Teste 1 Detailed Topics

Part 1 - RAID (Redundant Array of Independent Disks)

Why RAID?

- Disks are slow (compared to memory and CPU)
- Disks can fail, causing data loss

Hence a DBMS cannot rely on a single disk for performance or reliability.

What is RAID?

RAID organizes multiple physical disks into a single logical disk.

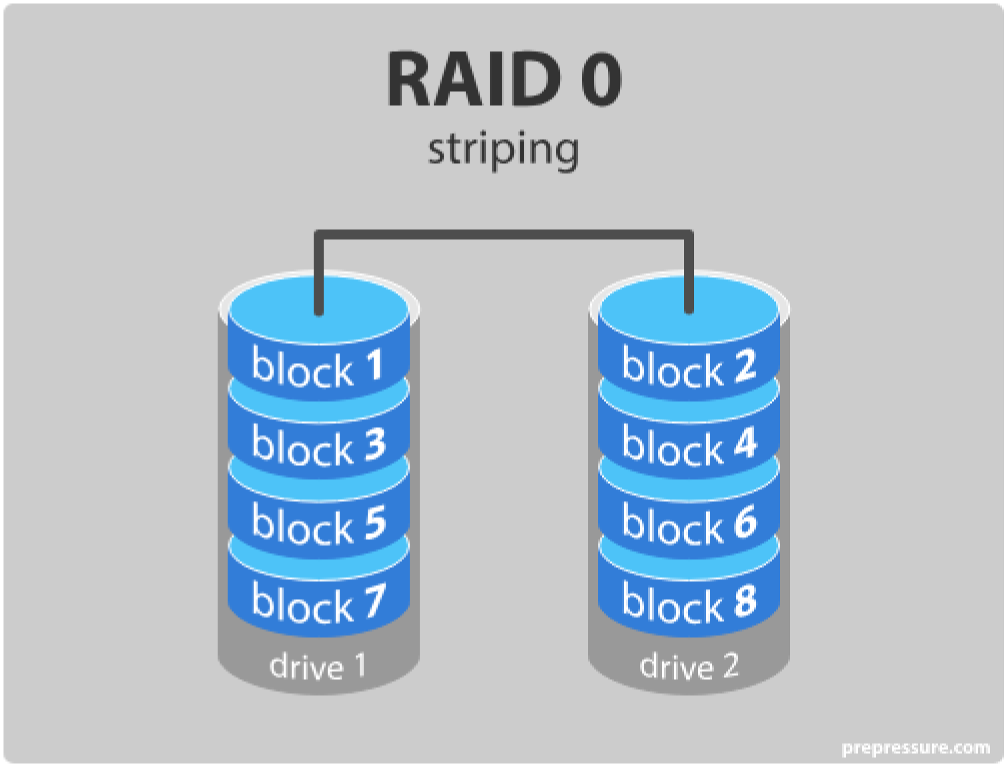
- The DBMS see one logical storage device
- RAID is transparent to higher layers
- Data is distributed across disks according to a RAID level

Two core ideas behind RAID

1. **Data striping**
    - Data is split into stripes
    - Stripes are distributed across multiple disks
    - Improves parallelism and throughput
2. **Redundancy**
    - Extra information is stored to recover from disk failures
    - Improves reliability and availability

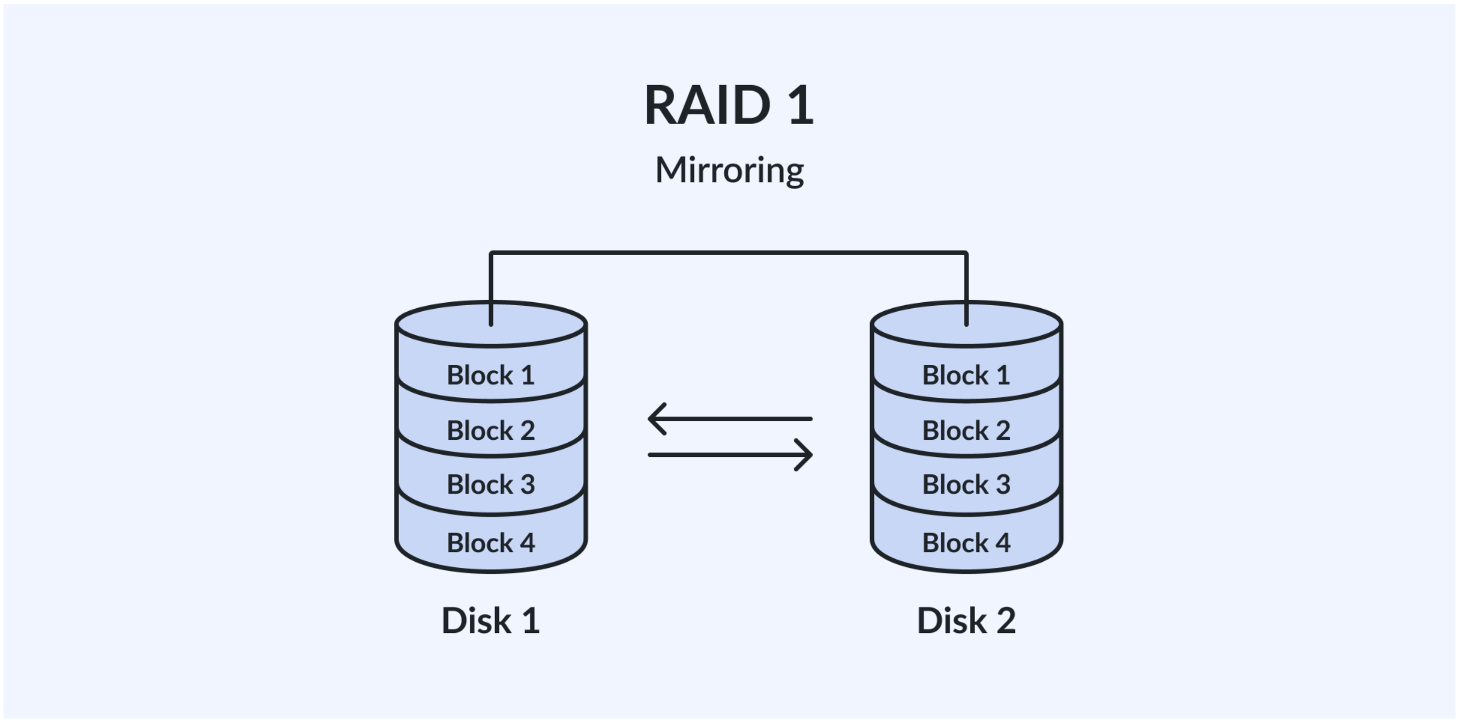
Common RAID levels

RAID 0 - Striping only (no redundancy)



- Improved performance
- no fault tolerance.
- One disk fails, all data is lost.

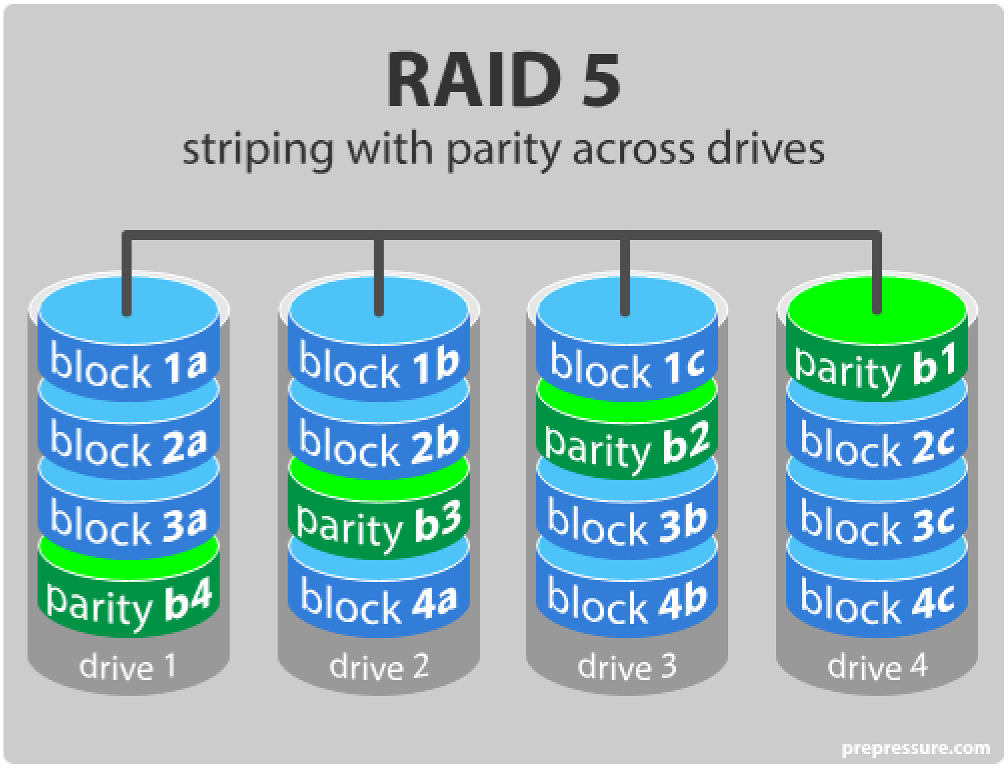
RAID 1 - Mirroring



Each disk has an exact copy on another disk.

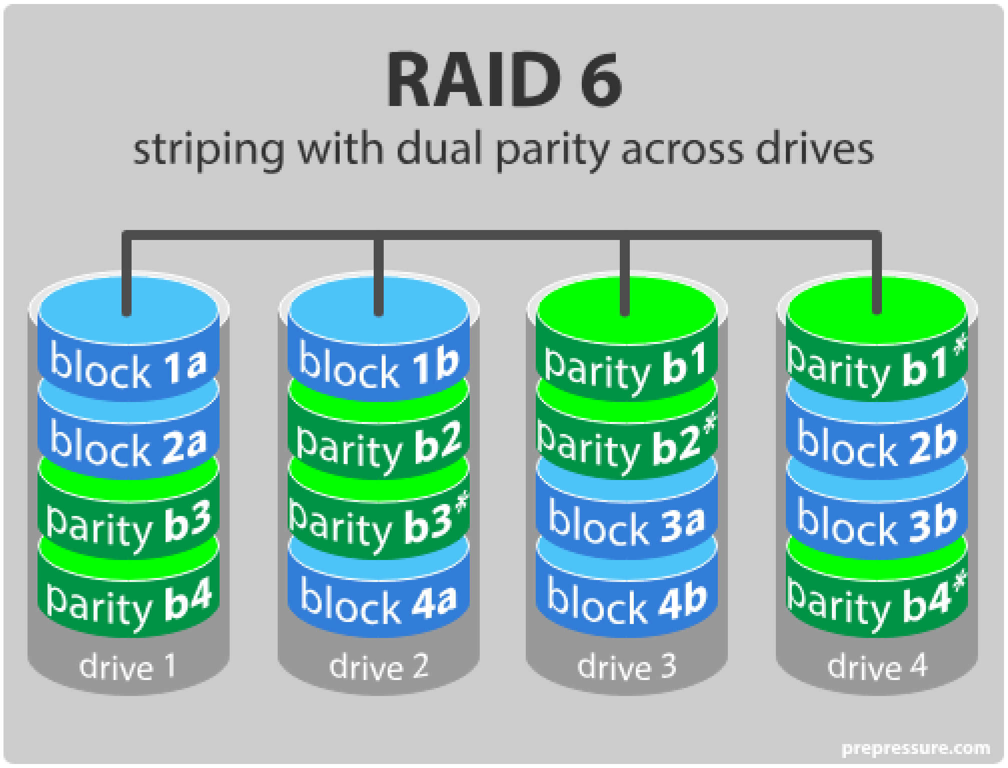
- High reliability
- Improved read performance
- Storage cost is doubled.
- Writes must be performed on both disks.
- Reads can be parallelized.

RAID 5 -Striping with distributed parity



- Data is striped across disks
- Parity information is also stored
- Parity is distributed, not centralized
- Can tolerate a single disk failure
- Better storage efficiency than RAID 1
- Write operations require updating data and parity.
- Small writes are more expensive due to parity updates.

RAID 6 - Double distributed parity



- Similar to RAID 5
- Stores two independent parity values
- Can tolerate two simultaneous disk failures
- Higher reliability
- Higher overhead for writes

RAID 6 is becoming increasingly important as disk sizes grow.
because disk rebuild times are longer, and a probability of a second failure during rebuild increases.

Comparison

RAID	Striping	Redundancy	Failures tolerated
RAID 0	Yes	No	0
RAID 1	No	Yes (mirror)	1
RAID 5	Yes	Yes (parity)	1
RAID 6	Yes	Yes (double parity)	2

What RAID does not do

RAID doesn't understand tuples, files or indexes.
It works with Blocks, Stripes and Disks.

Part 2 - Physical Data Storage

Why physiical storage matters

- A DBMS manages data stored on disk
- Disk access is orders of magnitude slower than memory access
- Therefore, DBMSs:
  - Organize data in blocks (pages)
  - Transfer data block byu block, not tuple by tuple
- This is why almost all cost formulas in the course are expressed in numbers of blocks transfers and seeks, not rows.

Blocks

- A block (or page) is the unit of data transfer between disk and main memory
- Typical block size: 4KB, 8KB, 16KB

Important consequences stated in the slides:

- The DBMS never reads a singles tuple directly
- When a tuple is accessed, the entire block containing it is read into memory

Tuples

- A tuple (or row) is a single record in a table
- Database records (turples) can be of variable length or fixed length
- A block does not store a fixed number of tuples
- The DBMS must use a record organization mechanism inside the block
- Common organizations:
  - Slotted page (fixed-length or variable-length records)
  - Heap file (variable-length records)

Calculating number of tuples per block

- The number of tuples depend on:

    - Block Size
    - Average tuple size
    - Block-level overhead (metadata)
- Formula:
    - T = ⌊ B / R ⌋
        - T = number of tuples per block
        - B = block size (in bytes)
        - O = block-level overhead (in bytes)
        - R = average tuple size (in bytes)

Numbers of blocks of a table

- The size of a relation is measured in number of blocks
- br = number of blocks in relation r

br = ⌈ Number of tuples x Tuple size / Block size ⌉

Main memory (buffer pool)

- The DBMS uses main memory to cache disk blocks
- Only a limited number of blocks fit in memory at a time
- **M = number of blocks that fit in main memory**
- If a relation fits in memory (br ≤ M), operations on that relation are much faster.

Disk access costs

1. **Seek time** - time to position the read/write head over the correct track.
2. **Transfer time** - time to read/write one block once the head is in position. denoted as tT.

- The total i/o cost is modeled as: number of seeks, number of block transfers.

Part 3 - File organization (Inside a block)

- A file is a collection of disk blocks, each file stores the tuples of a relation, blocks of a file are not necessarily stored contiguously on disk.
- **Logical adjacency != Physical adjacency**
    - Two blocks that are logically adjacent in a file may be physically stored in different locations on disk.

Tuples can be inserted, deleted and updated, can have variable length, etc.
Therefore, the DBMS need a structure that:

- Avoids moving tuples unnecessarily
- Allos space reuse
- Supports efficient access

Slotted-page organization

A slotted page is a block organized into:

1. A header

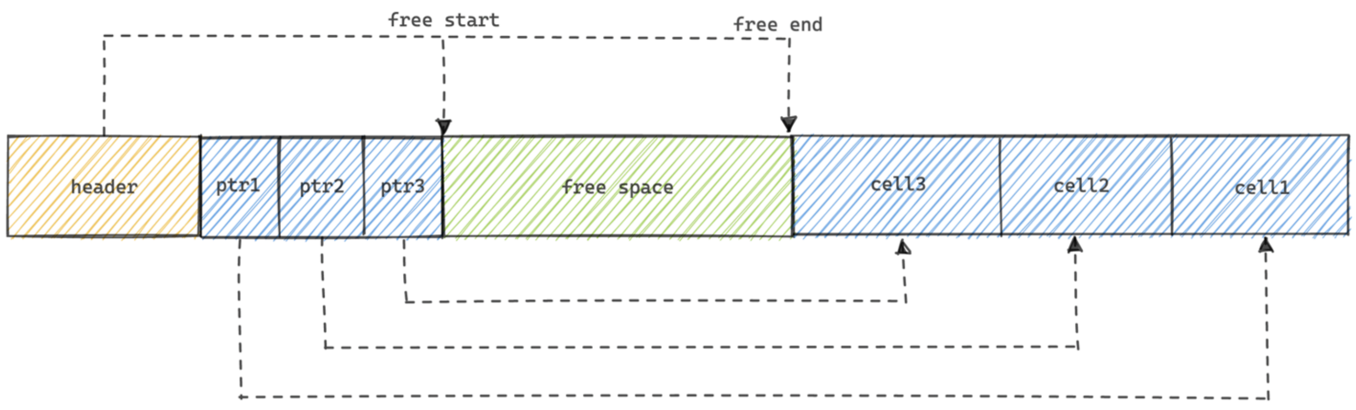
- fixed-length area that contains metadata about the block

2. A slot directory

- each slot contains the offset of a tuple inside the block or a marker indicating that the slot is empty.
- **slots are fixed-length, slot numbers don't change when tuples move.**

3. A tuple area - variable-length area that stores the actual tuples.

- Tuples are store from the end of the block towards the slot directory.



Why does this solve our problems?

- If a tuple changes size, the tupls can be moved inside the block, only the slot offset needs to be updated.
- External references use: (block id, slot number) pairs, so they don't change when tuples move.
- Avoiding updating indexes or pointers in other blocks.
- On deletion, the slot is marked as free, its space becomes part of the free space region.

- Allows efficient insertions and reduced fragmentation.
- **Tuples are not stored in any particular order inside the block.**
- Ordering must be provided by file organization or indexes.
- Slotted pages are used both for heap files, sorted files and indexed files, since they are independent of query language or index type.

Part 4 - File organizations

Heap (Unordered) File organization

Definition: A heap file is a collection of blocks storing tuples in no particular order.

Operations:

- **Insertion:** add the tuple to any block with enough free space, if none exist allocate a new block and add it to the file.
- **Deletion:** remove tuple, free space is recorded in the block, no reordering performed.
- **Search:** requires a **linear scan** of all blocks in the file, unless an index exists.
- Very efficient insertions
- Very inefficient searches
- No odering guarantees

Heap files are suitable when access is mostly via indexes.

Sequential (sorted) file organization

Definition: Tuples are stored sorted on a search key, and the file is physically ordered by that key.

Operations:

- **Insertion:** Tuple must be placed in the correct sorted position, may require shifting tuples or creating overflow blocks.
- **Deletion:** Remove tuple, may leave holes, does not automatically restore compactness.
- **Search**: Efficient for equality search and range queries using binary search at block level.
- Performance degrades as insertions and deletions occur.
- Therefore periodic reorganization may be required.
- This motivates the introduction of B +-tree later.

Multitable clustering file organization

Definition: Tuples from multiple relations are stored together in the same file, based on a common clustering key.

- Used to optimize join operations between the clustered relations.
- Reduces the number of disk accesses during joins.
- Worse performance for single-relation queries.

Hashed file organization

Definition: A hash function is applied to a search key, the result determines the bucket (block), each bucket corresponds to one or more blocks.

Operations:

- **Insertion:** Tuple is placed in its bucket, if full, use overflow blocks.
- **Deletion:** Remove tuple from its bucket, space reclaimed locally.
- Search: efficient for equality conditions, requires computing the hash value.

Limitations:

- No support for range queries
- Hashing destroys ordering

This is explicitly contrasted with ordered files and B+-trees indexes.

Comparison of file organizations

Organization	Order	Insert	Search	Range queries
Heap	None	Fast	Slow	No
Sequential	Sorted	Slow	Fast	Yes
Hashed	Hash-based	Fast	Fast (equality)	No
Multitable	Clustered	Slow	Fast (joins)	Limited

Part 5 - Indexing: Basic concepts

Why indexing?

- Searching heap files requires a linear scan, which can be expensive.
  therefore, **indexes are auxiliary data structures used to speed up access to data**
- Can be compared to a library catalog, where you locate a book without scanning all shelves.

What it is

- A data stucture that allows efficient retrieval of records based on the values of one or more attributes.
- An index is stored in a separate file
- It is smaller than the data file
- It contains index entries, not full tuples.

Structure

(search-key value, pointer)

- The search-key value is the attribute(s) used for lookup
- Pointer is a reference to a record or a block or a bucket.

The search key is **not necessarily** the primary key.

Search key vs primary key

- Search key: attribute(s) on which the index is built
- Primary key: attribute(s) that uniquely identify a tuple in a relation

A search key may or may not be unique.

What are they evaluated on (metrics)

- Access types supported efficiently (equality, range queries)
- Access time
- Insertion and deletion time
- Space overhead

These criteria are used to compare ordered vs hash indexes and justify index choices in exams.

Two fundamental index types

1. **Ordered indexes** - maintain entries in a sorted order based on the search key.
   - Support efficiently equality search and range queries and ordered retrieval.
   - The basis for B+-tree indexes.
2. **Hash indexes** - use a hash function to map search key values to index entries.
   - Efficient for equality search
   - **Hash indexes do not support range queries efficiently**

Indexes improve access but increase maintenance cost.

Cost of indexes

Every insertion, deletion, or update of a record may require updating all indexes on that relation

therefore indexes speed up reads but slow down writes.

Part 6 - Ordered indexes

In an ordered index, index entries are stored sorted on the search-key value.

Important consequences:

- Index entries have a total order
- Searching can exploit that order
- Range queries become efficient

Two roles of ordered indexes:

1. **Clustering index** - determines the physical order of data in the data file.
   - There is at most one clustering index per relation.
   - Data records are stored in the same order as the index entries.
2. **Non-clustering index** - does not affect the physical order of data.

Clustering index (also called primary index)

A clustering index is an index whose search key determines the physical order of the data file.

The data file is sequentially ordered on the search key
The search key is often the primary key, but not necessarily.

THERE CAN BE ONLY ONE CLUSTERING INDEX PER FILE. because a file can only be physically ordered in one way.

Secondary (non-clustering) index

A secondary index is an index whose search key specifies an order different from the sequential order of the file.

The data file is NOT ordered on the search key.
Index entries point to records scattered throughout the data file.

Sequential scans using a secondary index are expensive.

Each tuple may be in a different block, causing many i/o operations.

Dense vs sparse indexes

Density is a property of ordered indexes.

- **Dense index:** A dense index contains an index entry for every search-key value in the file.
- If search keys are unique, one entry per record.
- If search keys are not unique, one entry per distinct search-key value.
- larger index, faster lookups, more maintenance overhead.

- **Sparse index:** A sparse index contains index entries for only some of the search-key values in the file.
- **Sparse indexes are applicable only when the data file is sequentially ordered on the search key**

This ties sparse indexes directly to clustering indexes.

To find a record with search key k:

- Find the index entry with the largest search-key value < K
- Start a sequential scan from the pointed block

- Smaller index, less maintenance overhead, slower lookup.

Comparison dense vs sparse

Property	Dense	Sparse
Index entries	One per key	Some keys only
Index size	Larger	Smaller
Lookup speed	Faster	Slower
Maintenance	Higher	Lower
Requires sorted file	No	Yes

Secondary indexes are always dense

Data file is not ordered on the search key, therefore sparse entries would not be sufficient to locate all records.

Limitations

- Indexes improve search but:
  - Increase storage overhead
  - Increase maintenance cost on insertions, deletions, updates

Part 7 - B+-tree index files

Why B+-trees?

- Indexed-sequential files degrade as the file grows.
- This motivates B+-trees.

What is a B+-tree?

A B+-Tree index is an ordered index structure that:

- Automatically reorganizes itself
- Supports efficient insertion and deletion
- Maintains balanced height

- **Reorganization is local**
- No global reorganization of the entire file is needed.

Core structural properties of B+-trees

Let n be the maximum number of pointer in a node.

Tree Properties:

- All paths from the root to a leaf have the same length
- The tree is always balanced

Internal (non-leaf) nodes:

- Contain between $\lceil n/2 \rceil$ and n pointers (except root)
- Contain search keys and pointers to child nodes

Leaf nodes:

- Contain between $\lceil (n-1)/2 \rceil$ and n-1 search-key values
- Contain:
    - Search-key values
    - Pointers to records (or buckets)
- Leaf nodes are linked together in search-keyh order.

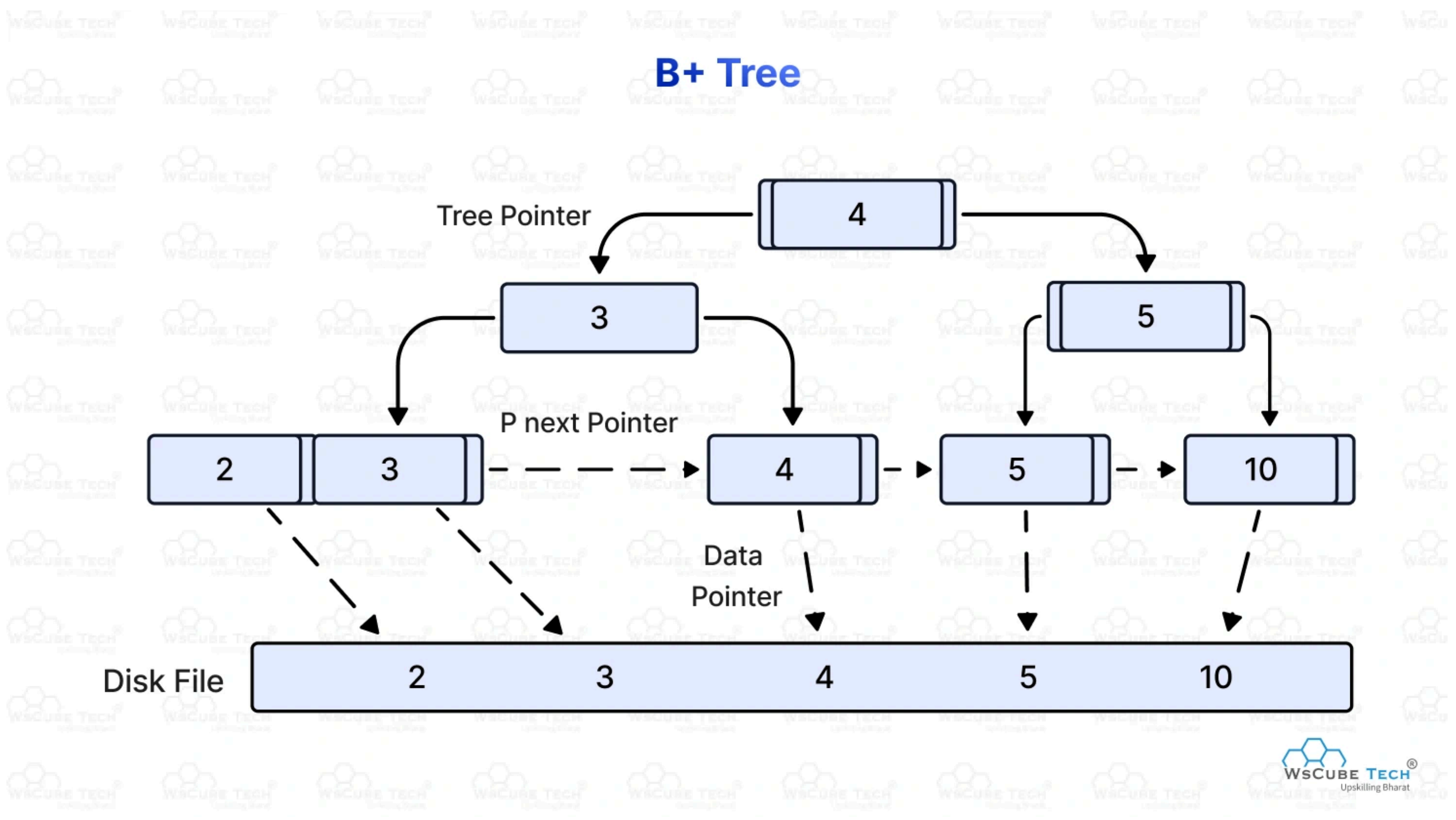
Root node:

- **If its not a leaf, contains at least 2 children.**
- **If its a leaf, it may have between 0 and n-1 values.**

Logical structure of a node

A typical node has:

- Search keys: K1, K2, ..., Kn-1
- Pointers: P1, P2, ..., Pn
  Meaning:
- All keys in subtree P1 are < K1
- Keys in subtree Pi are ≥ Ki-1 and < Ki for 2 ≤ i ≤ n-1
- Keys in subtree Pn are ≥ Kn-1



- All actual data pointers are stored in the leaf nodes.
- Internal nodes only store search keys and pointers.
- This is what distinguishes B+-trees from B-trees.

Sequential access via leaf linkage

- Leaf nodes are linked together in search-key order.
- This allows efficient range queries
- sequential access without returning to the root.

Height grows logarithmically with the number of search keys.

If there are K search-key values, the height is at most $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$

This is why very few nodes are accessed during searches.
Disk I/O cost is low.

- Typically, a node is the size of a disk block.
- For a 4kb block, a node can contain around 100 search keys

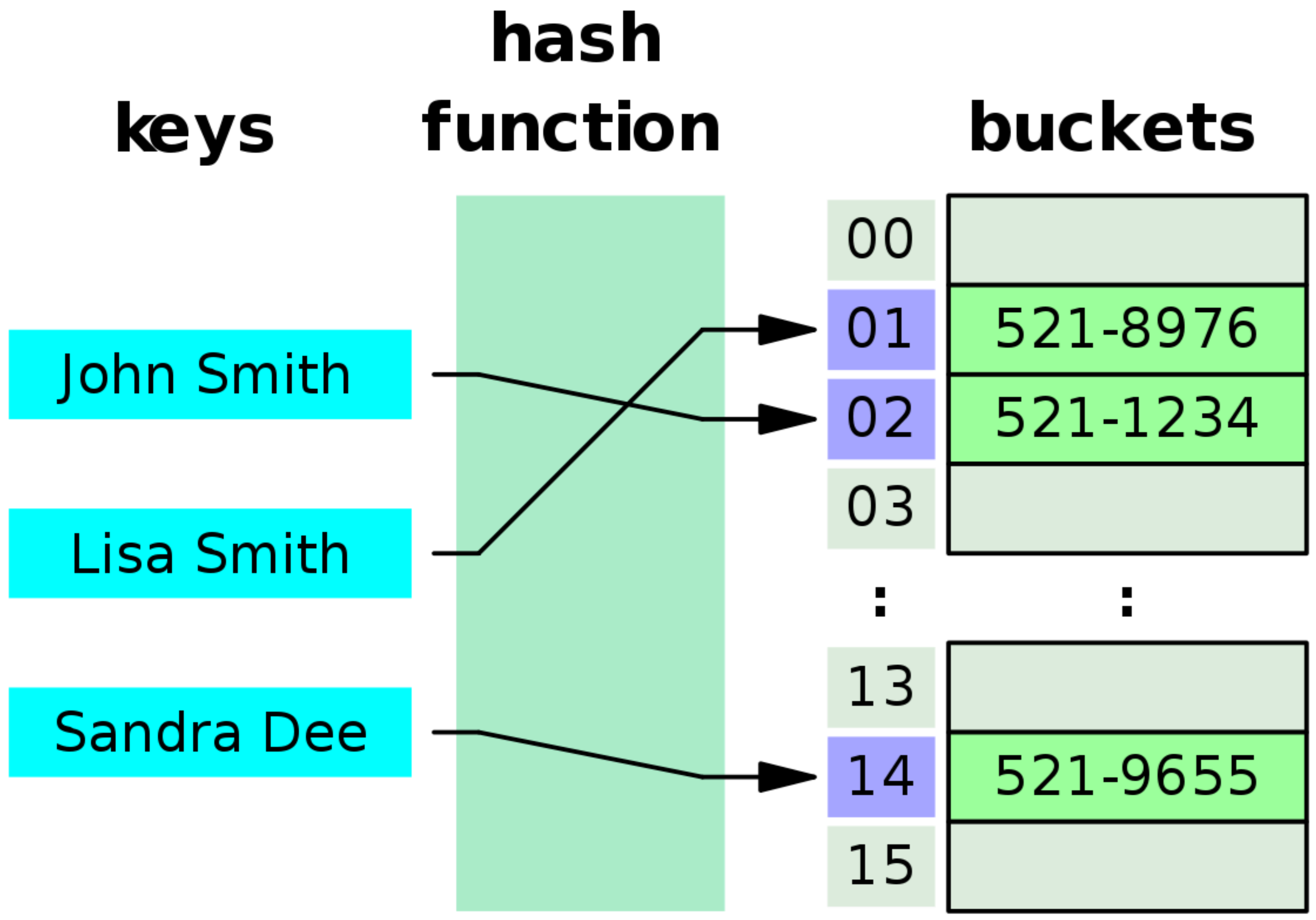
Non-unique search keys

- A b+-tree may store a composite key (search-key, record-id)

- Or points to buckets of records
  The handling of duplicates preserves ordering and the correctness of range queries.

Part 8 - Hash indexes

- Hash indexes are an alternative optimized for equality lookups.
- They use a hash function to distribute search-key values uniformly across buckets.



Static vs dynamic hashing

- Static hashing has a fixed number of buckets.
- Dynamic hashing allows the number of buckets to grow and shrink as needed to **reduce overflow chains**.

Hash indexes are appropriate when:

- queries use only equality conditions.

- range queries are not required.
- These are not a general replacement for ordered indexes.

Comparison ordered vs hash indexes

Feature	Hash Index	Ordered Index (B+-tree)
Equality search	Very efficient	Efficient
Range queries	Not efficient	Efficient
Ordering	No	Yes
Maintenance	Simple	More complex

Part 9 - Query processing

- Query processing involves translating a high-level query into a sequence of low-level operations that retrieve the required data efficiently.
- **SQL is declarative, so the DBMS must determine how to execute the query.**

This is a multi-stage process:

1. Parsing and translation
2. Optimization
3. Evaluation (execution)

Starting with evaluation.

Query evaluation

Logical operators:

Selection (σ)
Projection (π)
Join (⋈)

Physical operators:

- Implementation of logical operators
- Example:
- Different join algorithms

- Different access methods
- **A single logical operator may have multiple physical implementations.**

Operator-at-a-time vs pipeline processing

Operator-at-a-time (materialization):

- Each operator:
  - Reads its entire input
  - Produces its entire output
  - Stores the result (often on disk)
- The next operator then reads that result.
- High I/O cost, simple to implement.

Pipeline processing:

- Operators produce output tuples one at a time
- Output of one operator is passed directly to the next
- Intermediate results may not be materialized.
- This reduces I/O cost, but is more complex to implement.

Not all operators can be pipelined.

Some operators require all input before producing output (e.g., sorting, some aggregation operations). Blocking operators force materialization and break pipelines.

Choice of access method affects performance significantly.

EARLY SELECTION AND PROJECTION IS BENEFICIAL TO REDUCE DATA VOLUME EARLY IN THE PLAN.

Part 10 - Join algorithms

A join as two input relations:

Outer relation R and inner relation S.
And a join condition Equality or general condition

The terms outer and inner are algorithmic, not semantic.

Simple nested-loop join (NLJ)

The nested-loop join compares each tuple of the outer relation with each tuple of the inner relation

- For each tuple r ∈ R:
  - For each tuple s ∈ S:
    - Test join condition
    - If satisfied, output result

Very expensive, barely used in practice.

Block nested-loop join (BNLJ)

Instead of comparing tuple by tuple, compare block by block.

The outer relation is read block by block, and each block is compared with all blocks of the inner relation.

- Read a block of R into memory
- For each block of S:
  - Compare tuples in the blocks
  - Repeat for all blocks of R

Still expensive if both relations are large
Heavily dependent on which relation is chosen as outer and on the available memory.

The smaller relation should be the outer relation to minimize total I/O cost.
Because they are reread fewer times, and the i/o cost is reduced.

Index nested-loop join (INLJ)

If an index exists on the join attribute of the inner relation
The DBMS can avoid scanning the entire inner relation.

For each tuple of the outer relation, an index is used to retrieve matching tuples from the inner relation.

Efficient when:

- outer relation is small
- Inner relation has an apprpriate index.

Poor performance if index access is expensive or unselective.

Merge join (Sort-Merge join)

If both relations are sorted on the join attribute, they can be joined efficiently by scanning them in order.

Merge join requires both input relations to be sorted on the join attribute.

- Scan both relations sequentially
- Compare current tuples
- Advance pointers based on comparison
- Efficient for large relations
- Supports range joins
- May require sorting first

SORTING IS A BLOCKING OPERATION

Hash join

If the join condition is equality, hashing can be used to partition relations and reduce comparisons.

One relation is hashed into buckets, and the other relation probes those buckets.

- Build phase: Hash the smaller relation R into buckets in memory.
- Probe phase: For each tuple in S, compute hash and probe corresponding bucket in R.
- Requires memory for hash table, does not support range joins, but efficient for large relations with equality joins.

T07

Cost is measured in terms of **number of block transfers** and **number of seeks**.

br = number of blocks in relation r
bs = number of blocks in relation s
M = number of blocks that fit in main memory

define tT (transfer time per block) and tS (seek time).
Total cost example form: b* tT + S * tS.

- Buffers may already contain data; memory available varies due to concurrency; worst-case estimates assumes nothing is initially buffered.

Selection operation

Bitmap index scan - 1 bitmap per query, 1 bit per page

SELECT * FROM R WHERE age = 20 AND city = 'Lisbon';

- Secondary indexes exist
- Many matches
- DBMS doesn't know selectivity

What the DBMS does:

1. Scan index -> build bitmap (1 bit per block)
2. Mark blocks that contain matches
3. Do one linear scan, but read only marked blocks

Why is this efficient?

- Avoids random I/O
- Avoids fetching same block many times
- Adapts:
  - Few bits -> behaves like index scan
  - Many bits -> behaves like file scan

Never behaves very badly compared to best alternative.

What's a bitmap

A bitmap is an array of bits

Bitmap indices - Where each distinct value of an attribute has a bitmap.

- designed for efficient querying on multiple keys, for attributess with few distinct values (gender, country, income level, etc.)
- Records are assumed numbered.
- For an attribute A, you keep one bitmap per value v of A.
- In bitmap bitmap-A-v, bit i is 1 if record i has A=v, else 0.

Why need a DELETED bitmap and a NULL bitmap?

Deleted / existence bitmap, marks whether a record is deleted or not, preventing deleted slots from appearing in query results.
NULL bitmap intersects with other bitmaps to exclude NULL values from query results.