1. Storage Engine

* The storage engine (or database engine) is a software component of a database management system responsible for storing, retrieving, and managing data in memory and on disk.
* While databases can respond to complex queries, storage engines look at the data more granularly and offer a simple data manipulation API, allowing users to create, update, delete, and retrieve records.
* One way to look at this is that database management systems are applications built on top of storage engines, offering a schema, a query language, indexing, transactions, and many other useful features.
* Many of the storage engines were developed independently from the database management systems they’re now embedded into.
* At the same time, clear separation between database system components opens an opportunity to switch between different engines, potentially better suited for particular use cases.
* The "rest" of the database software provides a common interface for use through APIs or terminals, allows complex actions (like joins, subqueries, etc.) and manages the health/status and configuration of the included engines. There is a lot to the feeding and care of an engine, and the database software hides all that complexity.

Advanced considerations for Storage Engines

* Consistency, Transactions & Concurrency Control
* Compactions
* Compressions

But the most important aspect/component would be the **index management**:

* The effective organization and structure of the data to search for it and access it.
* In the case of SQL and NoSQL, both solutions build special data structures called "indexes."
* The data structure chosen often helps to determine the performance characteristics of the store and retrieve commands.

The **data structure** used highly affects the use case/workload performance in two areas:

* Read and write

The three best data structures:

1. B-Tree
   * A self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
   * Are used in most (if not all) RDBMS products.
   * In general, all operations perform well when the data size fits into the available memory.
     + By memory, I mean the amount of RAM accessible by the RDBMS in either the physical server or virtual server.
     + This memory limit is usually a hard restriction.
   * A combination of:
     + B-tree based primary/secondary indexes along,
     + with row-based storage (where a row is a single record in the database).
   * Pros and Cons
     + B-trees usually grow wide and shallow, so for most queries very few nodes need to be traversed.
     + The net result is **high throughput, low latency reads**.
     + However, the need to maintain a well-ordered data structure with random writes usually leads to poor write performance.
     + This is because **random writes to the storage are more expensive than sequential writes**.
     + Also, a minor update to a row in a block requires a read-modify-write penalty of an entire block.
   * Real World Usage
     + The default storage engines of **popular monolithic relational/SQL databases** follow the B-tree architecture
       - the list includes Oracle DB, MS SQL Server, IBM DB2, MySQL (InnoDB), and PostgreSQL.
     + While it may be tempting to conclude that B-tree engines are only for SQL databases, it should be noted that even **NoSQL databases** can be based on B-tree engines.
       - E.g. MMAPv1, the original storage engine of MongoDB (default in versions prior to 3.2), is B-tree based. Couchbase is another NoSQL database powered by a B-tree engine.
   * When is B-tree **a good** solution for your applications
     + When data size doesn’t exceed memory limits
     + When the application is mostly performing read (SELECT) operations
     + When read performance is more important than write performance
   * When is B-tree **not a good** solution for your applications
     + Events that might exceed B-tree performance limits include:
       - accepting and storing event logs;
       - storing measurements from a high-frequency sensor;
       - tracking user clicks; and so on.
   * In most cases, it is possible to solve B-tree performance issues with **more memory or faster physical storage**. But when hardware adjustments aren’t hits the hard limit, a **different data structure can help**.
2. LSM-Tree

* As data volumes grew in the mid 2000s, it became necessary to write larger datasets to databases.
* The LSM tree is a data structure with performance characteristics best fit for indexed access to files with **high write volume over an extended period**.
* Perfect for:
  + accepting and storing event logs;
  + storing measurements from a high-frequency sensor;
  + tracking user clicks;
  + etc
* The first mention of LSM trees dates back to 1996 and corresponds to Google BigTables.
* An LSM tree works by:
  + Storing incoming modify operations in a **buffer** (usually named "memtable")
  + Sorting and storing the data when the buffer is full
* Pros and Cons
  + LSM engines are the **de facto standard** today for handling workloads with **large fast-growing data**.
  + The primary reason behind their success is their ability to do **fast sequential writes** (as opposed to slow random writes in B-tree engines) especially on modern flash-based SSDs that are inherently better suited for sequential access.
  + LSM-tree based engines exhibit poor read throughput in comparison to B-tree based engines.
  + This is because LSM engines demonstrate [**higher read and space amplification**](https://dzone.com/articles/state-storage-engine) than B-trees.
  + LSM engines consume more CPU resources during **read operations and take more memory/disk storage**.
    - E.g., a point query with an LSM tree may require multiple random reads.
* Real World Usage
  + LSM engines are now default in popular **NoSQL databases** including Apache Cassandra, Elasticsearch (Lucene), Google Bigtable, Apache HBase, and InfluxDB.
  + Even widely adopted embedded data stores such as LevelDB and RocksDB are LSM based.
  + One such embedded RocksDB implementation is MyRocks that replaces the default InnoDB engine in MySQL.
  + DocDB, the YugabyteDB storage engine, is also built on a custom version of RocksDB.
  + Later it was implemented in products such as Cassandra, LevelDB, and most recently in RocksDB.

1. Fractal Tree

* Fractal Tree data structures are closer to traditional B-tree structures—but instead of applying changes immediately, **changes are buffered**. (Similar to LSM?)
* As information exceeds the limits of the main index memory, the tree data structure buffers large groups of messages.
* The buffered data is slowly pushed down the tree as the buffers fill up. When data gets to a leaf node, **there is a single IO applied to the data.**
* This helps avoid random operations causing performance degradation by performing buffer changes all at once.
* **Data compression reduces read IO further.**
* Fractal Trees are a good structure for:
  + databases with a lot of tables, indexes (preferably non-unique indexes),
  + and a heavy write workload.
  + It is also good for systems with slow **storage times**,
  + or for saving space when the **storage is fast but expensive**.
* Lastly, this is often a good fit for **cloud-based database environments**, where again storage is often slow (or if fast, expensive).

Therefore:

* At the core, database storage engines use an index management algorithm that is optimized for either **read performance (B-tree)** or **write performance (LSM)**.
* In the last 10+ years data volumes have grown significantly and LSM engines have become the standard.
  + LSM engines can be tuned more easily for higher read performance (e.g. using bloom filters and various compaction strategies) compared to tuning B-tree engines for higher write performance.
* Note that the supported data model (relational vs. non-relational) and the corresponding database client API (SQL vs. NoSQL) are **not directly tied to the design** of the storage engine.
  + Both SQL and NoSQL databases are built on B-tree and LSM engines.

References

1. <https://www.openlife.cc/blogs/2020/may/whats-database-storage-engine>
2. <https://blog.yugabyte.com/a-busy-developers-guide-to-database-storage-engines-advanced-topics>
3. <https://dzone.com/articles/how-three-fundamental-data-structures-impact-stora>
4. [Database Internals: a Deep Dive into How Distributed Data Systems Work](https://www.amazon.com/dp/B07XW76VHZ/?coliid=IW0E4GD9BIED6&colid=G2E7UL6TP3PG&psc=0&ref_=lv_ov_lig_dp_it)