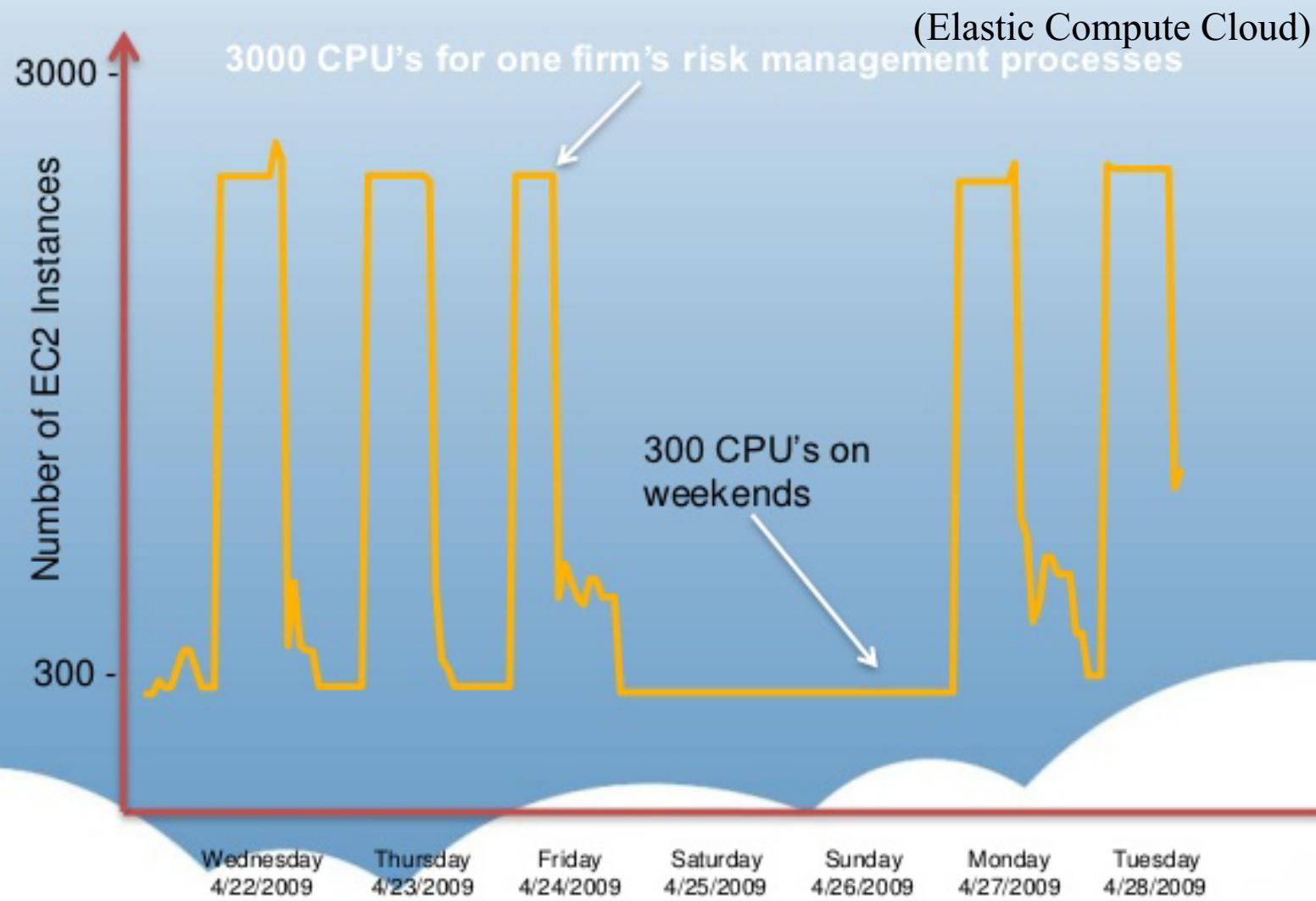
- 💡 No capital expenditure
- 💡 Pay as you go and pay only for what you use
- 💡 True **elastic capacity**; Scale up and down
- 💡 Improves time to market
- 💡 You get to focus your engineering resources on what differentiates you vs. managing the undifferentiated infrastructure resources



Elastic and Pay-Per-Use Infrastructure



Example: Wall Street App on Amazon EC2



Key DBMS Types

First: Considering how the data are stored

Simple file(s)

- As a plain text file. Each line holds one record, with fields separated by delimiters (e.g., commas or tabs).

Old but
Specialized DBs
do this still

RDBS

- As a collection of tables (relations) consisting of rows and column. A primary key is used to uniquely identify each row.

90% of DBs
today

Object oriented

- Data stored in the form of ‘objects’ directly (like OOP).

Idealistic

No-SQL

- Non relational – database modelled other than the tabular relations. Covers a wide range of emerging databases.

Getting
attention

Simple file(s)

- Usually very fast for simple (or sometimes specialized) applications but can be slow for “complex” applications
- Can be less reliable
- Application dependent optimisation
- Very hard to maintain them
- Concurrency problems
- Many of the required features (that exist in relational databases) need to be incorporated - unnecessary code development and potential increase in unreliability
- Eventually were left behind for most business cases of today
- Some specialized cases, e.g., scientific data, may use them

Relational DB systems

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

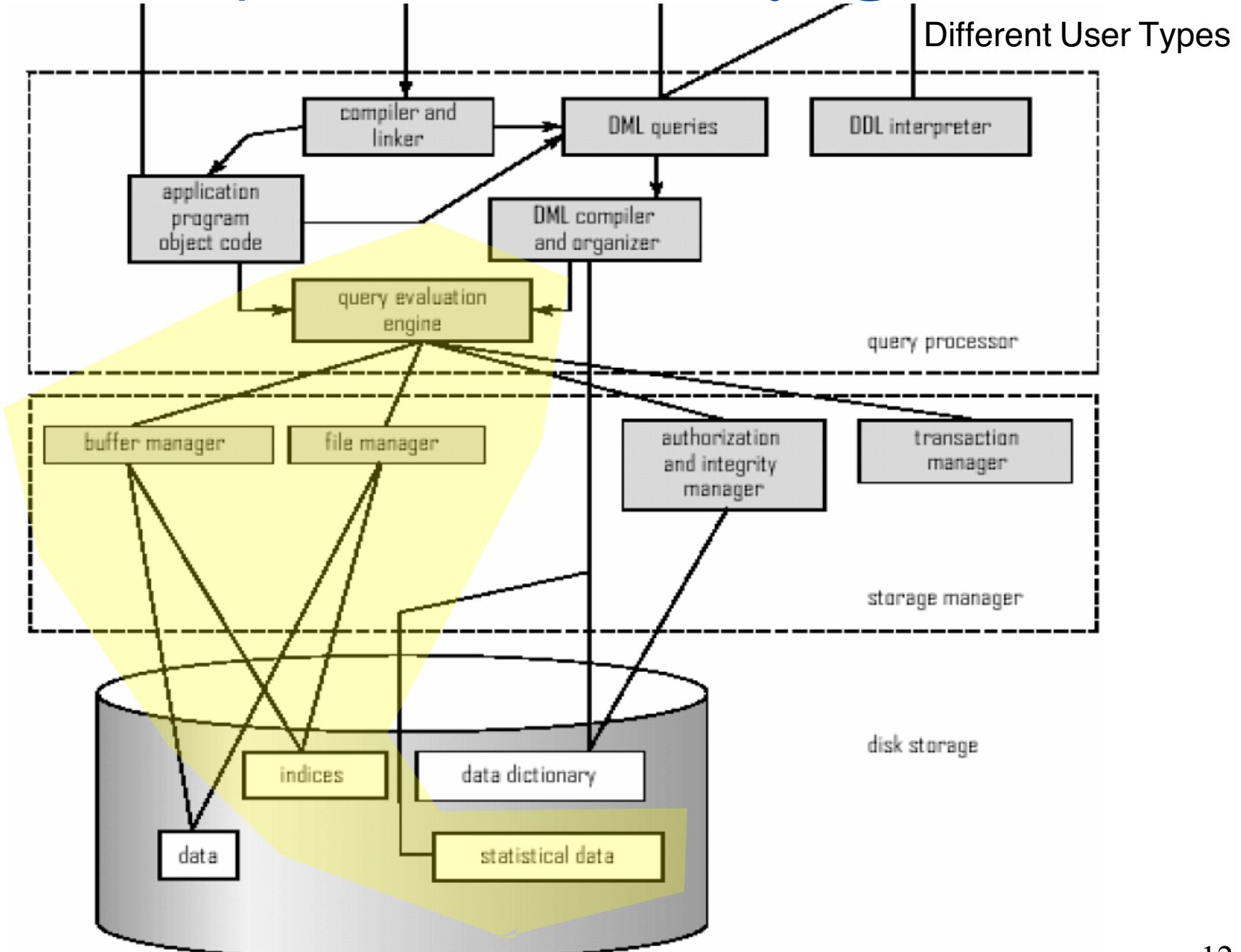
Activities Table

ID*	Activity*	Cost
084	Swimming	\$17
084	Tennis	\$36
100	Squash	\$40
100	Swimming	\$17
182	Tennis	\$36
219	Golf	\$47
219	Swimming	\$15
219	Squash	\$40

Source: <http://www.databasedev.co.uk>

- Very **reliable** in terms of consistency of data
- **Application independent optimisation**, so relatively fast already
- **Well suited to more and more applications** , increasingly very fast due to large main memory machines and increasing use of SSDs
- Some RDB also support Object Oriented model e.g., Oracle, DB2, and XML data+queries
- Can be slow for some special applications still

A RDB Architecture Shows why access/data can be tidy+generic+fast





We spend a decent no of weeks on Transaction Processing at RDBs

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

Structure of a transaction:

```
begin_Transaction()  
....  
<Sequence of operations to be performed>  
....  
if (successful)  
    commit_Trans() % Any changes made durable  
else rollback()      % Any changes made are all undone  
end_Transaction()
```



Transactions Rely on...

ACID properties

Atomicity: A transaction's changes to the state (Database) are atomic implying either all actions happen or none happen.

Consistency: Transaction are a correct transformation of the state. Actions taken as a whole by a transaction do not violate the integrity of the state. That is we are assuming transactions are correct programs.

Isolation: Even when several transactions are executed simultaneously, it appears to each transaction T that others executed either happen before T or after T but not at the same time.

Concurrent execution of transactions should not violate consistency of data.

Durability: State changes committed by a transaction survive failures.



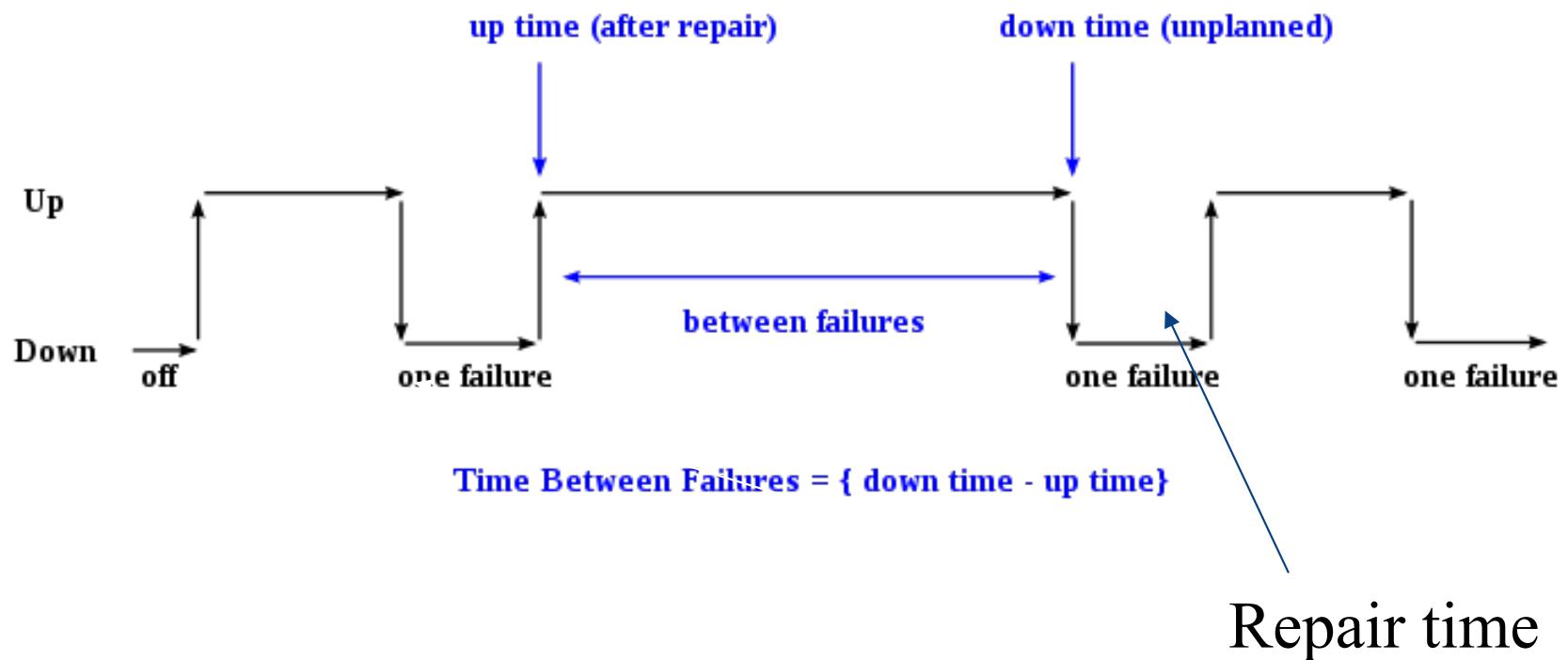
Transactions Contd...

Despite **adversarial conditions** DBMS should/will:

- Fast access to large amounts of data
- Provide a secure and stable repository when things fail
- Provides standard interfaces to data definition and manipulation with transactions
- Help multiuser accesses are done in an orderly manner but efficiently as well
- Allow convenient ways for report production and browsing
- Ease in loading data, archiving, performance tuning

...Unfortunately none of the ACID properties are trivial to achieve under adversarial conditions....!

First Reality Check: A Common System Lifecycle





A Common System Lifecycle Contd.

For example: HDDs have about 10 years to MTTF. If you have 360 HDDs in a SAN (assuming independent failures at uniformly random distribution), then one failure about every 10 days. Assuming it takes 1 day to replace the failed unit use the following:

Module availability: measures the ratio of service accomplishment to elapsed time

$$= \frac{\text{Mean time to failure}}{\text{Mean time to failure} + \text{mean time to repair}}$$

the **time** elapsing before a failure is experienced

10/11 or about 91% availability for your system. For a bank system your aim is to achieve levels like 99%+... How does DBMSs do that?

Another Reality Check: How to loose money and easily ?



Transaction A	Transaction B
<i>Initial value of b at bank is 150</i>	<i>Initial value of b at bank is 150</i>
	Start Transaction
Start Transaction	
	Balance = b.getBalance()
Balance = b.getBalance()	
b.setBalance(Balance + 100) commit	b.setBalance(Balance - 100) commit
<i>Final value of b at bank is 50</i>	<i>Final value of b at bank is 50</i>

Time



Back to How Data Stored Contd: Object Oriented (OO) DB Systems

- Stores as objects directly, not tables
- May contain both data (attributes) and methods – like OOP
- Can be slow for some applications
- Reliable
- Limited application independent optimisation
- Well suited for applications requiring complex data
- Unfortunately, many commercial systems started this way did not survive the force of RDB technology and basically disappeared from the market
- **A take away message: do not throwaway your RDBs yet, they appear to be very resilient!**



NoSQL (also called Not Only SQL)

- Flexible/ no fixed schema (unlike RDB)
- Provides a mechanism for storage and retrieval of data modelled in means other than the tabular relations, used with BigData
- Simple design, should linearly scale
- NoSQL has compromised consistency and allows replication - We will discuss this more later -> leads to supposedly faster exec
- Most NoSQL databases offer "eventual consistency", which might result in reading data from an older version, a problem known as stale reads – we will learn more on these later as well

Types of NoSQL databases

Key Value



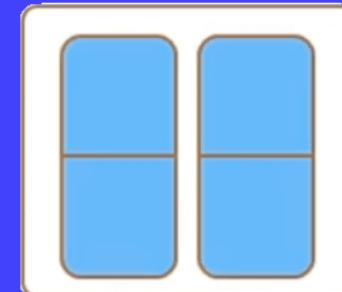
Example:
Riak, Tokyo Cabinet, Redis
server, Memcached,
Scalarmis

Document-Based



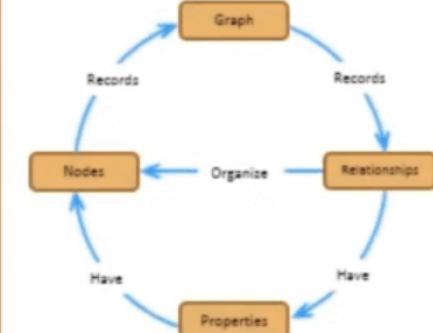
Example:
MongoDB, CouchDB,
OrientDB, RavenDB

Column-Based



Example:
BigTable, Cassandra,
Hbase,
Hypertable

Graph-Based



Example:
Neo4J, InfoGrid, Infinite
Graph, Flock DB

Image source: <https://jameskle.com/writes/no-sql>



Key-value pair database systems are becoming very popular

Stores data as a collection of key–value pairs, where each key is unique, each record many have different fields

Why useful: Many applications do not require the expressive functionality of transaction processing and rigidity – e.g. Web search, Big Data Analytics can use MapReduce technology

Used for building very fast, highly parallel processing of large data - Hadoop is another example here

Atomic updates at Key-value pair level (row update) only



Some other DB systems – Deductive database systems (DDBS)

- Allows recursion, similar to logic programming paradigm
 - Works on rules and facts and make “deductions”
 - Most of the application can be developed entirely using DDBS
-
- There are no big commercially available systems like RDBs
 - Many applications do not require the expressive power of these systems (e.g. many commerce related applications)
 - Many RDBs do provide some of the functionality – e.g. supporting transitive closure operation (a form of recursion in SQL2)



THE UNIVERSITY OF
MELBOURNE

COMP90050 Advanced Database Systems

Semester 1,
2023

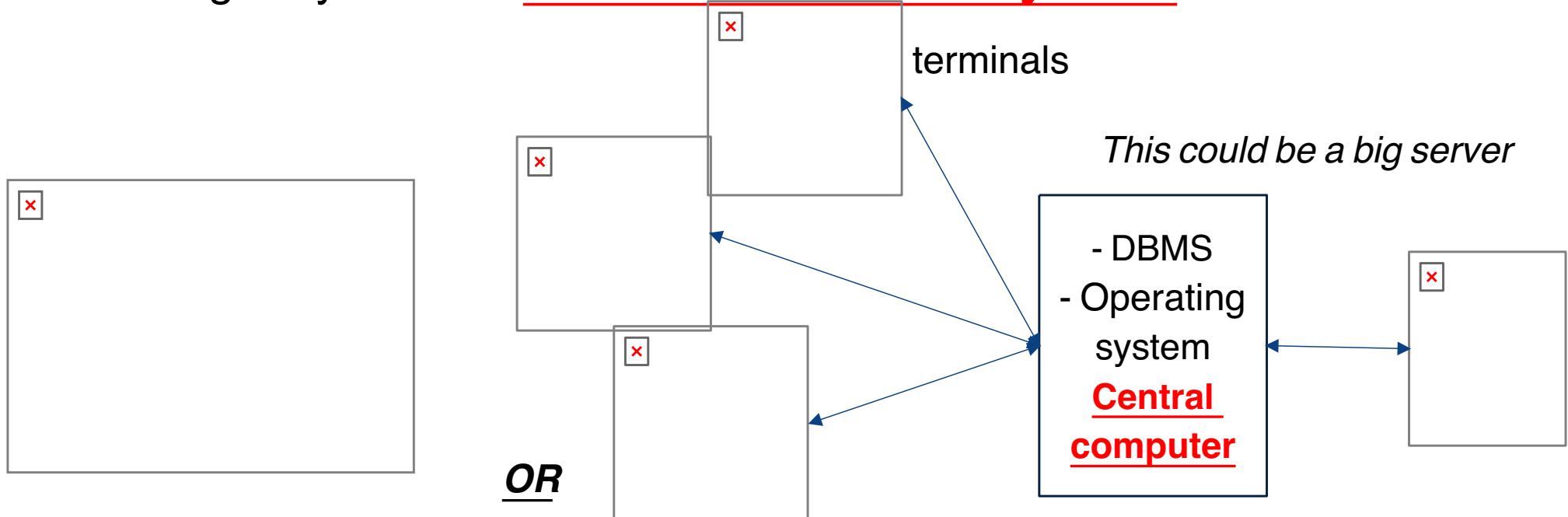
Wrapping up
Intro. and
Starting Part I of
ADBs

Another Taxonomy Worths Mentioning: Taxonomy based on Database Architectures

Centralized	Distributed	WWW	Grid	P2P	Cloud
<ul style="list-style-type: none">• Data stored in one location	<ul style="list-style-type: none">• Data distributed across several nodes, can be in different locations	<ul style="list-style-type: none">• Stored all over the world, several owners of the data, little organization	<ul style="list-style-type: none">• Similar to Distributed, but each node manages own resources but there is some structure	<ul style="list-style-type: none">• Like Grid, but nodes can join and leave network at will (unlike Grid)	<ul style="list-style-type: none">• Generalization of Grid, but resources are accessed on-demand, run by a company

Database Architectures Continued

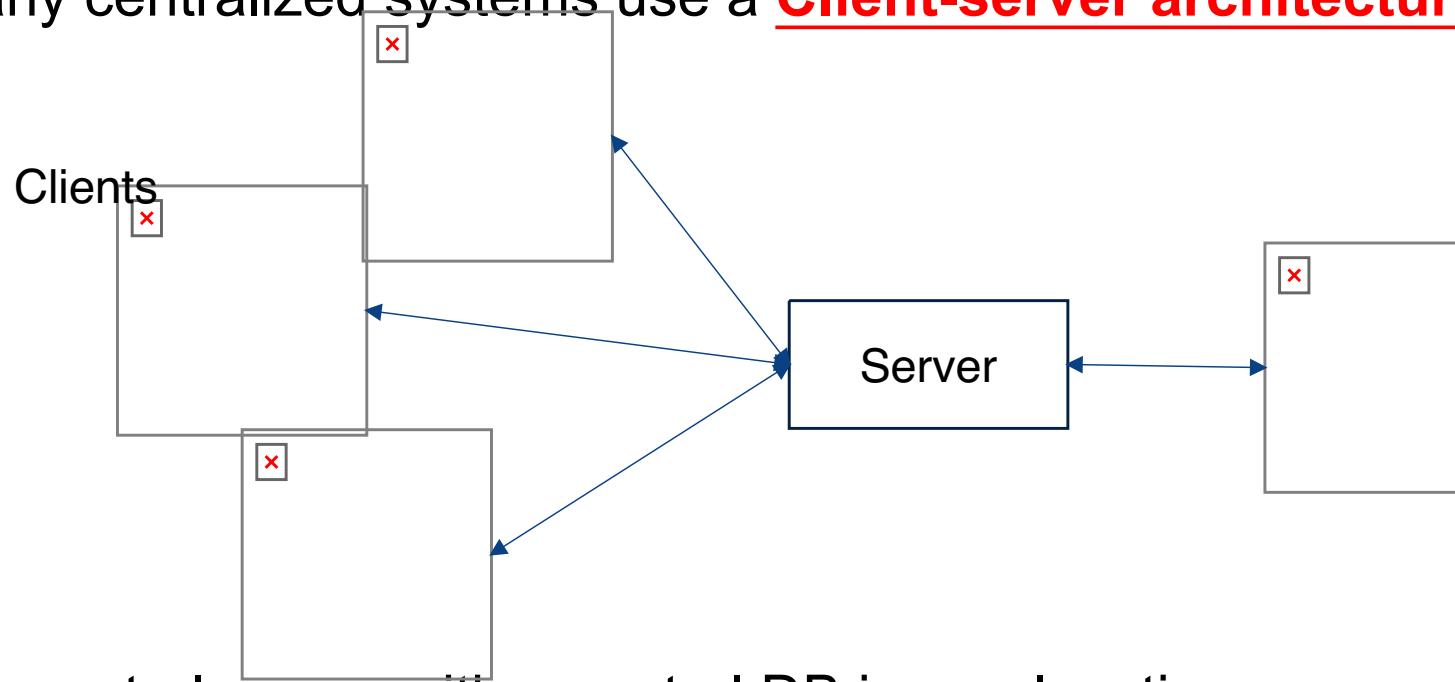
Originally we had: **Centralised database systems**



- Data in one location
- System administration is simple
- Optimisation process is generally very effective on simple transactions

Database Architectures

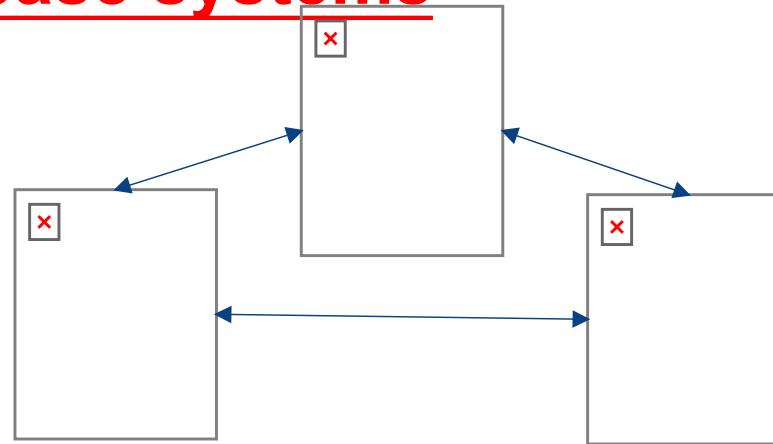
Many centralized systems use a **Client-server architecture** as well



- A central server with a central DB in one location (but client and server can be in different locations)
- Client generally provides user interfaces for input and output
- Server provides all the necessary database functionality
- System administration is relatively simple
- System recovery is basically same as previous case, i.e. simple

Database Architectures

Distributed database systems

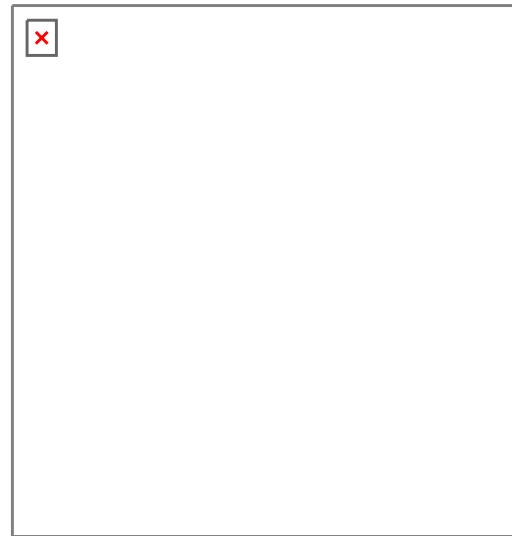


- Data is distributed across several nodes in different locations
- Nodes are connected by network
- System provides very advanced concurrency, recovery and other transaction processing capabilities
- System administration is very hard
- Crash recovery is complicated
- There is usually data replication – adds more inconsistency issues



Database Architectures

World Wide Web



- Data is stored in many many locations
- **Several owners of data** - no certainty of data availability or consistency
- Optimisation process is generally very ineffective
- **Dream database technology** -- no standards have been developed except in case of XML/http and some protocols for accessing data
- Security would be a problem
- Notions of transactions is much more difficult to enforce or non-existent



Database Architectures

Grid Computing and Databases

- Very similar to distributed database systems
- **Data and processing are shared among a group of computer systems** which may be geographically separated
- Usually designed for particular purpose – e.g., a scientific data management, and the grid is dedicated to that purpose
- **Administration of such systems are done locally by each owner of the system but there is some protocol/structure**
- Reliability, security etc of such systems are not well developed or studied
- *Grid systems became quickly outdated and overwritten by the Cloud Computing model*



Database Architectures

P2P Databases

- Data and processing is shared among **a group of computers** which are commonly geographically separated and form a large set
- **Computer nodes can join and leave the network at will** unlike in Grid Databases ↗ much harder to design **transaction models**
- Usually designed for particular usage – e.g. a scientific application, by citizen scientists
- Administration of such system is done by the owners of each PC
- An idea that keeps on coming back



Database Architectures

Cloud computing-based DB Systems

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 as they go** just like phone services, electricity, etc

Large potential with minimal infrastructure costs (this is a hot area of research from many angles and has taken off well)

Dynamicity of the resources is a challenge for transactional models

Cloud services are offered in several forms:

- **IaaS** – Infrastructure as a service (provide virtual machines)
- **PaaS** – Platform as a service (Provide environment like Linux, Windows, etc.)
- **SaaS** – Software as a service (also known as **DBaaS** where DBMS is there already)



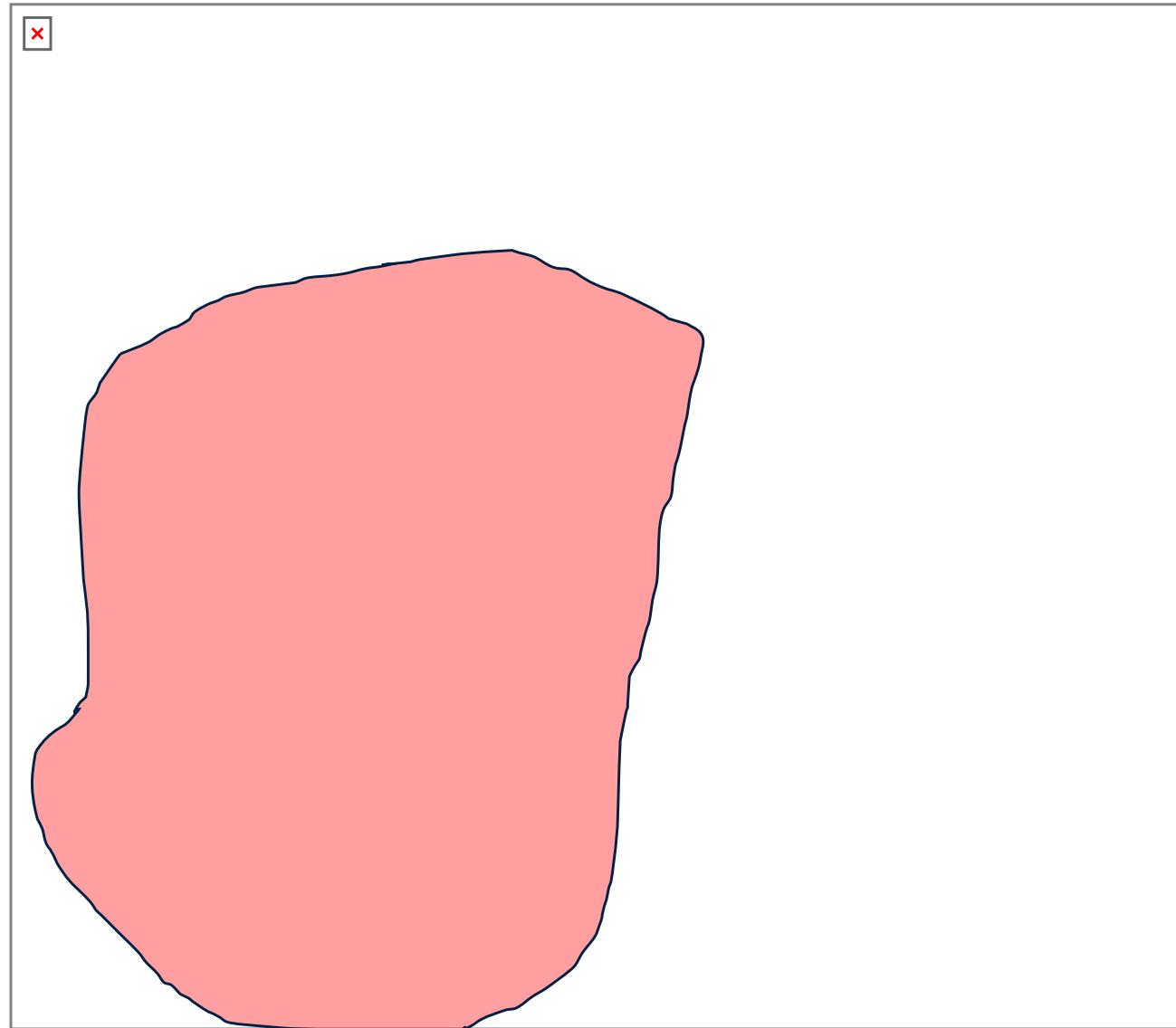
THE UNIVERSITY OF
MELBOURNE

COMP90050 Advanced Database Systems

Semester 1,
2023

Part I:
Performance

Performance and DBMS





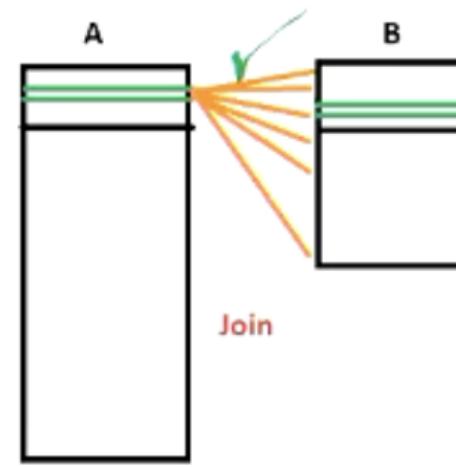
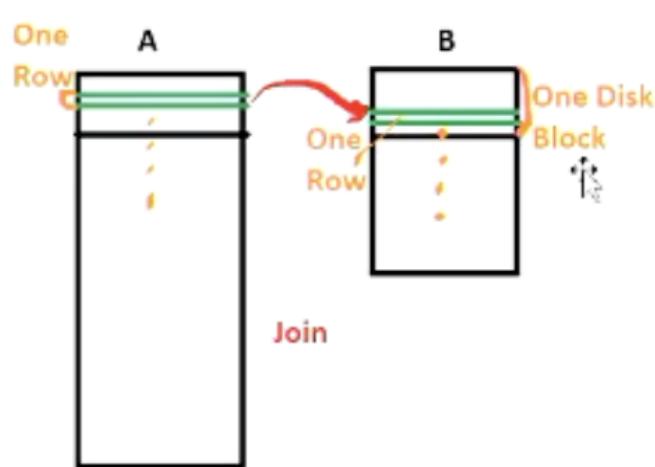
Query Processing Steps

✗

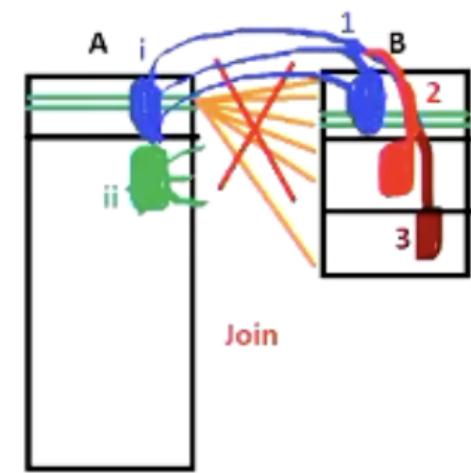
A Key Hurdle to get Fast Query Processing, Joins

Definition: $r \bowtie_{\theta} s$

- This operator is used to define a join of two tables on some condition, here r and s are tables and theta is a condition containing $<,>,=$ etc
- This is a very common operation due to the nature of RDBs
- But also a very expensive operation
- Key in getting good performance in RDBs



Option 1



Option 2



Joins Contd ? $r \bowtie ? s$

- SQL example of a join operation:

*Select * from T1 inner join T2 on T1.a = T2.b*

(DBMS converts this to relational algebra under the hood)

- A lot of investment gone into doing an efficient join operation on various data types
- It gives us a key example to see what RDBMSs do under the hood

Joins Continued

- Several different algorithms to implement joins exist that the optimizer can look at and pick the best
 - Nested-loop join
 - Block nested-loop join
 - Indexed nested-loop join
 - Merge-join
 - Hash-join
 - ...
- The same mentality exist for pretty much all query processing but we take joins as the key example now



Joins Contd.

- There is a choice on how to run a query on the server
- Choice is based on cost estimates:
 - statistics,
 - table sizes,
 - available indices
 - ...
- **Decisions effect performance dramatically**



A Simple Nested-Loop Join

- n To compute a theta 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  $\theta$  (??)
        if they do, add  $t_r \bullet t_s$  to the result.
    end
end
```



A Simple Nested-Loop Join Contd

- n r is called the **outer relation** and s the **inner relation** of the join.
- n Requires no indices and can be used with any kind of join condition.
- n **Expensive since it examines every pair of tuples** in the two relations.
- n Remember that for every retrieval, especially for a different item from the disk in a **nonconsecutive** location we pay a **seek time as a penalty**, this is where it could happen a large number of times.
- n Could be cheap if you do it on **two small tables** where they fit to main memory though (*disk brings the whole tables with first block access*).



Lets Calculate the Costs

Lets see an example with the following bank database information:

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

In the worst case, if there is enough memory only to hold one block of each table, the estimated cost is

$$n_r \lceil b_s + b_r$$

block transfers, and

$$n_r + b_r$$

seeks



So Two Options Are

With *depositor* as the outer relation:

- $\square\square\square\square\square\square$ $400 + 100 = 2,000,100$ block transfers,
- $5000 + 100 = 5100$ seeks

With *customer* as the outer relation:

- $\square\square\square\square\square\square$ $100 + 400 = 1,000,400$ block transfers and **10,400** seeks
- Remember the example from last lecture: **average seek time was 12 ms** which was the main cost, rotation delay 4 ms, transfer rate 4MB/sec
- This would mean: **$10,400 \times 12\text{ms seek time}$** which makes about **2 mins to join two small tables** (just seek time)
- If you had **1000,000 customers like a real bank then you would wait 3 hours for one simple join** (seek time only)!



A Better Way: Block Nested-Loop Join

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

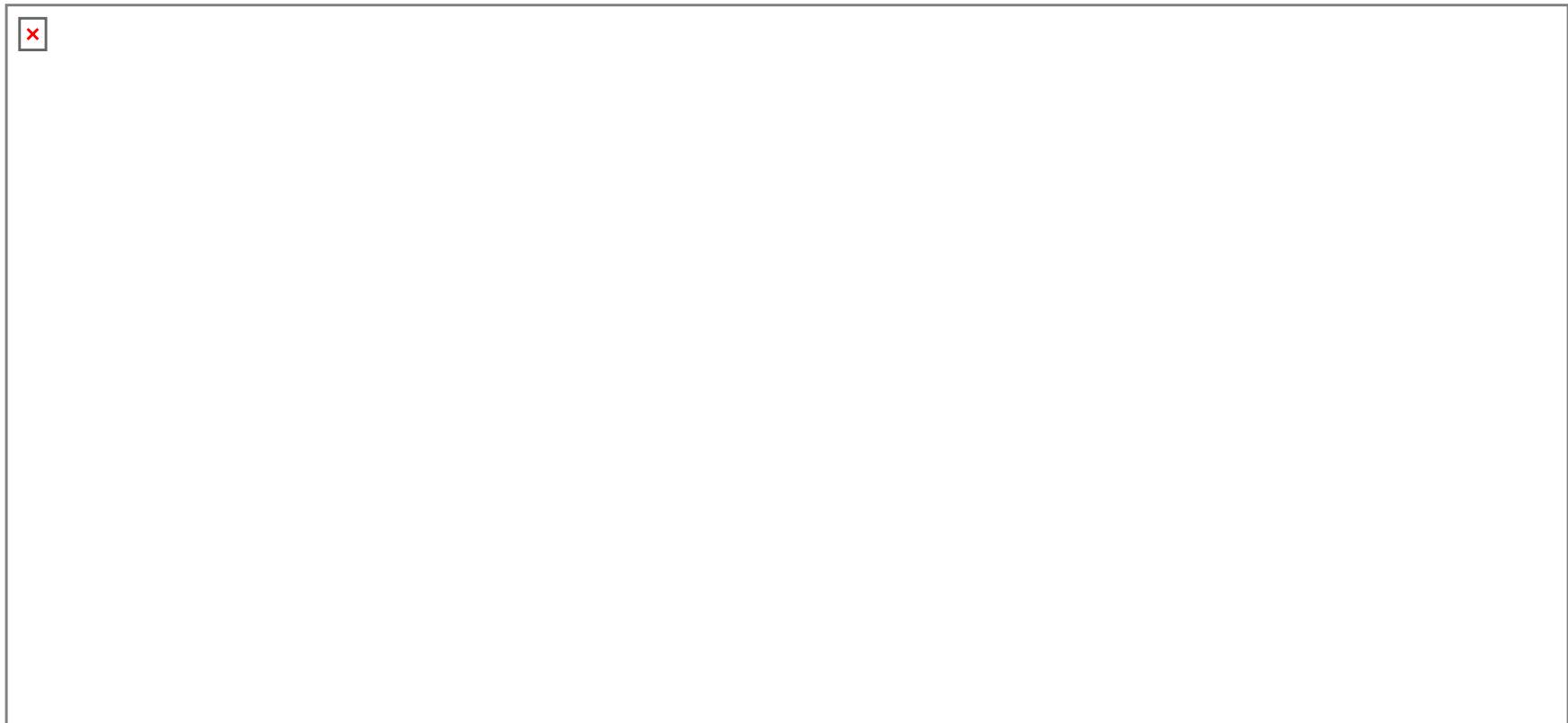
```
for each block  $B_r$  of  $r$  do begin
    for each block  $B_s$  of  $s$  do begin
        for each tuple  $t_r$  in  $B_r$  do begin
            for each tuple  $t_s$  in  $B_s$  do begin
                Check if  $(t_r, t_s)$  satisfy the join
                condition
                if they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

Lets Compute the Time Now

- n Estimate: $b_r \lceil b_s + b_r \rceil$ block transfers + $2 * b_r$ seeks
 - I 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)
- n Further improvements to nested loop and block nested loop algorithms exist but with the block nested loop we already get
 - I $400 * 100 + 400 = 40,400$ transfers and **800 seeks**
 - I **800 x 12ms is about 10 secs which is now an order of magnitude faster than the previous option**
- n Thus, researchers have invented many such methods to make things run faster and reduce in particular the seek time in DBMS



In general:
Query Optimization is about the right choices on a graph



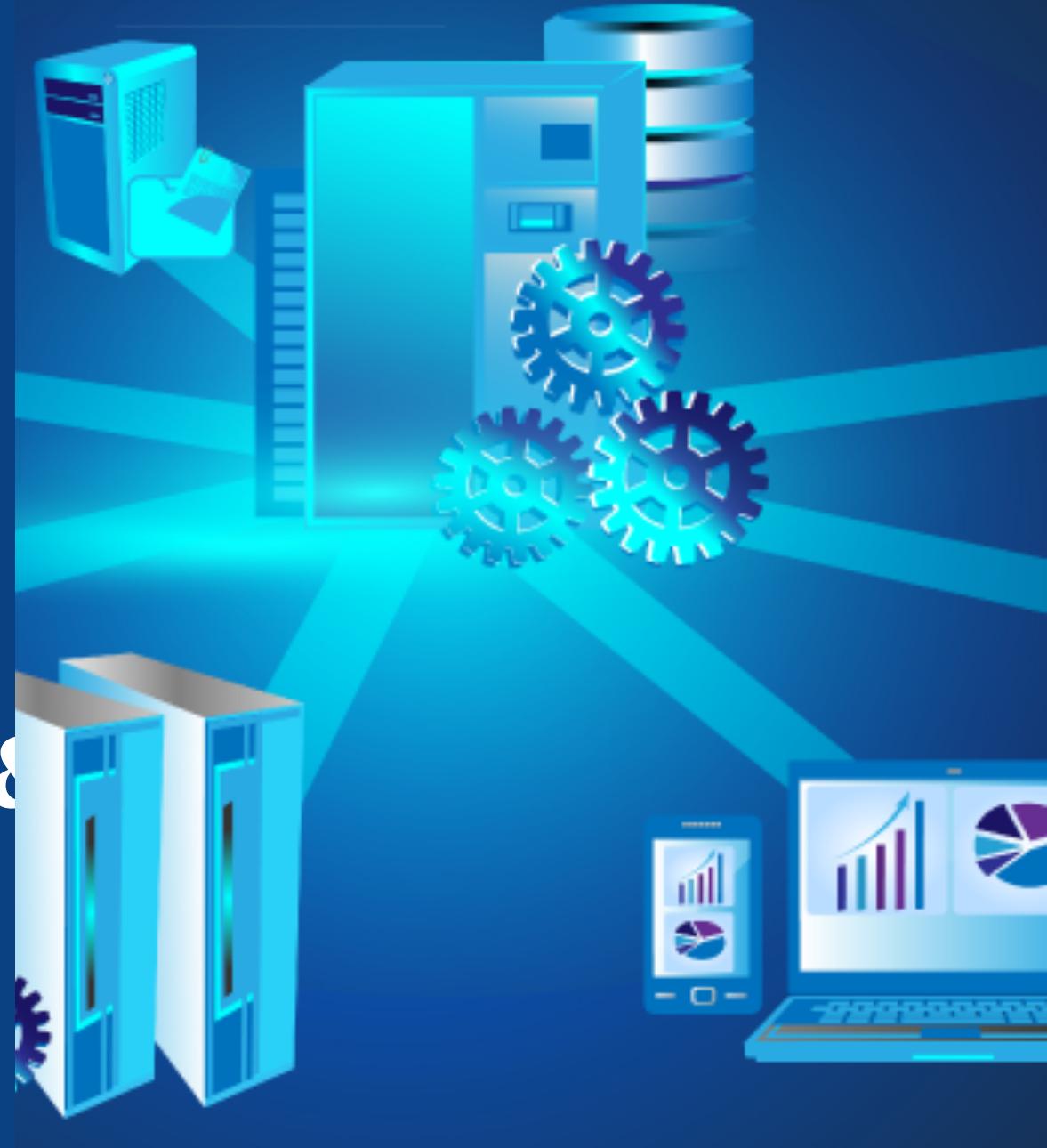


THE UNIVERSITY OF
MELBOURNE

COMP90050 Advanced Database Systems

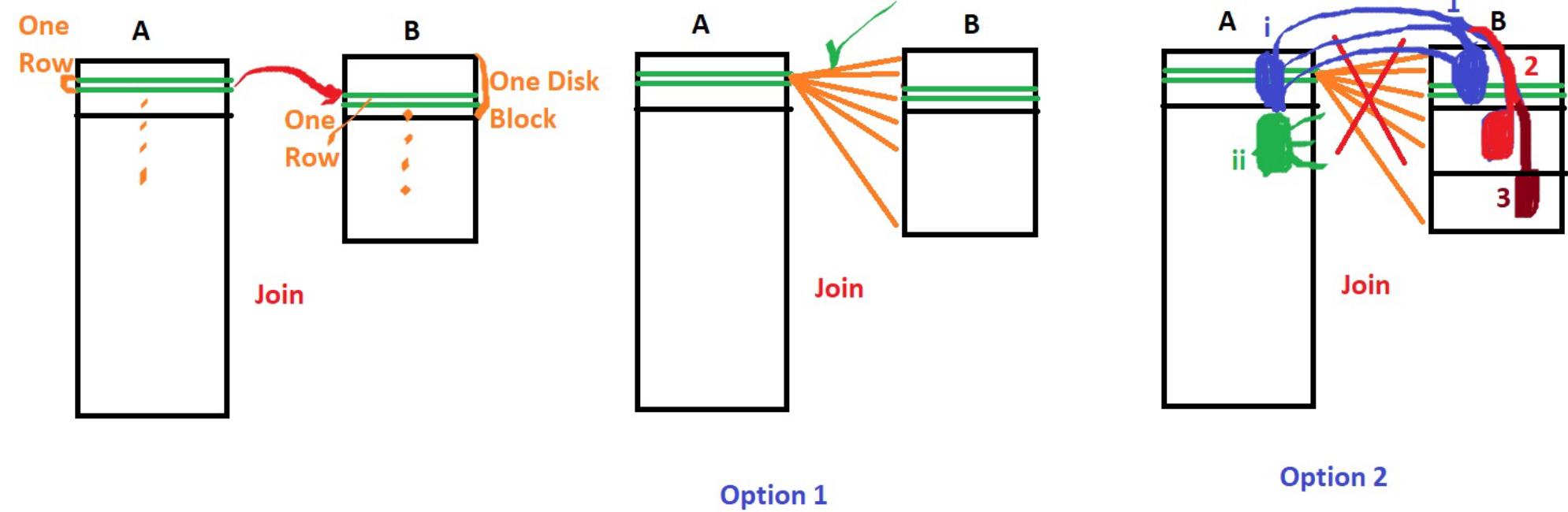
Semester 1, 2023

Optimization & Indexing



Recall we have a DBMS Optimization to do...

We have choices for running a query, even for the same relation algebra expression...





How do we make the choices?

Steps in **cost-based query optimization**

Generate **logically equivalent expressions** of the query

- An SQL query has many equivalent algebra expressions to it
- 2. Annotate resultant expressions to **get alternative query plans**
- 3. **Choose the cheapest plan** based on estimated cost with what you know about costs

Choices Contd.

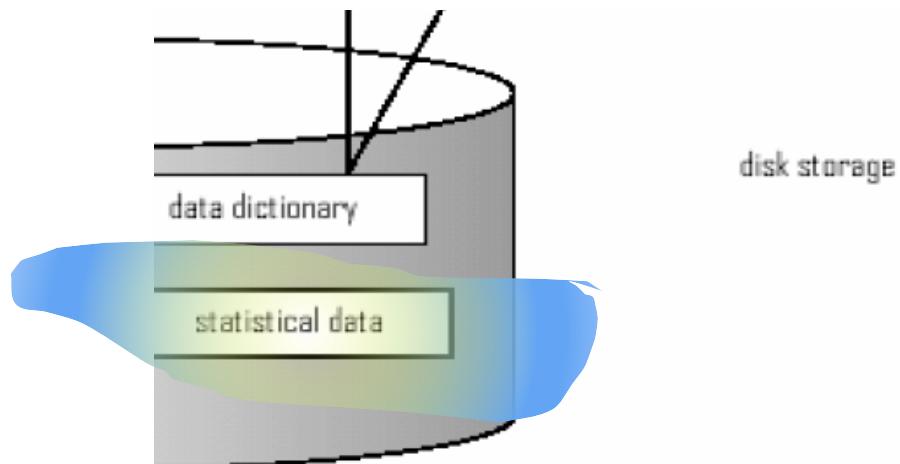
Estimation of plan cost based on:

- **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





First convert SQL to RA expression

select A1, A2, ..., An

from r1, r2, ..., rm

where P

.... is same as the following in relational algebra:

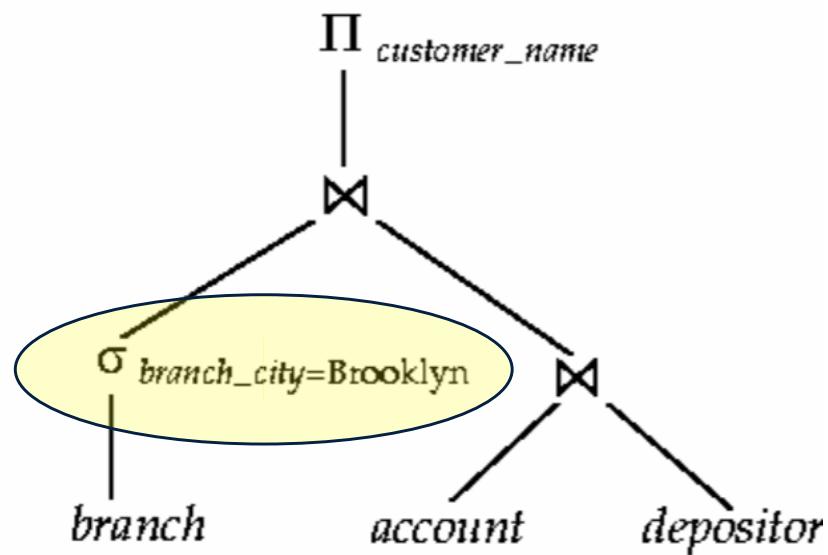
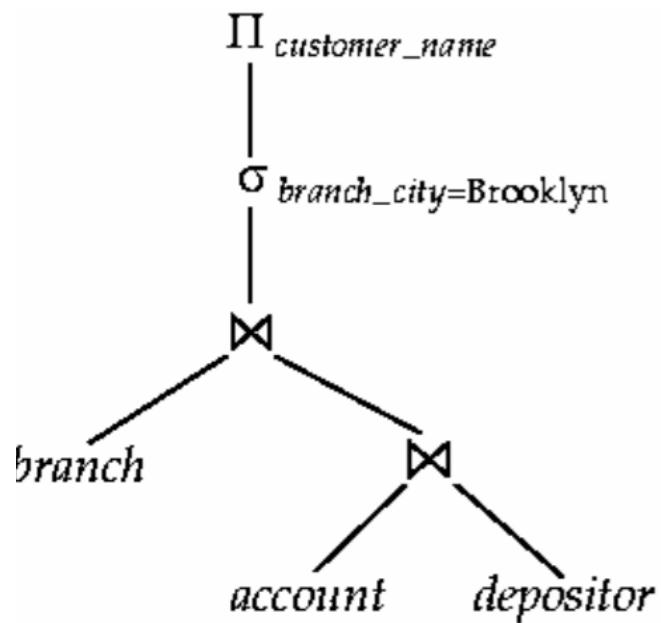
$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

and such an expression can be run in many different ways/orders...

For example: (A join B) join C ? A join (B join C)

Equal expressions ...but different in terms of performance...

Which one is likely to run faster?



Benefits of Relational Algebra and how to generate alternatives?

- Query optimizers use equivalence rules to systematically generate expressions equivalent to:
$$\text{select } A_1, A_2, \dots, A_n \\ \text{from } r_1, r_2, \dots, r_m \\ \text{where } P$$
- SQL does not give this power as it does not tell you what to do and not how you get it (good for the machine)
- And hence relational algebra is used, which is a procedural language and
$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$
- One can generate many alternatives to a relational algebra expression
- Note that the above approach is very expensive in space and time for complex queries



Note but...

- 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**
- Practical query optimizers incorporate elements of the following **two broad approaches**:
 1. Search all the plans and choose the best plan in a cost-based fashion.
 2. Uses heuristics to choose a plan.

Thus In Real Life....

Cost-based optimization is expensive thus....

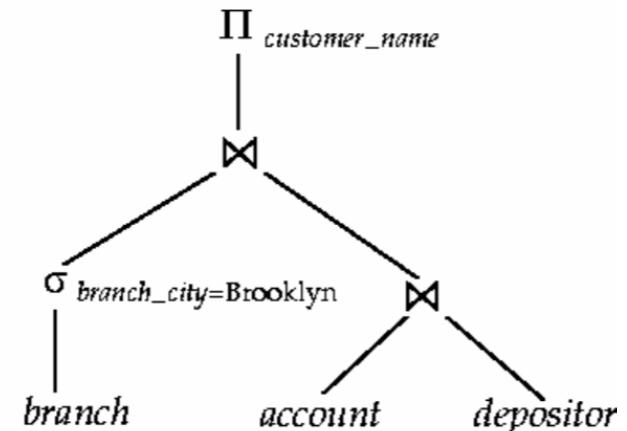
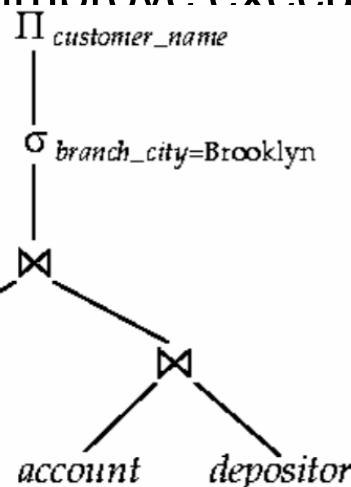
- **Systems may use heuristics to reduce the number of choices** that must be made in a cost-based fashion

- Heuristic optimization transforms the query-tree by **using a set of rules** that typically (but not in all cases) improve execution performance:

1. Perform selections
2. Perform projection
3. Perform most restrictive

smallest result size

branch



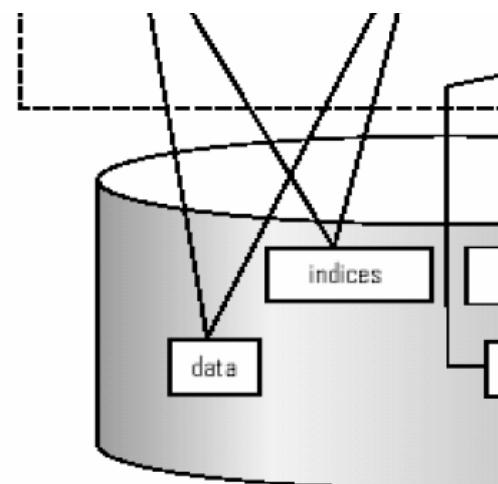


Real Life Contd.

- Some systems use only heuristics, others combine heuristics with cost-based optimization
- **Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries**

A key choice to make before optimization

- DBMS admin generally creates indices to allow almost direct access to individual items
- These indices are also good for some join operations
...if there is a join condition that restricts the no of items to be joined in a table

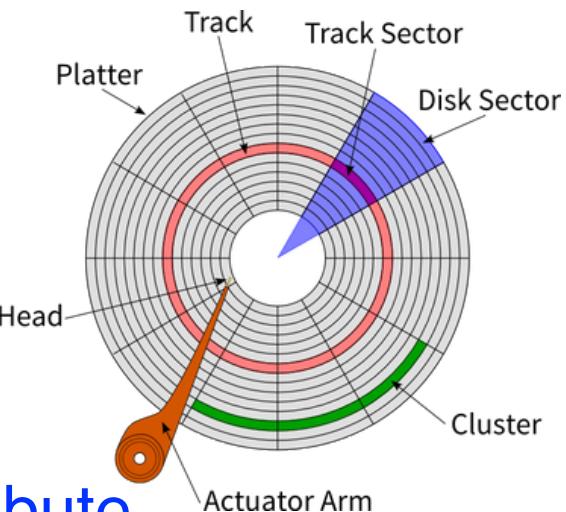




Indexing is Critical for Efficiency in General

- n Indexing mechanisms used to **speed up access** to desired data in a **similar way to look up a phone book**
- n **Search Key** - attribute or set of **attributes used to look up records/rows** in a system like an ID of a person
- n An **index file** consists of records (called index entries) of the form **search-key, pointer to where data is**
- n **Index files are typically much smaller** than the original data files and many parts of it are already in memory
- n Two basic kinds of indices:
 - I **Ordered indices:** search keys are stored in some order
 - I **Hash indices:** search keys are distributed hopefully uniformly across “buckets” using a “function”

But what gets faster?



- n **Disk access** becomes faster through:
 - | records with a specified value in the attribute accessed with minimal disk accesses
 - | or records with an attribute value falling in a specified range of values can be retrieved with a single seek and then consecutive sequential reads
- n Insertion time to index is also important
- n Deletion time is important as well
- n No big index rearrangement after insertion and deletion
- n Space overhead has to be considered for the index itself



Types Contd.

- n **Primary index:** in a sequentially ordered file, the index whose search **key specifies the sequential order of the file**
 - I The search key of a primary index is usually but not necessarily the primary key

- n **Secondary index:** an index whose search key specifies an order **different from the sequential order of the file**

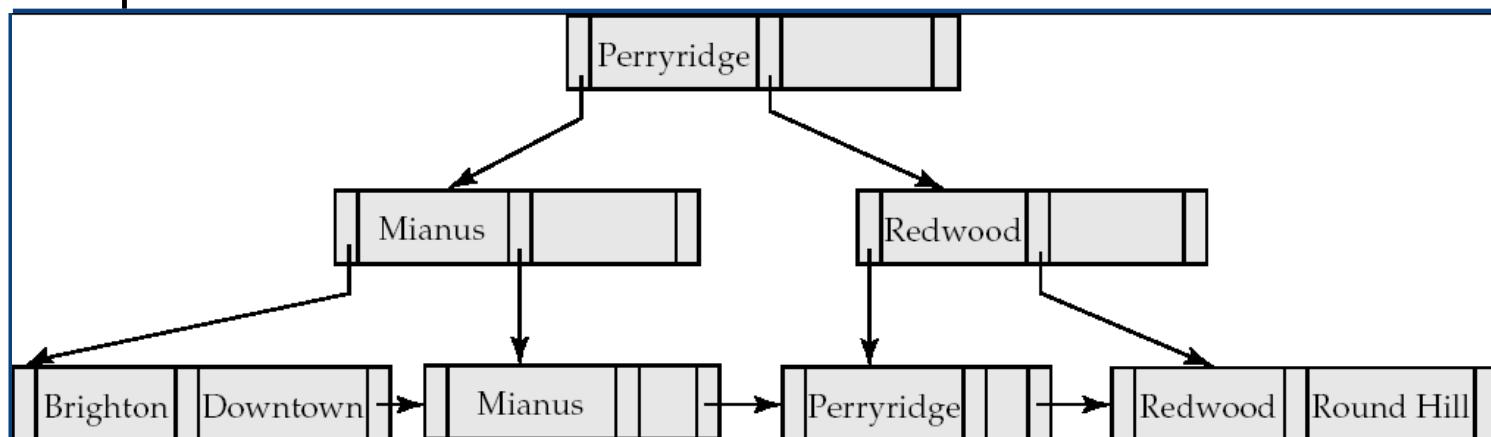
The most popular index in DBMS: **B+trees**

n Why need them:

- | Keeping files in order for fast search ultimately degrades as file grows, since many overflow blocks get created.
- | So binary search on ordered files cannot be done.
- | Periodic reorganization of entire file is required to achieve this.

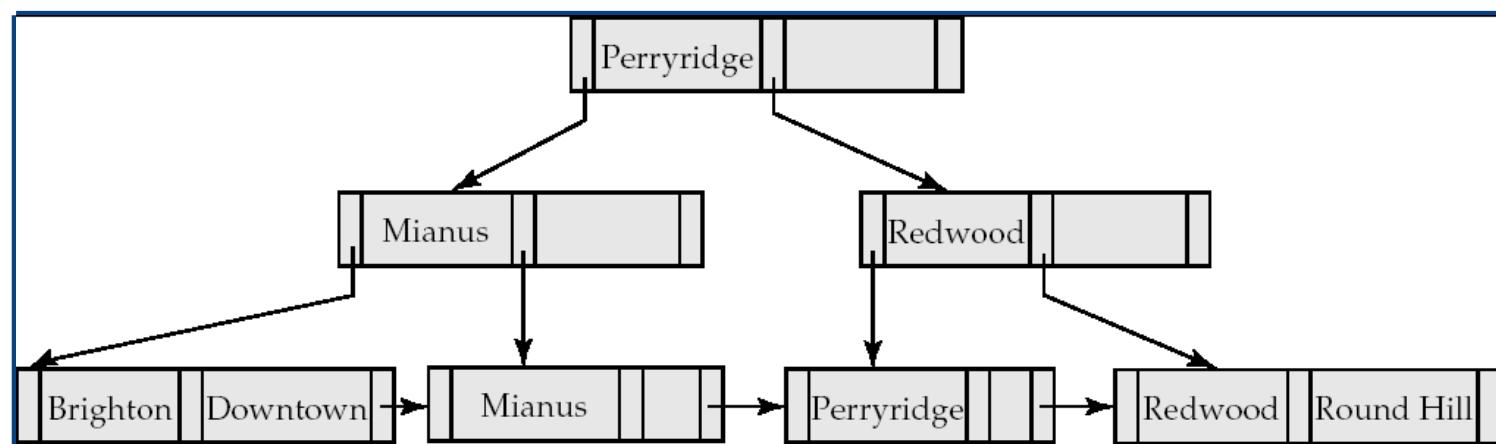
n Advantage of B+-tree index files:

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



B+trees Contd.

- n Similar to Binary tree in many aspects but **the fan out is much higher**
- n Disadvantage of B⁺-trees:
 - | Extra insertion and deletion overhead and space overhead
- n Advantages of B⁺-trees outweigh disadvantages for DBMSs
 - | B⁺-trees are **used extensively**



How is a B+tree defined?

- n It is similar to a binary tree in concept but with a **fan out that is defined through a number n**
- n **All paths from root to leaf are of the same length** (depth)
- n Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- n A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- n Special cases:
 - I If the root is not a leaf, it has at least 2 children.
 - I 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.

A Single Node

- n Typical node

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

| **K_i are the search-key values**

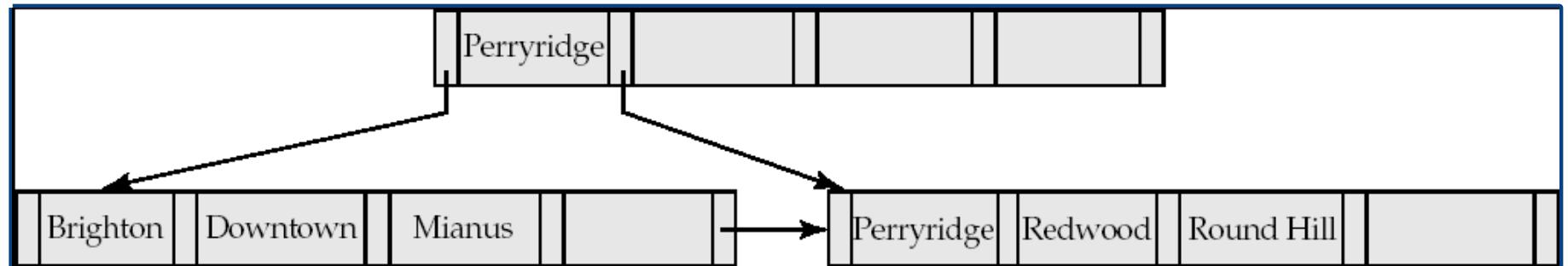
| **P_i are pointers to children** (for non-leaf nodes) **or pointers to records** or buckets of records (for leaf nodes)

- n The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

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

Example B+tree



B⁺-tree for *account* file ($n = 5$)

- n Leaf nodes must have between 2 and 4 values
($\frac{1}{2}n - \frac{n-2}{2}$ and $n-1$, with $n=5$).
- n Non-leaf nodes other than root must have between 3 and 5 children ($\frac{2}{3}n$ and n with $n=5$).
- n Root must have at least 2 children.

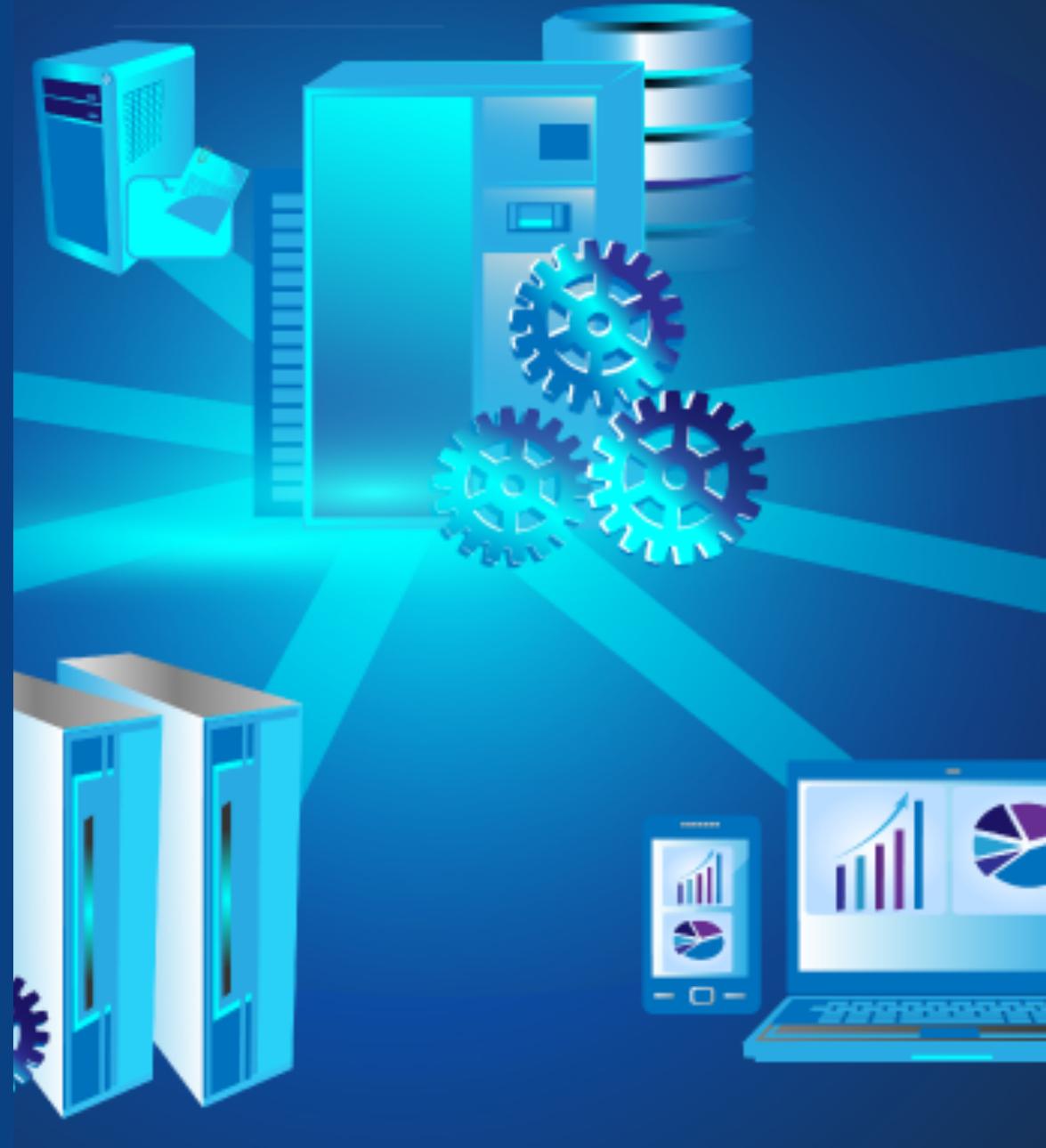


THE UNIVERSITY OF
MELBOURNE

COMP90050 Advanced Database Systems

Semester 1, 2023

Indexing Contd.



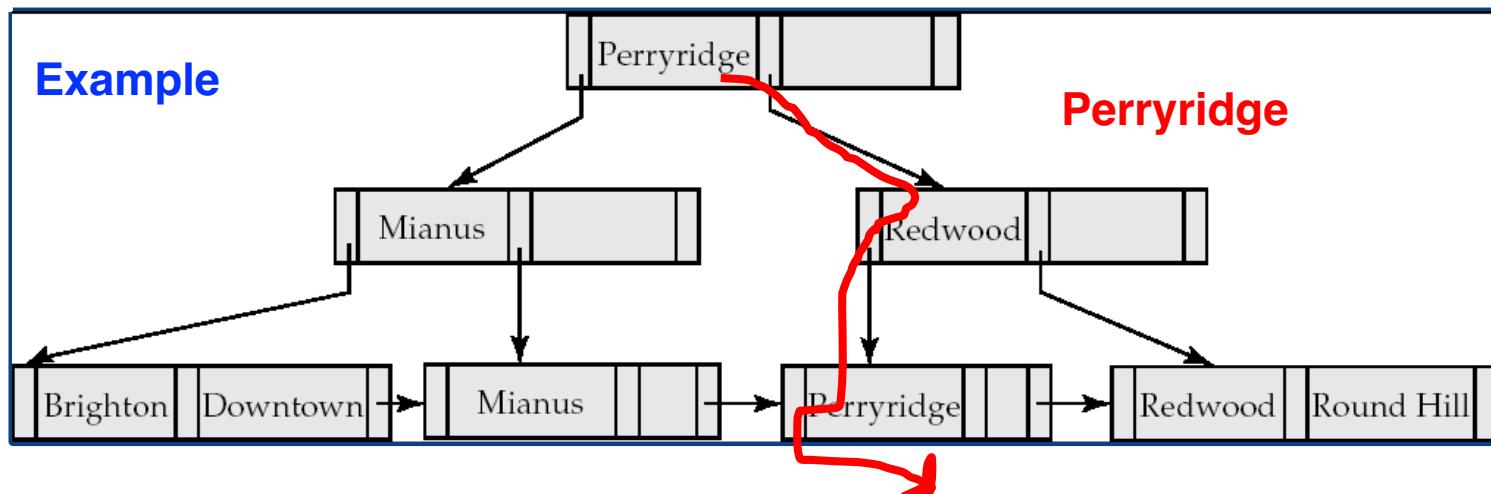
Running a query on B+trees?

n Finding all records with a search-key value of k .

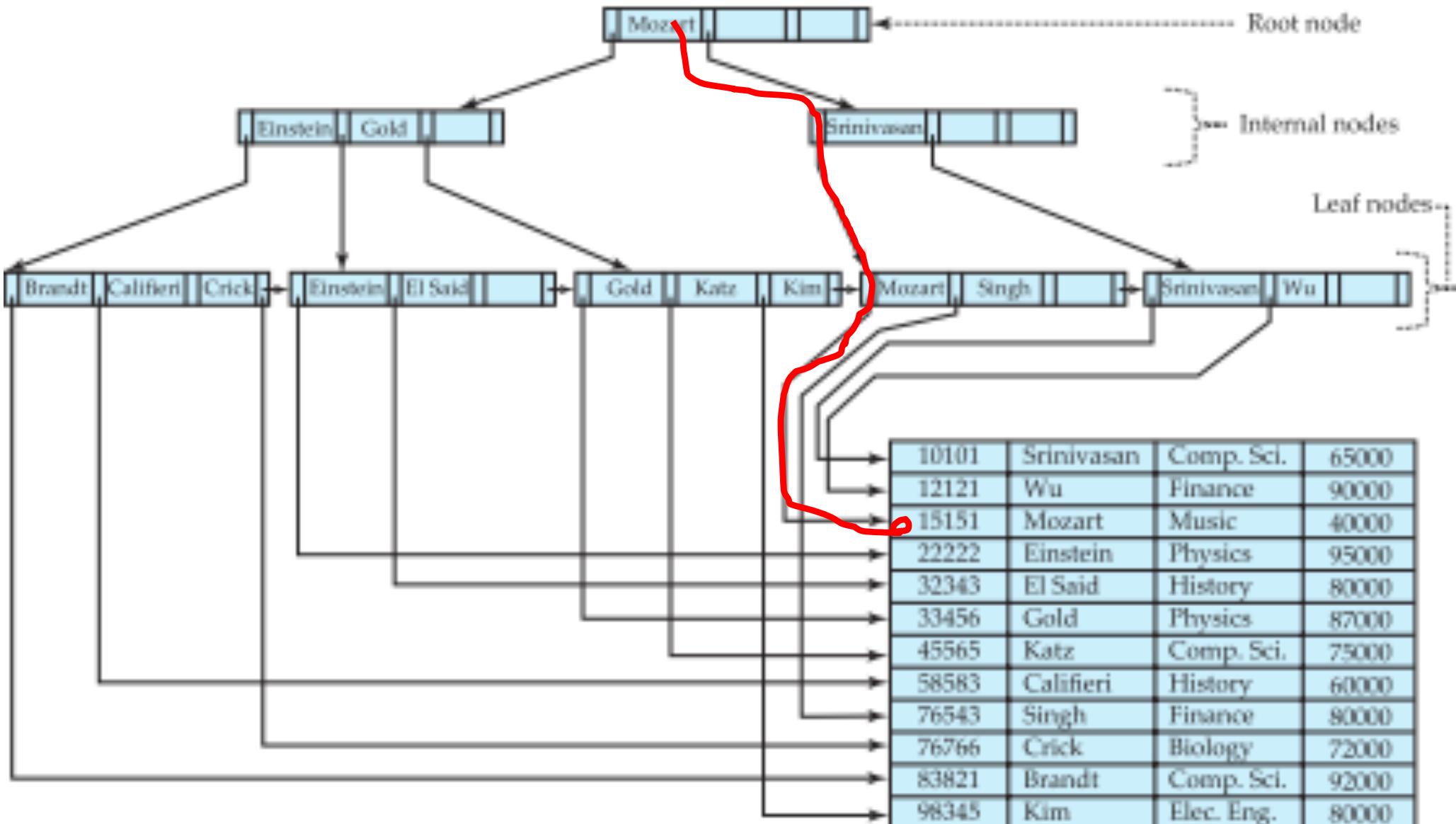
1. $N = \text{root initially}$
2. Repeat
 1. Examine N for the smallest search-key value $> k$.
 2. If such a value exists, assume it is K_i . Then set $N = P_i$
 3. Otherwise $k \leq K_{n-1}$. Set $N = P_n$

Until N is a leaf node

3. If for some i , key $K_i = k$ follow pointer P_i to the desired record or bucket.
4. Else no record with search-key value k exists.



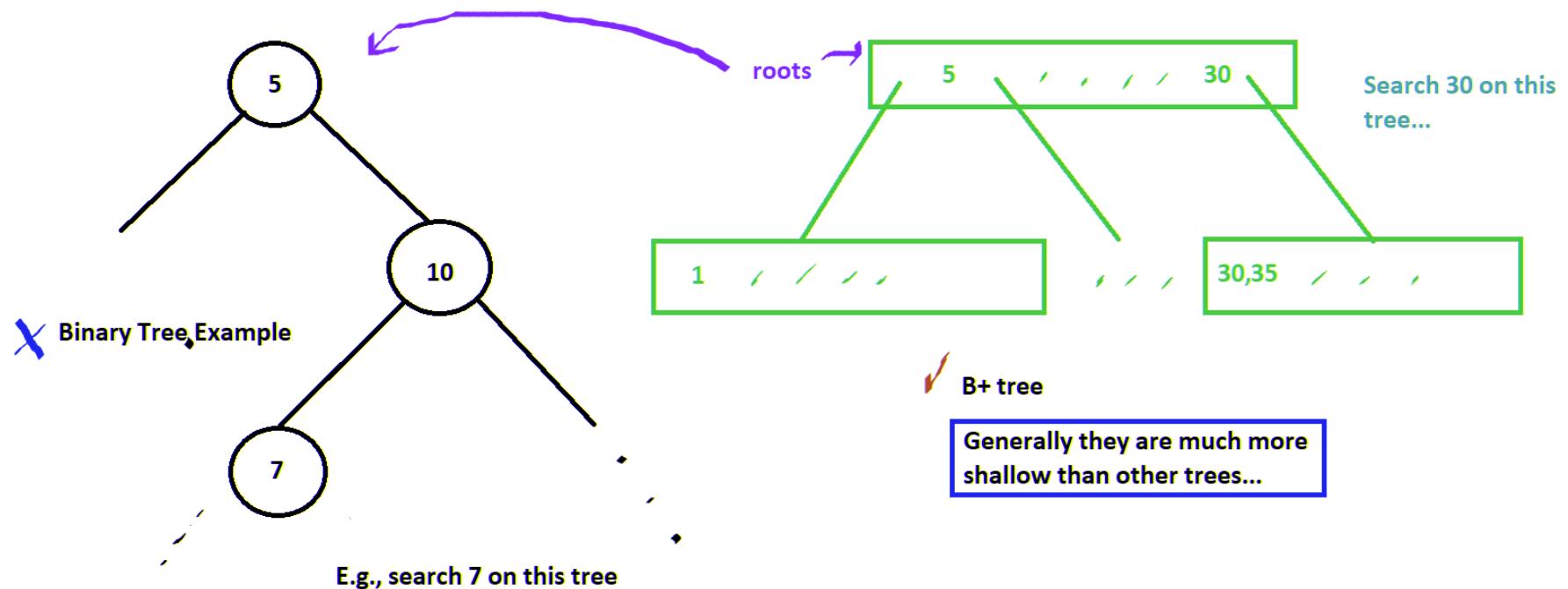
A full view of a B+-Tree



Metrics

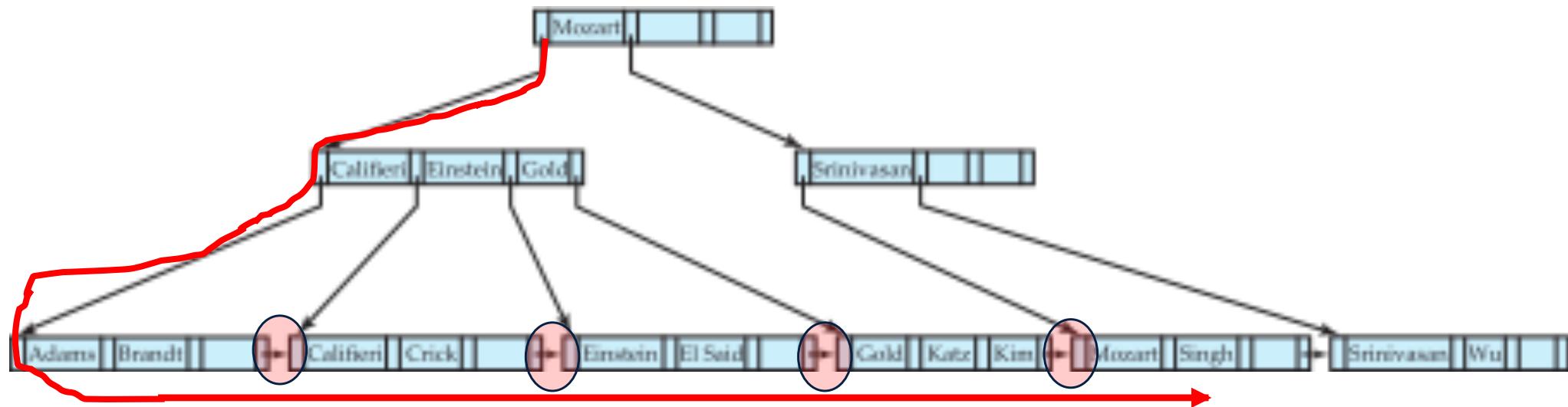
- n If there are **K** search-key values in the file , the height of the tree is **no more than $\log_{50}(K)$** and **it would be balanced**
- n A **node is generally the same size as a disk block** , typically 4 kilobytes
 - | and **n is typically around 100** (40 bytes per index entry).
- n With 1 million search key values and $n=100$
 - | $\log_{50}(1,000,000) = 4$ **nodes are accessed in a lookup** .
- n Contrast this with a balanced binary tree with 1 million search key values — around **20 nodes are accessed in a lookup**
 - | above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

A Quick Visual Search Comparison



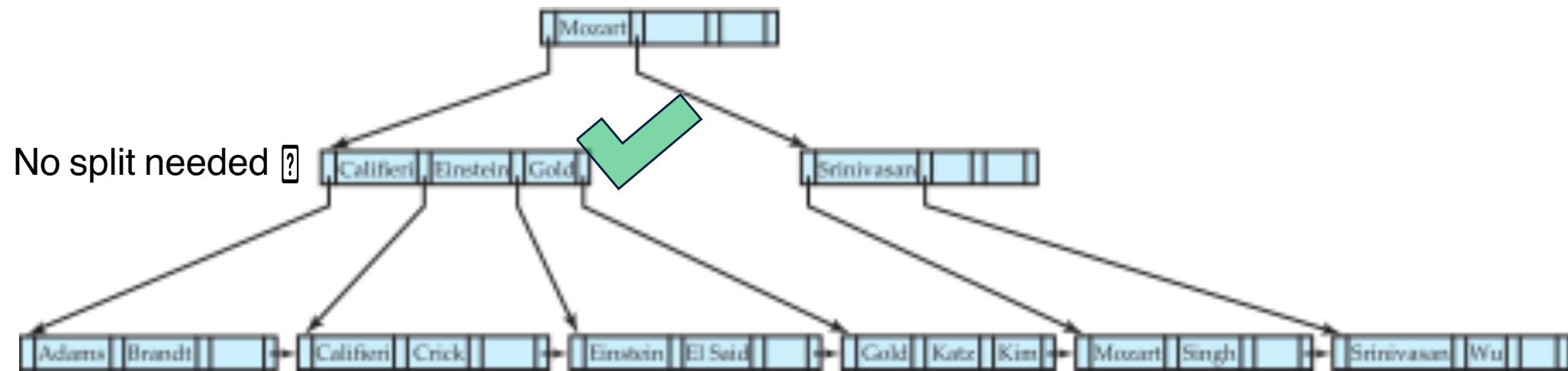
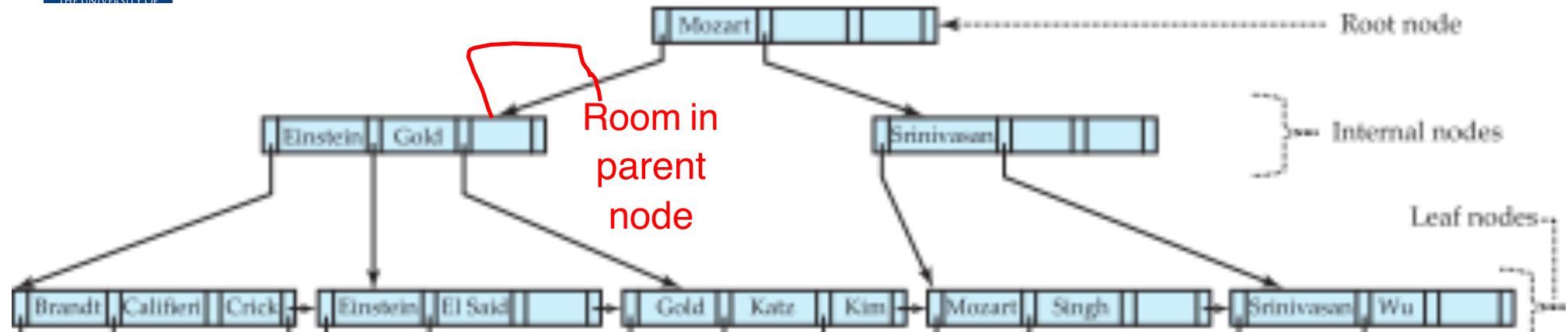
Range Queries on B+-Trees

Range queries find all records with search key values in a given range



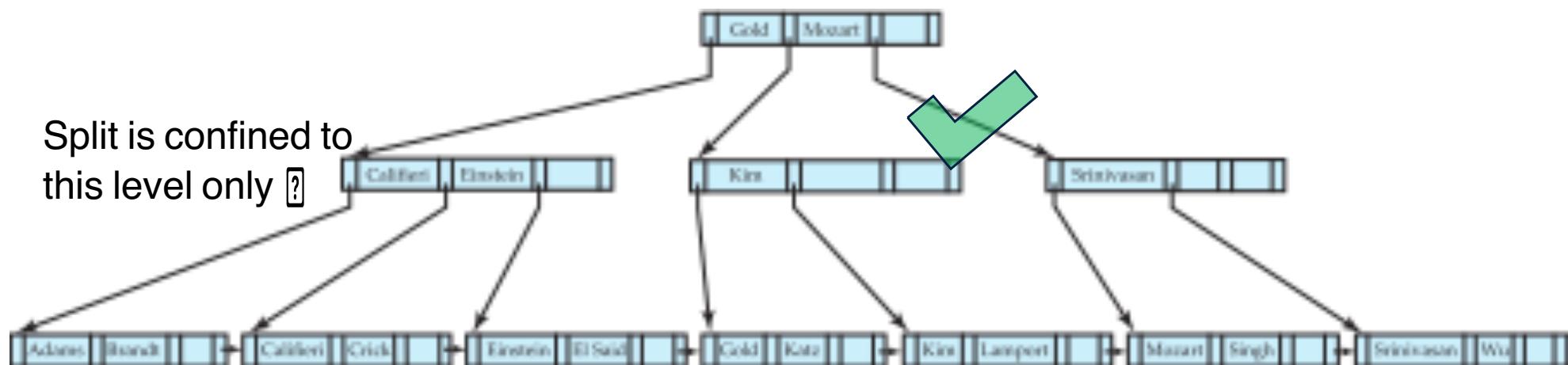
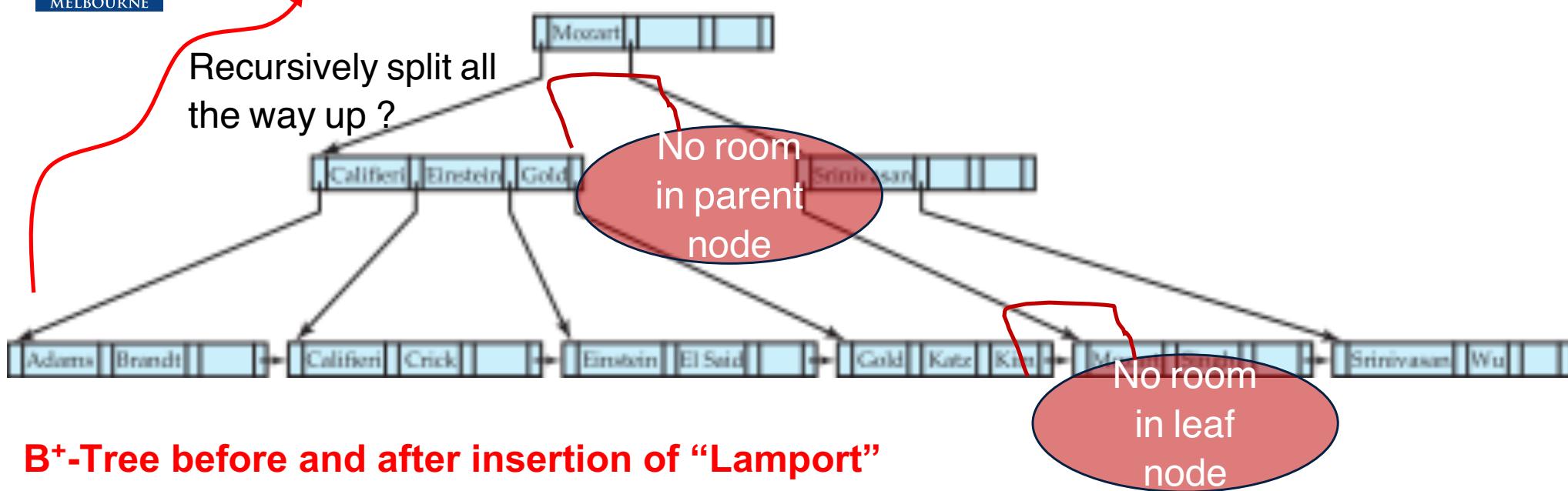


B+-Tree Insertion



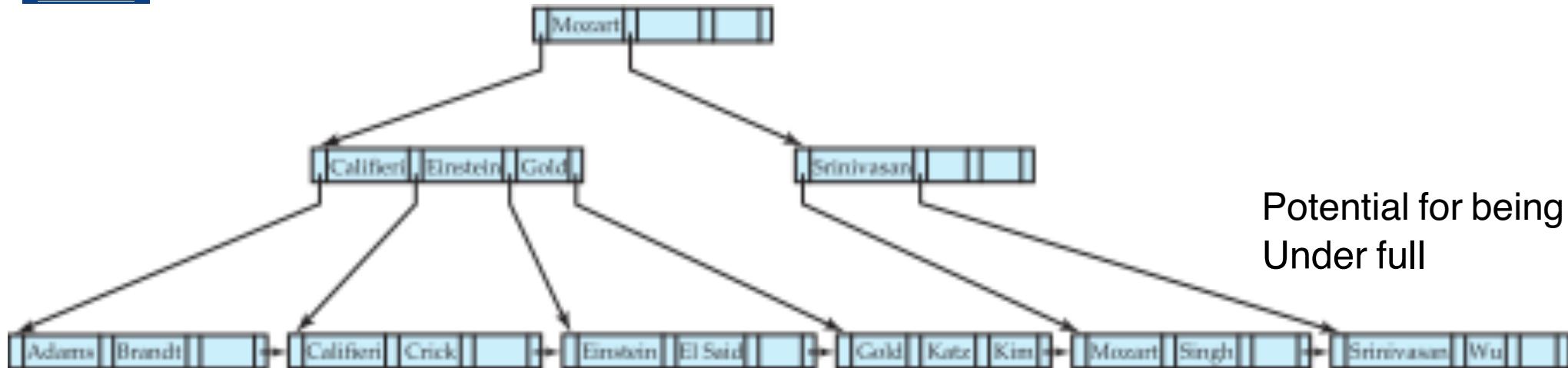
B+-Tree before and after insertion of "Adams"

B+-Tree Insertion Contd.

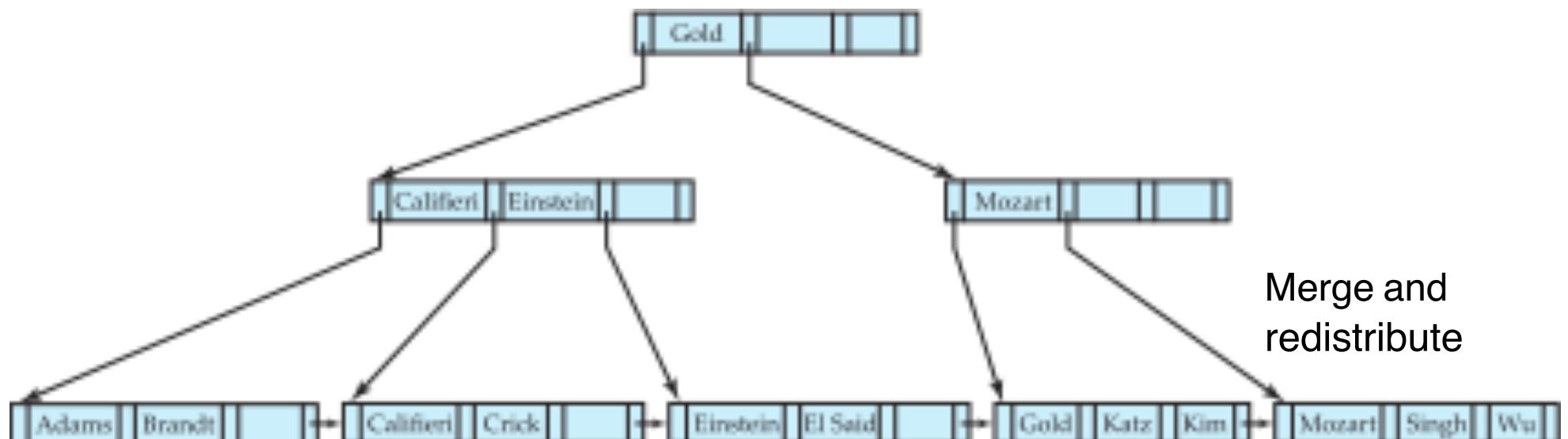




Examples of B+-Tree Deletion

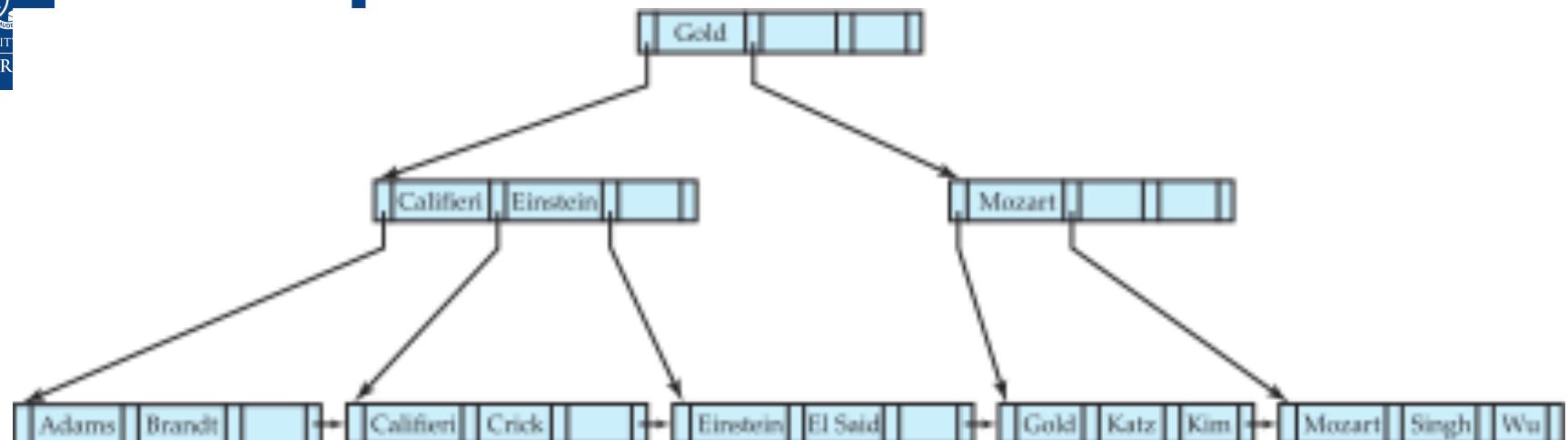


Before and after deleting “Srinivasan”

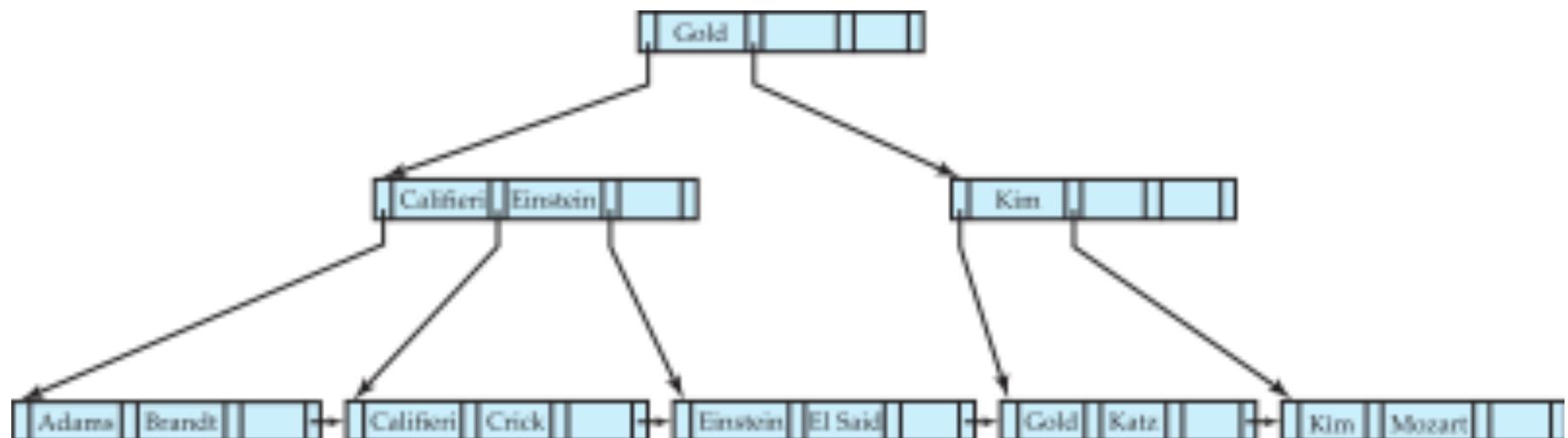


Deleting “Srinivasan” causes merging of under-full leaves

Examples of B+-Tree Deletion



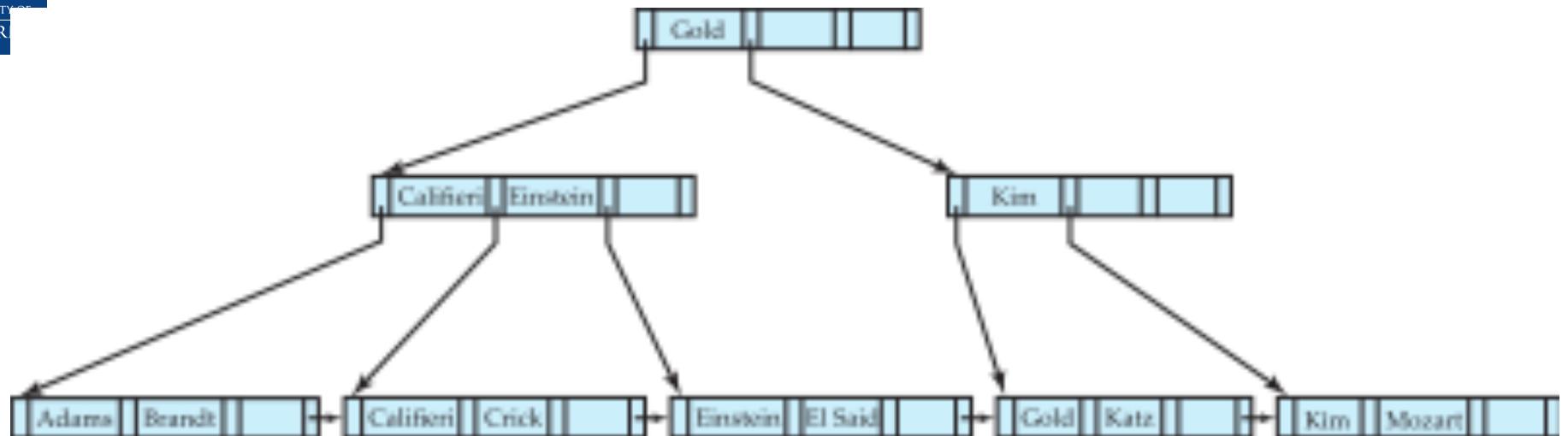
Before and after deleting “Singh” and “Wu”



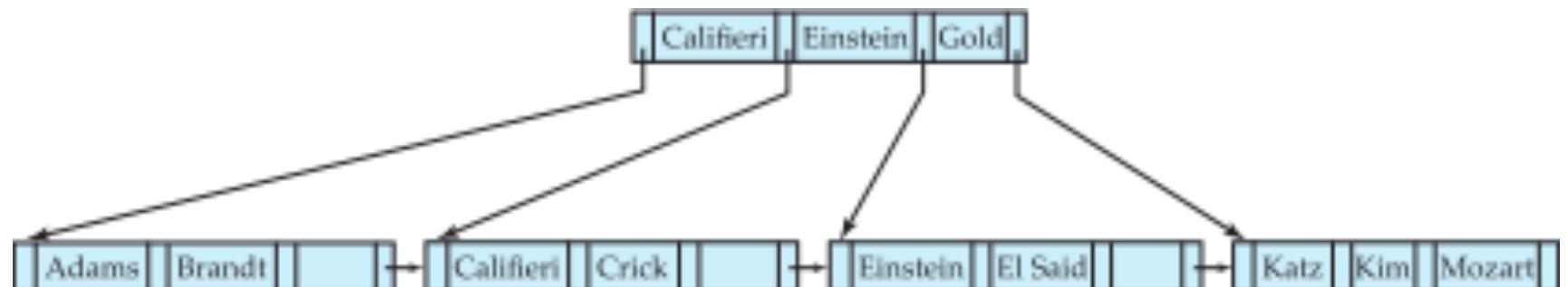
Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling

Search-key value in the parent changes as a result

Examples of B+-tree Deletion



Before and after deletion of “Gold”



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
- Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted



B+-Tree File Organization

- Leaf nodes in a B⁺-tree file organization store records, instead of pointers
- Helps **keep data records clustered** even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in a similar way as insertion and deletion of entries in a B⁺-tree index.



Index Definition in SQL

Create an index:

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

E.g.,: **create index b-index on branch(branch_name)**

Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key

To drop an index

```
drop index <index-name>
```

Most database systems allow specification of type of index and clustering

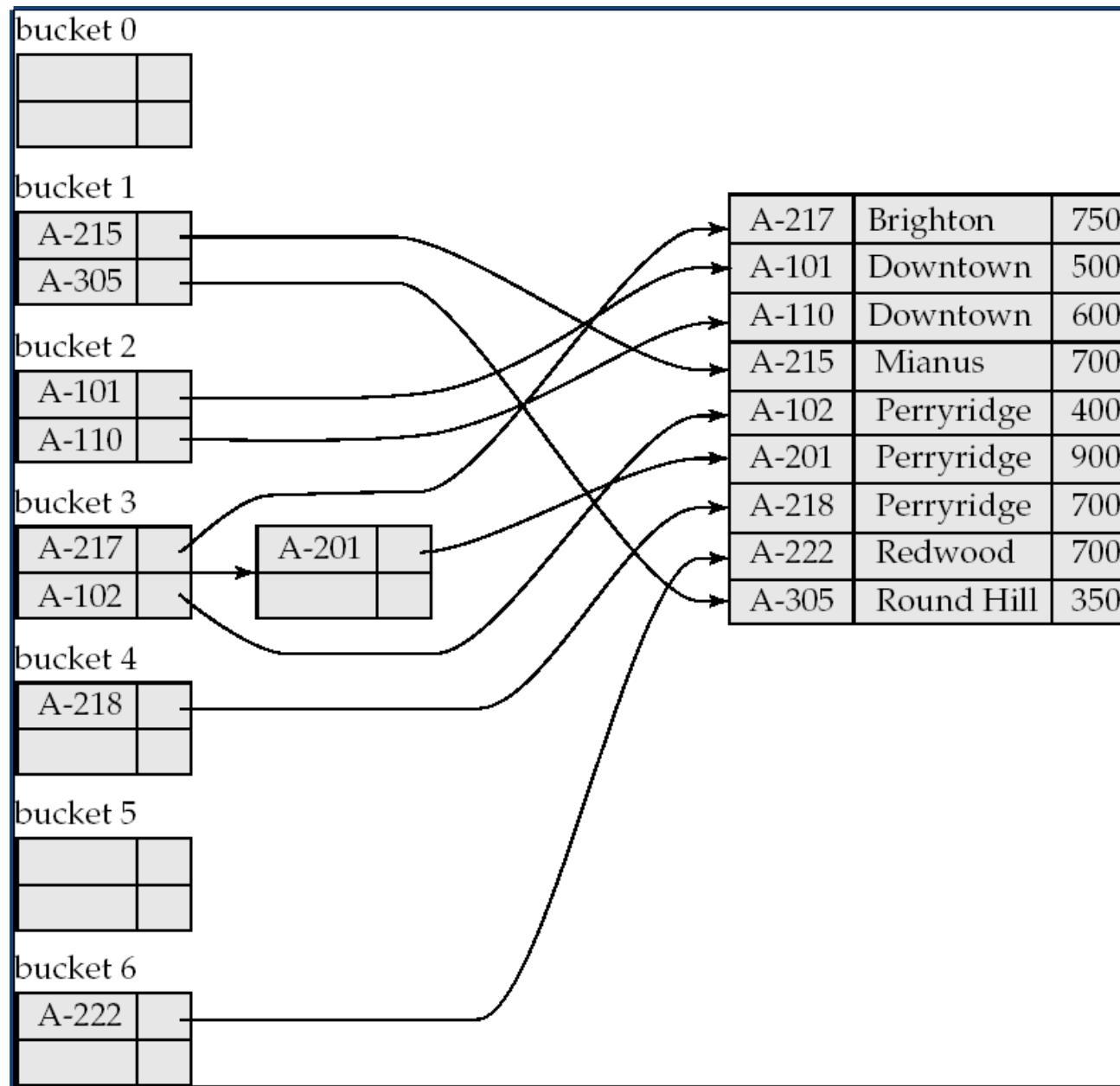


Hash Indices

- n A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure. **Order is not important.**
- n Hash indices are **always secondary indices**.
- n Given a key the aim is to find the related record on file in **one shot** which is important.
- n An **ideal hash function is uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- n **Ideal hash function is random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- n Typical hash functions perform computation on the internal binary representation of the search-key.

Example for Hashing

Keys not in order
As they are random



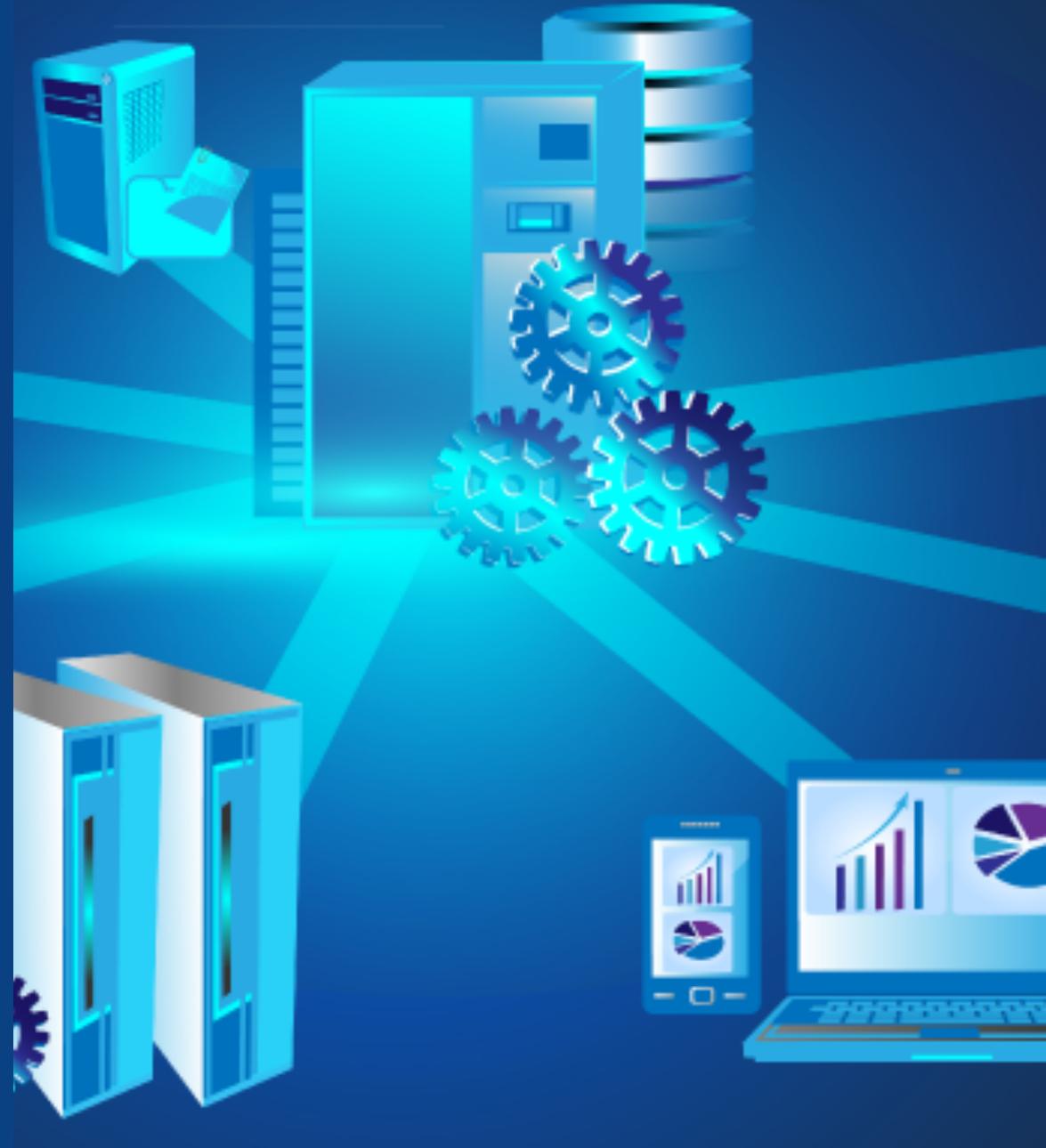


THE UNIVERSITY OF
MELBOURNE

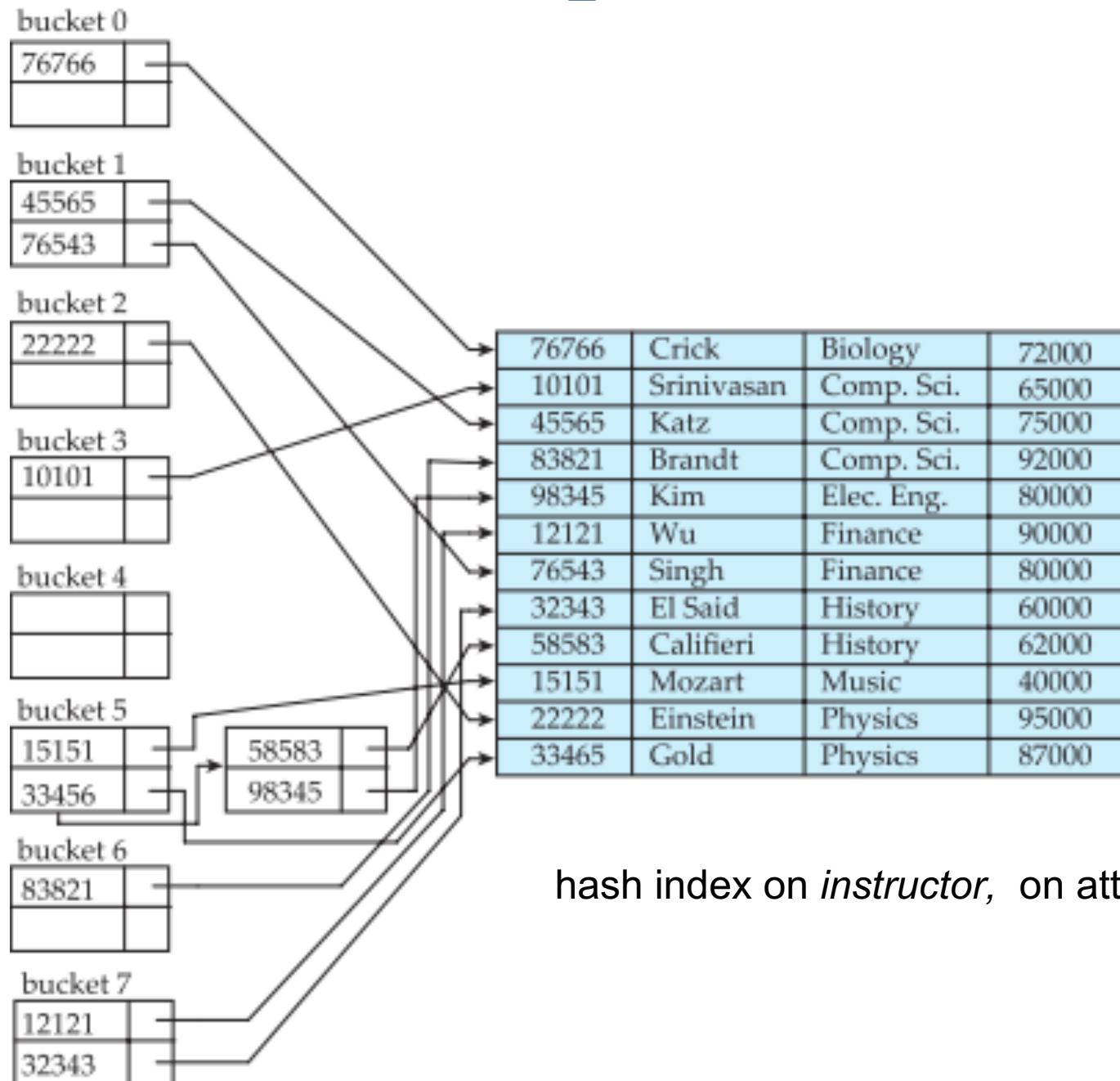
COMP90050 Advanced Database Systems

Semester 1, 2023

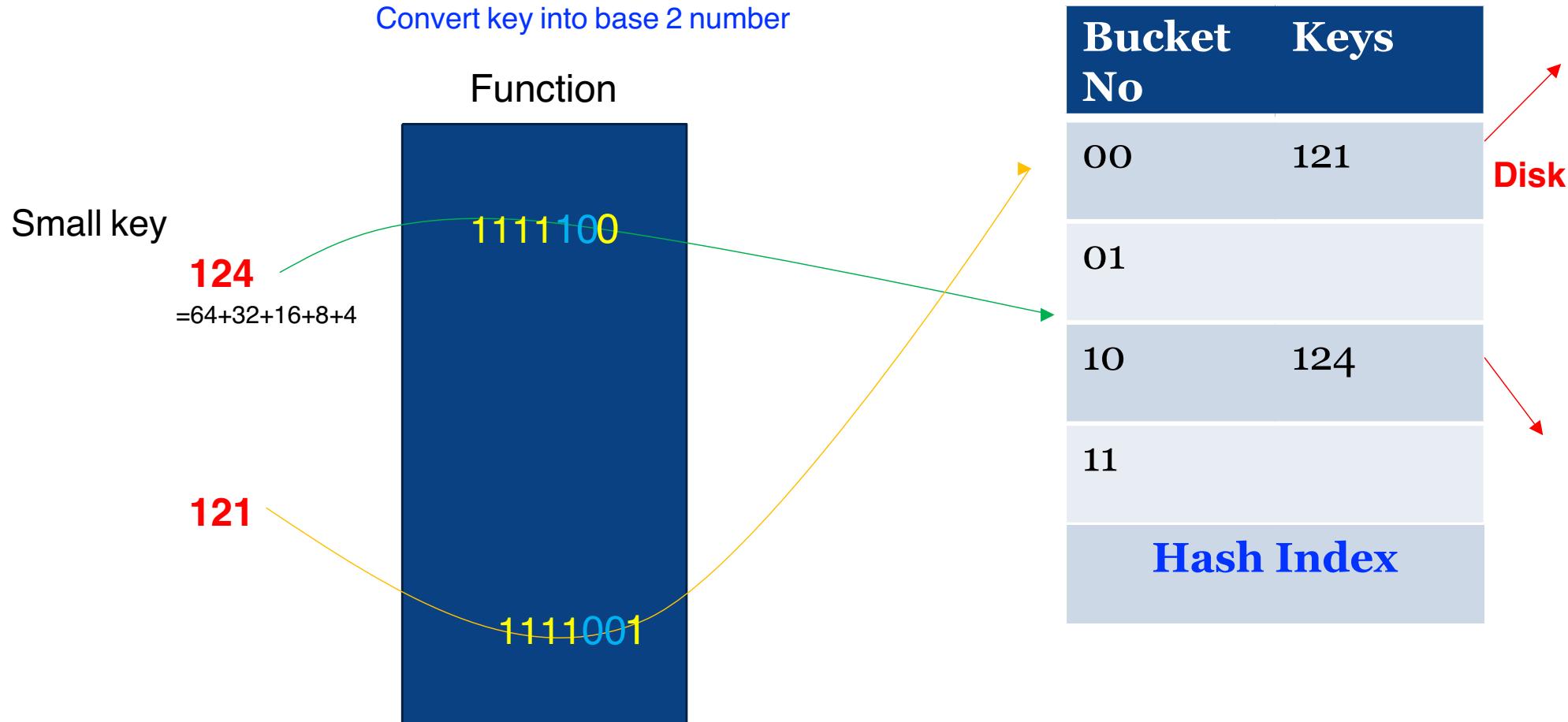
Indexing Contd.



Another example of Hash Index



What is a Hash Function?





Bitmap Indices

- n Records in a relation are assumed to be numbered sequentially from, say, 0
- n Applicable on attributes that take on a relatively small number of distinct values
 - | E.g. gender, country, state, ...
 - | E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- n A bitmap is simply an array of bits....

Example

- n In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute

- | **Bitmap has as many bits as records**
- | In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
- | Used for business analysis, where rather than individual records say how much of one type exists is the query/important

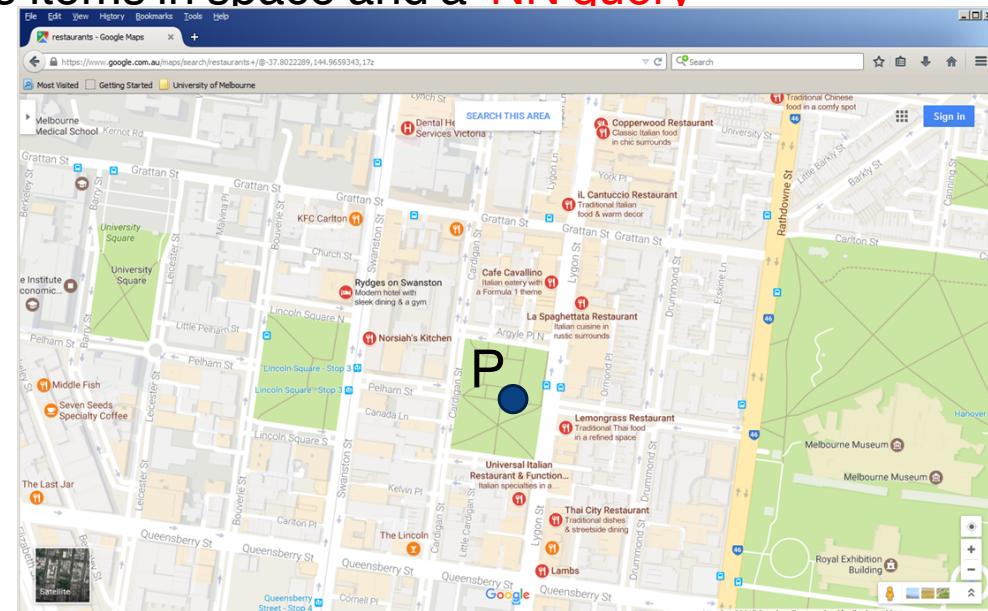
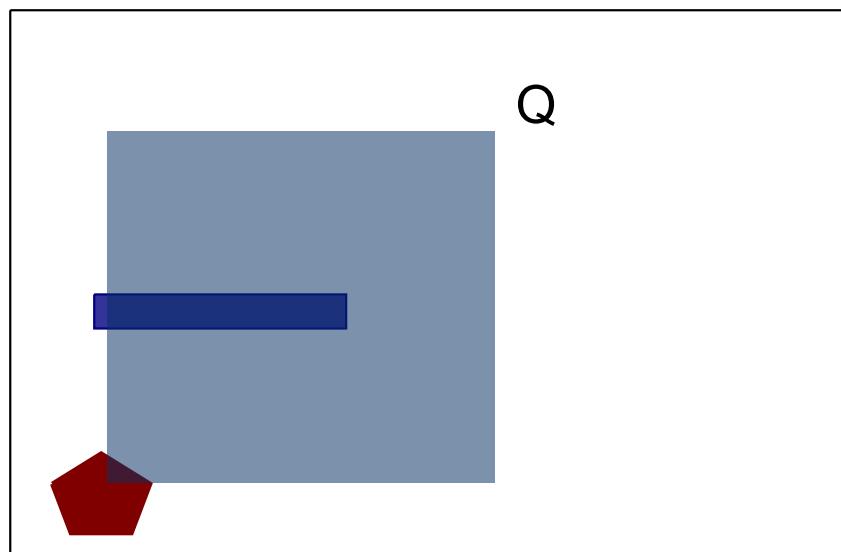
record number	<i>name</i>	<i>gender</i>	<i>address</i>	<i>income_level</i>	Bitmaps for <i>gender</i>	Bitmaps for <i>income_level</i>
0	John	m	Perryridge	L1	m 1 0 0 1 0	L1 1 0 1 0 0
1	Diana	f	Brooklyn	L2	f 0 1 1 0 1	L2 0 1 0 0 0
2	Mary	f	Jonestown	L1		L3 0 0 0 0 1
3	Peter	m	Brooklyn	L4		L4 0 0 0 1 0
4	Kathy	f	Perryridge	L3		L5 0 0 0 0 0

Indexing for Complex Data Types

Unlike things we can access by names, numerical ids, etc there is a lot of other types of data that exists and increasingly in use in DBMS and that requires special indexing

For example, spatial data requires more complex computations for accessing data, e.g., intersections of objects in space

There is no trivial way to sort items which is a key issue in that case, e.g., below is a common range query on a simple set of two items in space and a NN query



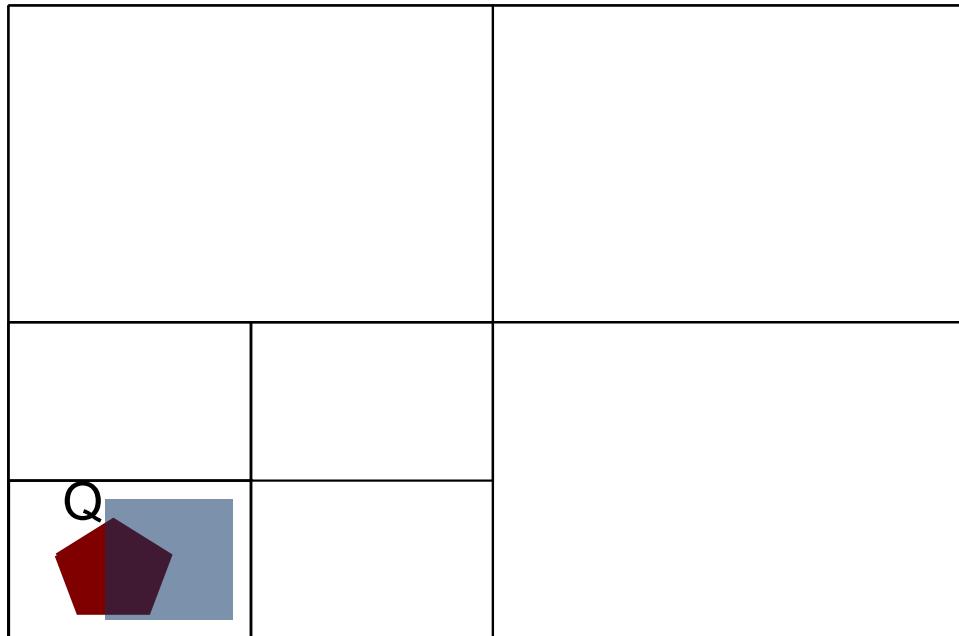


How do we create buckets for indexing in that case?

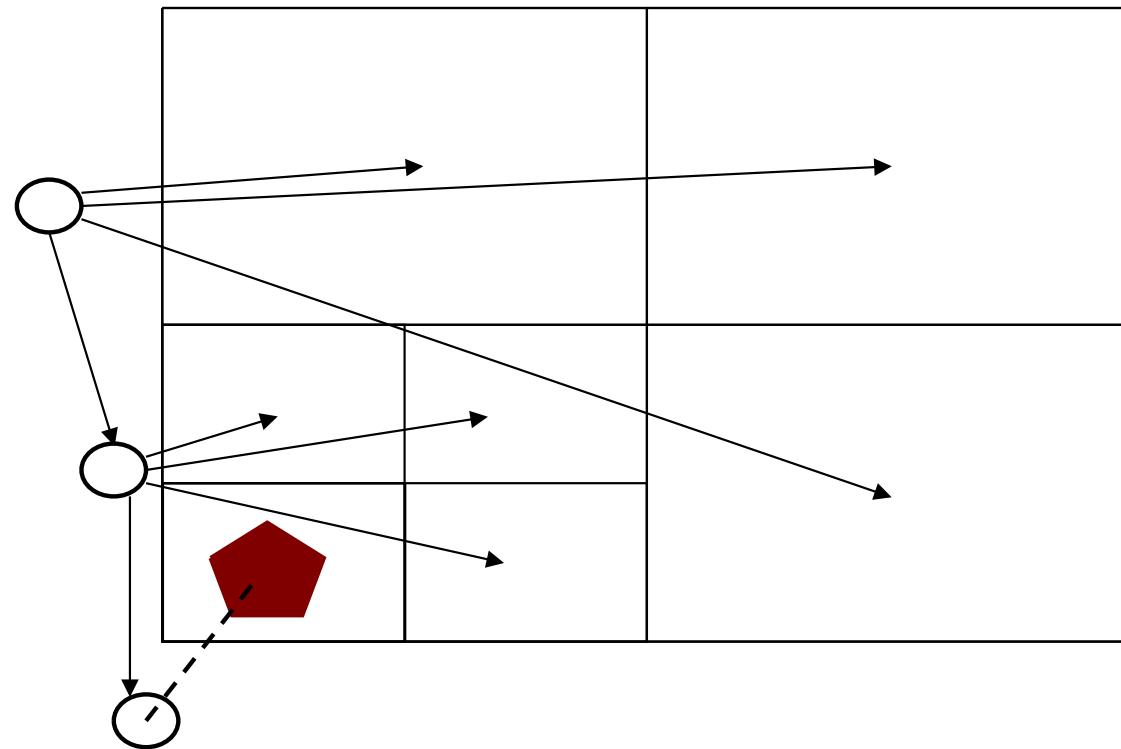
In 2-d space, similar to B+trees for **exponentially reducing** the number of comparisons/retrievals, but via a repetitive division of space, novel indices were invented

They are in use in Oracle Spatial and other comparable DBMS extensions

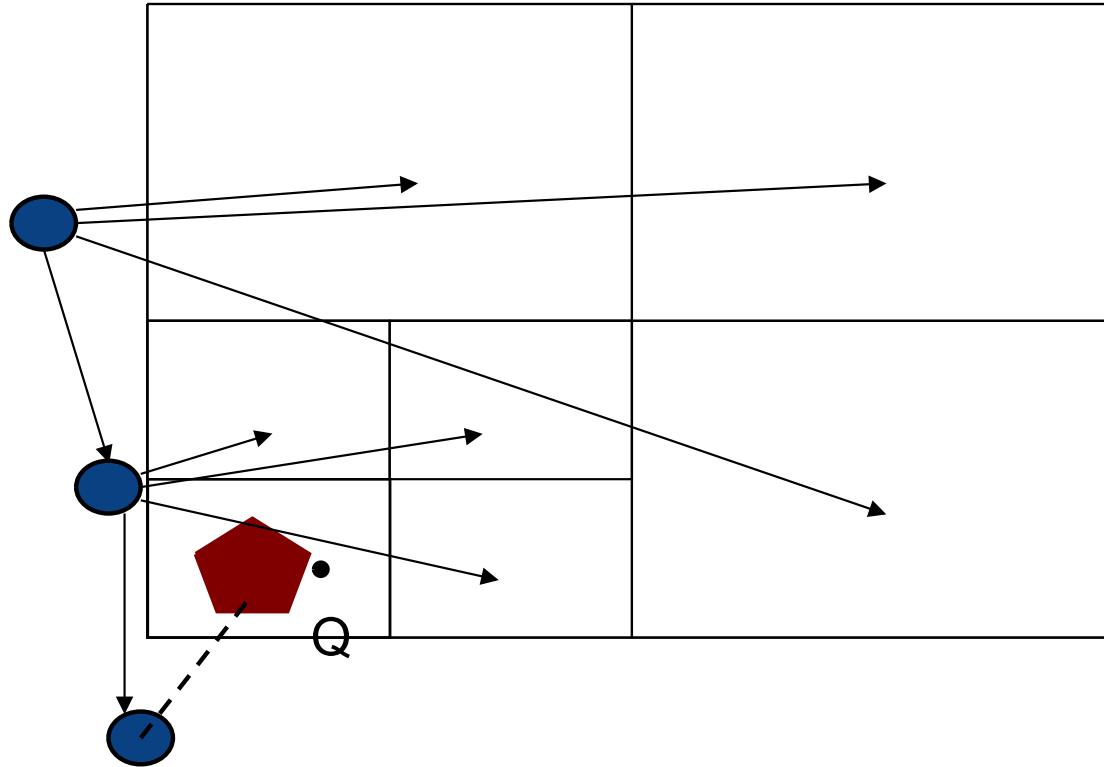
The following class of data structures is one such index: **Quadtrees**



But where is the tree structure?



How to run a Nearest Neighbor query?



Basically via similar approach that we do in other trees, e.g., Best First Search

Just order the access with respect to distance to a point

Versions of Quadtrees

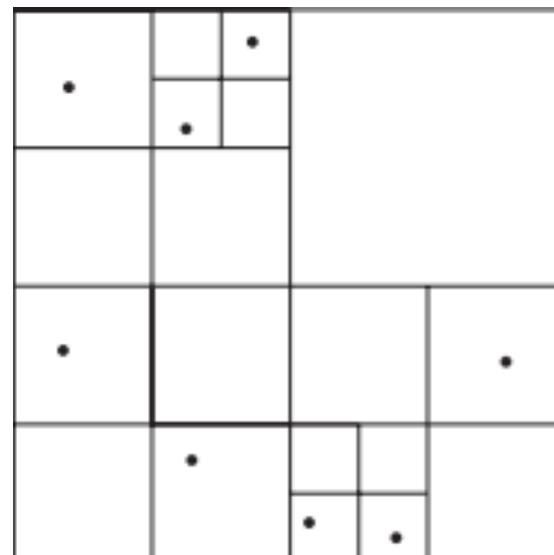
Each node of a quadtree is associated with a rectangular region of space ; the top node is associated with the entire target space.

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

Each non-leaf node 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.

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



Indexing of Spatial Data Contd.

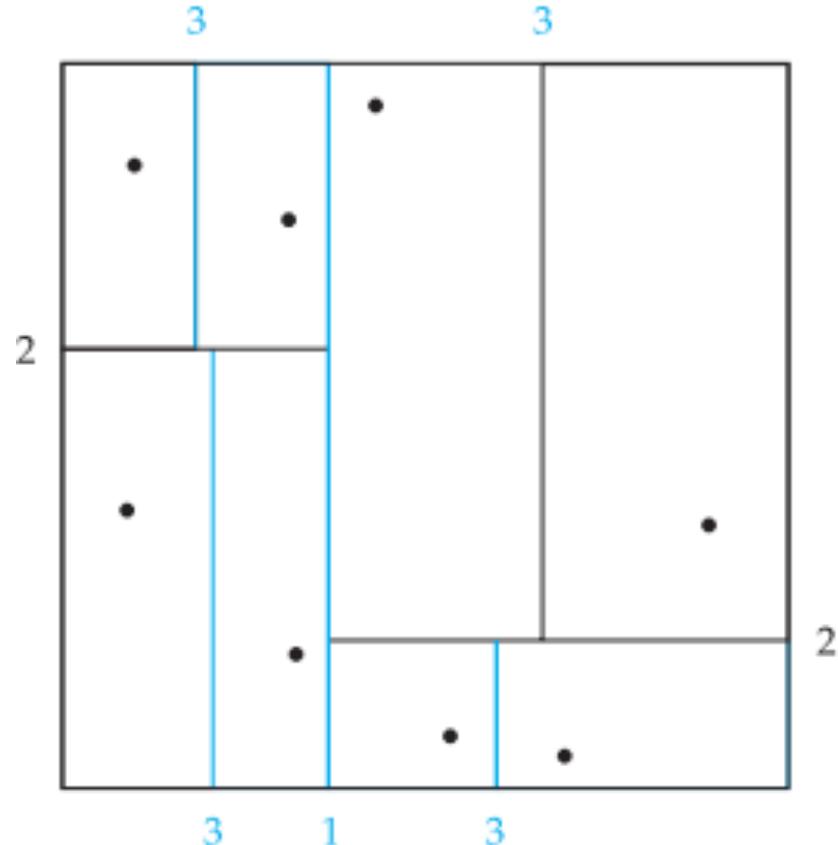
k-d tree – another structure used for indexing in multiple dimensions.

Each level of a **k-d tree** partitions the space into two.

- Choose one dimension for partitioning at the root level of the tree.
- Choose another dimension for partitioning in nodes at the next level and so on, cycling through the dimensions.

In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.

Partitioning stops when a node has less than a given number of points.



Feature	Quadtree	K-D tree
Implementation	Each node has four children, dividing space into quadrants.	Each node has two children, dividing space by a hyperplane.
Search	$O(\log n)$ for uniformly distributed data.	$O(\sqrt{n})$ for uniformly distributed data, $O(n)$ worst-case.
Data types	2D and 3D spatial data, images, terrain, and maps.	High-dimensional data, such as points in a k-dimensional space.
Split rule	Divides space into four equal quadrants.	Divides space along the median of a single dimension.
Query range	Easy to search for points within a rectangular area.	Difficult to search for points within a non-rectangular area.
Memory usage	Uses more memory than a K-D tree.	Uses less memory than a Quadtree.

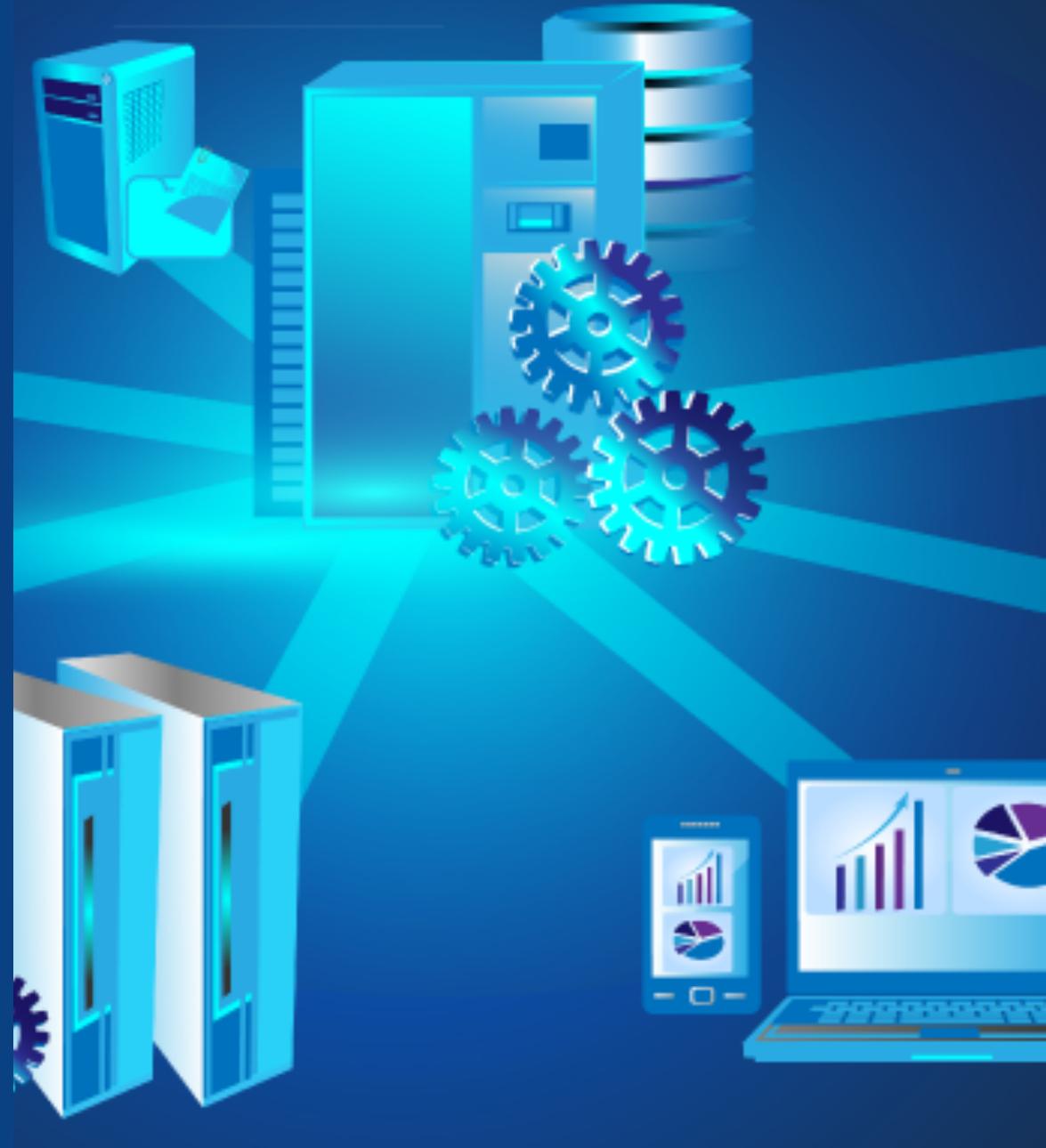


THE UNIVERSITY OF
MELBOURNE

COMP90050 Advanced Database Systems

Semester 1, 2023

Indexing Contd.



R-Trees

R-trees can be considered as a **N-dimensional version of B⁺-trees**, useful for indexing sets of rectangles and other polygonal data.

Supported in many modern database systems, with variants like R⁺-trees and R^{*}-trees.

Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an **N-dimensional interval**, that is, an N-dimensional rectangle.

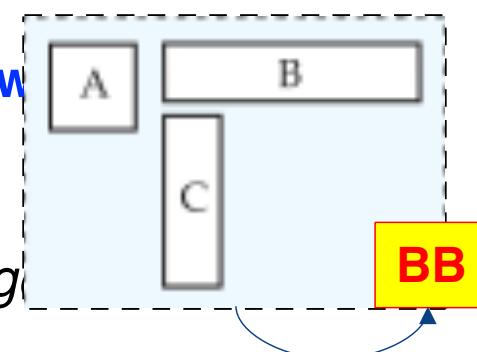
We will consider only the common two-dimensional case ($N=2$)

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

Disadvantage:

Bounding boxes of children of a node are allowed

A **bounding box** (BB) of a node is a minimum-sized rectangle that contains all the rectangles/polygons with the node

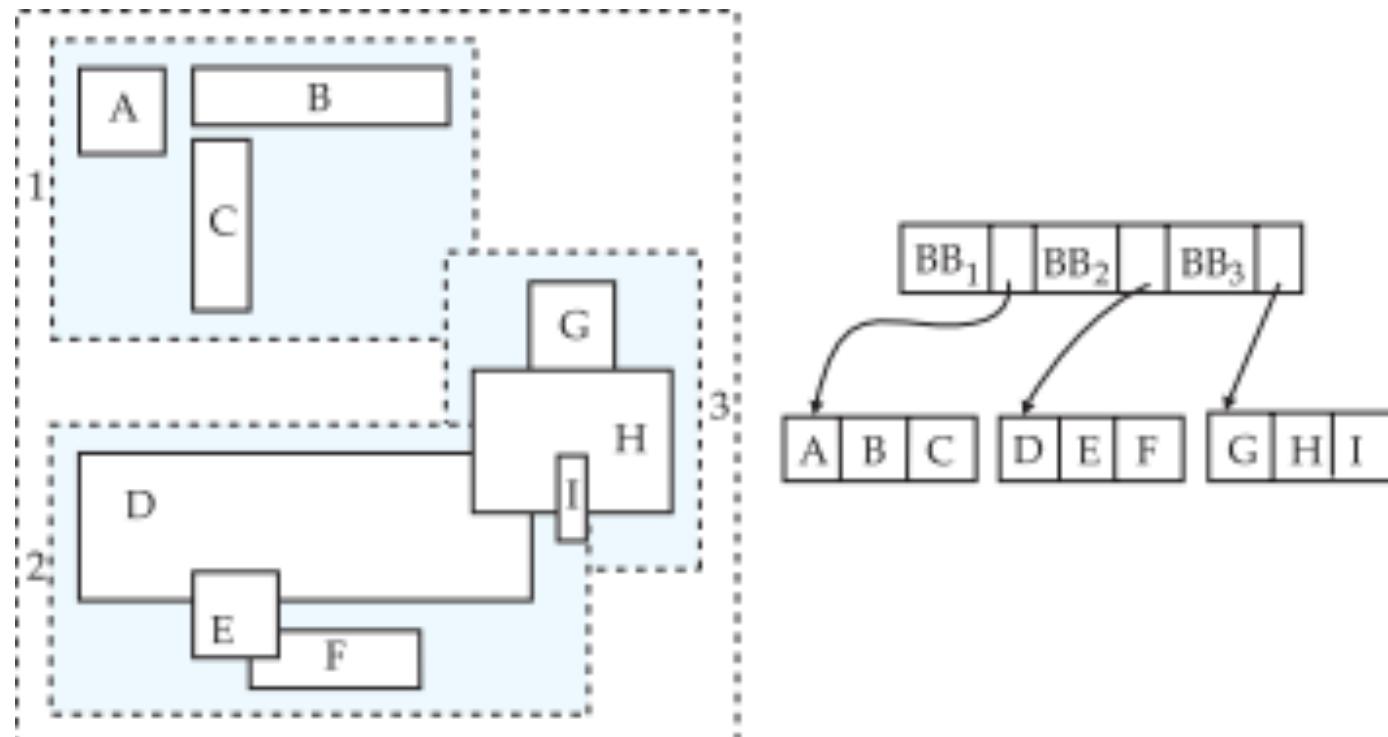


Example R-Tree

A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.

The **R-tree is shown on the right**.

The clustering of objects are based on a rule.

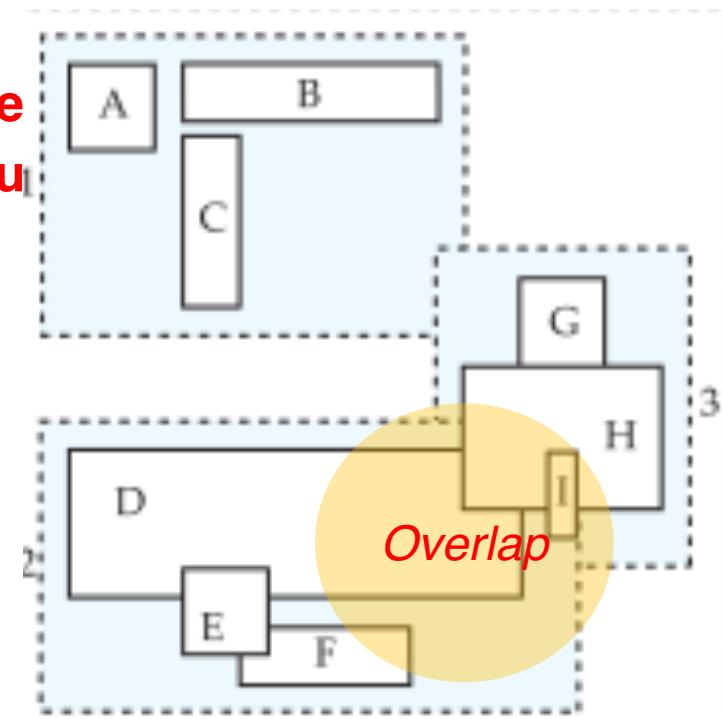


Search in R-Trees

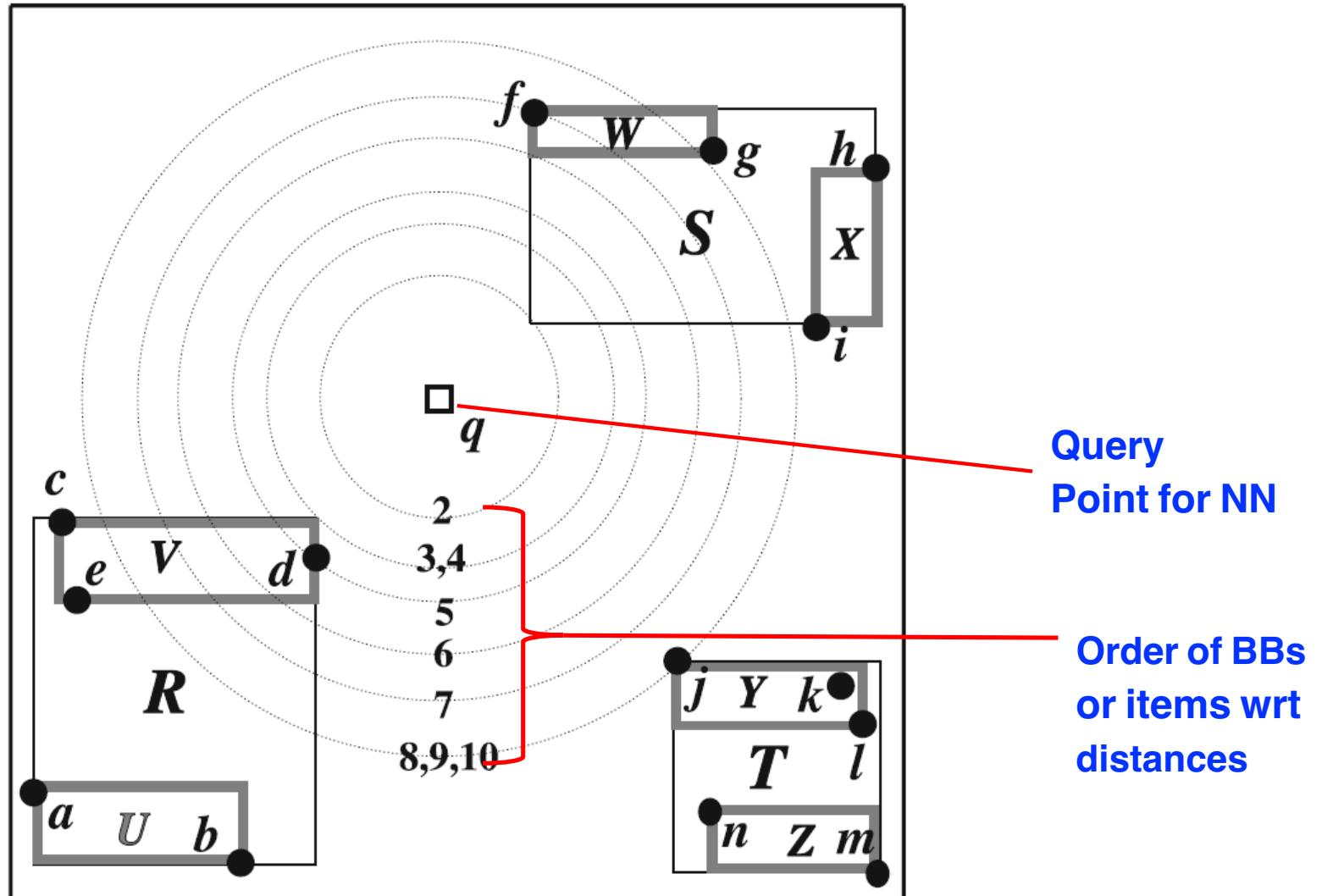
To find data items intersecting a given query point/region, do the following, starting from the root node:

- 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 intersects the query point/region, recursively search that child.

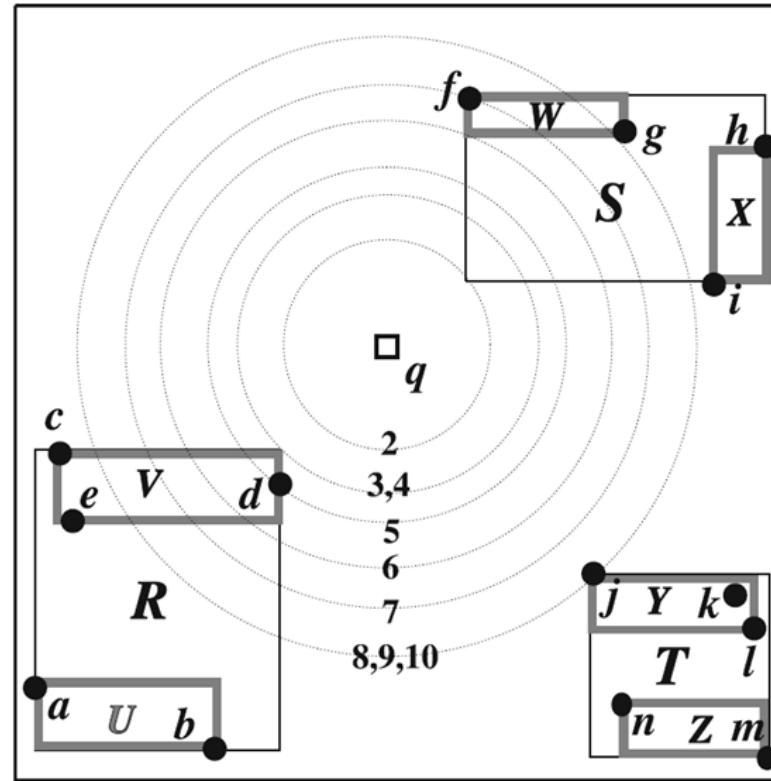
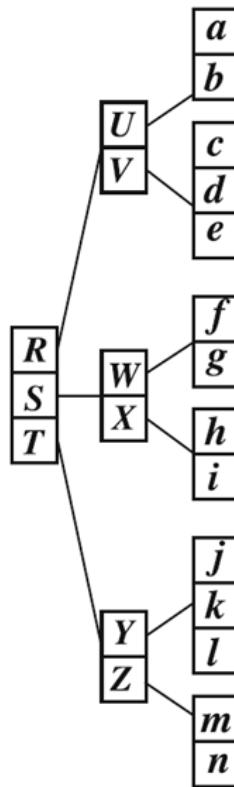
Can be very inefficient in worst case since multiple paths may need to be searched due to overlaps, but works fine in practice.



Another Example R-tree



Nearest Neighbor Query on R-tree



Step	Priority queue	Retrieved NN(s)
1	$\langle S, R, T \rangle$	$\langle \rangle$
2	$\langle R, W, T, X \rangle$	$\langle \rangle$
3	$\langle V, W, T, X, U \rangle$	$\langle \rangle$
4	$\langle d, W, T, X, c, e, U \rangle$	$\langle \rangle$

Step 5 finds d as the first NN (using Best First search)



What to do now?

- n **There is no point going through all index types** for all data types
 - | There are hundreds of them
 - | Even each type would have many subtypes
 - ?? E.g., MX-CIF quadtree is one quadtree type among many
 - | Same with other indices
- n Given a data set, when uploading to the 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** mainly when you have large data
 - | **Monitor performance**
 - | **Tune or create other indices**
 - | Your DBMS will have a version of the **create index** command

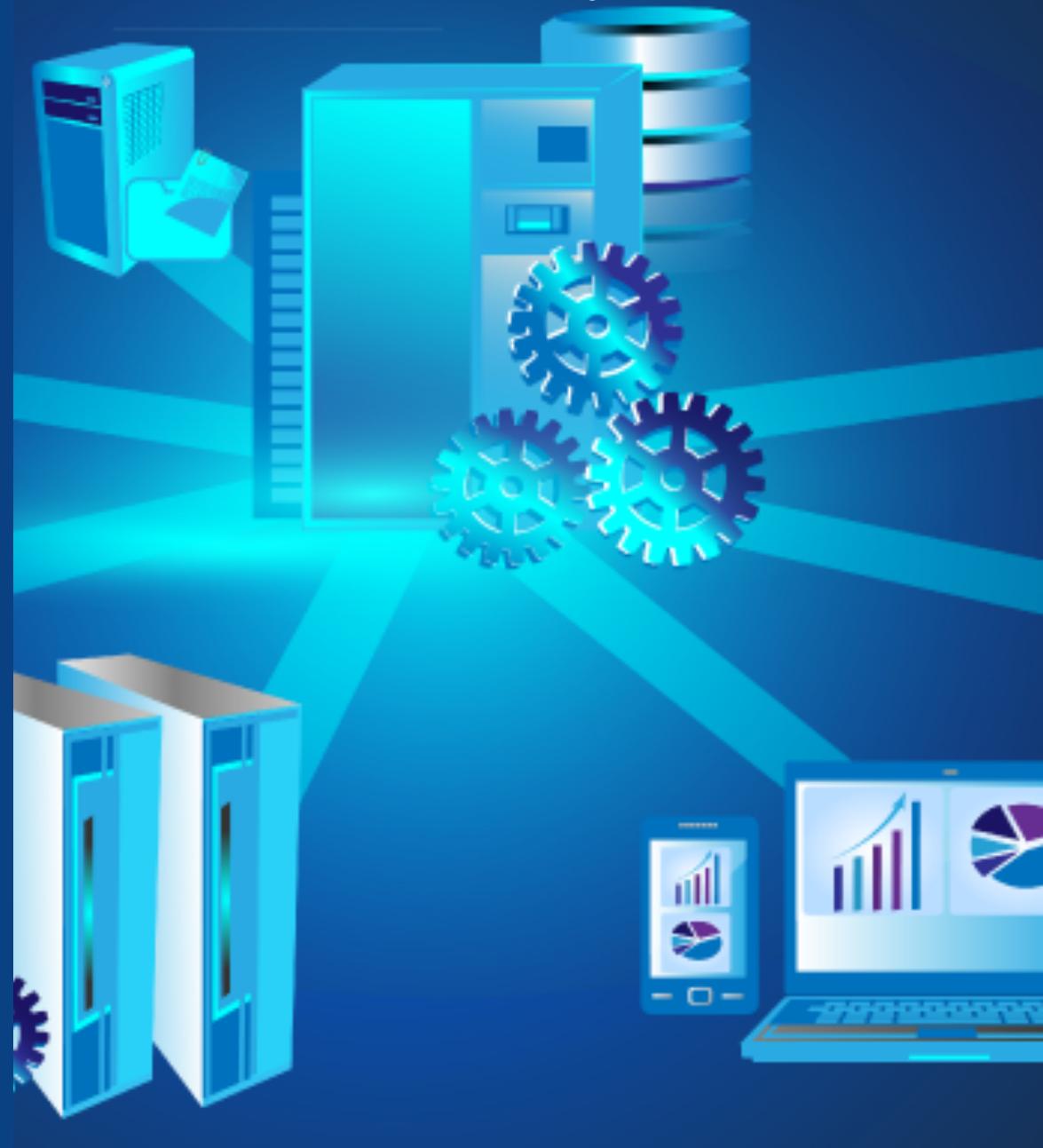


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART II:
TRANSACTIONS

COMP90050 Advanced Database Systems





Database Transactions

In relational DBs unit of work is not one SQL query at a time

Transaction is a unit of work in a database which may have multiple SQL queries bundled together

- A transaction can have many number and type of operations in it
- **Either all happens as a whole or not**, which is the most important factor that makes it an atomic unit of execution
- Transactions ideally have four properties, commonly known as **ACID properties**



ACID Properties

The text "ACID:" is in red, followed by a list of four properties: "Atomicity," "Consistency," "Isolation," and "Durability." Each property is preceded by a red letter and followed by a black letter. The entire list is contained within a yellow diagonal oval.

ACID:

Atomicity,
Consistency,
Isolation,
Durability



Lets list type of actions first...

- **Unprotected actions** - no ACID property, easy to run with weaker guarantees.
- **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 properties guaranteed.**
- **Real actions** - these are real physical actions once performed cannot be undone. In many situations, atomicity is not possible at all (e.g., printing two reports as a single atomic action). So if you hook your DB to a real world action, beware!

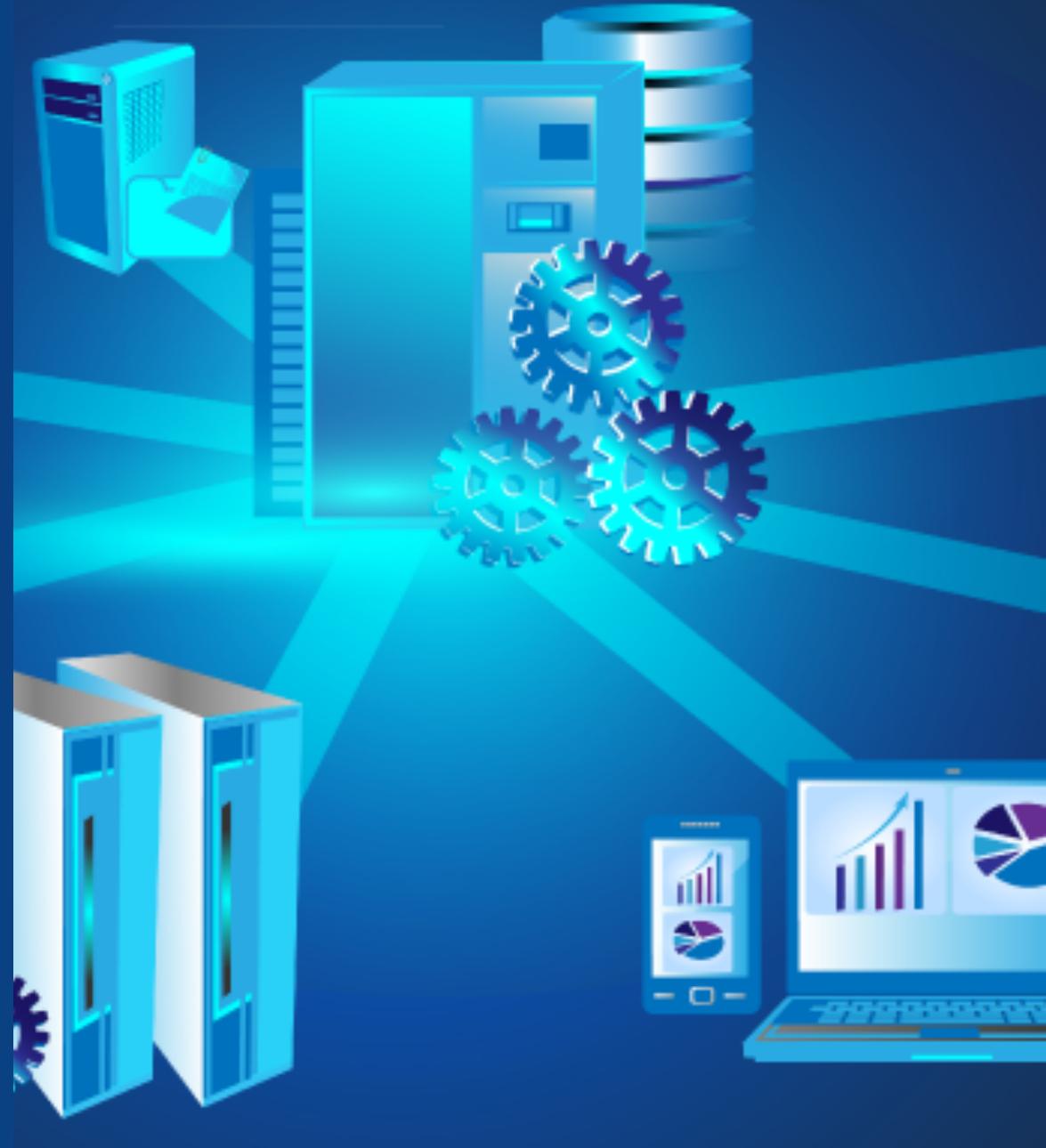


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART II:
TRANSACTIONS

COMP90050 Advanced Database Systems





ACID Properties: 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) then deposits \$100 to another account

...both actions will either happen together or none will happen!



But what if...

Things are **not as easy as it seems**. For example, what if you were to **show the user some data right after each step**?

For that sample transaction:

right after i) but before ii) we show some info:

(i) subtracts \$100 if balance >100

[PRINT]

(ii) deposits \$100 to another account

A failure in step ii) cannot be rolled back easily anymore
cause the user is already informed of the transaction status!



What if...

Transaction A	Transaction B
<i>Initial value of b at bank is 150</i>	<i>Initial value of b at bank is 150</i>
Start Transaction	Start Transaction
	Balance = b.getBalance()
	b.setBalance(Balance - 100)
Balance = b.getBalance()	
commit	Failure/rollback
<i>User in another trans. got 50</i>	<i>Final value of b at bank is 150</i>

This schedule of two transactions is not recoverable.

So we need to look at multiple transactions interacting as well...



But Consistency in ACID next...

Data is in a ‘consistent’ state when a transaction starts and should be in a consistent state 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
- It is not easily computable in general
- Only restricted type of consistency can be guaranteed
- Multiple transactions running concurrently can create inconsistent states as well



For example...

Transaction X

*Initial value of Sem
variable=Fall*

Start Transaction

...

aStudent.setSemester("Wnter")

...

commit

*Any retrieval of this field will
have issues starting this
point*

Solution:

```
Create table  
student_table (  
...  
Sem varchar(10),  
...  
Primary key (...),  
Check (Sem in  
("Fall","Winter","Spring",  
"Summer")))
```

**** check clause will
help in these cases**



ACID Contd.: Isolation

Transactions are executed as if it is the only one in the system .

- 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

- So basically the other transactions should be running in effect as if they came **before or after this transfer transaction ,** or in other words they run in a **schedule** with this transaction that is equal to one of the **serial executions**

BUT THEY DO RUN CONCURRENTLY INTERNALLY!



Here a deposit and a withdrawal transactions are considered: order is not equal to A ? B or B ? A

Transaction A	Transaction B
<i>Initial value of b at bank is 150</i>	<i>Initial value of b at bank is 150</i>
	Start Transaction
Start Transaction	
	Balance = b.getBalance()
Balance = b.getBalance()	
b.setBalance(Balance + 100) commit	b.setBalance(Balance - 100) commit
<i>Final value of b at bank is 50</i>	<i>Final value of b at bank is 50</i>



Why bother with concurrent execution in the first place?

- We could run transactions one after the other, so why we need concurrently running transactions, isolation is guaranteed otherwise...
- One can **reduce waiting time** for customers with concurrency:

Some transactions may continue on another part of the database while others are running for a long time on one part
- **Improved throughput and resource utilization:**

Some transaction can do I/O while another runs on CPU
This and other optimization opportunities increase throughput

So we need concurrent executions where they are
equal to A \sqcap B or B \sqcap A



ACID: Durability

The system should tolerate system failures and any **committed updates should not be lost**:

- This is kind of obvious, no one wants to hear that a bank has lost all of your money due to electricity going down
- But it is easier to say this then achieve it properly



How do we define transactions?

- In programming, origins came from Embedded SQL that people used with C, etc
- Basically write SQL within another language for data retrieval from a DBMS
- Main idea in many languages is the same even today
- We need constructs to be used to define a transaction, separate from the other programming constructs

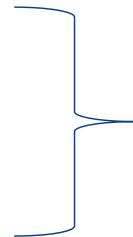


Basic Constructs

```
exec sql BEGIN DECLARE SECTION;  
exec sql END DECLARE SECTION;
```

```
exec sql BEGIN WORK;  
exec sql COMMIT WORK;
```

```
exec sql SELECT ...  
      FROM ...  
      WHERE ...  
      INTO ...
```



Transaction definition



Host variables

Declared in a section enclosed by the **BEGIN DECLARE SECTION** and **END DECLARE SECTION**

While accessing these variables, they are prefixed by a colon or a similar text indicator

The colon is essential to distinguish between host variables and database objects

```
int main()
{
    exec sql BEGIN DECLARE SECTION;
        int ID;
    exec sql END DECLARE SECTION;
    ....
```



Data types

The data **types** supported by a DBMS and a host language can be quite different.

Host variables play a dual role at times

Host **variables** are program variables, declared and manipulated by host language statements, and

They are **used in SQL** to retrieve database data

If there is no host language type corresponding to a DBMS data type, **DBMS automatically converts the data**

Host variable types must be chosen carefully



Error handling

The DBMS reports run-time errors to the applications program through what is called a SQL Communications Area

with a statement such as

WHENEVER...GOTO

tells that we need to do error-handling to process errors returned by the DBMS



Singleton Select and Cursor

A statement can be used to return the data:

- Either as a singleton SELECT: It returns only a single row
- Or uses a cursor to go through more rows if there are more



Example Code

```
int main()
{
    ...
    exec sql BEGIN DECLARE SECTION;
    /* The following variables are used for communicating
       between SQL and C */

    int OrderID;          /* Employee ID (from user) */
    int CustID;           /* Retrieved customer ID */
    char SalesPerson[10] /* Retrieved salesperson name */
    char Status[6];        /* Retrieved order status */

    exec sql END DECLARE SECTION;
    ...
}
```



Code Contd

```
/* Set up error processing */
```

```
exec sql WHENEVER SQLERROR GOTO query_error;
```

```
exec sql WHENEVER NOTFOUND GOTO bad_number;
```



Code Contd

```
/* Prompt the user for order number */

printf ("Enter order number: ");
scanf_s("%d", &OrderID);

/* Execute the SQL query, simple query no transaction definition */

exec sql SELECT CustID, SalesPerson, Status
    FROM Orders
    WHERE OrderID = :OrderID    // ":" indicates to refer to C variable
    INTO :CustID, :SalesPerson, :Status;
```



Code Contd

```
/* Display the results */
```

```
printf ("Customer number: %d\n", CustID);
```

```
printf ("Salesperson: %s\n", SalesPerson);
```

```
printf ("Status: %s\n", Status);
```

```
exit();
```

```
query_error:
```

```
printf ("SQL error: %ld\n", sqlca->sqlcode); exit();
```

// SQL Communications Area

```
bad_number:
```

```
printf ("Invalid order number.\n"); exit(); }
```



How to make it a transaction?

Everything inside **BEGIN WORK** and **COMMIT WORK** is at the same level;

That is, the transaction **will either survive together** with everything (**commit**),

Or it will be rolled back with everything (abort)

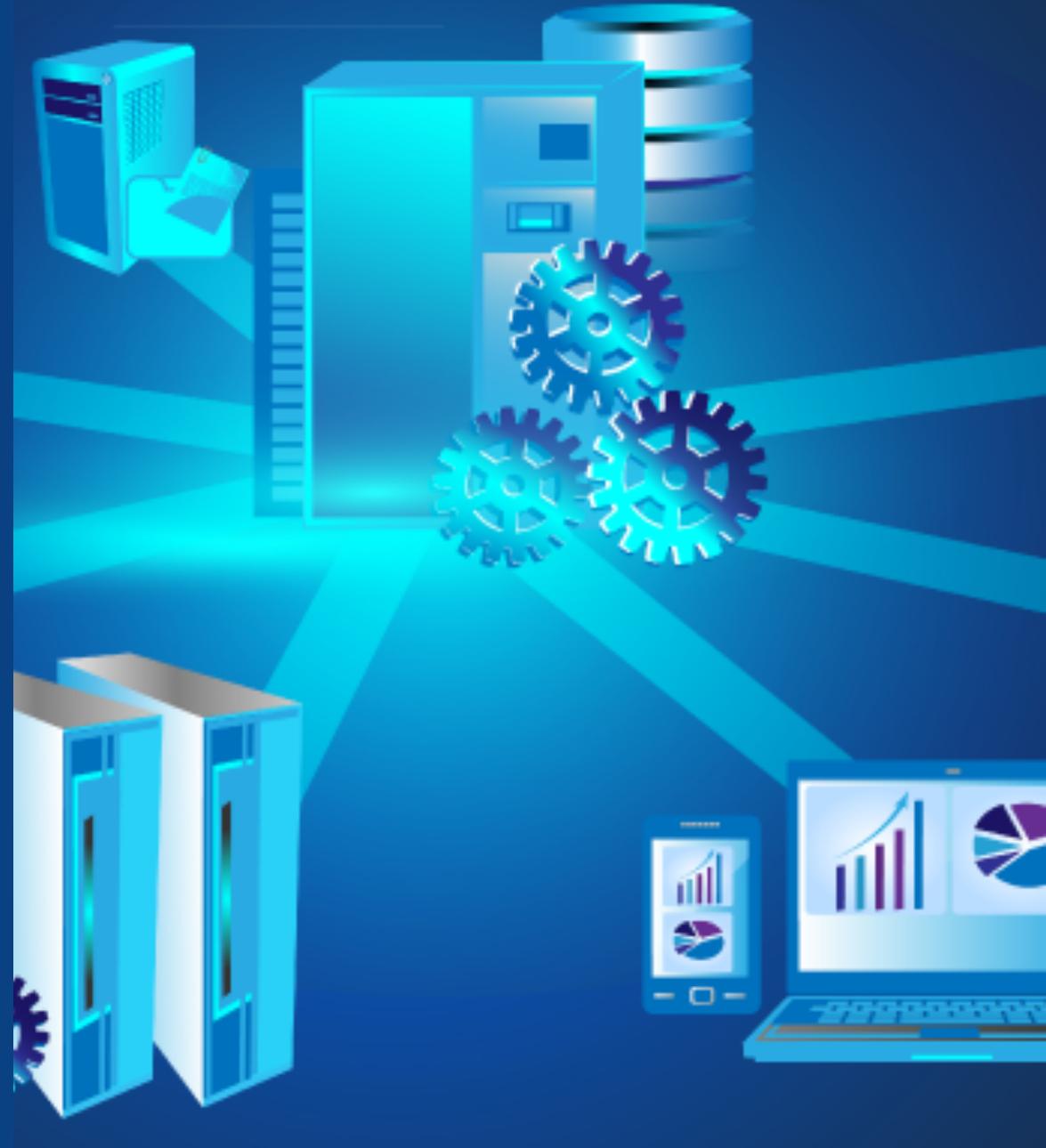


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART II:
TRANSACTIONS CONTD.

COMP90050 Advanced Database Systems





A Simple “Flat” Transaction Example...

```
exec sql BEGIN DECLARE SECTION;
    long AcclId, BranchId, TellerId, delta, AccBalance;
exec sql END DECLARATION;

/* Debit/Credit Transaction*/

DCApplication()
{
    read input msg;
    exec sql BEGIN WORK;
    AccBalance = DodebitCredit(BranchId, TellerId, AcclId,
delta);
    send output msg;
    exec sql COMMIT WORK;
}
```



```
Long DoDebitCredit(long BranchId, long TellerId, long AcctId, long delta) {  
  
    exec sql UPDATE accounts  
    SET AccBalance = AccBalance + :delta  
    WHERE AcctId = :AcctId;  
  
    exec sql SELECT AccBalance INTO :AccBalance  
    FROM accounts WHERE AcctId = :AcctId;  
  
    exec sql UPDATE tellers  
    SET TellerBalance = TellerBalance + :delta  
    WHERE TellerId = :TellerId;  
  
    exec sql UPDATE branches  
    SET BranchBalance = BranchBalance + :delta  
    WHERE BranchId = :BranchId;  
  
    Exec sql INSERT INTO history(TellerId, BranchId, AcctId, delta, time)  
    VALUES( :TellerId, :BranchId, :AcctId, :delta, CURRENT);  
  
    return(AccBalance);  
}
```



Let's include a check on the account balance and refuse any debit that overdraws the account...

...

```
DCAapplication() {  
    read input msg;  
    exec sql BEGIN WORK;  
    AccBalance = DodebitCredit(BranchId, TellerId, AcclId,  
    delta);  
    if (AccBalance < 0 && delta < 0) {  
        exec sql ROLLBACK WORK;  
    }  
    else {  
        send output msg;  
        exec sql COMMIT WORK;  
    }  
}
```

...

Alternative to **Embedded SQL**

- In programming we also do **Dynamic SQL**
- Here **SQL query will be defined at runtime** which is very flexible
- Embedded SQL is more static and does bind things at **compile time**
- Infact there is preprocessing in Embedded SQL to prepare queries for compilation
- This could be good to catch some errors but **did not win over many language vendors**



JDBC: Java Database Connectivity

Model for communicating with the database:

- **Open a connection**
- **Create a statement** object
- **Execute queries** using the statement object to send queries and fetch results
- **Exception mechanism to handle errors**



JDBC Code Example

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try (Connection conn = DriverManager.getConnection(
        "jdbc:oracle:thin: @db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement());
    {
        ... Do Actual Work ....      (queries here)
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```



JDBC Code Contd.

Update to database

```
try {  
    stmt.executeUpdate(  
        "insert into instructor values( '77987', 'Kim', 'Physics', 98000)");  
} catch (SQLException sqle)  
{  
    System.out.println("Could not insert tuple. " + sqle);  
}
```

Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(  
    "select dept_name, avg (salary)  
     from instructor  
     group by dept_name");  
  
while (rset.next()) {  
    System.out.println(rset.getString("dept_name") + " " +  
                      rset.getFloat(2));  
}
```



Transactions in JDBC

By default, each SQL statement is treated as a separate transaction that is committed automatically

- bad idea in general

Can turn off automatic commit on a connection

- `conn.setAutoCommit(false);`

Transactions must then be committed or rolled back explicitly

- `conn.commit();`
- `conn.rollback();`

`conn.setAutoCommit(true)` turns on automatic commit



Flat Transactions

Flat transactions do not model many real applications

They are extremely simple

Real life is more complex

BEGIN WORK

S1: book flight from Melbourne to Singapore

S2: book flight from Singapore to London

S3: book flight from London to Dublin

COMMIT WORK

Problem: if we cannot do any, say S3, we need to undo all...?

If we roll back we need to redo the booking from Melbourne to Singapore
which is a waste for many real scenarios!!!



Limitations: Flat Transaction Example

```
IncreaseSalary()  
{      real percentRaise;  
      receive(percentRaise);  
      exec sql BEGIN WORK;  
          exec SQL UPDATE employee  
              set salary = salary * (1 + :percentRaise)  
      exec sql COMMIT WORK;  
}
```

What is wrong with this transaction being flat?

This can be a **long running transaction!!!**

Any failure requires alot of unnecessary recomputation!!!



Improving with Save Points helps

- A **savepoint is established by invoking the SAVE WORK function** which **causes the system to record** the current state of processing.
- This returns to the application program **a handle that can subsequently be used to refer to that savepoint**. Typically, the handle is a monotonically increasing number.
- The only reason why an application program needs an identifier for a savepoint is that it may later want **to reestablish (return to) that savepoint**.
- To do that, the application invokes the **ROLLBACK WORK function**, but rather than requesting the entire transaction to be aborted, it **passes the number of the savepoint it wants to be restored** .
- As a result, it finds itself **reinstantiated at that very savepoint** .



Flat transaction with Save Points

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)

Action 8

Action 9

SAVE WORK4

Action 10

Action 11

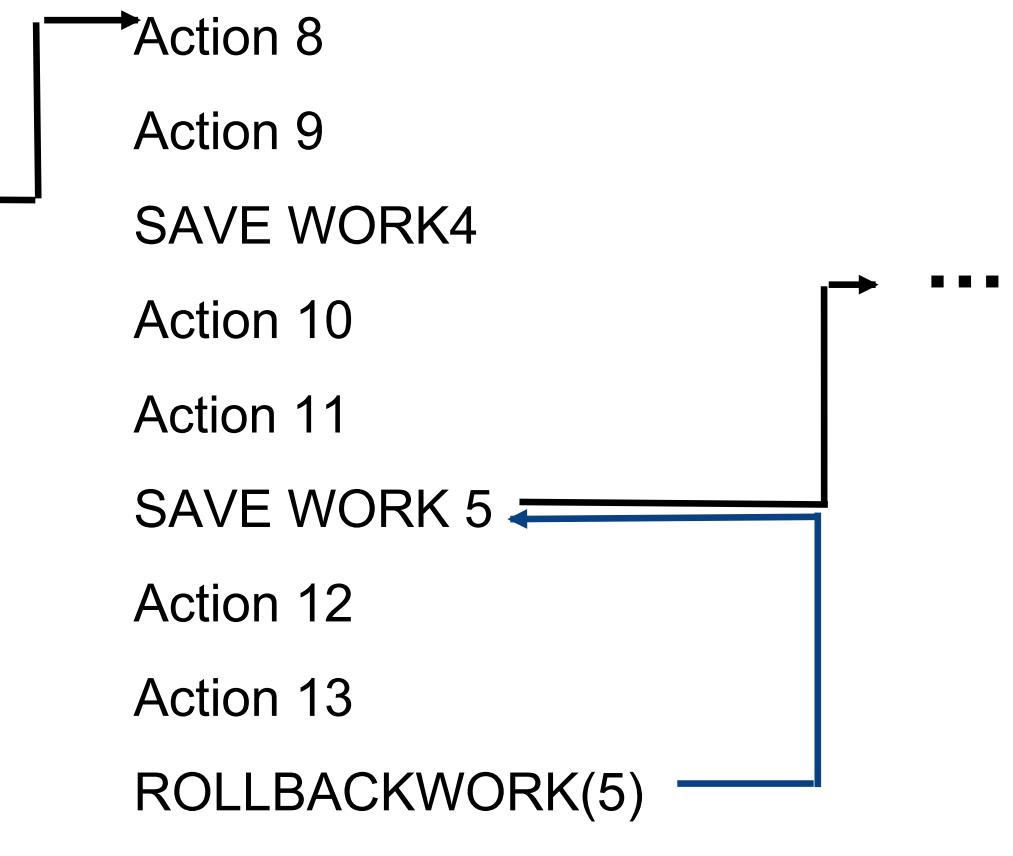
SAVE WORK 5

Action 12

Action 13

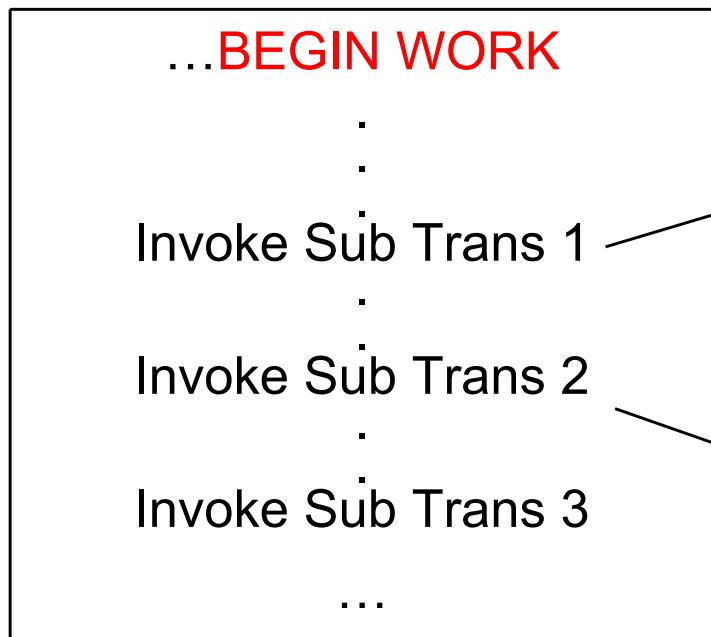
ROLLBACKWORK(5)

...





Nested Transactions are more capable



T1

BEGIN WORK

Invoke Sub Trans 11

Invoke Sub Trans 12

END WORK

T2

BEGIN WORK

Invoke Sub Trans 21

Invoke Sub Trans 22

END WORK



Nested Transaction Rules

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 **subtransaction** makes its **results available only to its parents.**



Nested Transaction Rules

Roll back rule

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

Visibility rule

Changes made by a subtransaction are visible to the parent only when it 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 Management: Monitors

The main function of a Transaction Processing Monitor is to integrate system components and manage resources so that transactions can run:

- TP monitors manage the transfer of data between clients and servers
- Breaks down applications or code into transactions and ensures that all the database(s) are updated properly
- It also takes appropriate actions if any error occurs



TP Monitor Services

Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID properties 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

Context management: E.g. maintaining the sessions etc.

Start/Restart: Note: there is no difference between start and restart in TP based systems



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 properties.

Control communications: 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.



TP Process Structure Types

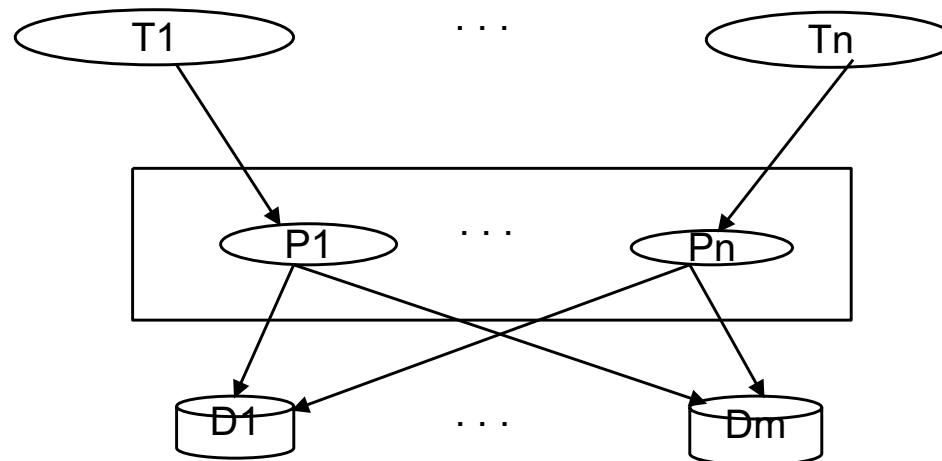
A terminal (client) wants some function to be executed by a server, which in turn needs some data from the database.

For each terminal, there must be a process that eventually gets the input, understands the function request, and makes sure that the function gets executed (or executes the function itself).

Lets see some ways to do this...

TP Process Structure I

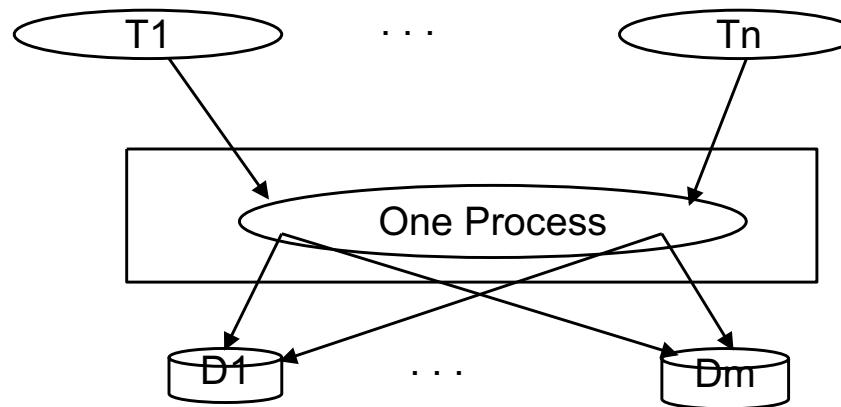
One process per terminal performing all possible requests



Very memory expensive, context switching causes problems too..

TP Process Structure II

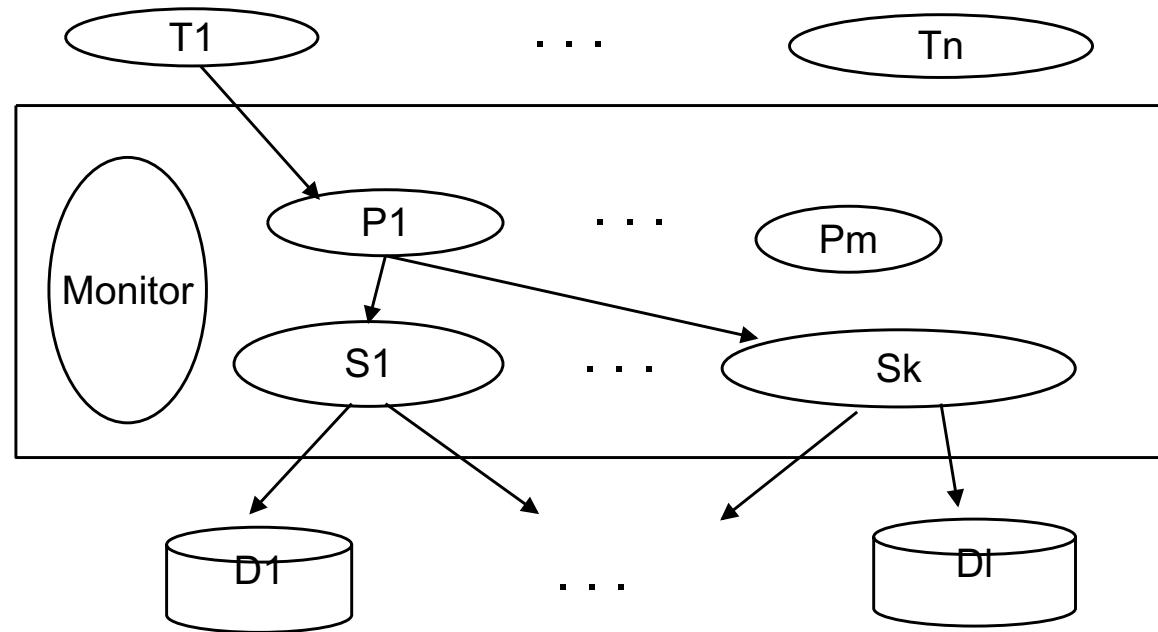
One process for all terminals performing all possible requests



This would work under a multithreaded environment but
**cannot do proper parallel processing, one error leads
to large scale problems, not really distributed and
rather monolithic**

TP Process Structure III

Multiple communication processes and servers



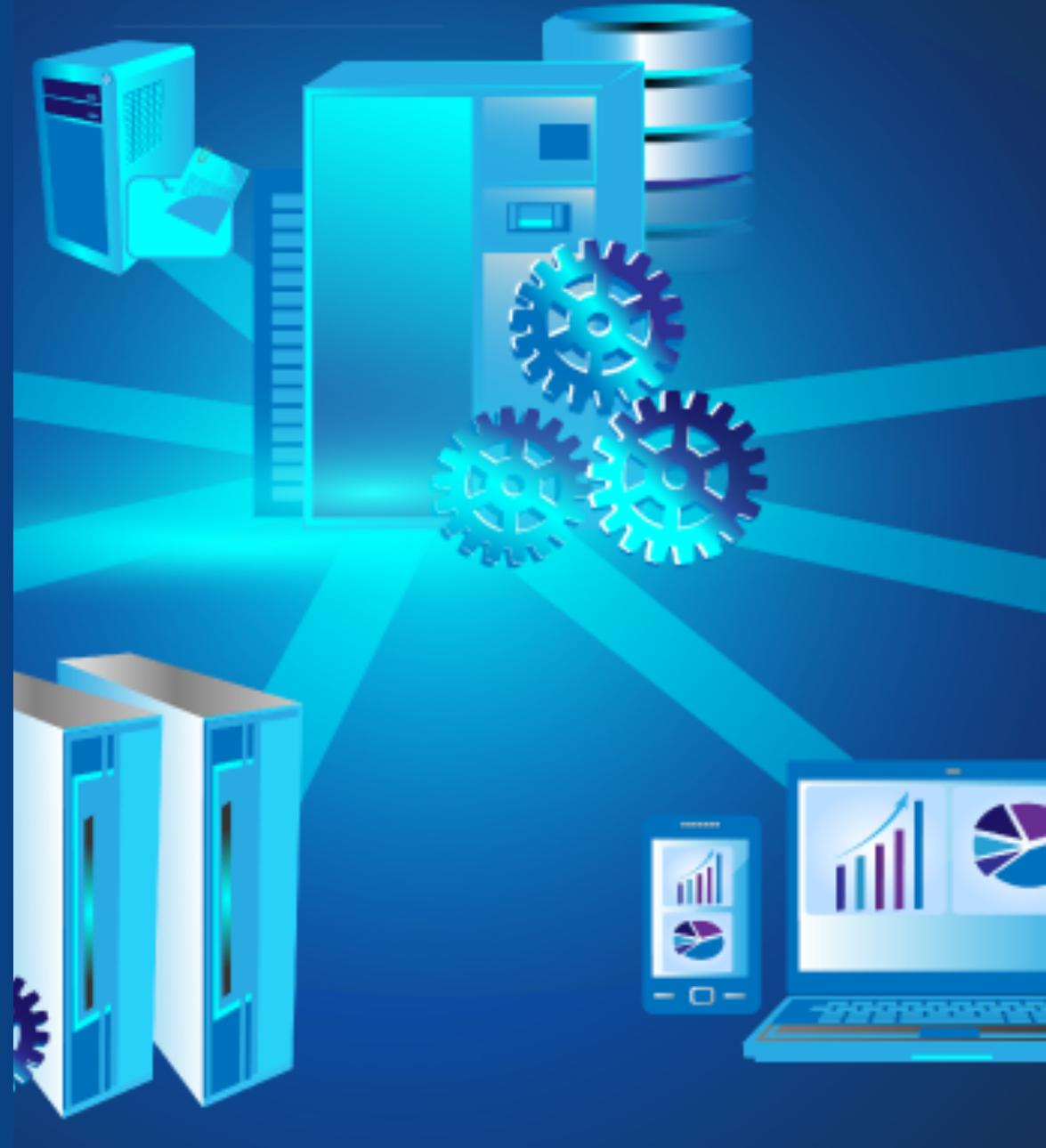


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART III:
CONCURRENCY CONTROL

COMP90050 Advanced Database Systems





Lets Revisit Issues Related to Uncontrolled Concurrent Access

Shared counter = 100; // initial value

Task1/Trans/Process/Thread
counter = counter +10;

Task2/Trans/Process/Thread
counter = counter +30;

Question: Task1 and Task2 are **running concurrently**.

What are the possible values of counter after the end of Task1 and Task2?

- a) counter == 110;
- b) counter == 130;
- c) counter == 140;

Answer: For correct execution we need to impose **exclusive access to the shared variable counter** by Task1 and Task2.



How can we get any of those three?

Shared counter = 100;

Task1/Trans/Process/Thread
counter = counter +10;

Task2/Trans/Process/Thread
counter = counter +30;

Task1 and Task2 run concurrently. What are the possible values of counter after the end of Task1 and Task2?

a) counter == 110

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T2: Writes counter == 100+30

T1: Writes counter == 100+10

b) counter == 130

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T1: Writes counter == 100+10

T2: Writes counter == 100+30

c) counter == 140;

Sequence of actions

T1: Reads counter == 100

T1: Writes counter == 100+10

T2: Reads counter == 110

T2: Writes counter == 110+30

Time



Concurrency Control: Mechanisms for Proper Concurrent Access

Is needed to preserve consistency of the data for some number of tasks to work on the same data, i.e. concurrently.

Some simple concurrency control approaches:

- Dekker's algorithm (write some code to guarantee exclusive access) - needs almost no hardware support, but the code turns out to be complicated to implement especially for more than two transactions/processes
- OS supported primitives (through interrupt calls) - expensive, independent of number of processes
- Spin locks (using atomic lock/unlock instructions) – common



Implementation of exclusive access: Dekker's algorithm

```
int turn = 0 ; int wants[2];           // both should be initially 0
...
while (true) {
    wants[i] = true;                  // claim desire
    while (wants[j]) {
        if (turn == j) {
            wants[i] = false;          // withdraw intention
            while (turn == j);
            wants[i] = true;           // wait and reassert
        }
    }
    counter = counter + 1;             // resource we want mutex on
    turn = j;                         // assign turn
    wants[i] = false;
}
...
```



Implementation of exclusive access

- Dekker's algorithm
 - needs almost no hardware support although it needs atomic reads and writes to main memory
 - the code is very complicated to implement especially if more than two transactions/process are involved
 - takes storage space
 - uses busy waiting
 - efficient if the lock contention is low (that is frequency of access to the locks)



Using OS for exclusive access

- 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 sometimes



Implementation of exclusive access with Spin Locks

Spin Locks

- Executed using **atomic machine instructions** such as test and set or swap
- Need **some 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

- **uses busy waiting**
- algorithm does not depend on number of processes
- very efficient for low lock contentions - many DBMSs systems use them



Test and set as spin lock

testAndSet(int *lock)

{

/* the following is executed atomically, memory bus can be locked
for up to two cycles (one for read and for writing)*/

if (*lock == 1) {*lock = 0; return (true)} else return (false);

}

int lock = 1; % initial value

T1

```
/*acquire lock*/  
while (!testAndSet( &lock );  
    /*lock granted*/  
    //exclusive access for T1;  
counter = counter+1;  
    /* release lock*/  
lock = 1;
```

T2

```
/*acquire lock*/  
while (!testAndSet( &lock );  
    /*lock granted*/  
    //exclusive access for T2;  
counter = counter+1;  
    /* release lock*/  
lock = 1;
```



Another one: Compare and swap

... again on a similar example:

```
temp = counter +1;  
  
counter = temp;
```

... but now we use the atomic operation Compare and swap

```
boolean CS(int *cell, int *old, int *new)  
  
{/* the following is executed atomically*/  
  
if (*cell == *old) { *cell = *new; return TRUE; }  
  
else { *old = *cell; return FALSE; }  
  
}
```

temp = counter;

do

new = temp + 1;

while(!CS(&counter,&temp,&new);

Using Compare and swap in
spin lock for exclusive access



Spin locks Contd.

We can implement a more generic lock mechanism using these constructs we saw.

We then just have to use the functions of that lock mechanism, named like the ones below, for exclusive access to data:

Lock(var);

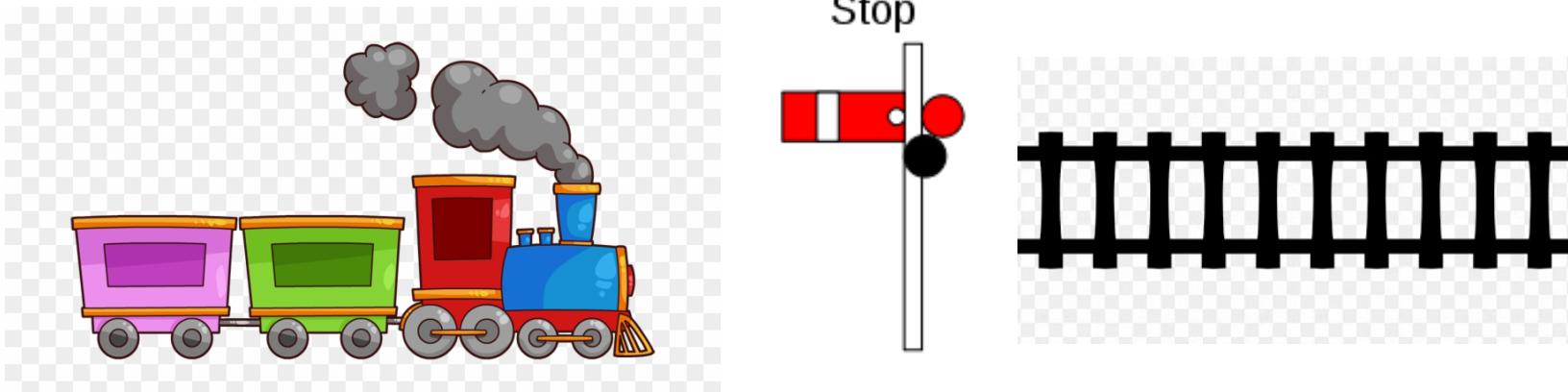
Unlock(var);



Semaphores generalize spin locks

Semaphores derive from the corresponding mechanism used for trains: **a train may proceed through a section of track only if the semaphore is clear**. Once the train passes, the semaphore is set until the train exits that section of track.

Try to **Get track** , wait if track not clear



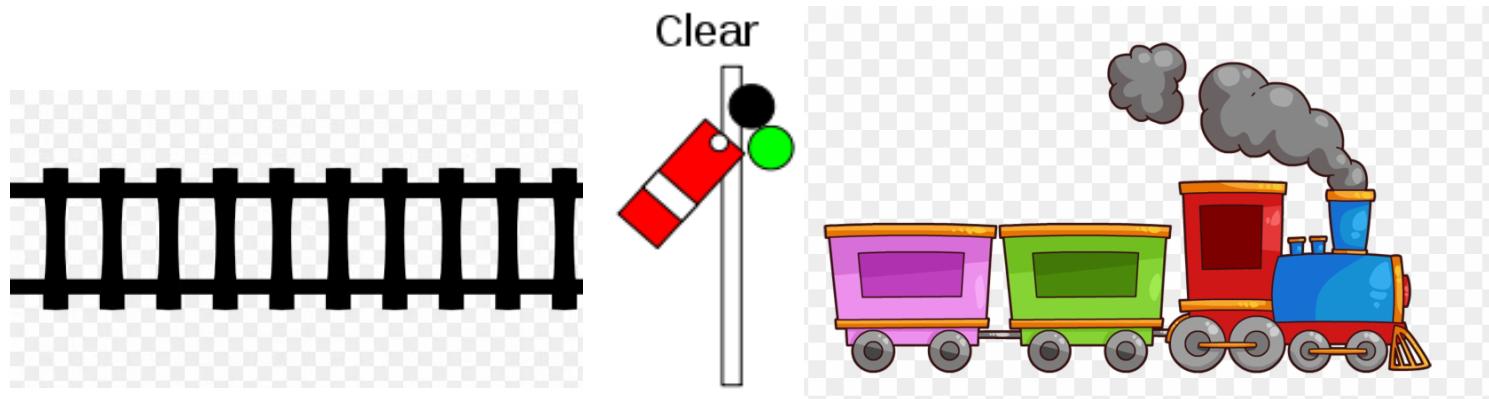
If Get track was successful, use it (no other train will be able to use it at the same time)!



You need to explicitly release...

Semaphores derive from the corresponding mechanism used for trains: a train may proceed through a section of track only if the semaphore is clear. Once the train passes, **the semaphore is set until the train exits that section of track** .

Release track after using
(so that others can use it)





Semaphores in computers

- Computer semaphores are similar
- They have a “**get**” routine that acquires the semaphore, basically **locks**
- Generally allow waiting in a queue for multiple requests until it is free
- And a “**give**” routine that returns the semaphore to the free state, signalling (waking up) a waiting process (perhaps from a queue), i.e. **unlocks**



Semaphores Contd.

- Needs a pointer to a queue of 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 has a linked list of waiting processes.
- The process owning the semaphore is at the top of this list.
- After usage, the owner process wakes up the oldest process in the queue (first in, first out scheduler)

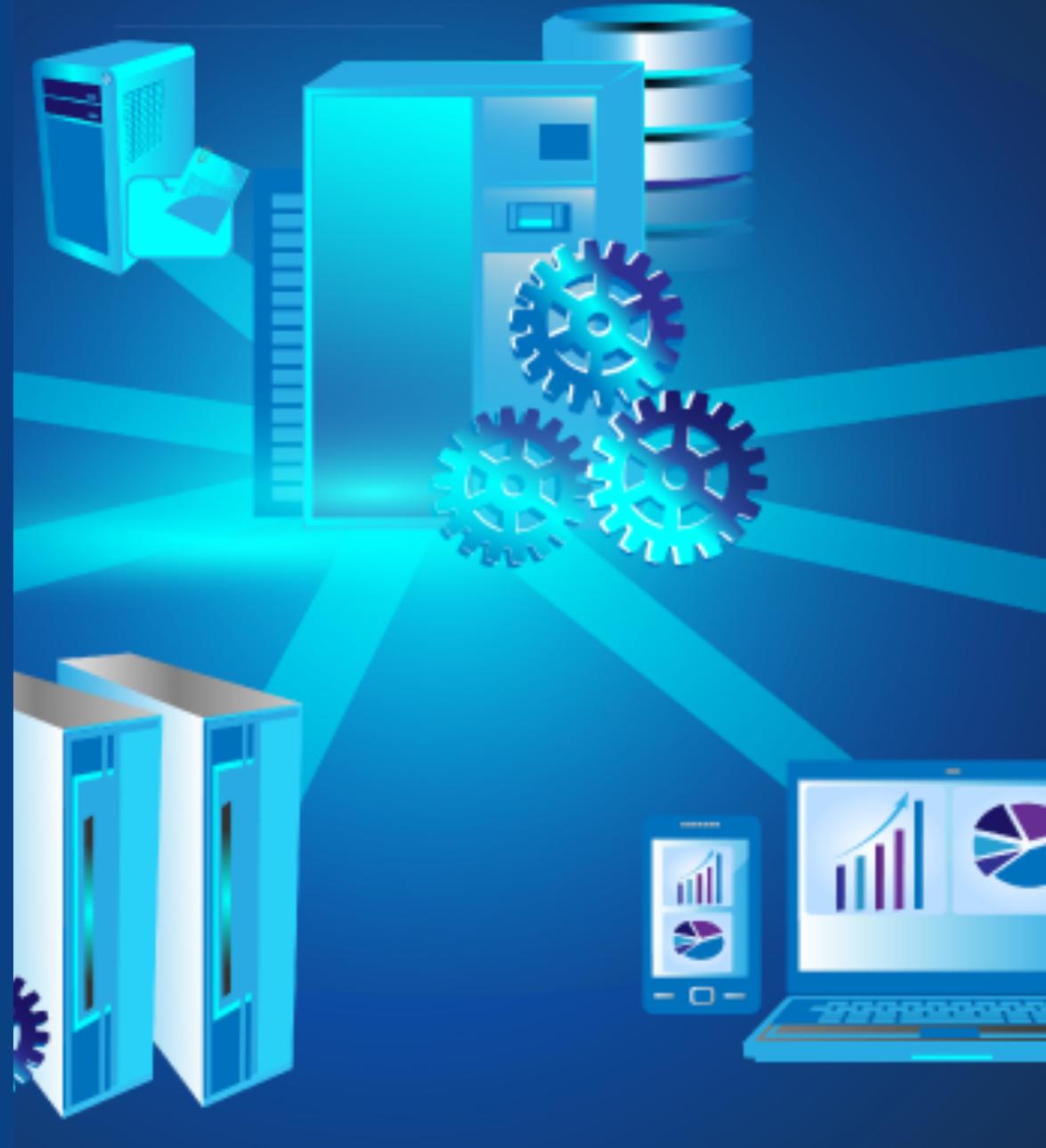


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART III
CONCURRENCY CONTROL
CONTD.

COMP90050 Advanced Database Systems





A simple implementation for semaphores

```
/* Xsem_get acquires access to an exclusive semaphore for the calling process. */  
/* The program adds a pointer to the calling process to the head of the semaphore list and */  
/* moves the current list head (typically NULL) to MyPCBP()->sem_wait. */  
/* This adds the calling process to the semaphore wait list if there is one. */  
/* If the semaphore was not free the process waits for a wakeup. */  
/* The semaphore is owned by the caller when this routine returns. */  
  
void Xsem_get(xsemaphore * sem) /* parameter is pointer to an X semaphore */  
{PCB * new = MyPCBP(); /* new will be the new value of the sem (me) */  
PCB * old = NULL; /* guess at old queue head = NULL */  
do /* loop until process is on queue (until CS is successful) */  
    new->sem_wait = old; /* move current queue head to my sem_wait */  
    while ( ! CS(sem, &old, &new)); /* replace queue head with pointer to me */  
    if (old != NULL) /* if queue was not null then I'm not yet the */  
        wait(); /* owner and must wait for a wakeup from owner. */  
    return; /* semaphore acquired, return to caller */  
}; /* */
```



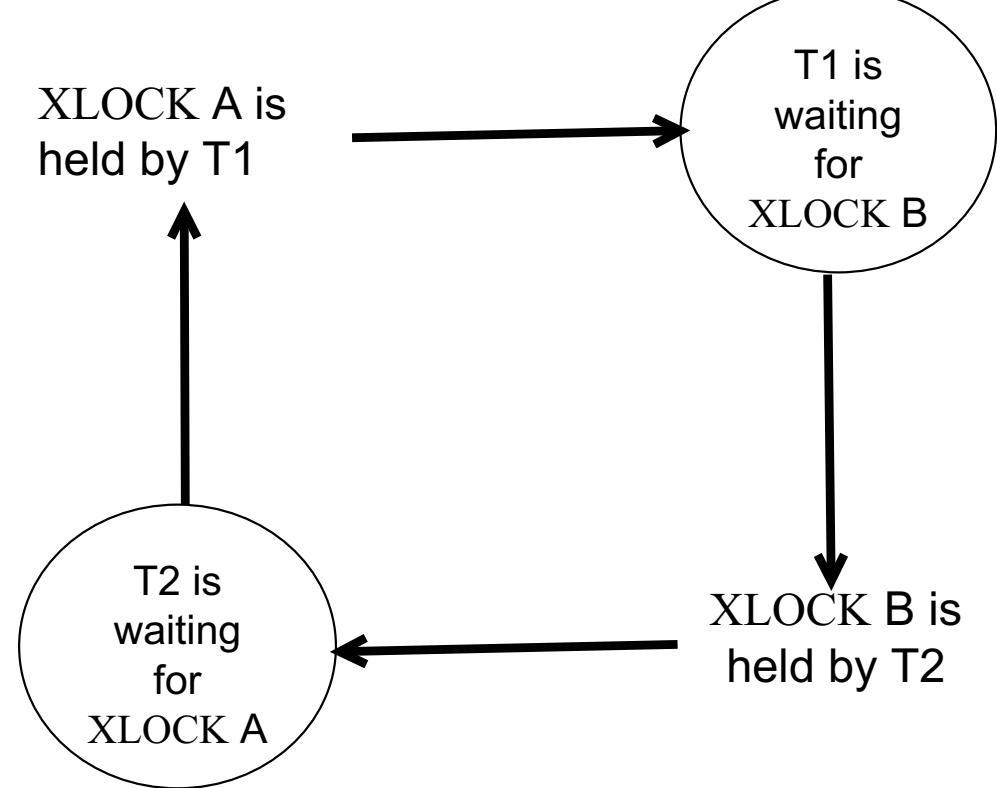
A simple implementation for semaphores

```
/* Code to release access to an exclusive semaphore for the calling process. */  
/* Program removes the process from the tail of the semaphore list and if the list is not empty, */  
/*      wakes up the previous process (FIFO scheduling) */  
void Xsem_give(xsemaphore * sem) /* parameter points to an exclusive semaphore */  
{PCB * new = NULL; /* guess that the new wait list is null */  
 PCB * old = MyPCBP(); /* guess that the old list is my process alone */  
if (CS(sem,&old,&new)) return; /* if guessed right, return; */  
while (old->sem_wait != MyPCBP()) /* if guessed wrong, old is now a non-null wait list */  
    { old = old->sem_wait; } /* advance to the end of the list */  
old->sem_wait = NULL; /* remove me from the end of the list */  
wakeup(old); /* wake up the process at the end of the list */  
/* (note there is one, since list was not null) */  
}; /* */
```

Locks commonly lead to Deadlocks

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

T1	T2
Begin	Begin
XLOCK(A)	XLOCK(B)
Write to A	Write to B
XLOCK(B)	XLOCK(A)
Write B	Write A
Unlock (A)	Unlock (A)
Unlock (B)	Unlock (B)
end	end





Deadlocks Contd.

Solutions:

- **Have enough resources so that no waiting occurs** – not practical in general.
- Do not allow a process to wait long, **simply rollback after a certain time**. This can create live locks which are at times 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.

Deadlocks Contd.

Pa: Holds resources at level i and request resource at level j which are held by Pb. $j > i$

Pq: Holds resources at level g and request resource at level l which are held by Pd. $l > g$

Pb: Holds resources at level j and request resource at level k which are held by Pc. $k > j$

Pc: Holds resources at level k and request resource at level l which are held by Pd $l > k$

Pd: Holds resources at level l and is currently running.

$l > k > j > i \text{ and } l > g$.

We cannot have loops. The graph **can be a tree or a linear chain** and hence cannot have cycles.



Deadlocks Contd.

- **Pre-declare all necessary resources** and allocate in a single request.
- **Periodically check the 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, etc) have chosen this approach.**



Going back to Isolation Property

- Now we know locks how do we build on top of this to make sure concurrent transactions do not cause problems.
- Isolation ensures that concurrent transactions leaves the database in the same state as if the transactions were executed separately.
- Isolation guarantees consistency, provided each transaction itself is consistent.



Isolation – Expected output

Shared counter = 100;

Task1/Trans/Process/Thread
counter = counter +10;

Task2/Trans/Process/Thread
counter = counter +30;

a) counter == 110
Sequence of actions

T1: Reads counter == 100
T2: Reads counter == 100
T2: Writes counter == 100+30
T1: Writes counter == 100+10

b) counter == 130
Sequence of actions

T1: Reads counter == 100
T2: Reads counter == 100
T1: Writes counter == 100+10
T2: Writes counter == 100+30

c) counter == 140;
Sequence of actions

T1: Reads counter == 100
T1: Writes counter == 100+10
T2: Reads counter == 110
T2: Writes counter == 110+30

Alternatively, T2 executing before T1 will also be fine



Recall that we want Concurrency!

We can achieve isolation by **sequentially processing each transaction**

This works assuming each transaction works but **not efficient**:

- Provides **poor response times**
- **Wastes useful resources** by not allowing transactions to use them

We need to **run transactions concurrently** but:

- Concurrent execution **should still make sense/work**
- Concurrent execution **should not cause application programs (transactions) to malfunction**
- Concurrent execution should **not have lower throughput or bad response times than serial execution**



What to watch out for in Concurrency in DBs

Given a set of transactions, **how can we determine which set of actions will lead to problems**

We saw that **working on the same set of objects concurrently is problematic but there is more to the story than this**

Concurrency control is about finding/addressing all the associated problems

For this we **first need to define what actions and what objects in multiple concurrent transactions can cause a problem**

In short **we define a Dependency model between transactions first**

Dependencies Contd.

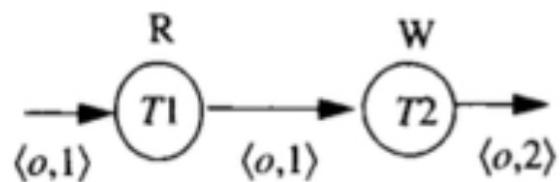
If the inputs and outputs of the concurrent transactions are not disjoint, the following dependencies are seen as important:

The transaction execution sequences

$T_1 \text{ READ } \langle o,1 \rangle$

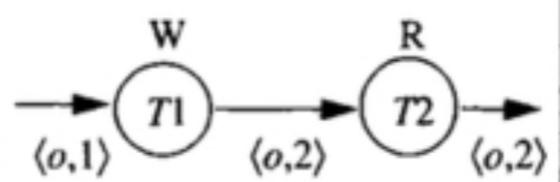
$T_2 \text{ WRITE } \langle o,2 \rangle$

The dependency graphs



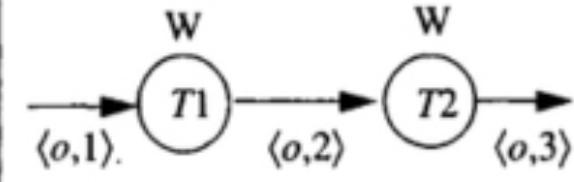
$T_1 \text{ WRITE } \langle o,2 \rangle$

$T_2 \text{ READ } \langle o,2 \rangle$



$T_1 \text{ WRITE } \langle o,2 \rangle$

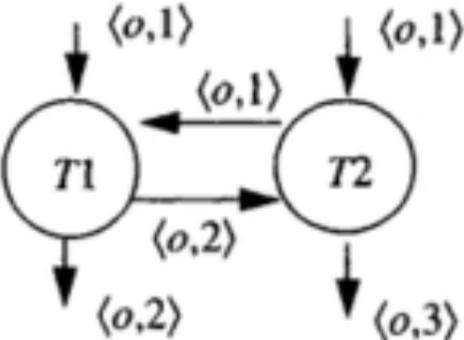
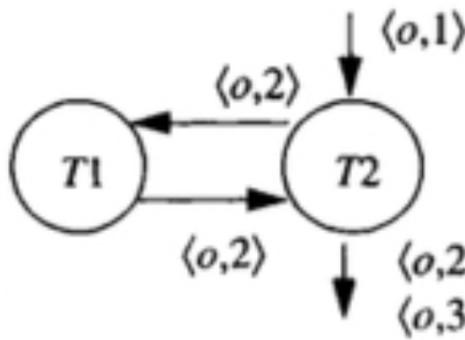
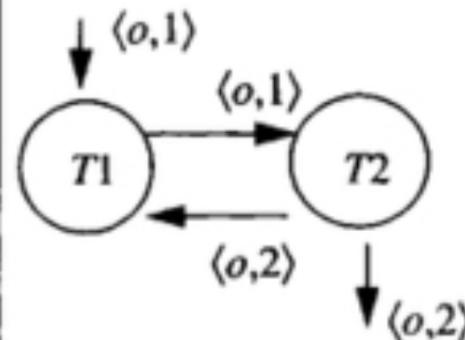
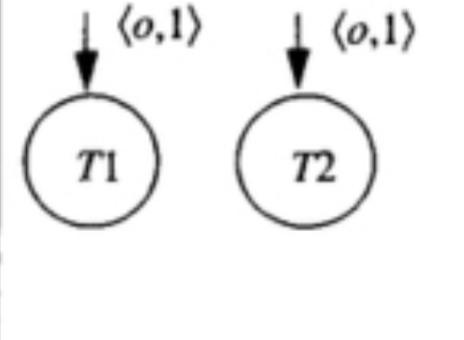
$T_2 \text{ WRITE } \langle o,3 \rangle$



Create a dependency graph first!

Observation: Read-Read dependency does not affect Isolation !!!

Possible dependencies

Lost Update	Dirty Read	Unrepeatable Read	OK
$T_2 \text{ READ } \langle o,1 \rangle$ $T_1 \text{ WRITE } \langle o,2 \rangle$ $T_2 \text{ WRITE } \langle o,3 \rangle$	$T_2 \text{ WRITE } \langle o,2 \rangle$ $T_1 \text{ READ } \langle o,2 \rangle$ $T_2 \text{ WRITE } \langle o,3 \rangle$	$T_1 \text{ READ } \langle o,1 \rangle$ $T_2 \text{ WRITE } \langle o,2 \rangle$ $T_1 \text{ READ } \langle o,2 \rangle$	$T_1 \text{ READ } \langle o,1 \rangle$ $T_2 \text{ READ } \langle o,1 \rangle$ $T_1 \text{ READ } \langle o,1 \rangle$
			

Different versions/names of the same graph are found in the literature:
 Precedence graph, conflict graph, serializability graph...

Dependencies Contd.

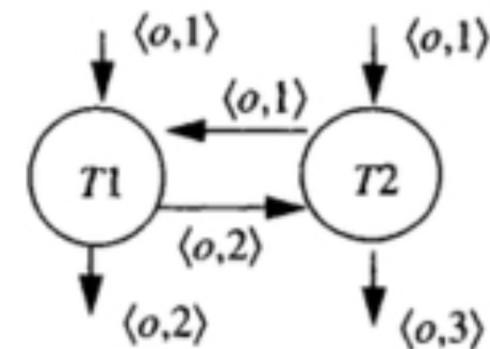
When dependency graph **has cycles then there is a violation** of isolation and a possibility of inconsistency

Lost Update

T_2 READ $\langle o,1 \rangle$

T_1 WRITE $\langle o,2 \rangle$

T_2 WRITE $\langle o,3 \rangle$

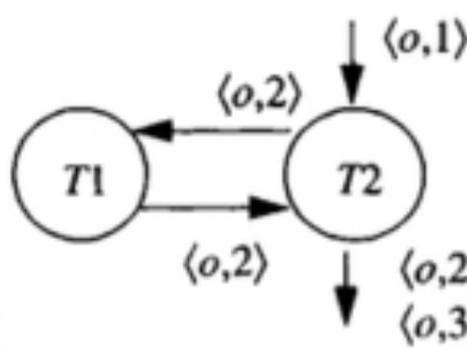


Dirty Read

T_2 WRITE $\langle o,2 \rangle$

T_1 READ $\langle o,2 \rangle$

T_2 WRITE $\langle o,3 \rangle$

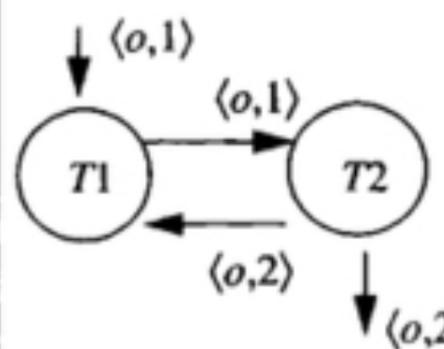


Unrepeatable Read

T_1 READ $\langle o,1 \rangle$

T_2 WRITE $\langle o,2 \rangle$

T_1 READ $\langle o,2 \rangle$

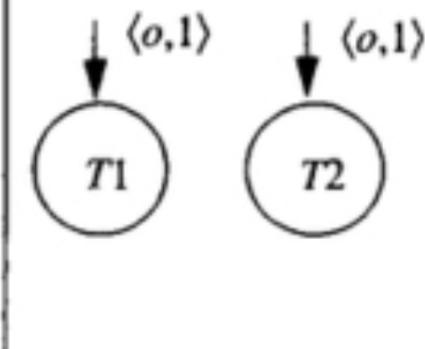


OK

T_1 READ $\langle o,1 \rangle$

T_2 READ $\langle o,1 \rangle$

T_1 READ $\langle o,1 \rangle$



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

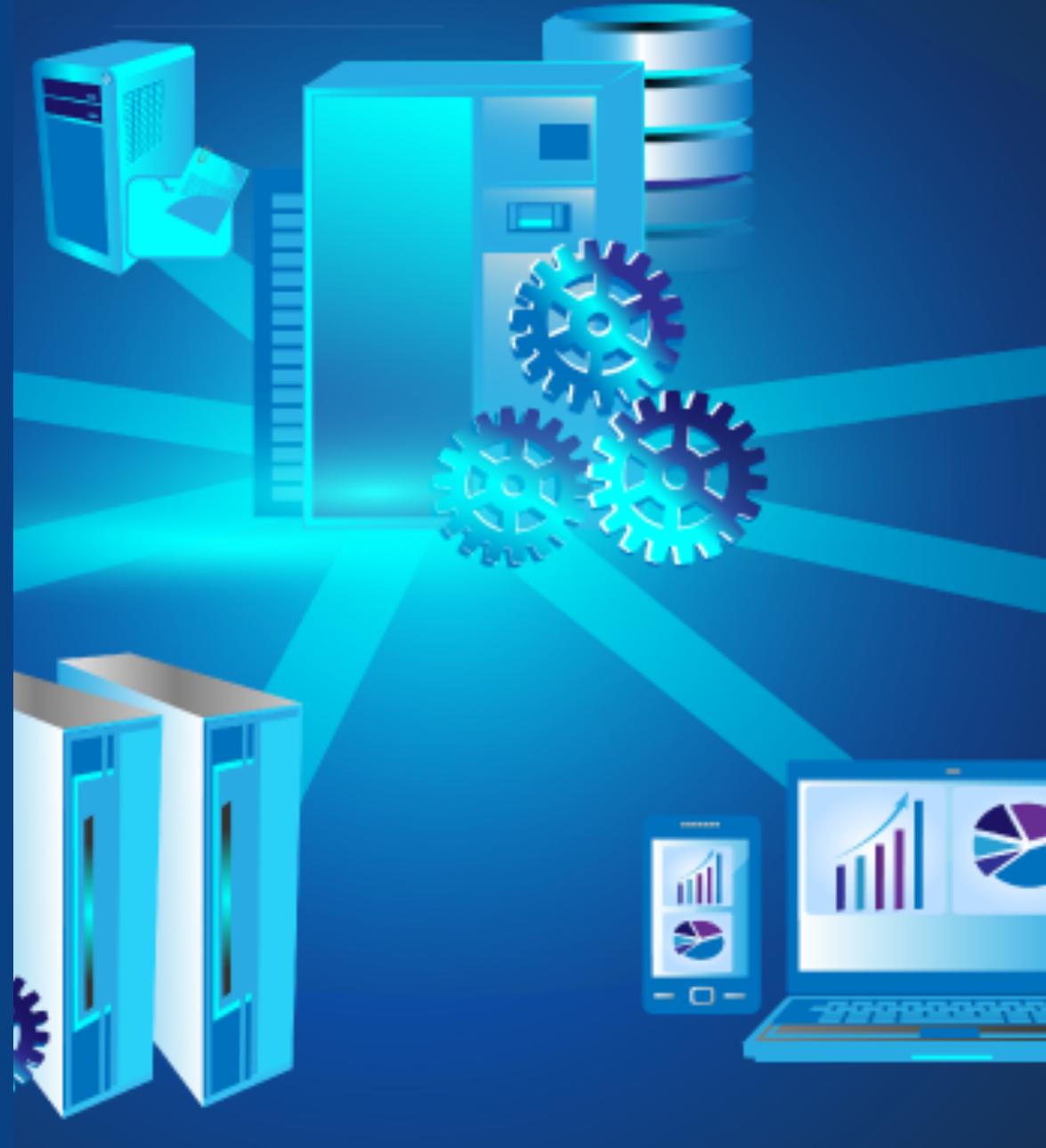


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART III:
CONCURRENCY CONTROL
CONTD.

COMP90050 Advanced Database Systems



Dependency Graph

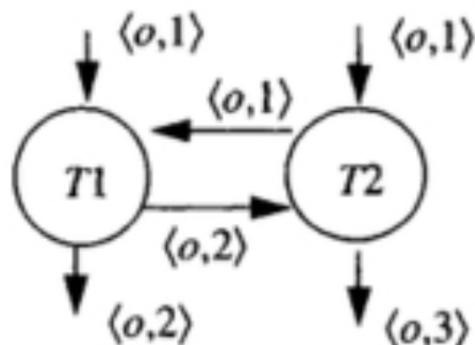
When **dependency graph has cycles** then there is a **violation** of isolation and a possibility of inconsistency

Lost Update

T_2 READ $\langle o,1 \rangle$

T_1 WRITE $\langle o,2 \rangle$

T_2 WRITE $\langle o,3 \rangle$

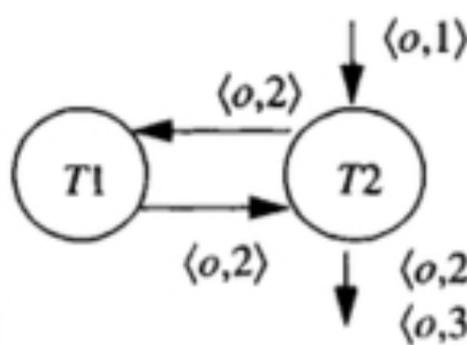


Dirty Read

T_2 WRITE $\langle o,2 \rangle$

T_1 READ $\langle o,2 \rangle$

T_2 WRITE $\langle o,3 \rangle$

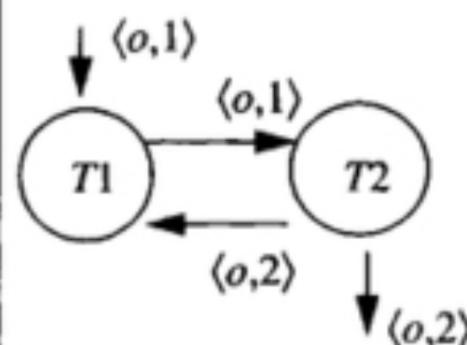


Unrepeatable Read

T_1 READ $\langle o,1 \rangle$

T_2 WRITE $\langle o,2 \rangle$

T_1 READ $\langle o,2 \rangle$

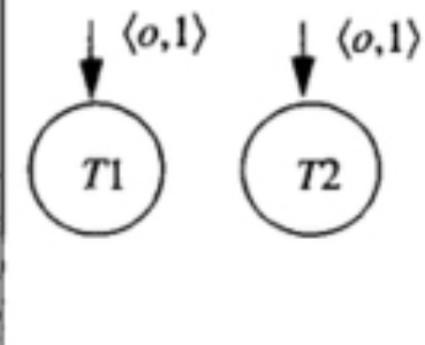


OK

T_1 READ $\langle o,1 \rangle$

T_2 READ $\langle o,1 \rangle$

T_1 READ $\langle o,1 \rangle$



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



Formally defining dependencies

Given a set of transactions, how can we determine which transaction depends on which other transaction?

Dependency model formalization

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 , Transaction T_j has no dependency on any transaction T_i if:

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

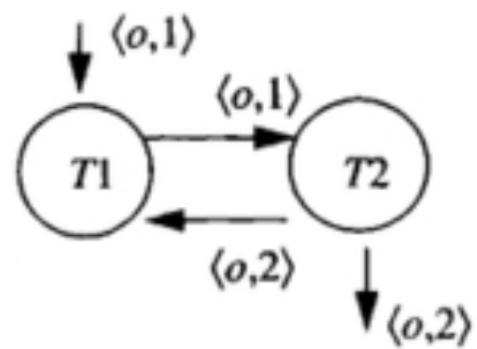
Note: This may not be planned ahead as in many situations inputs and outputs may not be known ahead of time

Dependency Graph

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

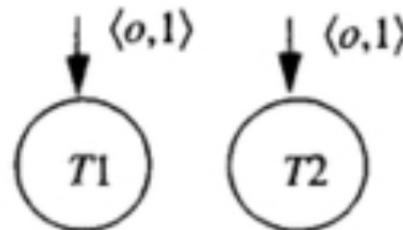
Unrepeatable Read

$T1$	READ	$\langle o,1 \rangle$
$T2$	WRITE	$\langle o,2 \rangle$
$T1$	READ	$\langle o,2 \rangle$



OK

$T1$	READ	$\langle o,1 \rangle$
$T2$	READ	$\langle o,1 \rangle$
$T1$	READ	$\langle o,1 \rangle$



O₂ intersects with I₁

**O₂ DOES NOT intersect with I/O₁
Opposite is also true**



Formal definitions contd.

Let **H** be a history sequence of tuples of the form **(T, action, object)**, E.g.,
 $H_1 = \langle (T_1, R, O_1), (T_2, W, O_5), \dots \rangle$

Let **T1** and **T2** are transactions in **H**. If T1 performs an action on an object O, then T2 performs an action on the same O, and **there is no write action in between by another transaction on O – T2 depends on T1**

Thus, the dependency of T2 on T1 (T_1, O, T_2) exists in history H if there are i and j such that $i < j$, $H[i]$ involves action a_1 on O by T1, (i.e., $H[i] = (T_1, a_1, O)$) and $H[j]$ involves action a_2 on O by T2 (i.e., $H[j] = (T_2, a_2, O)$) and **there are no other $H[k] = (T', WRITE, O)$ for $i < k < j$**

DEP(H) = { $(T_i, O, T_j) \mid T_j \text{ depends on } T_i$ } or an alternate representation:
Dependency graph: Transactions are nodes, and objects label the edges from the node T_i to T_j if (T_i, O, T_j) is in $\text{DEP}(H)$



Note that dependency actions are only...

We focus on the dependency in three scenarios

- $a_1 = \text{WRITE} \ \& \ a_2 = \text{WRITE};$
- $a_1 = \text{WRITE} \ \& \ a_2 = \text{READ};$
- $a_1 = \text{READ} \ \& \ a_2 = \text{WRITE};$

...again no read and reads!



Equivalence of dependencies

Given two histories H1 and H2, H1 and H2 are equivalent if $\text{DEP(H1)} = \text{DEP(H2)}$

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

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

$\text{DEP(H1)} = \{<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> \}$

$H2 = <(T1,R,O1), (T3,W,O1), (T3,W,O2),(T4,R,O2),(T1,W,O3), (T2, W, O5), (T5,R,O3), (T5,R,O4), (T6,W,O4)>$

$\text{DEP(H2)} = \{<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> \}$

Thus: $\text{DEP(H1)} = \text{DEP(H2)}$

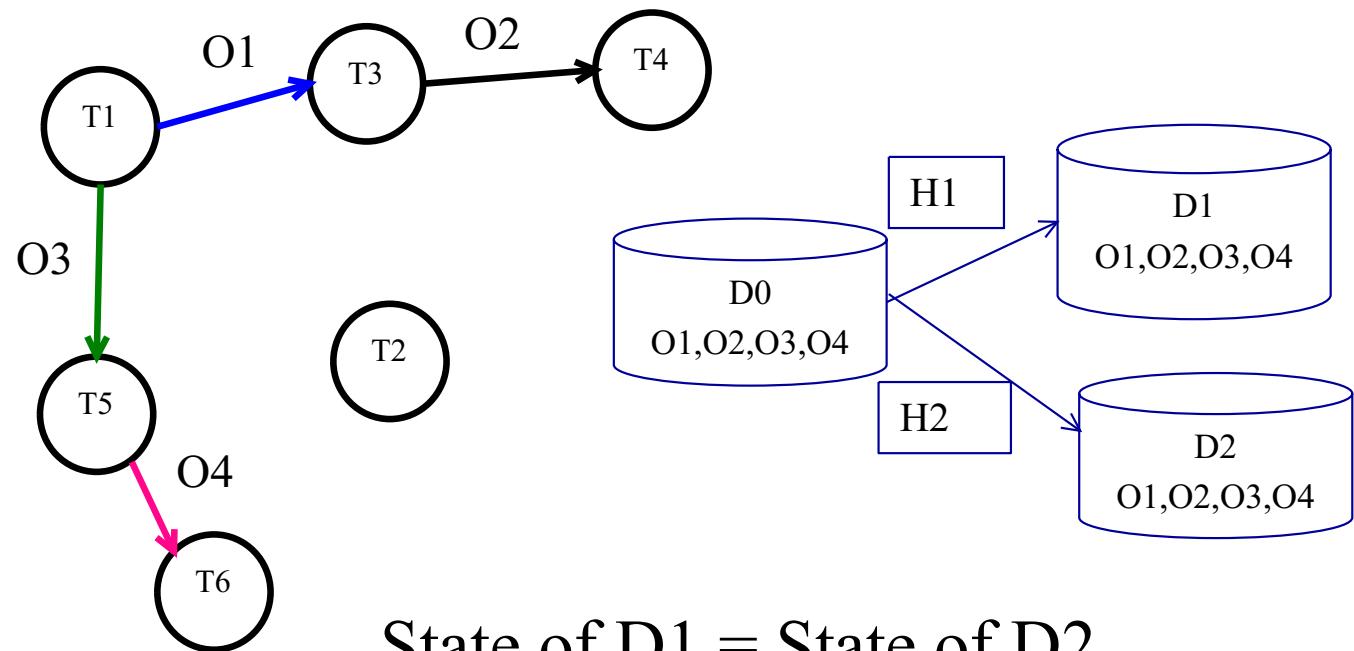
Dependency equivalence contd.

$H1 = \langle (T1, R, O1), (T2, W, O5), (T1, W, O3), (T3, W, O1), (T5, R, O3), (T3, W, O2), (T5, R, O4), (T4, R, O2), (T6, W, O4) \rangle$

$H2 = \langle (T1, R, O1), (T3, W, O1), (T3, W, O2), (T4, R, O2), (T1, W, O3), (T2, W, O5), (T5, R, O3), (T5, R, O4), (T6, W, O4) \rangle$

$\text{DEP}(H1) = \{ \langle T1, O1, T3 \rangle, \langle T1, O3, T5 \rangle, \langle T3, O2, T4 \rangle, \langle T5, O4, T6 \rangle \}$

$\text{DEP}(H2) = \{ \langle T1, O1, T3 \rangle, \langle T1, O3, T5 \rangle, \langle T3, O2, T4 \rangle, \langle T5, O4, T6 \rangle \}$





How does this all help with Isolation?

A history is said to be **isolated** if it is equivalent to a **serial history** or in other words as if all transactions are executed serially/sequentially

A serial history is history that is resulted as a consequence of running transactions sequentially one by one

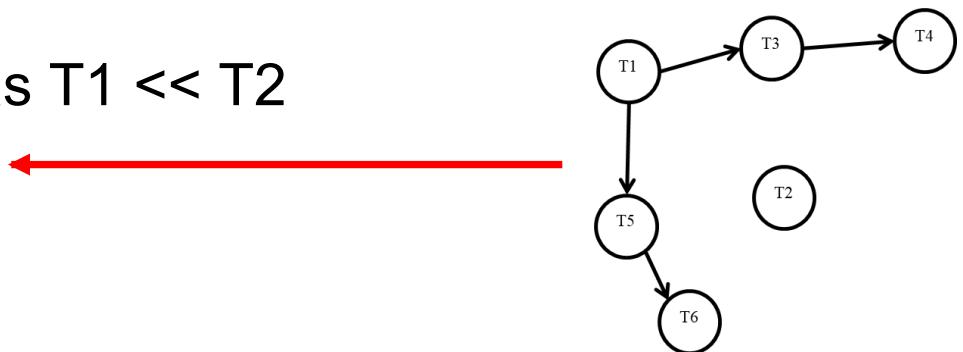
N transactions can result in a maximum of $N!$ serial histories

Bad Orders of Execution

If T_1 precedes T_2 , it is written as $T_1 \ll T_2$

$\text{Before}(T) = \{T' \mid T' \ll T\}$

$\text{After}(T) = \{T' \mid T \ll T'\}$



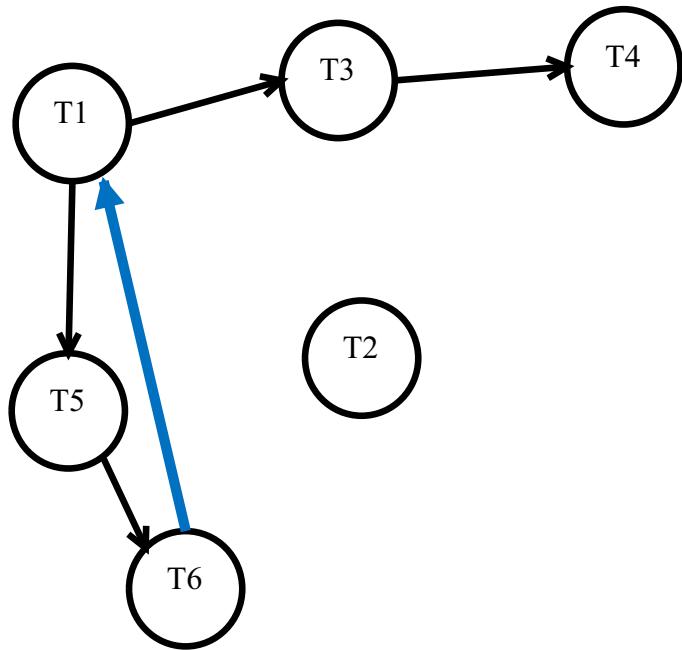
Wormhole Definition: A transaction T' is called a **wormhole transaction** if

$$T' \in \text{Before}(T) \cap \text{After}(T)$$

That is $T \ll T' \ll T$. This implies **there is a cycle in the dependency graph** of the history. Thus, presence of a wormhole transaction implies things are **not isolated**, i.e., **no matching to a serial execution**.

Wormhole theorem: A history is isolated if and only if it has no wormholes.

Examples



Example 1: $\text{After}(T1) = \{T5, T6, T3, T4\}$

$\text{After}(T3) = \{T4\}$

$\text{After}(T5) = \{T6\}$

$\text{After}(T3) = \{T4\} \dots$

There is no such T' as below:

$$T' \in \text{Before}(T) \cap \text{After}(T)$$

Example 2: $T' = T6$ where T is $T1$

$$T' \in \text{Before}(T) \cap \text{After}(T)$$



How do we ensure no wormholes?

We first need to **use locks to make sure access is done in an orderly fashion**

...And we need to avoid cycles in the dependency graph



How do we ensure no wormholes contd.

Solution: We will use locks but carefully thinking about types of locks and algorithms to get locks considering the progression of the transactions and not focusing on locking per resource access

(So things are more complicated than just getting a lock that we saw earlier, i.e., simple locking per resource access...)

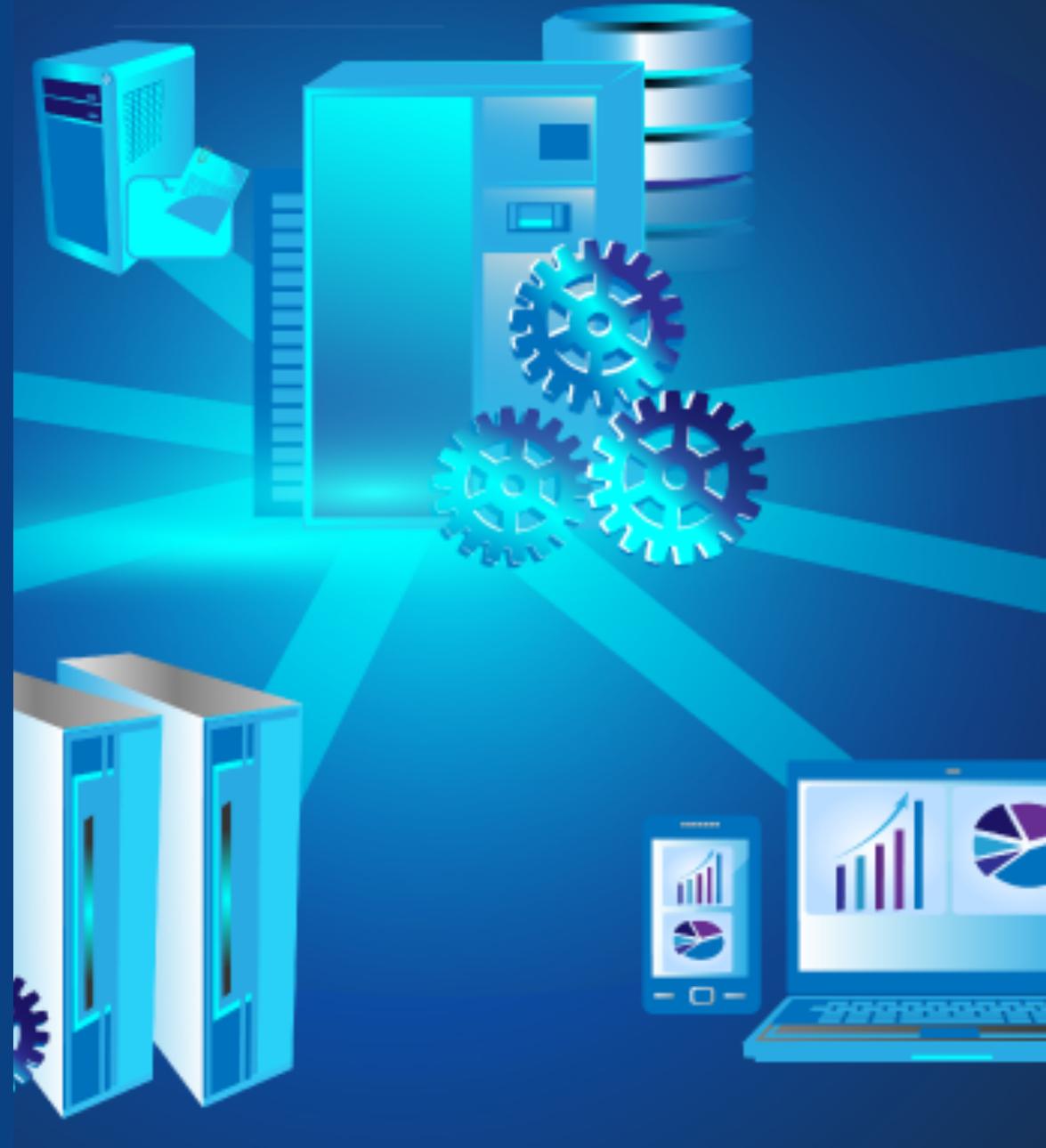


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

PART III:
CONCURRENCY CONTROL
CONTD.

COMP90050 Advanced Database Systems





What and how to lock

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

	Current Mode of Lock		
Request	Free	Shared	Exclusive
Shared request (SLOCK) Used to block others writing/modifying			
Exclusive request (XLOCK) Used to block others reading or writing/modifying			



When to use locks

Actions in Transactions are: **READ, WRITE, SLOCK, XLOCK, UNLOCK (+ BEGIN, COMMIT, ROLLBACK)**

T1

BEGIN
SLOCK A
XLOCK B
READ A
WRITE B
COMMIT
UNLOCK A
UNLOCK B
END

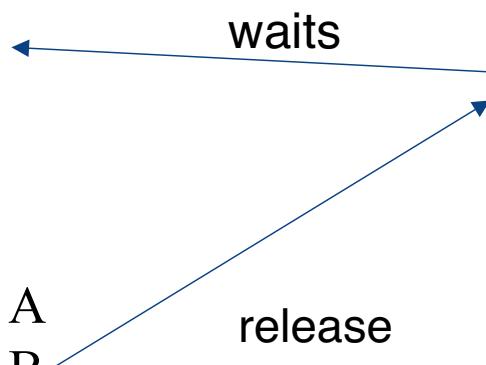
T2

BEGIN
SLOCK A
XLOCK B
READ A
WRITE B
ROLLBACK
UNLOCK A
UNLOCK B
END

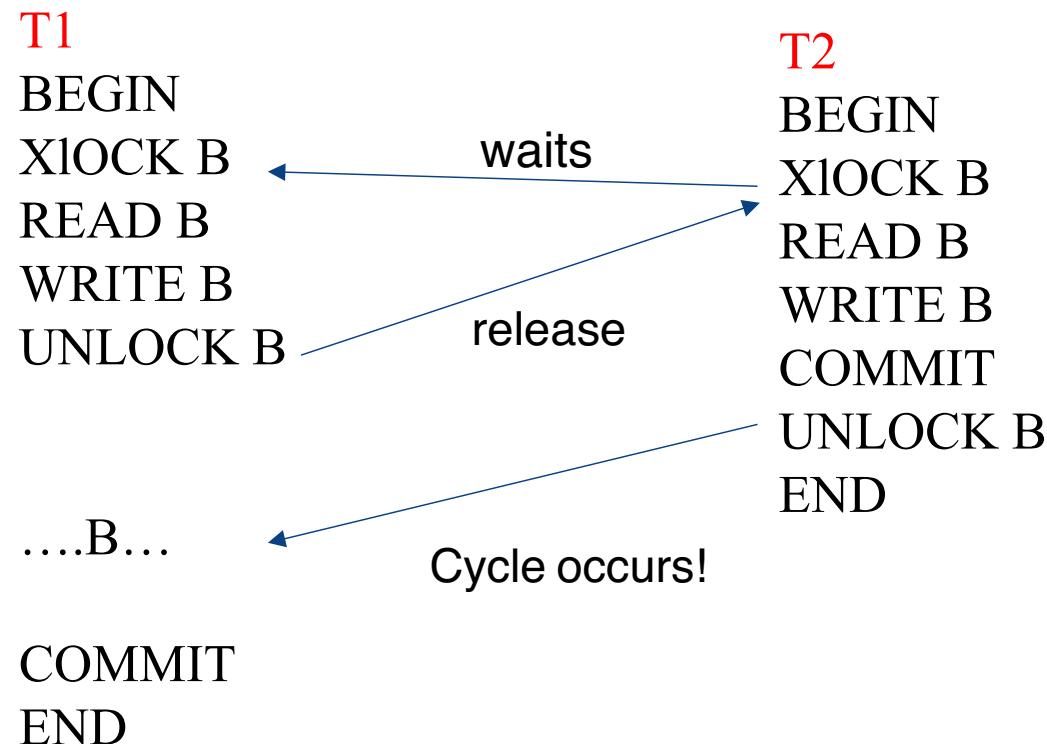
When to use locks contd.

T1
BEGIN
SLOCK A
XLOCK B
READ A
WRITE B
COMMIT
UNLOCK A
UNLOCK B
END

T2
BEGIN
SLOCK A
XLOCK B
READ A
WRITE B
ROLLBACK
UNLOCK A
UNLOCK B
END



Recall: Without a strategy for locking things may not work, for example...





A Two Phase Approach

A transaction is a sequence of READ, WRITE, SLOCK, XLOCK actions on objects ending with COMMIT or ROLLBACK.

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 it does not grant conflicting grants.

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



Locking Theorem

Locking theorem: If all transactions are **well formed** and **two-phased**, then any **legal** 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 transaction**.

Rollback theorem: An update transaction that **does an UNLOCK** and **then does a ROLLBACK** is **not two phase**.



Serializability is Guaranteed

- In short, using **two-phase locking we guarantee serializability**
- That is the concurrent execution of a set of transactions is equal to **one of the serial executions of the same set of transactions**
- Thus **isolation property is achieved**
- It is easy to see that it is different than locking each object independently
- The strategy ensures a transaction has the set of locks needed by getting all of them in phase 1 before releasing any in phase 2
- Thus **preventing any transaction getting in-between**



Strict two-phase locking (S2PL)

- This is when there is **no shrinking phase**
- Locks has to be released when commit/abort happens
- Reduces concurrency and efficiency further
- But good for preventing cascading aborts as an example:
 - These are cases when abort leads to many transactions aborting because they read an object that is written by the aborting transaction
 - So where such things can occur S2PL is better choice!



Some interesting issues with locks

- Phantoms: They commonly occur when we lock records but another transaction adds a new record
- So reading the same table twice from one transaction presents two different sets of records as the other has inserted some in the meantime
- Predicate locks solve this problem. Rather than locking records, lock based on condition
- Page Locks can solve this problem too. In this case, the storage should be organized based on values of attributes.
- High volume systems use specialized locking mechanisms as well:
 - Key Range Locks: to protect b-trees
 - Hole Locks: to protect space for uncommitted deletes



Efficiency concerns remain

- Unfortunately, we are not fully done with the 2PL solution
- The **more locking we do the more restricted we get** in terms of what concurrency we allow
- Practitioners observed that some of these are not needed in practice in every front and instead concurrency should take high precedence at times
- Thus one can define **degrees of isolation**

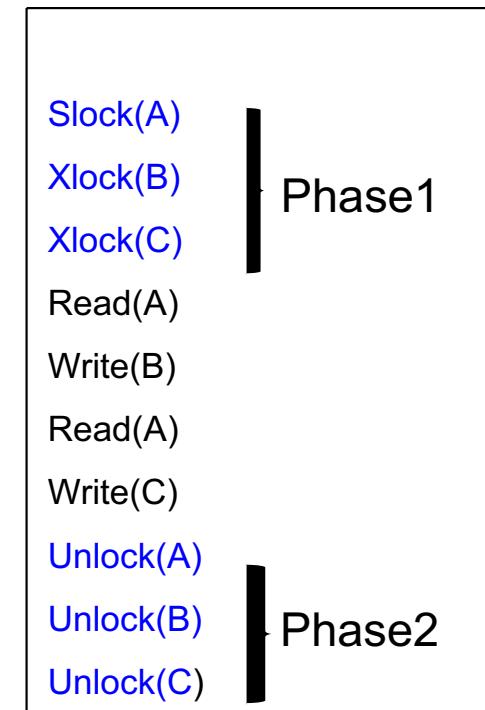
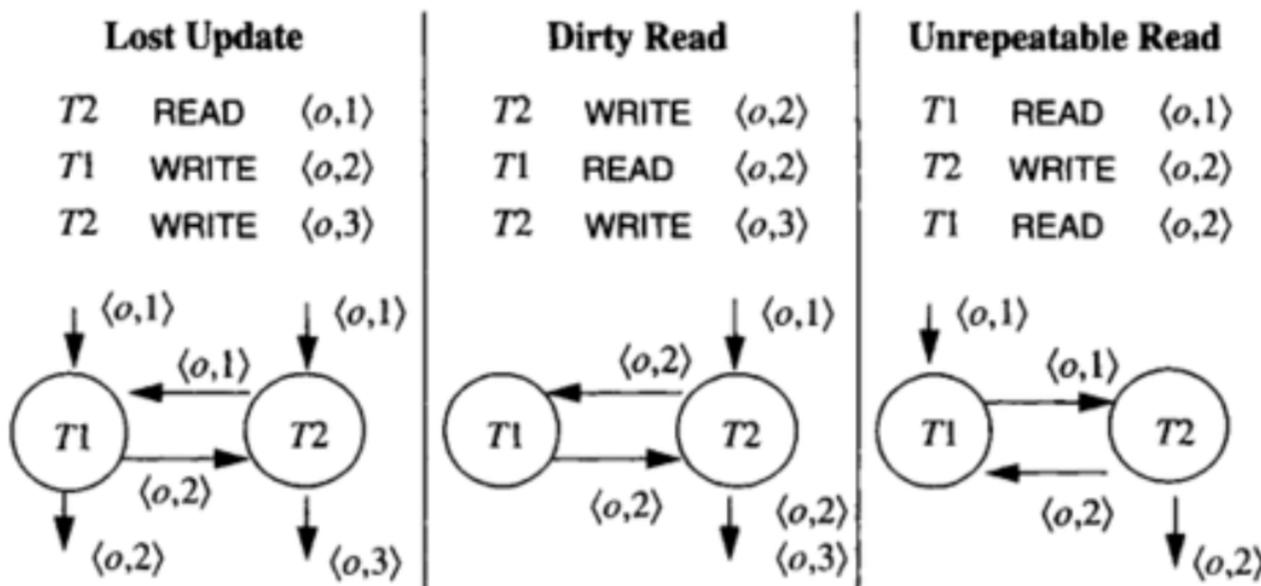
Degrees of Isolation

Degree 3: A third degree isolation of transactions will have no lost updates, no dirty reads 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





Degrees of Isolation Contd.

Degree 2: A second degree of isolation of 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 unrepeatable reads though!)

It is sensitive to the following conflicts:

write->write; write ->read;

Degree 2	
Slock(A)	
Read(A)	
Unlock(A)	
Xlock(C)	Phase1
Xlock(B)	With X-locks
Write(B)	
Slock(A)	
Read(A)	
Unlock(A)	
Write(C)	
Unlock(B)	Phase2
Unlock(C)	With X-locks



Degrees of Isolation Contd.

Degree 1

Note 1: Where are you going to use say Degree 1 isolation:

“This is intended for transactions that need **only an approximate picture of the database** .”

Note 2: These are set by a command like:

“**SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE**”

// Same as degree 3

writes.

It ignores all conflicts.

Write(C)
Unlock(C)

Protecting
writes

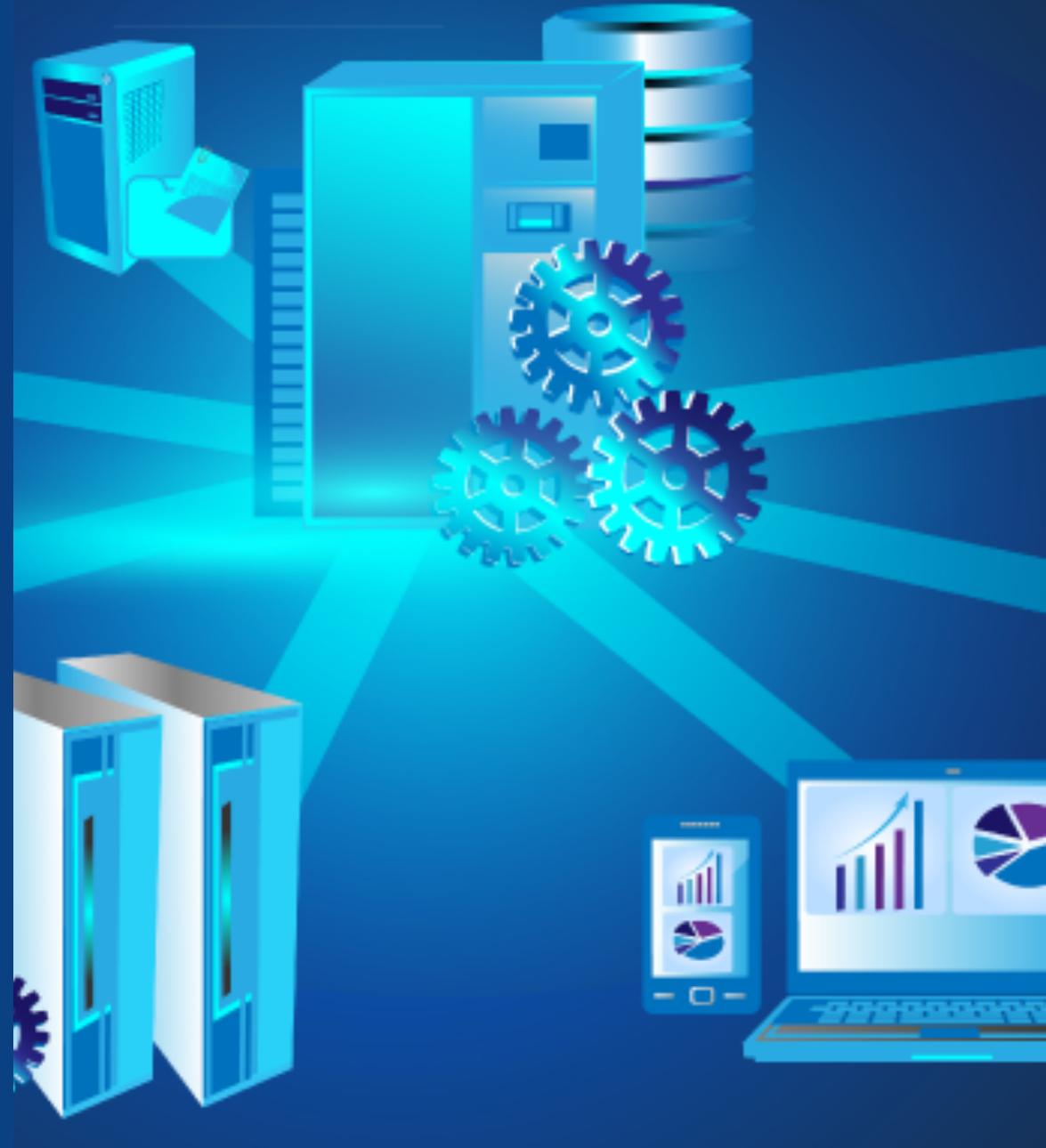


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

CONCURRENCY CONTROL
CONTD.

COMP90050 Advanced Database Systems





Degrees of Isolation Contd.

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 to following conflicts:

write->write;

Degree 0 : A Zero degree isolation of transactions **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 1

Read(A)
Xlock(C)
Xlock(B) }

Phase1

Write(B)

Read(A)

Write(C)

Unlock(B) }

Phase2

Unlock(C) }

Degree 0

Read(A)
Xlock(B)
Write(B) }

Protecting
writes

Unlock(B) }

Read(A)

Xlock(C)

Write(C) }

Protecting
writes

Unlock(C) }



Degrees of Isolation Contd.

- Where are we going to use say Degree 1 isolation:

This is intended for transactions that need **only an approximate picture of the database** .

- These are set by a command like:

SET TRANSACTION ISOLATION LEVEL

SERIALIZABLE

// Same as degree 3



Further Refinements: Granularity of Locks

Simplest: Lock the whole DB – simple, less conflicts, but poor performance

So we want to lock smaller units

E.g., Lock at individual records level – more difficult to manage, more individual locks, but better performance due to concurrency increases

So the idea is to do the right level of locking when needed



Granularity of Locks Contd.

To do this:

- we **build a hierarchy**
- then **locks can be taken at different levels**
- which will automatically grant the locks on its descendants



Granularity of Locks Contd.

- Acquire locks from root to leaf
- Release locks from leaf to root



Granularity of Locks Contd.

We create a new lock matrix as the first step for a hierarchy

Real DBMS have very complex and different matrices, following is a simple example

Intention mode locks are used on coarse granules:

+ granted, - delayed

Compatibility Matrix				
Mode	Free	I (Intent)	S (Share)	X (Exclusive)
I	+ (I)	+ (I)	- (S)	- (X)
S	+ (S)	- (I)	+ (S)	- (X)
X	+ (X)	- (I)	- (S)	- (X)



More Lock Types Exist in Real Life

- X eXclusive lock
- S Shared lock
- U Update lock
- IS Intent to set Shared locks at finer granularity
- IX Intent to set shared or exclusive locks at finer granularity
- SIX a coarse granularity Shared lock with an Intent to set finer granularity eXclusive locks

Compatibility Mode of Granular Locks

Current	None	IS	IX	S	SIX	U	X
Request	+ - (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)	- (IS)	- (IX)	+ (U)	- (SIX)	- (U)	- (X)
X	+ (X)	- (IS)	- (IX)	- (S)	- (SIX)	- (U)	- (X)



How it works

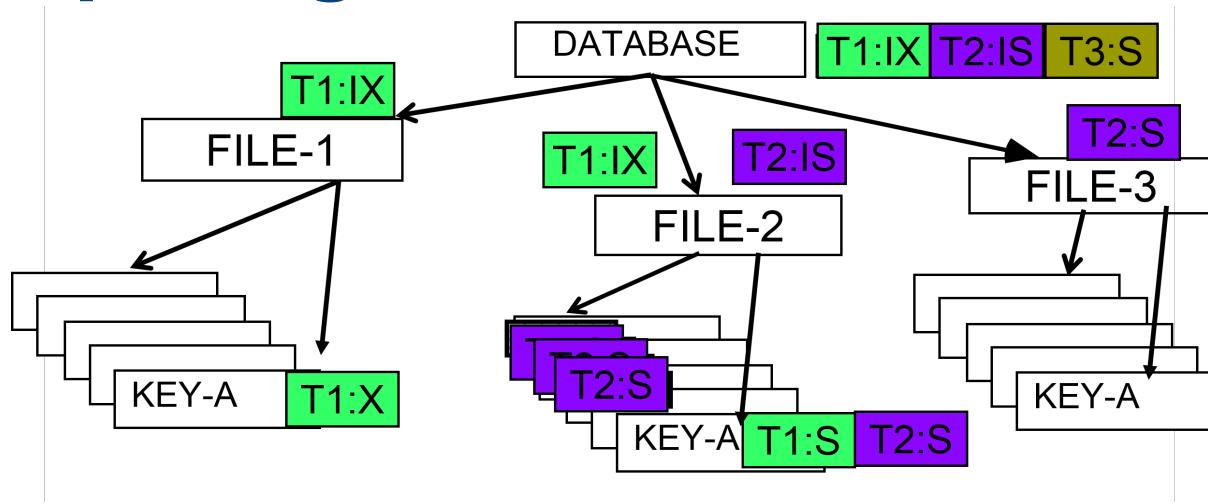
- IS (intention to have shared lock at finer level) allows IS and S mode locks at finer granularity and prevents others from holding X, U on this node.
- IX (intention to have exclusive lock at finer level) allows to set IS, IX, S, SIX, U and X mode locks at finer granularity and prevents others holding S, SIX, X, U on this node.
- S (shared) allows read authority to the node and its descendants at a finer granularity and prevents others holding IX, X, SIX on this node.



Lock modes contd.

- **SIX (share and intension exclusive) allows** reads to the node and its descendants as in IS and prevents others holding X, U, IX, SIX, S on this node or its descendants but allows the holder IX, U, and X mode locks at finer granularity. SIX = S + IX
- **U (Update lock) allows** read to the node and its descendants and prevents others holding X, U, SIX, IX and IS locks on this node or its descendants.
- **X (exclusive lock) allows** writes to the node and prevents others holding X, U, S, SIX, IS, IX locks on this node and all its descendants.

Acquiring locks on the tree



- To acquire an S mode or IS mode lock on a non-root node, **one parent must be held in IX mode or higher** (one of {IS,IX,S,SIX,U,X}).
- To acquire an X, U, SIX, or IX mode lock on a non-root node, **all parents must be held in IX mode or higher** (one of {IX,SIX,U,X}).



Update Mode Locks

T1:

SLock A

Read A

If ($A < 5$)

{

% Upgrading Slock to Xlock

Xlock A

Write A

}

Unlock A

T2:

SLock A

Read A

If ($A < 5$)

{

% Upgrading Slock to Xlock

Xlock A

Write A

}

Unlock A

This can cause a deadlock...



Update Mode Locks

T1:

```
Ulock A  
Read A  
If (A < 5){  
    Xlock A  
    Write A  
}  
Unlock A
```

T2:

```
Ulock A  
Read A  
If (A < 5){  
    Xlock  
        A  
        Write  
        A  
    }  
    Unlock A
```

T3:

```
Slock A  
Read A  
Unlock A
```

Ulock can also be downgraded to Slock if no update occurs...



Two-version Locking

- There are other extensions to locking mechanisms
- One of them is two-version locking
- It can **allow more concurrency** as well
- The idea is to allow **one transaction to write tentative versions**
- While **other transactions read from the committed versions**
- **Thus reads are only delayed only while transactions are being committed rather than during an entire transaction**



Nested Transactions and Locking

- Each subtransaction in a set **must not observe partial effects of others in the set**
- Rules:
 1. Each **subtransaction's locks are inherited by its ancestors** as a rule when it completes
 2. Inherited **locks should also be inherited recursively**
 3. This ensures **top level can decide ultimately**
 4. Also, **parents cannot run concurrently with children**
 5. Children can however **use the locks of parents**
 6. Subtransactions at same level that access same object **take turns to get locks of the parent for serializability**

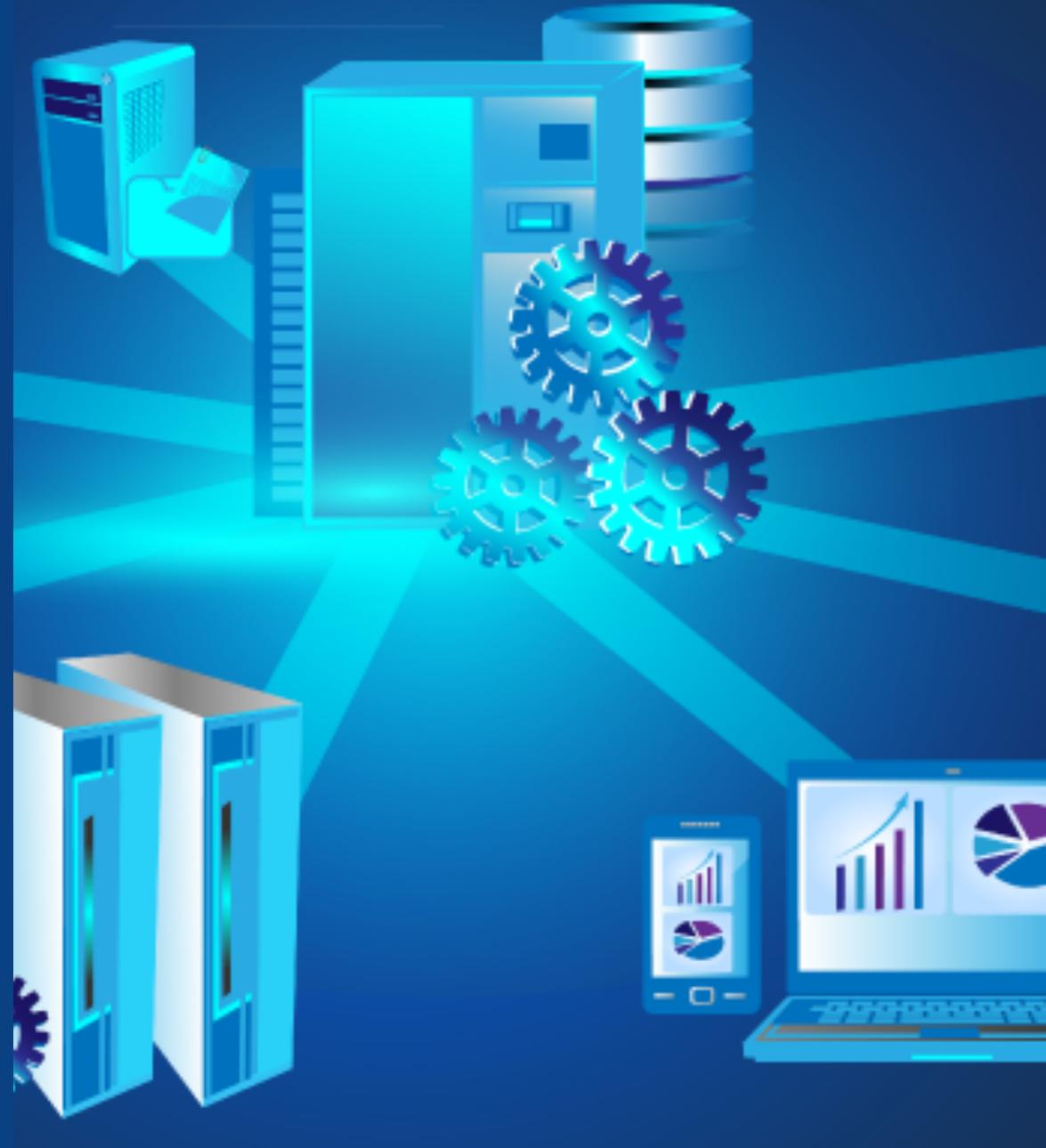


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

CONCURRENCY CONTROL
CONTD.

COMP90050 Advanced Database Systems





Other Types of Concurrency Control: Optimistic Concurrency Control

When conflicts are rare, transactions can execute operations without managing locks and without waiting for locks ↗ **higher throughput**



Read (A); Read (B); Read (C);...

T1

Read (C); Read (D); Write (E);...

T2

No conflict among these two transactions...



Other Types of Concurrency Control: Optimistic Concurrency Control

- Use data without locks
- Before committing:
 - **Each transaction verifies** that no other transaction has modified the data
 - **Brief locks may be needed** at the end but duration of locks are very short
 - If **any conflict found, one transaction repeats**
- If **no conflict, make changes and commit**



Optimistic Concurrency Control

Read A into A1

Read B into B1

Read C into C1

....Compute some values based on A1 and B1 and C1 e.g., $t = A1+B1+C1\dots$

% Start taking locks

Slock A; Read A into A2

Slock B; Read B into B2

Xlock C; Read C into C2

if ($A1 == A2 \& B1 == B2 \& C1 == C2$)

 Write t into C

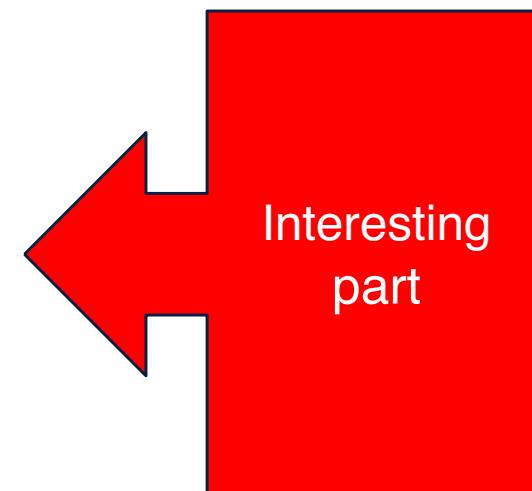
 commit

 Unlock A, B and C

else % data has changed

 unlock A ,B and C

 goto Start





Optimistic Concurrency Control Variations

In general there are two versions:

Optimistic Concurrency Control with **backward validation**

Optimistic Concurrency Control with **forward validation**



Backward validation

- **Transaction T which is the current transaction** makes the check
- Creates **a read set** that it has performed during execution
- **Looks whether any other transaction that it is overlapping with has changed anything it has read**
- If so then takes action, as we have seen earlier...

T

Read (A); Read (B); Read (C);...

Read (C); Read (D); Write (E);...



Forward validation

- Transaction T which is the current transaction makes the check
 - Creates a write set that it has performed during execution
 - Looks whether any other transaction that it is overlapping with has read anything it has written; we assume they are still active
 - Make a choice of:
 - Delay current transaction
 - Abort all the conflicting ones
 - Abort current transaction
 - Thus it has more choices
 - One has to decide based on cost
- T
- Read (A); Read (B); Read (C);...
- Read (C); Read (D); Write (E);...



Oracle used: Snapshot Isolation

Read C into C1

Read D into D1

Start of Loop:

Read A into A1

Read B into B1

... Compute new values based on A1 and B1

% Start taking locks on objects that need modification.

% Let new value for C is C3 and for D is D3.

Xlock C

Xlock D

Read C into C2

Read D into D2

if (C1 == C2 & D1 == D2)

% commit

write C3 to C

write D3 to D

commit

unlock(C and D)

else % not the first modifier

C1 = C2

D1 = D2

unlock(C and D)

goto Loop

- Not the same thing as **serializability**
- Checks if what needs to be written is changed
- If not it is allowed to commit
- Things read are not checked



Another Type of Concurrency Control: Time-stamp-based Concurrency Control

- Similarly a validation process happens like Optimistic Concurrency control
- The main difference is **validation happens as transactions do their operations**, i.e. continuously
- So validation is per operation and **abortion is imminent**
- As the name suggests **decisions are based on timestamps**
- Transactions and operations can be ordered based on timestamps



Rules for Time-stamp-based Conc. Cont.

To accept a write by transaction T on object D check

If $T \geq$ maximum read timestamp on D AND
 $T >$ write timestamp on current committed D
then write D with timestamp of T

Else

Abort T



Time-stamp-based Concurrency Contd

To accept a read by transaction T on object D check

If $T >$ write timestamp on current committed D
then perform read...

Else

Abort T



Comparison of Concurrency Control Schemes

- **Timestamp** ordering assigns orders for **transactions based on time of commencement**
- **Locking** has some sort of order which is **decided at object access time**
- When there are **many updates two-phase locking is good** as it has less aborts
- **Timestamp-based methods abort immediately** which may be good sometimes
- If **there aren't many updates an optimistic approach is better**
- ... so there is **no one winner for all DBMSs for all types of data/queries**

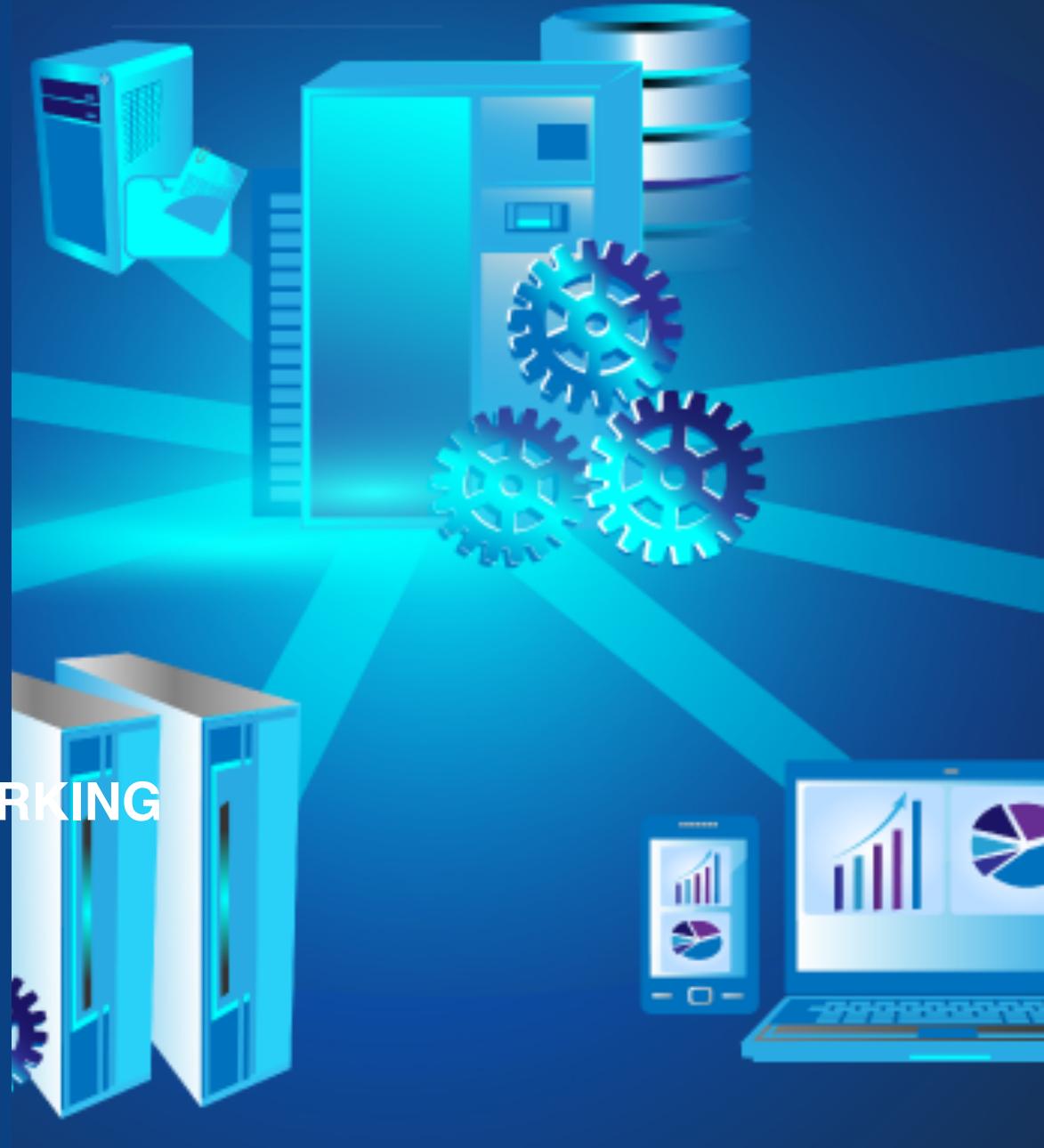


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

TRANSACTIONS & NETWORKING

COMP90050 Advanced Database Systems





DBMSs in the era of Networking

- Almost all **database management systems of today are networked**
- Some would use a very simple client-server architecture where a version of two-phase locking strategy with a simple commit/abort mechanism would work as everything is on one server
- Many large corporations today **have multiple servers working together** though
- These imply that many systems do not have a simple model but a more **complex distributed architecture**

Example: Nested transactions run fast in a distributed way

client transfers \$10 from A to C and then transfers \$20 from B to D

$T = openTransaction$

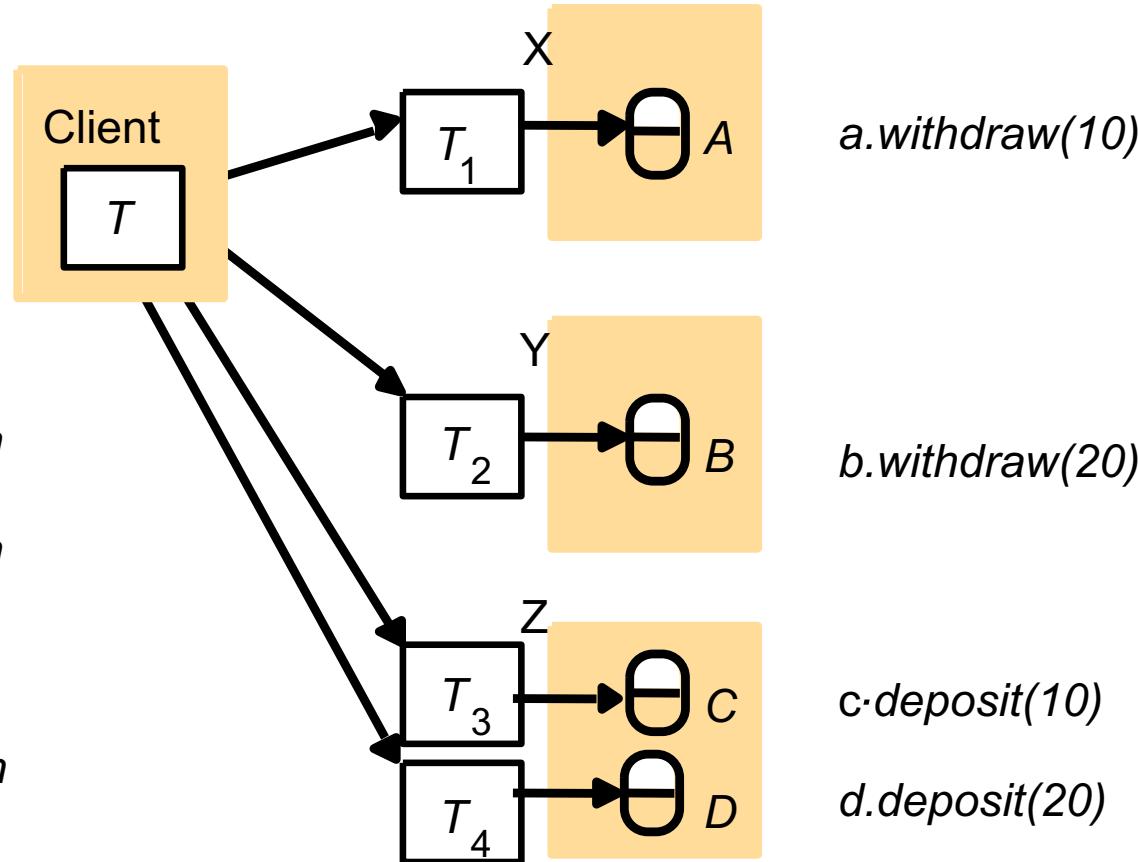
$openSubTransaction$
 $a.withdraw(10);$

$openSubTransaction$
 $b.withdraw(20);$

$openSubTransaction$
 $c.deposit(10);$

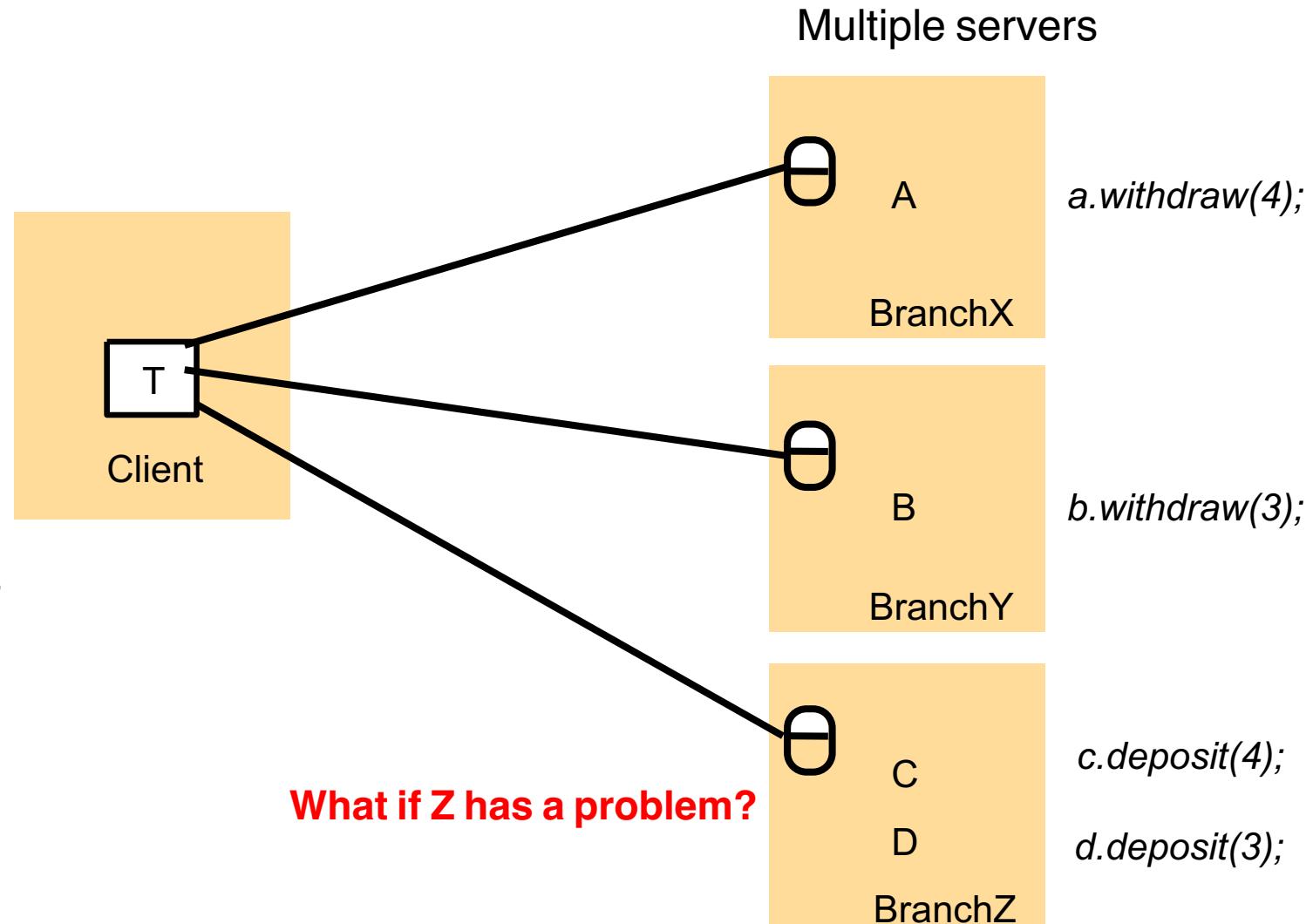
$openSubTransaction$
 $d.deposit(20);$

$closeTransaction$



**Requests can be run in parallel -
with several servers, the nested
transaction is more efficient**

Problems related to lack of having one data center





Problems related to lack of having one data center solved

Create a mechanism for:

- either all of the servers commit the transaction
- or all of them abort the transaction

One of the servers becomes the *coordinator*, it must ensure the same outcome at all of the servers

The *two-phase commit protocol* is the most commonly used protocol for achieving this...

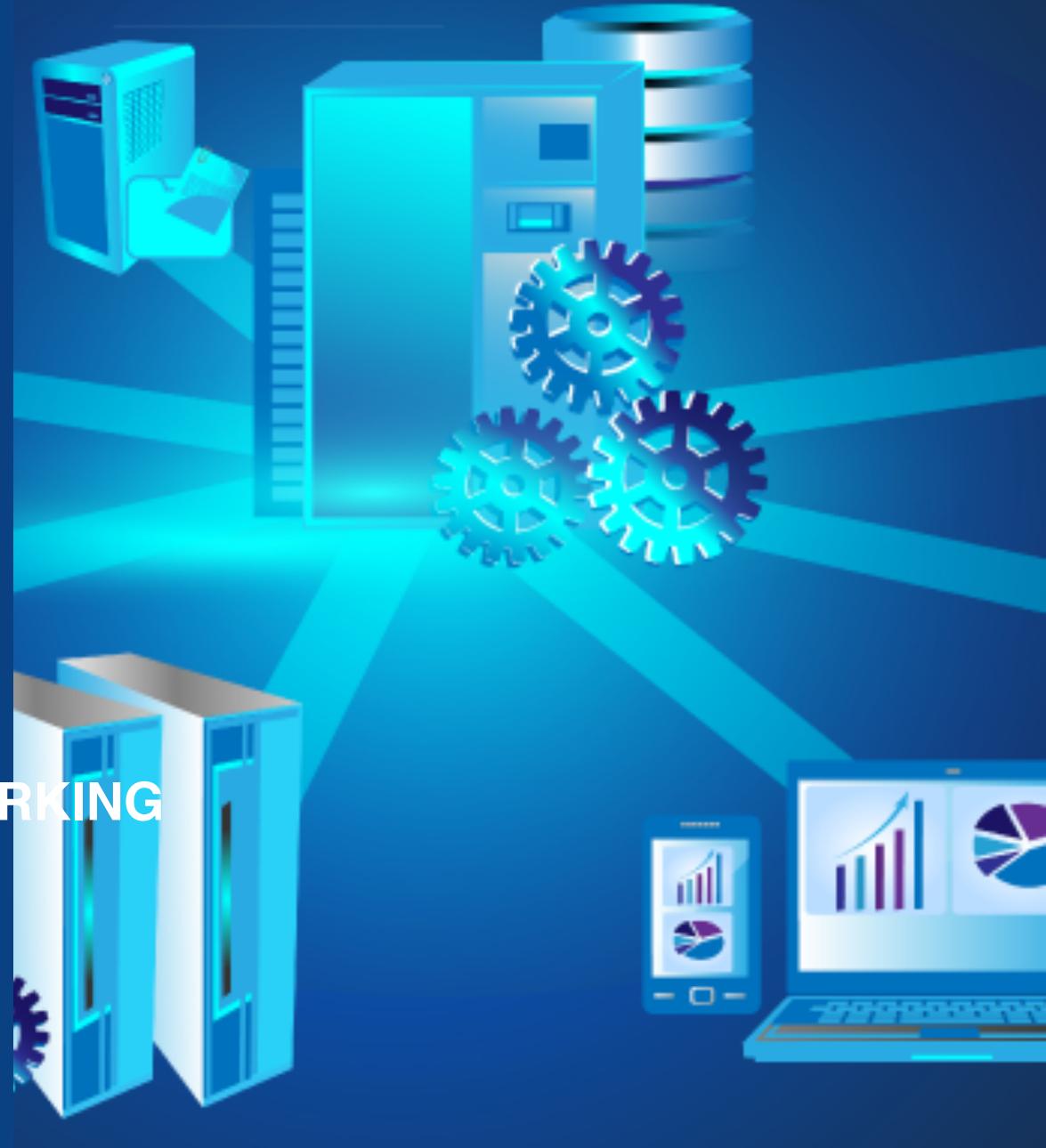


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

TRANSACTIONS & NETWORKING
CONTD.

COMP90050 Advanced Database Systems





Lets develop the Two-phase commit

Two-phase atomic commit protocol:

- is designed to allow **any participant to choose** to abort a transaction.
- ***phase 1*** - each participant votes.
 - If it votes to commit, it is ***prepared: It cannot change its mind.***
 - In case it can crash, it must ***save updates in permanent store.***
- ***phase 2*** - the participants carry out the joint decision.



Two-phase commit protocol Contd

2PC phase labels

voting phase: coordinator asks all participants if they can commit
if yes, participant records updates in permanent storage and then votes
so even when there is failure participant can still recover

completion phase: coordinator tells all participants to commit or abort

Note:

- if client or participant request abort, the coordinator informs the participants immediately



Functions to implement for Two-phase commit

participant interface- *canCommit, doCommit, doAbort*

coordinator interface- *haveCommitted, getDecision*

canCommit(trans)? Yes / No

Call from **coordinator** to participant to ask whether it can commit a transaction.

Participant replies with its vote.

doCommit(trans)

Call from **coordinator to participant to commit** its part of a transaction.

doAbort(trans)

Call from **coordinator to participant to abort** its part of a transaction.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

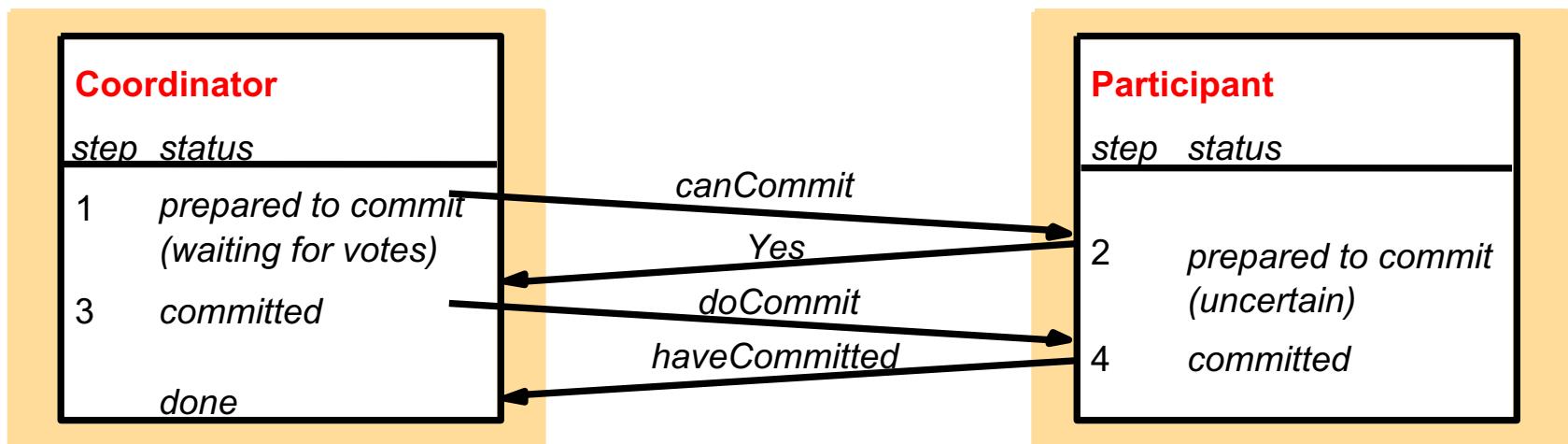
Just for deleting stale information on the coordinator.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay.

Used to recover from server crash or delayed messages.

Communication details in Two-phase commit protocol





What happens in failures...

time-out actions in the 2PC is used

- to avoid blocking forever when a process crashes or a message is lost

uncertain participant that has voted yes: it can't decide on its own

- it uses ***getDecision*** method to ask coordinator about outcome

coordinator delayed in waiting for votes: It can abort and send ***doAbort*** to participants

Note: participant has carried out client requests, but has not had a followup from the coordinator. It can abort unilaterally.



How about concurrency control in networked systems

- Each server is responsible for applying concurrency control to its own objects
- The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner
- **BUT servers independently acting would not work**
- If transaction T is before transaction U in their conflicting access to objects at one of the servers then:
 - They must be in that order at all of the servers whose objects are accessed in a conflicting manner by both T and U
- The central **Coordinator should assure this**



Other Considerations for Locking-based systems

- A local lock manager cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.
- The objects remain locked and are unavailable for other transactions during the commit protocol.
 - An aborted transaction releases its locks after phase 1 of the protocol.



Timestamp ordering concurrency control revisited

- The coordinator accessed by a transaction issues a globally unique timestamp
- The timestamp is passed with each object access
- The servers are jointly responsible for ensuring serial equivalence:
 - that is **if T access an object before U, then T is before U at all objects**



Optimistic concurrency control revisited

For distributed transactions to work:

- 1) **Validation takes place in phase 1 of 2PC protocol at each server**
- 2) **Transactions use a globally unique order for validation**

What if objects in different servers are replicas for increased availability

Client 1:	Client 2:	Server
$setBalance_B(x, 1)$		
$setBalance_A(y, 2)$		
	$getBalance_A(y) \boxed{?}$	
	$getBalance_A(x) \boxed{?}$	

Initial balance of x and y is \$0

The behaviour above cannot occur if A and B did not exist but we had one server



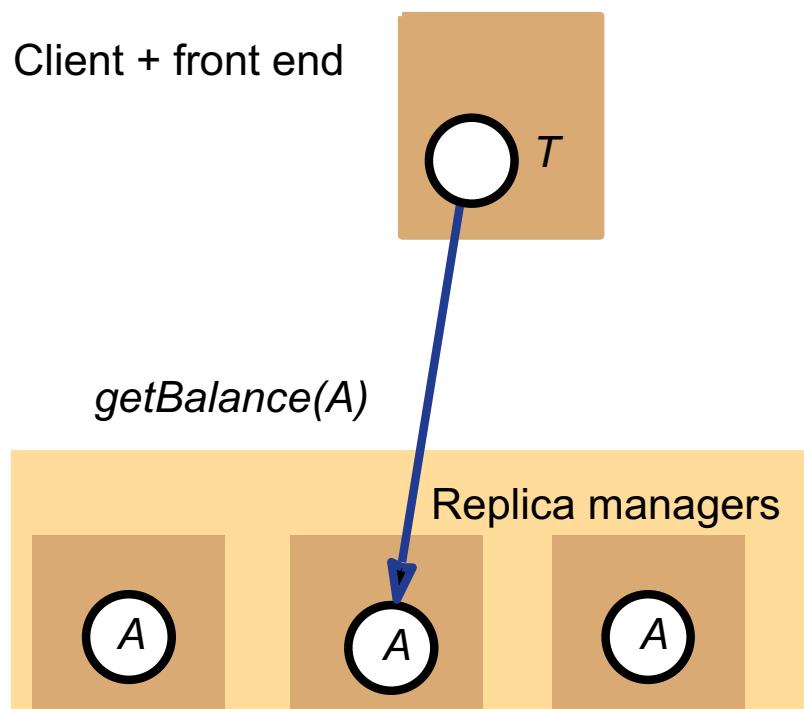
Transactions with replicated data

- the effect of transactions on replicated objects should be the **same as if they had been performed one at a time on a single set of objects**
- this property is called ***one-copy serializability***
- each server provides concurrency control of its own objects to start with
 - we assume two-phase locking in this section
 - other schemes can be updated accordingly as well

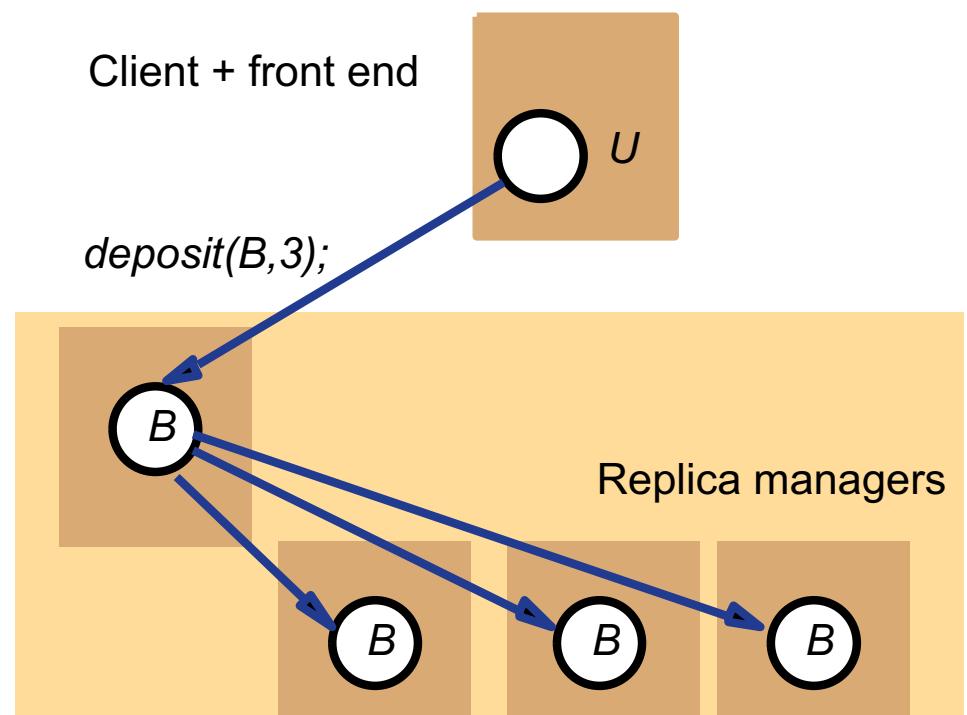
Transactions with replicated data

In **read one/write all replication**, one server is required for a *read* request and all servers for a *write* request

each read operation is performed by a single server, which sets a read lock



every write operation must be performed at all servers, each of which applies a write lock



Any pair of write operations will require conflicting locks at all of the servers
A read operation and a write operation will require conflicting locks on one server



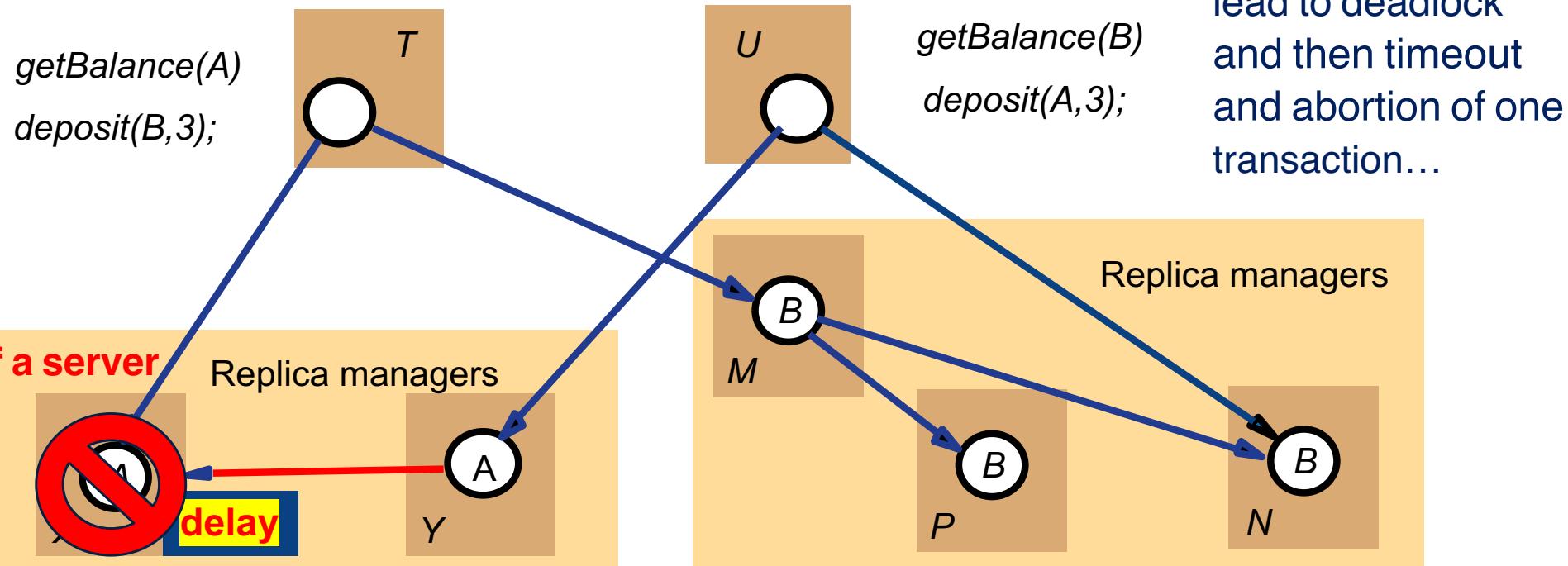
Here we assumed all servers are working all the time

The simple **read one/write all scheme is not realistic**

- because **it cannot be carried out if some of the servers are unavailable** which beats the purpose in many cases

The **available copies replication** scheme is designed to allow some servers to be temporarily unavailable

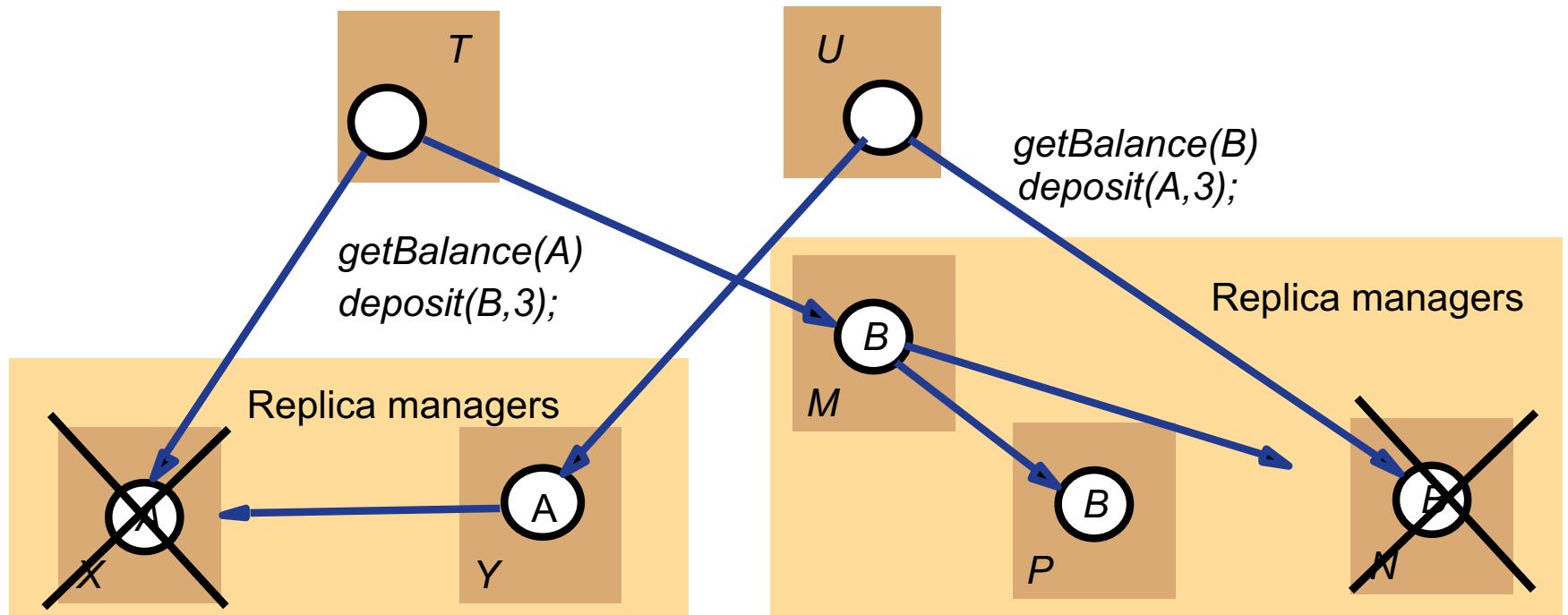
Lets build the solution step by step



At X, T has read A and has locked it. Therefore U's deposit is delayed until T finishes. Normally, this leads to good concurrency control only if the servers do not fail....

Situation analysis contd

assume that X fails just after T has performed $getBalance$
 and lets have N failing just after U has performed $getBalance$



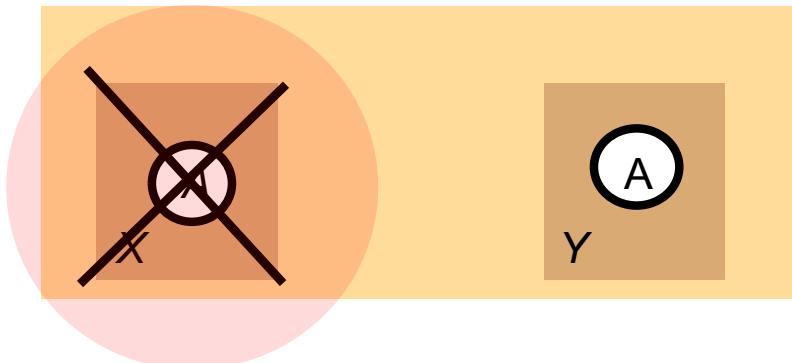
therefore T 's deposit will be performed at M and P (all available)
 and U 's deposit will be performed at Y (all available)

NOT GOOD!!

Available copies replication rule

Before a transaction commits, it checks for failures and recoveries of the RMs it has contacted, the set should not change during execution :

- E.g., T would check if X is still available among others.
- We said X fails before T 's *deposit*, in which case, T would have to abort.
- Thus no harm can come from this execution now.



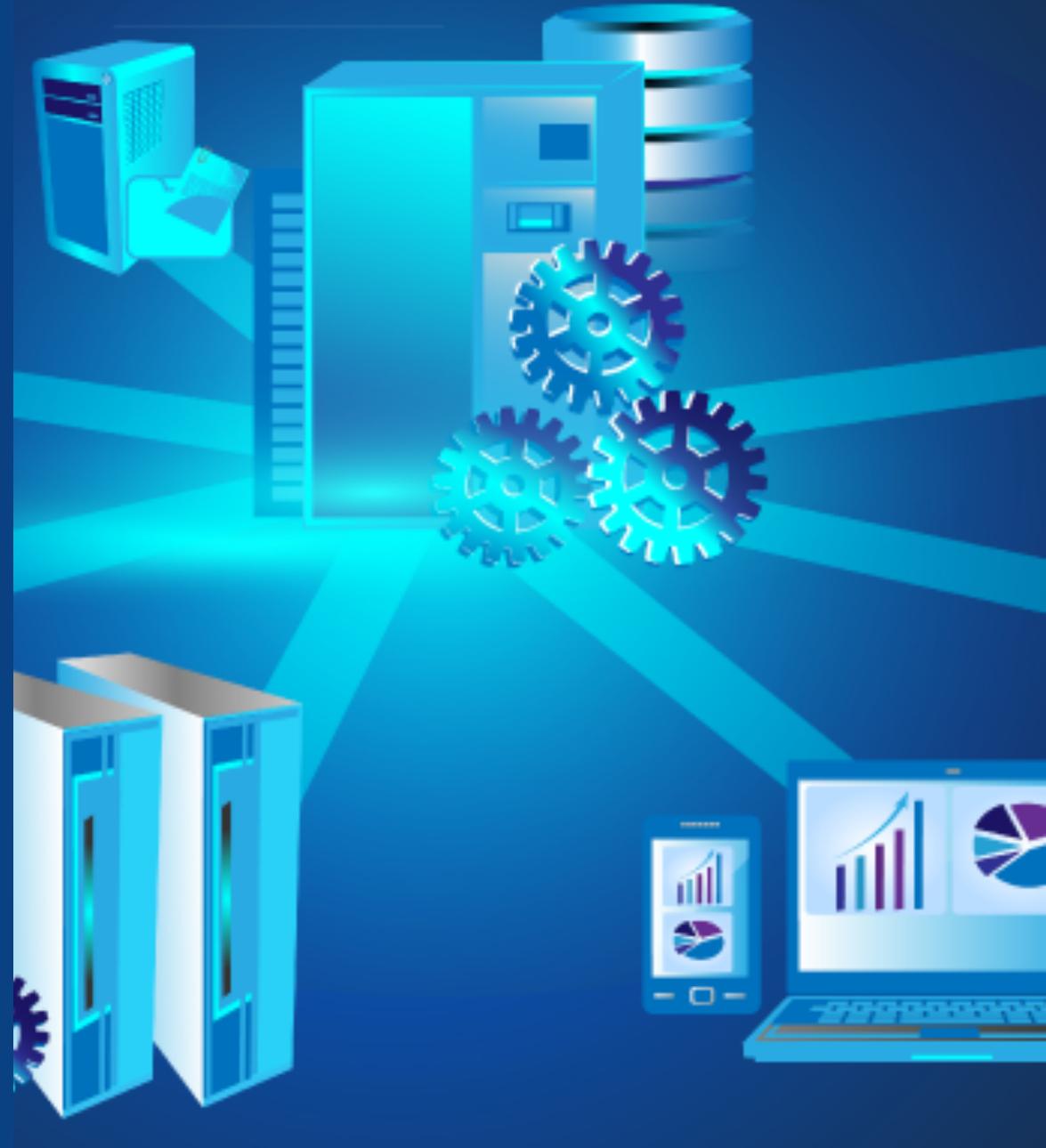


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

FAULT TOLERANCE

COMP90050 Advanced Database Systems





Fault Tolerance

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

For **database management systems preservation of data is the key** and thus they need to tolerate failures that may destroy data.

Hard disks have played a key role on this front.

Key questions that have risen on this front are of type:

- What are the chances of a hard-disk failing on a DB server?
- What are the chances of one of the hard disks failing that are installed in two servers but they were in the same server room?
-



Statistics and Probability are the key to understanding Fault Tolerance

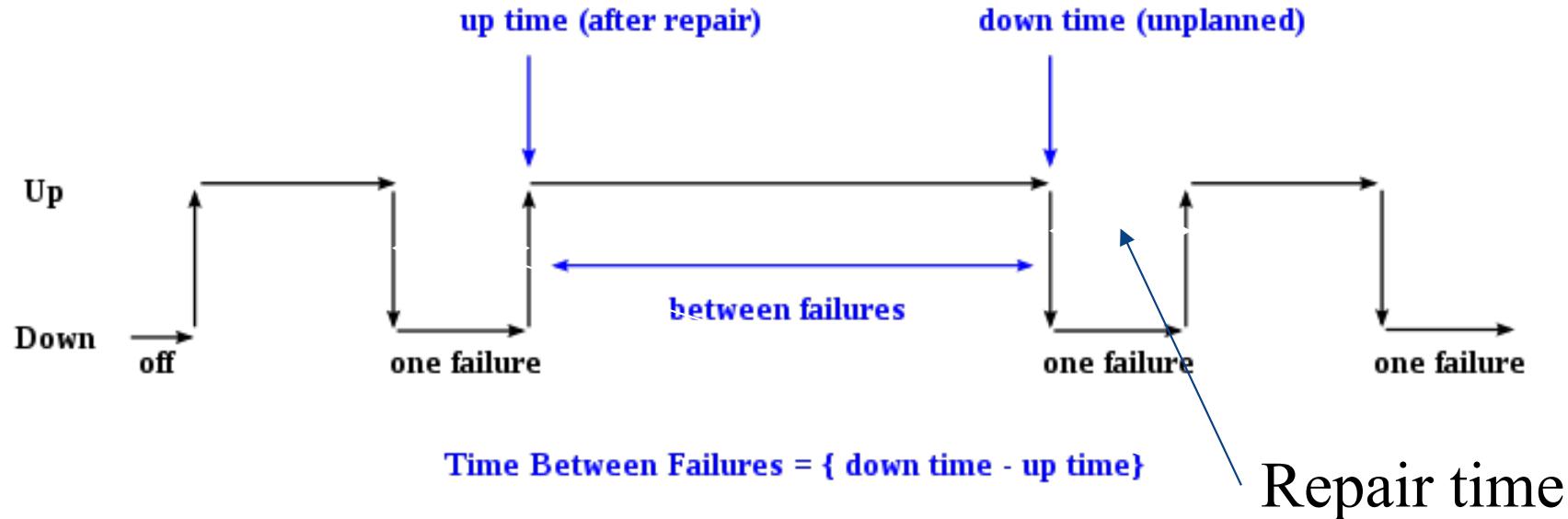
Everything fails eventually so it is all about time; or rather predicting events happening at a certain time.

Thus we estimate things like:

$P(A)$ = probability of a hard disk failing which we call event A, given a **certain duration of time of use**

As **t goes to infinity $P(A)$ approaches 1**

Recall system's lifecycle



Module availability: measures the ratio of service accomplishment to elapsed time

the **time** elapsing before a failure is experienced

$$= \frac{\text{Mean time to failure}}{\text{Mean time to failure} + \text{mean time to repair}}$$

Disk failure rates

Read error

Type of Error	MTTF	Recovery	Consequence
soft data	1 h	Retry	none
seek	6 h	Retry	none Data loss possible
Maskable hard data	3 d	ECC (Error Correcting Coding)	remap
Unmaskable hard data	1 y	none	remap
repair	5 y	repair	data unavailable

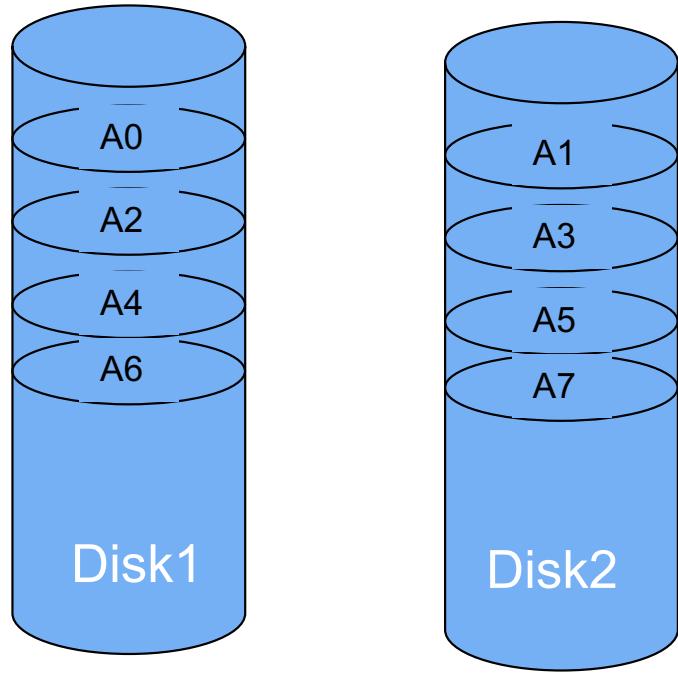


Fault tolerance by RAID: A famous architecture for storage

Redundant Array of Independent Disks/Drives:

- different ways to combine multiple disks as a unit
- for fault tolerance or performance improvement
- or commonly both
- commonly used with a DBMS

RAID (Level Zero)



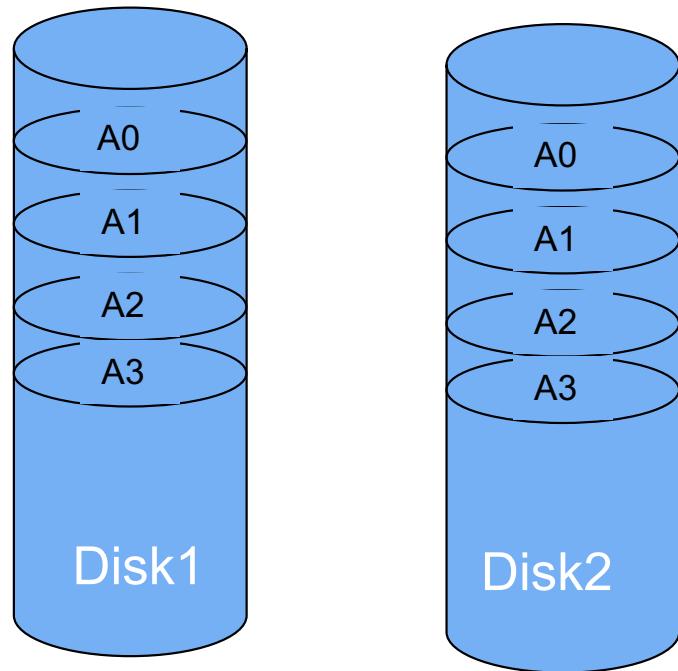
A0, A1, A2, ... are contiguous blocks of data of a file

- Provides balanced I/O of disk drives –**throughput ~doubles**
- Any disk failure will be catastrophic and MTTF reduces by a factor of 2

Higher throughput at the cost of increased vulnerability to failures

- A means Block (4K or 8K bytes of storage)
- **MTTF = Mean Time To Failure**

RAID 1 (Mirroring)

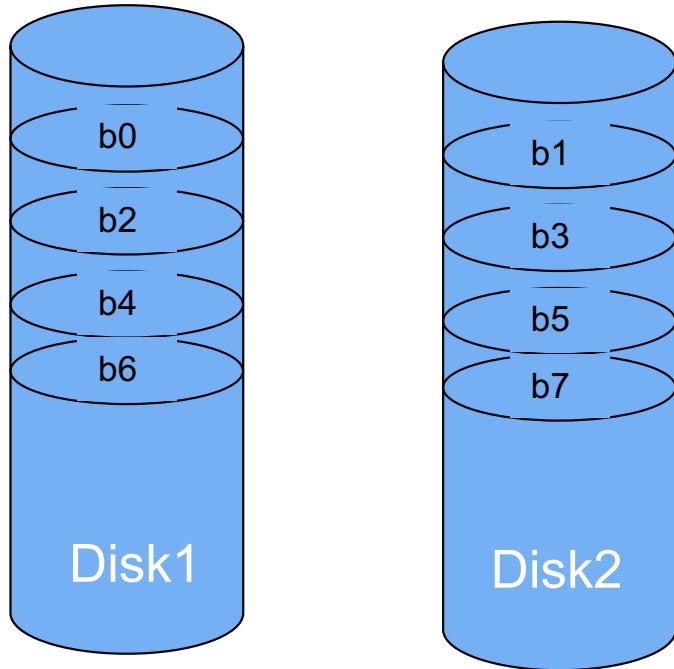


- Provides higher read throughput but lower write throughput
- Half storage utilization.
- MTTF increases substantially (quadratic improvement –i.e. $MTTF^2$!)

Continues to operate as long as 1 disk is functional

- A means Block (4K or 8K bytes of storage)
- MTTF = Mean Time To Failure

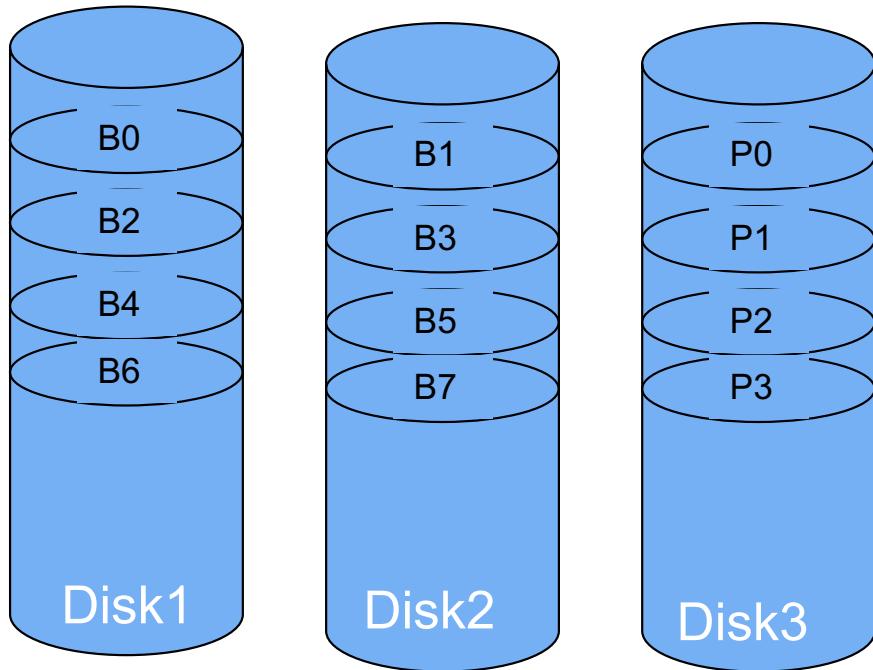
RAID 2 (Bit level striping)



- Striping takes place at bit level
- Provides higher transfer rate (double the single disk)
- MTTF reduced by half as in RAID 0
- rarely used

- b means bit
- MTTF = Mean Time To Failure

RAID 3 (Byte level striping)



- B means Byte
- P is parity
- MTTF = Mean Time To Failure
- Parity (or check bits) are used for error detection

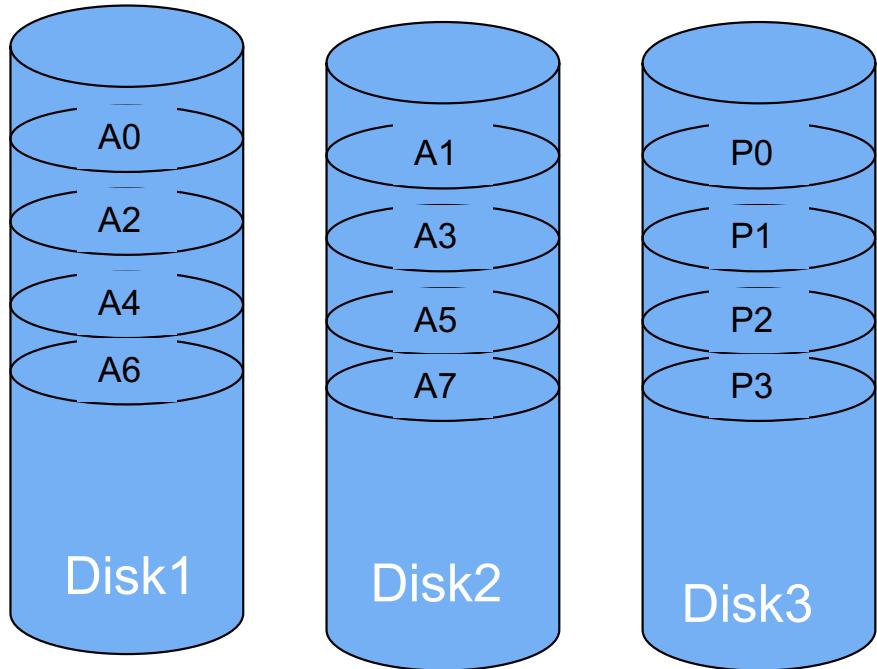
B0, B1, B2, B3, .. are bytes of data of file

- Striping takes place at byte level
- Rarely used
- Provides higher transfer rate as in RAID 0
- P0 is parity for bytes B0 and B1
- $P_i = B_{2i} \oplus B_{2i+1}$, here \oplus is exclusive-or operator

MTTF increases substantially:

1/3 of RAID1 = MTTF^{2/3}, as 1 disk failure can be recovered from the data of the other 2 disks

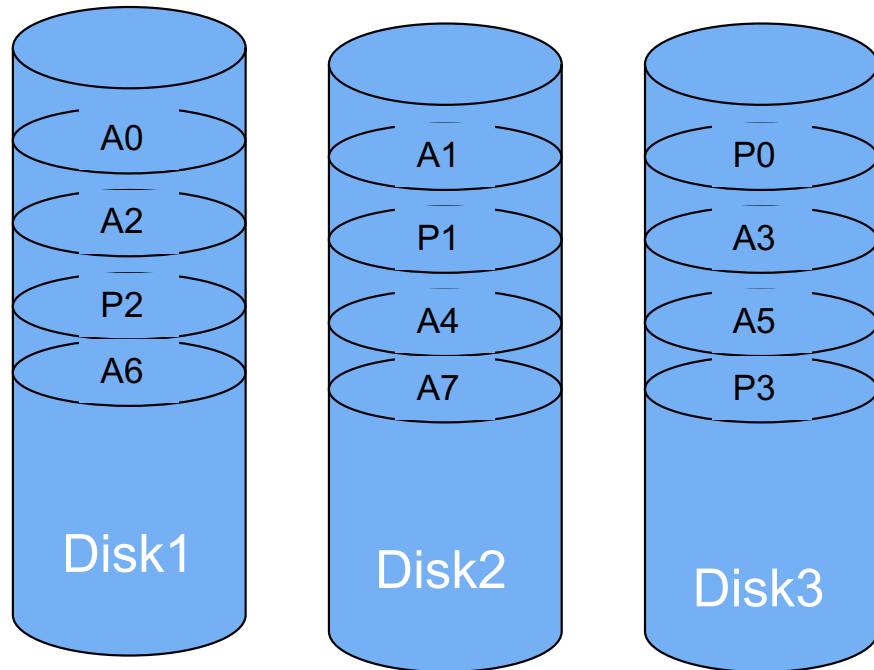
RAID 4 (Block level level striping)



- Striping takes place at block level
- Dedicated disk for parity blocks
- Provides higher throughput but very slow writes. **Disk3 has more writes as Parity needs to be updated for every data write.**
- MTTF increases substantially (same as RAID3)
- $P_i = A_{2i} \oplus A_{2i+1}$, here \oplus is an exclusive-or operator

- A means Block (4K or 8K bytes of storage)
- P is parity
- MTTF = Mean Time To Failure

RAID 5 with 3 disks (Block level striping)

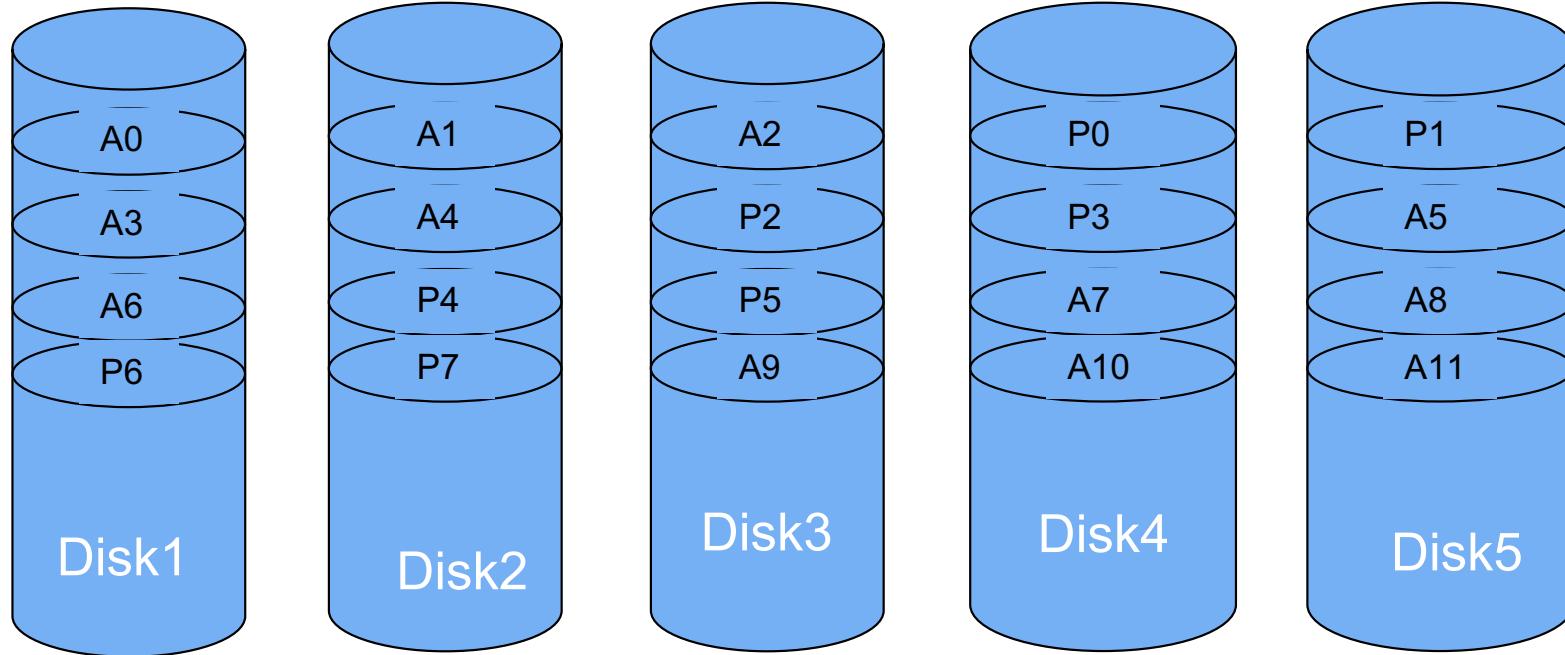


- A0,A1, A2, A3, .. are contiguous blocks of data of a file
- Striping takes place at block level
- Parity blocks are also striped
- Provides higher throughput but slower writes but **better than RAID 4** as parity bits are distributed among all disks and the number of write operations on average equal among all 3 disks.
- MTTF increases substantially (same as RAID3)
- $P_i = A_{2i} \oplus A_{2i+1}$, here \oplus is an exclusive-or operator

- A means Block (4K or 8K bytes of storage)
- P is parity
- MTTF = Mean Time To Failure

RAID 6 (Block level level striping)

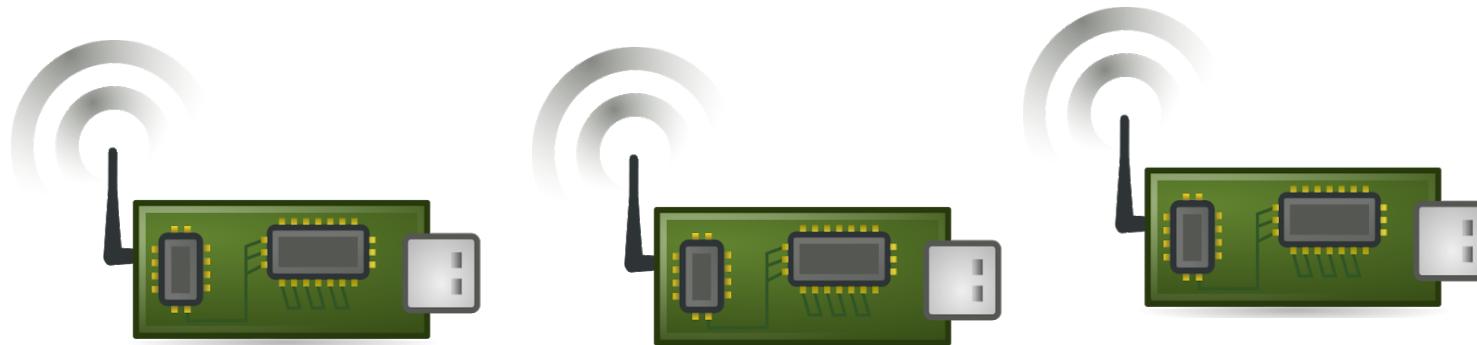
- A means Block (4K or 8K bytes of storage)
- P is parity



- Similar to RAID 5 except **two parity blocks used**.
- Reliability is of the order of **MTTF³/10**
- P0 and P1 are parity blocks for blocks A0, A1 and A2. These are computed in such way that any two disk failures can be safe to recover the data.

Fault tolerance by voting

Use more than one module, voting for higher reliability



Failvote - Stops if there is no majority agreement.

Failfast - Similar to failvote except the system senses which modules are available and uses the **majority of the available modules**.

- A 10 module Failfast system continues to operate until the failure of 9 modules whereas Failvote stops when 5 modules fail.
- Failfast system has better availability than failvoting (since failvote stops when there is no majority agreement).



Voting Contd

In Failfast, we are only concerned of majority among the working ones. We are assuming that we can tell which ones are working. Hence we can continue to operate until 2 working ones and if both agree we can proceed with the action. But if they differ the system stops:

- 0 devices are faulty, we have 10 working and we need at least 6 to agree
- 1 device is faulty, we have 9 working and we need at least 5 to agree
- 2 devices are faulty, we have 8 working and we need at least 5 to agree
- 3 devices are faulty, we have 7 working and we need at least 4 to agree
- 4 devices are faulty, we have 6 working and we need at least 4 to agree
- 5 devices are faulty, we have 5 working and we need at least 3 to agree
- 6 devices are faulty, we have 4 working and we need at least 3 to agree
- 7 devices are faulty, we have 3 working and we need at least 2 to agree
- 8 devices are faulty, we have 2 working and we need both to agree
- 9 devices are faulty, we have 1 working and we have to stop as nothing to compare!



Availability of failvote systems

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

Consider a system with modules each with MTTF of 10 years

Failvoting with 2 devices:

MTTF = $10/2 = 5$ years (system fails with 1 device failure)

Failvoting with 3 devices:

MTTF = $10/3$ for the first failure + $10/2$ for 2nd failure = 8.3 years.

Lower availability for higher reliability (multiple modules agreeing on a value means that value is more likely to be accurate/reliable)



Fault tolerance with repair

With repair of modules: the faulty equipment is **repaired with an average time of MTTR** (mean time to repair) as soon as a fault is detected

Probability of a particular module is not available

$$= \text{MTTR}/(\text{MTTF} + \text{MTTR})$$

$$\approx \text{MTTR}/\text{MTTF} \text{ if } \text{MTTF} \gg \text{MTTR}$$



Supermodule

In general, **a system with multiple hard disk drives** is expected to function with only one working disk

Use voting when multiple disks are working/available, but...

Still work when only one is available



Fault tolerance of a supermodule with repair

Prob that one module is not avail

=

Probability that (n-1) modules are unavailable ,

Probability that a particular i^{th} module fails,

Probability that the system fails with a particular i^{th} module failing last =

Probability that **a supermodule fails due to any one of the n modules failing last**, when other (n-1) modules are unavailable

Fault tolerance with repair examples

MTTF (days)	MTTR (days)	Number Devices	MTTF of the supermodule (days)	MTTF (years)
730	10	2	26645	73
730	10	3	1296723	3553
730	10	4	70995603	194509

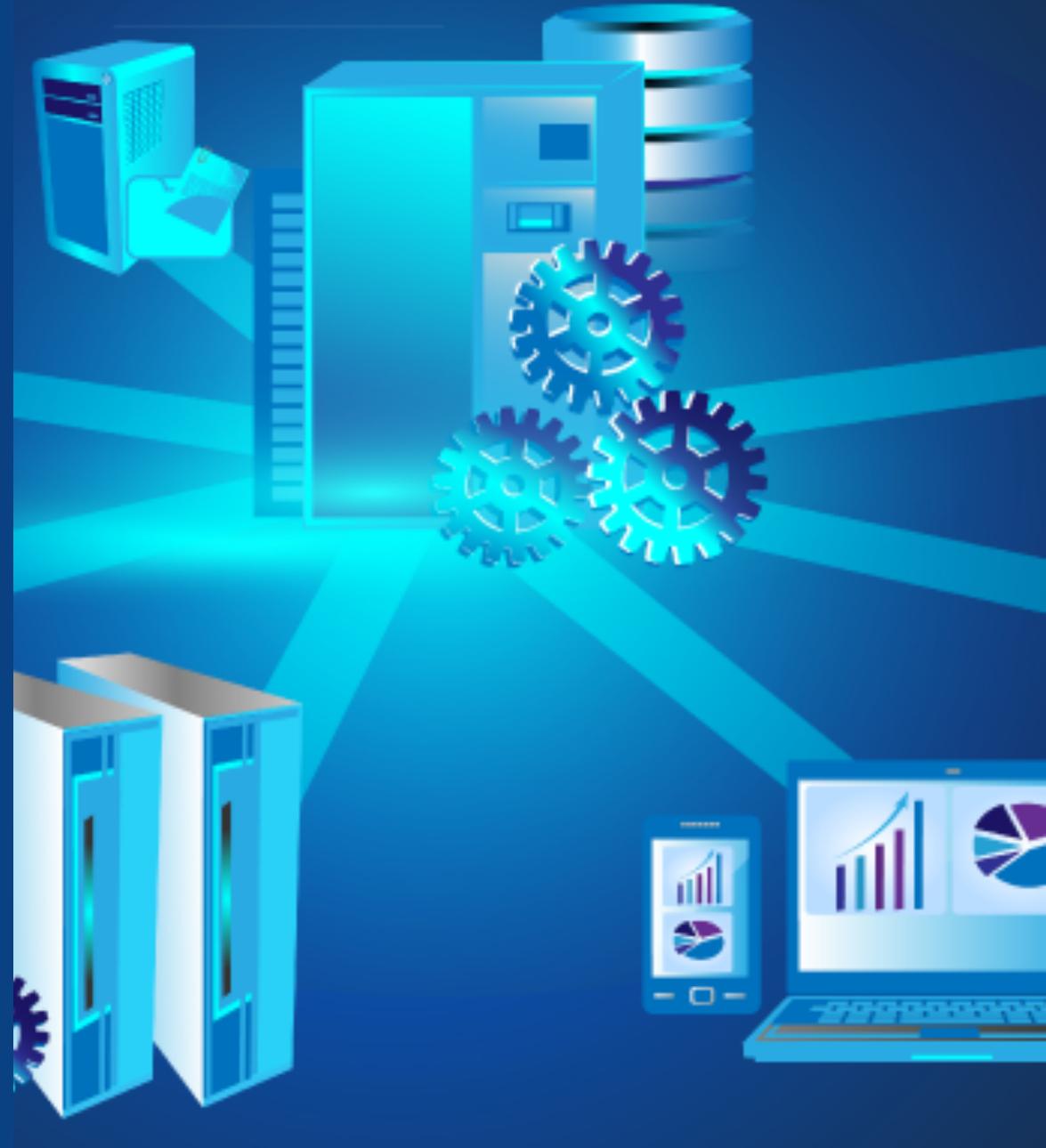


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

FAULT TOLERANCE CONTD.

COMP90050 Advanced Database Systems





Faults and Recovery

- o **Faults happen regardless**, leading to system failures that **may lead to lost data**
- o Regardless of how well a system is there is a life-time for every component
- o **Recovery mechanisms should exist in DBMSs to ensure data endures**
- o Techniques **to ensure database consistency** and **transaction atomicity** and **durability** despite failures are well researched in DBMSs
- o 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

How do we store data in a DBMS?

Volatile storage:

- does not survive system crashes
- such as main memory, cache memory

Nonvolatile storage / Persistent storage :

- survives system crashes **most of the time but by itself cannot be used for recovery**
- examples: **disk**, tape, flash memory

Stable storage:

- a **mythical** form of storage that survives all failures
- approximated by maintaining multiple copies on distinct nonvolatile media



Stable-Storage Basics Contd.

Maintain multiple copies of each block on separate disks/systems

- copies can be at remote sites to protect against disasters such as fires

Failure during data transfer can still result in inconsistent copies

For protecting storage media from failure during data transfer we **write two copies of each disk block** onto the same system:

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.



Data Access Details are Even more Complicated with Use of Buffers

There are actually two types of data blocks we need to consider:

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 data there.

In fact, in most real systems each transaction T_j 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 .

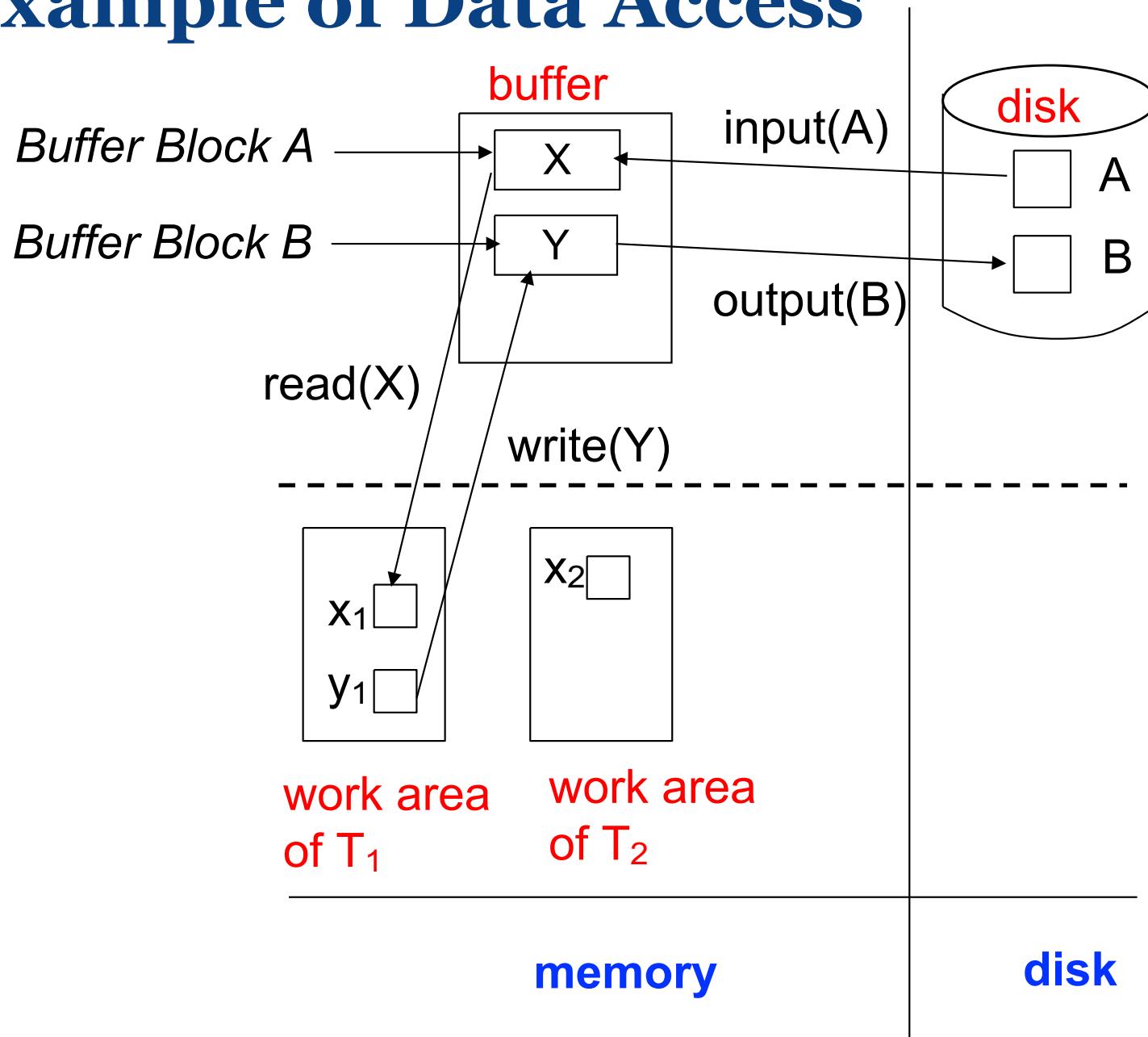


Data Access Contd.

Transactions transfer data between system buffer blocks and private work-area using the following operations :

- **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
- Both these commands may necessitate the issue of an **input(B_X)** instruction before the assignment, if the block B_X in which X resides is not already in memory
- **So our initial view of read and write has to be considered within a memory hierarchy**
- Thus transactions perform **read(X)** while accessing X for the first time
- **All subsequent accesses are to the local copy**
- After say last access, transaction can execute **write(X)**
- **output(B_X)** need not immediately follow **write(X)**
- System can perform the **output** operation later

Example of Data Access





Cyclic Redundancy Check

- A method heavily used in network communications
- DBMSs borrow this method to increase durability on disks
- It further complicates matters but increases durability
- It is a method, similar to parity computations we saw before, to catch problems in written data on disks
- Most errors in communications or in our case on disk happen contiguously, that is in bursts
- CRC can be used detect burst errors on disks with many bits flipping
- In fact the parity bit we saw can be seen as a 1 bit CRC computation
- CRC/parity can be used even when we do not use a RAID design on disks



Now we know basics of data access: How do recovery systems work?

- Consider transaction T_i that transfers \$50 from account A to account B ; goal is either to perform all database modifications made by T_i or none at all properly
- Several output operations may be required for T_i (to output A and B)
- A failure may occur after one of these modifications have been made but before all of them are made
- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
- Recovery systems are built to deal with such cases



Types of Recovery Systems

We study two main approaches:

- **log-based recovery**
- **shadow-paging**



Log-Based Recovery

A **log** is kept on stable storage:

- The log is a **sequence of log records**, and mainly maintains a record of update activities on the database
- There is actually a lot of other things saved to the log as well:
 - E.g., when transaction T_i starts, it registers itself by writing a $\langle T_i \text{ start} \rangle$ log record

Before T_i executes write(X):

- A log record $\langle T_i, X, V_1, V_2 \rangle$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X

When T_i finishes its last statement :

- Log record $\langle T_i \text{ commit} \rangle$ is written

Lets assume for now **that log records** are written directly to stable storage that is, they **are not buffered**



Two types of Logs to consider

Two approaches while using logs leads to two types of logs:

- **Deferred database modification**
- **Immediate database modification**

The **deferred database modification** scheme **records all modifications to the log**, but **defers all the writes to after partial commit**.

- Transaction starts by writing $\langle T_i \text{ start} \rangle$ record to log
- A **write(X)** operation results in a log record $\langle T_i, X, V \rangle$ being written
- Where V is the new value for X
- **Old value is not needed** for this scheme
- The write is not performed on X at this time, but is deferred
- When T_i partially commits, $\langle T_i \text{ commit} \rangle$ is written to the log
- The log records are read and used to actually execute the previously deferred writes



Deferred Database Mod. Contd.

- During recovery after a crash , a transaction needs to be redone if and only if both $\langle T_i \text{ start} \rangle$ and $\langle T_i \text{ commit} \rangle$ are there in the log
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values

Lets consider transactions T_0 and T_1 (assume T_0 executes before T_1):

$T_0:$	read (A)	$T_1 : \text{read (}Q\text{)}$
	$A:- A - 50$	$C:- C - 100$
	Write (A)	write (Q)
	read (B)	
	$B:- B + 50$	
	write (B)	

Deferred Database Modification

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

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$

$\langle T_1, C, 600 \rangle$

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$

$\langle T_1, C, 600 \rangle$

$\langle T_1 \text{ commit} \rangle$

(a)

(b)

(c)

If log on stable storage at time of crash is as in case:

- (a) No redo actions need to be taken
- (b) $\text{redo}(T_0)$ must be performed since $\langle T_0 \text{ commit} \rangle$ is present
- (c) $\text{redo}(T_0)$ must be performed followed by $\text{redo}(T_1)$ since $\langle T_0 \text{ commit} \rangle$ and $\langle T_1 \text{ commit} \rangle$ are present



Immediate Database Modification

The immediate database modification scheme

- Allows database **updates of an uncommitted transaction to be made as the writes are issued**
- Since undoing may be needed, **update logs must have both old value and new value**

Update log record must be written before database item is written

- We assume that the log record is output directly to stable storage
- Can be extended to postpone log record output, so long as prior to execution of an **output(B)** operation for a data block B , all log records corresponding to item B must be flushed to stable storage
- Output of updated blocks can take place at any time before or after commit
- Order in which blocks are output can be different from the order in which they are written



Immediate Database Modification Example

Log

Write

Output

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 1000, 950 \rangle$

$\langle T_0, B, 2000, 2050 \rangle$

$A = 950$
 $B = 2050$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$

$\langle T_1, C, 700, 600 \rangle$

$C = 600$

B_B, B_C

$\langle T_1 \text{ commit} \rangle$

B_A

Note: B_X denotes block containing X



Immediate Database Modification

Recovery procedure has two operations instead of one:

- **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
- **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

Both operations must be **idempotent**

- That is, even if the operation is executed multiple times the effect is the same as if it is executed once
 - Needed since operations may get re-executed during recovery

When recovering after failure:

- Transaction T_i needs to be undone if the log contains the record $\langle T_i \text{ start} \rangle$, but does not contain the record $\langle T_i \text{ commit} \rangle$.
- Transaction T_i needs to be redone if the log contains both the record $\langle T_i \text{ start} \rangle$ and the record $\langle T_i \text{ commit} \rangle$.
- Undo operations are performed first, then redo operations.



Immediate DB Modification Recovery Example

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

< T_0 start>

< T_0 , A, 1000, 950>

< T_0 , B, 2000, 2050>

< T_0 start>

< T_0 , A, 1000, 950>

< T_0 , B, 2000, 2050>

< T_0 commit>

< T_1 start>

< T_1 , C, 700, 600>

< T_0 start>

< T_0 , A, 1000, 950>

< T_0 , B, 2000, 2050>

< T_0 commit>

< T_1 start>

< T_1 , C, 700, 600>

< T_1 commit>

(a)

(b)

(c)

Recovery actions in each case above are:

- (a) undo (T_0): B is restored to 2000 and A to 1000.
- (b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
- (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

Problems in recovery procedure :

1. searching the entire log is time-consuming
2. we might unnecessarily redo transactions which have already output their updates to the database

Instead periodically perform **checkpointing**:

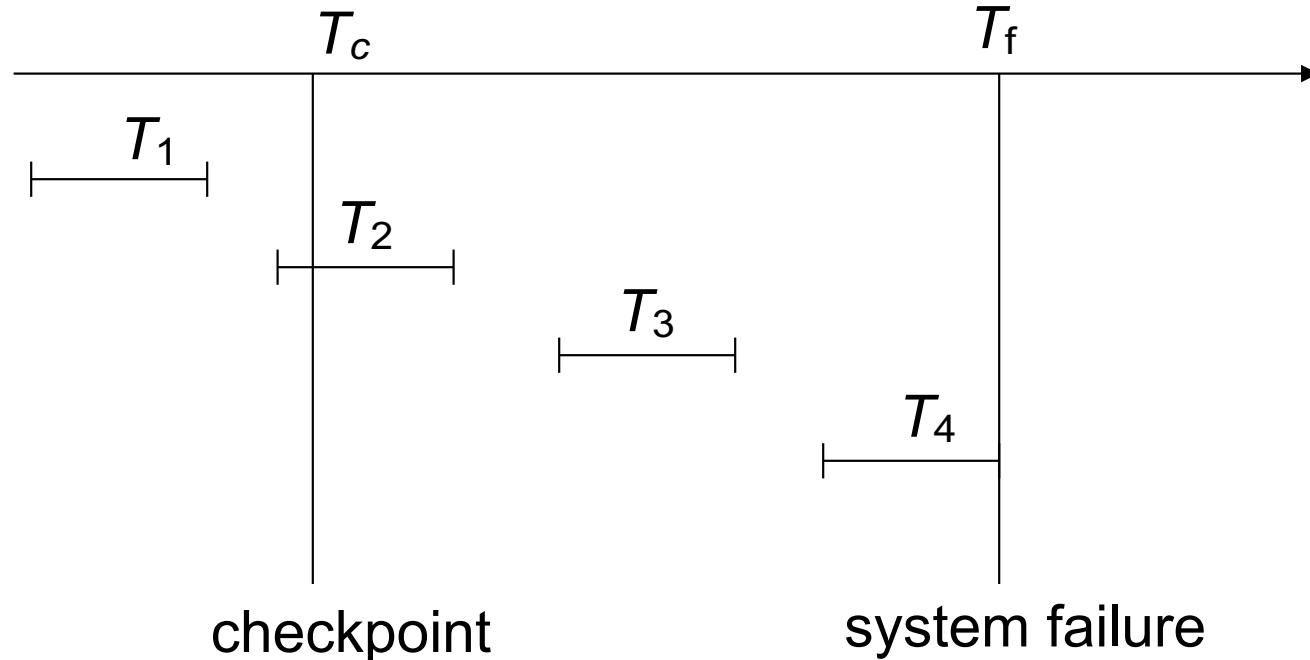
1. Output all log records currently residing in main memory onto stable storage
2. Output all modified buffer blocks to the disk
3. Write a log record <checkpoint> onto stable storage



Checkpointing Contd.

1. Scan backwards from end of log to find the most recent <checkpoint> record.
2. Continue scanning backwards till < T_i start> is found.
3. Need only consider the part of log following **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
4. For all transactions (starting from T_i or later) with no < T_i commit>, execute **undo**(T_i). (Done only in case of immediate modification.)
5. Scanning forward in the log, for all transactions starting from T_i or later with a < T_i commit>, execute **redo**(T_i).

Example of Checkpointing



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

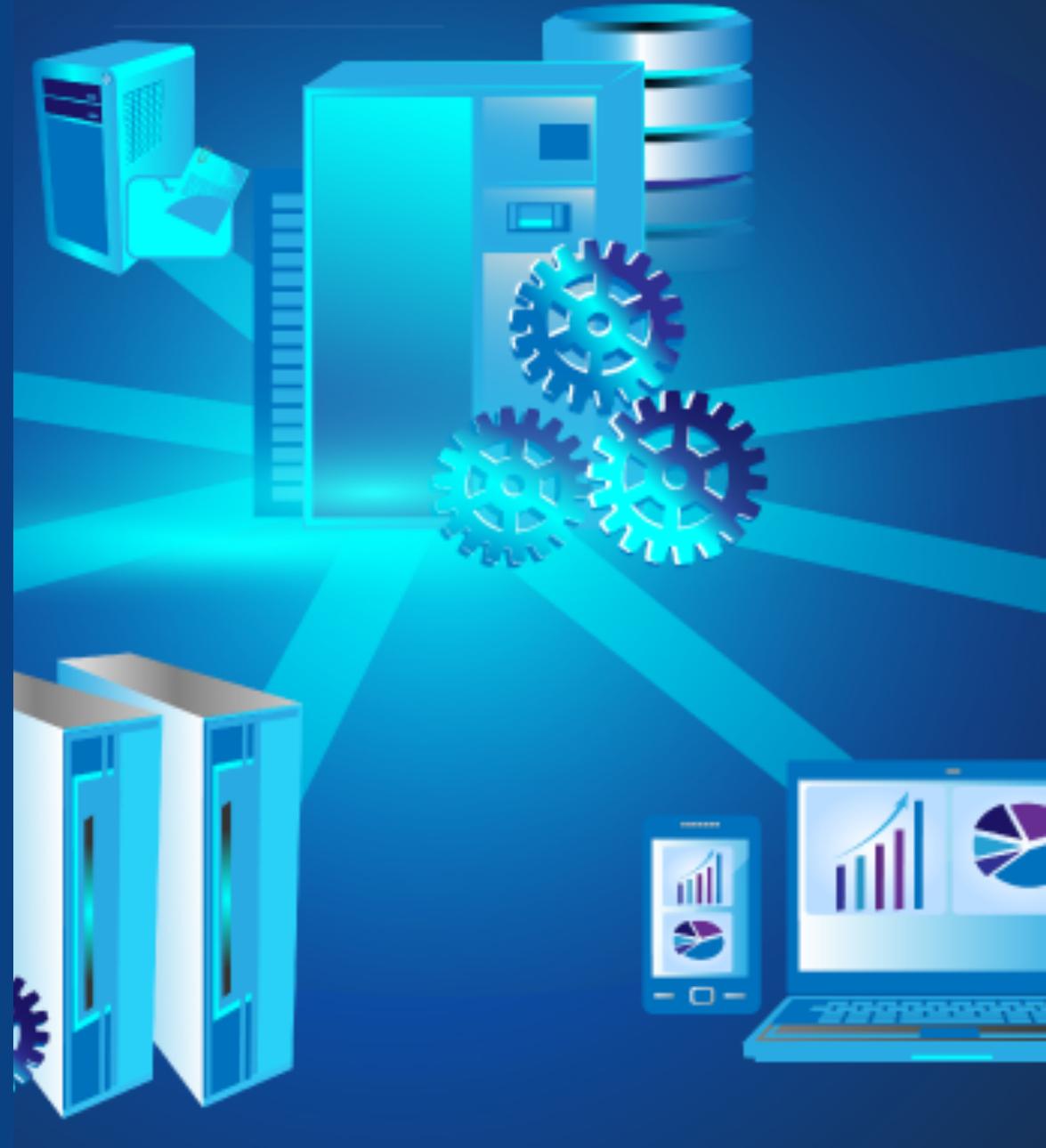


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

FAULT TOLERANCE CONTD.

COMP90050 Advanced Database Systems





Checkpoints

Problems in recovery procedure :

1. searching the entire log is time-consuming
2. we might unnecessarily redo transactions which have already output their updates to the database

Instead periodically perform **checkpointing**:

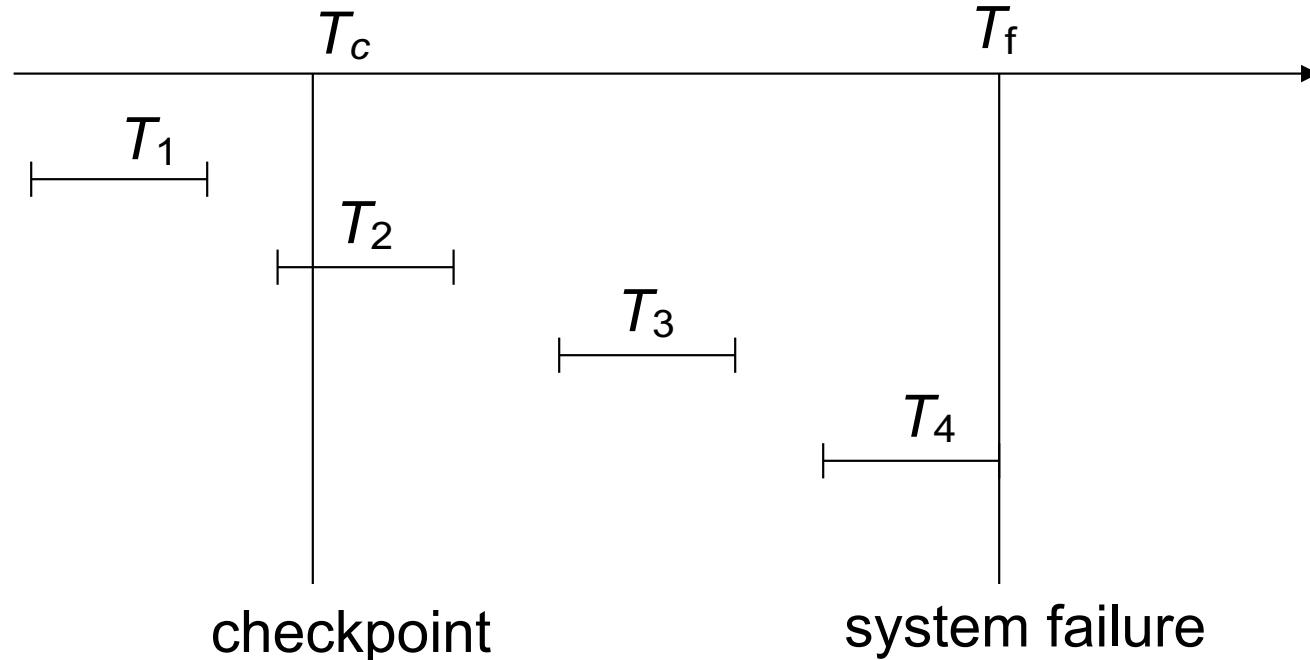
1. Output all log records currently residing in main memory onto stable storage
2. Output all modified buffer blocks to the disk
3. Write a log record <checkpoint> onto stable storage



Checkpointing Contd.

1. Scan backwards from end of log to find the most recent <checkpoint> record.
2. Continue scanning backwards till < T_i start> is found.
3. Need only consider the part of log following **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
4. For all transactions (starting from T_i or later) with no < T_i commit>, execute **undo**(T_i). (Done only in case of immediate modification.)
5. Scanning forward in the log, for all transactions starting from T_i or later with a < T_i commit>, execute **redo**(T_i).

Example of Checkpointing



T_1 can be ignored (updates already output to disk due to checkpoint)

T_2 and T_3 redone

T_4 undone



What about Concurrent Transactions

We modify the log-based recovery schemes to **allow multiple transactions to execute concurrently**:

- 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 concurrency control **using strict two-phase locking**:

- 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?

The **checkpointing technique** and actions taken on recovery **have to be changed**:

- Since **several transactions may be active when a checkpoint** is performed



Recovery with Concurrent Transactions

Checkpoints are performed as before, except that the checkpoint log record is now of the form: $\langle \text{checkpoint } L \rangle$

where **L** is the list of transactions active at the time of the checkpoint

- We assume no updates are in progress while the checkpoint is carried out

When the system recovers from a crash, what does it do?

1. Initialize *undo-list* and *redo-list* to empty
2. Scan the log backwards from the end , stopping when the first $\langle \text{checkpoint } L \rangle$ record is found
For each record found during the backward scan:
 - H if the record is $\langle T_i \text{commit} \rangle$, add T_i to *redo-list*
 - H if the record is $\langle T_i \text{start} \rangle$, then if T_i is not in *redo-list*, add T_i to *undo-list*
3. For every T_i in L , if T_i is not in *redo-list*, add T_i to *undo-list*



Recovery with Concurrent Transactions

At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.

Recovery now continues as follows:

1. Scan log backwards from most recent record, stopping when $\langle T_i \text{ start} \rangle$ records have been encountered for every T_i in *undo-list*.
 - n During the scan, perform undo for each log record that belongs to a transaction in *undo-list*
2. Locate the most recent $\langle \text{checkpoint } L \rangle$ record
3. Scan log forwards from the $\langle \text{checkpoint } L \rangle$ record till the end of the log
 - n During the scan, perform redo for each log record that belongs to a transaction on *redo-list*



Log Record Buffering

Log record buffering: in many systems **log records are buffered in main memory as well**, instead of being output directly to stable storage.

- Several log records can thus be output using a single output operation, reducing the I/O cost.
- 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.



Log Record Buffering Contd.

These **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 $\langle T_i \text{ commit} \rangle$ 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



More on 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**
- 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 (WAL)

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**
- Before a block is output to disk, the system acquires an exclusive latch on the block
 - **Ensures no update can be in progress on the block**



Buffer Management Types

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 Contd.

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



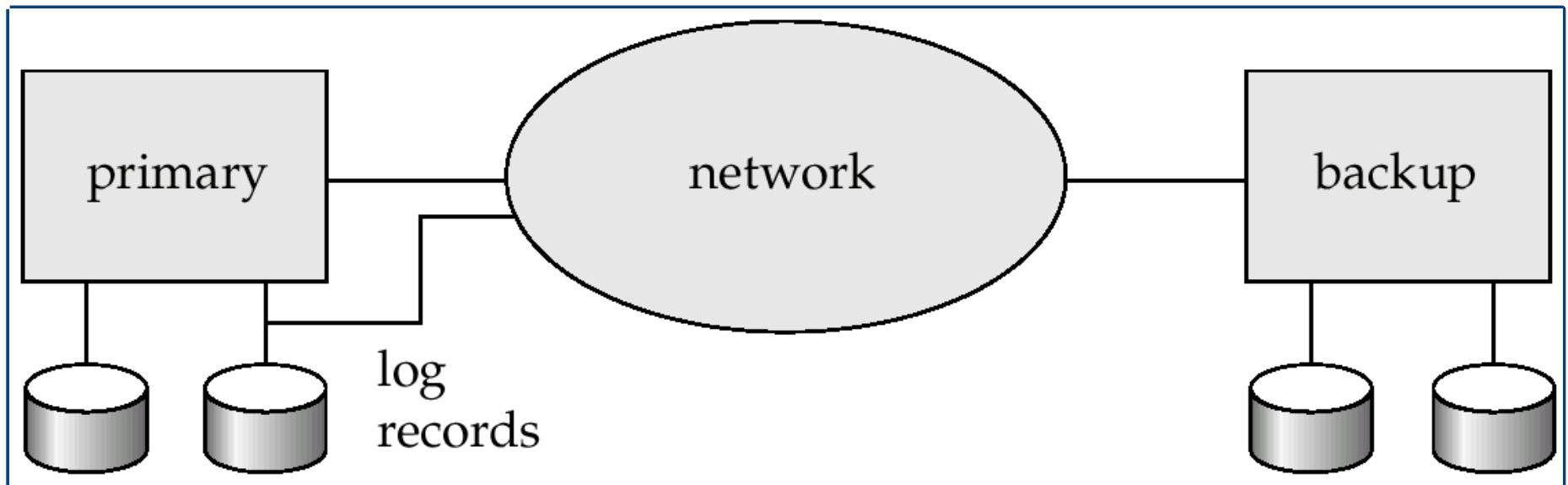
Advanced Recovery: Logical Logging

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)

Other Considerations: 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 Contd.

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
- **Use heart-beat messages**

How to Transfer of control :

- To take over control, **backup site first perform recovery using its copy of the database and all the log records** it has received from primary
 - Thus, completed transactions are redone and incomplete transactions are rolled back
- When the backup site takes over processing **it becomes the new primary**



Remote Backup Systems Contd.

Time to recover:

- To reduce delay in takeover, **backup site periodically processes 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**



Remote Backup Systems Contd.

Ensure durability of updates by delaying transaction commit until update is logged at backup

But we can 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 its commit log record is written at the primary

- Better availability than two-very-safe; avoids problem of lost transactions in one-safe.



Alternative to Logs: Shadow Paging

Shadow paging is an alternative to log-based recovery

Idea: maintain **two page tables** during the lifetime of a transaction –the **current page table**, and the **shadow page table**

Store the **shadow page table in nonvolatile storage**, such that state of the database prior to transaction execution may be recovered

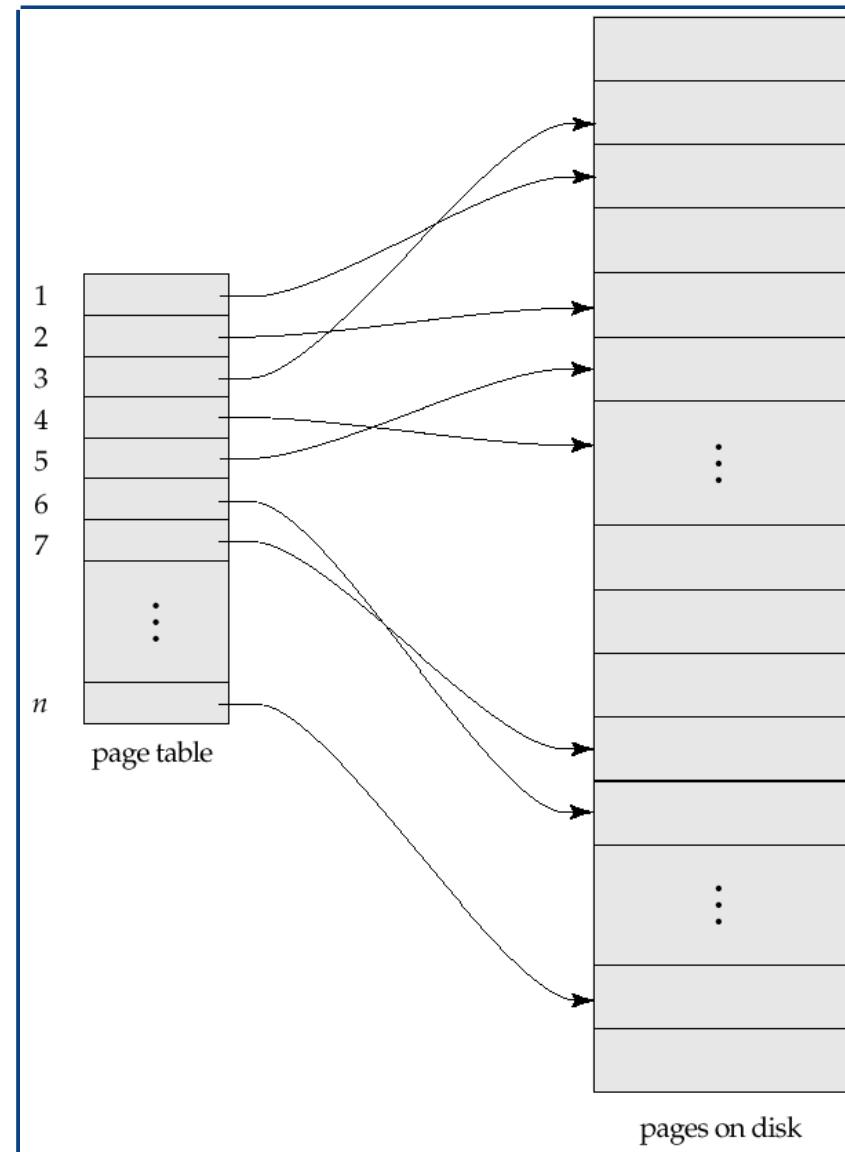
- Shadow page table is never modified during execution

To start with, both the page tables are identical. **Only current page table is used for data item accesses** during execution of the transaction

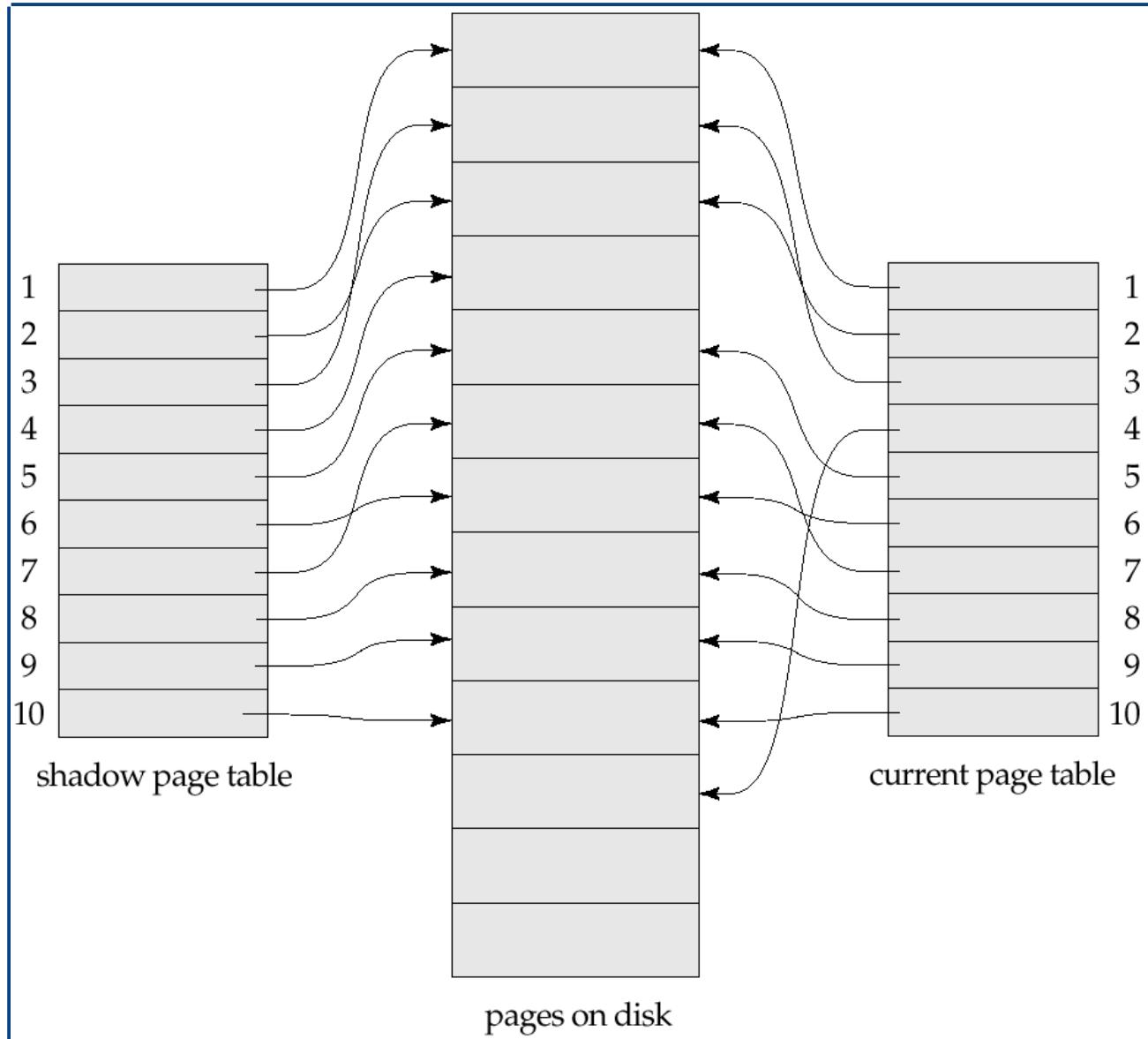
Whenever **any page is about to be written** :

- A **copy of this page is made onto an unused page**
- The **current page table is then made to point to the copy**
- The **update is performed on the copy**

Sample Page Table



Example of Shadow Paging



Shadow and current page tables after write to page 4



Shadow Paging Contd.

To commit a transaction :

1. **Flush all modified pages in main memory to disk**
2. **Output current page table to disk**
3. **Make the current page table the new shadow page table** , as follows:
 - keep a pointer to the shadow page table at a fixed (known) location on disk.
 - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk.

Once pointer to shadow page table has been written, transaction is committed.

No recovery is needed after a crash — new transactions can start right away, using the shadow page table.

Pages not pointed to from current/shadow page table should be freed (garbage collected).



Shadow Paging Contd.

Advantages of shadow-paging over log-based schemes:

- **No overhead of writing log records**
- **Recovery is trivial**

Disadvantages:

Copying the entire page table is very expensive when the page table is large

- Can be reduced by using a page table structured like a B ⁺-tree

But **commit overhead is high** even with above extension

- **Need to flush every updated page, and page table**

And **data gets fragmented** (related pages get separated on disk)

After every transaction completion, the database pages containing old versions of modified data **need to be garbage collected**

Hard to extend algorithm to allow transactions to run concurrently

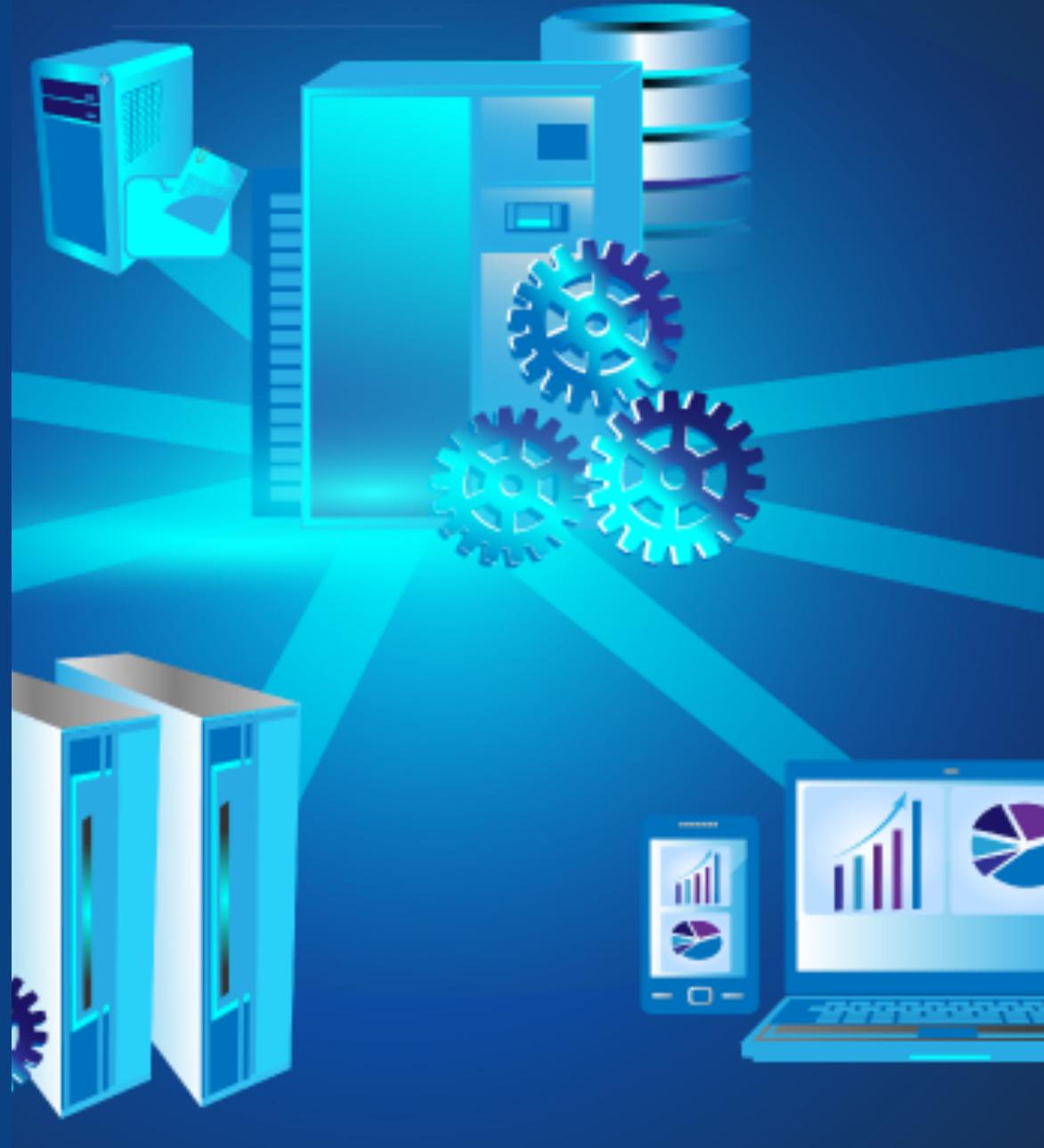


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

SPECIALIZED SYSTEMS

COMP90050 Advanced Database Systems





Data Warehousing

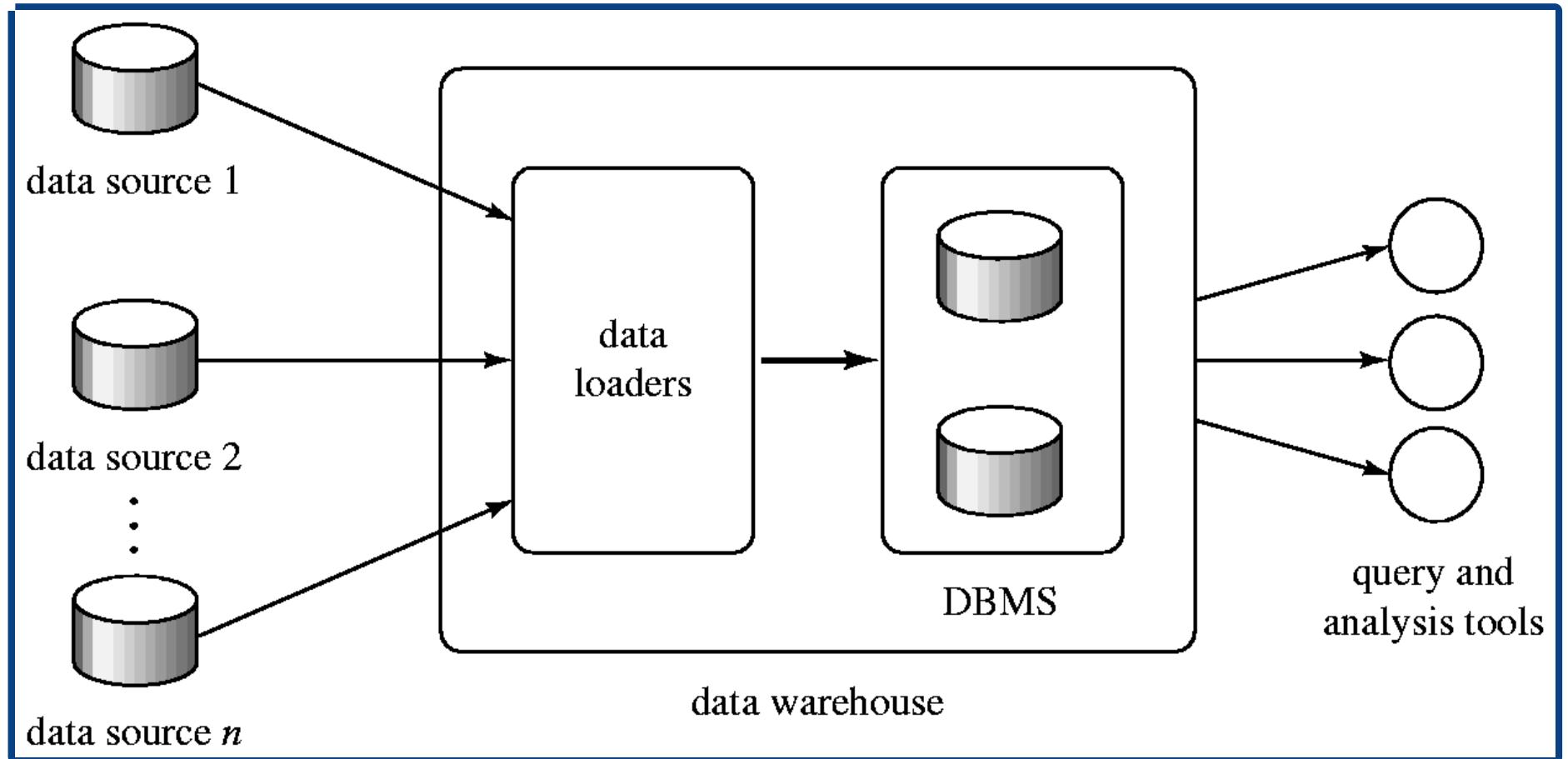
Data sources often store only current data , not historical data

Corporate **decision making requires** a unified view of all organizational data, including **historical data**

A **data warehouse** is a repository (archive) of information gathered from multiple sources, stored under a unified schema, at a single site

- **Greatly simplifies querying, permits study of historical trends**
- **Shifts decision support query load away from transaction processing systems**

Data Warehousing





Design Issues

When and how to gather data:

- **Source driven architecture**: data sources transmit new information to warehouse, either continuously or periodically (e.g. at night)
- **Destination driven architecture**: warehouse periodically requests new information from data sources
- Keeping warehouse exactly synchronized with data sources (e.g. **using two-phase commit**) is too expensive
 - Usually **OK to have slightly out-of-date** data at warehouse
 - Data/updates are periodically downloaded from online transaction processing (**OLTP**) systems (most of the DBMS work we have seen so far)



More Warehouse Design Issues

What schema to use

- Schema integration

Data cleansing

- E.g. correct mistakes in addresses (misspellings, zip code errors)
- Merge address lists from different sources and purge duplicates

How to propagate updates

- Warehouse schema may be a (materialized) view of schema from data sources

What data to summarize

- Raw data may be too large to store
- Aggregate values (totals/subtotals) often suffice
- Queries on raw data can often be transformed by query optimizer to use aggregate values



OnLine Analytical Processing

Online Analytical Processing (OLAP)

- Interactive analysis of data, commonly for data analysis as mentioned before
- Data to be summarized and viewed in different ways with negligible delay
- Extremely difficult to do with OLTP we saw with raw database tables

Data that can be modeled as “dimension attributes and measure attributes” are called **multidimensional data**:

- **Measure attributes**
 - measure some value
 - can be aggregated upon
 - e.g. the attribute *number* of the *sales* relation
- **Dimension attributes**
 - define the dimensions on which measure attributes (or aggregates thereof) are viewed
 - e.g. the attributes *item_name*, *color*, and *size* of the *sales* relation



Cross Tabulation of sales by *item-name* and *color*

size:

	<i>color</i>			Total
	dark	pastel	white	
<i>item-name</i>				
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

The table above is an example of a **cross-tabulation (cross-tab)**, also referred to as a **pivot-table**:

- Values for one of the dimension attributes form the row headers
- Values for another dimension attribute form the column headers
- Other dimension attributes are listed on top
- Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell



Relational Representation of Cross-tabs

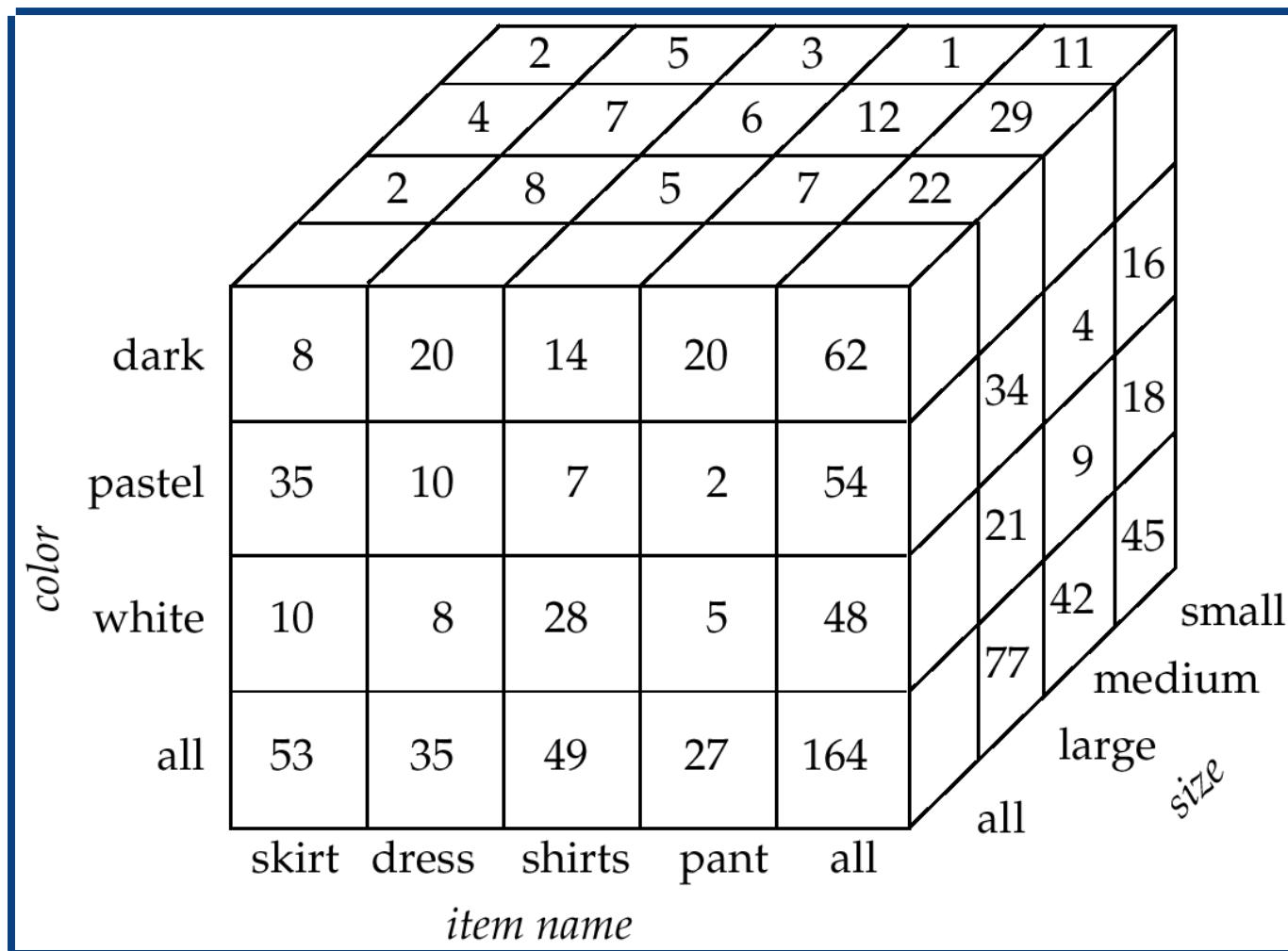
- n Cross-tabs can be represented as relations
 - n We use the value all to represent aggregates

item-name	color	number
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	all	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	all	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	all	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	all	27
all	dark	62
all	pastel	54
all	white	48
all	all	164



Data Cube

- n A **data cube** is a multidimensional generalization of a cross-tab
 - n Can have n dimensions; we show 3 below
 - n Cross-tabs can be used as views on a data cube





Online Analytical Processing

Pivoting: changing the dimensions used in a cross-tab

Slicing: creating a cross-tab for fixed values only

- Sometimes called **dicing**, particularly when values for multiple dimensions are fixed

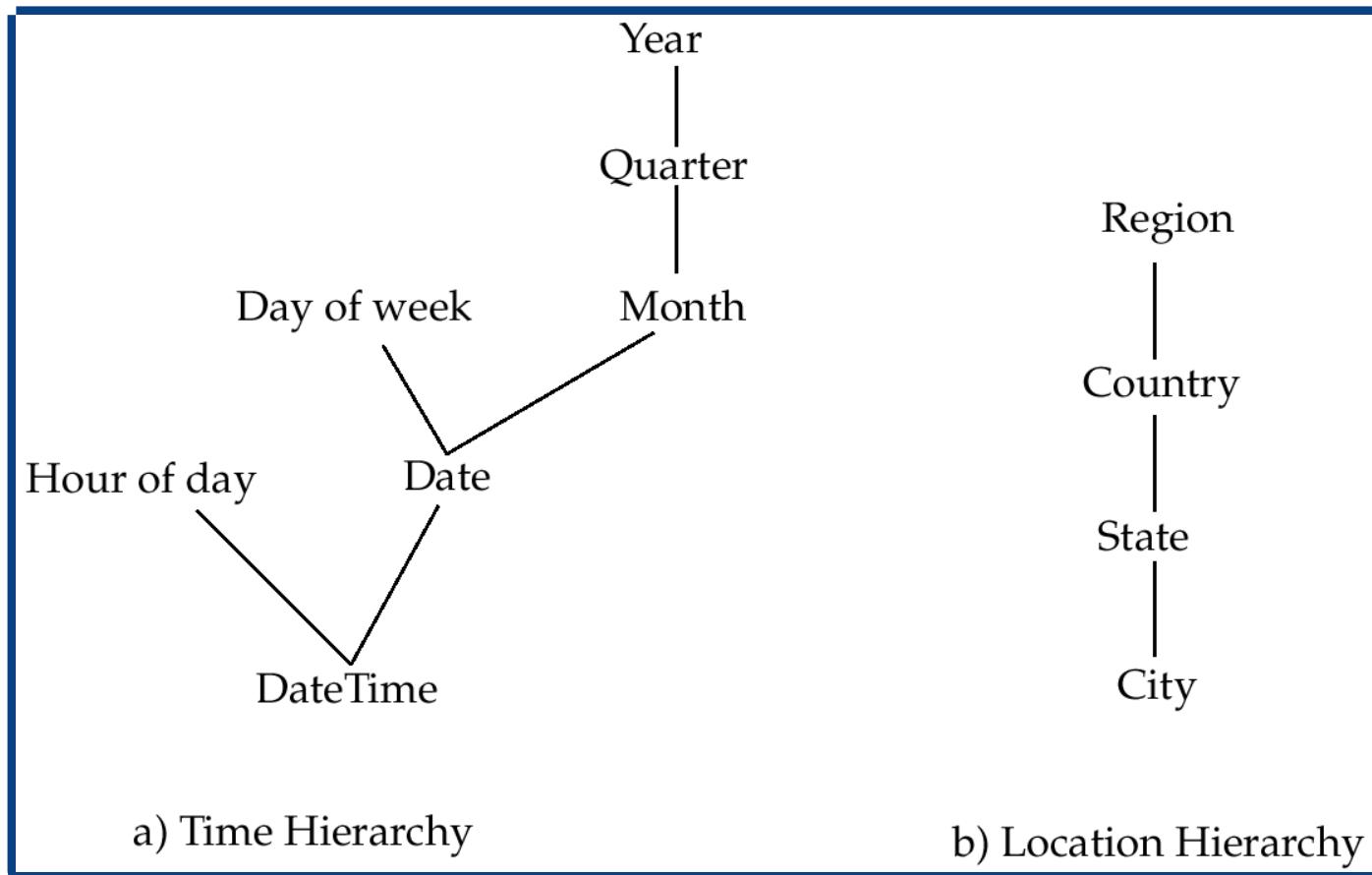
Rollup: moving from finer-granularity data to a coarser granularity

Drill down: The opposite operation - that of moving from coarser-granularity data to finer-granularity data

Hierarchies on Dimensions

n **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail

H E.g. the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



Cross Tabulation with Hierarchy

- n Cross-tabs can be easily extended to deal with hierarchies
 - H Can drill down or roll up on a hierarchy

<i>category</i>	<i>item-name</i>	dark	pastel	white	total
womenswear	skirt	8	8	10	53
	dress	20	20	5	35
	subtotal	28	28	15	88
menswear	pants	14	14	28	49
	shirt	20	20	5	27
	subtotal	34	34	33	76
total		62	62	48	164



OLAP Implementation

The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems

OLAP implementations using only relational database features using rows and columns and no precalculated data cubes called **relational OLAP (ROLAP)** systems

Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems



OLAP Implementation Contd

Early OLAP systems precomputed *all* possible aggregates in order to provide online response

Space and time requirements for doing so can be very high

- It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
- E.g. we can compute aggregate on (*item-name, color*) from an aggregate on (*item-name, color, size*)

Several optimizations available for computing multiple aggregates



Next: Information Retrieval Systems

Information retrieval (IR) systems use a simpler data model than database systems

- Information organized as a **collection of documents**
- **Documents are unstructured**, no schema like OLTP
- Information retrieval **locates relevant documents**, on the basis of user input such as **keywords** or example documents
 - e.g., find documents containing the words “OLTP Systems”
- As such **querying is much more focused**
- Can be used even on textual descriptions provided with non-textual data such as images
- **Web search engines** are the most familiar example of IR systems
- And they run much faster than DBMS for the purpose they were built



Information Retrieval Systems vs DBMS

Differences from database systems:

- IR systems don't deal with transactional updates (including concurrency control and recovery)
- Database systems deal with structured data, with schemas that define the data organization
- IR systems deal with some querying issues not generally addressed by database systems
 - Approximate searching by keywords
 - Ranking of retrieved answers by estimated degree of relevance



Indexing of Documents in IR

Inverted indices are used in this domain, for example:

Text 1: I would like to stay here

Text 2: You would like to stay there

Index:

Here > Text 1

I > Text 1

Like > Text 1, Text 2

Stay > Text 1, Text 2

There > Text 2

To > Text 1, Text 2

Would > Text 1, Text 2

You > Text 2

Inverted index may record:

Keyword locations within document to allow proximity based ranking

Counts of number of occurrences of keyword to compute term frequency

and operations: Finds documents that contain all of K_1, K_2, \dots, K_n .

or operations: Find documents that contain at least one of K_1, K_2, \dots, K_n



Next: In Memory Databases

High-performance hardware and parallelism help improve the rate of transaction processing, but are insufficient to obtain very high performance:

- **Disk I/O is still a bottleneck**
- **Parallel transactions may attempt to read or write the same data item**, resulting in data conflicts that reduce effective parallelism

We can reduce the degree to which a database system is disk bound by **increasing the size of the database buffer**



Main Memory and Databases

Commercial 64-bit systems can support **main memories of tens of gigabytes which is adequate for many databases**, e.g., simple employee databases

Memory resident data allows faster processing of transactions

Further considerations:

- Logging is a bottleneck when transaction rate is high which needs a consideration
- Using group-commit can reduce number of output operations
- If the update rate for modified buffer blocks is high, the disk data-transfer rate could become a bottleneck
- If the system crashes, all of main memory is lost though



Main Memory Database Optimizations Opportunities

To reduce space overheads use different data structures, i.e. do not use indices built for disks

No need to pin buffer pages in memory before data are accessed , since buffer pages will never be replaced

Design query-processing techniques to minimize space overhead - avoid exceeding main memory limits during query evaluation

Improve implementation of operations such as locking and latching , so they do not become bottlenecks

Optimize recovery algorithms, since pages rarely need to be written out to make space for other pages

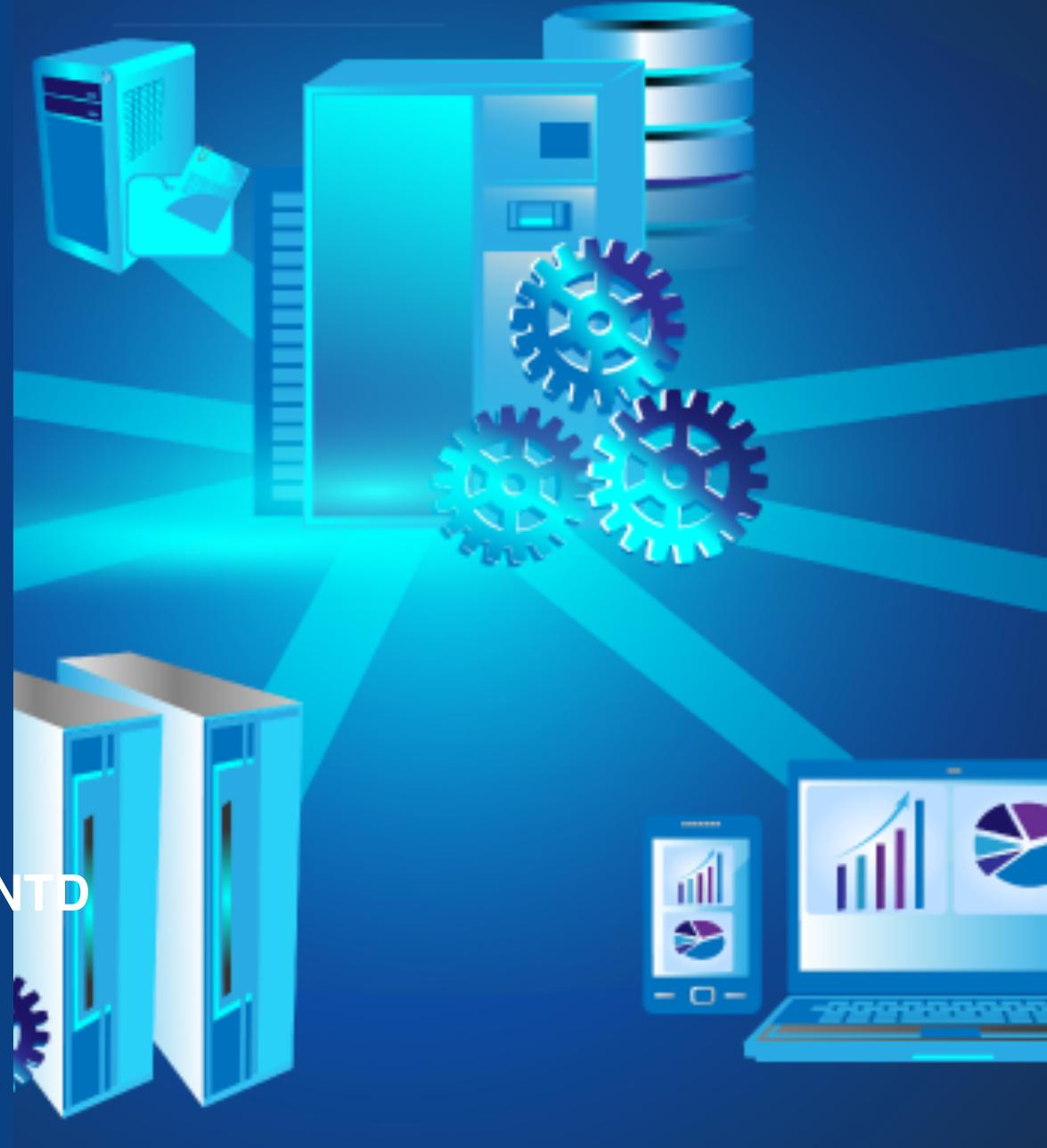


THE UNIVERSITY OF
MELBOURNE

Semester 1,
2023

SPECIALIZED SYSTEMS CONTD

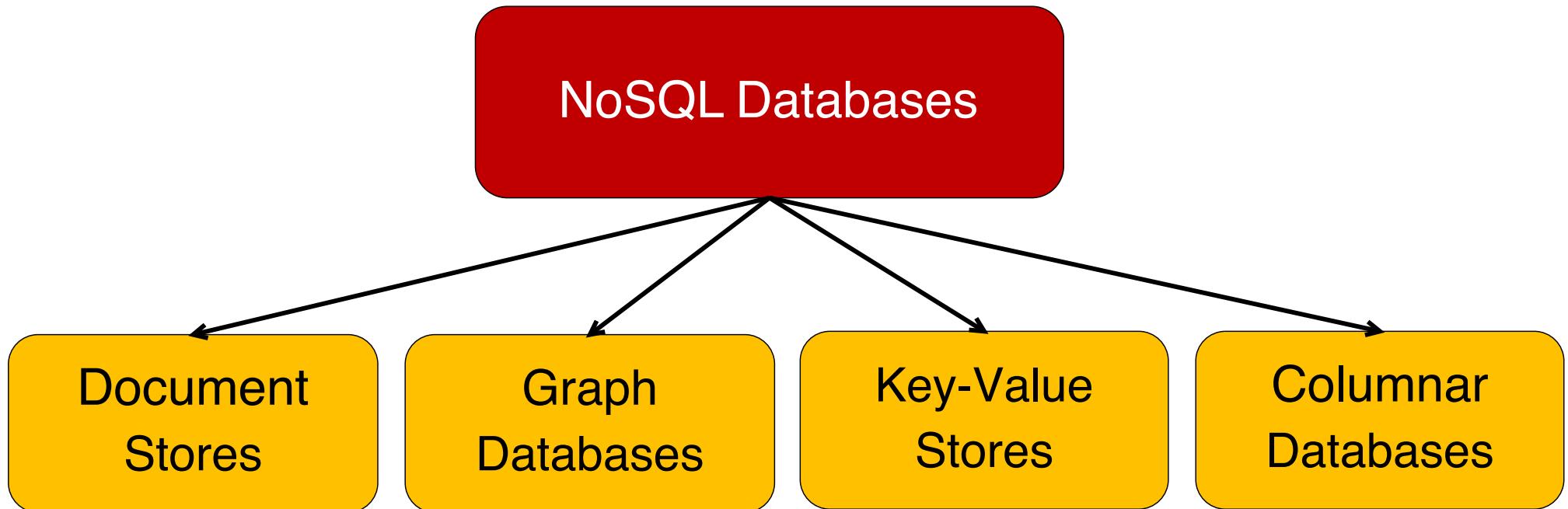
COMP90050 Advanced Database Systems





NoSQL Databases

- Used with big data
- Things are simplified
- Loose consistency**



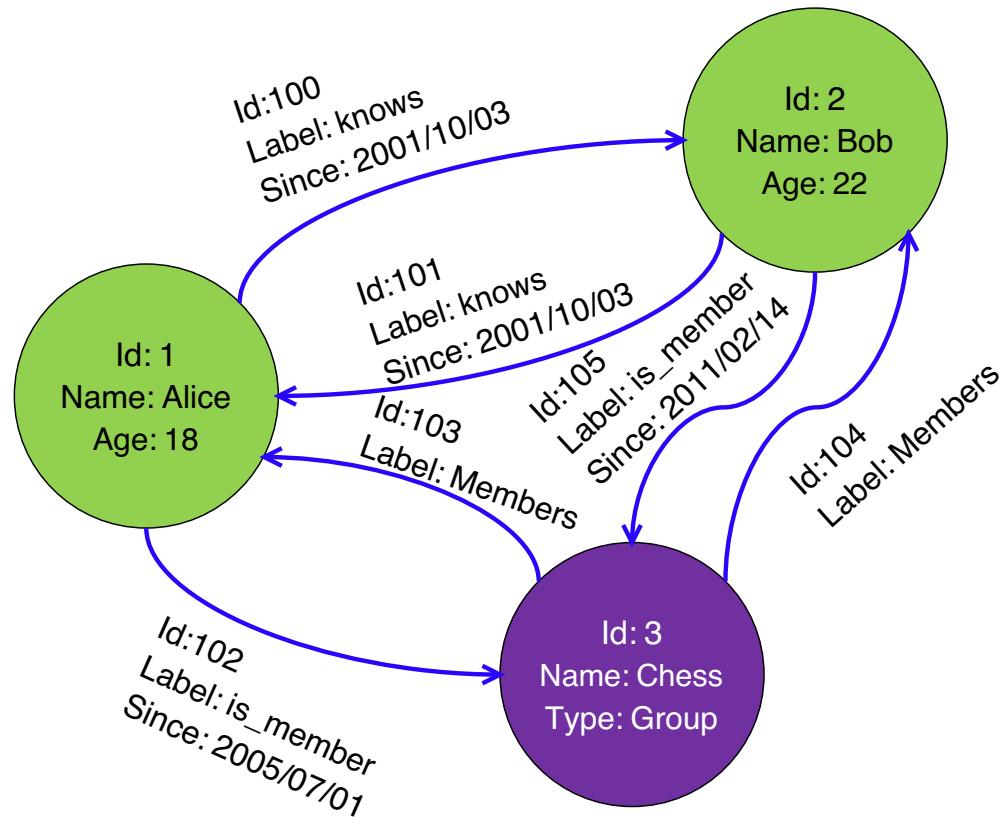


Document Stores

- ? Documents are stored in some standard format or encoding (e.g., XML, JSON, PDF or Office Documents)
- ? These are typically referred to as Binary Large Objects (BLOBs)
- ? Documents can be indexed, and in their own way
- ? This allows document stores to outperform traditional systems
- ? E.g., MongoDB and CouchDB

Graph Databases

Data are represented as vertices and edges



Graph databases are powerful for graph-related queries (*e.g., find the shortest path between two elements*)

Specialized algorithms for such special queries outperforms classical systems

E.g., Neo4j and VertexDB



Key-Value Stores

- ? Keys are mapped to (possibly) more complex value (e.g., lists)
- ? Keys can be stored in a hash table and can be distributed easily
- ? Such stores typically support regular CRUD (create, read, update, and delete) operations
 - That is there is a simplification with **no joins and aggregate functions**
- ? E.g., Amazon DynamoDB and Apache Cassandra



Columnar Databases

Columnar databases are a hybrid of RDBMSs and Key-Value stores

Values are stored in groups of zero or more columns, but **in Column-Order (as opposed to Row-Order)**

Record 1			
Alice	3	25	Bob
4	19	Carol	0
45			

Row-Order

Column A			
Alice	Bob	Carol	
3	4	0	25
19	45		

Columnar (or Column-Order)

Fast if you are not accessing individual records but need columns

Used in data analysis tasks

E.g., HBase and Vertica



Limitation of Classical Commercial Systems in the era of Networking

Most large corporations use distributed databases

It is **very difficult to have fast generic distributed databases**

Key limitation of distributed databases can be described in the so called the **CAP theorem...**

? **Consistency**: every node always sees the same data at any given instance (i.e., strict consistency)

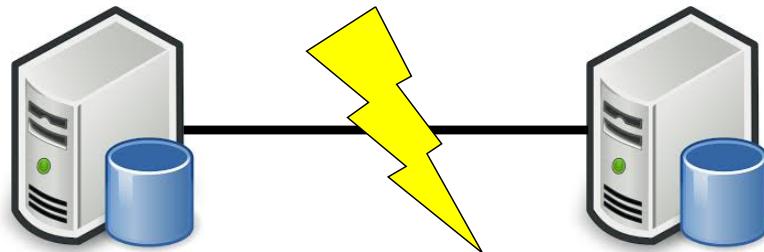
? **Availability**: the system continues to operate, even if nodes crash, or some hardware or software parts are down due to upgrades

? **Partition Tolerance**: the system continues to operate in the presence of network partitions

CAP theorem: any distributed database with shared data, can have at most two of the three desirable properties, C, A or P

The CAP Theorem Contd.

Let us assume two nodes on opposite sides of a network partition:



Availability + Partition Tolerance forfeit Consistency as changes in place cannot be propagated when the system is partitioned

Consistency + Partition Tolerance entails that one side of the partition must act as if it is unavailable, thus forfeiting Availability

Consistency + Availability is only possible if there is no network partition, thereby forfeiting Partition Tolerance



Availability Dominates

- ? When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
 - ? A few minutes of downtime means a lot of lost revenue
- ? With databases in 1000s of machines, the likelihood of a node or a network failure increases tremendously
- ? Therefore, in order to have strong guarantees on Availability and Partition Tolerance, they had to **sacrifice “strict” Consistency (*implied by the CAP theorem*)**



Other Types of Consistency

Strong Consistency

- After the update completes, **any subsequent access** will return the **same** updated value.

Weak Consistency

- It is **not guaranteed** that subsequent accesses will return the updated value.

Eventual Consistency

- Specific form of weak consistency
- It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the last updated value (e.g., *propagate updates to replicas in a lazy fashion*)



Eventual Consistency Variations

Causal consistency

- Processes that have causal relationship will see consistent data

Read-your-write consistency

- A process always accesses the recent data after it's update operation and never sees an older value

Session consistency

- As long as session exists, system guarantees read-your-write consistency



Eventual Consistency Variations

Monotonic read consistency

- If a process has seen a particular value of data item, any subsequent processes will never return any previous values

Monotonic write consistency

- The system guarantees to serialize the writes by the *same* process

In practice:

- A number of these properties can be combined
- Monotonic reads and read-your-writes are most desirable



Eventual Consistency - A Dropbox Example

Dropbox enabled immediate consistency via synchronization in many cases.

However, what happens in case of a network partition?





Eventual Consistency - A Dropbox Example

Let's do a simple experiment here:

- Open a file in your drop box
- Disable your network connection (e.g., WiFi, 4G)
- Try to edit the file in the drop box: can you do that?
- Re-enable your network connection: what happens to your dropbox folder?



Eventual Consistency - A Dropbox Example

Dropbox embraces eventual consistency:

- Immediate consistency is impossible in case of a network partition
- Users will feel bad if their word documents freeze each time they hit Ctrl+S , simply due to the large latency to update all devices across WAN
- Dropbox is oriented to personal syncing, not on collaboration though



Eventual Consistency

- An ATM Example

In design of automated teller machine (ATM):

- Strong consistency appear to be a natural choice
- However, in practice, **A beats C**
- Higher availability means **higher revenue**
- ATM will allow you to withdraw money *even if the machine is partitioned from the network*
- However, it puts **a limit** on the amount of withdraw (e.g., \$200)
- The bank might also charge you a fee when a overdraft happens





Dynamic Tradeoff between C and A

An airline reservation system:

- When most of seats are available: it is ok to rely on somewhat out-of-date data, availability is more critical
- When the plane is close to be filled: it needs more accurate data to ensure the plane is not overbooked, consistency is more critical



Heterogeneity: Segmenting C and A

No single uniform requirement

- Some aspects require strong consistency
- Others require high availability

Segment the system into different components

- Each provides different types of guarantees

Overall guarantees neither consistency nor availability

- Each part of the service gets exactly what it needs

Can be partitioned along different dimensions...



Partitioning Examples

Data Partitioning: Different data may require different consistency and availability

Example:

- Shopping cart: high availability, responsive, can sometimes suffer anomalies
- Product information need to be available, slight variation in inventory is sufferable
- Checkout, billing, shipping records must be consistent

Trading-Off Consistency

- ? Maintaining consistency should balance between the strictness of consistency versus availability/scalability
- ? Good-enough consistency *depends on your application*

Loose Consistency



Strict Consistency

Easier to implement,
and is efficient

Generally hard to implement,
and is inefficient



The BASE Properties

?

The CAP theorem shows that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions

?

This resulted in databases with relaxed **ACID guarantees**

In particular, such databases, e.g., NoSQL, apply the **BASE properties**:

?

Basically **A**vailable: the system guarantees Availability

?

Soft-State: the state of the system may change over time

?

Eventual Consistency: the system will eventually become consistent