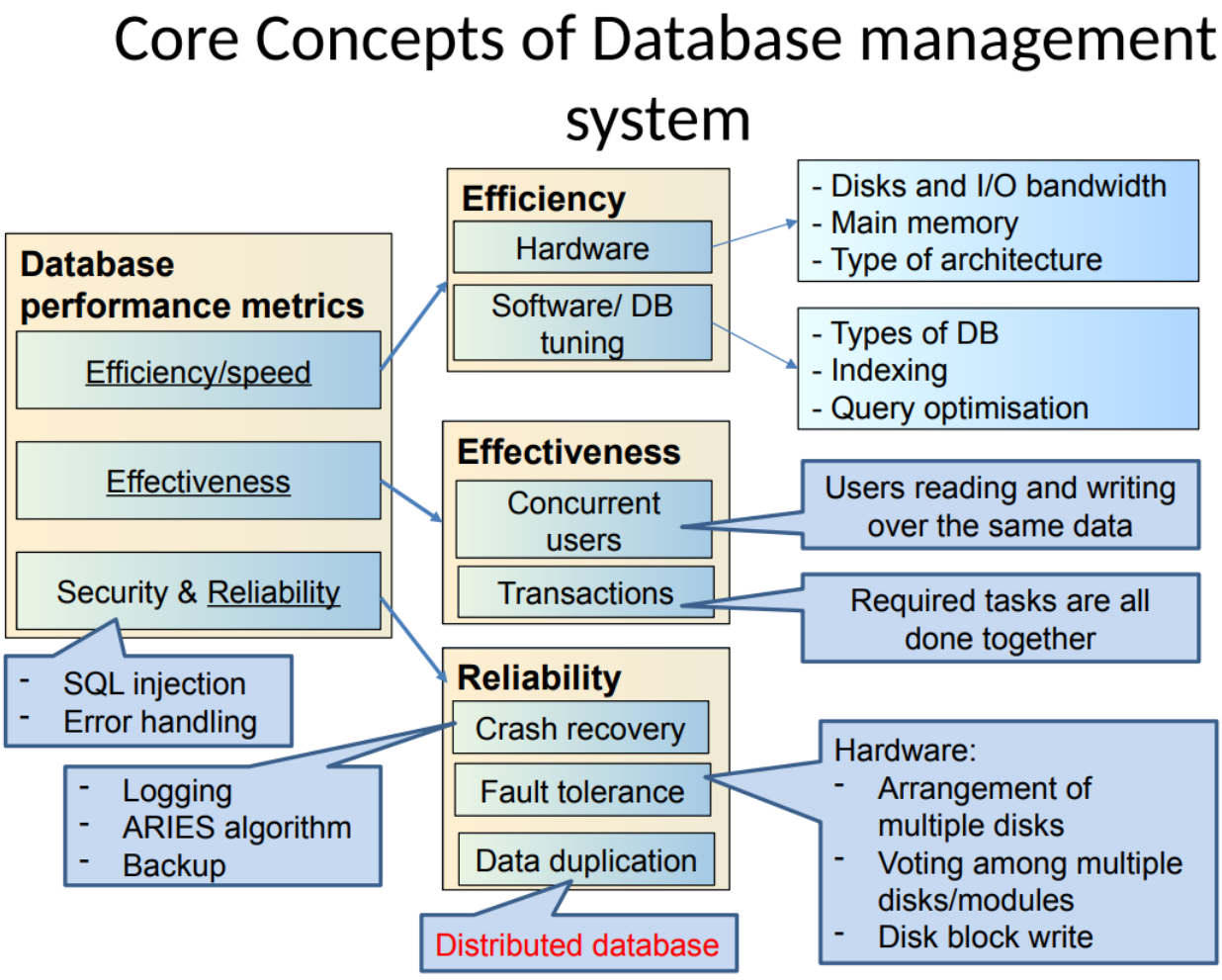
- WK1
  - Performance factors of DB
  - Disk drives
    - HDD
    - SSD
    - Disk Access time
    - Laws
  - Memory hierarchy
    - Effective Memory access time:
  - Communication costs
  - Types of database system
    - Simple file
    - Relational DB
    - Object Oriented (OO) DB Systems
    - NoSQL (also called Not Only SQL)
    - Key-value pair database systems
    - Deductive database systems (DDBS)
    - Document Storage
    - Graph Storage
  - Different database architectures
    - Centralised
    - Distributed
    - WWW
    - Grid
    - P2P
    - Cloud database
  - Fault tolerance
    - Storage Area Networks(SAN)
    - RAID
    - Failvote, Failfast,SuperModel
    - Availability of failvote systems
    - Fault tolerance with repair
    - Fault tolerance of a supermodule with repair
  - Communication reliability
  - Disk writes for consistency
    - Duplex write
    - logged write

        - Cyclic Redundancy Check (CRC)
    - Question
        - How to solve the performace issue?
- WK2
    - Database engine
    - Relational algebra
    - Join algorithms
    - How Data are stored:
    - Query costs
    - Query plans and optimisation
        - cost-based query optimisation
        - query plan cost的估计:
        - How to estimate costs?(reduction factor)
        - Simple Nested-Loop Join
        - Page-Oriented Nested-Loop Join
        - Why good query optimiser?
        - Heuristic VS Enumerating
        - Heuristic optimization
        - Adaptive plans
    - Query costs – in practice
        - Troubleshooting to manage costs
        - Query with suboptimal performance
        - Query reuse
        - Further lower query costs?
    - Indexing
        - What is indexing
        - Different types of indexes
        - B+tree
            - 知道n求height
        - hash index
        - bitmap index
        - Quadtree
        - R-tree
            - search in R tree
            - Nearest Neighbor Query on R-tree
        - Usage of indexes
        - Indexing in practice
    - SQL injection

- WK3
  - Transactions
    - ACID
    - Types of Actions
    - Types of transactions
      - Flat transaction：
      - Nested transaction
    - Transaction processing monitor(TP monitor)
      - TP monitor services
  - Concurrency control
    - Dekker's algorithm (using code)
    - OS supported primitives
    - Spin locks
      - Atomic operations
    - Semaphore
      - Implementation of Exclusive mode Semaphore(说白了就是链表排队)
      - Convoy avoiding semaphore
  - Deadlocks
    - Solutions
  - Isolation concepts
  - Dependency relations
    - write-write (Lost update):
    - read-write (Unrepeatable read):
    - write-read (Dirty Read):
    - Equivalence
  - Isolated history
  - lock
  - Wormhole
  - Isolation concept & theorems
    - Well-formed transactions
    - Two phase transactions
  - Degrees of isolation
  - Granular locks
  - Optimistic locking
  - Snapshot isolation
  - Timestamping
- WK4

- Crash recovery
  - Buffer pool and disk transfers
  - Logging and WAL
    - Logging
  - Write-Ahead Logging(WAL)Protocol
  - WAL log
  - Normal Execution of an Xact
  - Checkpointing
  - Simple transaction abort
  - Transaction commit
  - ARIES algorithm
    - Analyse
    - Redo
    - Undo
- Backups
  - Remote backups
    - Detection of failure:
    - Transfer of control:
    - Time to recover:
    - Hot-Spare configuration
    - One-safe:
    - Two-very-safe:
    - Two-safe:
  - Shadow paging
    - To commit a transaction:
    - Advantages
    - Disadvantages:
  - Backup design strategy
- Distributed databases
  - Atomicity in distributed DB
    - Two phase commit protocol
    - Abort transaction
  - Concurrency Control
    - Locking-based systems
    - Timestamp ordering
    - Optimistic
  - Transactions with replicated data
  - CAP theorem

2023 Winter COMP90050 ADBS Final Notes



WK1

Performance factors of DB

A **Database Management System (DBMS)** is a software system designed to store, manage, and facilitate access to databases.

A database system should provide

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

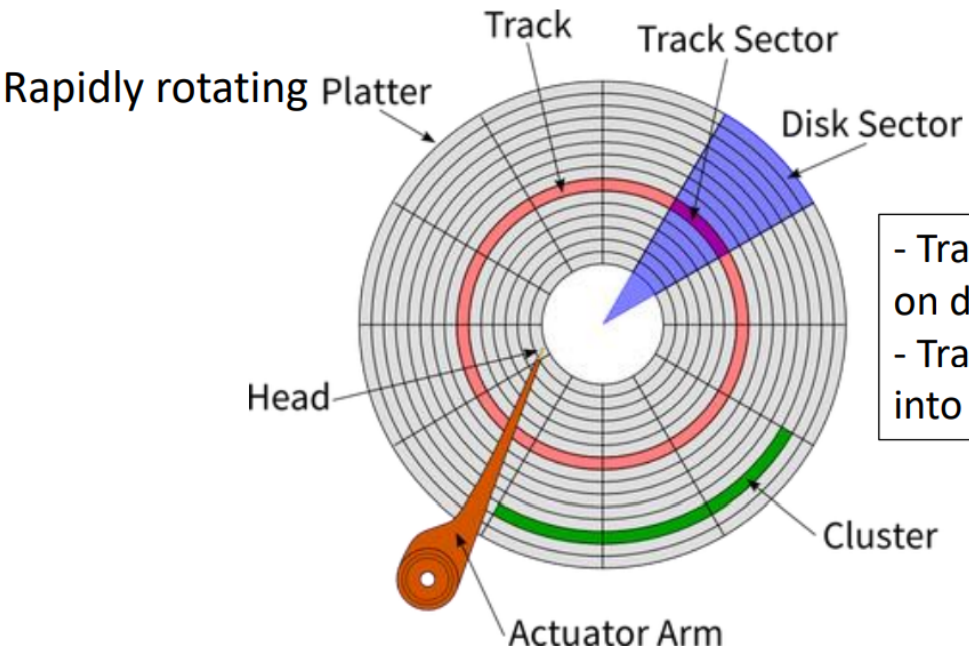
Database performance metrics

Disk drives

HDD

(Hardware disk硬盘)

Basic Hardware of a classical disk



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

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

SSD

(Solid-State Drive/Solid-State Disk固盘)

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
- Certain read/write limitations plagued it for years

Advantages of SSD: • Faster, Quiet, Not sensitive to movement or light hit.

Disadvantage of SSD: • It is slow when overwriting data. • More expensive

How to solve the performance?

Disk Access time

Disk access time(HDD) = seek time + rotational time + transfer length/bandwitdth

Disk access time(SSD) = transfer length/bandwitdth

Laws

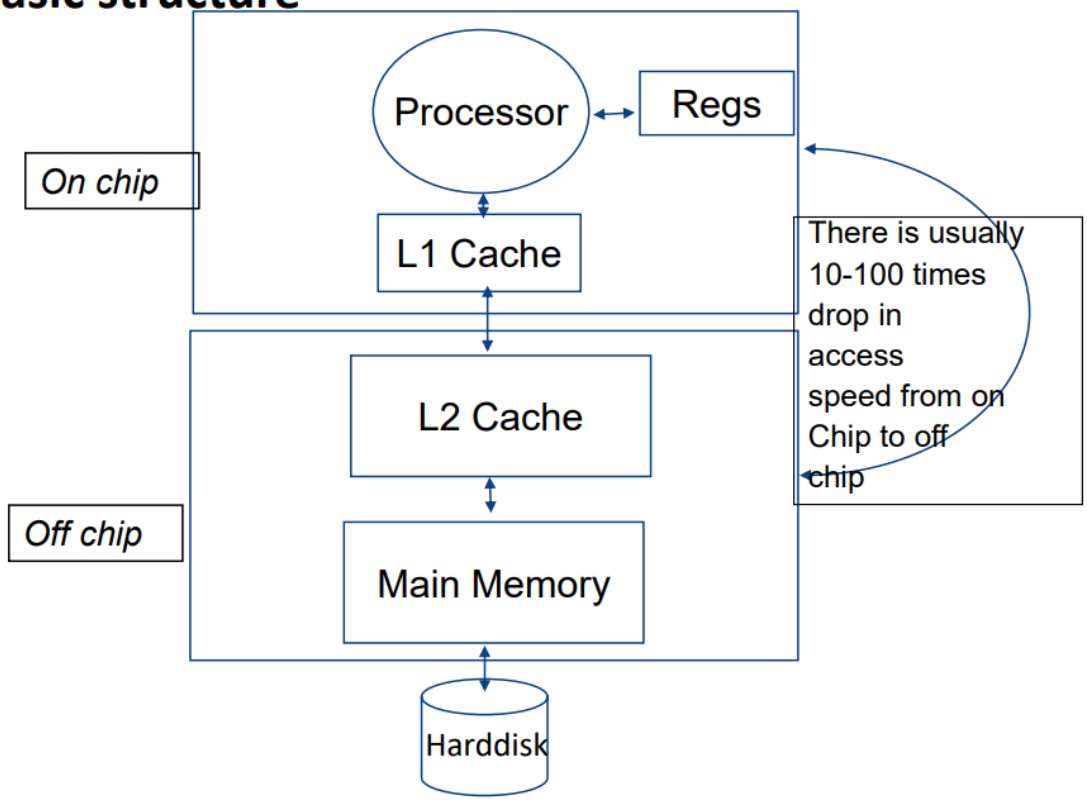
Moore's law(memory chip): Chip capacity = $2^{\frac{2*(year-1970)}{3}}KB/chip$232∗(year−1970)KB/chip

Joy's law(processor): Processor performance = $2^{(year-1984)/2}mips$2(year−1984)/2mips

Memory hierarchy

So where do we store data: The Memory Hierarchy

Basic structure



17

Effective Memory access time:

For SSD:

Memory Hierarchy

$$Hit\ ratio = \frac{references\ satisfied\ by\ cache}{total\ references}$$

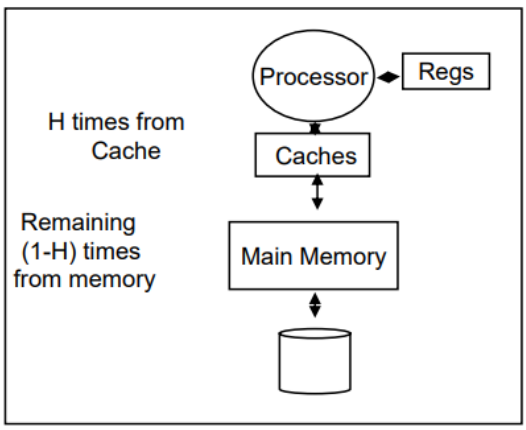
Effective memory access time,

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

where H = hit ratio,

C = cache access time;

M = memory access time



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

For HDD:

If data needs to be transferred from HDD

$$Disk\ access\ time = seek\ time + rotational\ time + \frac{transferlength}{bandwidth}$$

Caching provided with HDD for access

Effective disk buffer access time,
EA = HB*BC+(1-HB)*D where
HB = hit ratio of the disk buffer , BC = buffer access time; D = disk access time

Hit ratio	Effective access time as multiple of BC, D = 1000 C
50.00%	500.5
99.00%	100.9
99.90%	1.999
99.99%	1.099

21

larger cache with higher cache hit ratio.

Communication costs

harddisk是数据库设计的基石，除此之外，目前的数据库大多为分布式数据库，因此还要考虑传输速度，线缆(光纤)的传输速度目前已达光速，所以当前的硬件latency无法降低。

Communication Costs

Increasingly, another item to model the cost of is data transfer:

transmit time = (distance/c) + (message_bits/bandwidth)

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

*This means we can **no longer reduce latency on contemporary hardware further** and increasingly the motto is that the **message length should be large to achieve better utilization**.*

Can you relate the same idea for reading from HDD?

Types of database system

Simple file

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

RDBS

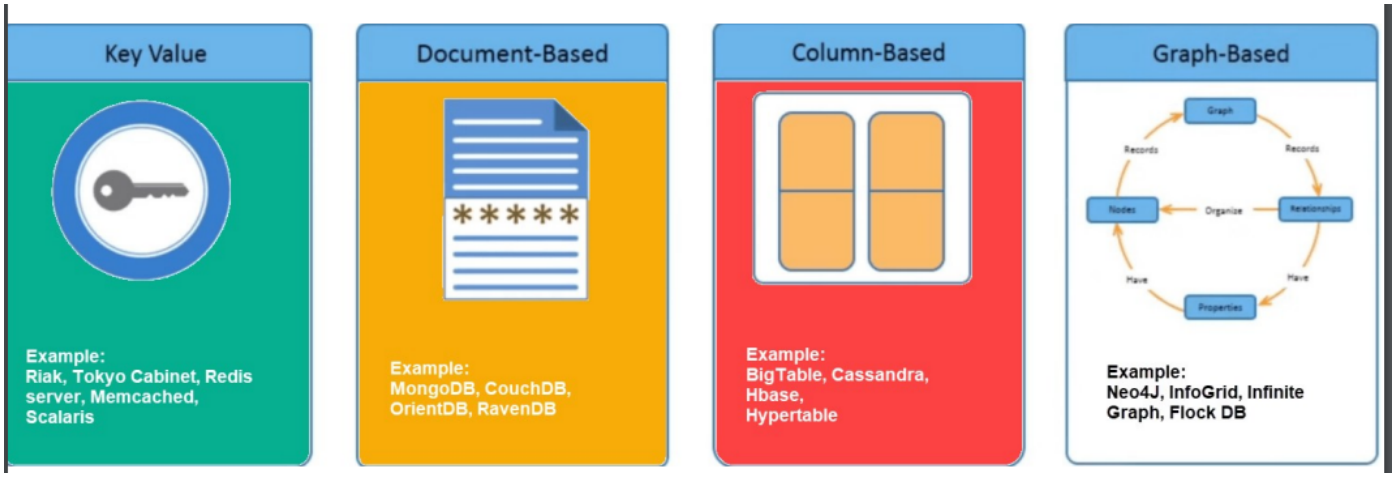
- As a collection of tables (relations) consisting of rows and column. A primary key is used to uniquely identify each row.

Object oriented

- Data stored in the form of 'objects' directly (like OOP)

No-SQL

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



Simple file

Simple file

- Usually very fast for simple 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

Relational DB

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* (consistency of data – we will learn more later)
- Application independent optimisation
- Well suited to many applications, very fast due to large main memory machines and SSDs.
- Some RDB also support Object Oriented model e.g., Oracle, DB2, and XML data+queries
- Can be slow for some simple applications

Object Oriented (OO) DB Systems

Object Oriented (OO) DB Systems

- Stores as objects directly, not tables
- May contain both data (attributes) and methods – like OOP
- Can be slow on some applications
- Reliable
- Limited application independent optimisation
- Well suited for applications requiring complex data
- Unfortunately, many commercial systems started did not survive the force of RDB technology and basically disappeared from the market.

NoSQL (also called Not Only SQL)

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
- Simple design, should linearly scale
- NoSQL has ***compromise consistency*** and allows replications - We will discuss this more later.
- 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 later

Key-value pair database systems

Key-value pair database systems

Stores data as a collection of key–value pairs, where each key is unique

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

- Can be seen as a type of NoSQL database
- Used for building very fast, highly parallel processing of large data - MapReduce and Hadoop are examples
- Atomic updates at Key-value pair level (row update) only.

Deductive database systems (DDBS)

Some other DB systems – Deductive database systems (DDBS)

- Allows recursion
- Most of the application can be developed entirely using DDBS
- There are no 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).
- E.g.

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Document Storage

Document Storage

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

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

Graph Storage

Graph Storage

- Graphs capture connectivity between entities. Searching and traversing by relations are very fast in such structures.
  - The links can be material or immaterial:
  - Links between two streets are junctions;
  - Links between people as their facebook connections (non material links)
  - A graph is a structure amounting to a set of objects (called vertices) where some pairs of the objects are connected/related in some sense. A connection is called an edge.

Applications of different forms of Databases

- Applications for key-value databases – Suitable if the dataset do not need complex relational table type of structure, but can be expressed with simple key-value pairs. The simple structure allows faster insertion and search, and scales quickly. For example – shopping cart in an e-commerce site.
- Applications for document storages – Well suited when different kinds of documents do not always have the same structure/sections. For example – news articles.
- Applications for graph databases – well suited for connection data: social network connections (e.g., who are my friends of friends) , spatial data (e.g., route planning – which ways can I go now to reach destination).

Different database architectures

Centralized	Distributed	WWW	Grid	P2P	Cloud
• Data stored in one location	• Data distributed across several nodes, can be in different locations	• Stored all over the world, several owners of the data	• Like distribu-ted, but each node manages own resource; system doesn't act as a single unit.	• Like grid, but nodes can join and leave network at will (unlike Grid)	• Generalization of grid, but resources are accessed on-demand

Centralised

– suitable for simple applications, easy to manage; may not scale well

Distributed

– Scalable, suitable for large applications and applications that need data access from different physical locations; System administration and crash recovery is difficult, usually have some data inconsistency

WWW

– Very convenient to access and share data; security issues, no guarantee on availability or consistency

Grid

– Less used now-a-days, very similar to distributed systems with administration done locally by each owner

P2P

-Suitable when the nodes of the network cannot be planned in advance, or some may leave and join frequently. For example, sensor network

Cloud database

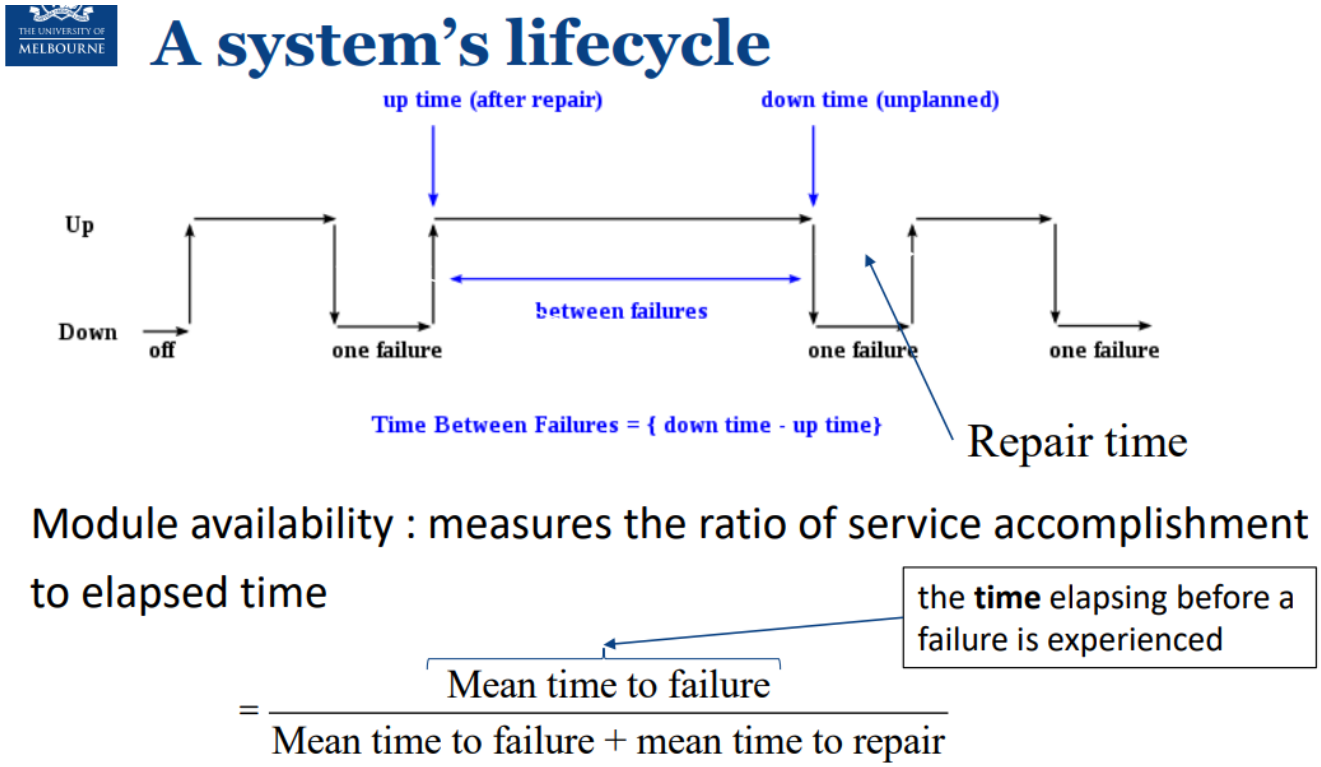
– on-demand resources, cost-effective, maintenance done externally by the cloud provider; some privacy and confidentially issue – but most trusted providers well-address them

Fault tolerance

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

Main time to failure;

Module availability



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

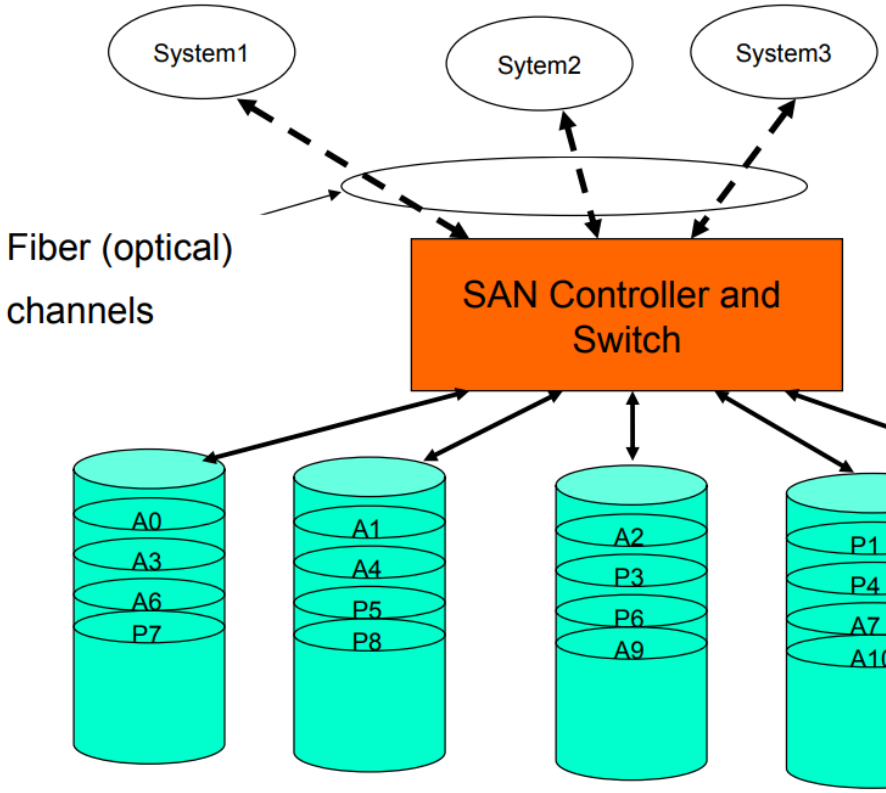
the **time** elapsing before a failure is experienced

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

Storage Area Networks(SAN)

Storage Area Networks (SAN)

A dedicated network of storage devices



- Storage can be organized as RAID (Redundant Array of Inexpensive Disks) systems .
- Storage is partitioned and allocated to each system and can also be shared.

More on SANs

- They are used for shared-disk file systems
- **Automated backup** 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
- But in short, **failure probability of one disk is different than 100s of disks -** which requires design choices

- RAID 0 (block level striping)
    - striping at block level
    - throughput double and MTTF(mean time to failure) half
    - MTTF/2

- RAID 1 (mirroring)
    - higher read throughput
    - lower write throughput
    - half storage utilization
    - $MTTF^2$MTTF2

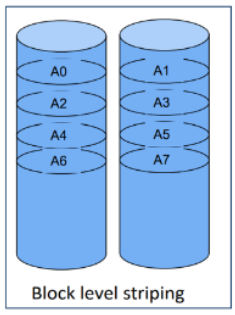
- RAID 2 (bit level striping)
    - similar to raid 0, striping 1 bit level
    - MTTF/2

- RAID 3 (byte level striping)
    - byte level striping
    - transfer rate > RAID 0
    - $MTTF^2/3$MTTF2/3
    - parity

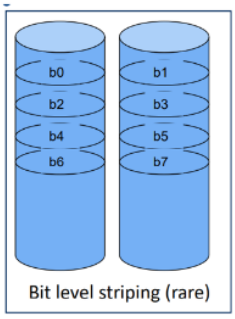
- RAID 4 (block level striping)
  - similar to RAID 3
  - Dedicated disk for parity blocks
  - higher throughput but very slow writes
  - $MTTF^2/3$MTTF2/3

- RAID 5 (block level striping)
  - similar to RAID 4, 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.
  - $MTTF^2/3$MTTF2/3

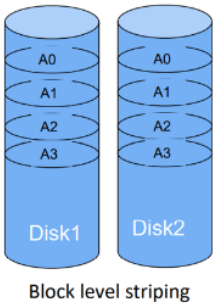
- RAID 6 (block level striping)
  - similar to RAID 5 two parity blocks used
  - $MTTF^3/10$MTTF3/10(two disk breakdown tolerance)



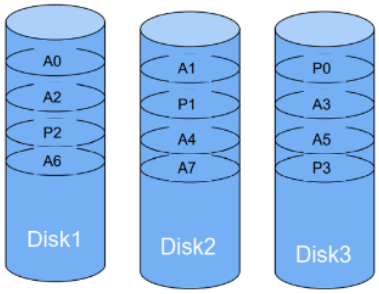
RAID 0



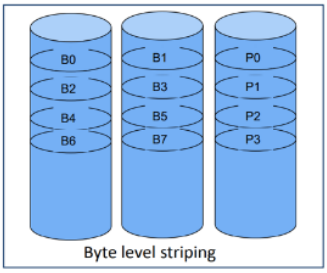
RAID 2



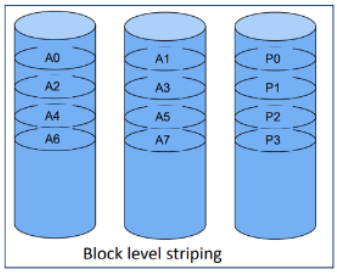
RAID 1



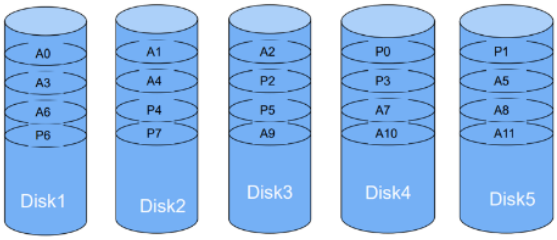
RAID 5



RAID 3



RAID 4

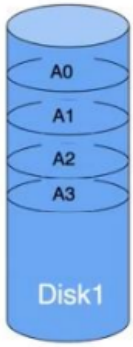


RAID 6

Availability / how to calculate the MTTF

(Review) Baseline: One Disk

- MTTF: $1/p$
  - p: probability of failure of disk
  - $MTTF(disk) = \frac{1}{p}$



Failvote, Failfast,SuperModel

failvote: stop if there are no majority agreement不满足最小vote数即fail

failfast: uses the majority of the available modules.灵活的最小vote数，最后一个vote不满足即fail

Supermodule: Naturally, 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 even when only one is available)全死完才fail

十个盘里坏了5个failvote就罢工，坏9个failfast罢工,supermodel继续工作

minimum vote number:

- odd: (n+1)/2
- even: n/2 +1

Availability of failvote systems

低availability 确保了高reliability,(disk更少，可用性越低，但可靠性提高)

Fault tolerance with repair

可以兼顾二者。

发现fault时立刻修复，不会等到whole system fail

e.g. 对于failvote,假设存在10个modules,5个无法工作时system fail,确保当有module fault时立刻修复，确保整个系统的可用性。

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 (Sometimes MTTR is just time needed to replace)

Typical Values for recent disks:

MTTR = Few hours (assuming we stock spare disks) to 1 Day

MTTF = 750000 hours (~ 86 years) [hard fault]

Probability of a particular module is not available

= MTTR/(MTTF+MTTR)

≅ MTTR/MTTF if MTTF >> MTTR

Fault tolerance of a supermodule with repair

Probability that (n-1) modules are unavailable, $P_{n-1} = \left(\frac{MTTR}{MTTF}\right)^{n-1}$

Probability that a particular ith module fails, $P_f = \left(\frac{1}{MTTF}\right)$

Probability that the system fails with a particular ith module failing last =

$$P_f * P_{n-1} = \left(\frac{1}{MTTF}\right)\left(\frac{MTTR}{MTTF}\right)^{n-1}$$

Probability that a supermodule fails due to any one of the n modules failing last, when other (n-1) modules are unavailable $= \left(\frac{n}{MTTF}\right)\left(\frac{MTTR}{MTTF}\right)^{n-1}$

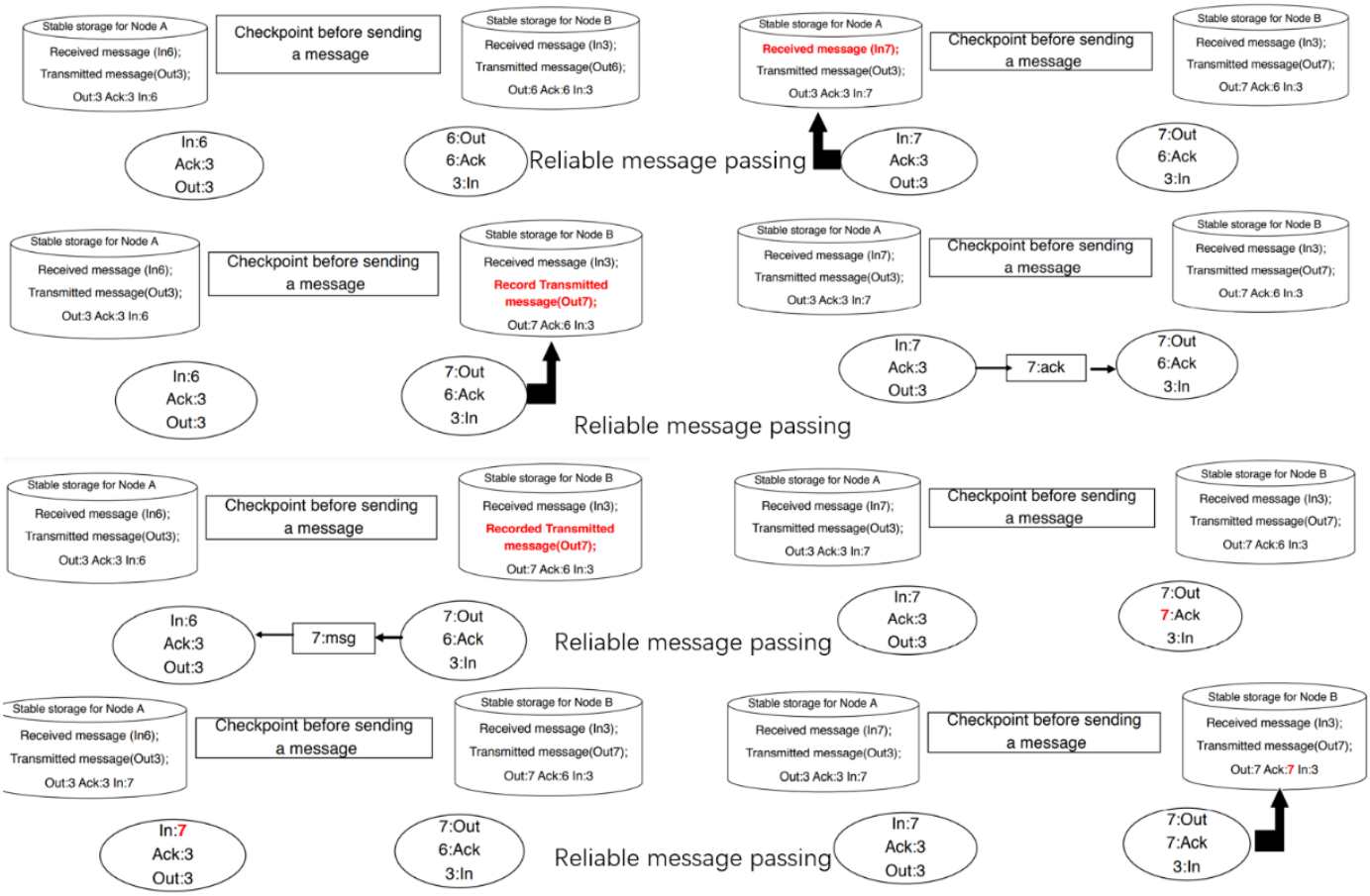
What will this value for failvote and for failfast?

n代表有n个modules, n-1 可以为其他两种系统fail需要的module数。

Communication reliability

note: in step 6, if 7:ack do not send success, node B will wait for a certain time and send msg again and wait for the ack.(当B没有收到ack时会等待2-5min并且继续向A发送msg)

Ack:我方收到的从对方来的Ack



Disk writes for consistency

Either entire block is written correctly on disk or the contents of the block is unchanged. To achieve disk write consistency we can do –

Duplex write

使用CRC

• Each block of data is written in two places sequentially

• If one of the writes fail, system can issue another write

• Each block is associated with a **version number**. The block with the latest version number contains the most recent data.

• While reading - we can determine error of a disk block by its CRC.

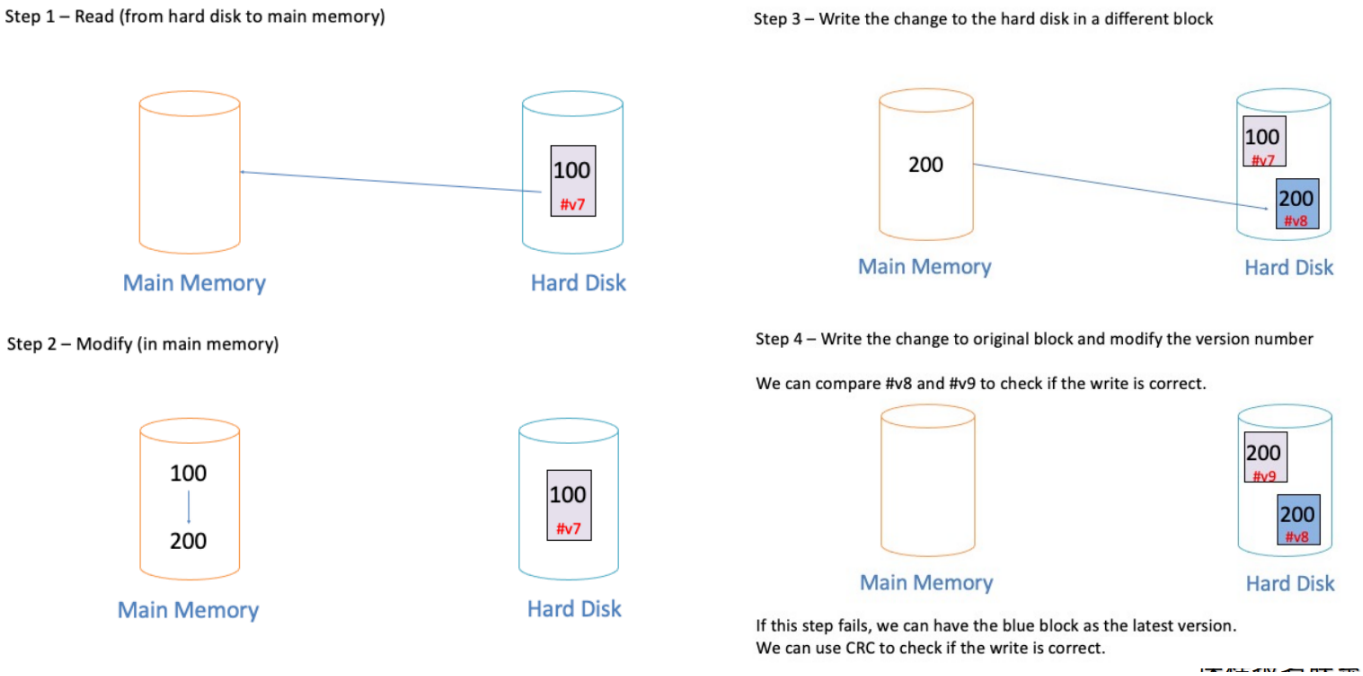
• It always guarantees at least one block has consistent data.(永远确保至少一个Block有完好的数据)

logged write

similar to duplex write, except one of the writes goes to a log.

This method is very **efficient** if the **changes to a block are small**.

(从RAM写进disk:main memory to log, which based on the disk)



Cyclic Redundancy Check (CRC)

An error detection algorithm

1. A polynomial needs to be specified
2. A sequence of bitwise exclusive-or (XOR) operation needs to be performed
3. The final CRC value needs to be stored for each data block (or the data unit on which CRC is performed)
4. Data correctness can be checked with CRC -

   a. its corresponding CRC value is retrieved

   b.A sequence of bitwise XOR operation needs to be performe to find out the correctness of data

CRC可以发现最高位poly式子系数位数的brust error(突发错误，连续性，无间断)

如 x^32 + x^23 + x^7 +1, 可以察觉所有<=32bit的brust error.

如何选择poly位数？取决于系统中频发的error位数。

0. 有一个polynominal e.g. x^3 + x + 1
1. Add n zero bits as ‘padding’ to the right of the input bits.
2. Compute the (n + 1)-bit pattern representing the CRC’s divisor (called a “polynomial”)
3. Position the (n + 1)-bit pattern representing the CRC’s divisor underneath the left-hand end of the input bits.
4. The algorithm acts on the bits directly above the divisor in each step.

CRC calculation step:

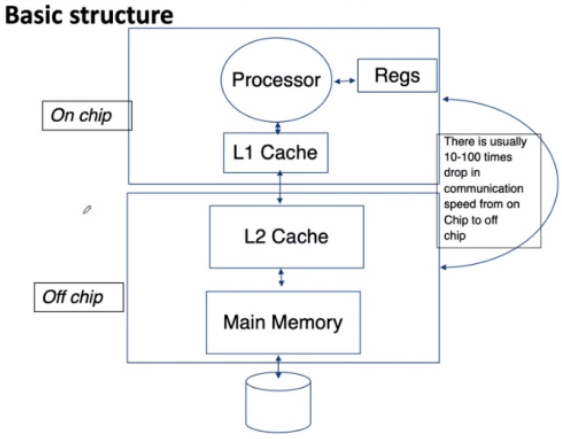
详见笔记本

Question

How to solve the performace issue?

How to solve the performance issue?

- Any ideas?
  - Improve the access time for the memory.
    - Requires new technological design.
    - SSD is pretty much at the limits of a fast memory.
    - Fast memories will be very expensive.
  - Change the structure
    - Not all the data in memory is always required.
    - Design a new fast and small memory: Cache.
    - Read from Cache instead of main memory.
  - Access time for hierarchical structure with one cache.

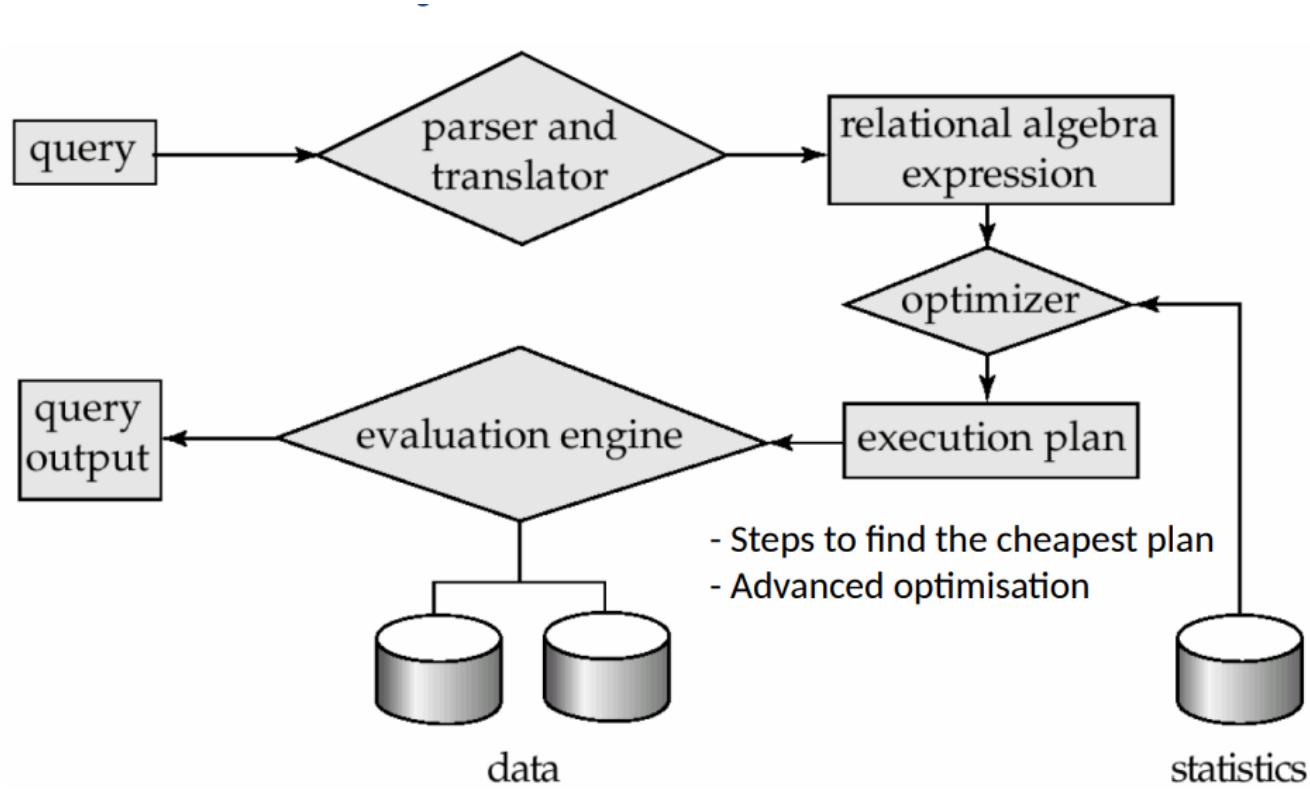


$$access\ time = access\ cache * hit\ ratio + access\ mem\ * (1 - hit\ ratio)$$

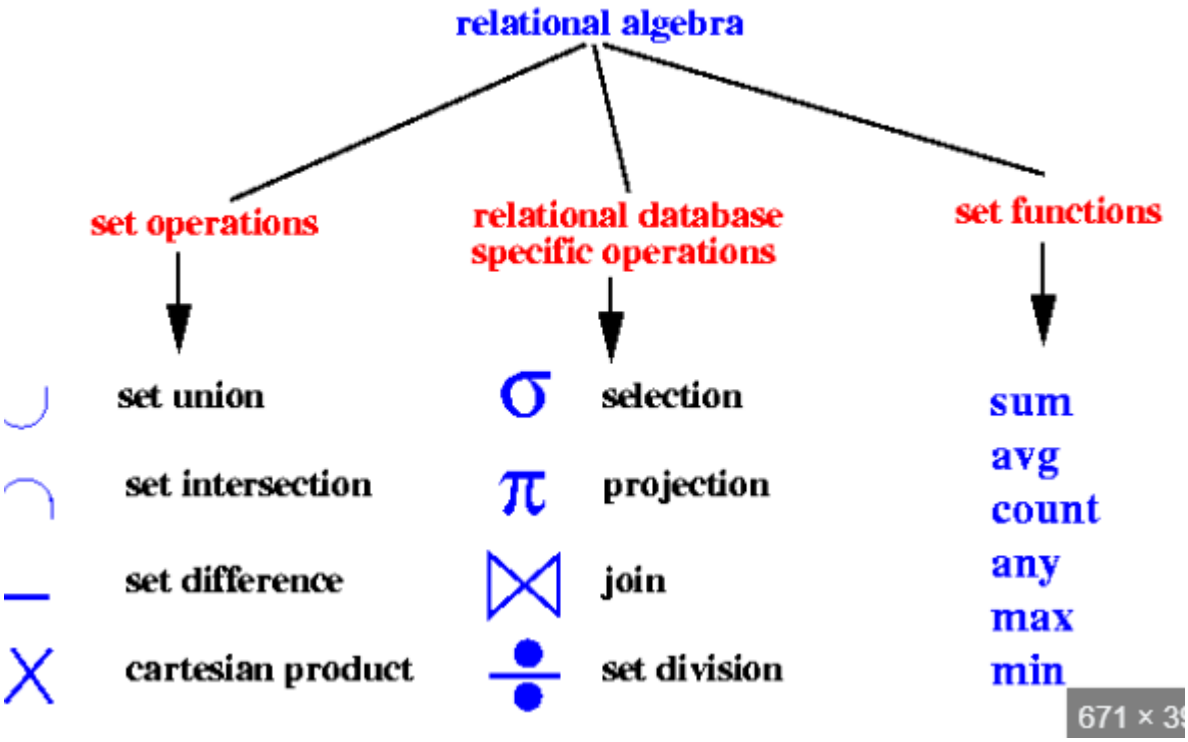
WK2

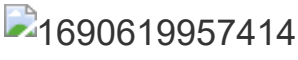
Database engine

Query processing steps



Relational algebra





Join algorithms

More on Joins

$r \bowtie_\theta s$ $r \bowtie s$

Natural join

- Here, *r* and *s* are tables
- Theta (θ) is the condition (for example, <,>,=)
- Natural join – joining based on common columns
- Joins are very common and also very expensive

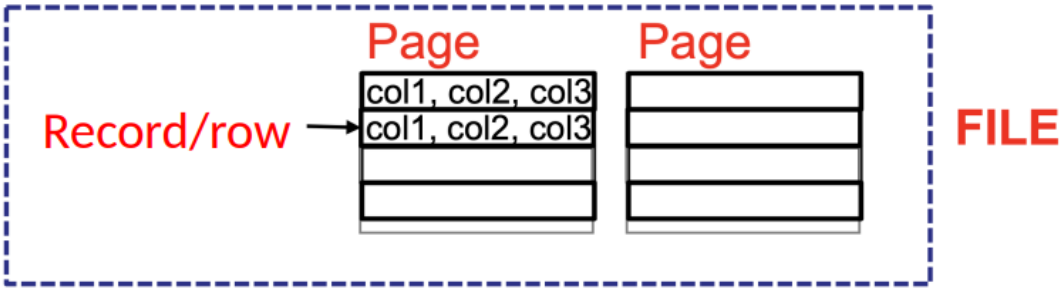
join常见但是昂贵(花费时间多)，左为条件Join，右为natural join,会自己寻找type,value一致的column作为条件进行join.

r 为 **outer relation,** s 为**inner relation**.

How Data are stored:

How data are stored

- **Files** – A database is mapped into different files. A file is a sequence of records.
- **Data blocks** – Each file is mapped into fixed length storage units, called data blocks (also called logical blocks, or pages)



若干个record or row组成了Page,若干个page组成file.

Query costs

- **Cost of a query** – The number of pages/ disk blocks that are accessed from disk to answer the query

Most dominant cost
(Recall from memory hierarchy!

query cost: 为了回答query而访问disk去查找page的页数(访问cache/register的时间很小，主要由disk的访问时间构成)

Query plans and optimisation

cost-based query optimisation

1. 生成逻辑等效的query的不同表达式
2. 获得备选(使用heap scan还是 index scan;使用什么种类的join?)
3. 选择cost最小的方案。

Query plans and optimisation

Steps in **cost-based query optimisation**

1. Generate logically equivalent expressions of the query
2. Annotate resultant expressions to get alternative query plans
   - Heap scan/Index scan?
   - What type of join algorithm?

3. Choose the cheapest plan based on estimated cost

Estimation of plan cost based on:

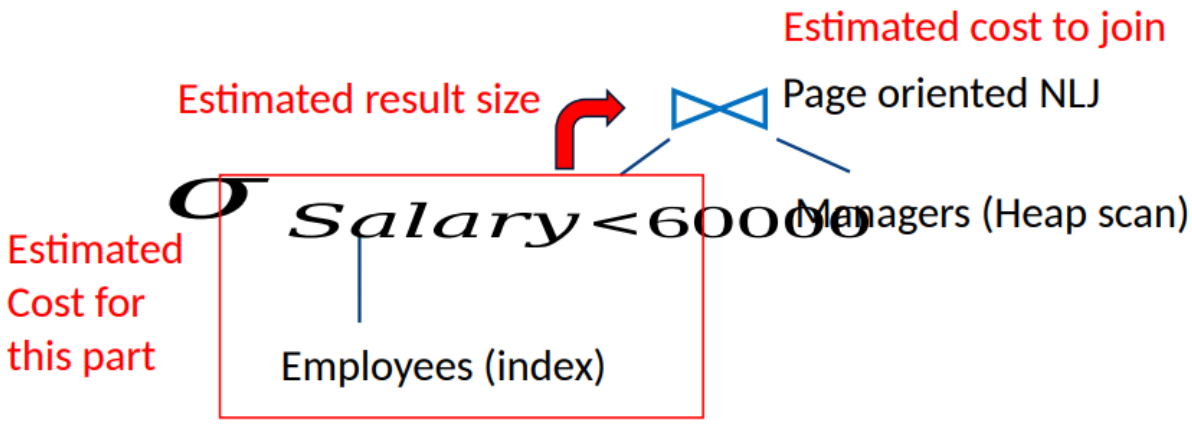
- Statistical information about tables.

  Example: number of distinct values for an attribute
- Statistics estimation for intermediate results to compute cost
- Cost formulae for algorithms, computed using statistics again

query plan cost的估计:

1. 估计选中表的page
2. 估计apply select条件后的结果的cost
3. 估计join所需的cost

Estimation of query plan cost



How to estimate costs?(reduction factor)

How to estimate costs

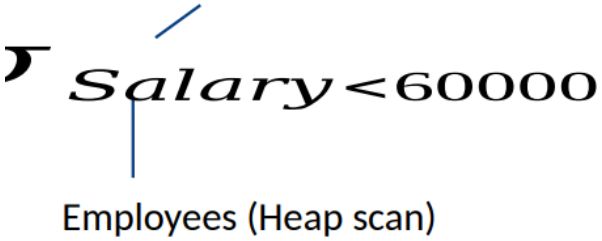
Step 1: Result size calculation using **Reduction Factor**

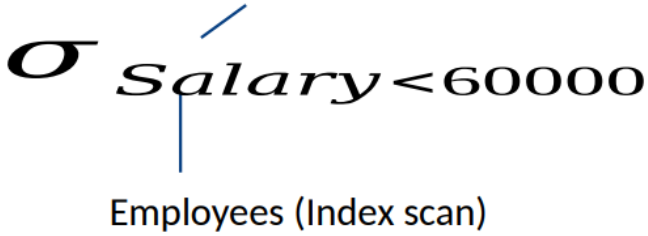
What can go wrong?

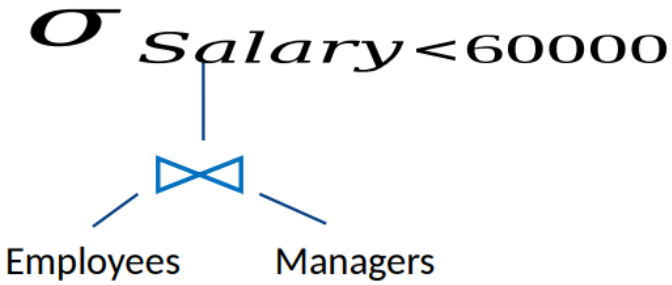
Depends on the type of the predicate:
1.Col = value: **RF = 1/Number of unique values (Col)**
2.Col > value: **RF = (High(Col) – value) / (High(Col) – Low(Col))**
3.Col < value: **RF = (val – Low(Col)) / (High(Col) – Low(Col))**
4.Col_A = Col_B (for joins):
RF = 1/ (Max number of unique values in Col_A,Col_B)

How to estimate costs

Step 2: Different options for retrieving data and calculating cost (again, **estimation)**







What can go wrong?

Simple Nested-Loop Join

完全遍历两个表的的所有page以及r的record number，最坏的情况为一个表占用一个page

适用于两个表都很小

About Nested-Loop Join

- 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 (θ)
  if they do, add t_r t_s to the result.
  end
  end

- The number of block transfers is $n_r * b_s + b_r$.
- The number of seeks is $n_r + b_r$

Let's Calculate the Costs

Let's see an example with the following bank database:

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

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

$b_r + (n_r * b_s)$ Page access

So Two Options Are

With *depositor* as the outer relation:

100 + (5000 ∗ 400) = 2,000,100 page access,

With *customer* as the outer relation:

400 + (10000 ∗ 100) = 1,000,400 page access

If you had 1000,000 customers, then you would wait several hours for one simple join!

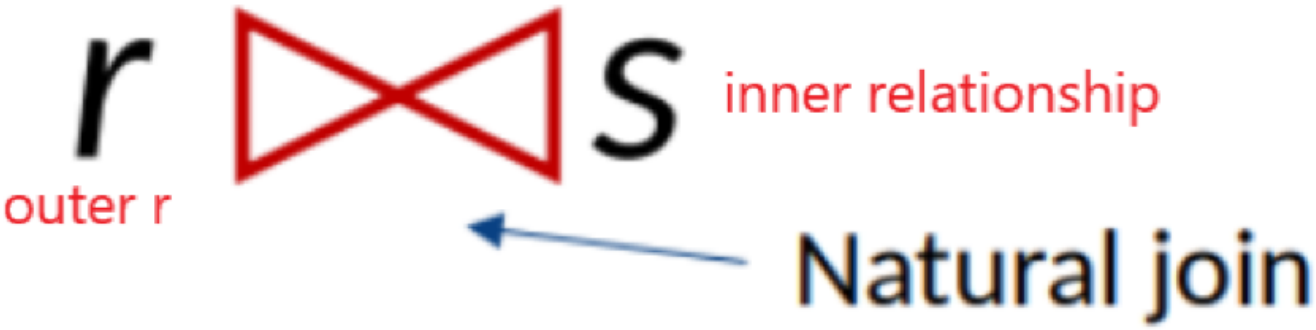
Page-Oriented Nested-Loop Join

只对r和s的page进行pair，所以复杂度较低，效率较高。

Improved Nested-Join; Block Nested-Join

```
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 • t_s to the result.
            end
        end
    end
end
```

- Number of block transfers: $b_r \times b_s + b_r$
- Number of seeks: $2 \times b_r$



Let's Calculate the Costs

Let's see an example with the following bank database:

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

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

$$b_r + (b_r * b_s)$$ Page access

So Two Options Are

With *depositor* as the outer relation:

$100 + (100 * 400) = 40100$ page access,

With *customer* as the outer relation:

$400 + (400 * 100) = 40400$ page access

Several order or magnitude faster than NLJ!

Why good query optimiser?

(好的query 优化的重要性)

1. It's the heart of query efficiency (query 高效性的核心)

2. Generating all equivalent expressions exhaustively - very expensive(彻底生成所有等效query表达式十分昂贵，消耗时间较多)
3. Must consider the interaction of evaluation techniques when choosing evaluation plans. Choosing the cheapest algorithm for each operation independently may not yield best overall cost(必须要考虑每一步的相互作用，每一个都是最优选但是结果可能并不是)
4. Estimations of the result size may not be accurate(估计的结果size也许并不精确)

Heuristic VS Enumerating

简单的queryA使用Heuristic，复杂的queryB使用Enumerating

Strategies for Query Optimizations

- Searching/Enumerating all the plans and choose the best one.
  - Used for quires that require accurate results.
  - Well suited for handling complex quires.
- Using a heuristic approach
  - Is like using a solution randomly.
  - Is a good option when accuracy is not a priority.
  - Can handle simple and straight-forward cases.

Scenario A: Given a table with 1000 tuples, run the following query:

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

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

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

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

Adaptive plan is executing a part/some parts of a query plan first to re-evaluate the cost of the other parts of the query plan that haven't been executed yet, to have a better overall cost estimation (and hence, choosing a better plan).

For Scenario A, the heuristic approach can be suitable due to the simplicity of the query and small size of the table. For Scenario B, the enumerating approach can be more suitable due to the complexity of the query.

Adaptive plan is used for better estimation of cost, hence, cannot be used when the query optimiser is purely heuristic based for a query (so cannot be used for scenario A if the plan is heuristic based). Adaptive plan can be used for cost-based query optimiser (either exhaustive enumeration of all plans or a combination), hence, can be used for Scenario B.

Heuristic optimization

所以在现实使用中往往使用启发式优化，

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 early (reduces the number of tuples)
2. Perform projections early (reduces the number of attributes)
3. Perform most restrictive selection and join operations (i.e.,with smallest result size) before other similar operations

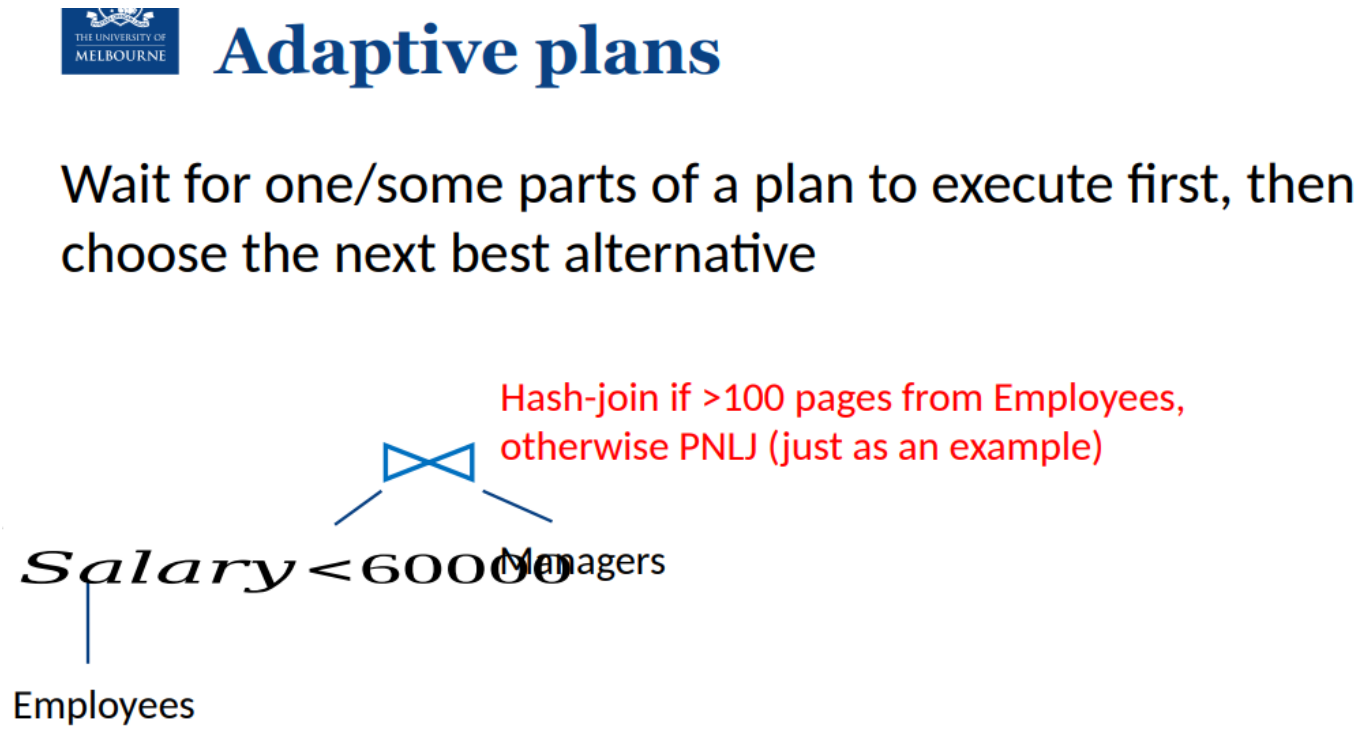
Some systems use only heuristics, others combine heuristics with cost-based optimization(一些系统使用启发式与枚举式的结合)

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

(便宜的query使用启发式，消耗更高更贵的query会枚举且选出最优方案)

Adaptive plans

逐步选出最优解，第一步最右后再进行第二步

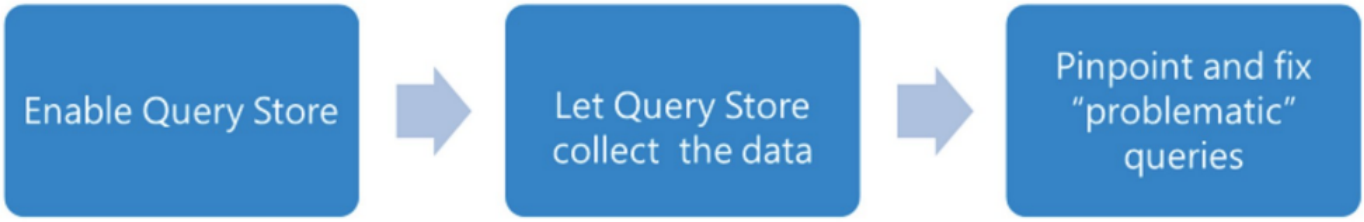


Query costs – in practice

Troubleshooting to manage costs

- Identify ‘regressed queries’ - Pinpoint the queries for which execution metrics have recently regressed (for example, changed to worse).识别发生退步的query, (performance go worse),表现变差
- Track specific queries - Track the execution of the most important queries in real time.(实时追踪重要的query)

启用query store -> 使query store收集数据 -> 识别并且修复出现问题的query



Query with suboptimal performance

当识别出一个query的性能下降:

- Force a query plan instead of the plan chosen by the optimizer (force a query plan:人为手动规定query Plan的使用，而不是由优化器生成)
- Do we need an index?(加入Index)
- Enforce statistic recompilation(强制重新编译统计信息)
- Rewrite query?(重写一个query)

A particular query on table A used to run quite efficiently in a DBMS. After inserting many records and deleting many other records from table A, that same query is now taking more time to run, even when the total number of records has not changed. What can be the reason for that? What can you do as the user/database administrator of that DBMS to improve the performance of this query?

Solution: After insertions and deletions, the statistics of a table may not be updated instantly. As the statistics are used to estimate the cost of a query, wrong statistics are causing wrong cost estimations, and hence the query optimiser is now choosing a query plan that is no longer optimal for that query.

Enforcing statistical recompilation option can be used to update the statistics of the table.

Query reuse

我们希望optimizer 生产的query plan可以被复用，当仅仅是参数改变。

Query costs – in practice

Query rewriting with parameters for execution plan reuse

```
SELECT *
FROM Product
WHERE categoryID = 1;
```

```
SELECT *
FROM Product
WHERE categoryID = 4;
```

We expect the optimizer to generate essentially the same plan and reuse the plans - parameterize

```
DECLARE @MyIntParm INT
SET @MyIntParm = 1
EXEC sp_executesql
  N'SELECT -
  FROM Product
  WHERE categoryID = @Parm',
  N'@Parm INT',
  @MyIntParm
```

Further lower query costs?

-Store derived data(存储已经被query导出的data)

- When you frequently need derived values(当此部分数据频繁被query)
- Data do not change frequently(当数据不常改变)

-Use pre-joined tables(使用提前join的table)

- When tables need to be joined frequently(当table需要被经常join)
- Regularly check and update pre-joined table for updates in the original table(需要经常检查即更新提前join的table)
- May still return some ‘outdated’ result(也许会读出过期的数据)

e.g.使用 此命令行 make the table stay in the main memory

Memory optimised table

Can you make a table stay in main memory?

```
CREATE TABLE dbo.Customer (
    CustomerID char (5) NOT NULL PRIMARY KEY,
    ContactName varchar (30) NOT NULL
) WITH (MEMORY_OPTIMIZED=ON)
```

Query processing and query optimisation for memory optimised tables?

Indexing

Background:

1. **DBMS** must support:(DBMS需要支持增删改查，遍历)

–insert/delete/modify record

–read a particular record (specified using record id)

–scan all records (possibly with some conditions on the records to be retrieved), or scan a range of records

2. **DBMS** admin generally creates indices to allow almost direct access to individual items

DBMS admin通过创建index来直接访问独立的item。

当join需要条件时，Index也可以改善

- These are also good during join operations if there is a join condition that restricts the number of items to be joined in a table

What is indexing

Indexing 对于efficiency至关重要.

- Indexing mechanisms used to speed up access to desired data in a similar way to look up a phone book or dictionary(索引使访问数据像查字典)
- Search Key - attribute or set of attributes used to look up records/rows in a system like an ID of a person()
- An index file consists of records (called index entries) of the form **search-key, pointer to where data is**(index file由搜索键、指向数据所在位置的指针等形式的记录(称为索引项)组成)
- Index files are typically much smaller than the original data files and many parts of it are already in memory(索引文件通常比原始数据文件小得多，并且它的许多部分已经在内存中)
- Disk access becomes faster(Disk访问变快)
- **Insertion time** to index is also important()
- **Deletion time** is important as well()
- No big index rearrangement after insertion and deletion(增加和删除数据后 index 没有发生明显的rearrangement)
- Space overhead needs to be considered for the index itself(需要考虑index本身的space开销)
- No single indexing technique is the best. Rather, each technique is best suited to particular applications(没有哪一种indexing技术是最好的，只有最合适的)

Different types of indexes

Search using Indexes

We have seen some example index structures

- B+ tree – finding a particular value, finding a range of values
- Hash index – finding a particular value
- Bitmap index – finding total number
- Spatial indexes – range query, nearest neighbor query
  - Quadtree
  - R-tree

基于search key的基础分类

Ordered indices: search keys are stored in some order(search key有序)

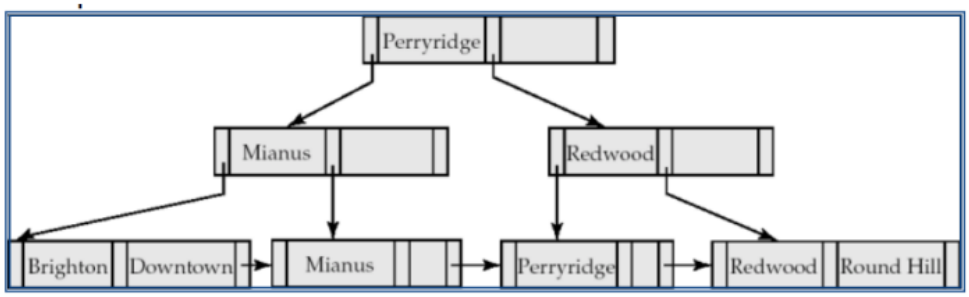
- Clustering index/ Primary index:线性连续的，通常使用主键作为key，例如111，222，333
- Non-clustering index/ Secondary index:非连续性，但仍然存在order，例如a,b,c,d,e 提高了使用cluster索引的搜索键以外的键的查询的性能。

Hash indices: search keys are distributed hopefully uniformly across “buckets” using a “function”(search key 用哈希表储存)

B+tree

Most Popular Index in DMBS: B+Tree

- Features:
  - Keeps the data in order
  - Enables Binary Search
  - Maintained by periodic reorganization
  - If there are K search-key values in the file, the height of the tree is no more than $\log_{n/2}(K)$ and it would be balanced.
  - Perfect for Range queries.



Why need them:

- Keeping files in order for fast search ultimately degrades as file grows, since many overflow blocks get created.(更快的增加新文件，防止溢出的block被创建)
- So binary search on ordered files cannot be done.
- Periodic reorganization of entire file is required to achieve this.

Advantage of B+-tree index files**:**

- Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.(面临插入与删除时，自动识别小改动，无需重新组织整个文件)
- Reorganization of entire file is not required to maintain performance.

Disadvantage of B+ -trees:

- Extra insertion and deletion overhead and space overhead(需要额外的空间进行insert 和delete,需要额外的空间)

由于优点远大于缺点，B+tree被广泛使用。

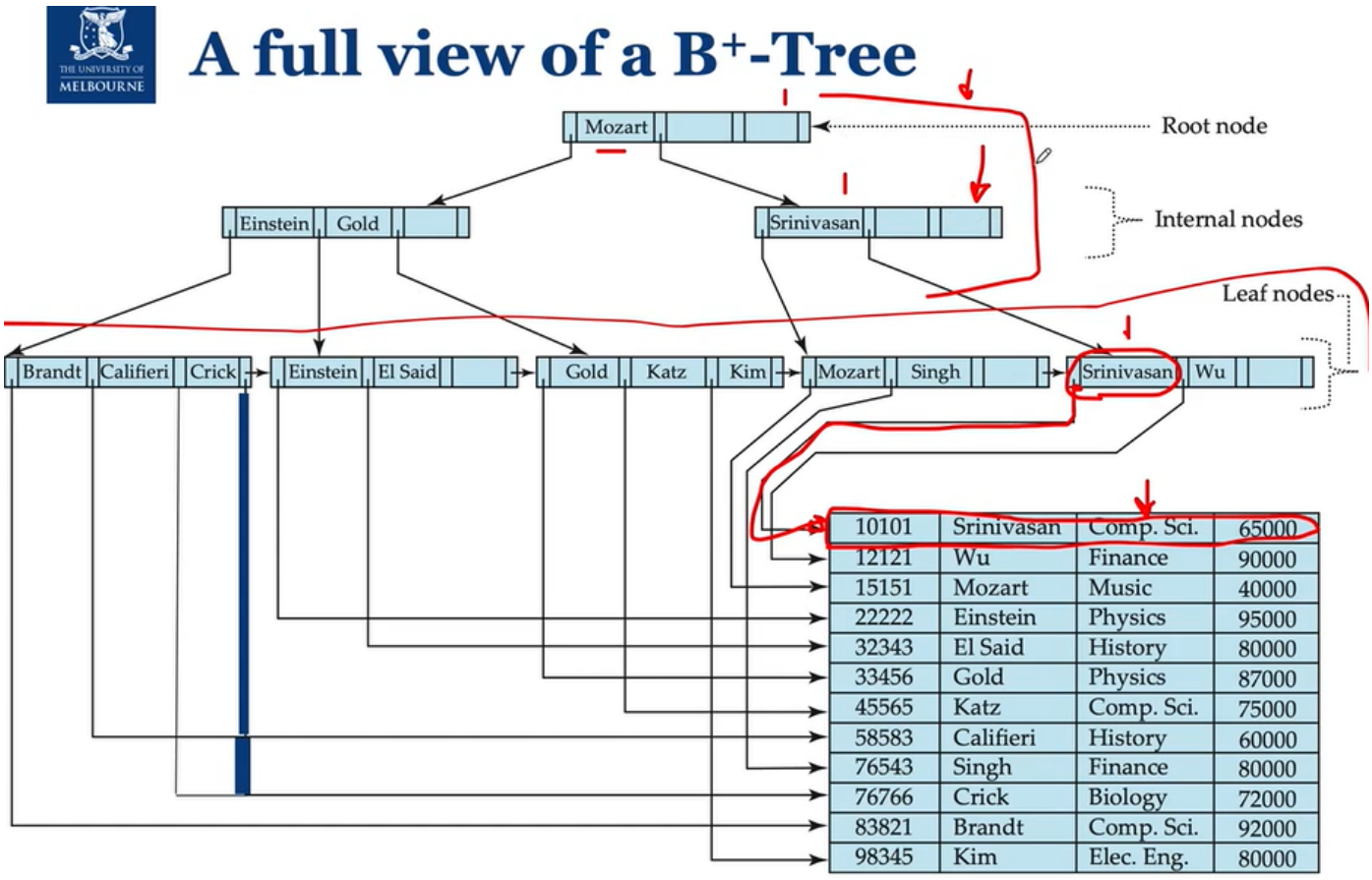
A Single Node

- Typical node

  - 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)
- The search-keys in a node are ordered

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

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



- B+tree的Root node和Internal node被储存在main memory(RAM)中，Leaf node通常被储存在disk中，所以访问前两个node会比后者快
- leaf node直接存储数据而不是pointer.

知道**n**求**height**

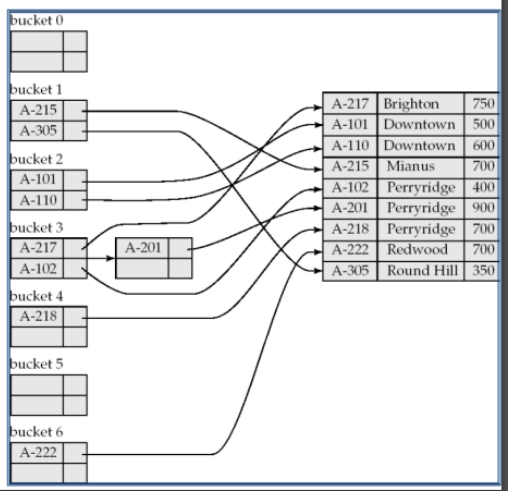
The maximum number of children of a node, is denoted as n.

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ and **it would be balanced**
- 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).
- With 1 million search key values and $n = 100$
  - $log_{50}(1{,}000{,}000)$ = **4 nodes are accessed in a lookup**.
- 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

hash index

Other forms of indices: Hash Indices

- Unordered, but record pointers are used with search keys.
- Given a key the aim is to find the related record on file in one shot which is important.
- An good hash function is uniform (same per buc)
- Ideal hash function is random



bitmap index

Bitmap Indices

- Records in a relation are assumed to be numbered sequentially from, say, 0
- 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)
- A bitmap is simply an array of bits.

- 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 Num	Name	State	Income_level
0	John	VIC	L1
1	Diana	NSW	L2
2	Xiaolu	WA	L1
3	Anil	VIC	L4
4	Peter	NSW	L3

Bitmap for income level	Income_level
L1	1 0 1 0 0
L2	0 1 0 0 0
L3	0 0 0 0 1
L4	0 0 0 1 0
L5	0 0 0 0 0

Quadtree

More on Quadtrees

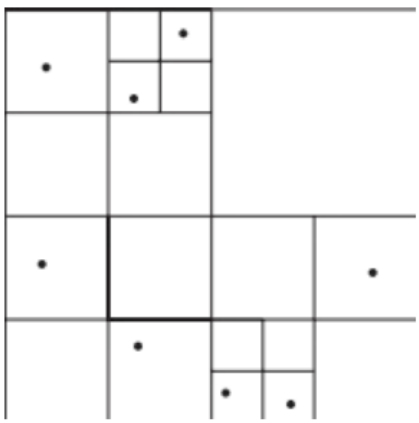
Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.

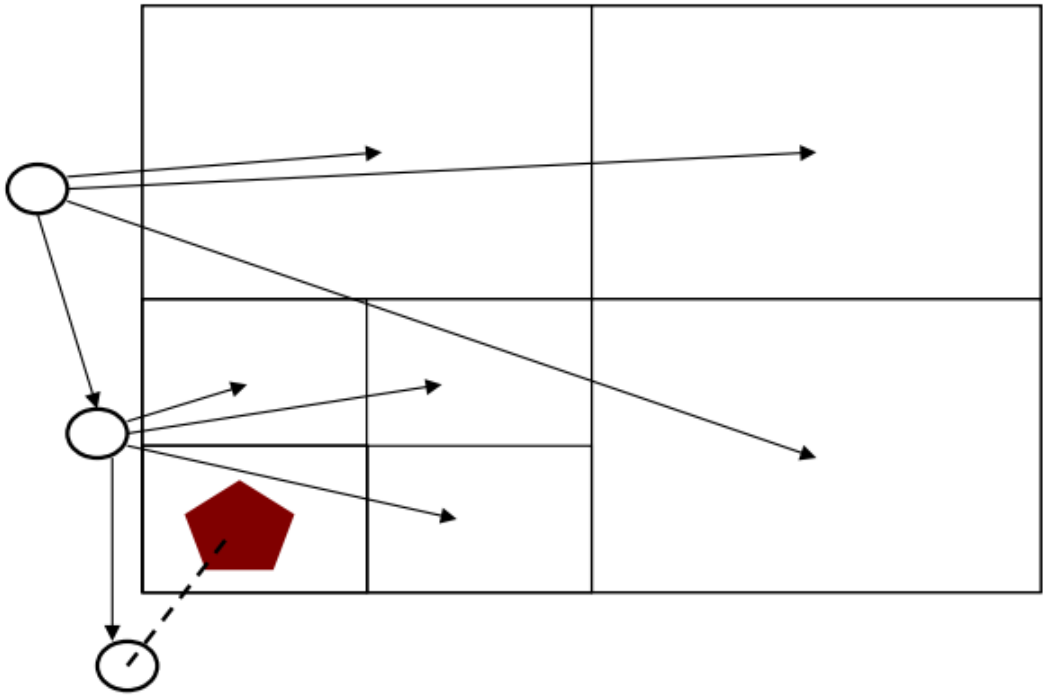
Each division happens with respect to a rule based on data type.

Each non-leaf nodes divides its region into four equal sized quadrants

Thus each such node has four child nodes corresponding to the four quadrants and division continues recursively until a stopping condition

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





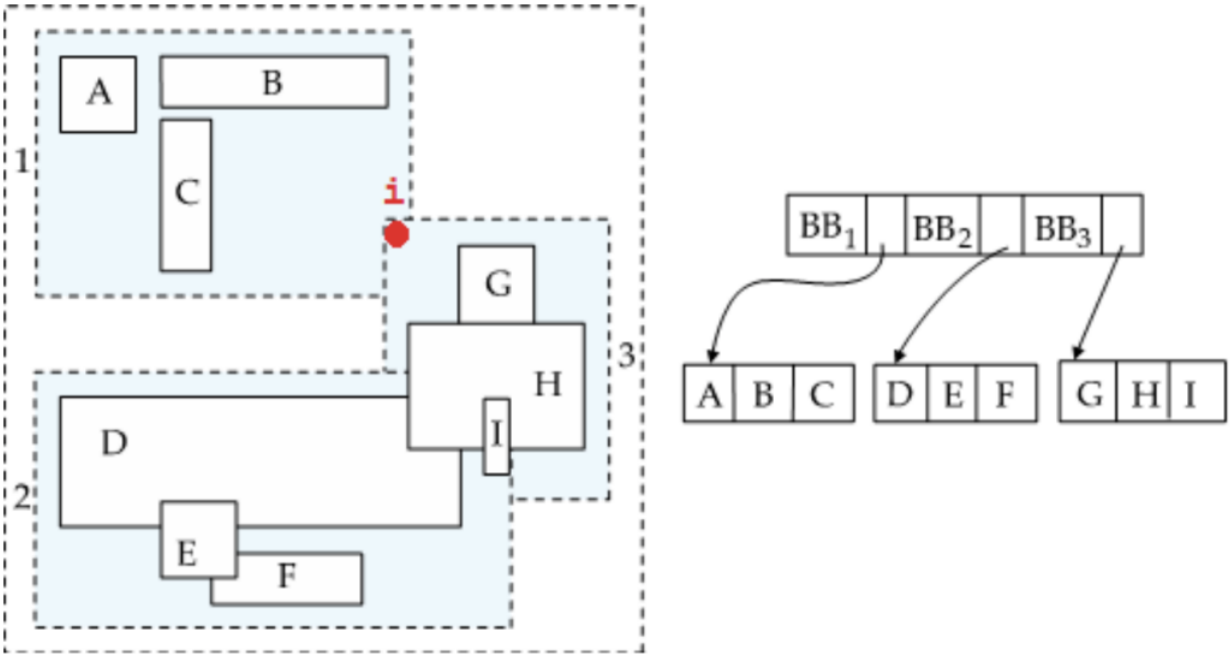
R-tree

R-Trees

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

search in R tree

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



In this traversal we first visit node BB1 as it overlaps with the query point. And find that object B is the closest to query point i. The issue here is that we cannot stop at this point in the traversal as i overlaps with BB3 as well so we need to investigate the data there too. We then figure out that G is the closest object overall. Due to overlaps in R-tree branches two or more branches of an Rtree need to be investigated in many query types. In addition, as each internal node represents a bounding box, thus we are not sure about the position of objects in a bounding box which may necessitate that we investigate multiple bounding boxes to determine a nearest neighbour in this case.

Search in R-Trees：

- 从root node开始
- 如果root 为搜索对象即返回
- else 递归搜索child下所有满足bounding的区域，若满足两个rule，则都要搜索。
- 由于overlap, worst case会非常inefficiency,但是实际使用情况可以被接受。

Nearest Neighbor Query on R-tree

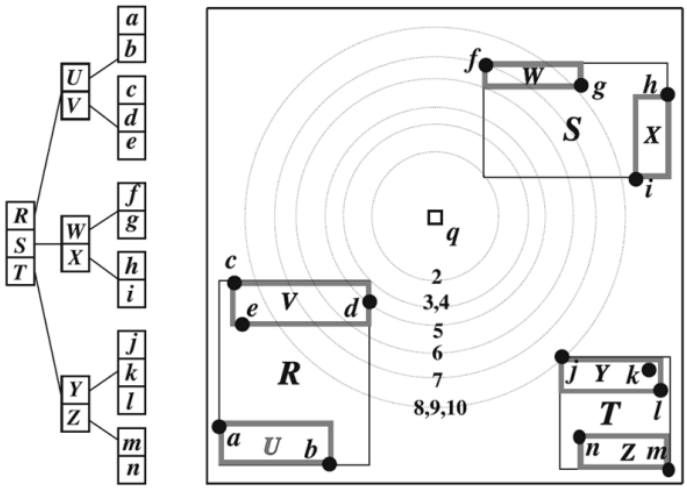
Nearest Neighbour in R-Trees

To find the nearest neighbour of a given query point/region, do the following, starting from the root node:

- Use a sorted priority queue of the R-tree nodes based on the minimum distance from the query
- Traverse the node that is in the top of the priority queue, and put its elements in the queue. Continue
- Stop when the top node is a data object (first NN has been found)

This algorithm is a best-first search algorithm

Nearest Neighbor Query on R-tree



Step	Priority queue	Retrieved NN(s)
1	⟨S, R, T⟩	⟨⟩
2	⟨R, W, T, X⟩	⟨⟩
3	⟨V, W, T, X, U⟩	⟨⟩
4	⟨d, W, T, X, c, e, U⟩	⟨⟩

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

Usage of indexes

- index可以被DBMS自动创建或者被手动创建
- **UNIQUE** constraint, DBMS creates a **nonclustered** index.
- **PRIMARY KEY**, DBMS creates a **clustered** index
- can create indexes on any relation (or view)

Indexing in practice

Create & drop

创建时可以指定type

unique index is one in which no two rows are permitted to have the same **index key value**

Create an index

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

To drop an index

drop index <index-name>

Most database systems allow specification of type of index, and clustering

Index Definition in SQL

1. Create a clustered index on a table

   CREATE CLUSTERED INDEX index1 ON table1 (column1);

2. Create a non-clustered index with a unique constraints

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

   (A unique index is one in which no two rows are permitted to have the same index key value)

Index Definition in SQL

Specialized indexes

Filtered index

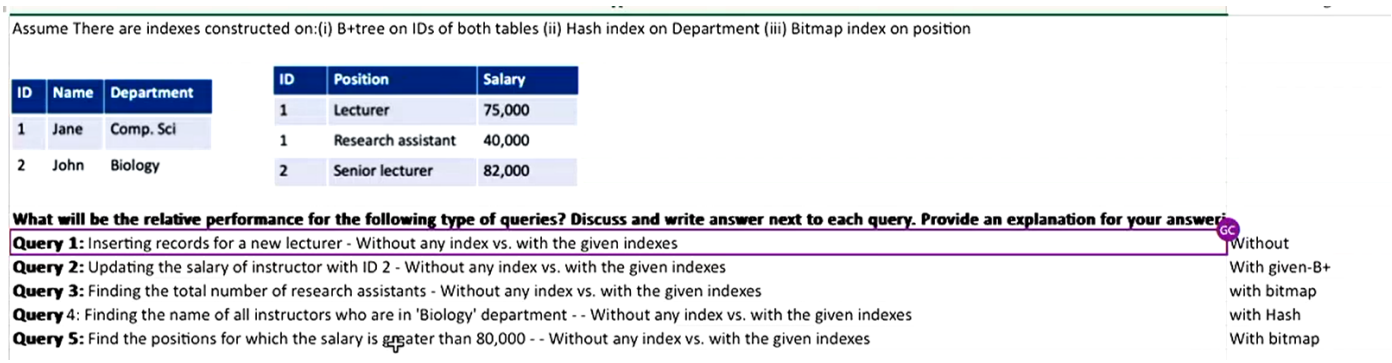
CREATE INDEX index1 ON table1 (column1)

WHERE Year > '2010';

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

Spatial index

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



Assume There are indexes constructed on:(i) B+tree on IDs of both tables (ii) Hash index on Department (iii) Bitmap index on position

ID	Name	Department
1	Jane	Comp. Sci
2	John	Biology

ID	Position	Salary
1	Lecturer	75,000
1	Research assistant	40,000
2	Senior lecturer	82,000

What will be the relative performance for the following type of queries? Discuss and write answer next to each query. Provide an explanation for your answer

Query 1: Inserting records for a new lecturer - Without any index vs. with the given indexes	Without
Query 2: Updating the salary of instructor with ID 2 - Without any index vs. with the given indexes	With given-B+
Query 3: Finding the total number of research assistants - Without any index vs. with the given indexes	with bitmap
Query 4: Finding the name of all instructors who are in 'Biology' department - - Without any index vs. with the given indexes	with Hash
Query 5: Find the positions for which the salary is greater than 80,000 - - Without any index vs. with the given indexes	With bitmap

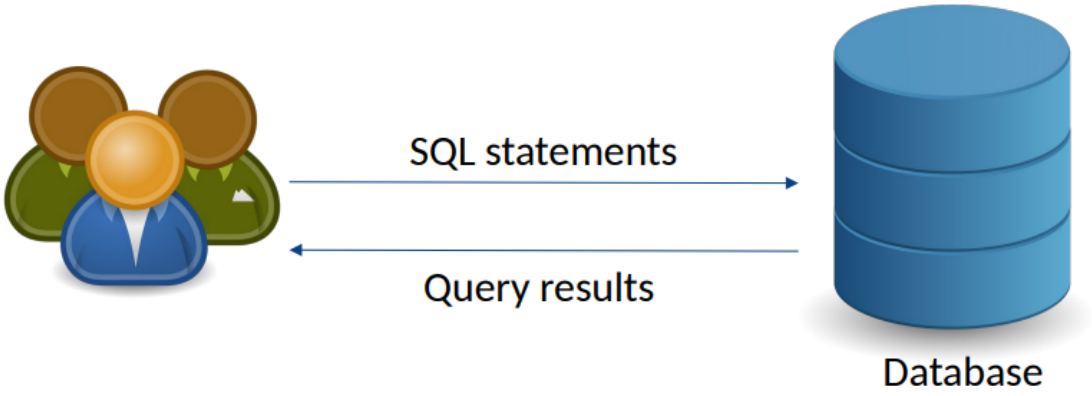
query1.当insert 发生时，此情况下没有Index最快，两张表都要改动，bitmap无法handle。

5为 without

SQL injection

没有对用户输入进行格式化与处理

Discussion topic – SQL injection



Can happen when an application executes database query using user-input data, and the user input or part of the user input is treated as SQL statement

如何预防？

用户参数化查询/准备语句——允许数据库区分代码和数据

Prevention

User parameterized query/prepared statement - allows the database to distinguish between code and data

```
String query = "SELECT * from login where user = "
+ request.getParameter("userName");
```

WK3

Transactions

Transaction - A **unit of work** in a database

- A transaction can have any number and type of operations in it
- Either happens as a whole or not

- Transactions ideally have ACID properties((Atomicity, Consistency, Isolation, Durability)

ACID

Atomicity(单元性) - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are. Example – A transaction that (i) subtracts $100 if balance > 100 (ii) deposits 100$ to another account (both actions with either happen together or none will happen)(所有的action要么一起发生要么一起不发生)

Consistency(一致性) - Data is in a 'consistent' state when a transaction starts and when it ends – in other words, any data written to the database must be valid according to all defined rules (e.g., no duplicate student ID, no negative fund transfer, etc.) • What is 'consistent' - depends on the application and context constraints • It is not easily computable in general • Only restricted type of consistency can be guaranteed, e.g. serializable transactions which will be discussed later.

Isolation(隔离性)- transaction are executed as if it is the only one in the system.(互不影响) • For example, in an application that transfers funds from one account to another, the isolation ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability(持久性)- the system should tolerate system failures and any committed updates should not be lost.

Types of Actions

- **Unprotected actions** - no ACID property
- **Protected actions** - these actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.(可以被rollback)
- **Real actions** - these are real physical actions once performed cannot be undone. In many situations, atomicity is not possible with real actions (e.g., firing two rockets as a single atomic action)(无法被rollbaclk)

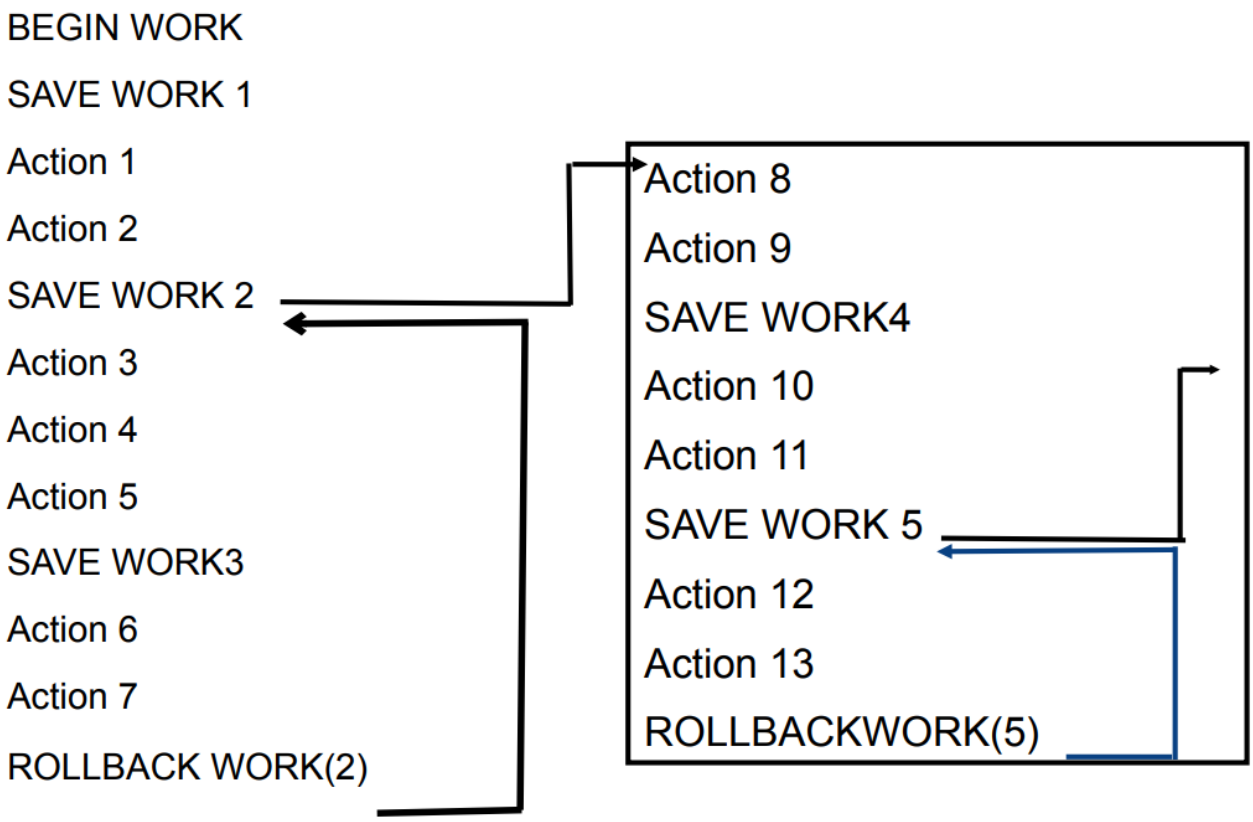
Types of transactions

Flat transaction：

Any failure of transaction requires lot of unnecessary computation. (一旦出错所有的工作都rollback)

Everything inside BEGIN WORK and COMMIT WORK is at the same level; that is, the transaction will either survive together with everything else (commit), or it will be rolled back with everything else (abort)

Flat transaction with **save points**



rollback回checkpoint

Nested transaction

Commit rule

A subtransaction can either commit or abort, however, commit cannot take place unless the parent itself commits.

Subtransactions have A, C, and I properties but not D property unless all its ancestors commit.

Commit of a sub transaction makes its results available only to its parents.

Roll back rule

If a subtransaction rolls back, all its children are forced to roll back.(父退回时子退回)

Visibility Rules

Changes made by a subtransaction are visible to the parent only when the subtransaction commits. All objects of parent are visible to its children. Implication of this is that the parent should not modify objects while children are accessing them. This is not a problem as parent does not run in parallel with its children.(子提交的父可见，父的所有子可见，子占用时父不占,子可以与其他子并行，不可与父并行)

Transaction processing monitor(TP monitor)

The main function of a TP monitor is to integrate other system components and manage resources.

TP监视器的主要功能是集成其他系统组件和管理资源。

- TP monitors manage the transfer of data between clients and servers

  TP监视器管理客户端和服务器之间的数据传输

- Breaks down applications or code into transactions and ensures that all databases are updated properly

  将应用程序或代码分解为事务，并确保所有数据库都正确更新

- It also takes appropriate actions if any error occurs

  它也采取适当的行动，如果任何错误发生

TP monitor services

Heterogeneity: If the application needs access to different DB systems, local ACID properties of individual DB systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure the overall ACID property. A form of 2 phase commit protocol must be employed for this purpose.

Control communication: If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes to be able to recover from a crash.

Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID property between the client and the server

processes.

Presentation service: this is similar to terminal management in the sense it has to deal with different presentation (user interface) software – e.g. X-windows

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

Start/Restart: There is no difference between start and restart in TP based system.

Concurrency control

why: Implementation of exclusive access(目的是实现独有访问)

To resolve conflicts and preserve database consistency(解决冲突，保持数据一致性)

通常使用spin lock

Dekker's algorithm (using code)

needs almost no hardware support, needs atomic reads and writes to main memory, That is exclusive access of one time cycle of memory access time!

the code is very complicated to implement **for more than two** transactions

harder to understand the algorithm for more than two transactions

takes lot of storage space

efficient for low lock contention

uses busy waiting

OS supported primitives

need no special hardware

independent of number of processes

machine independent

through an interrupt call, the lock request is passed to the OS

are very **expensive** (several hundreds to thousands of instructions need to be executed to save context of the requesting process.)

do not use busy waiting and therefore **more effective**

Spin locks

(访问之前查看上锁状态)

using atomic lock/unlock instructions such as test and set or compare and swap

缺点**need hardware support** - should be able to lock bus (communication channel between CPU and memory + any other devices) for two memory cycles (one for reading and one for writing). During this time no other devices' access is allowed to this memory location.

优点algorithm does not depend on number of **processes**

优点 are very efficient for low lock contentions (all DB systems use them)

use busy waiting

Atomic operations

to get and release locks

Test and set: 查看是否上锁，没上锁就上锁返回true，上了锁返回false

Compare and swap: 查看是否和旧值相等，相等则赋予新值并返回true，否则重置旧值返回false

Semaphore

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.

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

Semaphores are very simple locks; indeed, they are used to implement

general-purpose locks.

Implementation of Exclusive mode Semaphore(说白了就是链表排队)

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

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

Convoy avoiding semaphore

避免信号队列过长

The previous implementation may result a long list of waiting processes, called convoy

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

In that case, each of those **processes** will have to **re-execute** the routine for acquiring semaphore.(重新排队)

A quick summary

Concurrency problems – why we need concurrency control

Implementation of exclusive access –

- Dekker's algo
- OS primitives
- **Spin locks** - Atomic operations to get and release locks

Semaphores - get lock, release lock, maintain queue of processes

Avoiding long queues in semaphores

Deadlocks

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

连个程序占用资源，等待对方释放

deadlock不成立的情况: no cycle in the locks held and requested by the transaction.

Solutions

Have enough resources so that no waiting occurs(不现实)

Pre-declare all necessary resources and allocate in a single request.(不现实)

Linearly order the resources and request of resources should follow this order.This type of allocation guarantees no cyclic dependencies among the transactions.

Periodically check the resource dependency graph for cycles. If a cycle exists - rollback (i.e., terminate) one or more transaction to eliminate cycles (deadlocks). The chosen transactions should be cheap.

Allow waiting for a maximum time on a lock then force Rollback.(被大多数公司使用**)**

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

Isolation concepts

Isolation ensures that concurrent transactions leaves the database in the same state as if the transactions were executed separately.隔离确保并发事务使数据库处于相同的状态，就好像这些事务是单独执行的一样。

Isolation guarantees consistency, provided each transaction itself is consistent.隔离保证一致性，前提是每个事务本身是一致的。

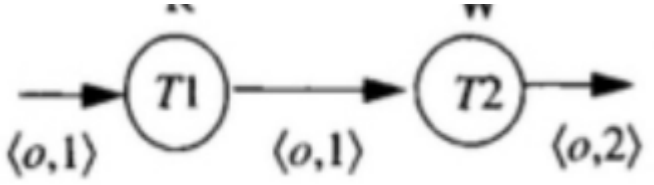
We can achieve isolation by sequentially processing each transaction - generally not efficient and provides poor response times.我们可以通过顺序处理每个事务来实现隔离——通常效率不高，响应时间也很长。

We need to run transactions concurrently with the following **goals**:

• concurrent execution should not cause application programs (transactions) to malfunction.并发执行不应该导致应用程序(事务)故障。

• Concurrent execution should not have lower throughput or bad response times than serial execution.并发执行不应该比串行执行有更低的吞吐量或糟糕的响应时间。

Dependency relations



write-write (Lost update):

覆盖其他transaction的update

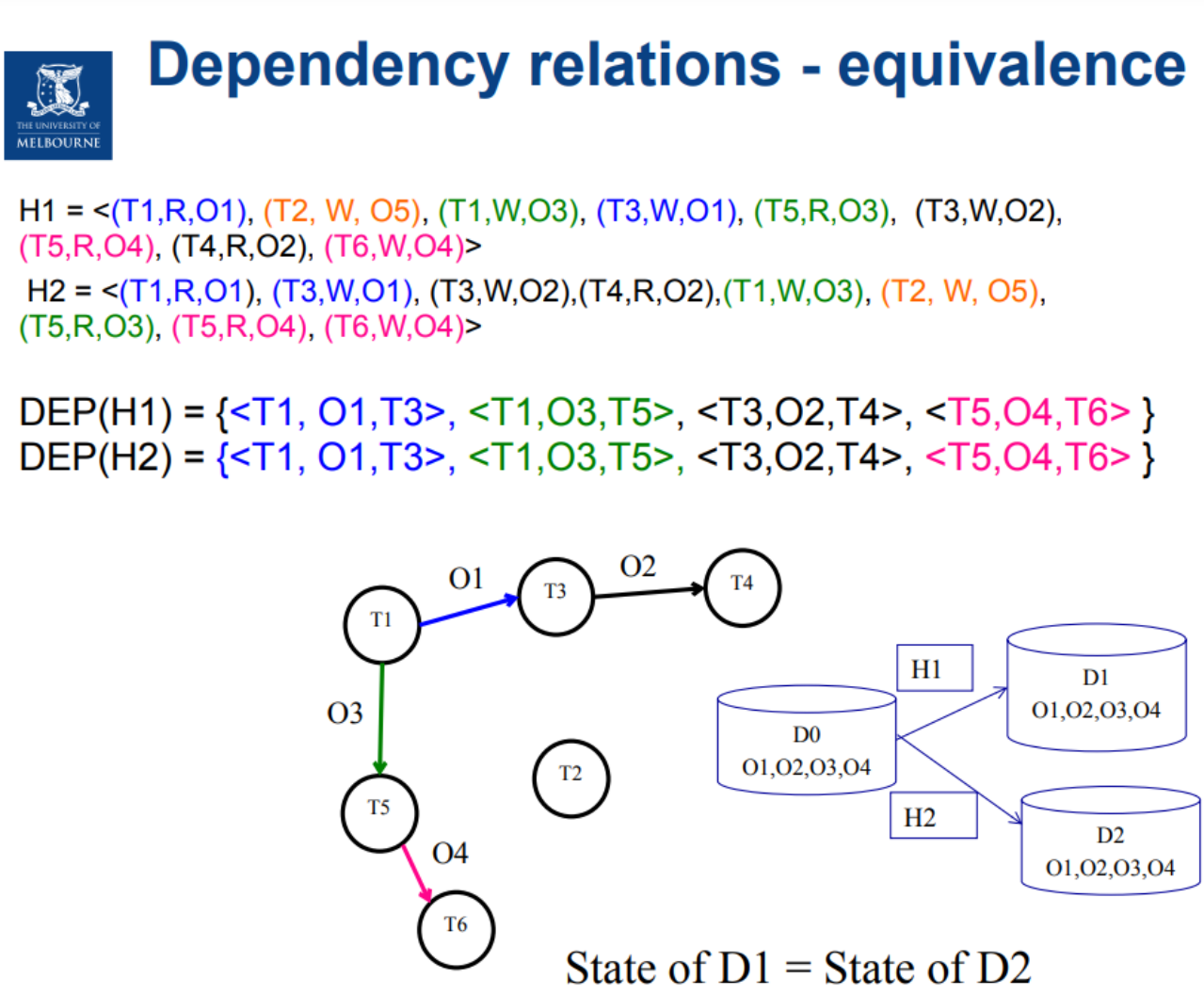
read-write (Unrepeatable read):

之前读取的被另外一个transaction修改导致再次读取结果不一致

write-read (Dirty Read):

读取另外一个transaction未提交的结果

Equivalence



Isolated history

A serial history = an isolated history.

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.

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

A history is legal if does not grant conflicting grants.

lock

SLOCK (shared lock) that allows other transactions to read, but not write/modify the shared resource

为了读取数据而上的锁，防止读取过程中被其他transaction修改

To grant lock or not to…

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

Lock Compatibility Matrix

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

当不处于free状态时，只有SLOCK可以共存。

Actions in Transactions are: READ, WRITE, XLOCK, SLOCK, UNLOCK, BEGIN, COMMIT, ROLLBACK

Wormhole

Wormhole Transaction:既在某一个transaction T前，又同时发生在T之后，形成悖论

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

Isolation concept & theorems

Well-formed transactions

(格式良好**)**:

A transaction is well formed if all READ, WRITE and UNLOCK operations are covered by appropriate LOCK operations

Two phase transactions

(两段式**)**:

A transaction is two phased if all LOCK operations precede all its UNLOCK operations.

A **transactions** 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 does not grant conflicting grants.

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

why two-phase locking guarantees serializability？

• Two-phase locking (2PL) is a protocol that ensures serializability in concurrent transaction execution. Serializability is a property that ensures the results of executing concurrent transactions is equivalent to those results that would have been obtained had the transactions been executed serially (one after the other). • To ensure serializability, we need to prevent scenarios where transactions interleave in such a way that they form a cycle in the precedence graph, as such cycles are the primary source of non-serializable schedules. A cycle in a precedence graph signifies that transactions are dependent on each other in a circular manner, which implies a non-serializable schedule. • The Two-phase locking protocol works in two phases: • Growing Phase (also known as Locking or Expansion phase): The transaction may obtain (but not release) locks. • Shrinking Phase (also known as Unlocking or Contraction phase): The transaction may release locks, but cannot obtain new ones.

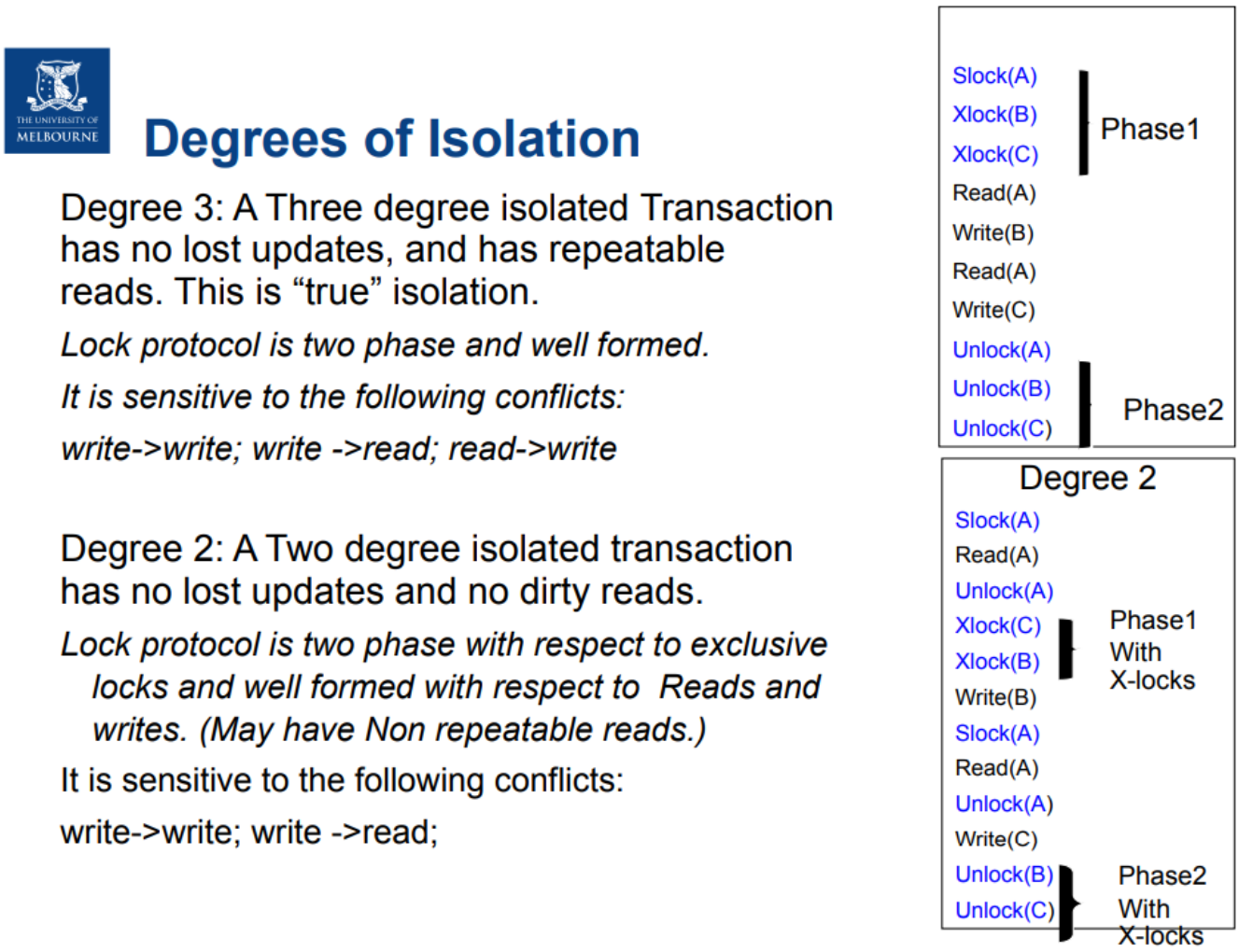
Locking theorem: If **all transactions** are **well formed** (READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation) and **two-phased** (locks are released only at the end), then any **legal** (does not grant conflicting grants) history will be isolated.

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

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

先UNLOCK再ROLLBACK就不是两段式

Degrees of isolation

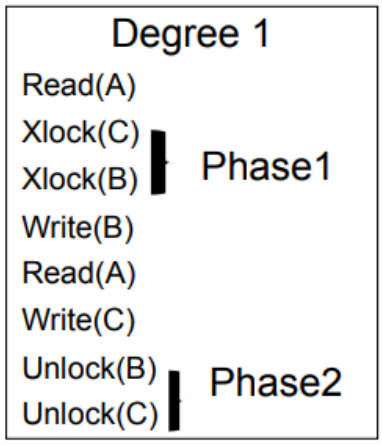


Degree 1: A One degree isolation has no lost updates.

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

It is sensitive the following conflicts:

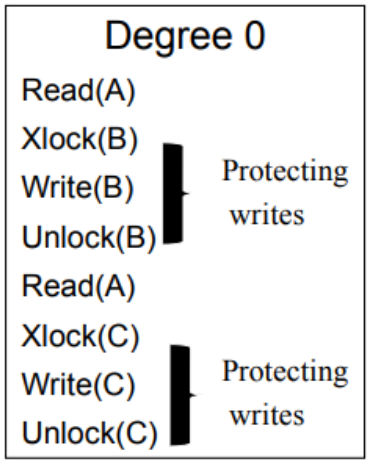
write->write;



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

Lock protocol is well-formed with respect to writes.

It ignores all conflicts.



Degree 3: A Three degree isolated Transaction has no lost updates, and has repeatable reads. This is “true” isolation. (read write两段锁)

sensitive to: write->write; write ->read; read->write

Degree 2: A Two degree isolated transaction has no lost updates and no dirty reads. (write锁是两段，read完美锁)

sensitive to: write->write; write ->read

Degree 1: A One degree isolation has no lost updates. (write两段锁，read没有锁)

sensitive to: write->write

Degree 0 : A Zero degree transaction does not overwrite another transactions dirty data if the other transaction is at least One degree. (write完美锁，read没有锁)

It ignores all conflicts

Granular locks

背景:锁住整个数据库太过于浪费，划分层级，使用粒度锁，锁越多，冲突越少。

Use granular locks - we need to build some hierarchy, then locks can be taken at any level, which will automatically grant the locks on its descendants.

Lock the whole DB – less conflicts, but poor performance

Lock at individual records level – more locks, better performance

To acquire an S mode or IS mode lock on a non-root node, one parent must be held in IS 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}).

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)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)

注:current是S可以用U，current是U无法用S

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.

S shared lock. Allows read authority to the node and its descendants at a finer granularity and prevents others holding IX, X, SIX on this node.

U update lock – Intention to update in the future. 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.

IS Intent to set shared locks at finer granularity. Allows IS and S mode locks at finer granularity and prevents others from holding X, U on this .nu node.

IX Intent to set shared or eXclusive locks at finer granularity. 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.

SIX a coarse granularity shared lock with an Intent to set finer granularity exclusive locks. 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.

Optimistic locking

乐观锁，当冲突较少时使用

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

Before committing, each transaction verifies that no other transaction has modified the data - duration of locks are very short(在做update的时候先检查一下用到的值始否改变了)

- If any **conflict** found, the transaction **repeats** the **attempts**
- If **no conflict**, make **changes** and **commit**

Snapshot isolation

***不锁读，保证了高并发性

Do not guarantee Serializability. However, its transaction throughput is very high compared to two phase locking scheme.

不保证可序列化性。然而，与两阶段锁定方案相比，其事务吞吐量非常高。

Disadv：Snapshot isolation does not care if the read set, which is not modified by the current transaction, is already modified by another transaction when the current transaction commits. This can cause consistency issues in applications.

不关心当前事务提交时未被当前事务修改的读集是否已被另一个事务修改。这可能会导致应用程序中的一致性问题。

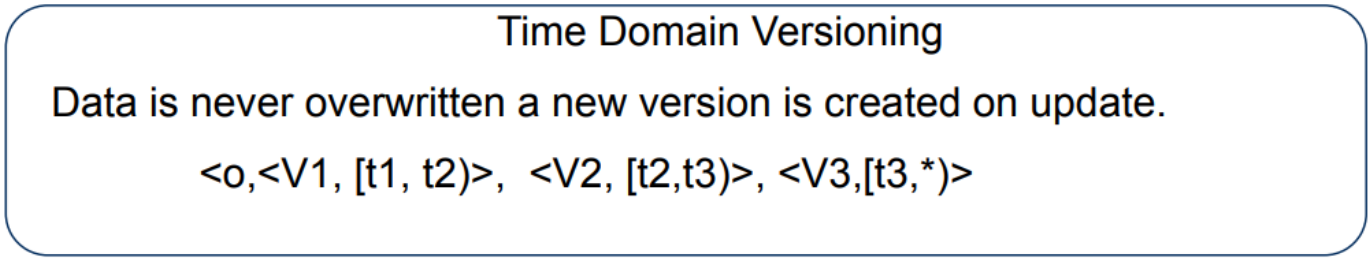
Adv：As a kind of Optimistic Locking, snapshot isolation guarantees all the read operations made in a transaction will see a consistent snapshot of the database, but the transaction will abort when other concurrent transaction which have conflict with it.

作为乐观锁的一种，快照隔离保证了在一个事务中进行的所有读操作都能看到一个一致的数据库快照，但当其他并发事务与它发生冲突时，该事务将终止。

Timestamping

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

这是乐观并发控制的一种特殊情况。在提交时，检查时间戳。如果时间戳比事务读取时间更近，则事务将中止.



At the commit time, the system validates all the transaction’s updates and writes updates to durable media. This model of computation unifies concurrency, recovery and time domain addressing.在提交时，系统验证所有事务的更新并将更新写入持久介质。该计算模型将并发、恢复和时域寻址相结合

T1	T2
select average (salary) from employee	update employee set salary = salary*1.1 where salary < $40000

当T1正在读取时，通常来说T2应该等待SLOCK 解开，然而有了timestamp,它可以不用等待。

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 some times
- 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!!***

WK4

Crash recovery

Buffer pool and disk transfers

！！！！！：被存入disk里的数据视为durable,存储在buffer pool里的数据遭遇crash后不会幸存

Data is stored on disks(数据存在disk上)

Reading a data item requires reading the whole page of data (typically 4K or 8K bytes of data depending on the page size) from disk to memory containing the item.(读数据时需要把数据从disk读到memory，一页一页读)

Modifying a data item requires reading the whole page from disk to memory containing the item, modifying the item in memory and writing the whole page to disk.(修改数据需要把所有的数据从memory存回disk)

Steps 2 & 3 can be very expensive and we can minimize the number of disk reads and writes by storing as many disk pages as possible in memory (buffer cache) - this means always check in buffer cache for the disk page of interest if not copy the associated page to buffer cache and perform the necessary operation.(存读的过程很贵，尽可能多的把数据读到memory上就不要动)

When buffer cache is full we need to evict some pages from the buffer cache in order fetch the required pages from the disk.(buffer满了之后要释放一些page到disk)

Eviction needs to make sure that no one else is using the page and any modified pages should be copied to the disk.(释放的时候要确保没人要用这些page)

Since several transactions are executing concurrently this requires additional locking procedures using latches. These **latches** are used only for the duration of the operation (e.g. **READ/WRITE**) and can be released immediately unlike record locks which have to be kept locked until the end of the transaction.(使用latch, 类似于lock, 来保证读写的过程,一旦完成R/W立即释放，不像lock会等Xact完成才释放)

fix(pageid)

- reads pages from disk into the buffer cache if it is not already in the buffer cache
- fixed pages cannot be dropped from the buffer cache as transactions are accessing the contents(还在用的page不能丢出buffer)

unfix(pageid)

- The page is not in use by the transaction and can be evicted as far as the unfix calling transaction is concerned. (We need to check to see that no one else

wants the page before it can be evicted)（踢回disk之前要见检查page没有被使用）

Force write to disk at commit?

- Poor response time.
- But provides durability.

NO Force leave pages in memory as long as possible even after commit without modifying the data on the disk.（不强制把commit的存入disk会给Durability增加难度）

- Improves response time and efficiency as many reads and updates can take place in main memory rather than on disk.
- Durability becomes a problem as update may be lost if a crash occurs
- Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

Steal buffer-pool frames from uncommitted Xacts?（没有commit就存到disk会给Atomicity增加难度）

- To steal frame F: Current page in F (say P) is written to disk; some Xact holds lock on P.
- If not, poor throughput.
- If so, how can we ensure atomicity?

Logging and WAL

Logging

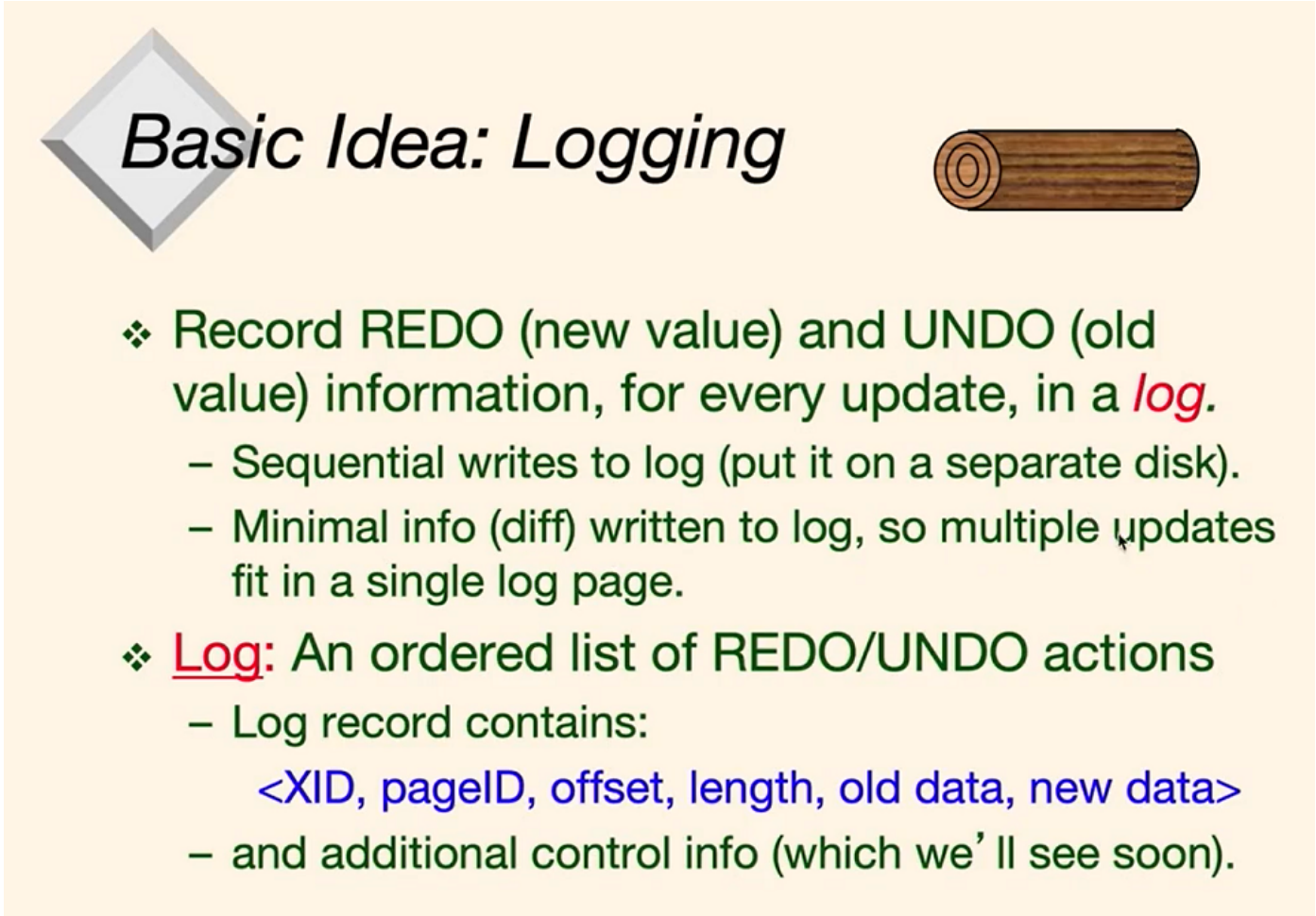
Record REDO (new value) and UNDO (old value) information, for every update, in a log.

- **Sequential** writes to log (put it on a **separate disk**).注. Log存在disk上
- **Minimal info** (diff) written to log, so multiple updates fit in a single log page.

Log: An **ordered** list of REDO/UNDO actions

– Log record contains:

- <XID, pageID, offset, length, old data, new data>
- and additional control info



Write-Ahead Logging(WAL)Protocol

log先写入协议

- Must force the log record which has both old and new values for an update **before** the corresponding data page gets to disk (stolen).必须在对应的数据页进入磁盘(steal)之前强制更新具有新旧值的日志记录。
- Must write all log records to disk (force) for a Xact before commit.必须在提交之前将所有日志记录写入磁盘(force)
- guarantees Atomicity because we can undo updates performed by aborted transactions and redo those updates of committed transactions.
- guarantees Durability. Exactly how is logging (and recovery!) done.
- Use WAL to allow STEAL/NO-FORCE with out sacrificing correctness.WAL可以在不牺牲正确性的情况下达到STEAL/NO-FORCE
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).通过prevLSN查找前一条log
- pageLSN allows comparison of data page and log records.

WAL log

Each log record has a unique Log Sequence Number (LSN).

- LSNs always increasing.

Each **data page** contains a **pageLSN**.

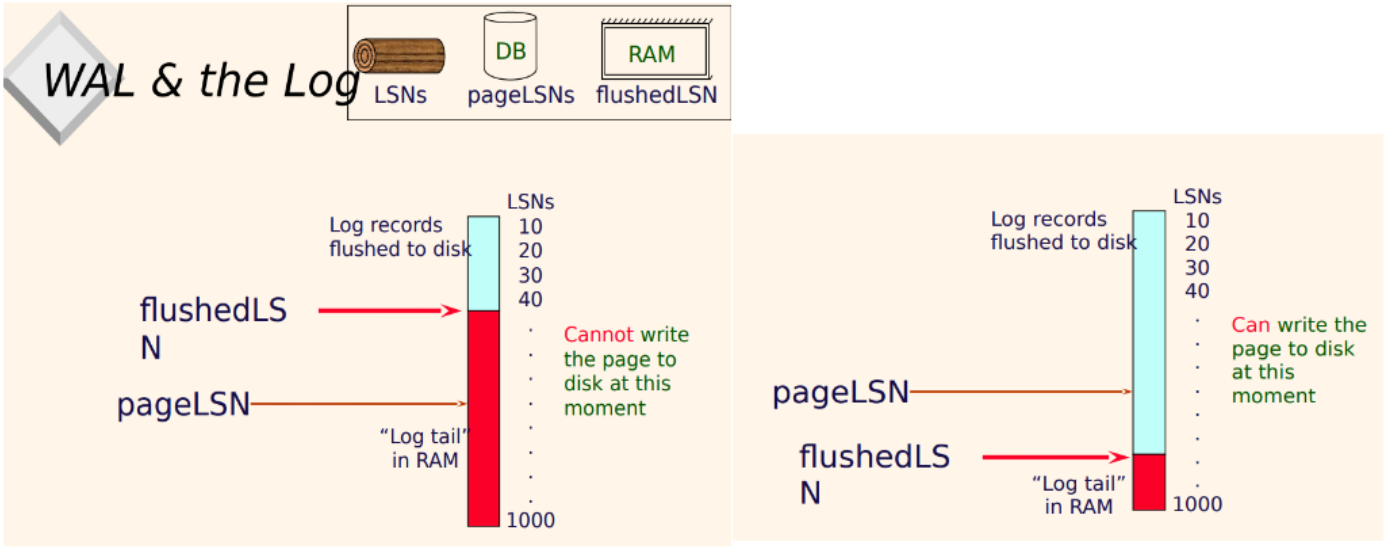
- The LSN of the most recent log record for an update to that page.

System keeps track of flushedLSN.

- The max LSN flushed so far.

WAL: Before a page is written to disk make sure pageLSN <= flushedLSN

- Log, Transaction and Dirty Page tables



Log Records

LogRecord fields:

prevLSN
XID
type
pageID
length
offset
before-image
after-image

update records only

Possible log record types:

- Update
- Commit
- Abort
- End (signifies end of commit or abort)
- Compensation Log Records (CLRs)
  - for UNDO actions

The Big Picture: What's Stored Where

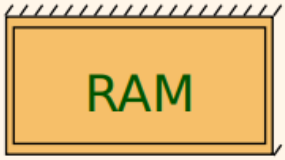
LogRecords

prevLSN
XID
type
pageID
length
offset
before-image
after-image

master record

Data pages

each
with a
pageLSN



Xact Table

lastLSN
status

Dirty Page Table

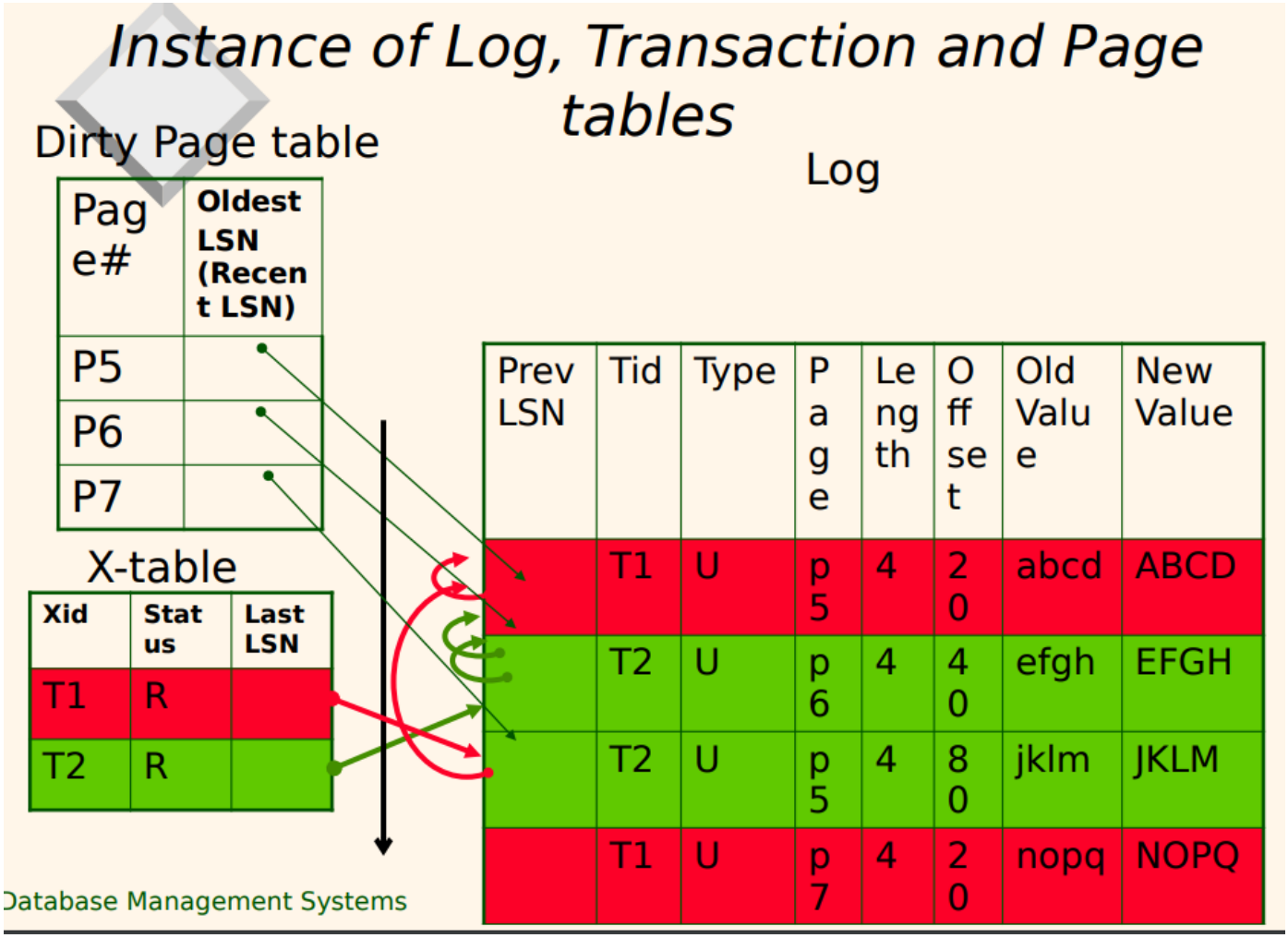
recLSN

flushedLSN

Dirty page table的LSN是最早修改这个page的LSN

Transaction table的LSN是transaction对应的最新LSN

PrevLSN: previous LSN, RecLSN: Record LSN



Normal Execution of an Xact

一个正常的Xact不是被commit就是abort

- Series of reads & writes, followed by **commit** or **abort**.

– We will assume that write is atomic on disk. In practice, additional details to deal with non-atomic writes. We discussed how we do this earlier.

- Strict 2PL.
- STEAL, NO-FORCE buffer management, with
- Write-Ahead Logging.

Checkpointing

Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash.

Begin checkpoint record: Indicates when chkpt began.

End checkpoint record: Contains current Xact table and dirty page table. This is a ‘fuzzy checkpoint’:

- Other Xacts continue to run; so these tables accurate only as of the time of the begin checkpoint record.
- No attempt to force dirty pages to disk; effectiveness of checkpoint is limited by the oldest unwritten change to a dirty page. (So it's a good idea to periodically flush dirty pages to disk!)

Store LSN of chkpt record in a safe place (master record).

无论abort或是commit，成果后都会以"end" record结尾

Simple transaction abort

"play back" the log in reverse order, UNDOing updates.

Get lastLSN of Xact from Xact table.

Can follow chain of log records backward via the prevLSN field.

Before starting UNDO, write an Abort log record.

-For recovering from crash during UNDO!

Before restoring old value of a page, write a CLR (Compensation Log Record):

You continue logging while you UNDO!!

CLR has one extra field: undonextLSN

Points to the next LSN to undo (i.e. the prevLSN of the record we’re currently undoing).

CLRs never Undone (but they might be Redone when repeating history: guarantees Atomicity!)

At end of UNDO, write an “**end**” log record.

Transaction commit

Write commit record to log.

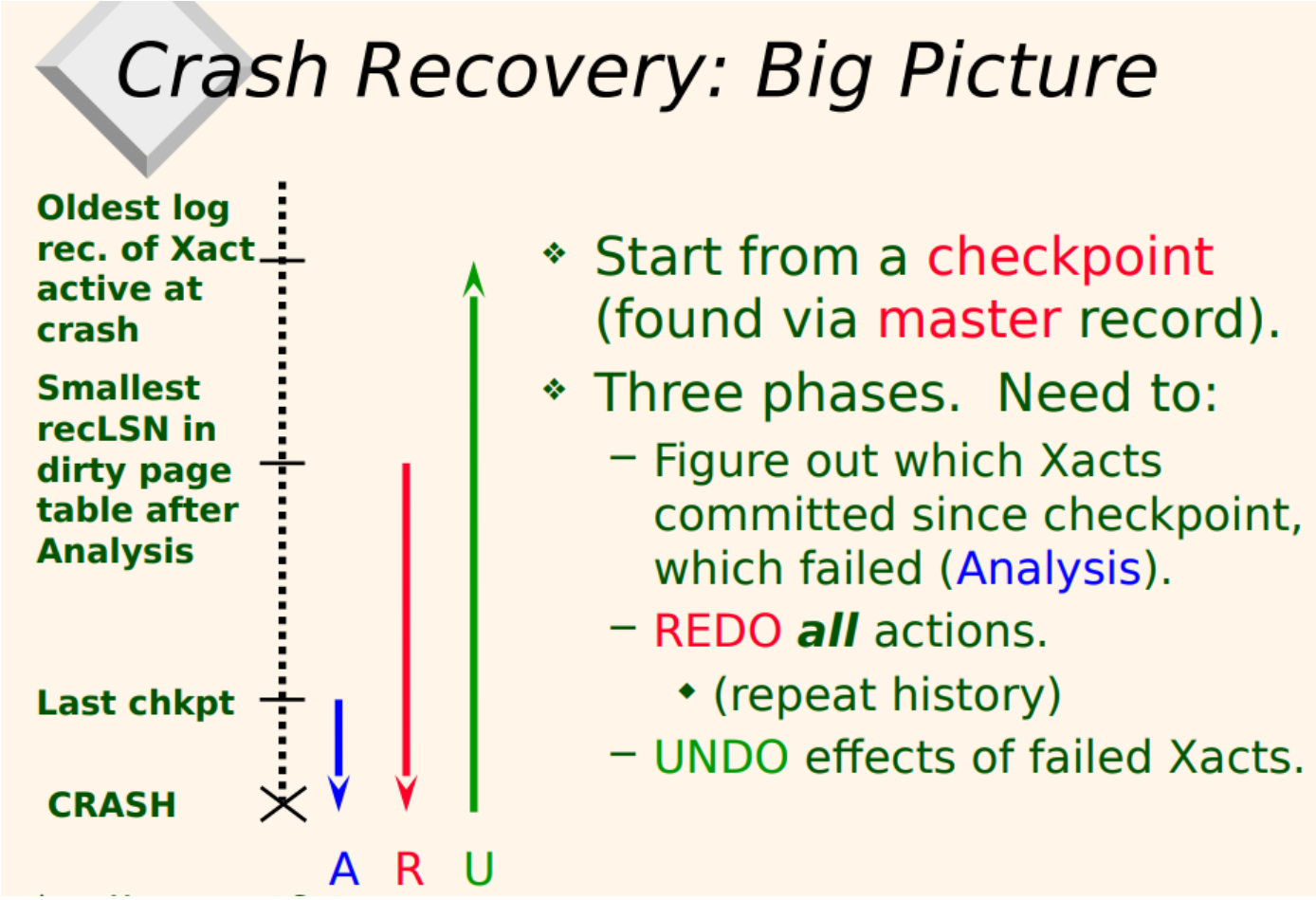
All log records up to Xact's lastLSN are flushed.

- Guarantees that flushedLSN>> lastLSN.
- Note that log flushes are sequential, synchronous writes to disk - (very fast writes to disk).
- Many log records per log page - (very efficient due to multiple writes).

Commit() returns.

Write **end** record to log.

ARIES algorithm



Analyse

Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via end_checkpoint record.
- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
  - Update record: If P not in Dirty Page Table,
    - Add P to D.P.T., set its recLSN=LSN.

X-table

Xid	Status	Last LSN
T1	R	
T2	R	

Dirty Page table

Page#	Oldest LSN (Recent LSN)
P5	
P6	
P7	

Redo

Recovery: The REDO Phase

- We *repeat History* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- Scan forward from log rec containing smallest recLSN in D.P.T. For each CLR or update log rec LSN, REDO the action unless:
  - Affected page is not in the Dirty Page Table, or
  - Affected page is in D.P.T., but has recLSN > LSN, or
  - pageLSN (in DB) ≥ LSN.
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging!

Recovery: The UNDO Phase

ToUndo={ *l* | *l* is a lastLSN of a "loser" Xact}

We can form this list from Xtable.

Repeat:

- Choose the largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
  - Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
  - (Q: what happens to other CLRs?)
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T2: UPDATE P3 (OLD: UUU NEW: VVV)
20	BEGIN CHECKPOINT
25	END CHECKPOINT (XACT TABLE=[[T1,10],[T2,20]]; DPT=[[P1,10],[P3,15]])
30	T1: UPDATE P2 (OLD: WWW NEW: XXX)
35	T1: COMMIT
40	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
45	T2: ABORT
50	T2: CLR P1(ZZZ), undonextLSN=15

Analysis phase: Scan forward through the log starting at LSN 20. LSN 25: Initialize XACT table with T1 (LastLSN 10) and T2 (LastLSN 20). Initialize DPT to P1 (RecLSN 10) and P3 (RecLSN 15). LSN 30: Set LastLSN=30 for T1 in XACT table. Add (P2, LSN 30) to DPT. LSN 35: Change T1 status to "Commit" in XACT table LSN 40: Set LastLSN=40 for T2 in XACT table. LSN 45: Change T2 status to "Abort" in XACT table LSN 50: Set LastLSN=50 for T2 in XACT table.

Redo phase: Scan forward through the log starting at LSN 10. LSN 10: Read page P1, check PageLSN stored in the page. If PageLSN<10, redo LSN 10 (set value to ZZZ) and set the page's PageLSN=10. LSN 15: Read page P3, check PageLSN stored in the page. If PageLSN<15, redo LSN 15 (set value to VVV) and set the page's PageLSN=15. LSN 30: Read page P2, check PageLSN stored in the page. If PageLSN<30, redo LSN 30 (set value to XXX) and set the page's PageLSN=30. LSN 40: Read page P1 if it has been flushed, check PageLSN stored in the page. It will be 10. Redo LSN 40 (set value to TTT) and set the page's PageLSN=40. LSN 50: Read page P1 if it has been flushed, check PageLSN stored in the page. It will be 40. Redo LSN 50 (set value to ZZZ) and set the page's PageLSN=50.

Undo phase: T2 must be undone. Put LSN 50 in ToUndo. LSN 50: Put LSN 15 in ToUndo LSN 15: Undo LSN 15 - write a CLR for P3 with "set P3=UUU" and undonextLSN=NULL. Write UUU into P3.

Additional Crash Issues

What happens if system crashes during Analysis? During REDO? 初了UNDO会接着之前的步骤，A和R的数据都会丢失

How do you limit the amount of work in REDO?

- Flush asynchronously in the background.
- Watch “hot spots”!

How do you limit the amount of work in UNDO?

- Avoid long-running Xacts.

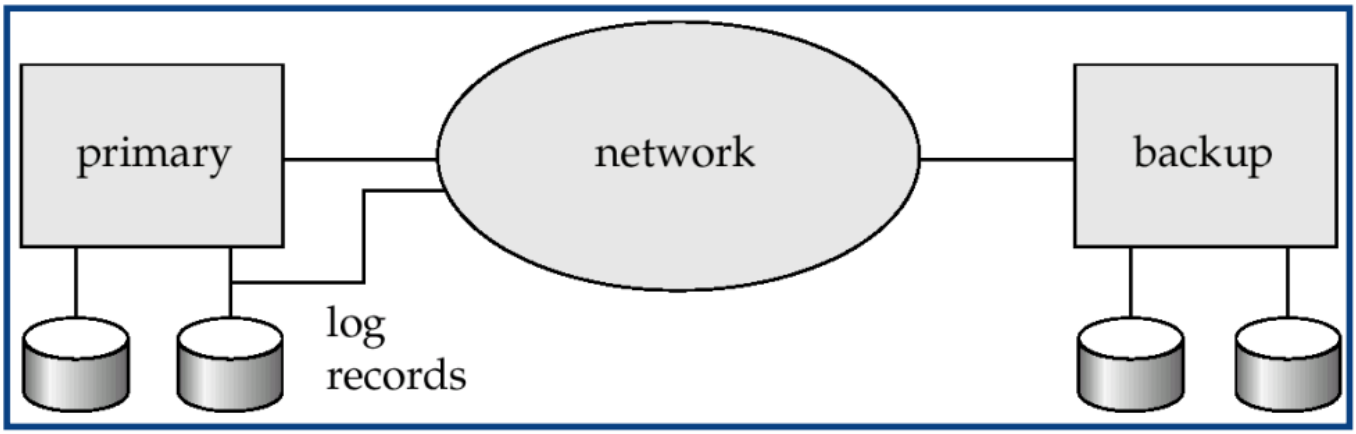
Summary, Cont.

- Checkpointing: A quick way to limit the amount of log to scan on recovery.
- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint.
  - Redo: Forward from oldest recLSN.
  - Undo: Backward from end to first LSN of oldest Xact alive at crash.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!

Backups

Remote backups

Remote backup systems provide high **availability** by allowing transaction processing to continue even if the primary site is destroyed. backup确保高可用性和**durability**



初始数据库(primary)会把data/update of data和log一起发给backup database.

Detection of failure:

- Backup site must detect when primary site has failed备份数据库必须要探测到原数据库fail
- To distinguish primary site failure from link failure, maintain several communication links between the primary and the remote backup通过建立多重连接来确保不是通信的故障而是源数据库故障
- Use heart-beat messages

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。backup数据库通过获得的源数据库的log和data perform recovery成为新的源数据库，并且开始重复之前源的任务

– Thus, completed transactions are redone and incomplete transactions are rolled back

• When the backup site takes over processing it becomes the new primary

Time to recover:

为了减少delay，一段时间就用log备份，有自己的ckpt

- To **reduce delay** in takeover, backup site **periodically processes the redo log records**
- In effect, it performs a **checkpoint**, and can then delete earlier parts of the log

Hot-Spare configuration

permits very fast takeover: log一来就做

• 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

To ensure durability of updates - delay transaction commit until update is logged at backup

But we can avoid this delay by permitting lower degrees of durability

使用backup来确保高durability

源数据库delay commit直到backup数据库也收到log(最大程度确保二者同步)

但是会造成Xact的delay

基于实际应用，有三种方式(源数据库如何commit)：

One-safe:

源数据库一旦写下commit log就commit xact

- 源数据库xact不会delay,但是backup数据库会lost update

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:

源数据库commit xact,一旦backup数据库收到commit log

- pros:二者一起commit,没有东西丢失，更加durability
- cons:需要等待，availability下降,一个没有成果commit,另一个也会失败。

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:

介于前两者之间，当都正常时用two-very-safe，只有一个正常用one-safe

proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary

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

Shadow paging

Shadow paging is an alternative to log-based recovery

基于日志的恢复的一种替代方案

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

在事务的生命周期中维护两个page——**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**存储在非易失性存储中，以便可以恢复事务执行之前的数据库状态

• Shadow page table is never modified during execution在执行期间永远不会被修改

To start with, both the page tables are identical. Only the current page table is used for data item accesses during execution of the transaction首先，两个页表是相同的。在事务执行期间，只有**current page table**用于数据项访问

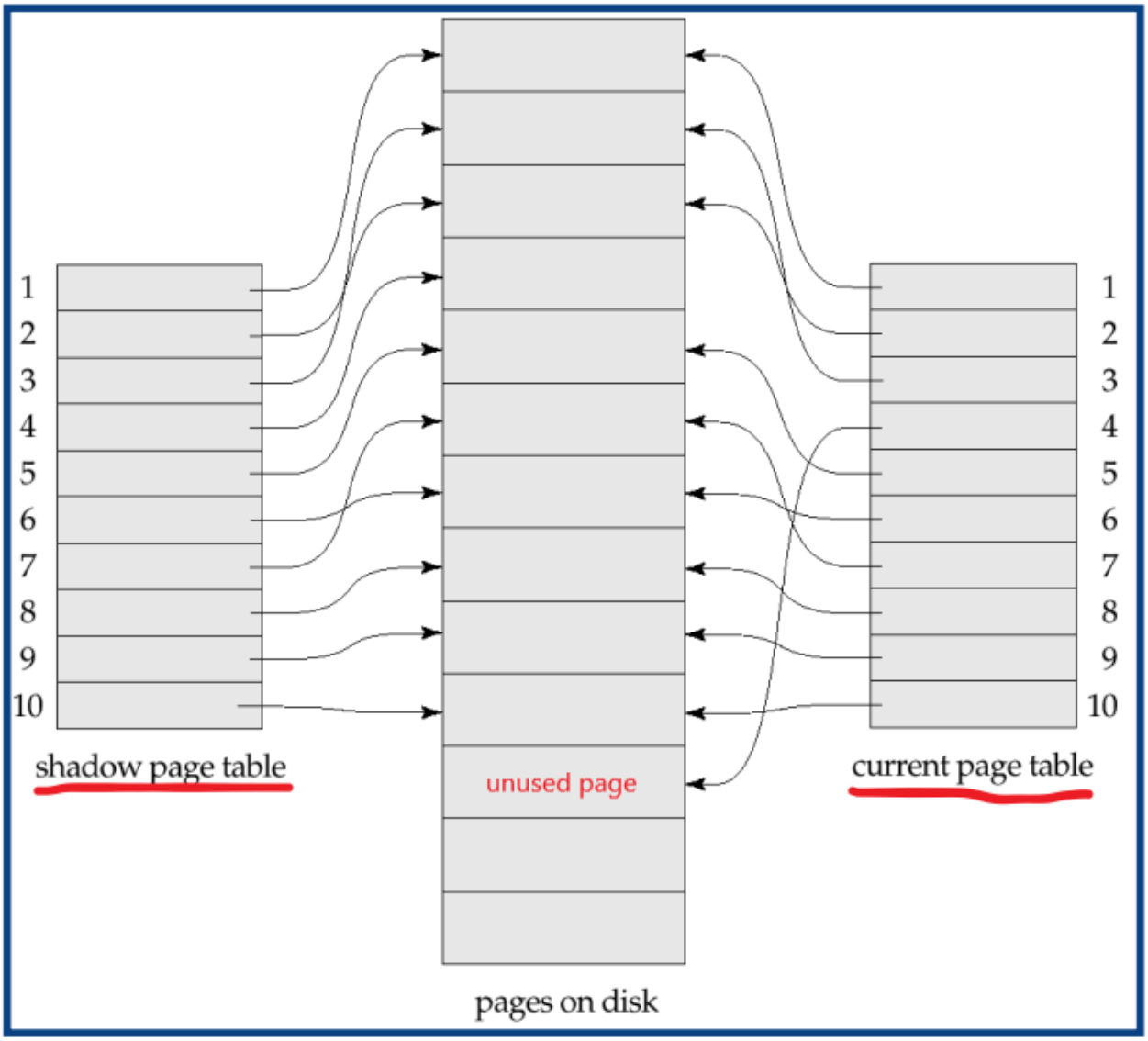
Whenever any page is about to be written:当任何一页要被写入时

• 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**

Example of Shadow Paging



Shadow and current page tables after write to page 4

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.

Advantages

(compare with log)

• No overhead of writing log records

• Recovery is trivial

Disadvantages:

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

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

• **Commit overhead is high** - flush every updated page, and page table

• Data gets fragmented (related pages get separated on disk)

• **Hard to extend** algorithm to allow transactions to run concurrently

Backup design strategy

Strategy plan based on:

• Goals and requirement of your organization/task

• The nature of your data and usage pattern

• Constraint on resources

Design backup strategy:

• Full disk backup vs partial - Are changes likely to occur in only a small part of the database or in a large part of the database?

• How frequently data changes

– If frequent: use differential backup that captures only the changes since the last full database backup

• Space requirement of the backups – depends on the resource

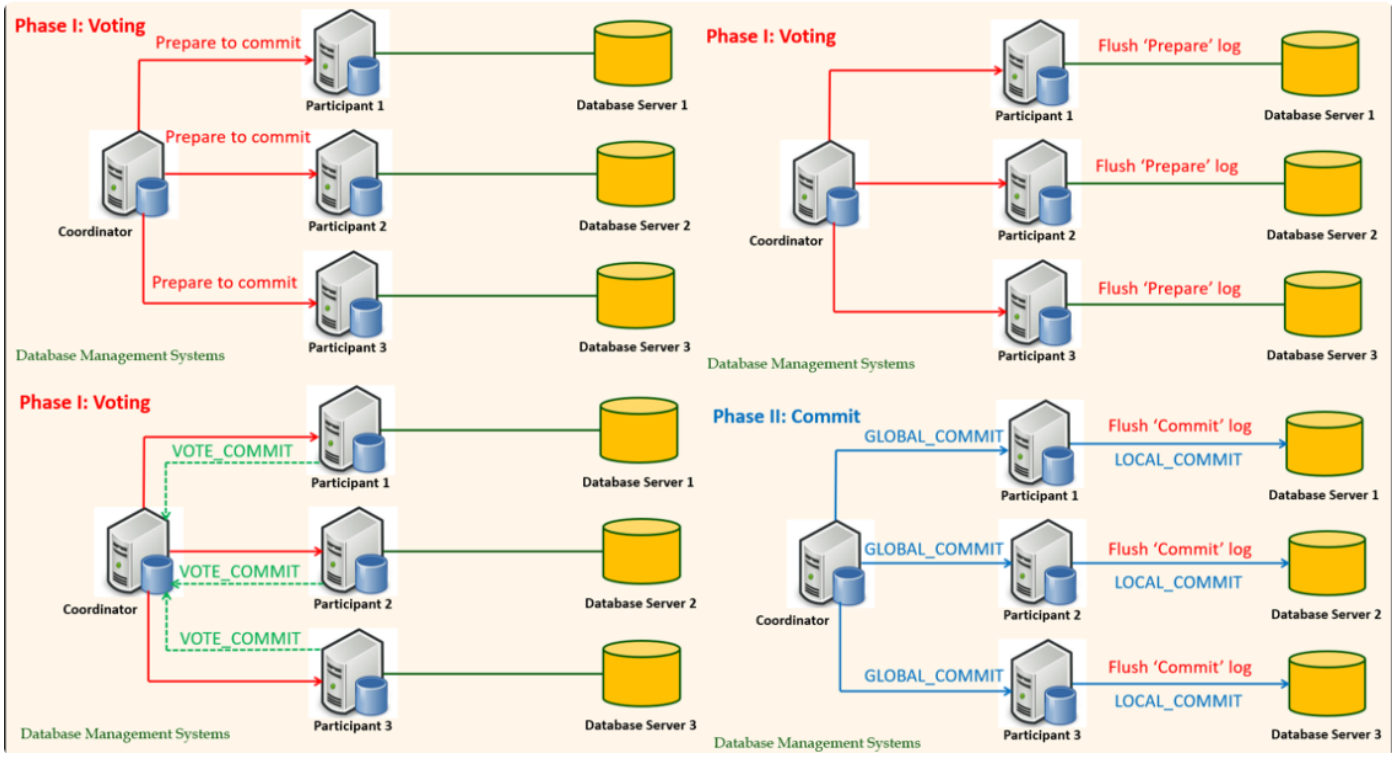
• Multiple past instances of backup – useful if point-in-time recovery is needed

Distributed databases

Atomicity in distributed DB

Two phase commit protocol

The two-phase commit protocol (2PC) can help achieve atomicity in distributed transaction processing



Abort transaction

Coordinator or participant can abort transaction

- If a participant abort, it must inform coordinator
- If a **participant** does not respond within a **timeout** period, **coordinator** will **abort**

If abort, coordinator asks **all participates** to **rollback**

If abort, abort logs are forced to **disk** at **coordinator** and all **participants**

Concurrency Control

• Each server is responsible for applying concurrency control to its own objects

每个服务器负责对自己的对象应用并发控制

• The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner

分布式事务服务器集合的成员共同负责确保它们以串行等效的方式执行

• BUT servers independently acting would not work

但服务器独立运行将无法工作

• If transaction T is before transaction U in their conflicting access to objects at one of the servers then:

如果事务T在事务U之前，在其中一个服务器上对对象的冲突访问中，则:

– They must be in that order at all of the servers whose objects are accessed in a conflicting manner by both T and U

在所有的服务器上，如果对象被T和U以冲突的方式访问，它们必须按照这个顺序

• The central **Coordinator** should assure this 应确保这一点

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.

本地锁管理器在知道事务涉及的**所有服务器都**已提交或中止事务之前，不能释放任何锁。

The objects remain locked and are unavailable for other transactions during the commit protocol.

在提交协议期间，对象保持锁定状态，对其他事务不可用。

- An aborted transaction releases its locks after phase 1 of the protocol. 终止的事务在协议的第一阶段后释放锁。

Timestamp ordering

The coordinator accessed by a transaction issues a globally unique timestamp

事务访问的协调器发出全局唯一的时间戳

The timestamp is passed with each object access

时间戳在每次对象访问时传递

The servers are jointly responsible for ensuring serial equivalence:

服务器共同负责确保串行等效:

- hat is if T access an object before U, then T is before U at all objects
- 也就是说，如果T访问一个在U之前的对象，那么T在所有对象中都在U之前

Optimistic

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

要使分布式事务工作:

1)验证在每个服务器的2PC协议的第一阶段进行

2)事务使用全局唯一的顺序进行验证

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

事务在复制对象上的效果应该与在一组对象上一次执行一个事务的效果相同

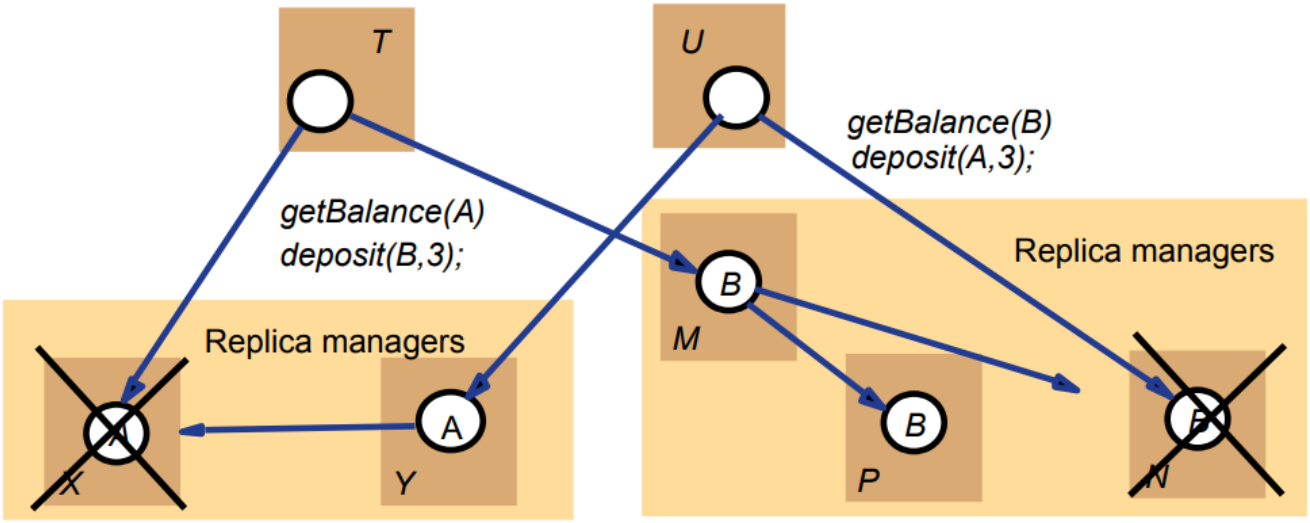
• this property is called **one-copy serializability**

此属性称为单副本序列化性

• If all servers are available then no issue – but what if some servers are not available?

如果所有服务器都可用，那么没有问题-但如果一些服务器不可用怎么办?

Before a transaction commits, it checks for failed and available servers it has contacted, the set should not change during execution:在事务提交之前，它检查它所联系的失败服务器和可用服务器，该设置在执行期间不应更改:



– E.g., T would check if X is still available among others.

例如，T将检查X是否仍然可用

– We said X fails before T's deposit, in which case, T would have to abort.

我们说X在T存款前失败，在这种情况下，T必须中止。

– Thus no harm can come from this execution now.

因此现在这个执行不会造成伤害。

CAP theorem

The **limitations** of distributed databases can be described in the so called the CAP theorem

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

一致性，可用性，可分割性

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

Trade off 权衡

Availability + Partition Tolerance forfeit Consistency as changes in place cannot be propagated when the system is portioned.(单节点上改变不能保证所有节点上一致)

Consistency + Partition Tolerance entails that one side of the partition must act as if it is unavailable, thus forfeiting Availability(所有节点上一致导致有时候数据短时间内不能获取到)

Consistency + Availability is only possible if there is no network partition, thereby forfeiting Partition Tolerance(没有节点分区以保证一致性和可获取)

Types of consistency

Strong Consistency(统一返回更新的值)

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

Weak Consistency(不一定返回更新的值)

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

Eventual Consistency(如果没有额外的更新，最终会返回更新的值)

- 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

Eventual Consistency Variations

Causal consistency

Processes that have causal relationship will see consistent data(如果Process A通知Process B它已经更新了数据，那么Process B的后续读取操作则读取A写入的最新值，而与A没有因果关系的C则可以最终一致性)

Read-your-write consistency

- A process always accesses the data item after it’s update operation and never sees an older value(process对某个值更新后就看不到旧的)

Session consistency

- As long as session exists, system guarantees read-your-write consistency
- Guarantees do not overlap sessions(单个session内的consistency保证)

Monotonic read consistency

- If a process has seen a particular value of data item, any subsequent processes will never return any previous values(看到了新的就看不到旧的)

Monotonic write consistency

- The system guarantees to serialize the writes by the same process(保证系统会序列化执行一个Process中的所有写操作)

实践中会结合不同的consistency

Dynamic trade-off between consistency and availability

多用户: 使用Eventual Consistency，减少load,增加availability

ATM: A>C

Airline订票:当飞机很空时A>C确保可用性，当飞机快满时使用C>A确保一致性，不会有很多人同时订上最后一个座位

Data partitioning for trade-off

Different data may require different consistency and availability

Heterogeneity: Segmenting C and A(不同的部分侧重的方向不同，不追求整体)

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

Data Partitioning

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**

What if there are no partitions?

Tradeoff between Consistency and Latency:

• Caused by the possibility of failure in distributed systems

– High availability -> replicate data -> consistency problem

• Basic idea:

– Availability and latency are arguably the same thing: unavailable -> extreme high latency

– Achieving different levels of consistency/availability takes different amount of time

Maintaining consistency should balance between the strictness of consistency versus availability/scalability

BASE properties

CAP can not granted at the same time result in databases with **relaxed ACID guarantees**

In particular, such databases apply the BASE properties:

Basically Available: the system guarantees Availability

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

Eventual Consistency: the system will eventually become consistent

PACLE

If there is a **partition,** how does the system trade off **availability** and **consistency** (A and C); else (E),

when the system is running normally in the absence of **partitions**, how does the system trade off **latency** (L) and **consistency**(C)?

Example

PA/EL Systems: Give up both Cs for availability and lower latency

- Dynamo, Cassandra, Riak

PC/EC Systems: Refuse to give up consistency and pay the cost of availability and latency

- BigTable, Hbase, VoltDB/H-Store

PA/EC Systems: Give up consistency when a partition happens and keep consistency in normal operations

- MongoDB

PC/EL System: Keep consistency if a partition occurs but gives up consistency for latency in normal operations

- Yahoo! PNUTS

NoSQL and BASE

NoSQL (or Not-Only-SQL) databases follow the BASE properties:

- Basically Available
- Soft-State
- Eventual Consistency

NoSQL

- **Document Stores**(Mongodb, CouchDB)
  - Documents are stored in some standard format or encoding
  - Documents can be indexed
- **Graph database**(Neo4j)
  - Data are represented as vertices and edges
  - Graph databases are powerful for graph-like queries
- **Key-value database**(Amazon DynamoDB and Apache Cassandra)
  - Keys are mapped to (possibly) more complex value
  - 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(没办法join和aggregate)
- **Columnar database**(HBase and Vertica)
  - 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
  - Values are queried by matching keys

Data warehousing

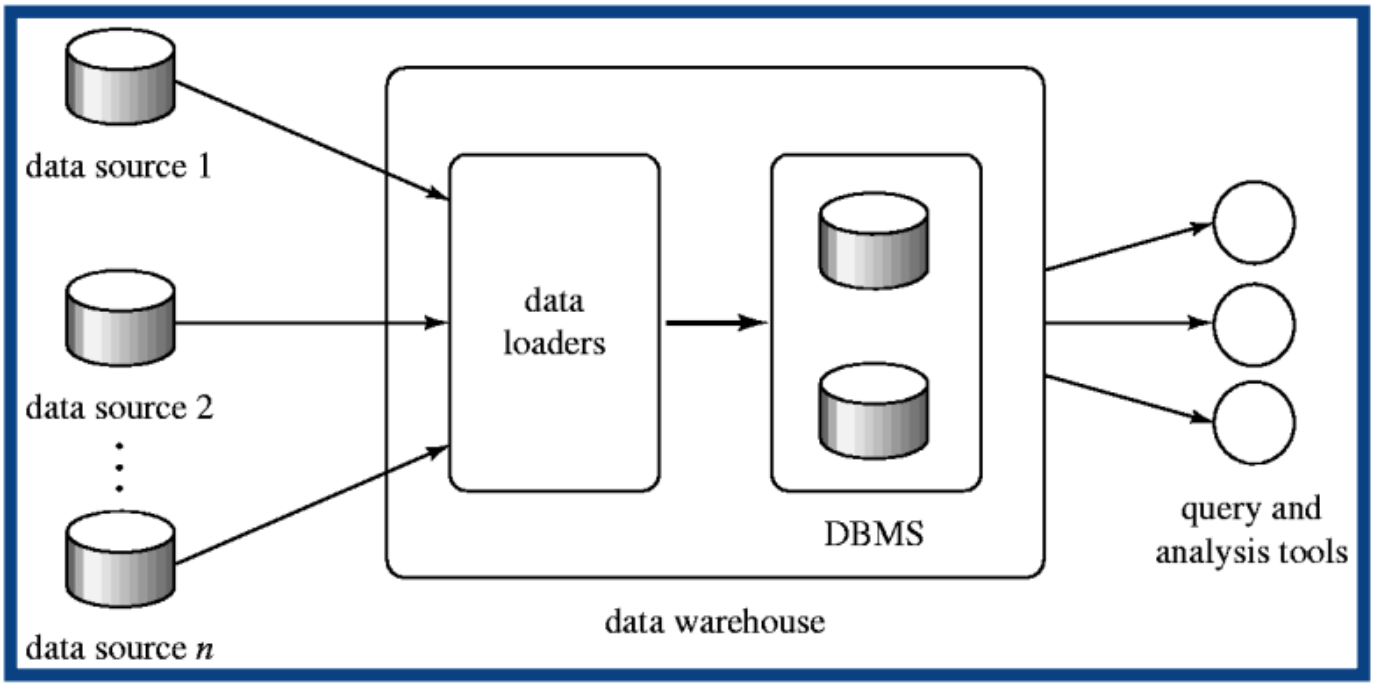
Corporate decision making requires a unified view of all organizational data, including **historical data**

公司的决策制定需要所有组织数据(包括历史数据)的统一视图

A data warehouse is a repository (archive) of information gathered from **multiple sources**, stored under a **unified schema**, for analytics and reporting purposes

数据仓库是从多个来源收集的信息的存储库(存档)，存储在统一的模式下，用于分析和报告目的

- Greatly simplifies querying, permits study of historical trends 极大地简化了查询，允许研究历史趋势
- Shifts decision support query load away from transaction processing systems 从关注处理transaction转移到执行query



*multisource

When and how to gather data:

• **Source driven architecture**: data sources transmit new information to warehouse, either continuously or periodically (e.g. at night)数据库发送

• **Destination driven architecture**: warehouse periodically requests new information from data sources 数仓请求

Keeping warehouse exactly synchronized with data sources (e.g., using two-phase commit) is too expensive

保持数仓与数据库同步很贵

– Usually OK to have **slightly out-of-date** data at warehouse

– Data/updates are **periodically** downloaded form online transaction processing (OLTP) systems (most of the DBMS work we have seen so far)

What schema to use

• Depends on purpose

• 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

• The data stored in a data warehouse is **documented** with an element of **time**, either explicitly or implicitly

What data to summarize

类似couchDB写view,需要等待生成。

• 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