Apache Lucene is a text search engine library written mainly in Java(a python port is also available called pylucene). In this paper I will explore the functionality and high level overview of Apache lucene. We will also briefly explore popular applications that use Lucene such as Solr.

In order to quickly retrieve text and as mentioned in our lectures, Apache Lucene stores data in indexes. Apache Lucene by default stores the indexes in disk instead of memory. It also supports in memory indexing(classes called MemoryIndex and RamIndex) for performing full text searches for full documents (sizes < 10MB file). On a high level, in order to create an Index in lucene we will first need to create a Lucene Document object. We will then pass in a group of Lucene object fields to the document object. The Lucene object field represents the text content that will be searchable via query later(IE in an academia paper example, for each paper that we wanted to index we can store separately using many fields such as Author, Abstract, Body, Reference. Then we will store the contents of each part separately which we can then specify when searching (IE, search the Author field that starts with B)).

Each lucene field object can also be configured to also store important metrics which will then be used for certain search algorithms(IE: Term Frequency), it also has other properties such as if the contents needs to be persisted/tokenized. After the Lucene document object is initiated with the right fields we will then pass it to Lucene's IndexWriter class to disk.

Unlike relational databases that store records using a B+ tree data structure for indexing, Lucene's inverted index for terms and documents are stored in small pieces using array-like data structures called segments. Is it important to note that Lucene's segments are immutable, if you are indexing a new document it will create a new segment instead of updating an existing segment(since it is an array like and adding new terms will be expensive). The segments will be then merged with the original posting segments in the background job, this causes lucene's update/add to happen in an eventual consistent manner instead of a strong consistency. Each document in the posting list also has a bit flag to indicate if a document has been deleted or not, when a document is deleted only the bit flag is updated but the actual deletion will happen later on which causes deletion in Lucene to not immediately delete the data from disk. The reason Lucene chooses this approach is in order to make disk reads much more efficient because the way the data is accessed is sequential(due to using arrays) which makes the data filesystem cache friendly. The downside of the implementation is that Lucene can be a bit slow (depending on the use case), when adding bulk documents, since the operations require the creation of new segments and merges for document additions.

Given a query, Lucene is also able to utilize multiple indexes parallelly at once this allows Lucene to be able to search much faster compared to other RDBMS. Lucene also uses delta compression to store document id. Lucene also has an API(Lucene's codecs class) that can be overridden if the developer wanted to create their own implementation of compression and serialization/deserialization logic.

After documents are added and written to disk using Lucene's indexing library we can perform searches on them. Lucene supports some scoring algorithms such as BM25, TF-DF(full list in the Lucene's Similarity package). It also supports probabilistic language model scoring like Dirchlet Prior and Jelinek Mercer. In order to perform a search using a specific scoring algorithm we will first need to specify the Similarity class to the IndexWriter class when creating the index to make sure that all the required fields are written and exist for the scoring algorithm to work. Afterwards we will need to instantiate Lucene's IndexSearcher class and create some query related classes in which we can specify how many top scored documents we wanted to search for (IE return only the top 10 documents). Lucene's query search will look up fields in the disk and therefore in order to make Lucene's searches run really fast we need to make sure that the instance has high memory so that documents can fit in the file system cache in memory. It is essential that the process running Lucene needs to keep growing the memory as more documents are added to ensure that the file system cache can fit the right amount of data to not slow down the search query.

Solr is a web app written in Java that is used for searching documents and it uses Lucene as the engine for indexing and searching. Solr provides the rest APIs needed to access Lucene's library for indexing and searching. Solr requires a schema written in XML in order to let them know what fields to index instead of a Java/Python class like Lucene. As it uses lucene as the backing engine it also supports similar scoring functions like BM25.

Solr also supports ways to scale and provides failure tolerance for your lucene indexes by providing sharding and replication. These features are built into SolrCloud. Index Sharding, is Solr's way to split up the index across multiple machines allowing it to scale horizontally. After the index is separated across multiple instances SolrCloud will handle the logic on how to distribute the query to the instances. Replication meanwhile is the process of duplicating the indexes across multiple backup machines that will be elected as the master in case the previous master dies. (it also provides redundancy in case of permanent failures). Elasticsearch is also another similar (semi) open source web app that performs exactly like Solr where it uses Lucene as it's search and indexing engine and provides similar functionality and additional production support like sharding and replication

In conclusion, Lucene is a java library that is widely used for searching and is efficient at indexing and searching text documents. Lucene also supports a wide variety of search algorithms that are popular in the text retrieval world like (BM25 and DP/JM). However, for production that has a really high traffic where we need horizontal scaling it is better that we use Solr(SolrCloud)/ElasticSearch since they already implemented ways to horizontally scale lucene indexes to multiple instances.

References:

Analysis high level of lucene =
https://alibaba-cloud.medium.com/analysis-of-lucene-basic-concepts-5ff5d8b90a53
What is in a Lucene index? =
https://www.youtube.com/watch?v=T5RmMNDR5XI&ab_channel=LuceneSolrRevolution
What is in a Lucene index? =
https://www.slideshare.net/lucenerevolution/what-is-inaluceneagrandfinal
Lucene Index documentation =
https://lucene.apache.org/core/8_10_1/core/org/apache/lucene/index/package-summary.html
Lucene Memory Index = https://lucene.apache.org/core/8_10_1/memory/
Lucene Similarity =
https://lucene.apache.org/core/8_0_0/core/org/apache/lucene/search/similarities/Similarity.html
SOLR = https://solr.apache.org/guide/6_6/a-quick-overview.html#a-quick-overview
SOLR-Sharding =
https://solr.apache.org/guide/6_6/shards-and-indexing-data-in-solrcloud.html