Relational Databases Management Issues (Part 3)

Overview

- Snapshot isolation
- Timestamping concurrency control
- Security
  - Last week

- Recovery in DBMSs
  - This week

- The lecture slides are based on the ones provided with the textbooks:
  - Connolly. and Begg, C. "*Database Systems - A Practical Approach to Design, Implementation, and Management*.", 6th Ed., Pearson Education Ltd
  - Korth, H., Silberschatz, A. and Sudarshan S., *"Database Systems Concepts*.". 7th Ed., International Ed, McGraw Hill, 2019 (ISBN 9781260084504)

Database Recovery

- Definition: The process of restoring database to a correct state in the event of a failure (reliable DB)
- Recovery algorithms have two parts:
  - Actions taken *during normal transaction processing* to ensure that enough information exists to allow recovery from failures.
  - Actions taken *after a failure* to recover the database state to one that ensures database consistency, transaction atomicity, and durability.
- The recovery scheme must also provide *high availability*; that is, it must minimize the time for which the database is not usable after a failure.

Types (and Causes) of Failures

- System crashes, e.g. loss of main memory
- Media failure, e.g. head crashes, loss of data in a block of secondary storage
- DBMS software failures, e.g. wrong results, etc.
- Application SW error, e.g. bad input, logical errors
- Natural physical disaster – fire, flood, power cut, etc.
- Carelessness – destruction of data or facilities by operators or users
- Malicious, security-related – corruption of data, HW, or SW, sabotage

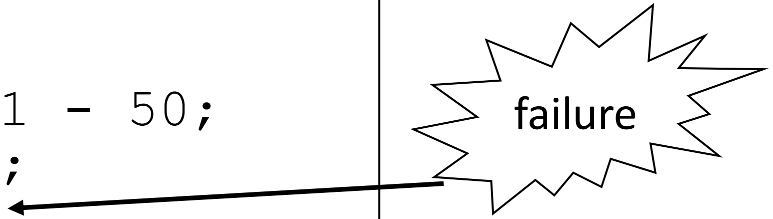
Types of Failures; Some Examples

- Transaction failure
  - Logical error - the transaction cannot continue with its normal execution because of some internal condition, e.g. due to bad input, data not found, or resource limit exceeded
  - (Database) system error - the system enters an undesirable state (e.g. deadlock), and the transaction cannot continue in a normal way. The transaction can, however, be executed at a later time.
- System crash
  - A hardware malfunction, or a bug in the OS/database causes the loss of the content of volatile storage and brings transaction to a halt.
- Disk failure
  - A disk block loses its content because of a head crash or failure during data transfer operation.
    - Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure

Need for recovery; an example

- Transfer of £50 from ACC1 to ACC2, with the initial values being £1000 and £2000, respectively

```
T:
read(ACC1);
ACC1  := ACC1  - 50;
write(ACC1);
read(ACC2);
ACC2  := ACC2  + 50;
write(ACC2).
```

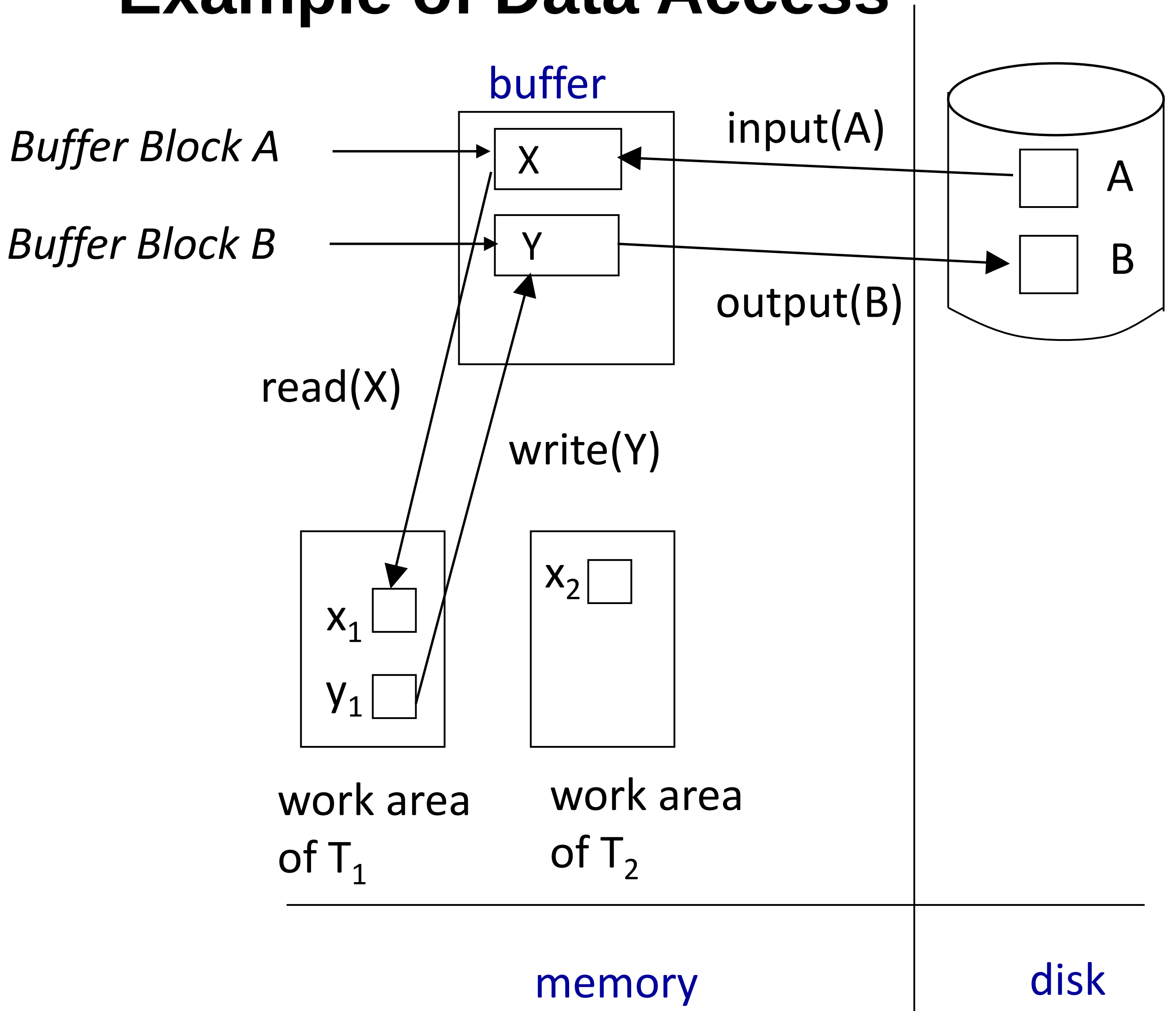


- *Re-execute T:* ACC1 = £900 and ACC2=£2050
- *Do not re-execute T:* ACC1 = £950 and ACC2= £2000
- In either case, £50 is unaccounted for in one of the accounts!
- Note: The example does not satisfy (fully) ACID properties
  – The final result to be: ACC1 = £950 and ACC2= £2050, or values remain unchanged

Transactions & Recovery

- Transactions represent basic unit of recovery
- Use of *DB Buffer* in main memory from which data is transferred to and from secondary memory (e.g. disk)
- When the buffer is *flushed* to secondary memory, the effects of write operations become permanent
- Buffer is flushed either when the buffer is full, or transaction has committed

Example of Data Access



Note: There's a difference between volatile (RAM) memory, non-volatile i.e. stable (magnetic/optical disk, magnetic tape, flash) memory, and approximation to stable memory (RAID levels).

Transactions & Recovery

Failure between writing to the buffer and flushing buffer to secondary storage…

- Recovery manager (see Week 3 slide on Transaction Subsystem) responsible for atomicity and durability.
- If failure occurs after transaction has issued its commit, then recovery manager has to ***redo*** (*rollforward*) transaction's updates, for *durability (*and *atomicity*: "all or nothing")*.*
- If transaction had not committed at failure time, recovery manager has to ***undo*** (*rollback*) any effects of that transaction for *atomicity.*

- Partial undo – only one transaction has to be undone
- Global undo – all active transactions have to be undone

Recovery Facilities

- DBMS should provide *various* facilities to assist with recovery:

- Backup mechanism
  - Makes periodic backup copies of the database
- Logging facilities
  - Keep track of the current state of transactions and database changes
- Checkpoint facilities
  - The tentative, in-progress database updates are made permanent
- Recovery manager
  - The system restores the database to a consistent state following a failure

Log File

- Log File:
  - It contains information about all update activities in the database
    - Transaction records
    - Checkpoint records
- Log files are duplexed or triplexed
  - In Oracle, if redo logs are corrupt, there is no way to nicely recover the database to a technically consistent state (at least not without restoring data from backups).
- Log files stored on a fast direct-access storage, for quick recovery
- Log files – a potential (critical) performance bottleneck
  - PostgreSQL – best practice is to place Write-Ahead-Log (WAL) on a separate disk/machine: the possibility of a (HW) failure affecting both data pages AND log record is thus *minimized.*
- Log files could be split into separate random-access files.
- Used for other purposes: auditing (e.g., log ons/offs etc), (non-diverse) replication.
  - Installing changes in this way is faster than re-executing SQL operations

Database Transaction Log
(a related slide exists in week 3 lecture)

- In order to support atomicity/durability the system has to *maintain a record of all database modifications on a stable storage*, before actually modifying the database
- Physically, it is *a file of updates done to the database*, stored in stable storage
- Redo a modification to *ensure atomicity and durability*, or undo a modification to *ensure atomicity* in case of a failure
- Before a user receives a "Commit complete" message, the system must *first successfully write the new or changed data to a log file*.
  - All changes included in the transaction are first written into the log buffer.
    - Using memory in this way for the initial capture aims to reduce disk IO. Of course, when a transaction commits, the log buffer must be flushed to disk, because otherwise the recovery for that commit could not be guaranteed.

Transaction Log Record

- Transaction identifier
- Type of log record - operation (transaction start, abort or commit; insert, update, delete)
- Identifier of data item affected by the database action (insert, delete, and update operations)
- Before-image of data item
- After-image of data item
- Log management information (pointer to previous and next log records for the transaction - all operations)

CSR Building confidence in a computerised world
www.csr.city.ac.uk

Sample Log File

Time precision can be, and usually is, higher than seconds

Pointers to the previous and next record in the log, for **each** transaction

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

No Tid entry needed – this is a system-wide action

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions (see slide 8 "Example of Data Access")
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise, how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (rigorous two-phase locking)
- Log records of different transactions may be interspersed in the log.

Undo and Redo Operations

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

Undo and Redo on Recovering from Failure

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

Checkpoint Record

In presence of failure, how much do we need to search in the log file and redo the transaction?

Redoing/undoing **all** transactions recorded in the log (since the particular DB was created!) can be very slow/infeasible

1. processing the entire log is time-consuming (if the system has run for a long time)
2. we might unnecessarily redo transactions which have already output their updates to the database.

Checkpointing: Point of synchronisation between database and the log file. All buffers are force-written to secondary storage

Checkpoint operations

- Write all Log records in the main memory to secondary storage
- Write the modified blocks in the database buffers to secondary storage
- Write a checkpoint record to the Log file.
  - This record contains the IDs of all txns that are active at the time of the checkpoint, e.g. see "**Sample Log File**" slide

- All updates are suspended while doing checkpointing
  - Note: There exist improvements to this approach... but we will not cover them in the module.

Recovery Algorithm

So far: we covered key concepts

Now: we present the components of a basic recovery algorithm

Recovery Algorithm

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

* E.g. initiated by the user/client application

Recovery Algorithm (cont.)

T_i has committed before the checkpoint

<T_i commit> record before **<checkpoint>** record

- Any DB modification made by T_i has been written to the DB either prior to the checkpoint, or as part of the checkpoint itself, thus in any case:
  – No need to REDO T!

Recovery Algorithm (cont.)

- **After failure…**
- A simple checkpoint scheme assumed: a) does not permit any updates to be performed while the checkpoint operation is in progress, and b) outputs all modified buffer blocks to disk when the checkpoint is performed.

- When transactions are performed concurrently
  - After a system crash has occurred, the system examines the log to find the most recent `<checkpoint L>` record
  - The redo or undo operations need to be applied only to a list of transactions active at the time of the checkpoint (L), *and* to all transactions that started execution after the `<checkpoint L>` record was written to the log. Let us denote this set of transactions as T.
  - For all transactions T_k in T that have neither $<T_k$ commit$>$ record nor $<T_k$ abort$>$ record in the log, execute undo(T_k).
  - For all transactions T_k in T such that either the record $<T_k$ commit$>$ or the record $<T_k$ abort$>$ appears in the log, execute redo(T_k).

Recovery Algorithm (cont.)

- It may seem strange to redo T_k if the record `<T`$_k$ `abort>` is in the log.
- But note that if `<T`$_k$ `abort>` is in the log, so are the **redo-only** records written by the *undo* operation. Thus, the end result will be to undo T_k' s modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time.
  - The undo operation not only restores the data items to their old value, but also writes log records to record the updates performed as part of the undo process. These log records are special **redo-only** log records (**compensation** log records) since they do not need to contain the old-value of the updated data item
  - Applying these redo-only records will have an effect of undoing T_k's modifications - these steps restore old values!
    - Redo-only records are created only after the transaction was rolled-back – i.e. `<T abort>` record exists in the log – this is **not** the case for the Txns active at failure time!
  - Need to cater for failures during recovery too!

Recovery Algorithm (Cont.)

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

Recovery Algorithm (Cont.)

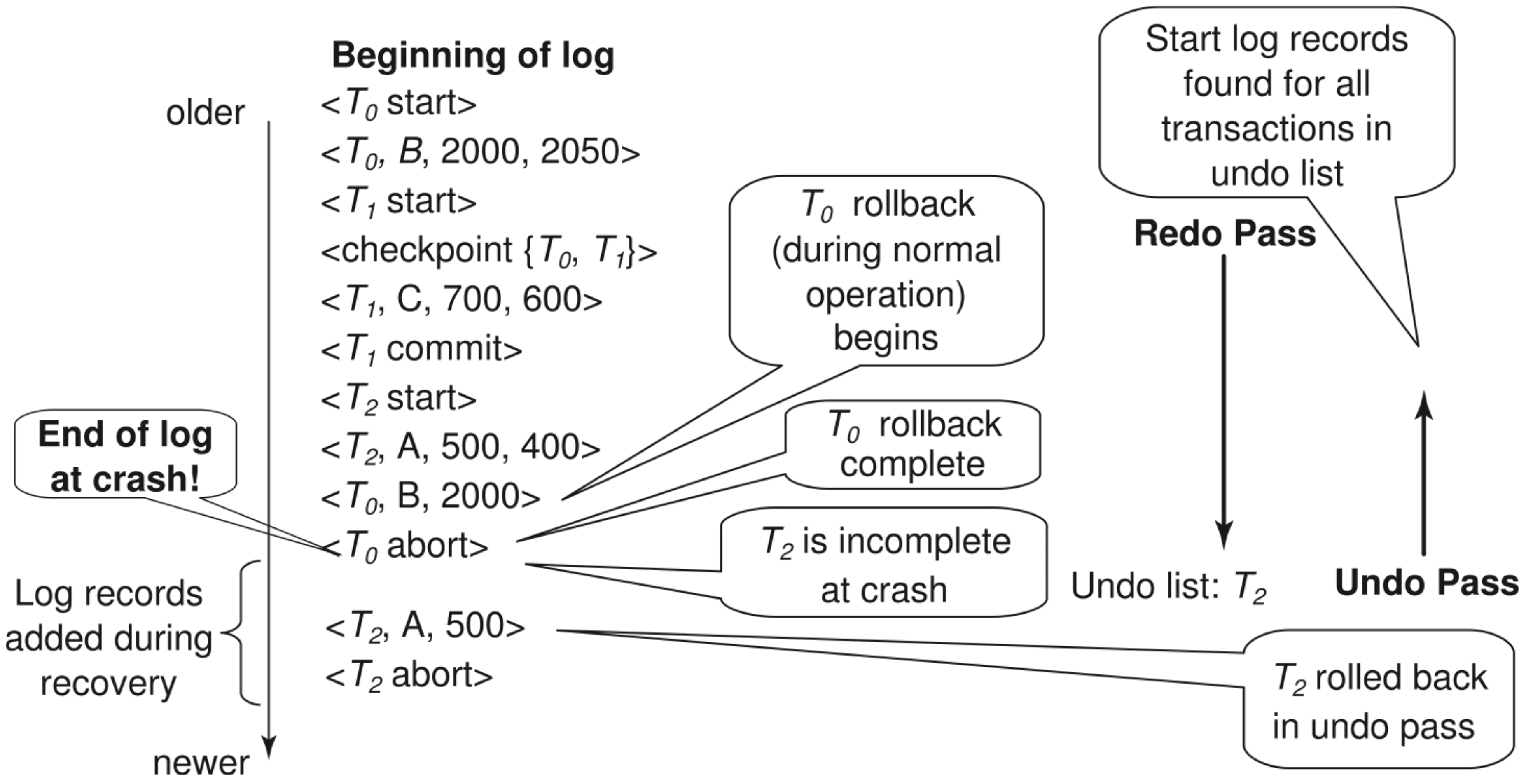
- **Undo phase:**
  I. Scan log backwards from the end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        a) perform undo by writing V_1 to X_j.
        b) write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        a) Write a log record <T_i **abort**>
        b) Remove T_i from undo-list
     3. Stop when undo-list is empty

        i.e. <T_i **start**> has been found for every transaction in undo-list

- After undo phase completes, normal transaction processing can commence

- Must remember that changes are done to **both**:
  - transaction log – e.g. see 1.b) above, and
  - data (files) – e.g. see 1.a) above

Example of Log Evolution (incl. during recovery)



- Simplification due to T0 being recovered in the REDO pass (not in the UNDO pass). But, this is at expense of more log records being written during normal operation - <T_0 , B, 2000>. Recovery phase performance is improved, however.
  - Note: This is a simple example

Main Points (of Lectures 3, 4 and 5)

- Maintain consistency and reliability of the database in the presence of failures of hardware and software, and especially when users access database concurrently

- It is necessary to support three main functions:
  - Transaction management
  - Concurrency control
  - Database recovery

Please recall – extensive resources available :

http://readinglists.city.ac.uk/index.html

http://libguides.city.ac.uk/computing

Required Reading

- Connolly,T. and Begg, C. "*Database Systems - A Practical Approach to Design, Implementation, and Management*.", 6th Ed., Pearson Education Ltd
  - Recovery – Section 22.3;
- Korth, H., Silberschatz, A. and Sudarshan S., "Database Systems Concepts." 7th Ed., International Ed, McGraw Hill, 2019 (ISBN 9781260084504)
  - Especially relevant material about Recovery: sections 19.1-19.4

- This list is only indicative, and the text in these sections should be of course looked at as "additional", but required, reading to the lecture slides.
- This is certainly not an exhaustive list.
- The content of the chapters referred to might NOT be fully relevant to the module. You need to make the final decision about what is relevant and what is not yourselves, given the material that we covered in the lecture.