Storage & Indexing

Storage Hierarchy

- Databases rely on multiple layers of storage, each with different speed, size, and cost:
  - CPU Registers, Caches, DRAM: Fast, expensive, volatile.
  - SSD, HDD, Network Storage, Tape Archives: Slower, cheaper, non-volatile(Lecture 7).

Physical Database Design

- Databases are stored primarily on disk, with data movement between volatile (memory) and non-volatile storage managed by the DBMS.
- Random access on disk is slower than sequential access, so the DBMS aims to maximize sequential access and minimize random disk operations(Lecture 7).

Disk-Based Architecture

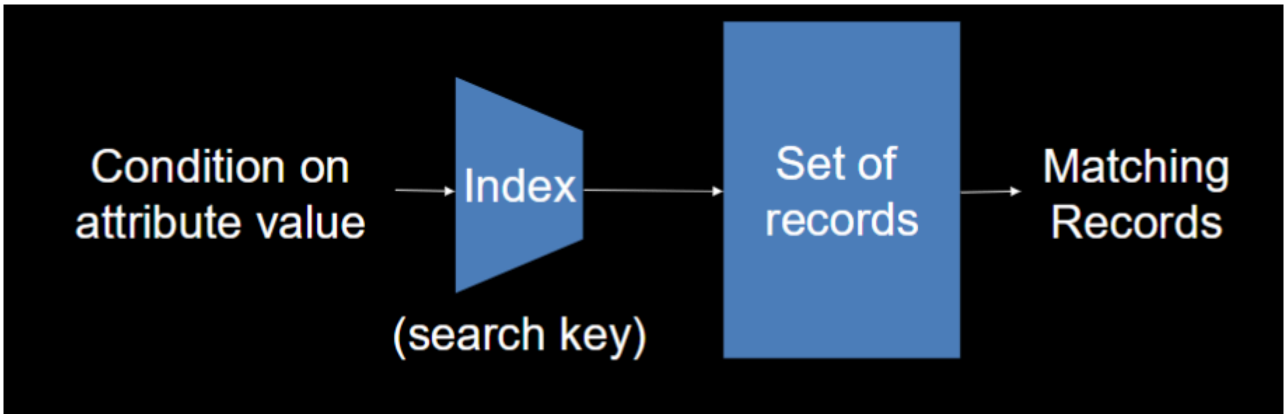
- DBMS assumes non-volatile disk as the primary storage for databases, and moves data to/from volatile memory as needed(Lecture 7).

System Design Goals

- Manage databases larger than available memory.
- Optimize disk access to avoid performance stalls.
- Use memory mapping (`mmap`) to map file contents into program memory, but avoid depending on OS virtual memory, which can cause issues such as I/O stalls(Lecture 7).

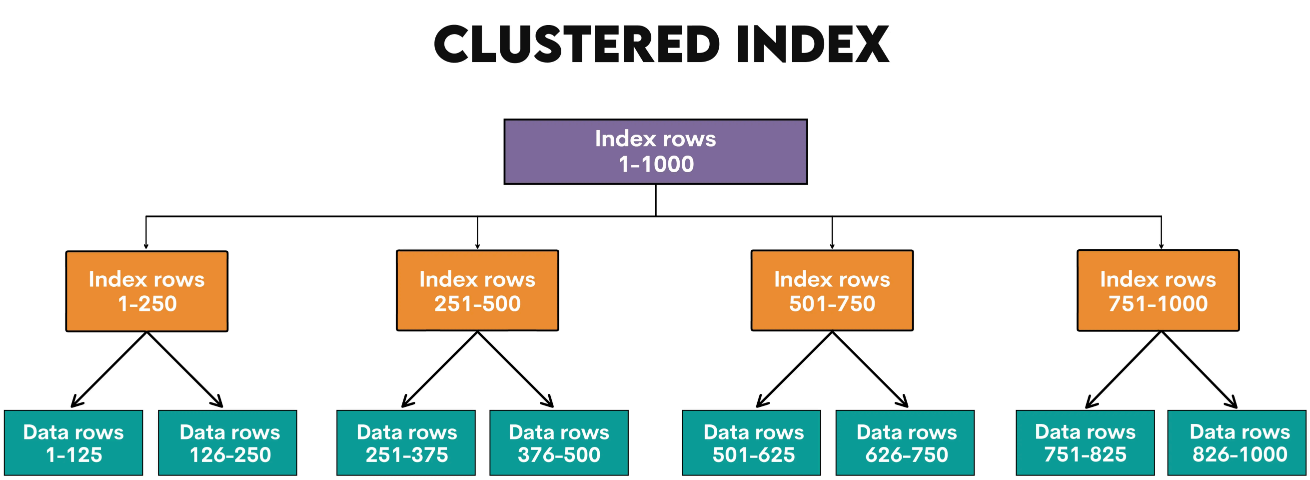
Indexing

- **Full Table Scan**: Reads all rows in a table to find matching records. Efficient for large result sets.
- **Point Queries**: Searches for a specific value (e.g., `SELECT * FROM person WHERE birthdate='1975-02-06'`).
- **Range Queries**: Searches for a range of values (e.g., `SELECT * FROM person WHERE birthdate BETWEEN '1975-02-01' AND '1975-02-28'`).
- **Indexes** improve query performance by creating a structure that maps attribute values to row locations.



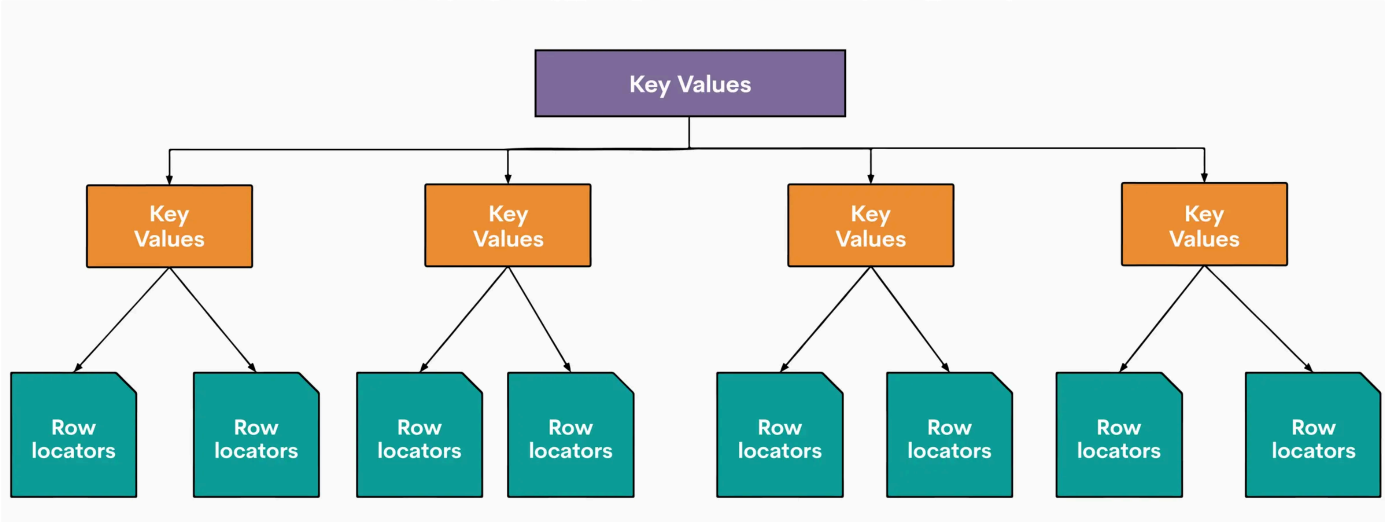
Types of Indexes

- **Clustered Index**: Data stored in the same order as the index. Efficient for point and range queries. There can only be one clustered index per table.



- **Unclustered (Non-clustered) Index**: Additional indexes that can exist alongside the clustered index. Efficient for point queries with high selectivity.

NON-CLUSTERED INDEX



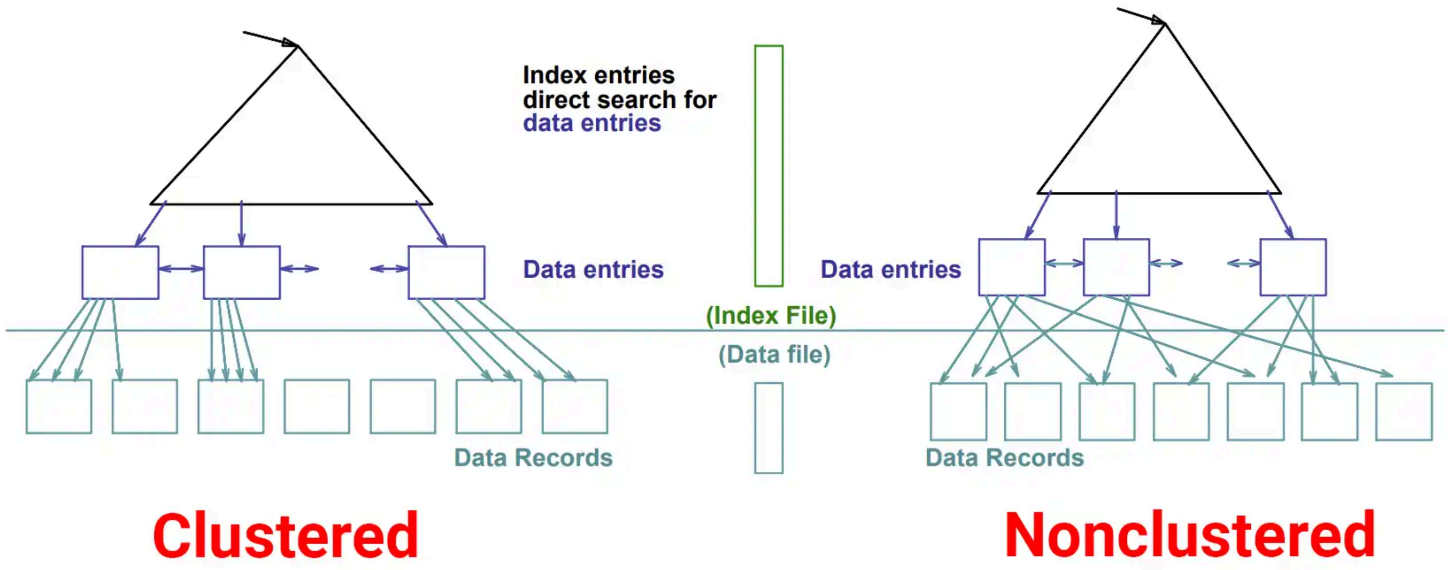
- **Covering Index**: An index that includes all attributes in a query, reducing the need for disk access.

```
SELECT COUNT(*) FROM movie WHERE year=1948;
CREATE INDEX movieyear ON movie(year);
/* Multi-Attribute indexes */
SELECT name FROM person WHERE height=170;
CREATE INDEX phn ON person(height,name);
```

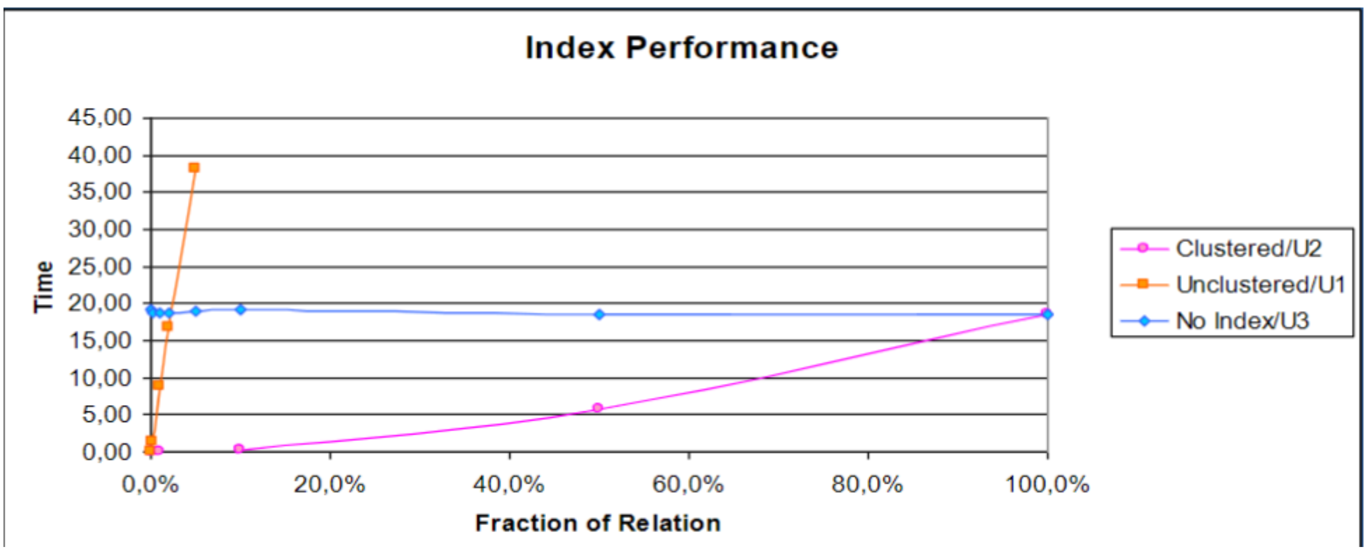
Complexity of clustering

To retrieve M records, where M is small:

- Clustered: Probably one disk read
- Unclustered: Probably M random disk reads
- Covering: Definitely need 0 disk reads
  To retrieve M records, where M is large:
- Clustered: Probably M/records_per_page sequential disk reads
- Unclustered: Up to M random disk reads
- Covering: Definitely need 0 disk reads
  We still need to read the index itself – same for both!



Performance of clustering



Techniques for Indexing

- **Hashing**: Uses a transformation algorithm for equality queries.
- **Tree Search**: Suitable for range queries and provides a versatile method for indexing.

Choosing Indexes

- Indexes should be chosen based on query patterns, like columns used in `WHERE`, `GROUP BY`, or `ORDER BY` clauses. Note: in PostgreSQL indexing is made by DBMS for every instance of a query
- Large or frequently updated columns are poor candidates for indexing.
- **Clustered vs. Unclustered**: Clustered indexes are best for sequential access, while unclustered indexes handle point queries.

Conclusion and Takeaways

- DBMS optimizes storage and access based on the characteristics of the storage layers (e.g., disk, memory).
- Understanding how and when to use indexes can greatly improve database performance, particularly for large databases.

- Sequential access is always faster than random access, especially for hard disks.