



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David



RECOVERY & FILE ORGANISATION

21-Nov-2023

Week 8: Database Systems (DBS)
Dr Suresh Shanmugasundaram

So far...



1. ER Modeling
2. ER Models to Relational Tables
3. Functional Dependencies
4. Normalisation
5. Transactions & its properties
6. Concurrency Control and its problems
7. Locking & Deadlocks

Learning Objectives



- ❑ Recovery of a database
- ❑ Log based recovery
- ❑ Recovery for concurrent execution
- ❑ Checkpoints
- ❑ Backups
- ❑ File Organisation & Indexing

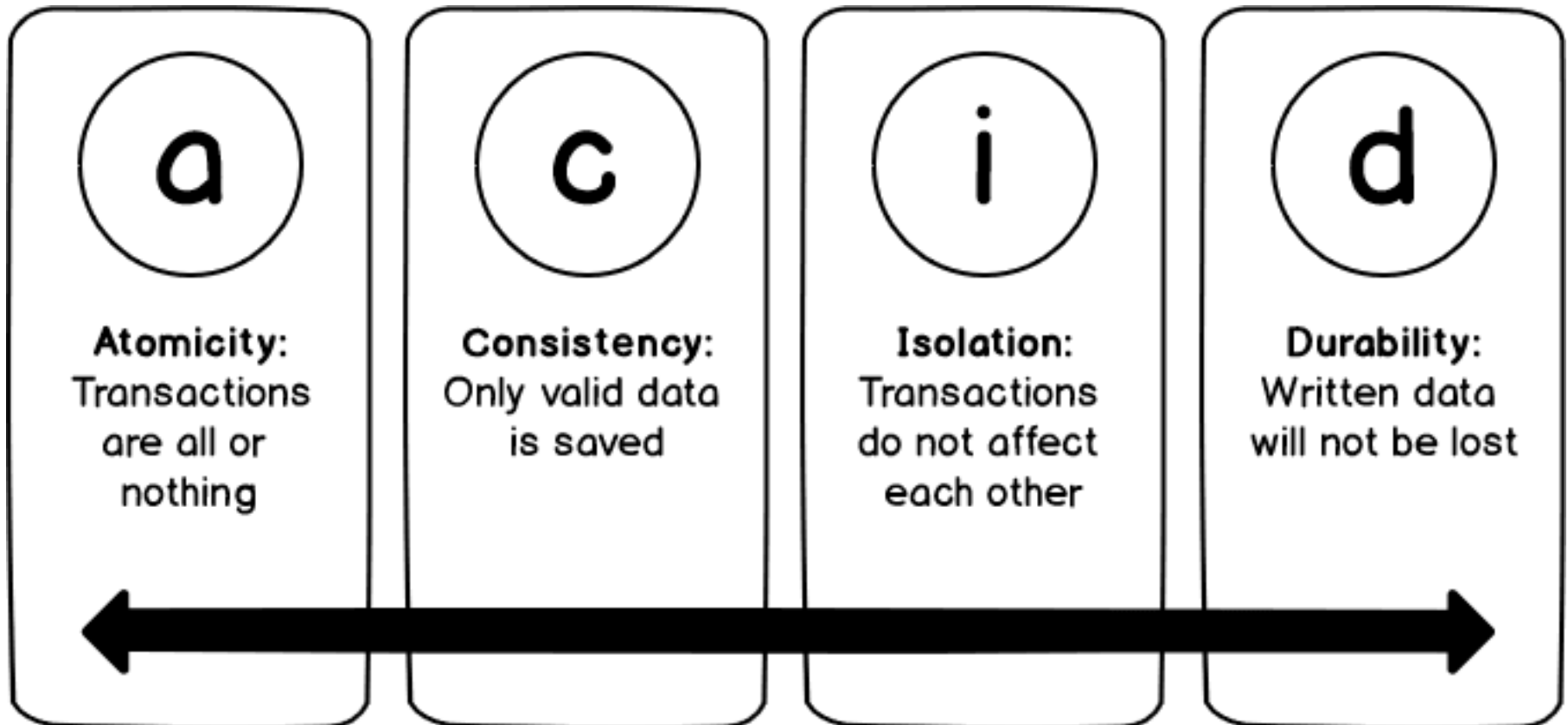
Recap - Quiz



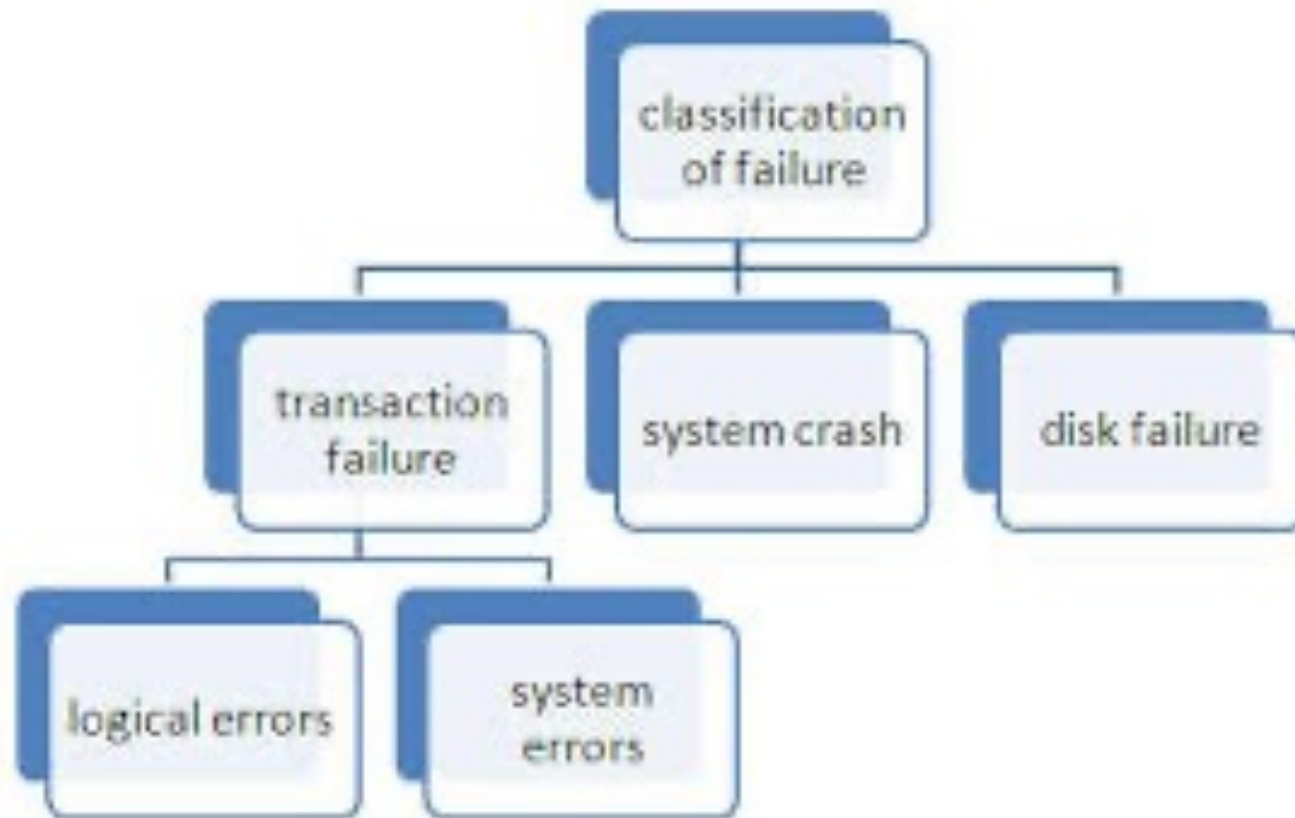
<https://forms.office.com/e/b0fGr0YTpG>

Durability – One of the ACID properties

- **Durability.** If a transaction is committed, its changes to the database are permanent, **even in the presence of system failures.**

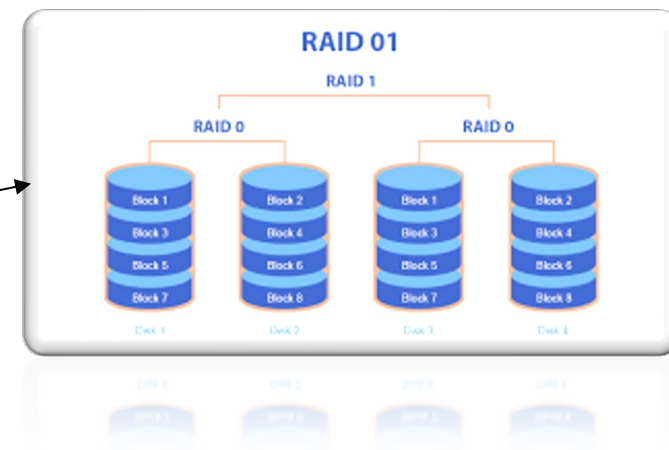


Failure Classification



Recovery

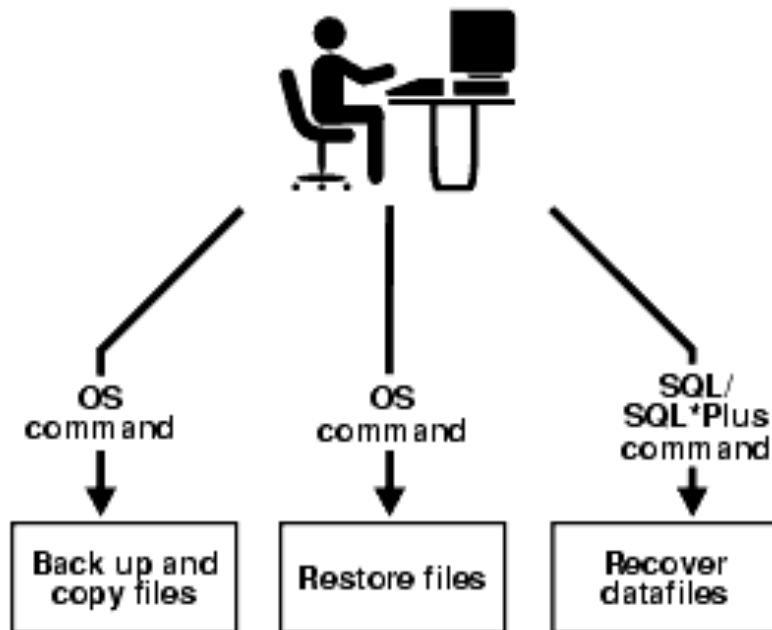
- Prevention is better than a cure
 - Reliable OS
 - Security
 - UPS and surge protectors
 - RAID arrays



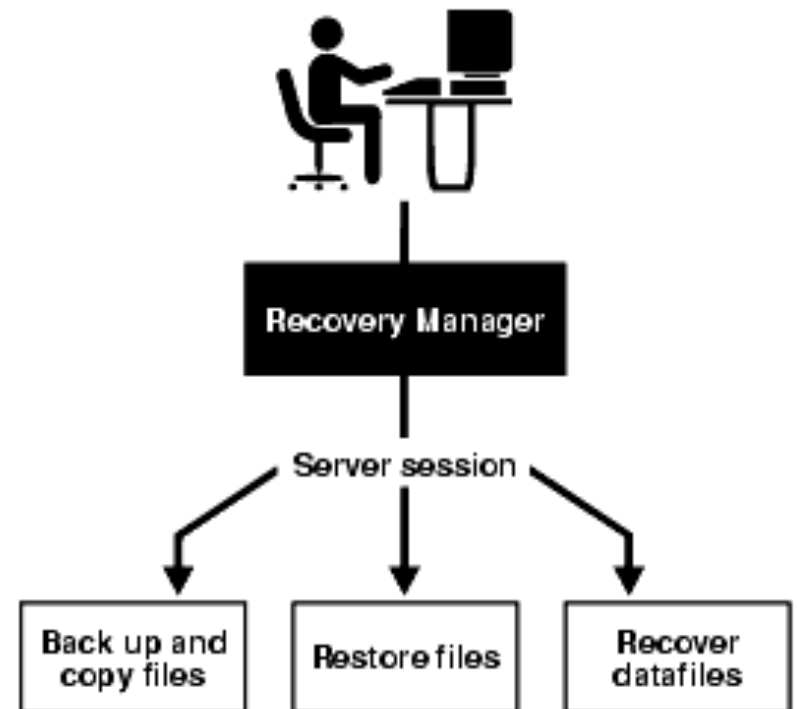
- Can't protect against everything, **system recovery** will be necessary.

Recovery Manager

User-Managed backup and recovery



Backup and recovery with Recovery Manager

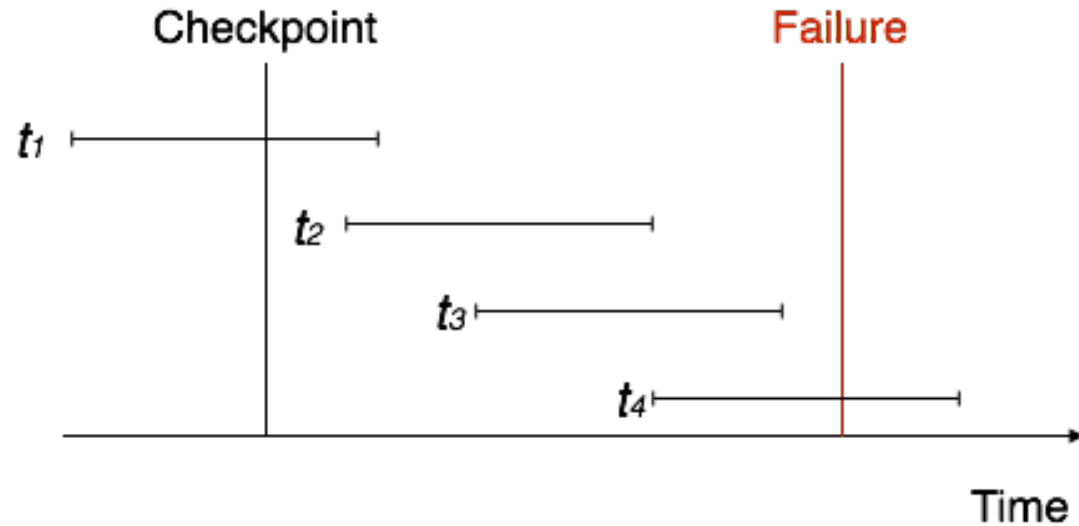


Recovery Algorithms

- Consider transaction T_i that transfers £50 from account A to account B
 - ▣ Two updates: 1. subtract 50 from A and 2. add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
 - ▣ A failure may occur after one of these modifications have been made but before both of them are made.
 - ▣ Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
 - ▣ Not modifying the database may result in lost updates if failure occurs just after transaction commits.

Recovery Algorithms

- Has two parts:
 1. Actions taken during **normal transaction processing** to ensure enough information exists to recover from failures.
 2. Actions taken **after a failure** to recover the database contents to a state that ensures atomicity, consistency and durability.



Storage Structure

□ **Volatile storage:**

- does not survive system crashes
- examples: main memory, cache memory

□ **Nonvolatile storage:**

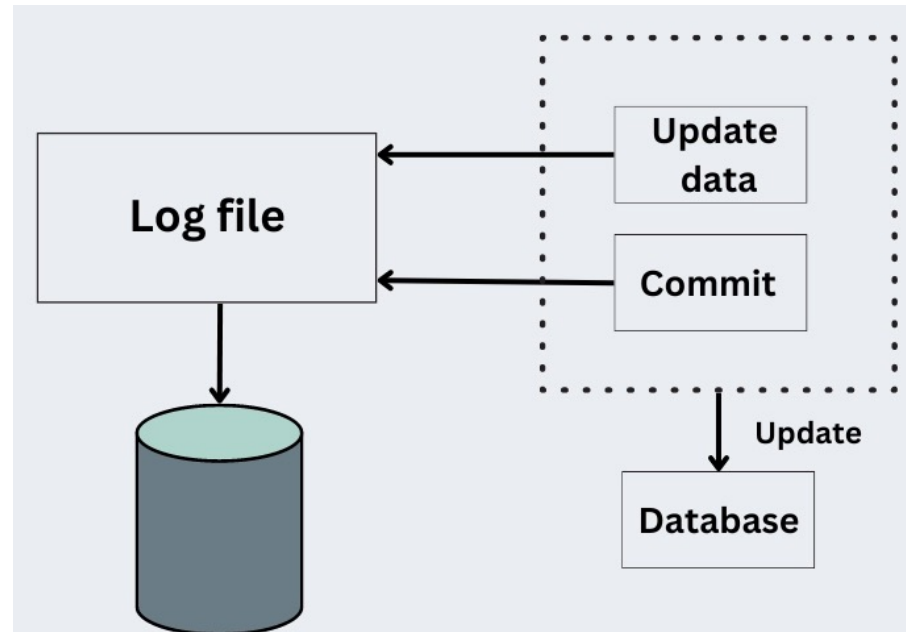
- survives system crashes
- examples: disk, tape, flash memory,
non-volatile (battery backed up) RAM
- but may still fail, losing data

□ **Stable storage:** (This is our assumption for this lecture)

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

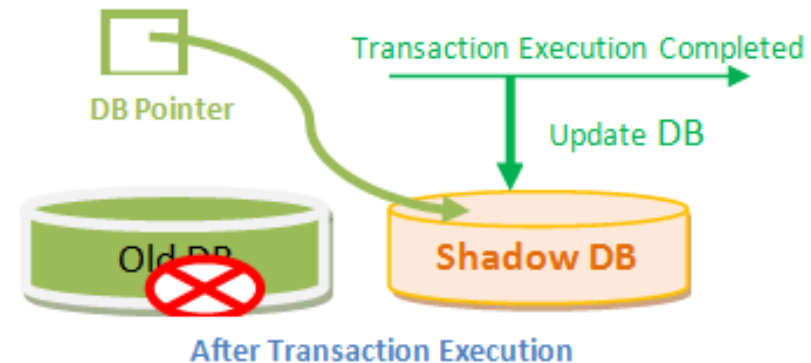
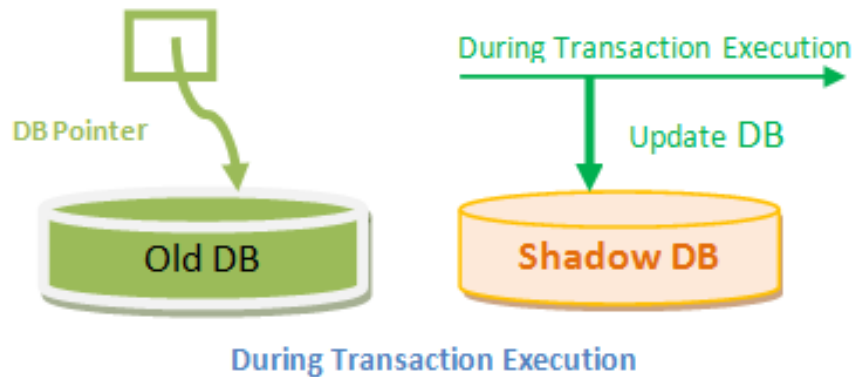
Recovery and Atomicity

- ❑ Use **log-based recovery mechanisms**.
- ❑ Less used alternative: **shadow-copy & shadow-paging**



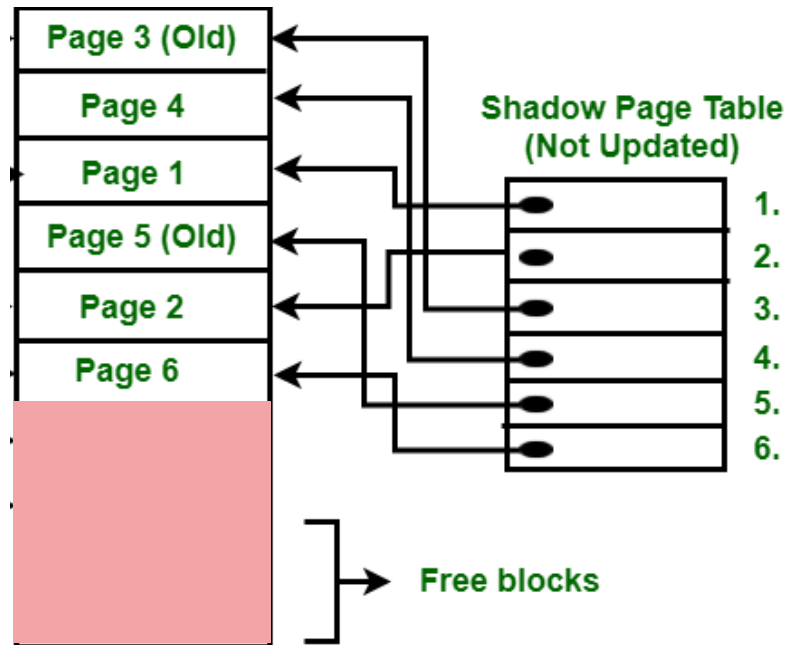
Recovery and Atomicity

- Less used alternative: shadow-copy & shadow-paging



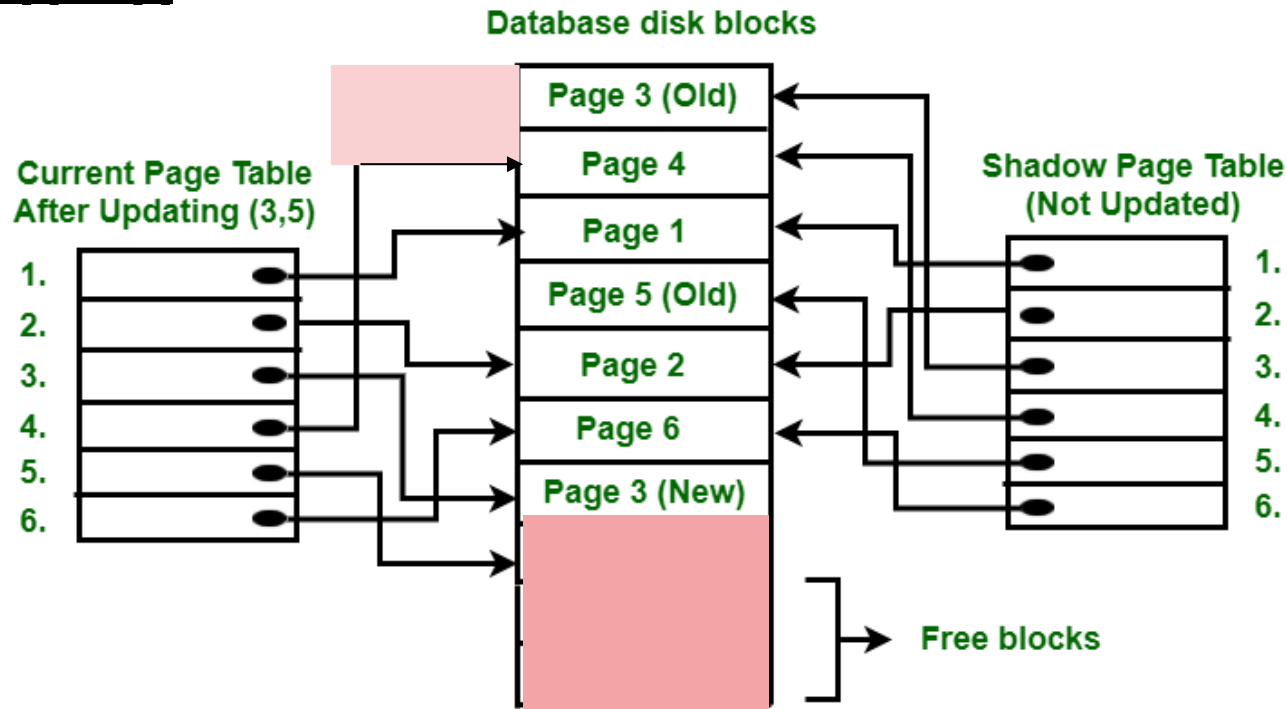
Recovery and Atomicity

- Less used alternative: **shadow-copy & shadow-paging**



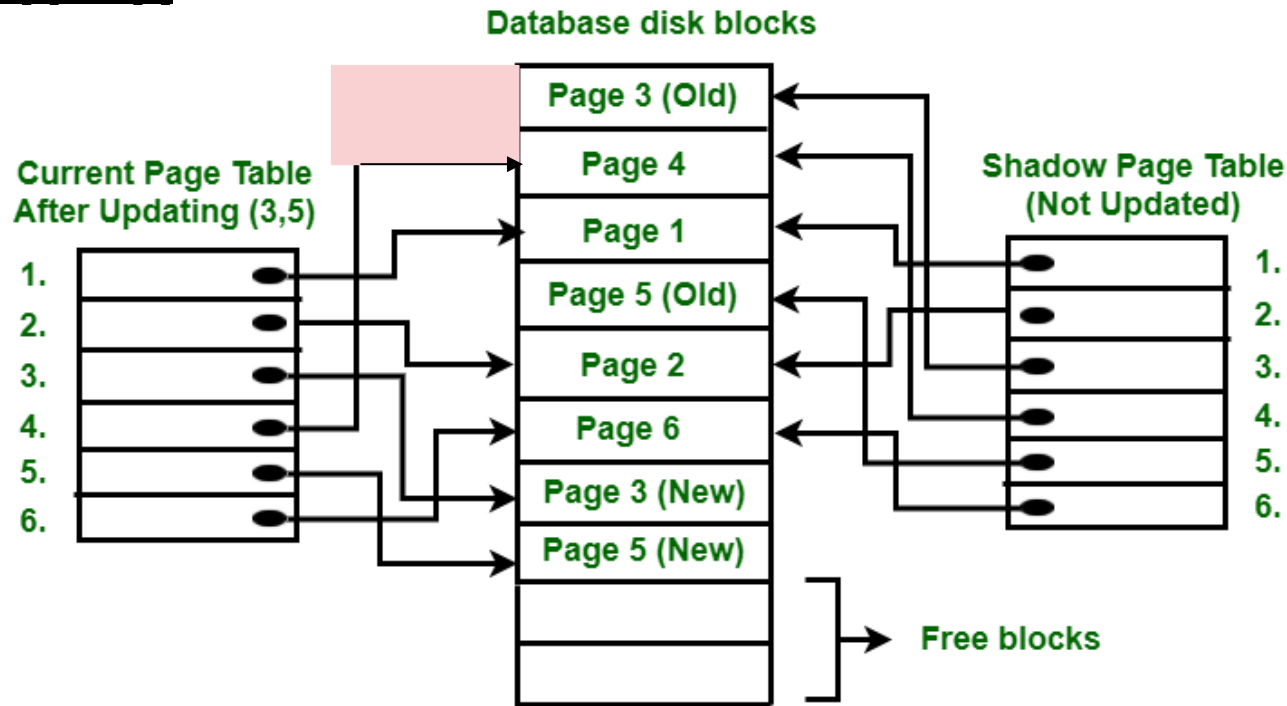
Recovery and Atomicity

- Less used alternative: **shadow-copy & shadow-paging**



Recovery and Atomicity

- Less used alternative: **shadow-copy & shadow-paging**



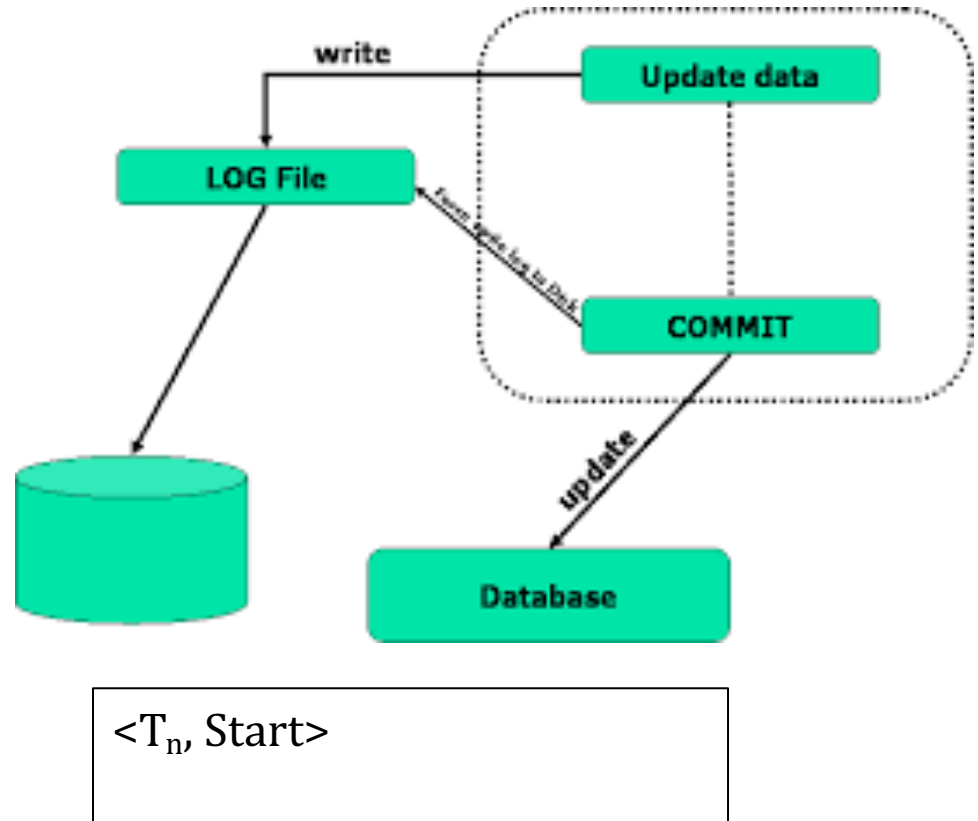
Transaction Log

- The transaction log records details of all transactions
 - Any changes the transaction makes to the database
 - When transactions complete and how
- The log is stored on disk, not in memory
 - If the system crashes, the log is preserved
 - **Write ahead** log rule
 - The entry in the log must be made before COMMIT processing can complete

Write-ahead logging: Any change to a database object is first recorded in the log; the record in the log must be written to stable storage before the change to the database object is written to disk.

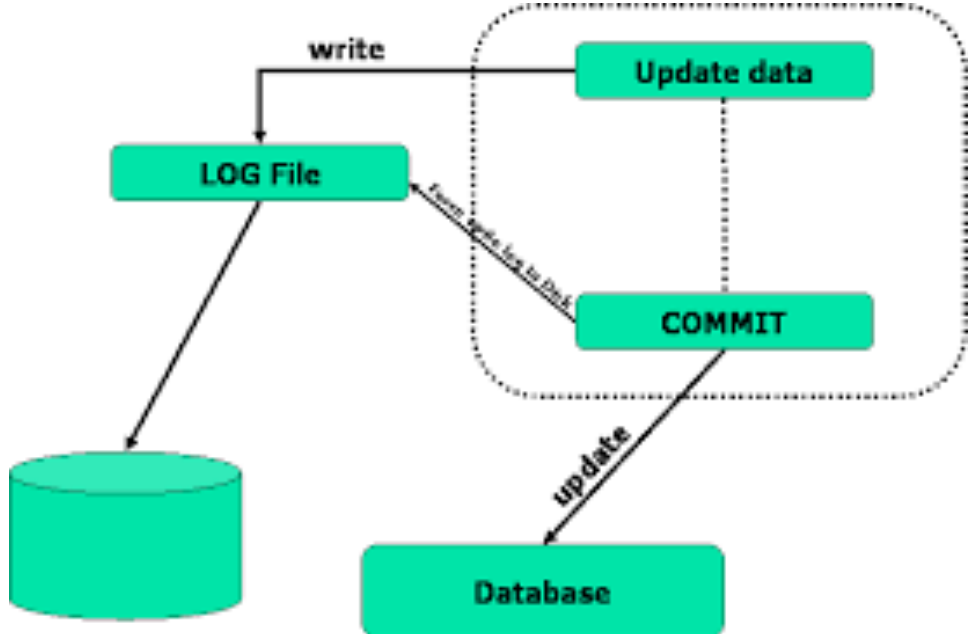
Log based recovery

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.



Log based recovery

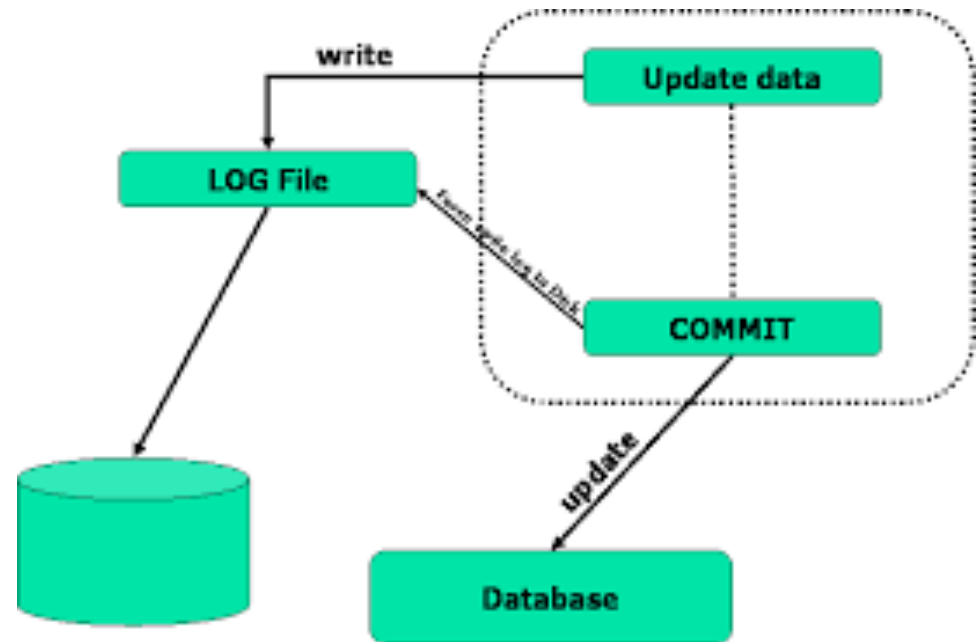
- 



$\langle T_n, \text{Start} \rangle$
 $\langle T_n, X, V_1, V_2 \rangle$

Log based recovery

- When the transaction finishes, it logs – as shown here



$\langle T_n, \text{Start} \rangle$
 $\langle T_n, X, V_1, V_2 \rangle$
 $\langle T_n, \text{commit} \rangle$

Log based recovery

- A **log** is kept on stable storage.
 - The log is a sequence of **log records** and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a $\langle T_i \text{ start} \rangle$ log record
- Before T_i executes **write**(X), a log record $\langle T_i, X, V_1, V_2 \rangle$ is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes its last statement, the log record $\langle T_i \text{ commit} \rangle$ is written.

Log based recovery

- $\langle T_n, \text{Start} \rangle$
- $\langle T_n, \text{City}, \text{'Cardiff'}, \text{'Swansea'} \rangle$
- $\langle T_n, \text{Commit} \rangle$

Log based recovery: Two approaches

- a. **Immediate database modification** - allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits.
- b. **Deferred database modification** - Performs updates to buffer/disk only at the time of transaction commit.

Log based recovery - Example

A-1000, B-2000, C-700

		log	writing operations to DB
T0	T1		
read(A)		<T0 start>	
A = A - 50			
write(A)		<T0, A, 1000, 950>	update A to 950
read(B)			
B = B + 50		<T0, B, 2000, 2050>	update B to 2050
write(B)			
commit		<T0 commit>	
	read(C)	<T1 start>	
	C = C - 100		
	write(C)	?	update C to 600
	commit	?	

<T start>, <T, X, V_{old} , V_{new} >, <T commit>

writing is performed *along with* the creation of the log.

Log based recovery - Example

A-1000, B-2000, C-700

		log	writing operations to DB
T0	T1		
read(A)		<T0 start>	
A = A - 50			
write(A)		<T0, A, 1000, 950>	update A to 950
read(B)			
B = B + 50		<T0, B, 2000, 2050>	update B to 2050
write(B)			
commit		<T0 commit>	
	read(C)	<T1 start>	
	C = C - 100		
	write(C)	<T1, C, 700, 600>	update C to 600
	commit	<T1 commit>	

<T start>, <T, X, V_{old} , V_{new} >, <T commit>

writing is performed *along with* the creation of the log.

Log based recovery - Example

<Ti, Start> and <Ti, Commit> or

<Ti, Commit>

Transaction Ti needs to be redone.

<Ti, Start>

Transaction Ti needs to be undone.

Example 1 with crash

log

<T0 start>

<T0, A, 1000, 950>

<T0, B, 2000, 2050>

<T_i, Start>

Transaction T_i needs to be undone.

Example 1 with crash

log

<T0 start>

<T0, A, 1000, 950>

<T0, B, 2000, 2050>

- Consider the log found after a recovery.
- T0 *did not finish*, and must be aborted.
 - ▣ However, it already made some changes in the database (by modifying A to 950 and B to 2050).
 - ▣ Therefore, aborting it calls for **un-doing** these changes.
- Un-doing is done in the **bottom-up** order.
 - ▣ First, set B to 2000, according to the last log record.
 - ▣ Then, set A to 1000, according to the last-but-one record.

Example 2 with crash

log

```
<T0 start>  
<T0, A, 1000, 950>  
<T0, B, 2000, 2050>  
<T0 commit>  
<T1 start>  
<T1, C, 700, 600>
```

- ❑ T0 committed. We do not need to do anything for it (its changes have been physically applied).
- ❑ T1 needs to be aborted.
- ❑ We un-do its changes by setting C to 700.

Example 3 with crash

log

```
<T0 start>  
<T0, A, 1000, 950>  
<T0, B, 2000, 2050>  
<T0 commit>  
<T1 start>  
<T1, C, 700, 600>  
<T1 commit>
```

- ❑ Both transactions committed. Redone for both during recovery
- ❑ We do not need to do anything.

Log based recovery : Problem

- In practice, a transaction may have a large number of statements, which in turn generates a **long log** list.
- Problems in recovery:
 - ▣ Searching inside the log is time-consuming (e.g., for start/commit records).
 - ▣ Most transactions that need to be re-done may have already written all the changes to the disk physically. Although re-doing them causes no harm, it makes the recovery process longer.
- Solution: **checkpoints** — a record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified.

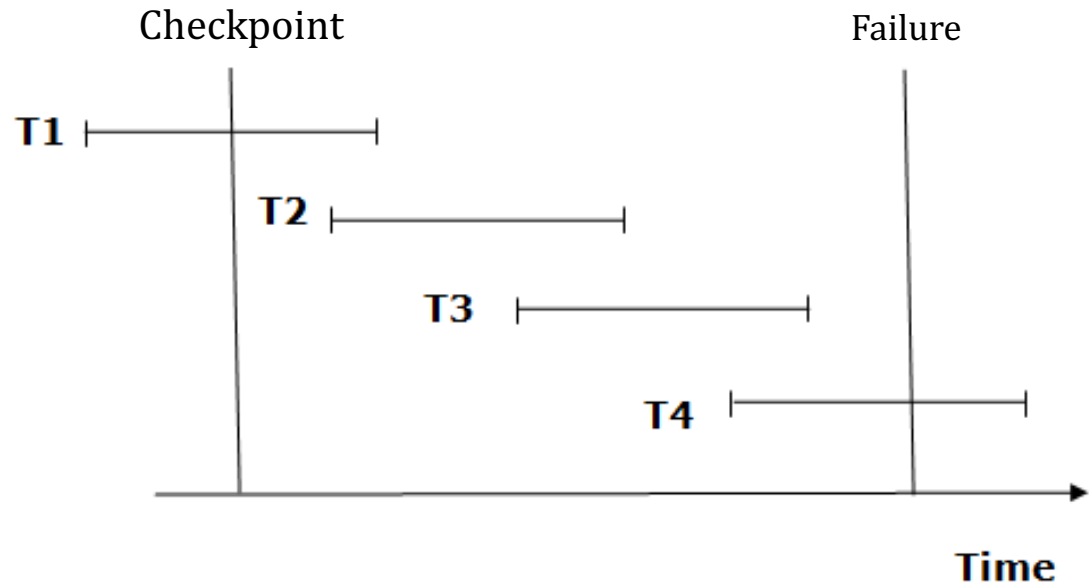
Transaction Timeline

- Recovery with checkpoint
 - ▣ Any transaction that was **running at the time of failure** needs to be **undone**
 - ▣ Any transactions that **committed since the last checkpoint** need to be **redone**
- **<checkpoint *L*>**
- *L* - the list of transactions active at the time of the checkpoint

Transaction Timeline

Any transaction that was **running at the time of failure** needs to be **undone**

Any transactions that **committed since the last checkpoint** need to be **redone**



<Tn, Start> and <Tn, Commit> or <Tn, Commit>.

<Tn, Start> but no commit

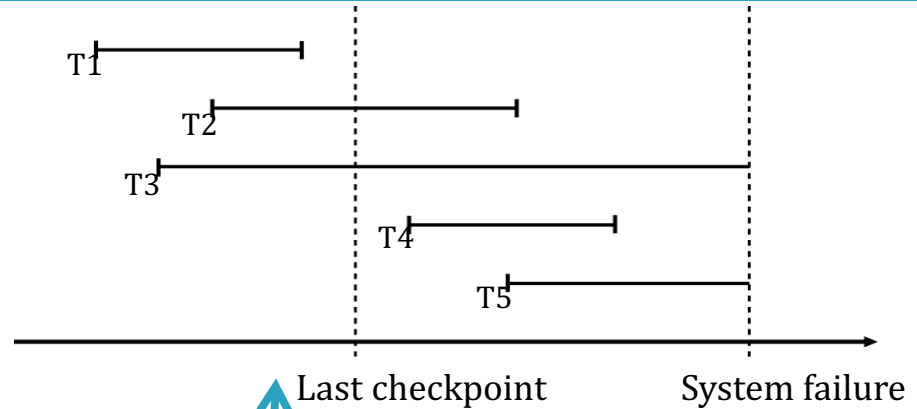
Re-do list :
transactions removed – redone – save logs

Undo list :
transactions undone – logs removed

Checkpoint Example

log

```
<T1 start>
<T1, A, 0, 10>
<T3 start>
<T3, E, 0, 12>
<T2 start>
<T2, B, 0, 10>
<T1 commit>
<checkpoint T2,T3 >
<T4 start>
<T4, A, 10, 20>
<T4, D, 0, 10>
<T5 start>
<T5, A, 20, 30>
<T2, C, 0, 10>
<T2 commit>
<T4 commit>
Failure
```

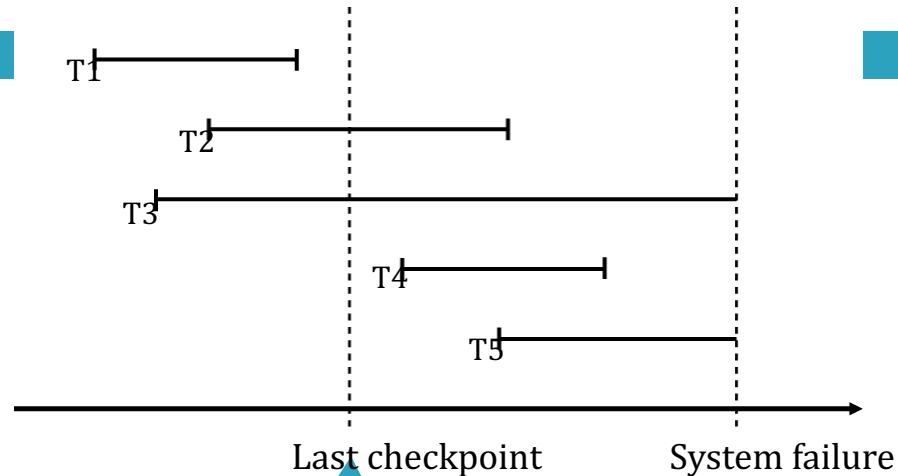


← T2, T3 is the transactions active at the time of the checkpoint

Checkpoint Example

log

```
<T1 start>
<T1, A, 0, 10>
<T3 start>
<T3, E, 0, 12>
<T2 start>
<T2, B, 0, 10>
<T1 commit>
<checkpoint T2,T3 >
<T4 start>
<T4, A, 10, 20>
<T4, D, 0, 10>
<T5 start>
<T5, A, 20, 30>
<T2, C, 0, 10>
<T2 commit>
<T4 commit>
Failure
```



← T2, T3 is the transactions active at the time of the checkpoint

- ❑ T1 can be ignored (updates already output to disk).
- ❑ T2: the part of it **after** the checkpoint is re-done.
- ❑ T4: completely redone.
- ❑ T3, T5: completely undone.

Media Failures

- ❑ System failures are not too severe
 - ▣ Only information since the last checkpoint is affected
 - ▣ This can be recovered from the transaction log
- ❑ Media failures (e.g. Disk failure) are more serious
 - ▣ The stored data is damaged
 - ▣ The transaction log itself may be damaged

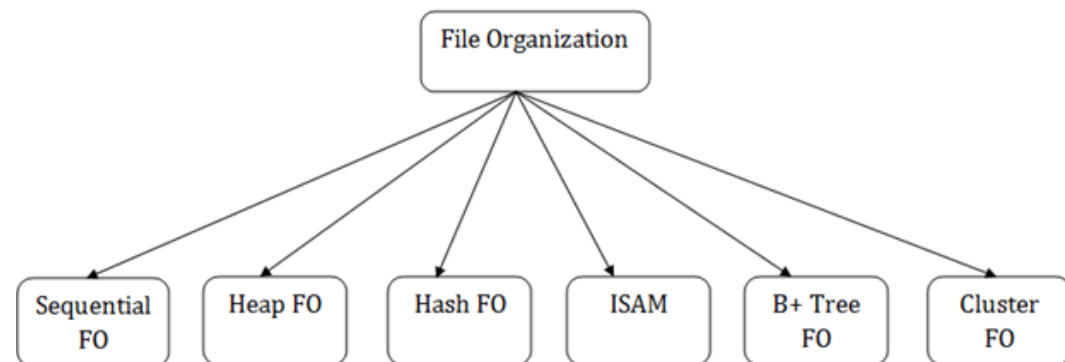
Backups

- Backups are necessary to recover from media failure
 - ▣ The transaction log and entire database is written to secondary storage
 - ▣ Very time consuming, often requires downtime
- Backup frequency
 - ▣ Frequent enough that little information is lost
 - ▣ Not so frequent as to cause problems
 - ▣ Every night is a common compromise

Recovery from Media Failure

1. Restore the database from the last backup
 2. Use the transaction log to redo any changes made since the last backup
- If the transaction log is damaged you can't do step 2
 - ▣ Store the log on a **separate** physical device to the database
 - ▣ This reduces the risk of losing both together

File Organisation



File Organization

- ❑ Databases are stored physically as files of records.
- ❑ These are stored in a **physical storage medium**.
- ❑ Disk space management:
 - Should Operating System services be used?
 - Should RDBMS manage the disk space by itself ?

Storage Access

- Organisation of files is important, especially for gigantic dataset. Why?
- All commercial database systems provide **indexing** mechanisms to accelerate the processing of SQL queries.

Operations on Files

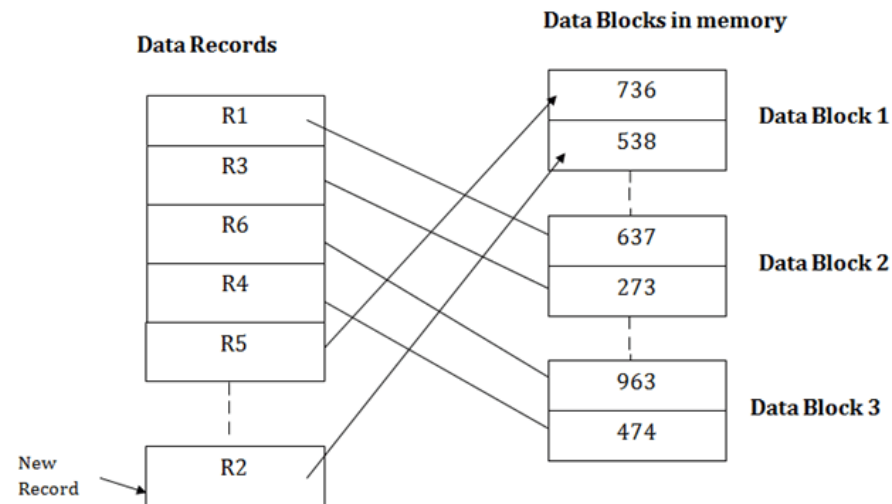
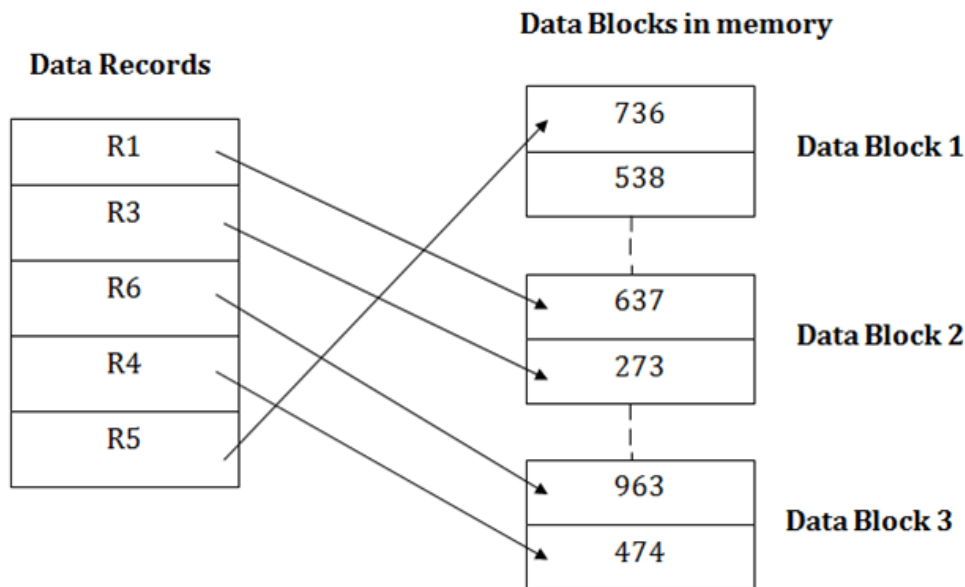


- Insertion of a new record
- Deletion of a record
- Update a record field/fields
- Search for a record
- Range search

File Organisation Methods

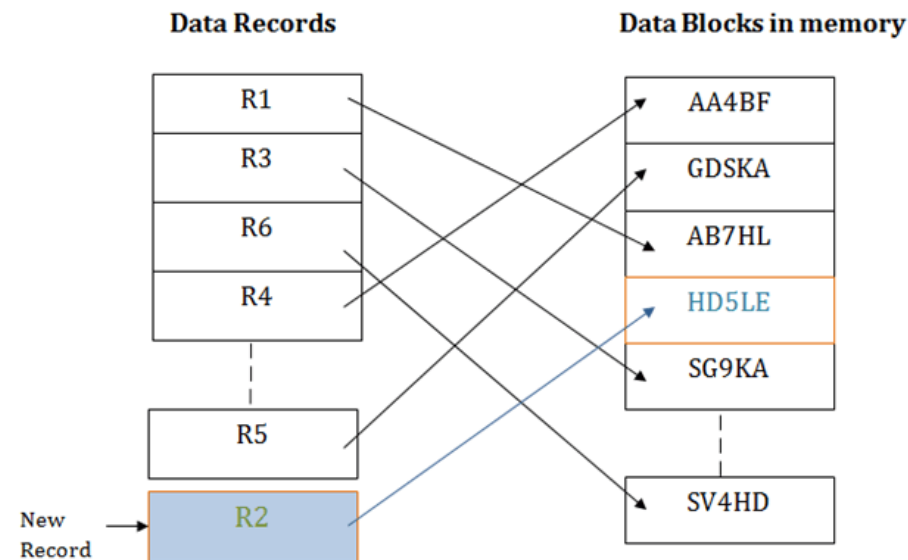
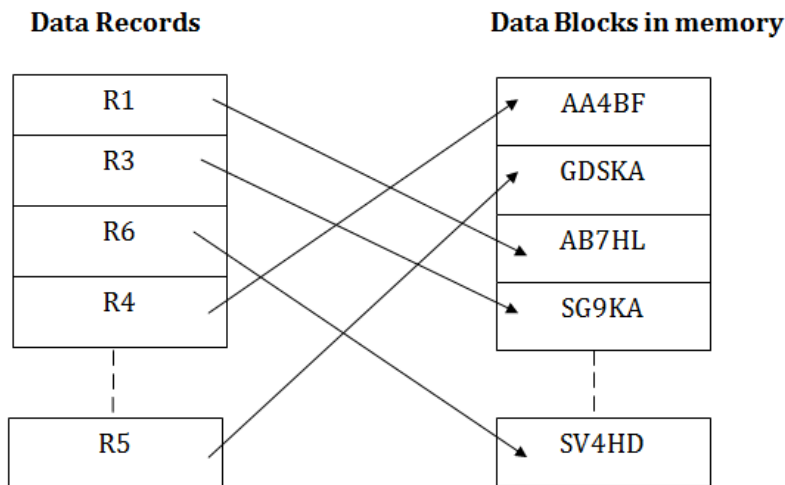
❑ **Heap Files:** unordered data, a record can be placed in the order of their entry, i.e new records at the end, insertion is very efficient, requires a linear search to locate records.

File



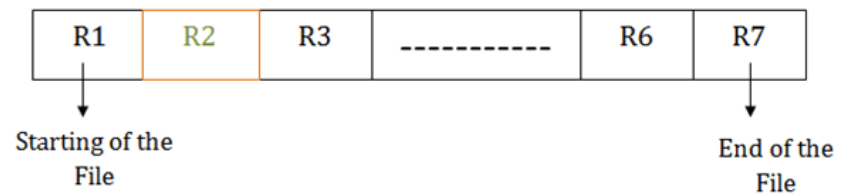
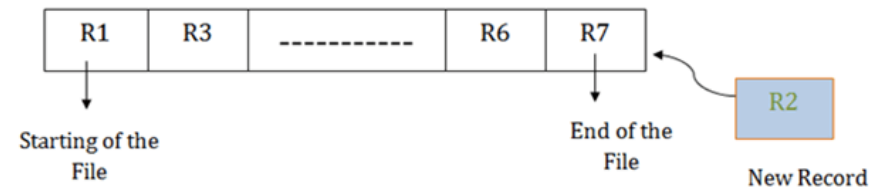
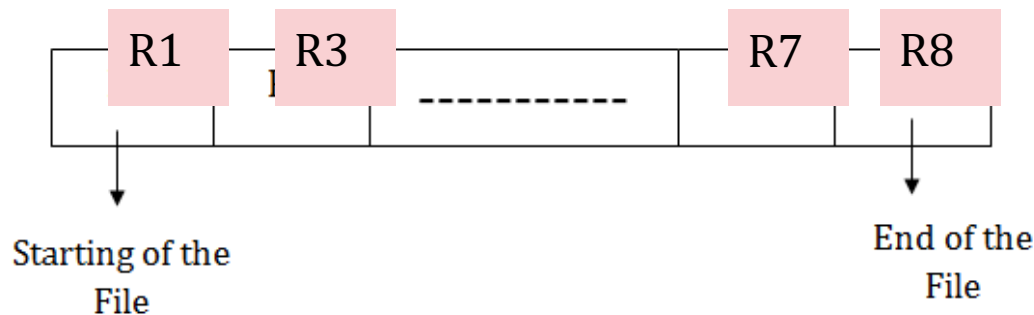
File Organisation Methods

❑ **Hashed Files:** a hash function is computed on some attribute (**hash field**) of each record which specifies the block of the file where the record will be placed. If hash field == key field(PK) then its called **hash key**, provides fast access to arbitrary record.



File Organisation Methods

❑ **Sorted Files:** order the records of a file based on the values of their **ordering field**, if ordering field=key field (PK) then it is called **ordering key** for the file, reading is efficient, uses binary search on ordering field, insertion and deletion is costly, used in databases only with **primary index** file, if ordering field is not a key then its called **clustering index** file.

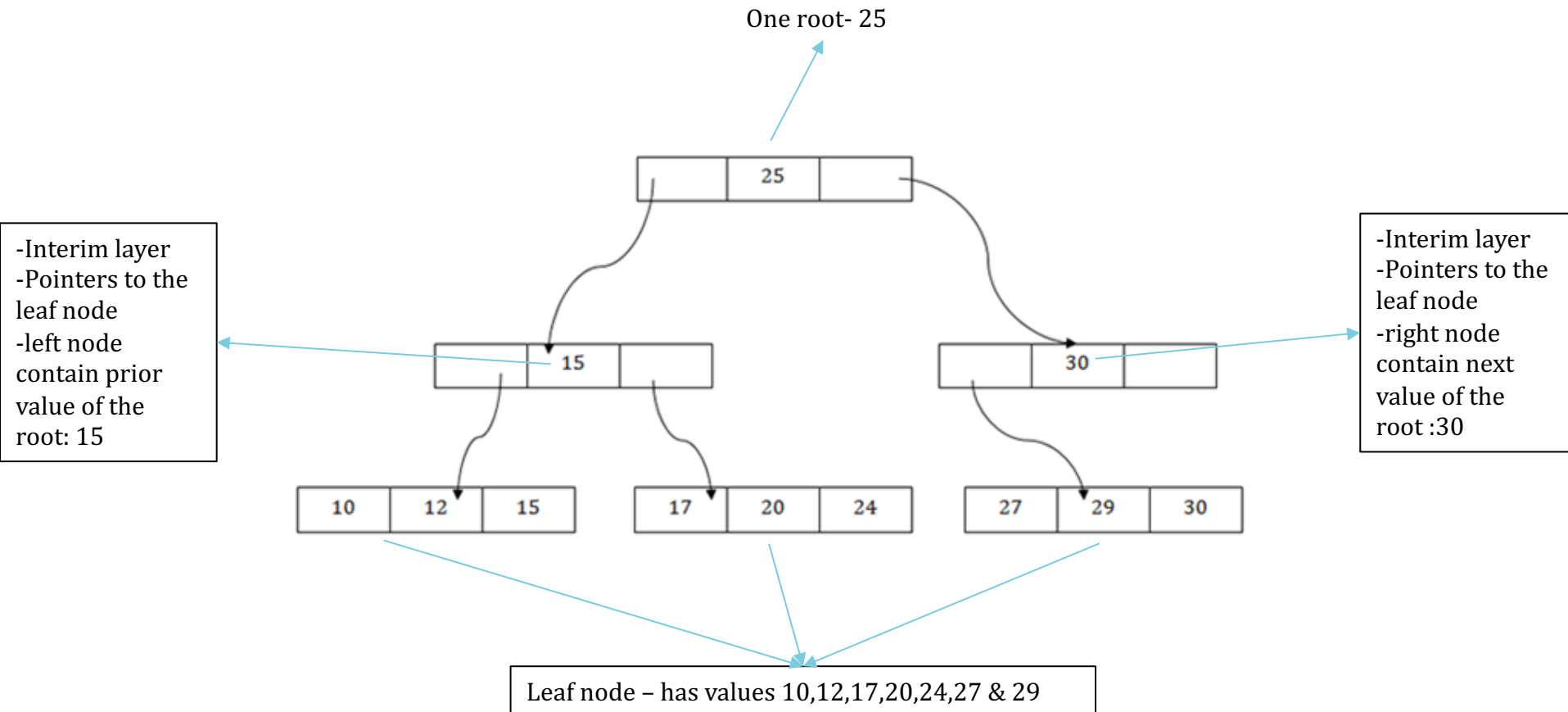


Other Primary File Storage and Organisation

- a. B-trees
- b. Files of mixed records (which implement relationships among records of different types physically as part of the storage structure)
- c. RAID (Redundant Array of Independent Disks) (a standard technique in large enterprises to provide better reliability and fault tolerance features in storage)
- d. Enterprise storage systems: SANs(Storage area networks), NAS (network attached storage), etc...

Other Primary File Storage and Organisation

□ B+ tree file organization





Indexing



Indexing

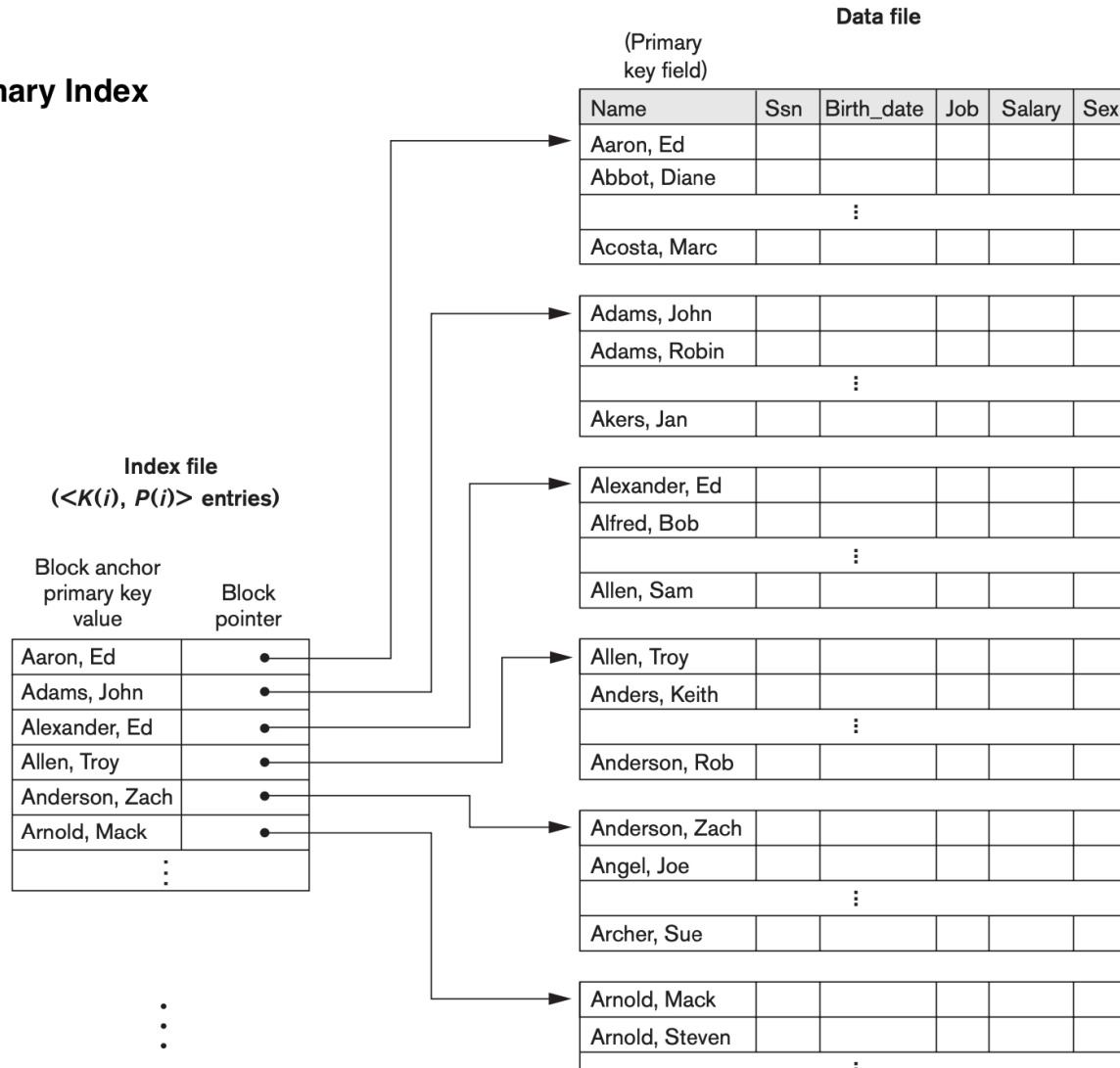
- ❑ Indexes are additional auxiliary **access structures** which are used to speed up the retrieval of records in response to certain search conditions.
- ❑ They enable efficient access to records based on the **indexing fields** that are used to construct the index.
- ❑ To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located.
- ❑ Most prevalent indexes are based on **ordered files** and use tree data structures to organize the index. They can also be constructed using hashing.

Types of Single-Level Ordered Indexes

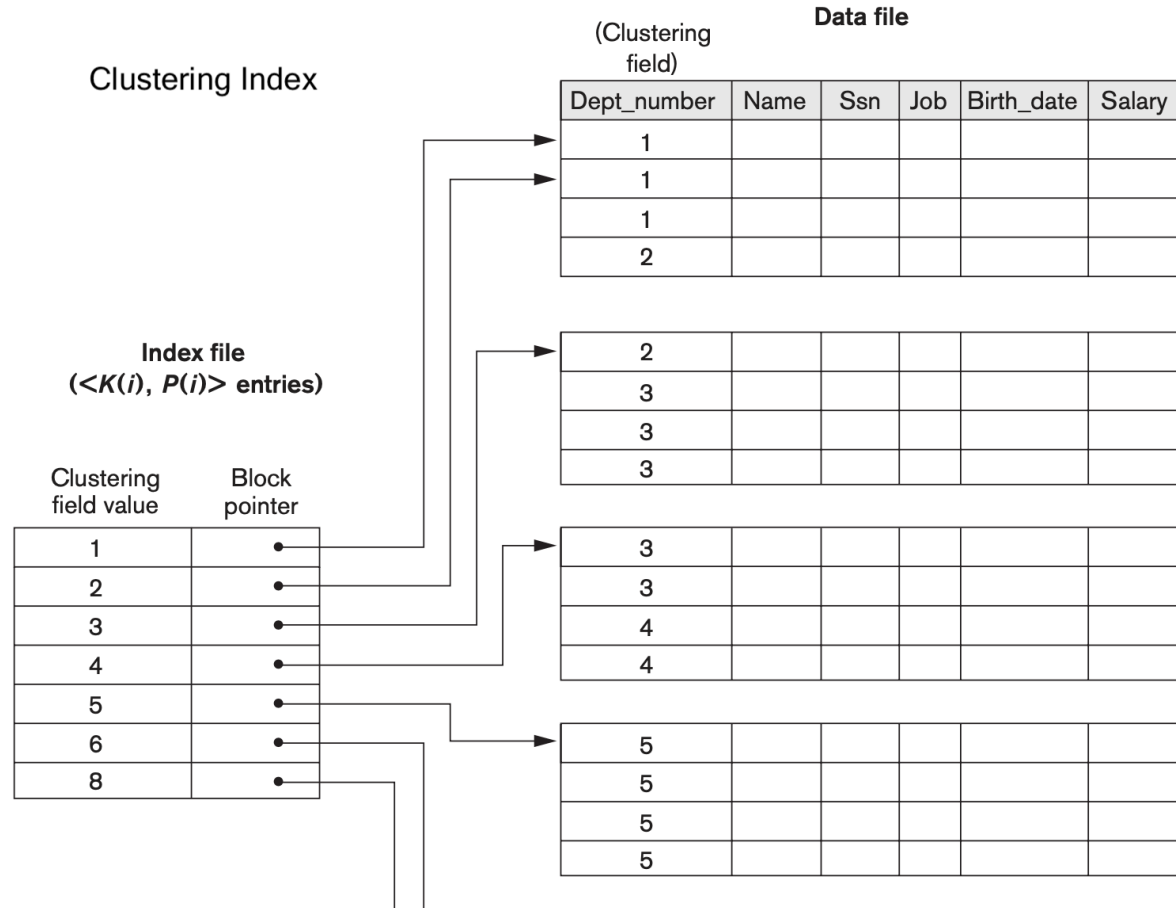
- Index access structure is defined on an **indexing field** and is ordered in order to use binary search on the index.
- **Types of ordered indexes:**
 - a. Primary Index: Index built on *ordering key field* of a file.
 - b. Clustering Index: Index built on *ordering non-key field* of a file.
 - c. Secondary Index: Index built on *any non-ordering field* of a file.
- The indexes can be dense (index record appears for every search-key value in the file) or sparse.

Primary Index - Example

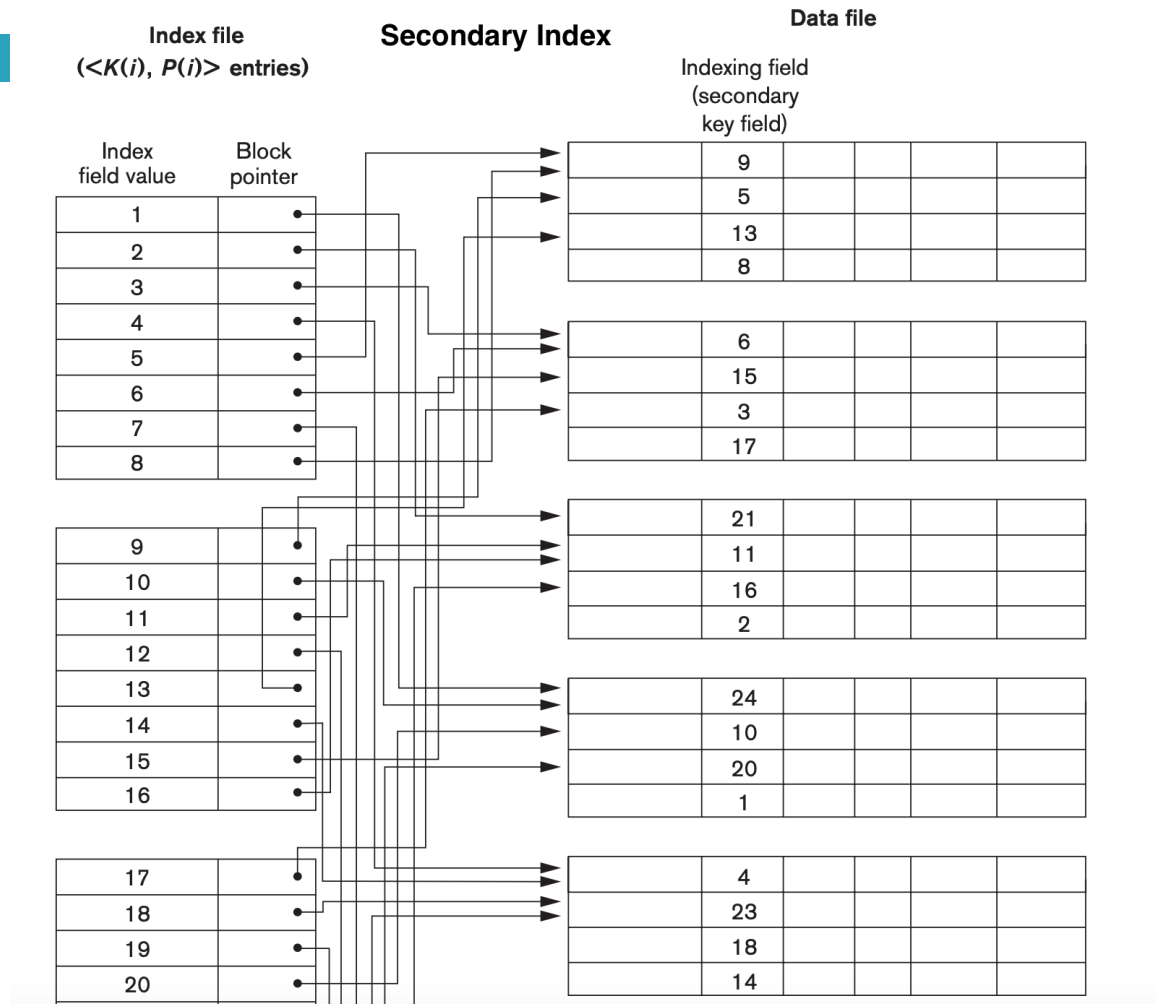
Primary Index



Clustering Index - Example



Secondary Index - Example



Dense/Sparse Index - Example

UP	•	→	UP	Agra	1,604,300
USA	•	→	USA	Chicago	2,789,378
Nepal	•	→	Nepal	Kathmandu	1,456,634
UK	•	→	UK	Cambridge	1,360,364

UP	•	→	UP	Agra	1,604,300
Nepal	•	→	USA	Chicago	2,789,378
UK	•	→	Nepal	Kathmandu	1,456,634
		→	UK	Cambridge	1,360,364

Summary of Single Level Indexes

Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no ^a
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records ^b or number of distinct index field values ^c	Dense or Nondense	No

^aYes if every distinct value of the ordering field starts a new block; no otherwise.

^bFor option 1.

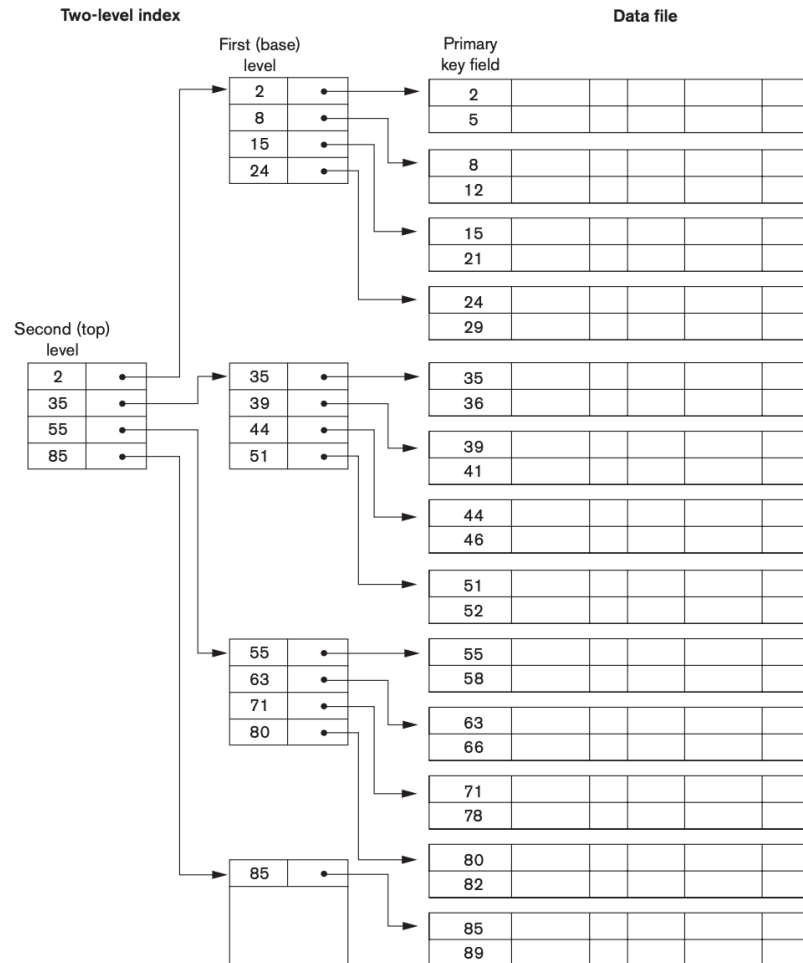
^cFor options 2 and 3.

MultiLevel Indexes

- ❑ Even with sparse indexes, the index file sizes may still grow and binary search might take longer.
- ❑ **Multilevel index** can be used to reduce the part of the index that we continue to search and there by reducing the search space making it much faster.
- ❑ A multilevel index considers the index file, referred to as the **first (or base) level** of a multilevel index, as an *ordered file* with a *distinct value* for each key; create a primary index for the first level which is called the **second level**, and repeat the proces for a **third level**.
- ❑

Two Level Index (ISAM)

A two-level primary index resembling ISAM (indexed sequential access method) organization.



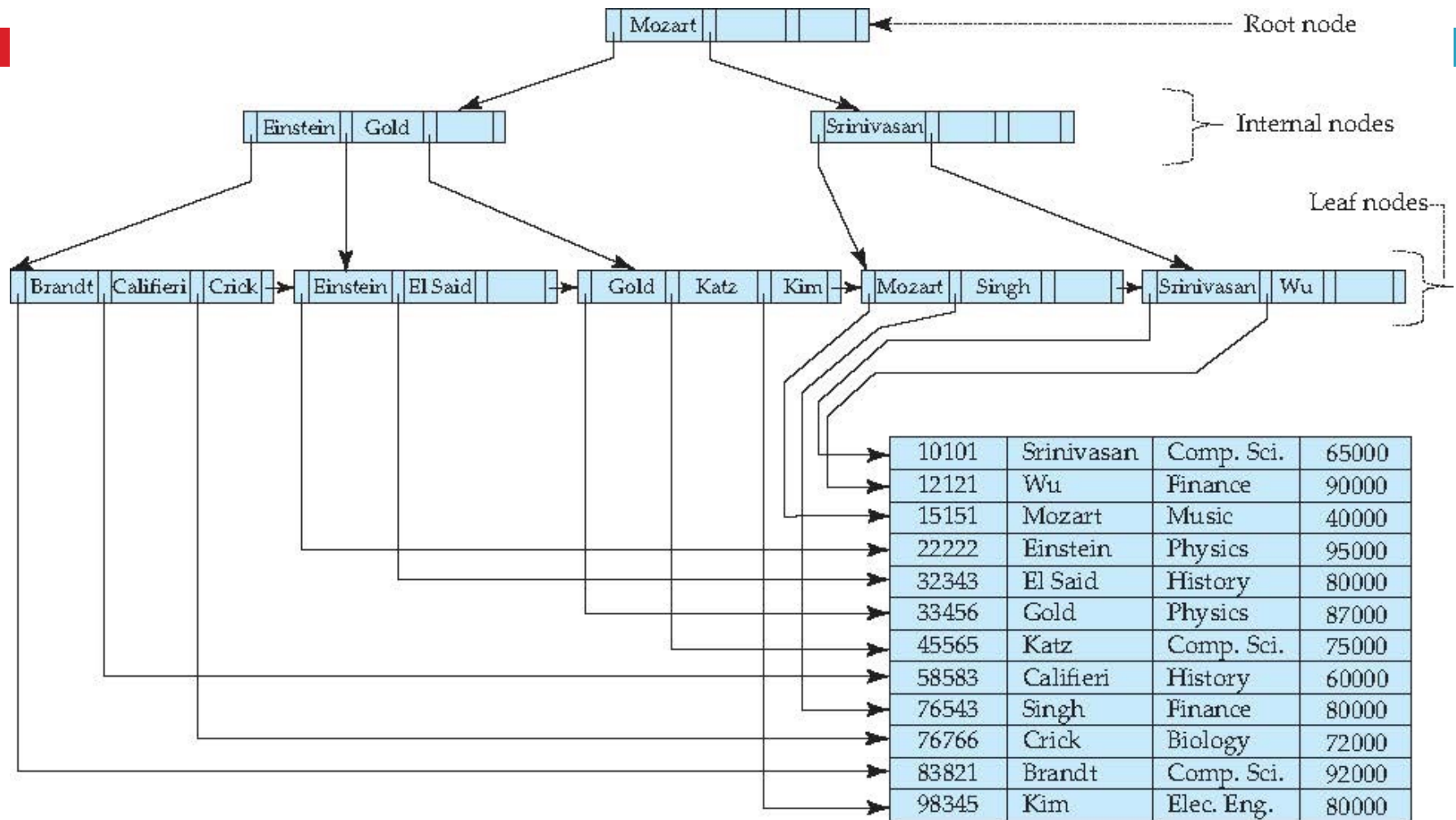
Indexed Sequential Access Method (ISAM)

- ❑ IBM's ISAM files – Ordered files with a multilevel primary/clustering index.
- ❑ Insertions: Handled using overflow chains at data file blocks.
- ❑ Deletions: Handled using deletion markers.
- ❑ Most suitable for files that are relatively static.
- ❑ If the files are dynamic, we need to go for dynamic multi-level index structures based on B+- trees.

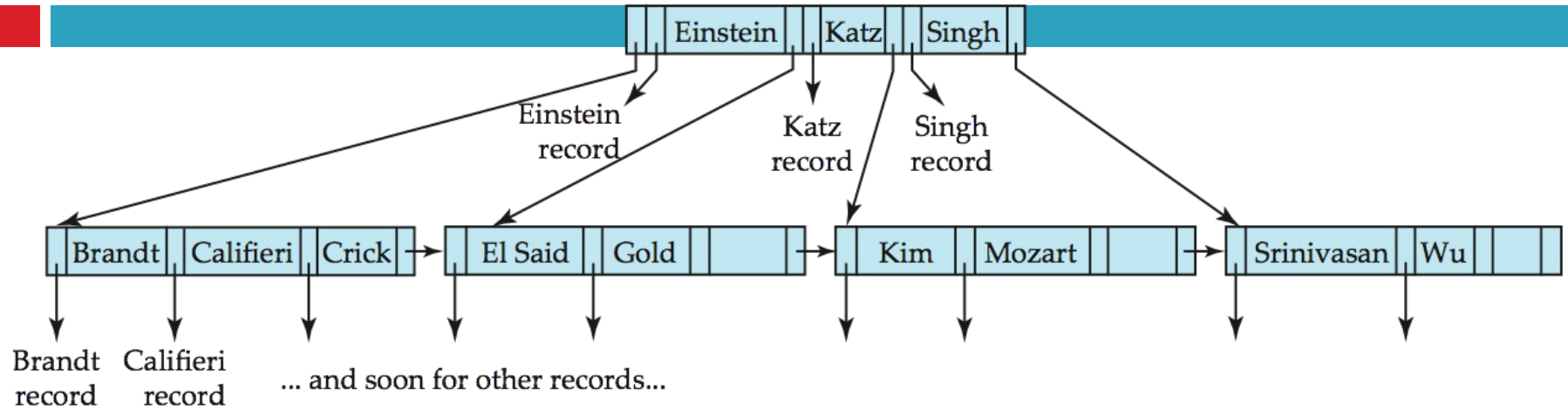
Other Data Structures for Multilevel Indexes

- ❑ Multilevel indexes can also be implemented using B-trees and B+-trees (which are dynamic structures that allow an index to expand and shrink dynamically).
- ❑ B+-trees can generally hold more entries in their internal nodes than can B-trees, so they may have fewer levels or hold more entries than does a corresponding B-tree.

B+- Tree



B- Tree Index File



Multiple Key Access

- Use multiple indices for certain types of queries.

- Example:

select *ID*

from *instructor*

where *dept_name* = "Finance" **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
 1. Use index on *dept_name* to find instructors with department name Finance; test *salary* = 80000
 2. Use index on *salary* to find instructors with a salary of £80000; test *dept_name* = "Finance".
 3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Composite Search Key - Example

- Suppose we have an index on combined search-key
- $(dept_name, salary)$.
- With the **where** clause
 where $dept_name = \text{"Finance"}$ **and** $salary = 80000$
the index on $(dept_name, salary)$ can be used to fetch only records that satisfy both conditions.
 - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
 where $dept_name = \text{"Finance"}$ **and** $salary < 80000$
- But cannot efficiently handle
 where $dept_name < \text{"Finance"}$ **and** $balance = 80000$
 - May fetch many records that satisfy the first but not the second condition

Selection Indices

- **Index scan** – search algorithms that use an index
 - ▣ selection condition must be on search-key of index.

Different types of scans:

Full table scan - A scan of table data in which the database sequentially reads all rows from a table and filters out those that do not meet the selection criteria.

Full Index scan - A scan of an index in which the database reads the entire index in order.

Index in SQL

- ❑ Create an index

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

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

- ❑ Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
 - ▣ Not really required if SQL **unique** integrity constraint is supported
- ❑ To drop an index

drop index <index-name>
- ❑ Most database systems allow specification of type of index, and clustering.

Design Decision on Indexing

- Performance of queries largely depends on indexes or hashing schemes.
- But indexes can also add overhead for insert/delete/update.
- Design questions to consider:
 - a. Whether to index an attribute:* attribute is a key or used by a query.
 - b. What attribute(s) to index on:* single or multiple.
 - c. Whether to setup a clustered index:* one per table.
 - d. Whether to use a hash index over a tree index:* has indexes do not support range queries.
 - e. Whether to use dynamic hashing:* appropriate to very volatile files.

Summary



- ❑ Recovery of a database
- ❑ Log based recovery
- ❑ Recovery for concurrent execution
- ❑ Checkpoints
- ❑ Backups
- ❑ File Organisation & Indexing

Lab Exercise



- ❑ Download Instructions from Moodle