Big Data indexing techniques

indexing strategies can be categorized into Artificial Intelligence (AI) approach and Non-Artificial Intelligence (NAI) approach.

The Non-Artificial Intelligence techniques:

A hash-based indexing technique: designed to be search-efficient in the context of high-dimensional data. Using the compact Steiner tree to allow for improved keyword searches on relational databases. Hash indexing accelerates information retrieval by detecting duplicates in a large dataset,

Hash indexing strategy is used in password-checking systems and DNA sequence matches.

Hash is used in Big Data indexing to index and retrieve data items (in a dataset) that are similar to the searched item.

It uses a hashed key (which is computed by the hash function and usually shorter than the original value) to store and retrieve indexes.

hash indexing is more efficient than the tree-based indexing in terms of equality or point query.

Search on shorter hashed keys can be faster than search on unpredictable length key. hashing technique works fine with limited data size, it tends to exhibit indexing computational overhead as data size increases.

In a hash indexing strategy, data is organized into a hash table, which is a data structure that maps keys to indices in the dataset. The mapping is done using a hash function, which takes in a key and produces an index (also known as a "hash value") that is used to locate the data in the hash table.

To look up a data item in a hash table, you compute its hash value using the hash function and then use the hash value to index into the table. If the data item is present in the table, it will be located at the index corresponding to its hash value. If the data item is not present, the table will return a "not found" value.

Hash indexing is fast and efficient because it allows for constant-time lookups, insertions, and deletions. However, it is not suitable for datasets that have many duplicate values or that require range queries, as the hash function will map multiple keys to the same index,

causing collisions. In these cases, a different indexing strategy, such as tree-based indexing, may be more appropriate.

The performance of a hash indexing strategy depends on the quality of the hash function and the load factor of the hash table.

The hash function should be designed to distribute the keys evenly across the indices of the table, so that the number of collisions is minimized. A good hash function will have a low collision rate, which means that it will map different keys to different indices with high probability.

The load factor of the hash table is the ratio of the number of items in the table to the number of indices in the table. A high load factor indicates that the table is heavily populated and may result in a higher collision rate. To maintain good performance, the load factor should be kept as low as possible, ideally below 0.7.

In general, a hash indexing strategy can provide fast and efficient data access and manipulation, as long as the hash function is well-designed and the load factor is kept low. However, it may not be the best choice for datasets that have a high degree of duplication or that require range queries. In these cases, a different indexing strategy, such as tree-based indexing, may be more appropriate.

The Tree-based indexing strategies: The Tree indexing structures are the B-tree, R-tree, Xtree.

retrieval of data is done in sorted order, following branch relations of the data item.

This satisfies nearest neighbor queries. According to researches, the Tree indexing strategies are being displaced by other indexing strategies because they are generally outperformed by simple sequential scans.

To search for a data item in a tree-based index, you start at the root of the tree and compare the data item to the key at the root. If the data item is less than the key, you go to the left child; if it is greater, you go to the right child. You repeat this process until you reach a leaf node, at which point you will either find the data item or determine that it is not present in the tree.

Insertion and deletion operations in a tree-based index follow a similar process, but also require the maintenance of the tree's balance. This involves adding or removing nodes and adjusting the tree structure as needed to ensure that the tree remains balanced and the data is properly sorted.

Tree-based indexing is a good choice for datasets that require range queries or that need to be sorted, and it can provide efficient data access and manipulation in these cases.

However, it may not be as efficient as hash indexing for datasets that do not require these capabilities.

The performance of a tree-based indexing strategy depends on the balance of the tree and the complexity of the operations being performed.

In a well-balanced tree, the height of the tree is kept as small as possible, which means that the number of comparisons needed to search for a data item or perform an insertion or deletion operation is minimized. This results in faster data access and manipulation.

The complexity of the operations being performed also affects the performance of a tree-based index. For example, a search operation in a B-tree has an average-case complexity of O(log n), where n is the number of data items in the tree. This means that the time required to search for a data item in the tree increases logarithmically with the size of the tree. Insertion and deletion operations in a B-tree have a similar complexity.

In general, a tree-based indexing strategy can provide efficient data access and manipulation, as long as the tree is well-balanced and the complexity of the operations being performed is kept low. However, it may not be as efficient as hash indexing for datasets that do not require range queries or sorting.

B-tree: A B-tree works like the Binary tree search, but in a more complex manner. This is because the nodes of B-tree have many branches, unlike the binary tree which has two branches per node. So a B-tree is more complicated than a binary tree.

B-tree indexes satisfy range queries and similarity queries also known as Nearest Neighbor Search (NNS), using comparison - operators (<, < =, =, >, > =).

In the B-tree, the keys and all records are normally stored in leaves, but copies (of the key) are stored in internal nodes, the leaves might include pointers to the next node, showing the path to the searched item.

only suitable for one-dimensional access method unlike other tree-based access methods or indexing strategies such as the R-tree.

the B-tree algorithm consumes huge computing resources when performing indexing on Big Data [9]. Other variations of the B-tree are the B+tree, B*tree, KDB-tree, and so on.

R-tree: This is an indexing strategy used for spatial or range queries. It is mostly applied in geospatial systems with each entry having X and Y coordinates with minimum and maximum values, The advantage of using an Rtree over a B-tree is that, the R-tree satisfies multi-dimensional or range queries, whereas the B-tree does not.

Given a query range, using the R-tree makes finding answers to queries quick.

The idea is to group data items according to their distance from each other, and assign minimum and maximum bounds to them. Each record at the leaf node, describes a single item. Each internal node describes a collection of items or objects.

Though the R-tree is preferred over the B-tree in the case of indexing spatial data, the R-tree does not find the exact answer as query results. It merely limits the search space. Also, it

consumes memory space because coordinates are stored along with the data. Variants of the R-tree are R*tree, and R+-tree.

X-tree: This type of indexing strategy, based on the R-tree, satisfies range queries. The X-tree is similar to the R-tree and operates just like the R-tree.

Although, unlike the R-tree which satisfies 2-3 dimensional range queries, the X-tree satisfies queries of many dimensions.

This implies that the X-tree is a more complicated version of the R-tree. The advantage of the X-tree over the R-tree is that it covers more dimensions, otherwise, the X-tree also consumes memory space due to the storage of coordinates.

The Artificial Intelligence techniques:

Hidden Markov Model (HMM): The Hidden Markov Model (HMM) indexing approach is an access method developed from the Markov model.

A Markov model is made up of states which are connected by transitions, where future states are solely dependent on the present state and independent of historical states. the HMM uses pattern recognition and relationship between data.

In the HMM indexing approach, data or characteristics which the states depend on during query, are categorized and stored in advance.

The query results are usually predictions of future states of an item, based on the current or present state. The present state is used to predict the future states using the dependent data or characteristics of the states.

HMM was used to classify and store motion data used by robots. The classification was based on the acceleration information, consisting of the position information and the pure force.

the prediction of the next series (sequence) of motions was dependent on the position informant and the pure force. The motion data is stored and classified in advance, before the quick search for motion is conducted.

The performance of an HMM depends on the quality of the model and the amount and complexity of the data being modeled. HMMs can be trained using maximum likelihood estimation or other techniques, and the model's parameters can be fine-tuned to improve its performance.

In general, HMMs can be effective at modeling and predicting sequences of data, but they may not be as suitable as other techniques for indexing large datasets. They are more commonly used for tasks such as natural language processing and speech recognition, where they can be used to model the underlying structure of the data and make predictions about future events.

In natural language processing, HMMs can be used to model the structure of a language and predict the next word in a sentence or the part of speech of a word. In speech recognition, they can be used to model the sounds of a language and predict the words

being spoken. In bioinformatics, they can be used to model the structure of proteins and predict the function of a protein based on its sequence.

To use an HMM for these tasks, you would need to specify the model's parameters, such as the number of hidden states, the probabilities of the transitions and observations, and any additional features of the data that you wish to model. You would then use a training dataset to estimate the model's parameters and evaluate the model's performance on a separate test dataset. Once the model has been trained and evaluated, it can be used to make predictions on new data.

Comparison of indexing strategies:

Indexing strategy	Data-Type	Query-Type	Properties	Challenges
A hash-based indexing technique	Log data, multimedia data	Point query → Equality search	- Presents the exact answer (uses '=' operator) - Quick information retrieval	- Computational overhead
The Tree-based indexing strategies	Spatial data, multimedia, Log data	Range query, similarity queries (NNS): 1 D, Spatial or Range query: 2-3 D, Spatial or Range query: multi-dimension	- One-dimensional access method -Tree structure with nodes and pointers - Scales linearly - More scalable than the B-tree - 2 to 3-dimensional access method - Multidimensional access method	- Waste storage space - Not suitable for multidimensional access - Consumes huge computing resources - Index consumes more memory space - Consumes memory space
Hidden Markov Model (HMM)	Multimedia data, unstable signals	Ad-hoc query	- Based on the Markov model - Recognizes relationships between data	- Demands high computational performance