CSCU9B3

Database Principles and Applications

Distribution,

Backup and concurrency,

recovery

Topic Overview

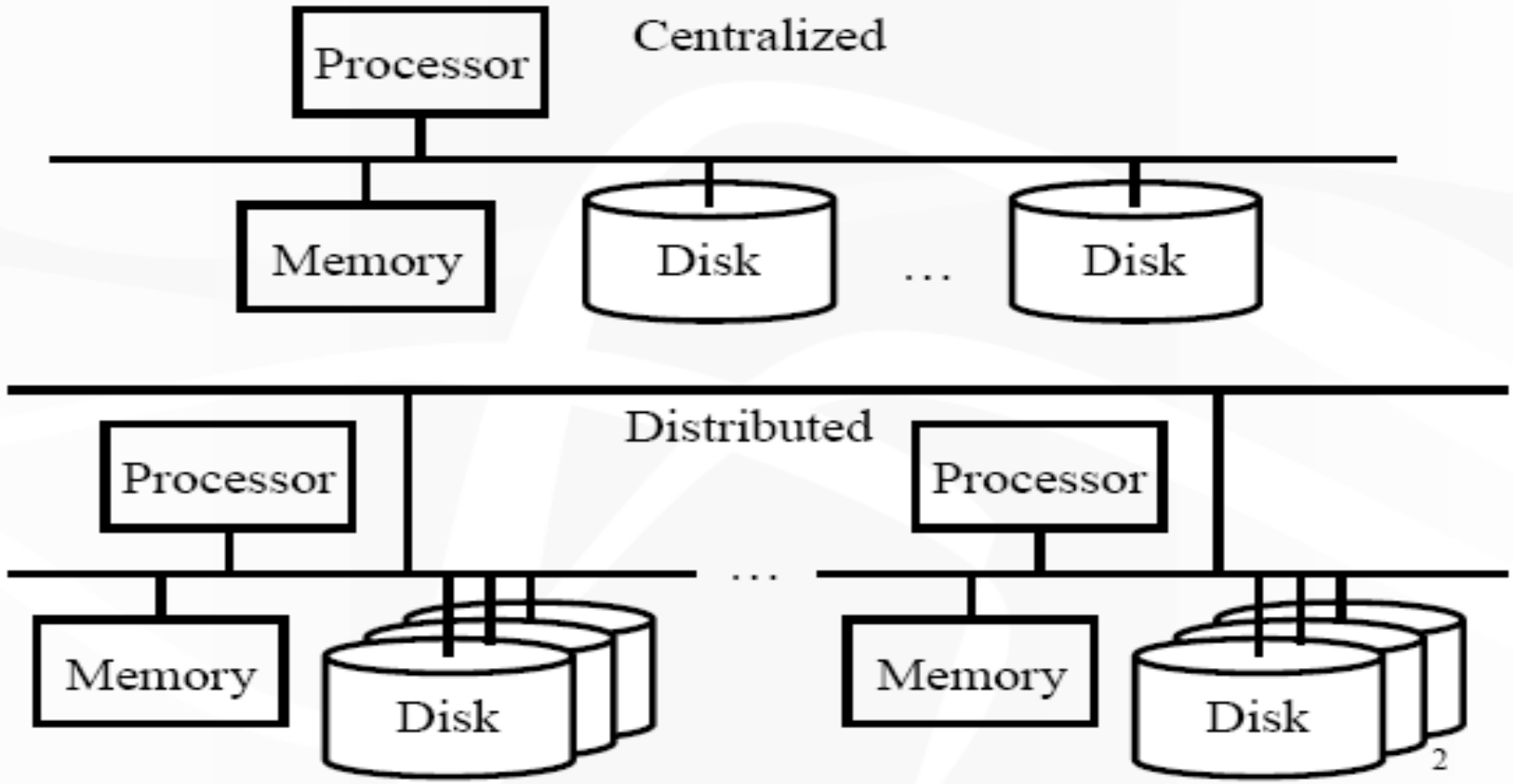
- A DBMS must ensure that the database is **reliable** and remains in a **consistent** state.
- This reliability and consistency must be maintained in the presence of failures of both hardware and software components.
- The DBMS must also cope with **multiple users** accessing the database, possibly in different locations.
- There are four main issues that we will examine:
  - **Distribution** of a database across the network
  - the concept of a **transaction**
  - **Concurrency** control
  - **Recovery**

Distributed Databases

- **Distributed Database**
  - A logically interrelated collection of shared data (and a description of this data), physically distributed over a computer network.
  - Could be across two computers in the same room or lots of computers across the world
  - Under the control of more than one CPU

- **Distributed DBMS**
  - Software system that permits the management of the distributed database and makes the distribution transparent to users.

Centralized versus distributed DBMS



Key Concepts

- Collection of logically-related shared data.
- Data split into fragments.
- Fragments may be replicated.
- Fragments/replicas allocated to sites.
- Sites linked by a communications network.
- Data at each site is under control of a DBMS.
- DBMSs handle local applications autonomously.
- Each DBMS participates in at least one global application.

Advantages of DDBMS

- Reflects organizational structure — database fragments are located in the departments they relate to.
- Local autonomy — a department can control the data about them (as they are the ones familiar with it.)
- Improved availability — a fault in one database system will only affect one fragment, instead of the entire database.
- Improved performance — data is located near the site of greatest demand, and the database systems themselves are parallelized, allowing load on the databases to be balanced among servers. (A high load on one module of the database won't affect other modules of the database in a distributed database.)
- Economics — it costs less to create a network of smaller computers with the power of a single large computer.
- Modularity — systems can be modified, added and removed from the distributed database without affecting other modules (systems).

Disadvantages of DDBMS

- Complexity — extra work must be done by the DBAs to ensure that the distributed nature of the system is transparent. Extra work must also be done to maintain multiple disparate systems, instead of one big one. Extra database design work must also be done to account for the disconnected nature of the database — for example, joins become prohibitively expensive when performed across multiple systems.
- Economics — increased complexity and a more extensive infrastructure means extra labour costs.
- Security — remote database fragments must be secured, and they are not centralized so the remote sites must be secured as well. The infrastructure must also be secured (e.g., by encrypting the network links between remote sites).
- Difficult to maintain integrity — in a distributed database, enforcing integrity over a network may require too much of the network's resources to be feasible.
- Inexperience — distributed databases are difficult to work with, and as a young field there is not much readily available experience on proper practice.

DDBMS

- DDBMS is a Distributed Database Management System with
  - Extended communication services.
  - Extended Data Dictionary.
  - Distributed query processing.
  - Extended concurrency control.
  - Extended recovery services.
- Plus all the functionality you would expect from a centralized DBMS

Two Types of DDB

- **Homogeneous DDBMS**
  - All sites use same DBMS product.
  - Much easier to design and manage.
  - Approach provides incremental growth and allows increased performance.
- **Heterogeneous DDBMS**
  - Sites may run different DBMS products, with possibly different underlying data models.
  - Occurs when sites have implemented their own databases and integration is considered later.
  - Translations required to allow for:
    - Different hardware.
    - Different DBMS products.
    - Different hardware and different DBMS products.
  - Typical solution is to use gateways.

Which Type and Why?

- **Homogeneous DDBMS**
  - Databases that are homogeneous were probably designed to be distributed from the start
  - The designers had the luxury of being able to choose a common DBMS
  - Foreign keys allow tables in one database to be linked to tables in another
  - The physical distribution is probably for security and service protection.
  - Far fewer problems to solve when building and maintaining such a database
  - This type of design is known as **top down**.

Which Type and Why?

- **Heterogeneous DDBMS**
  - The different databases probably already exist and are in use by different users
  - It becomes desirable to join them into one apparently single database, without simply throwing them all away and starting again
  - They could each have a different DBMS
  - They will be more than one DD, DA and DBA
  - They may not be an easy way of matching fields from one to equivalent fields in another
  - Joining such databases is known as a **Bottom Up** join

Example

- A bank has a database of current account users, another with mortgage holders and a third with customer relations data.
- If the marketing department want to see what services a customer has bought, and when they last wrote to them, they have to look the customer up in three different databases.
- The databases can become out of synch, for example if a person moves house and tells the bank once. The mortgage account DB might get updated, but the current account DB might not, and the customer relations DB almost certainly will not.
- By joining the databases, the bank can have a ‘single customer view’, which is something they like.

Joining a DDBMS Bottom Up

- The job of the DDBMS is to sit above the several separate databases and provide the user with a single view as if there were really only one database
- It does not integrate the databases into one new single database. They stay where they are.
- One method of achieving this is to use a gateway, which translates from one DBMS to another
- There are also attempts being made to develop open specifications that will allow one DBMS to communicate with another without the need for a gateway.

- A much more difficult problem is joining the data from more than one DB, for example, the current account DB has William Smith, but marketing communicate with Bill Smith.

Joining a DDBMS Bottom Up

- Bottom up joins have many difficulties because the databases were not designed to be joined.
- Each will have its own DBA, so there needs to be human integration too
- The purpose for which a DB was designed may not match the reasons that it is needed when it has been joined

- E.g. city council and social services

Partitioning Schemes

- Any part of a distributed database is called a fragment
- For homogeneous DDBs, the most common forms of partitioning are horizontal and vertical:

- Horizontal partitioning is where different full records from a table are stored in different locations.
- Vertical partitioning is where different attributes are stored in different locations.

Partitioning Schemes

- Example (horizontal):

Imagine the car dealership chain, Arnold Clark. They will use a dealer management system such as Kerridge. The tables in each dealership will have the same structure (homogeneous) but different customers will appear in each. If head office want to write to every customer, regardless of where they bought the car, they need the DDBMS to be able to respond to a query that generates every customer across all the databases in each dealership.

- Example (vertical):

I probably appear on a few databases in the university: payroll, car parking, timetabling, sports centre …

If the university wanted to be able to make a single query and see everything they have about me, it would be from a number of different databases. Chances are, they couldn't do it because the DBs are not under a DDBMS.

Backup and Concurrency

- Regardless of whether a database is distributed or not, loss and inconsistencies can still occur:
  - Many DBMSs allow users to undertake **simultaneous operations** on the database.
  - If these operations are not controlled, they can interfere with each other, and the database may become **inconsistent**.
- To overcome this the DBMS implements a concurrency control protocol.
- Database recovery is the process of restoring the database to a correct state following a failure
  - The failure may be the result of a system crash, media failure, software error, or accidental or malicious destruction of data.
- Whatever the cause of failure, the DBMS must be able to restore the database to a consistent state.

Transaction Management

- Both **concurrency** and **recovery** control are required to protect the database from data inconsistencies and data loss.
- Many DBMSs allow users to undertake **simultaneous operations** on the database.
- If these operations are not controlled, they can interfere with each other, and the database may become **inconsistent**.
- To overcome this the DBMS implements a concurrency control protocol.
- Database recovery is the process of restoring the database to a correct state following a failure
  - The failure may be the result of a system crash, media failure, software error, or accidental or malicious destruction of data.
- Whatever the cause of failure, the DBMS must be able to restore the database to a consistent state.

The Concept of a Transaction

- The idea of a **transaction** is central to both concurrency control and recovery.
- A **transaction** is an action or series of actions, carried out by a single user or application program, which accesses or changes the content of the database.
- So, a transaction is a **logical unit** of work on the database.
  - It may involve one or more operations on the database
  - It could be as simple as a single SQL command, or as complex as the set of accesses performed by an application program.
- We shall now look at some examples

Transaction Examples

- Consider two tables from the Estate Agent database:
  - Staff (StaffNo, Name, Address, TelNo, Position, DOB, Salary, BranchNo)
  - Property (PropertyNo, Address, Type, Rooms, Rent, OwnerNo, StaffNo)
- A simple transaction is to update the Salary of a particular staff member (perhaps to add a 10% raise).
- We could represent the operations of this transaction as follows:

```
read(StaffNo = x, Salary)
Salary = Salary * 1.1
write(Staffno = x, Salary)
```

x is the StaffNo of the Staff member we wish to update

- Here our transaction consists of two database accesses (read and write), and a non-database activity (updating the salary).

Transaction Examples

- A more complicated transaction is to delete a particular member of staff.
- To do this, we must delete the correct row from the staff table, and then reassign all the property records managed by that member of staff.
- We can represent this complex transaction as follows:

```
delete (Staffno = x)                          ! Delete staff member x
for all Property records, p
    read(PropertyNo = p, StaffNo)
    if (StaffNo = x) then                     ! Is property managed by x?
      StaffNo = NewStaffNo                    ! Assign the property to NewStaffNo
      write(PropertyNo=p, StaffNo)
    endif
endfor
```

Transactions and consistency

- Notice that **all** of the updates in the above transaction must take place to ensure that the database remains consistent.
  - Otherwise, some properties will be managed by a staff member who no longer exists in the database (violating **referential integrity**).
- A transaction should always transform the database from one consistent state to another.
  - Note that consistency may be violated while the transaction is in progress.
  - For example, while a staff member is being deleted, some properties will be managed by the deleted member of staff, while others will have been reassigned.
  - However, at the **end** of the transaction, the database is consistent.

Transaction Outcomes

- A transaction can have one of two outcomes:
- If it completes successfully, the transaction is said to have **committed** and the database reaches a new consistent state.
- If it does not execute successfully, the transaction is **aborted**.
  - In this case, the database must be restored to the consistent state it was in before the transaction started.
  - This is known as **roll-back**.
- Whatever the outcome, the database is in a consistent state at the end of the transaction.

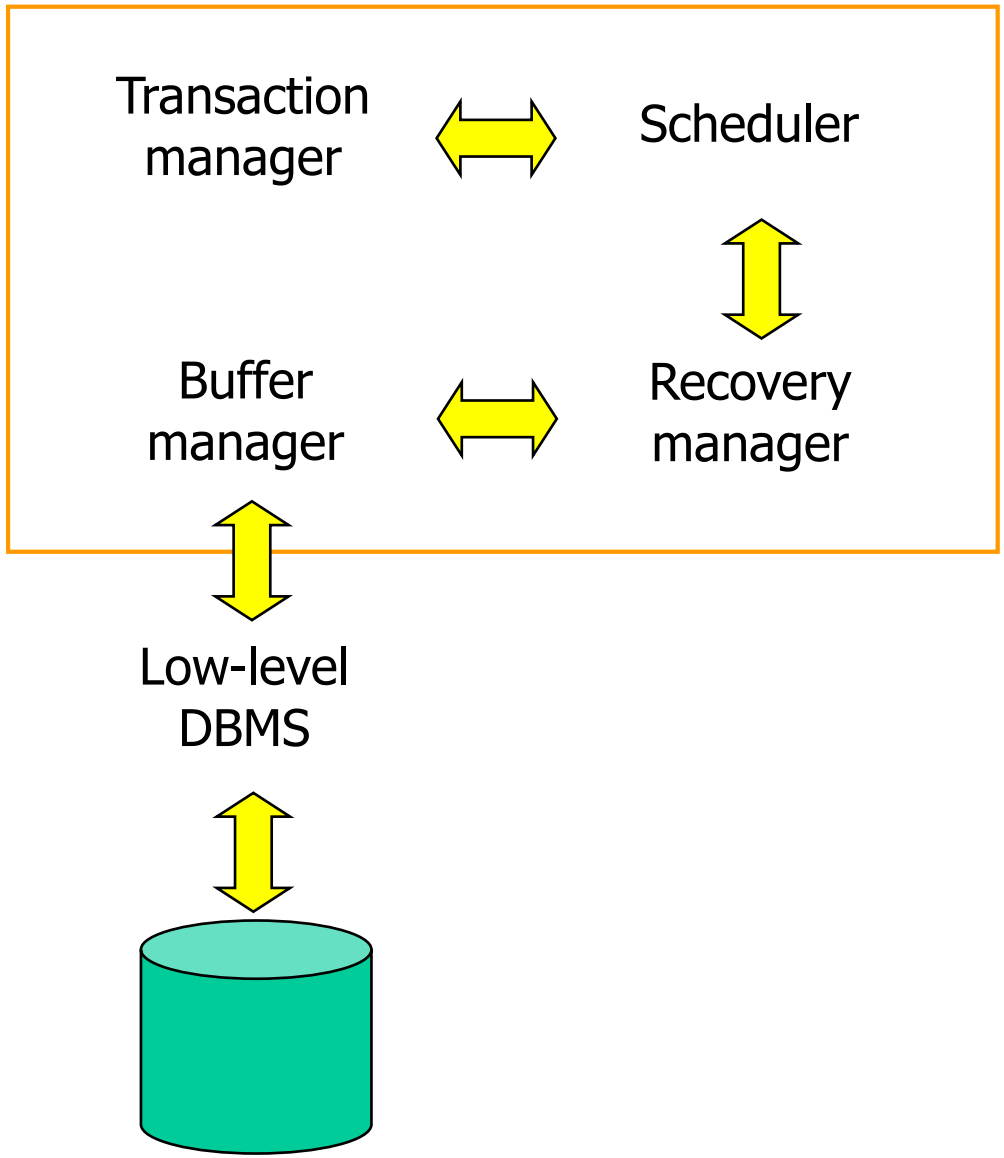
Transaction Support

- Once a transaction has committed, it cannot be aborted.
  - Thus, if we decide that a committed transaction was a mistake, then we must perform another transaction to reverse it.
- On the other hand, an aborted transaction can be restarted later, and depending on the cause of failure, may successfully execute and commit at that time.
- A DBMS has no way of knowing which updates are grouped together to form a single, logical transaction.
- Therefore, the user must be provided with a way to indicate the boundaries of each transaction.
  - For example, there may be keywords such as BEGIN_TRANSACTION, COMMIT, and ROLLBACK to delimit a transaction.
  - If such delimiters are not used, the whole program is usually treated as a single transaction with the DBMS automatically performing a COMMIT upon successful termination, or ROLLBACK if not.

Properties of Transactions

- There are certain properties that all transactions should possess.
- The four basic (so-called **ACID**) properties are:
- **Atomicity** (the “all or nothing” property)
  - A transaction is an indivisible unit that is either performed in its entirety, or not performed at all
- **Consistency**
  - A transaction transforms the database from one consistent state to another.
- **Independence**
  - Transactions execute independently of one another. In other words, the partial effects of an incomplete transaction should not be visible to other transactions.
- **Durability**
  - The effects of a committed transaction are permanent and must not be lost because of subsequent failure.

Database Architecture



Database Architecture

- The components of a DBMS that manage transactions are as follows:
- The **transaction manager** coordinates transactions on behalf of application programs.
- It communicates with the **scheduler**, which implements a particular strategy for concurrency control.
  - The scheduler tries to maximise concurrency without allowing transactions to interfere with one another.
- If failure occurs during a transaction, the **recovery manager** ensures that the database is restored to the state it was in before the start of the transaction.
- The **buffer manager** is responsible for the transfer of data between disk storage and main memory.

- Next, we look at concurrency control in more detail…

Concurrency Control

- **Concurrency control** is the process of managing simultaneous operations on the database without having them interfere with one another.
- Concurrency control is needed because many users are able to access the database simultaneously.
- Note that managing concurrent access is easy if all users are only **reading** data.
  - There is no way that such uses can interfere with one another.
- However, when two or more users are accessing the database simultaneously, and at least one of them is **updating** data, there may be interference that can cause inconsistencies.
- We will now look at some examples of the inconsistencies that may arise in the absence of concurrency control.

The Lost Update Problem

- Imagine that a customer wants to withdraw £30 from a bank account.
- At the same time, the bank is crediting this month's salary.

Time	T1 (withdrawal)	T2 (credit salary)	Balance
t1		begin_transaction	100
t2	begin_transaction	read(balance)	100
t3	read(balance)	balance=balance+1000	100
t4	balance=balance-30	write(balance)	1100
t5	write(balance)	commit	70
t6	commit		70

- Both transactions occur at roughly the same time and read the same initial balance.
  - The last transaction to commit **overwrites** the update made by the first.

The Uncommitted Dependency Problem

- The uncommitted dependency problem occurs when one transaction sees the intermediate results of another (aborted) transaction.

Time	T1 (withdrawal)	T2 (credit salary)	Balance
t1	begin_transaction		100
t2	read(balance)		100
t3	balance=balance-30		100
t4	write(balance)	begin_transaction	70
t5		read(balance)	70
t6	rollback	balance=balance+1000	70
t7		write(balance)	1070
t8		commit	1070

- For some reason, the withdrawal transaction is aborted, but the salary credit transaction has already seen the update.
- When T2 commits, the balance is incorrect (it should be 1100).

The Inconsistent Analysis Problem

- The previous problems involved simultaneous updates to the database. However, problems can also result if a transaction merely reads the result of an uncommitted transaction.
- Below, one transaction (T1) is transferring £10 from account Balw to Balz, and at the same time, another transaction (T2) is summing all the accounts (Balw, Balx, Baly and Balz). Try to figure out (AS A HOMEWORK EXERCISE!) what's gone wrong….

Time	T1 (transfer funds)	T2 (sum accounts)	Balw	Balx	Baly	Balz	Sum
t1		begin_transaction	100	50	10	25	
t2	begin_transaction	sum=0	100	50	10	25	0
t3	read(Balw)	read(Balw)	100	50	10	25	0
t4	balw=Balw - 10	sum = sum + Balw	100	50	10	25	100
t5	write(Balw)	read(Balx)	90	50	10	25	100
t6	read(Balz)	sum = sum + Balx	90	50	10	25	150
t7	balz = Balz + 10	read(Baly)	90	50	10	25	150
t8	write(Balz)	sum = sum + Baly	90	50	10	35	160
t9	commit	read(balz)	90	50	10	35	160
t10		sum = sum+Balz	90	50	10	35	195
t11		commit	90	50	10	35	195

Concurrency Control Techniques

- To prevent problems such as those we have just seen, a DBMS must implement some sort of concurrency control protocol.
- A commonly used technique is **locking**.
- The locking technique operates by preventing a transaction from improperly accessing data that is being used by another transaction.
- Before a transaction can perform a read or write operation, it must claim a **read** (shared) or **write** (exclusive) lock on the relevant data item.
- Once a **read lock** has been granted on a particular data item, other transactions may read the data, but not update it.
- A **write lock** prevents all access to the data by other transactions.

Lock semantics

- Since read operations cannot conflict, it is acceptable for more than one transaction to hold read locks simultaneously on the same item.
- On the other hand, a write lock gives a transaction exclusive access to the data item.
- Locks are used in the following way:
  - A transaction needing access to a data item must first lock the item, requesting a read lock for read-only access or a write-lock for read-write access.
  - If the item is not already locked by another transaction, the lock request will be granted.
  - If the item is currently locked, the DBMS determines whether the request is compatible with the current lock. If a read lock is requested on an item that is already read locked, the request is granted, otherwise the transaction must wait until the existing write lock is released.
  - A transaction holds a lock until it explicitly releases it, commits, or aborts.

Two-Phase Locking (2PL)

- The use of locks must occur in a controlled way to prevent any of the problems we have seen from occurring.
- Many database systems employ a **two-phase locking** protocol to control the manner in which locks are acquired and released.
- Each transaction is divided into two phases:
  - a **growing phase**, in which it acquires all the locks needed, but cannot release any locks;
  - a **shrinking phase**, in which it releases its locks but cannot acquire any new locks.
- The rules of the protocol are as follows:
  - A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
  - Once the transaction releases a lock, it can never acquire any new locks.

Solving the Lost Update Problem with 2PL

- In this solution, we have introduced two-phase locking.
- Now, the withdrawal transaction is forced to wait until the salary credit transaction releases its lock.

Time	T1 (withdrawal)	T2 (credit salary)	Balance
t1		begin_transaction	100
t2	begin_transaction	write_lock(balance)	100
t3	write_lock(balance)	read(balance)	100
t4	WAIT	balance=balance+1000	100
t5	WAIT	write(balance)	1100
t6	WAIT	unlock(balance)	1100
t7	read(balance)	commit	1100
t8	balance=balance-30		1100
t9	write(balance)		1070
t10	unlock(balance)		1070
t11	commit		1070

Problems with Locking - I

- The use of locks and the 2PL protocol prevents many of the problems arising from concurrent access to the database.
- However, it does not solve all problems, and it can even introduce new ones.
- Firstly, there is the issue of **cascading rollbacks**:
  - 2PL allows locks to be released before the final commit or rollback of a transaction.
  - During this time, another transaction may acquire the locks released by the first transaction, and operate on the results of the first transaction.
  - If the first transaction subsequently aborts, the second transaction must abort since it has used data now being rolled back by the first transaction.
  - This problem can be avoided by preventing the release of locks until the final commit or abort action.

Problems with Locking - II

- Secondly, there is the problem of **deadlock**.
  - Deadlock occurs when two or more transactions reach an impasse because they are waiting to acquire locks held by each other.
- For example, consider these two transactions:

Time	T1	T2
t1	begin_transaction	
t2	write_lock(Balx)	begin_transaction
t3	read(Balx)	write_lock(Baly)
t4	balx = Balx - 10	read(Baly)
t5	write(Balx)	Baly = Baly+1000
t6	write_lock(Baly)	write(Baly)
t7	WAIT	Write_lock(Balx)
t8	WAIT	WAIT
t9	WAIT	WAIT
t10	…	…

Problems with Locking - III

- Deadlock is a serious problem that the DBMS must deal with.
- Once deadlock occurs, there is only one way to break it.
  - One or more of the deadlocked transactions must be aborted.
  - This involves undoing all the changes made by the transaction(s).
- For example, we might decide to abort T2.
  - In this case, the update to Baly is undone during rollback, and the write lock is released.
  - T1 may then proceed.
- Deadlock should be **transparent** to the user.
  - That is, a DBMS should automatically restart any transaction aborted to resolve deadlock.
- There are actually several different ways to deal with deadlock
  - outside scope of this course - see Textbook for more details

Recovery Control

- Database **recovery** is the process of restoring the database to a correct state in the event of failure.
- There are many different types of failure, each of which must be handled appropriately.
- Among the causes of failure are:
  - System crashes due to hardware or software errors.
  - Media failures.
  - Natural disasters such as fire, flood, etc.
  - Human error on the part of operators or users.
  - Sabotage
- Whatever the cause of failure, there are two main effects that must be considered:
  - The loss of main memory, including database buffers
  - The loss of the disk copy of the database.

Transactions and Recovery

- Transactions are the basic unit of recovery in a database system.
- The recovery manager must guarantee two of the four ACID properties of transactions in the presence of failure:
  - **Atomicity**
  - **Durability**
- The recovery manager must ensure that, on recovery from failure, either **all** of the effects of a transaction are permanently recorded in the database, or **none** of them are.
- The situation is complicated because the act of writing information to the database is not an **atomic** action.
- Therefore, it is possible for a transaction to have committed, but for its effects not to have been permanently recorded, simply because they have not yet reached the database.

Failure Example - I

- Consider the following example transaction from before:

  Read(Sno = x, Salary)

  Salary = Salary * 1.1

  Write(Sno = x, Salary)

- For the Read operation, the DBMS performs the following steps:
  - Find the address of the disk block containing the record with key value x.
  - Transfer this disk block into a database buffer in main memory.
  - Copy the salary data from the database buffer into the variable Salary.
- For the Write operation, the following steps occur:
  - Find the address of the disk block containing the record with key value x.
  - Transfer this disk block into a database buffer in main memory.
  - Copy the salary data from the variable Salary into the database buffer.
  - Write the database buffer back to disk.

Failure Example - II

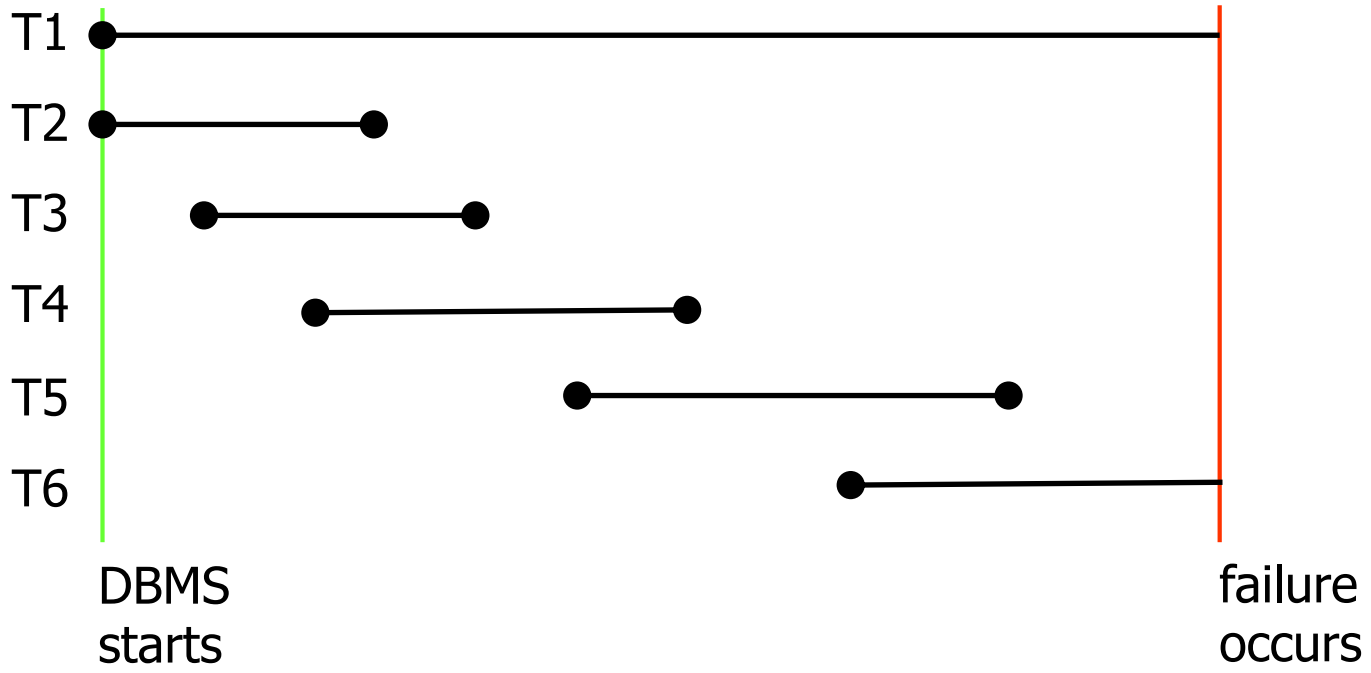
- The database buffers function as an intermediate storage location (or cache) to minimise the cost of reading and writing data on disk.
  - So, it is common to delay writing them to disk until absolutely necessary.
- It is only once the buffers have been **flushed** to disk that any update operations can be regarded as permanent.
- The flushing of buffers can be triggered by special commands (e.g., transaction commit), or automatically when the buffer becomes full.
  - Explicit flushing of the buffers is called **force-writing**.
- Failure may occur between writing to the buffers and flushing the buffers to disk.
  - In this case, the recovery manager must determine the status of the transaction that performed the write at the time of failure.

Failure Example - III

- If the transaction that wrote prior to failure had issued its commit, then the recovery manager must **redo** the transaction's updates to ensure **durability**.
  - This is also known as **roll-forward**.
- On the other hand, if the transaction had not committed at the time of failure, then the recovery manager must **undo** any effects of that transaction to ensure **atomicity**.
  - This is also known as **roll-back**.
- We will now consider another example to reinforce these ideas.

Use of REDO/UNDO

- Consider a number of concurrently executing transactions, T1…T6.



Use of UNDO/REDO

- Clearly, T1 and T6 had not committed at the time of the crash.
- Therefore, at restart, the recovery manager must undo all changes made by T1 and T6.
- However, it is not clear to what extent the changes made by the committed transactions have been propagated to the database on disk.
- There is no way to know whether or not the volatile database buffers were written (flushed) to disk before the crash occurred.
- In the absence of any other information, the recovery manager is forced to *redo* (roll-forward) the (committed) transactions T2…T5: Why?
  - To ensure durability!

– however, see more, later, on this (use of checkpointing)!

Recovery Facilities

- A DBMS should provide the following facilities to assist with recovery:
  - A **backup mechanism**, which makes periodic backup copies of the data.
  - **Logging facilities**, which keep track of the current state of transactions and database updates.
  - A **checkpoint facility**, which enables updates to the database which are in progress to be made permanent.
  - A **recovery manager**, which allows the system to restore the database to a consistent state following a failure.
- The **backup** mechanism should allow the database and the log file to be archived at regular intervals without having to stop the system.
- The backup archive copies can be used to recover from severe failures in which the storage media are damaged.

Logging Facilities

- To keep track of database transactions, the DBMS maintains a special file called a **log**.
- The log contains information about all updates to the database.
- The log primarily contains **transaction records**.
- A transaction record consists of:
  - A transaction identifier.
  - A record type (transaction start, insert, update, delete, abort, commit).
  - The identity of the affected data item (for insert, delete, and update).
  - **Before-image** of the data item (its value before an update or delete).
  - **After-image** of the data item (its value after an update or insert).
  - Log management information.

Example of Log Contents

- Below is a segment of a log file showing three concurrent transactions T1, T2, and T3.

Lid	Tid	Time	Operation	Object	BI	AI	PPtr	Nptr
1	T1	10:12	START				0	2
2	T1	10:13	UPDATE	STAFF SL21	(old)	(new)	1	8
3	T2	10:14	START				0	4
4	T2	10:16	INSERT	STAFF SG37		(new)	3	5
5	T2	10:17	DELETE	STAFF SA9	(old)		4	6
6	T2	10:17	UPDATE	PROPERTY PG16	(old)	(new)	5	10
7	T3	10:18	START				0	11
8	T1	10:18	COMMIT				2	0
9		*10:19*	*CHECKPOINT*	*T2, T3*				
10	T2	10:19	COMMIT				6	0
11	T3	10:20	INSERT	PROPERTY PG4		(new)	7	12
12	T3	10:21	COMMIT				11	0

Checkpointing-I

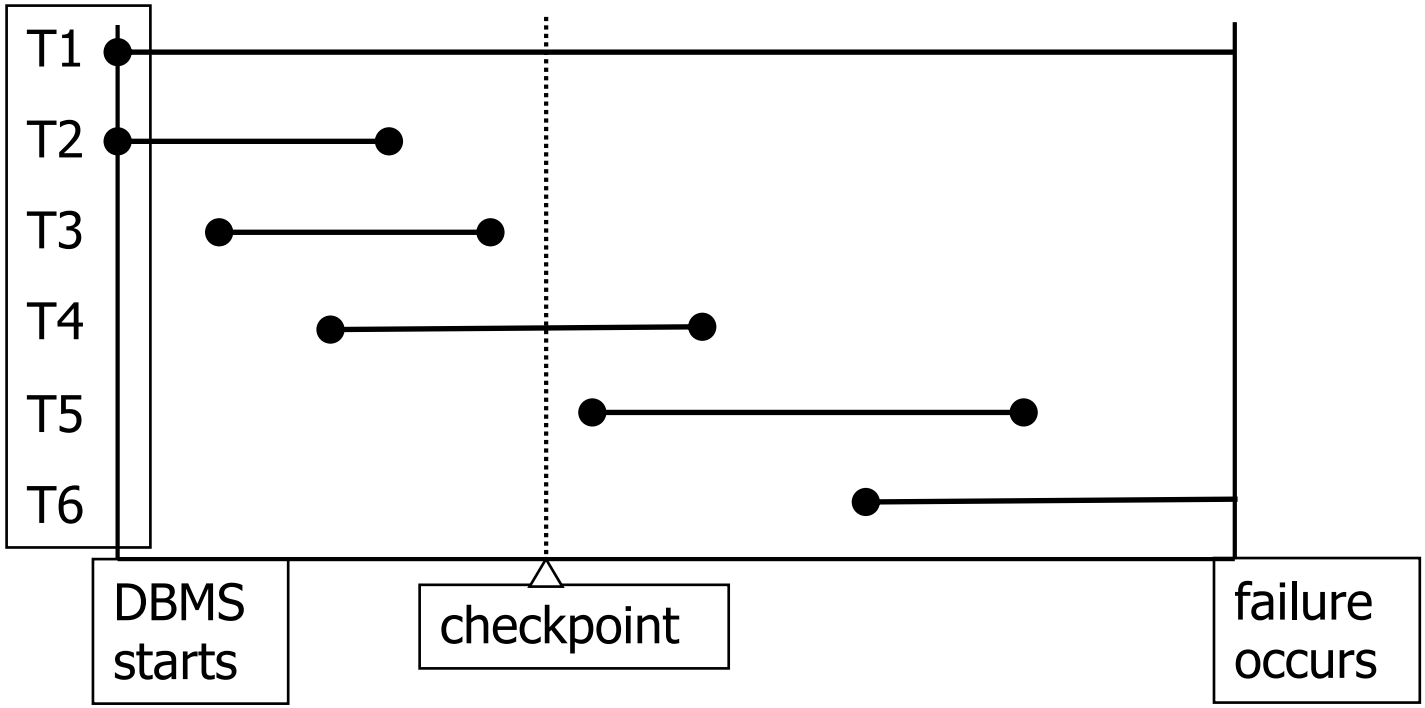
- The information in the log file is used to recover from a failure.
- To recover, we need to know from which point in the log we need to start undoing and redoing changes.
- The problem is that the log describes updates which have been made (to the buffers), but which have not necessarily been written to disk.
  - i.e. they may still be in a buffer waiting to be flushed.
- Without further information, we will needlessly end up redoing transactions that have been safely written to the disk.
- To limit how much this has to be done, we use **checkpointing**.
- A **checkpoint** is a point of synchronisation between the database on disk and the transaction log file
  - At the time of a checkpoint, all buffers are **force-written** to disk.

Checkpointing-II

- Checkpoints are scheduled at pre-determined intervals and involve the following operations:
  - Writing all log records in main memory out to disk.
  - Writing the modified blocks in the database buffers out to disk.
  - Writing a checkpoint record to the log file containing the identifiers of all transactions that are active at the time of the checkpoint (see previous example)
- When a crash occurs, the recovery manager examines the log file for the last checkpoint record.
  - All transactions that have committed since (i.e. after) the last checkpoint are redone
  - Any transactions active at the time of the crash are undone.
- Since checkpoints are relatively inexpensive, it is often possible to take 3 or 4 per hour
  - This way, no more than 15-20 minutes work is lost.

Use of REDO/UNDO with Checkpointing

- Consider, again, the example from before:



- When failure occurs, we can assume that T2 and T3 are permanently recorded (since they committed before the checkpoint).
- However, T1 and T6 must be undone (since they were active at the time of the crash), and T4 and T5 must be redone (since they committed after the checkpoint i.e. their updates will not have reached the disk).

Recovery Techniques

- The recovery manager must be able to restore a database to a consistent state after a crash has occurred.
- We assume that the database is not physically damaged, just inconsistent.
  - For a damaged database, the backup archives must be used.
- To restore consistency, the recovery manager must process the before- and after-images held in the log to undo and redo the changes which have caused the inconsistency.
- We will now examine two different recovery techniques.
  - Deferred update
  - Immediate update

Deferred Update

- Using this technique, updates are not written to the database until after a transaction has reached its commit point.
- Thus, if a transaction fails before the commit point, it will not have modified the database (its updates will not have reached the database buffer or disk).
  - Therefore, no changes need to be undone.
- However, it may be necessary to redo the updates of transactions which committed after the checkpoint as their effect may not have been written to the database on disk at the time of a crash.
- The way in which the log is used is described next...

Deferred Update Log Usage

- When a crash occurs, the log is examined to see which transactions were active at the time of the failure.
- Starting at the last log record, we go back to the last checkpoint.
  - Any transaction with both START and COMMIT records should be redone (since it committed after the checkpoint, and not 'all' its updates will have reached the disk).
  - The *redo* procedure applies the after-image log records to the database on disk in the *original order* in which they were written to the log.
  - Any transaction with START and ABORT records is ignored, since no database updates need to be redone or undone (hence, the before-image is not required for deferred update).
- Note that further system crashes do not affect recovery
  - It does no harm to redo database updates more than once.

Immediate Update

- Using this technique, updates are applied to the database at the same time *as they occur*.
  - It is not necessary to wait until the commit point is reached.
- We still have to redo the updates of committed transactions following a failure (i.e. those transactions which committed ‘after’ the last checkpoint).
- However, it may now be necessary to undo the effects of transactions that had not committed at the time of failure - since their (immediate) updates will have reached the buffers, and subsequently the disk (at checkpoint).
- The way in which the log is used is described next...

Immediate Update Log Usage

- If a transaction aborts, the log can be used to undo it using the saved before-images.
  - Since a transaction may change an item several times, the changes made by an aborted transaction are undone in reverse order.
  - This guarantees that the database is restored to the state before the transaction started.
- On failure, recovery involves using the log to undo or redo transactions.
  - Transactions with both START and COMMIT records are redone (since the transaction has committed after the last checkpoint), using the after-image information in the log (redo is performed in log order).
  - Transactions with a START but no COMMIT record (i.e. active at time of crash) are undone using the before-image information (undo is performed in reverse log order).
- *If interested you can read more about details of the actual log operations for Deferred Update and Immediate Updates, in Connolly & Begg, Sec. 17.3*

End of Lecture

Would you like to ask anything?

Don’t forget to read the notes again, and Ritchie Chapters 6 and 7.