1. Basic Definition

- Disk-based indexing stores index structures on persistent storage (hard drives/SSDs) rather than in volatile RAM.
- It uses data structures optimized for block storage access patterns (e.g., B+ trees) to minimize disk I/O operations[1][6].

2. Key Components

Aspect	Description
Data Locality	Stores related data in contiguous disk blocks to reduce seek times[5].
Block-Oriented Design	Structures like B+ trees match disk block sizes for efficient read/write[6].
Persistence	Survives system crashes/reboots, unlike in-memory indexes[4].

3. Why It Matters

1. Handles Large Datasets
    - Supports tables larger than available RAM by storing indexes on disk[4].
    - Example: A 100 GB database can use disk-based indexes without requiring 100 GB of RAM.
2. Reduces Disk I/O
    - Minimizes full-table scans by allowing direct access to data locations.
    - Source[2] reports ~30% reduction in disk I/O with proper indexing.
3. Optimizes Query Types
    - Range queries: Linked leaf nodes in B+ trees enable sequential access[1].
    - Exact-match queries: Hash-based indexes provide O(1) lookups[1].
4. Balances Tradeoffs

- Read vs. Write Costs: Indexes speed up queries but slow down inserts/updates[5].
- Space Overhead: Indexes consume disk space (e.g., 20% extra storage for common workloads)[6].

4. Critical Disk vs. Memory Tradeoffs

Factor	In-Memory Index	Disk-Based Index
Speed	Nanosecond access (RAM)	Millisecond access (disk)
Capacity	Limited by RAM size	Scales to petabytes
Use Case	Real-time analytics	OLTP databases, large datasets[4]
Cost	Expensive (RAM costs)	Cost-effective (disk storage)

5. Implementation Essentials

1. Data Structures
   - B+ Trees: Default for most databases (balanced height, sequential leaf nodes)[6].
   - Hash Indexes: For fast equality checks (e.g., `WHERE id = 123`)[1].
2. Index Types
   - Clustered: Physically orders data rows (e.g., primary key in SQL Server)[3].
   - Non-Clustered: Separate structure with pointers to data[1].
3. Storage Models
   - Row-Oriented: Optimized for transactional workloads[5].
   - Column-Oriented: Better for analytical queries (e.g., aggregations)[5].

6. Real-World Impact

- Case Study: A database without indexes might require scanning 1M rows for a query, while a B+ tree index reduces this to ~20 disk accesses (
- $O(\log_m N)$
- $O(\log$
- m
- $N))$[6].
- Tradeoff Example: Adding an index improves read performance by 10x but increases write latency by 15%[5].

By understanding these principles, you can explain how disk-based indexing enables databases to manage massive datasets efficiently while balancing speed, cost, and storage constraints.

To understand why indexing is so efficient, you need to understand how computers connect with the datasource - simplistically in the case of a disk it will 'read' a block of data from the disk (i.e. a page of the librarian index), not sure what it is these days but probably something like 4kb look for what is required and if it doesn't find it, read the next block. If it is reading blocks with whole records (for a sequential search) it might pick up say 100 records, but if indexed it might pick up 1000 (because indexes are smaller) so will find your record 10 time quicker - and with indexing algorithms it will have a better idea of what block to read next - in the analogy above, you look at the first page - books starting with A, but since you are looking for a book starting with Z, you know to go to the last page and work backwards.

Indexing does have a time overhead when a record is inserted, deleted or the indexed field changed because it needs to be updated to maintain the index order. But this more than pays dividends when you want to find that record again.

However there is no point in indexing for the sake of it, just those fields you are going to join on or regularly sort and/or filter on. There is also little point in indexing fields which have few distinct values (like Booleans) or contain a lot of nulls because the index itself will not be that efficient (although with Access you can set the index to ignore nulls). Go back to library analogy - how useful is it to have pages of 'Yes' followed by pages of 'No'?

Clearly the size of the index field will also have an impact - the larger the datatype, the fewer elements can be read in a single block. Longs (typically used for autonumber primary keys) are 4 bytes in length, dates (used for timestamps for example) are 8 bytes in length whilst text is 10 bytes plus the number of characters.

Users will still need to search on text (for a name for example), but performance can be improved by using a numeric link between a relatively short list of author names and the long list of books they have written. Primary key is in the author name table and foreign/family key is in the book table.

Using an initial * in a like comparison, prevents the use of indexing because indexing starts from the first character (think of the librarian index above) so should be avoided as much as possible - how many users would look for 'Smith' by entering 'ith'? This is also a good reason for storing names in two fields (firstname/lastname) rather than a single field so each can be indexed separately.

Sometimes it can't be avoided - but better to either train users to enter a * when required, or provide a button/option for begins with/contains to implement the initial * when required.