On the most fundamental level, a database needs to do two things: when you give it some data, it should store the data, and when you ask it again later, it should give the data back to you.

Data Structures That Power OLTP Databases

- There are two main families of storage engines commonly used in OLTP:
    - Log-structured: More recent invention, only allows appending to files and deleting obsolete files. Examples include Bitcask, SSTables, LSM-trees, LevelDB, Cassandra, HBase, Lucene. Log-structured storage engines optimize writes by converting random-access writes into sequential writes, resulting in higher write throughput.
    - Update-in-place (page-oriented): Treats the disk as fixed-size pages that can be overwritten. B-trees are a common example used in relational and nonrelational databases.

Simplest database (key-value store):

```
#!/bin/bash
db_set () {
        echo "$1,$2" >> database
}
db_get () {
        grep "^$1," database | sed -e "s/^$1,//" | tail -n 1 # last occurrence
}
```

- Writes:
    - Every call to db_set appends to the end of the file, preserving previous versions of the value (log).
    - Appending to a file is efficient, since append to a log is the simplest way to writing something to db.
- Reads:
    - Every call to db_get scans the entire database file from beginning to end in order to find occurrences of the key.
    - The performance of lookups is poor, with a time complexity of O(n). As the number of records in the database increases, the lookup time also increases proportionally.
- Log - append-only sequence of records.
- To efficiently retrieve the value of a specific key in the database, an index is required.

Indexes:

- Indexes - additional structures (metadata) derived from the primary data that help in locating specific data efficiently. They act as signposts and improve the performance of queries.
- Adding and removing indexes has no impact on the contents of the database, but it affects query performance. Indexes provide a trade-off, speeding up read queries while slowing down writes due to the need for index updates during data writes.

Hash indexes

- The simplest indexing strategy is to use an in-memory hash map where each key is mapped to a byte offset in the data file. This allows quick lookup of values by seeking to the corresponding offset in the file. When new key-value pairs are appended to the file, the hash map is updated to reflect the new offsets.
- Simple, but it is an effective solution. Bitcask, the default storage engine in Riak, uses a similar approach, requiring that all the keys fit in the available RAM
- If you keep a hash map in the RAM, it is well suited to situations where the value for each key is updated frequently, but the overall number of distinct keys is smaller (to fit into memory). To avoid running out of space, we can segment the logs and use compaction.
    - Compaction - involves discarding duplicate keys in the log and retaining only the most recent update for each key.
- Issues with implementing a log-structured engine with hash indexes include:

Issue	**Description**
File format	Use a binary format that includes the length of a string in bytes (CSV not preferable).
Crash recovery	In-memory hash maps are lost if the database is restarted, requiring a rebuild of the index.
Partially written records	The database may crash during the process of appending a record to the log, leading to incomplete or corrupt data.
Concurrency control	While writes are typically sequential with a single writer thread, multiple threads can read concurrently from immutable data file segments.

Memory limitations	The hash table must fit in memory, making it challenging to handle a large number of keys.
Inefficient range queries	Scanning over a range of keys requires individual lookups in the hash maps, making range queries less efficient.

why don't you update the file in place, overwriting the old value with the new value? But an append-only design turns out to be good for several reasons

- Appending and segment merging are sequential write operations, which are gen- erally much faster than random writes, especially on magnetic spinning-disk hard drives. T
- Concurrency and crash recovery are much simpler if segment files are appendonly or immutable

However, the hash table index also has limitations:

- The hash table must fit in memory, so if you have a very large number of keys, you're out of luck. In principle, you could maintain a hash map on disk, but unfortunately it is difficult to make an on-disk hash map perform well.
- Range queries are not efficient.

SSTables and LSM-Trees

- The Sorted String Table (SSTable) is a storage engine where key-value pairs are sorted by key. It enables index-free lookup, eliminating the need to keep an index of all keys in memory. The idea of using SSTables in storage engines is seen in systems like Cassandra and HBase, which were inspired by Google's Bigtable paper.
  - We also require that each key only appears once within each merged segment file (the compaction process already ensures that)
- SSTables are often used in LSM-trees, where a cascade of SSTables is merged in the background. This approach allows for high write throughput as disk writes are sequential.
- Maintaining a sorted structure on disk is possible, but maintaining it in memory is much easier. There are plenty of well-known tree data structures that you can use, such as red-black trees or AVL trees
  - When a write comes in, add it to an in-memory balanced tree data structure (for example, a red-black tree). This in-memory tree is sometimes called a memtable.
  - When the memtable gets bigger than some threshold—typically a few megabytes —write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance.

    - In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
    - From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.
- LSM-tree algorithm can be slow when looking up keys that do not exist in the database
- In order to optimize this kind of access, storage engines often use additional Bloom filters. (A Bloom filter is a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.)
- We can also determine the order and timing of how SSTables are compacted and merged, by using size-tiered and leveled compaction.
    - In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables.
    - In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate "levels," which allows the compaction to proceed more incrementally and use less disk space.

B-Trees

- B-trees are widely used indexing structures that keep key-value pairs sorted by key, allowing for efficient lookups and range queries. They organize the database into fixed-size blocks or pages, with one designated as the root. Updates and additions are performed by searching for the relevant leaf page, modifying the value, and updating the parent page if necessary. They remain the standard index implementation in almost all relational databases, and many nonrelational databases use them too
- B-trees maintain balance, ensuring a logarithmic depth. Write operations involve overwriting pages on disk, and careful concurrency control is needed for multi-threaded access. A write-ahead log (WAL) is often used for crash recovery.
- Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient keyvalue lookups and range queries.
- B-trees break the database down into fixed-size blocks or pages, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time.
- Each page can be identified using an address or location, which allows one page to refer to another—similar to a pointer, but on disk instead of in memory.
- One page is designated as the root of the B-tree; whenever you want to look up a key in the index, you start here. The page contains several keys and references to child pages. Each child is responsible for a continuous range of keys, and the keys between the references indicate where the boundaries between those ranges lie
    - Eventually we get down to a page containing individual keys (a leaf page), which either contains the value for each key inline or contains references to the pages where the values can be found.
    - The number of references to child pages in one page of the B-tree is called the branching factor.
- a B-tree with n keys always has a depth of O(log n).

- The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. LSM Trees only append to files and delete obsolete files but never modify files in place.
- In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a write-ahead log (WAL, also known as a redo log).
    - This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself. When the data- base comes back up after a crash, this log is used to restore the B-tree back to a consistent state
- An additional complication of updating pages in place is that careful concurrency control is required if multiple threads are going to access the B-tree at the same time —otherwise a thread may see the tree in an inconsistent state. This is typically done by protecting the tree's data structures with latches (lightweight locks). Log structured approaches are simpler in this regard, because they do all the merging in the background without interfering with incoming queries and atomically swap old segments for new segments from time to time.
- Optimizations:
    - Instead of overwriting pages and maintaining a WAL for crash recovery, some databases (like LMDB) use a copy-on-write scheme
    - We can save space in pages by not storing the entire key, but abbreviating it.
    - Additional pointers have been added to the tree. For example, each leaf page may have references to its sibling pages to the left and right, which allows scanning keys in order without jumping back to parent pages.

B-Tree vs LSM Tree:

- A B-tree index must write every piece of data at least twice: once to the write-ahead log, and once to the tree page itself. There is also overhead from having to write an entire page at a time, even if only a few bytes in that page changed.
- Log-structured indexes also rewrite data multiple times due to repeated compaction and merging of SSTables.
- An advantage of B-trees is that each key exists in exactly one place in the index, whereas a log-structured storage engine may have multiple copies of the same key in different segments.
    - This makes B-trees attractive in databases that want to offer strong transactional semantics: in many relational databases, transaction isola- tion is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree

LSM Trees:
- Faster writes
- Higher throughput (lower write amplification)
- Better compression
- Impact from compaction

- Disk Space requirements
- Configuration Impact

B-trees:

- Faster reads
- Strong semantics
- Predictable performance
- Lock-based isolation
- Attached locks
- Key-based locks

Other Indexing Structures

- Secondary indexes are crucial for efficient joins and improved query performance in databases. In relational databases, you can create multiple secondary indexes on a table using the CREATE INDEX command.
- A primary key uniquely identifies one row in a relational table, or one document in a document database, or one vertex in a graph database. Other records in the database can refer to that row/document/vertex by its primary key (or ID), and the index is used to resolve such references.
- R-Trees: Geospatial indexes (e.g., PostGIS)
- Fuzzy indexes: Full-text search (e.g., Lucene, ElasticSearch)
- Storing indexes:
    - Heap Files: Index key maps to pointer in heap file
    - Clustered Index: Data stored directly in the index (e.g., MySQL's InnoDB)
    - Covering Index: Data split between index and heap files
    - Concatenated Index: Indexing on multiple columns by concatenating keys
    - Multi Column indexes:
        - The most common type of multi-column index is called a concatenated index, which simply combines several fields into one key by appending one column to another (the index definition specifies in which order the fields are concatenated).
        - A multi-dimensional index gives complex queries high-speed access to combinations of dimensions across data sources.

Keeping everything in memory

- disks have two significant advantages: they are durable (their contents are not lost if the power is turned off), and they have a lower cost per gigabyte than RAM.
- As RAM becomes cheaper, the cost-per-gigabyte argument is eroded. Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory, potentially distributed across several machines. This has led to the development of in memory databases.
- Some in-memory key-value stores, such as Memcached, are intended for caching use only, where it's acceptable for data to be lost if a machine is restarted.

- other inmemory databases aim for durability, which can be achieved with special hardware (such as battery-powered RAM), by writing a log of changes to disk, by writing peri- odic snapshots to disk, or by replicating the in-memory state to other machines.
- When an in-memory database is restarted, it needs to reload its state, either from disk or over the network from a replica (unless special hardware is used). Despite writing to disk, it's still an in-memory database, because the disk is merely used as an append-only log for durability, and reads are served entirely from memory.
- the performance advantage of in-memory databases is not due to the fact that they don't need to read from disk. Even a disk-based storage engine may never need to read from disk if you have enough memory, because the operating sys- tem caches recently used disk blocks in memory anyway. Rather, they can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk
- Redis offers a database-like interface to various data structures such as priority queues and sets

Transaction Processing or Analytics?

- Transaction - group of reads and writes that form a logical unit, this pattern became known as online transaction processing (OLTP).
- Data warehouse - separate database that analysts can query to their heart's content without affecting OLTP operations. (OLAP)

	OLTP(Transaction)	**OLAP (Analytics)**
Use	Handling high volumes of user-facing requests	Analyzing large amounts of data (gigabytes to petabytes)
Access	Small number of records accessed	Large number of records scanned
Indexing	Relies on indexes	Less reliant on indexes
Bottleneck	Disk seek time	Disk bandwidth
Query	Key-based queries	Scanning large numbers of rows
Updates	Frequent and concurrent updates	Fewer queries, but more demanding

Purpose	Low latency and high throughput. Represents latest state of data	Fast query performance and data analysis. Maintains historical data.
Examples	Online shopping, banking, e-commerce applications	Business intelligence, data warehousing
Technologies	MySQL, LevelDB, Cassandra, HBase, Lucene	Hive, Presto, Spark, Redshift, BigQuery

Data Warehouses:

- OLTP systems are usually expected to be highly available and to process trans- actions with low latency, since they are often critical to the operation of the business.
- We are reluctant to let business analysts run ad hoc analytic queries on an OLTP data-base, since those queries are often expensive, scanning large parts of the dataset, which can harm the performance of concurrently executing transactions
- Data warehouses follow a star schema, where facts are captured as individual events, providing flexibility for analysis. The fact table can grow to be very large. Dimensions in data warehouses represent various attributes related to the event, such as who, what, where, when, how, and why.
- A variation of this template is known as the snowflake schema, where dimensions are further broken down into subdimensions.

Column-Oriented Storage

- In most OLTP databases, storage is laid out in a row-oriented fashion: all the values from one row of a table are stored next to each other.
- Column-oriented storage improves query performance by efficiently retrieving relevant columns.
- It reduces storage requirements through effective compression techniques.
- Bitmap encoding and indexing are well-suited for various query types in column-oriented storage.
- Vectorized processing optimizes CPU utilization.
- LSM trees are commonly used for indexes in column-oriented storage.
- Writes can be slower due to the lack of in-place updates and the need to rewrite column files.
- Materialized aggregates, such as materialized views and data cubes, enhance query performance but require additional maintenance during writes.