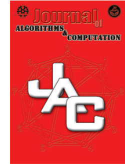
Journal of Algorithms and Computation

journal homepage: http://jac.ut.ac.ir



Efficient Storage and Retrieval of In-Memory Static Data

Anuj Kapoor*[1]

[1]Senior Software Engineer, Department of Technology, Priceline LLC, 800 Connecticut Ave, Norwalk, CT 06854, USA.

ABSTRACT

Hash or B-Tree based composite indexes, are the two most commonly used techniques for searching and retrieving data from memory. Although these techniques have a serious memory limitation, that restricts *freedom* to search by any combination of single key/data attribute, that comprises the composite search key, the techniques are still accepted considering the trade offs with better performance on insert and update operations. But when the data is semi-static, which does not change often, there is a need and scope for a better technique that provides the flexibility and freedom to efficiently search by any possible key, without creating any composite index. This paper explains such algorithmic technique along with its data structures.

Keyword: static data, trie, search algorithm, composite index, combination key

ARTICLE INFO

Article history:
Received 15, October 2019
Received in revised form 17, April 2020
Accepted 10 May 2020
Available online 01, June 2020

AMS subject Classification: 05C40..

1 Introduction

Hash Indexes Hash indexes provide a very efficient way to retrieve data, with an average time complexity of constant time, i.e. $\theta(1)$, provided we choose the hash function

*Corresponding author: A. Kapoori. Email: anuj.kapoor@kapoorlabs.com

carefully, and, we keep the load factor of hash table to be less than 1 [3, 2]. Such good performance also comes with a trade off, that is having lack of flexibility. Each Hash index corresponds to a particular lookup key, if we need to search the database using another key for lookup, we need to have a separate index built using that key.
Take the following table-1 as an example of the data, we want to search through, with the first row representing different data attributes/column names.

Table 1: Sample data with introductory example

Country	Region	State	City Code	Airport Code	Major Airport
USA	EAST	NY	NYC	JFK	Y
IN	NORTH	PB	ASR	ATQ	Y
USA	EAST	RI	PVD	PVD	N

- If we want to search the database using Country as a lookup key, we would create a hash index using Country as a key.
- If we want to search the database using Country and Region as a lookup key, we would create a hash index using Country + Region as a key.
- If we want to search the database using Region as a lookup key, we would create a hash index using Region as a key.

And so on..
Precisely we would need 2^n indexes, if we want the capability of searching through the data, using any key at a consistent high performance.

B-Tree indexes B-Tree indexes[1, 4] overcome this limitation to a certain extent, by increasing the time complexity of lookup from constant time to logarithmic scale. Taking the same example again to compare with B-Tree indexes, following are some use cases, in regards to usage of index.

- A B-Tree index, created with a combination of following ordered keys, Country + Region + State, can be used to lookup, using following keys: -
  - Country + Region + State
  - Country + Region
  - Country

  The order of the keys matter, so we cannot use this index to search using Region or Region + State as a lookup key.
- A B-Tree index, created with a combination of following ordered keys, Region + City Code, can be used to lookup, using following keys: -

- Region + City Code
- Region

And so on..

Precisely, using B-Tree indexing we would need $1 + \sum_{n=1}^{n-1} n$ indexes, if we want the capability of searching through the data, using any possible key combination, at a consistent high performance. The book Physical Database Design [4] recommends using B+Tree indexes for searches with composite keys, marking usage of single-key indexes to search for multiple keys, much slower, due to slower merging and more block access operations. However, this paper explains an indexing technique for in-memory static data, that operates on simple single indexes, with search response times as good as, or in some cases even better, when compared to searches with composite B+Tree indexes, also in addition, this technique provides flexibility to search for any combination of keys, which is not feasible for B-Tree based indexes beyond some point, as explained in table-2, with growing number of possible keys in search criteria.

Proposed Solution While B-Tree indexing was designed for disk storage, it is still a very good choice for in-memory databases that support insert, update and delete operations. But if we have an in-memory database that support only read operations, then we can greatly improve the flexibility of searching by any combination key, at consistent high performance.
To achieve this goal, this paper explains a technique by organizing the database in a Trie structure[2], where each level in the Trie comprises all the values associated with a single key. Each level is indexed using a Hash index. The structure is further provided numeric ids using depth first traversal technique[1], during load time. Once loaded, the database can be searched using any combination of keys, using hash indexes at each level, which is then reduced using logarithmic binary search of Ids, assigned during the load time of depth first search traversal.
This technique drastically improves the number of indexes needed to search with any possible composite key, with consistently good performance. The comparison with number of indexes required is compared with other techniques in the table-2.

Table 2: Required number of indexes for all possible key combinations

N	Hash indexes (2^n)	B-Tree indexes ($1+\sum_{n=1}^{n-1} n$)	**Proposed solution** (n)
1	2	1	**1**
5	32	11	**5**
10	1,024	46	**10**
100	1.2676506e+30	4,951	**100**
500	3.273391e+150	124,751	**500**
1,000	∞	499,501	**1,000**
10,000	∞	49,995,001	**10,000**
100,000	∞	4,999,950,001	**100,000**

2 Data Structure

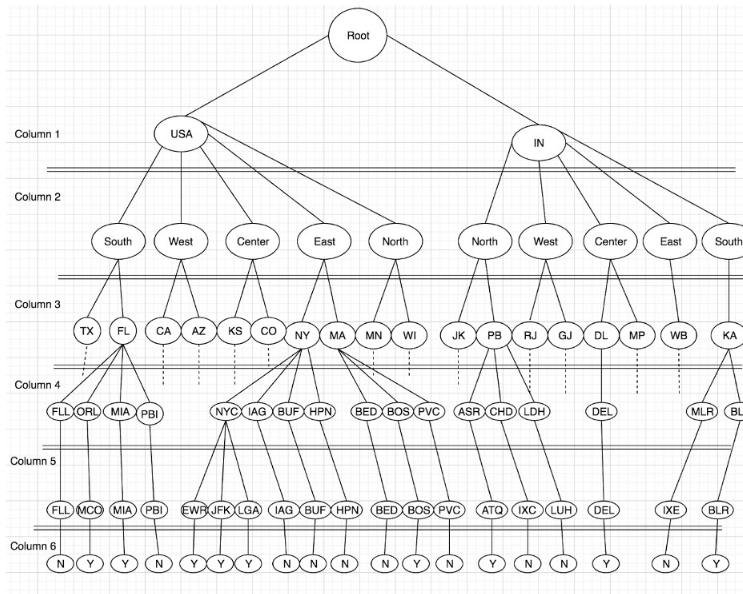


Figure 1: Inverted Indexed Trie Structure with sample data

Table 3: Sample data for example

Contry	Region	State	City Code	Airport Code	Major Airport
USA	SOUTH	FL	FLL	FLL	N
USA	SOUTH	FL	ORL	MCO	Y
USA	SOUTH	FL	MIA	MIA	Y
USA	SOUTH	FL	PBI	PBI	N
USA	EAST	NY	NYC	JFK	Y
USA	EAST	NY	NYC	LGA	Y
USA	EAST	NY	NYC	EWR	Y
USA	EAST	NY	IAG	IAG	N
USA	EAST	NY	BUF	BUF	N
USA	EAST	NY	HPN	HPN	N
USA	EAST	MA	BED	BED	N
USA	EAST	MA	BOS	BOS	Y
USA	EAST	MA	PVC	PVC	N
IN	NORTH	PB	ASR	ATQ	Y
IN	NORTH	PB	CHD	IXC	N
IN	NORTH	PB	LDH	LUH	N
IN	CENTER	DL	DEL	DEL	Y
IN	SOUTH	KA	MLR	IXE	N

Structure Figure- 1 explains the structure of the data built with the sample data from Table-3, with the first row explaining the data attributes and rest of the rows as records. Trie is constructed in such a way that each level in the constructed Trie represents the same data attribute. In this example Level 1 represents the Country, Level 2 represents Region, Level 3 represents State and so on.

De-Duplication of Data The apparent benefit of storing data in a trie is memory optimization by de-duplicating data[2]. At each level we only store unique values for that

data attribute and create nodes for it. For example, the Trie in Fig-1, only creates one node for USA and only one node for IN. Then within USA, there will be hundreds of airports in one region, say EAST, but we only create a single node for a the region EAST, instead of duplicating it for hundreds of records. Thus, Trie helps us de-duplicate data, while still maintaining its semantic relations with other attributes.

Structure of the Trie Node As explained in Fig-2, the Trie node will have the following attributes: -

- Value
- List of Children Nodes for downward traversal
- Parent reference for upward traversal
- Pre-visit number (lower bound) To identify Ancestor child relation
- Post-visit number (upper bound)- To identify Ancestor child relation

Pre-visit and Post-visit numbers [6] are the result of a Depth First Search conducted on the structure starting from the root. Every time we visit/backtrack from a node, we increment the counter. The node we visit, gets the pre-visit number as the counter value, when we backtrack from a node, post-visit number of that node gets the counter value.

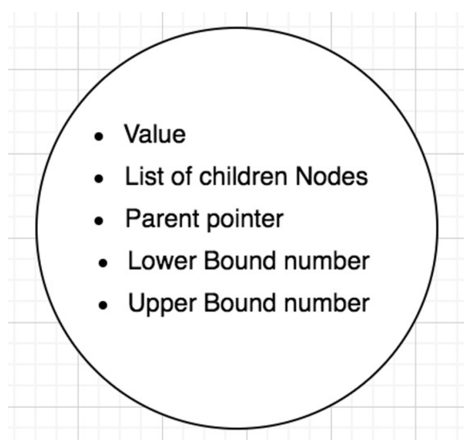


Figure 2: Pre-visit and Post-visit calculation with Depth first search

For example, consider a tree in fig-3 with computed pre-visit and post-visit values. The benefit of a pre-visit and post-visit number calculation from a depth first traversal is to quickly identify ancestor-grandchild relation[6], or to identify if there is a connection between two distinct nodes. We can quickly conclude that a node with preVisit 6 and postVisit 7 is a grandchild of a Node with pre-visit 1 and post-visit 12, because 6-7

falls within the range of 1-12. Similarly, if we see a node with pre-visit as 14 and post-visit 15, we would know that this node has no relation with a node of pre-visit 1 and post-visit 12, because the ranges dont overlap. This property would help us quickly identify relations between data nodes that satisfy a single key match. Also, the difference between (postVisit-preVisit) / 2 will give us the number of children and grandchildren for a particular node.

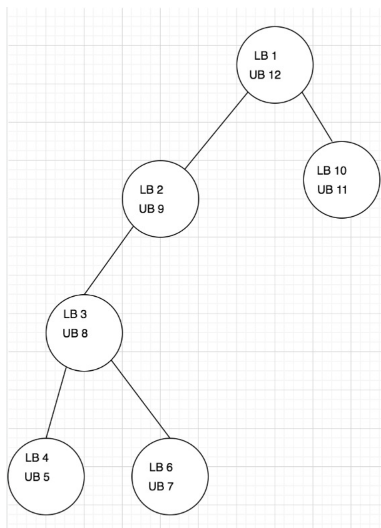


Figure 3: Sample tree with pre-visit an post-visit values plotted after DFS traversal

Inverted Index After we plot pre-visit and post-visit numbers on our Trie structure, it looks like fig-4. pre-visit is written on top of the node and post-visit is written on the bottom. Lastly, we create a single inverted index at each level, post which our data structure would be complete and ready for searching. We can build the inverted index in the same step of depth first search, which we do to compute pre-visit and post-visit values. Whenever we backtrack from a node, we can calculate the post-visit number and enter its reference in the index/map.
As depicted in Fig-4, each level int the trie, corresponds to a particular key/ data attribute, and for each level, we will create a Hash index. Where key-value pair would be as follows: -

- Key Value of the Trie node
- Value Array List of references to nodes, with the same value as specified in the key.

Note: - As we are building the index during our Depth first search step, the values in the Array List of the map will always be in increasing order of pre-visit values. We will use this property to binary search this Array List in our algorithm. The reason we chose

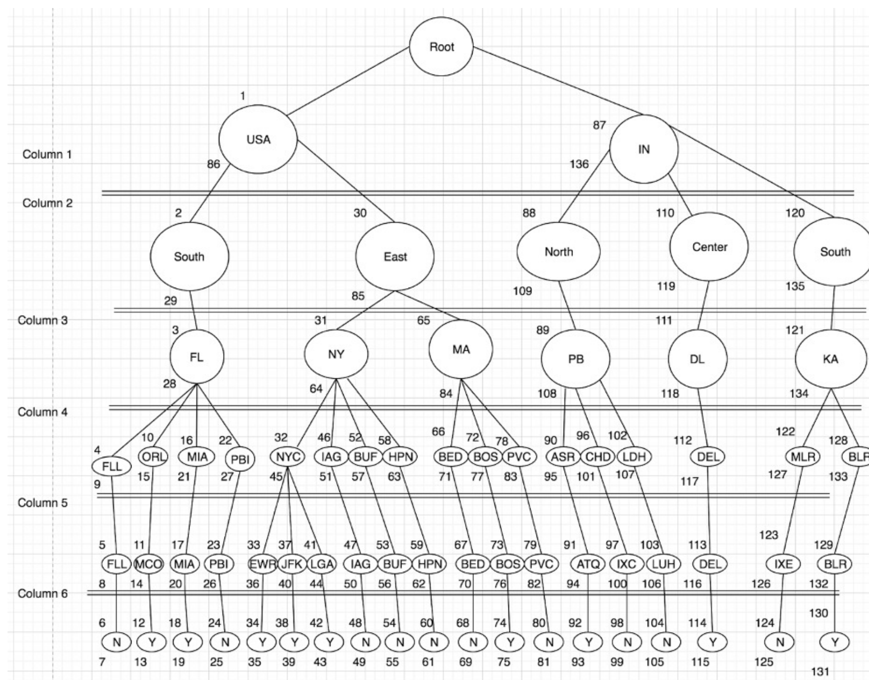


Figure 4: Trie data structure, after plotting pre-visit and post-visit values for each node

Array list over Linked List was to be able to perform binary search over this list in the future step.

3 Algorithm

3.1 Algorithmic Goal

The goal of the algorithm is to provide flexibility in searching through the data, using any possible combination of keys in consistent high performance, without creating any composite index[1].

3.2 Basic Algorithmic operations

The algorithm discussed in this paper has two basic operations.

- Key-level lookup
- Binary reduction of Key-level lookups

Key-level lookup During key-level lookup operations, we will lookup each level's index and get the array list of all the nodes, whose value is equal to the key. Keep in mind that the array list that we get from the operation is in sorted order of pre-visit numbers. For example, lets say our search criteria for the records described in Fig-4 is as below: -
Get all the airports where

- Country = US
- Region = SOUTH
- Major Airport = Y

Here we will perform 3 key level hash lookups separately, and the results will be as follows: -

Country = US Resultant array list: -

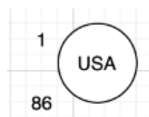


Figure 5: Resultant array list from hash lookup of Country = US

[1]Composite index is an index created from a combination of more than one key

Region = SOUTH Resultant array list: -

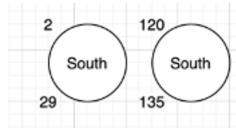


Figure 6: Resultant array list from hash lookup of Region = SOUTH

Major Airport = Y Resultant array list: -

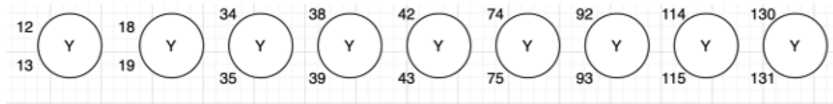


Figure 7: Resultant array list from hash lookup of Major Airport = Y

Binary reduction of Key-level lookups In binary reduction of key-level lookups, we take a pair of resultant array list from previous step and reduce it to one list, containing the nodes that has overlapping ranges of pre-visit and post-visit values.
Doing an operation similar to binary search[5] on nodes with preVisit-postVisit ranges [2-29, 120-135] we find that the node with range 2-29 overlaps with the range 1-86, and hence forms our answer for the reduction step. We will repeat this process until all resultant array lists have been reduced.

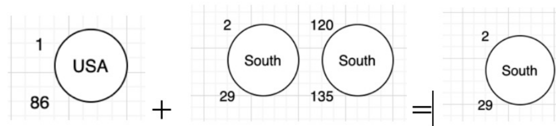


Figure 8: Binary reduction of key lookups, Country = US and Region = SOUTH

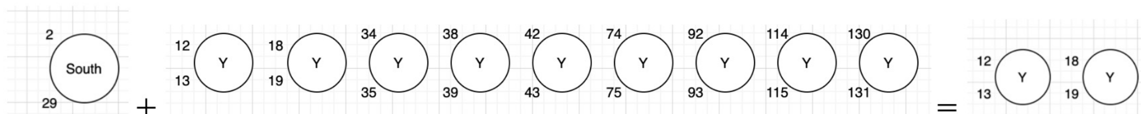


Figure 9: Binary reduction of previously reduced lookup + key lookup of Major Airport = Y

3.3 Algorithm in steps with example

Lets consider the same example, and walk through entire steps.
Search criteria: Get Airport code and city code where
Country = USA,Region = SOUTH AndMajor Airport = Y
We will process this in order of the column level as depicted in fig-4.
Country has level 1 Region has level 2 Major Airport has level 6

- Step 1: We use the Hash Index of the country and get the Array List of nodes that has country as USA.

  Result: Array list containing one Node USA with preVisit 1 and postVisit 86.

- Step 2: We use the Hash Index of the Region and get the Array List of nodes that has region as SOUTH.

  Result: Array list containing two Nodes SOUTH with preVisit 2 and postVisit 29 and another with preVisit 120 and postVisit 135.

- Step 3: Identify the list from previous 2 steps that has lesser elements. We will use lesser to binary search the greater.

  Result: Array List of country identified as lesser. We will binary search if elements in Array list of regions are contained within array list of country

- Step 4: Binary search the range [1-86] in ranges [2-29, 120-135] and get all ranges that are contained in range being searched, i.e. [1-86].

  Result: [Node with range 2-29]

- Step 5: We use the Hash index of the Major Airport and get the Array List of nodes that has Value "Y".

  Result: Nodes with ranges: - [12-13,18-19,34-35,38-39,42-43,74-75,92-93,114-115,130-131]

- Step 6: Identify the list from previous 2 steps that has lesser elements. We will use lesser to binary search the greater.

Result: [Node with range 2-29]

- Step 7: Binary search the range [2-29] in ranges [12-13,18-19,34-35,38-39,42-43,74-75,92-93,114-115,130-131] and get all ranges that are contained in range being searched, i.e. [2-29].

  Lets talk through our steps of binary search. Get the mid: 38-39 Range 2-29 is lesser, so we search the left half and disregard right half.

  Get the mid: 18-19 18-19 is within 2-29, so add Node with range 18-19 in the result. Next iterate left and right until the range breaks

  12-13 is within 2-29, add that to the result. Exhausted left, now iterate right of 18-19

  34-35 was in the right half that was discarded so we break

  We try to get another range from the ranges being searched [2-29], but there is none so we break.

Result: Nodes with ranges: [12-13, 18-19]

Result of step 7 is our result set nodes whose children and parents/grandparents satisfy all the conditions of the query.

Now to form the response we just get all the parents and the children.

In this case, there are no children, so we just move up to find parents, until we have everything that was asked.

Our query asked for only airport codes and city codes, so we will move up to column 4 and 5.

- Step 8: From the result of Step 7 that is nodes at level 6 with ranges [12-13,18-19], traverse upwards, using parent pointers, to level 5 and 4, to get airport code and city code

Result: 18-19 − > AIRPORT CODE = MIA, CITY CODE = MIA, 12-13 − > AIRPORT CODE = MCO, CITY CODE = ORL

It is apparent from the Fig-4, that it is correct response.

4 Runtime Analysis

Operational breakdown Lets break our analysis into load time(*which is a one-time operation*) and search time.

4.1 Load Time:

1. To build the Trie, we have to go through all data elements. So, the time complexity to build initial Trie is θ(MN)[2, 1]. Data with M rows and N columns.
2. To compute pre-visit and post-visit, we do a depth first search and at each node we enter an entry to a Hash Index to the end of Array List. Time for depth first search is θ(MN)[2, 1] and time for entry in Hash Index is θ(1)[3].

So, the total time to build the Trie is θ(MN) + θ(MN) = θ(MN)

- M $->$ Number of data attributes/ columns.
- N $->$ number of records, or rows.

4.2 Search time :

Worst Case : Lets say we have K conditions specified in search criteria.
For each criteria we get the nodes from the inverted index in O(1) operation.
In the worst case, the number of nodes returned will be equal to the number of rows, that is the ***entire database***.
For each con

So, the worst case will be KNLog(N).

- K $->$ Number of keys used in search.
- N $->$ Total number of rows in data table.

Once we have the nodes, the next step is to get its parents and children.
In worst case we get all parents and all children.
So, if the last criteria was at level T.
To find the parents from that node, we need T steps to go up
To find the children we need to traverse all the children, this is a little tricky to estimate, since we there is no relation to duplicate data that can exist in the table. So, for this estimation we assume unique data, with only one child for each node. So, we need M-T steps to go down.
That is, M steps in total.

So, the total worst run-time results into θ(KMNLog(N)).

- K $->$ Number of keys used in search.
- M $->$ Total number of data attributes.
- N $->$ Total number of records in database

Average Case : But the worst case is a rare scenario, search queries rarely ask almost all the records in the database. In an average case, value of N returned by inverted index will be much lower. For example, if a search query, intends to return back about 10% of the total records in the database, then run-time becomes:-

θ**(KMN'Log(N'))**

- K $->$ Number of keys used in search.
- M $->$ Total number of data attributes.
- **N' $->$ 10 % of the total number of rows in database, which is a good average case.**

5 Future research work

KiaraDB Using the concepts mentioned in this paper, we have developed a complete in memory database for java based applications. The package is host in the central maven repository at:-

Maven package groupId : com.kapoorlabs, artifactId:kiara, version:1.0.0

Open source The source code is hosted at: https://github.com/kapoorlabs/kiara

features The database support a lot of different operations in addition to the equality, like inequality, partial match, list match, relational operations such as less than, greater than, and the work further continues, in this exciting field of flexibility to search by any key with consistent performance, without creating any composite index.

References

[1] , Cormen, T. H., Leiserson, C. E., Rivest, R. L.,Stein, C., Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

[2] Knuth, D. E. The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, 1997.

[3] Korotkevitch, D. Hash Indexes. In: Expert SQL Server In-Memory OLTP, Apress, Berkeley, CA, 2017.

[4] Lightstone, S., Teorey, T. Physical Database Design, Morgan Kaufmann, 2007.

[5] Nowak, R., Generalized binary search, in: 2008 46th Annual Allerton Conference on Communication, Control, and Computing, 2008, pp. 568-574.

[6] Papadimitriou, M. P. Christos, Depth first search and it's applications (1998).