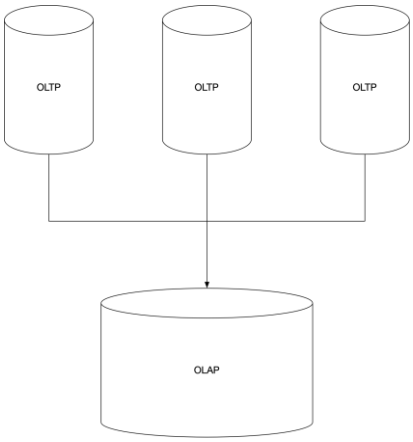
Data Engines

Before we look at different data engine implementations, we should first distinguish between two broad categories of usage: Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP). OLTP systems access data by key and updates occur in response to interactive user input. OLAP systems support data analytics and involve scanning a huge number of records, reading a few fields from each and then providing aggregate statistics. Typically, an organisation will organise their topology so that the data stores supporting OLAP are separate from the data stores supporting OLTP. A separate database, known as a data warehouse, is usually used for data analytics. Data is extracted from various OLTP databases into the data warehouse.

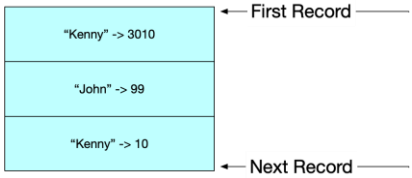


The reason for this separation is that OLTP systems need to have high availability with low latency. We do not want OLTP users impacted by heavy data analytic queries. Microsoft SQL Server supports both transaction processing and data analytics. Internally, each use case has its own database engine with both engines supporting the same SQL interface. Another advantage of having a separate OLAP system is that we can use different structures that are optimised for this use case. In particular, the indexing solutions used in OLTP are not optimal for OLAP. We look first at which indexing solutions are appropriate for OLTP and then move on to looking at which are suitable for OLAP. We are particularly interested in indexing techniques which are used to speed up read access.

OLTP Data Engines

Logs

A log is an append only file. For efficient use it is typically in a binary format.



ADVANTAGES

- Appending is a sequential write operation which is faster than random writes.
- Concurrency is simpler if we have append-only logs.
- Fast Constant time writes.

DISADVANTAGES

- Poor read performance requiring scans.

Indices

To improve read performance, we add additional meta-data structures called indices. Each additional index supports querying by a different piece of data. The downside of using indices is that every index we add slows down writes. The reason for this is that each index needs to be updated when a write is performed. Indexes can be categorised in different ways. We now look at three techniques that can be used within storage engines to support efficient read performance.

- Hash Index
- Sorted String Table (SST)
- B-trees

Before we do, however let us first describe the kinds of index we use.

Primary Key Index

A primary key uniquely identifies one row in a relational database or one document in a document database. Primary key indices can be created using the key-value indices using the techniques of SST and B-Trees which we will describe in subsequent sections.

Secondary Index

With relational databases we can add several secondary indices to a single table. With secondary indices the indexed values are not necessarily unique. There could be multiple rows under a single index value. We can use SST and B-trees, but we need to modify them so that each value in the index is a list of row identifiers

Clustered Index

If the value in an index is the actual row object itself then we have a clustered index. There is no extra level of indirection and performance is fast. SQL server supports one clustered index per table.

Non-Clustered Index

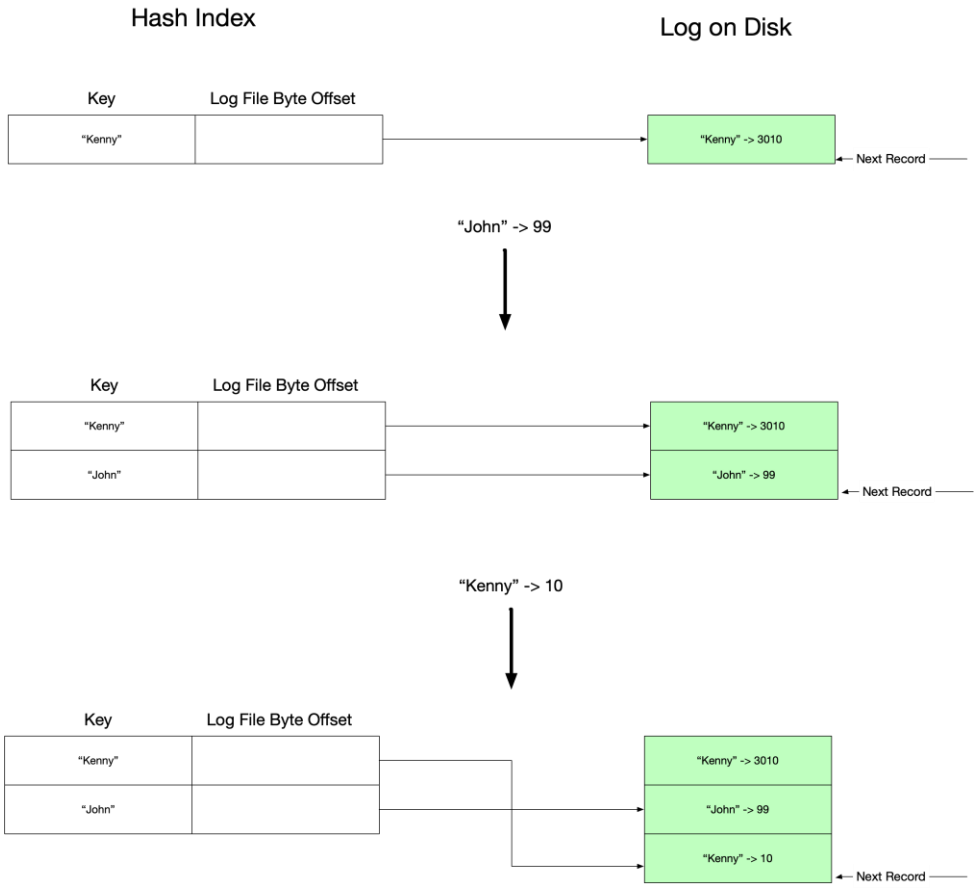
When using a non-clustered index, the values within an index are just references to the row objects.

Multi-Column Index

Most database engines support multi-column indexes using concatenation. As the order of the columns in the index defines the sort order, given an index on `{last-name, first-name}`, we can efficiently query on `last-name` or `{last-name, first-name}` but we cannot efficiently query on `first-name`.

Hash Index

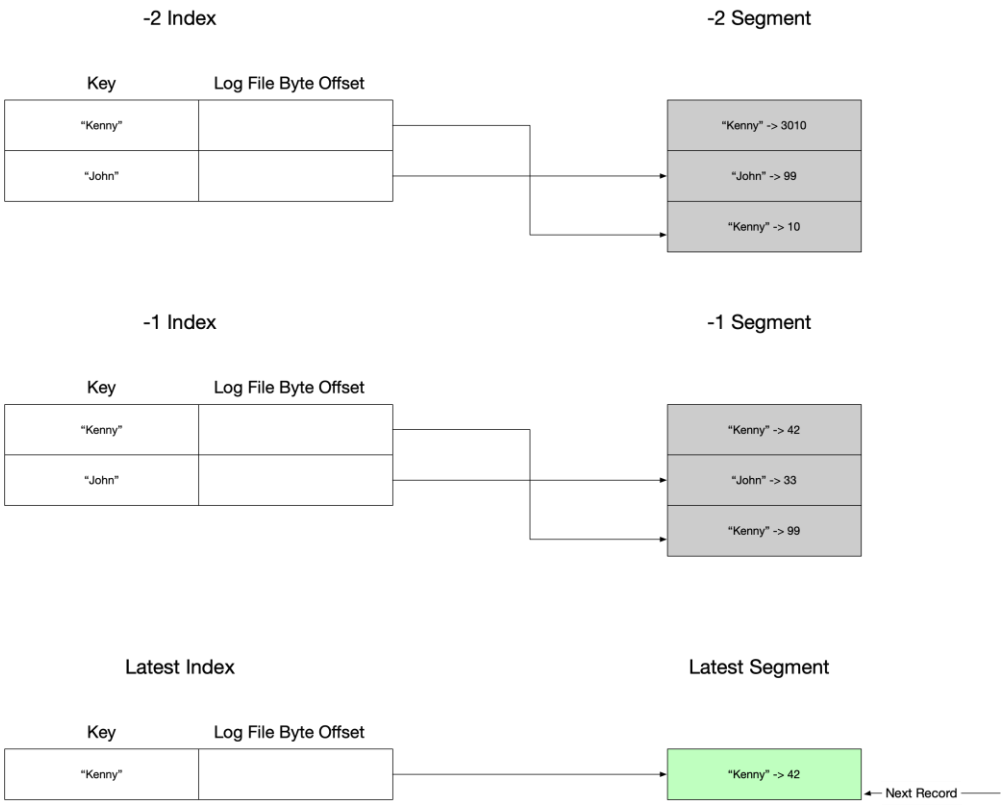
A hash index is the disk based equivalent of a hash table. We maintain an in-memory map of every key to the byte offset of its value in the Log file. The following shows how the structure evolves. We start with a single value in the log and show what happens as we add two values. The second value is an update rather than an insert.



An update involves appending to the log on disk and updating the in-memory hash table. If we only ever did this, the log on disk would eventually run out of space.

Compact and Merge

In order to enable compaction, we use the following strategy. Rather than using a single log file on disk, we instead use a set of log segments each of which is a separate file on disk. When the most recent segment reaches a certain size, we open a new segment, and all subsequent writes go to the new segment.



As all writes are against the most recent segment (coloured green) , old segments can be compacted and merged on a background thread. Writes against the most recent segment and reads from the segments being compacted can occur in parallel with the compaction process. Once the compaction process completes the uncompacted are replaced with the compacted ones and then freed for deletion

Each segment needs its own in memory hash table. Looking up a value involves checking the hash table for the most recent segment. If no match is found, we look at the hash table for the second most recent segment and so on.

This simplistic approach is what the Riak storage engine BitBask does. The major limitation is that all the keys need to fit into memory. For this reason, it is best suited to use cases where we have a limited number of keys that are updated frequently.

Disadvantages and Limitations

- All keys must fit in memory.
- Range queries are inefficient and require a linear traversal of all data.

Sorted String Table (SST) and Log Structured Merge Trees (LSM Trees)

Sorted String Tables address the problem of efficient range queries by making a modification to the format of the segments on disk. Unlike with hash indexes we ensure the values in the segments are sorted by key.

In order to ensure we only write to the disk using sequential IO we create segments as follows. When updates come in, we write them into an in-memory data structure that maintains values ordered by key such as a red-black tree. This in memory structure is known as a memtable. When the memtable reaches a certain size, we write it to a new ordered segment on disk. Doing it this way mean disk writes always use sequential IO. When we have written the men table to disk we need to keep a sparse index in memory for that segment.

A read request then proceeds as follows. First, we look in the memtable. If the value is not found we then look in the sparse index for the most recent segment on disk and the next most recent until we find the value we want.

Merging and compacting segments while keeping the compacted segment in order is trivial using a procedure like the one used in mergesort. If we have values for the same key in two different segments being merged, we choose the one from the most recent segment. As the segments are read-only merging and compaction can occur on a background thread.

In order to make sure we do not lose the memtable should the server crash we can use a simple log file to write values to. In case of a crash, we can then restore the memtable from the log.

SSTs are used in RocksDB and Level DB key-value pair storage engines. Similar techniques are used in Casandra and HBase.

One edge case that performs badly is looking for a key that does not exist. A technique called Bloom Filters can be used in such optimise this use case.

B Trees

B-trees are the standard index implementation is relational databases. B-trees use fixed sized blocks known as pages. The idea is the page size aligns with the fixed sized blocks on the disk. Pages are overridden in place so we do not change their location and hence don’t update any other pages which reference the page being overridden.

ToDo: Add more detail on b-trees

To increase reliability b-tree implementations use a write ahead log.

Online Analytic Processing

As we mentioned previously, the indexing techniques appropriate for OLTP are not suitable for OLAP use cases. An OLAP fact table can store trillions of rows. Even if a fact table has hundreds of columns, each query typically is only interested in a few of them. Laying out data on disk in a row-by-row fashion as most OLTP engines do is inefficient. We may load hundreds of columns per row even if we are only interested in two of them.

Columnar Databases

The basic idea of columnar databases is that rather than storing the data for each row together we instead store all the data for each column together. Each column can be stored in a separate file and we only need to read the files wholes columns are specified in the query. Storing data in columns can often make compression more effective. Two encoding techniques that are particularly effective with columnar data are bitmap encoding together with run-length encoding. The Logical operators AND/OR can then bit performed in a bitwise fashion when determining the rows we need.