A Survey On Indexing Structures of Databases

Erhan Sezerer
232630003
erhansezerer@iyte.edu.tr

Abstract

The fact that the databases are getting larger and larger, accessing them by searching records exhaustively becomes a questionable method. Without index mechanisms, we would have to search for a result to a query by reading a whole database starting from the beginning until we find it. There are various index mechanism introduced to the literature to solve this problem and speed up database systems. In this paper, I will introduce many indexing mechanism including bitmap index using B-trees, R-trees and so on.

Introduction

In large databases where we have to query within millions of records and queries are issued to the system frequently by a lot of users concurrently, using indexing can help us speed up the system by a considerable amount. Using the correct index system in a specific query is very important, considering that there are various indexing mechanism that indices the database according to columns or according to records and so on. Also there are special type of indexing mechanisms for handling queires in spatial or spatiotemporal databases. Although the indexing can increase the amount of CPU work because of the address calculation overhead, I/O operations are reduced so much that it makes up for the increased CPU time with reading much less data. Further discussions of performances will be in next chapters.

In this paper, I will introduce various indexing strategies: B-trees, bitmap index, projection index, bit-Sliced index, groupset index, R-trees, IR-trees and discuss their advantages and disadvantages after briefly describing each of them.

Indexing Mechanisms

B-Trees

B-Trees are one of the earliest indexing strategies proposed by (Bayer et al, 1972). An example B-tree structure looks like the one in Figure 1.

Each node inside the fragments corresponds to a pair (x, α) where x is a key and α is a pointer to a file where the associated record resides. This way of representing the files can reduce the problem of finding a record to a binary search in the B-Tree which is a lot faster algorithm than the exhaustive ones where we have to traverse every document.

There is one more important detail about the B-Trees in order to make it efficient for indexing. Each node in the B-Tree, except for the root node and leaf nodes, can have at least $k + 1$ number of children and at most $2k + 1$ number of children where k is a natural number. This is done for the purpose of keeping the database balanced. Because if we let one fragment to expand to a size $\gg (2k + 1)$ where every other fragment is of size $< (2k + 1)$, then the binary search will not work as efficient as we would like since a considerable amount of indices lies in a single fragment. But this balancing mechanism also introduces additonal CPU cost because of the splitting that occurs when we try to insert a new index on a full node or a catenation when we try to delete an index which is the only node in the leaf.

Same issue needs to be address when we are determining the number k. If we set it too small then the height of the tree increases and the number of iterations in the binary search increases and therefore CPU operations on comparison of indices also increases. On the other hand, if we set it too large then all of the indices will be kept in small amount of fragments and therefore the amount of I/O operations needed increases drastically. In conclusion, decision of the number k is simply a time/space trade-off. Normally, since the amount of I/O operations dominates the complexity of a database system, It is more efficient to keep it as small as possible.

Using the aforementioned structures, B-Tress reduces the amount of I/O operation from polynomial amounts to a logarithmic scale on the number of total nodes.

Variant Indices

Various kinds of indices which are given below,are examined and proposed by (O'Neil et al, 1997)

Bitmap Indices (Value-List Indices) Value-List indices (O'Neil et al, 1997) improves the perfromance of standard B-Tree indices by using bitmaps as a value in (x, α) instead of RID lists. An example of a bitmap index can be seen in Figure 2.

A bitmap index with a column C with values $v_1, v_2 ... v_n$ looks like the one in the Figure 2 where keys are the column identifiers and the values are the bitmaps in which a 1 corresponds to a row that has the given property, or a 0 corresponds to a row that doesnot have that property in that col-

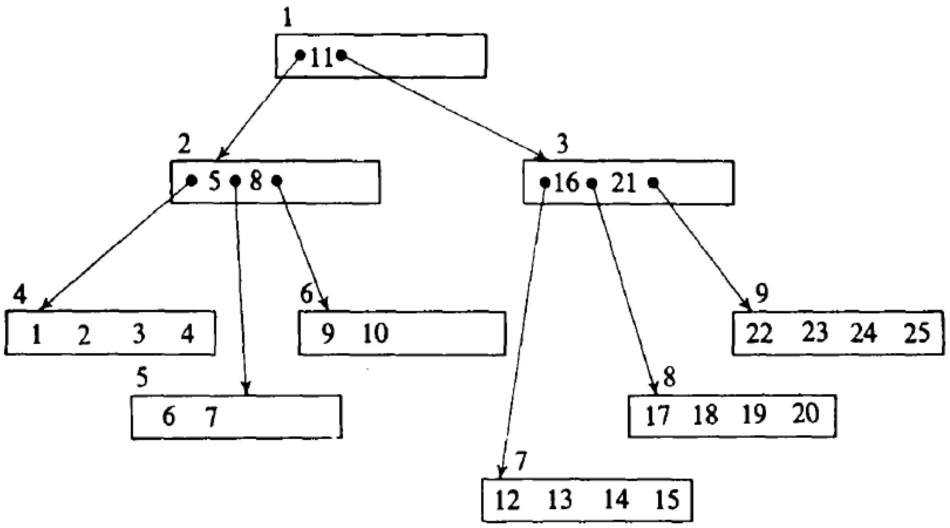


Figure 1: An example B-Tree (Taken from (Bayer et al, 1972))

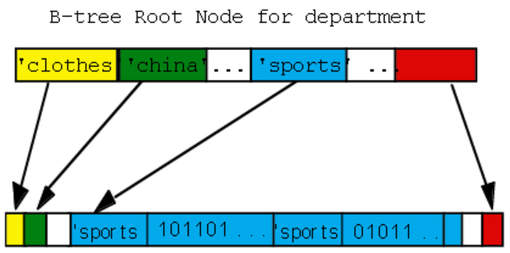
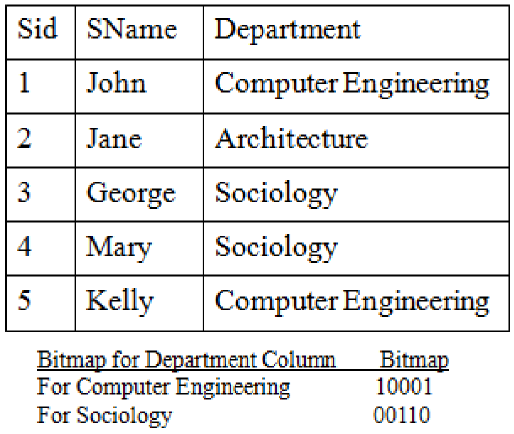


Figure 2: An Example Bitmap Index (Taken From (O'Neil et al, 1997))



Sid	SName	Department
1	John	Computer Engineering
2	Jane	Architecture
3	George	Sociology
4	Mary	Sociology
5	Kelly	Computer Engineering

Bitmap for Department Column	Bitmap
For Computer Engineering	10001
For Sociology	00110

Figure 3: An Example Bitmap of a Student Table

umn. For example Table 3 shows an example database with corresponding bitmap for one of the columns for a property. As we can see each bit in in the bitmap for Computer Engineering corresponds to a row which has the property of "Computer Engineering", within the table.

as the author pointed out, using bitmaps as an index structure turns out to be very effective since any king of operation on the indices will use logical operations like AND, OR or NOT which are faster than RID-list operations.

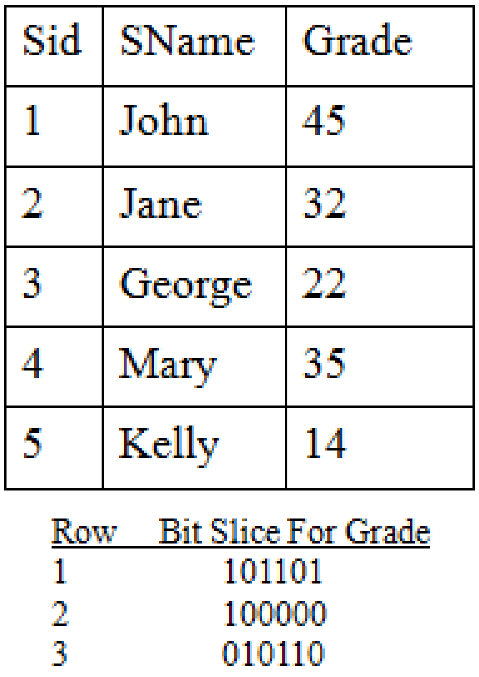
Value-List indices are especially powerful on queries like:

SELECT * FROM *Student*
WHERE *Department* = *"some_value"*

The reason behind it is, a simple binary search through the B-Tree will give us the bitmap that corresponds to the row "Department" for value "some_value" then we can directly send an I/O request to the files containing those rows without reading any unnecessary files or rows.

Projection Indices Projection indices stores tehe value of columns in fixed length maps with the same order as the rows. If a query comes, all we have to do in projection indices to calculate the position of the column with $page\_number = row\_number/value\_per\_page$ and $slot = row\_number\%value\_per\_page$. For example for a table with 12513 rows, considering that we can fit 1000 column values in a page $page\_number = 12513/1000 = 12$ and $slot = 12513\%1000 = 513$. So we will know that the corresponding column value is stored in 513th slot in the12th page.

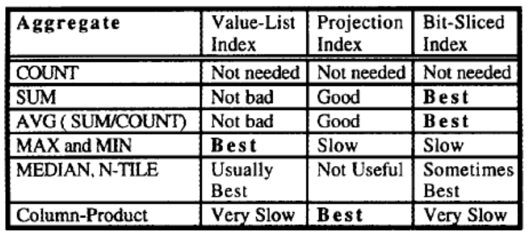
Projection indices are especially powerfull when it comes to queries which parses the entire column without any predicate like aggregate functions(AVE,SUM,etc.) and column products.i.e.:



Sid	SName	Grade
1	John	45
2	Jane	32
3	George	22
4	Mary	35
5	Kelly	14

Row	Bit Slice For Grade
1	101101
2	100000
3	010110

Figure 4: An Example Bit-Sliced Index



Aggregate	Value-List Index	Projection Index	Bit-Sliced Index
COUNT	Not needed	Not needed	Not needed
SUM	Not bad	Good	**Best**
AVG (SUM/COUNT)	Not bad	Good	**Best**
MAX and MIN	**Best**	Slow	Slow
MEDIAN, N-TILE	Usually Best	Not Useful	Sometimes Best
Column-Product	Very Slow	**Best**	Very Slow

Figure 5: Performances of Index Mechanisms on Different Type of Queries (Taken from (O'Neil et al, 1997))

SELECT *AVE(Grade)* FROM *Student*

This is because if we use value-list indices instead of a projection index, then we will have to search for the bitmaps of every value. Instead in projection index, we can easily parse through the column values.

We should also note that, as authors mentioned in (O'Neil et al, 1997), projection indices can and mostly used as a complementary structure. Using value-list indices with projection indices at the same time will increase the overall efficiency of the sytem since we can choose from the types of different indices to use for each different query.

Bit-sliced Indices Bit-sliced indices are vertical to the projection indices according to their orientation. Each bit-sliced index holds the value of a single column where each bit in bit-sliced index corresponds to the binary version of the value that resides in the same digit. An example bit-slices can be seen in the Figure 4. As you can can see each bitslice corresponds to the number in the column. Bit-slices indices can be used on any column which has a numeric value like an integer or a float, etc. Although the bit-sliced methods can be used on other columns (i.e. columns with a string in it), it will not end up being efficient enough to be considered.

Bit-sliced indices are especially efficient on the queries with aggregate functions. For Example;

SELECT *SUM(Grade)* FROM *Student*

An overall efficiency of indices on different type of queries can be examined from the Figure 5.

Join Indices (Groupset Indices) According to (O'Neil et al, 1997), a join index can be defined as:

Definition: A join index is an index on a table for a quantity that involves a column value of a different table through a commonly encountered join.

If a common join is discovered on a database then a join index can be created where indices are built upon a single column in T_1 with a single column in T_2. The reason why such join indices exist is because when it comes to queries like this,

SELECT * FROM *Student,Department*
WHERE *Student.department = Department.DeptID*

all the index methods mentioned before fails in case of a join,because we have to search and retrieve a record from the index for each record in the table that joins it. So, it is proposed that if we can see a pattern of join which occurs frequently then creating an index for it drastically improves the performance of that database system.

Index Structures For Multi-Dimensional Data

All of the indices we have seen so far is aimed at solving the problems of databases with single dimensional data. However, substantial amount of databases start having multi-dimensional spatial data. Examples of such data exist in computer-aided-design software, geographic services and spatio-temporal web searches. In order to cope with the multi-dimensionality various index structures have been proposed by researches, including R-Trees (Guttman, 1984) and IR-Tress (Li et al, 2011).

R-Trees Since the region searches need a range of search, classic index structures does not work on them. R-Trees are introduces with the intention of solving this problem. An

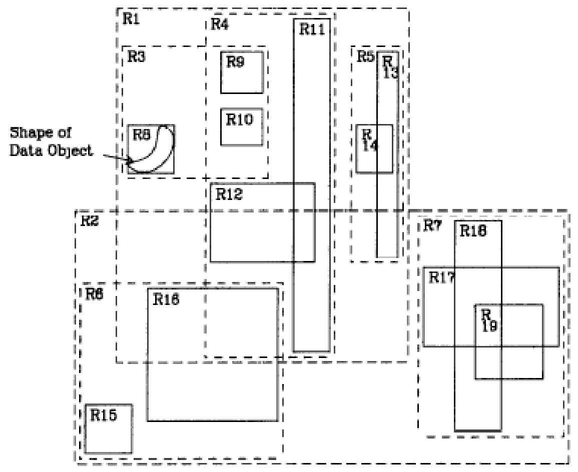


Figure 6: An example Region (Taken from (Guttman, 1984))

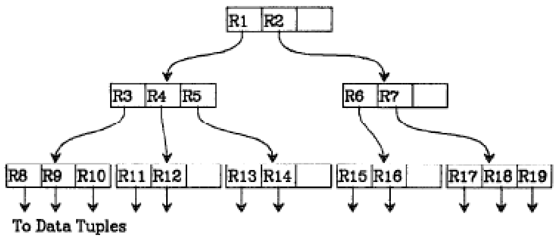


Figure 7: An example R-Tree (Taken from (Guttman, 1984))

example of an R-Tree can be seen in Figure 7 which is contructed from the region in Figure 6. Similar to a B-Tree, R-trees are also balance trees where each node has limit on the maximum (k) and minimum($k/2$) number of children (except root node, again similar to the B-trees). however there are same difference that an R-trees has in order to incorporate the differences of multi-dimensional data.

First important difference is, unlike one-dimensional data, a record of a location may match to more than one region. Region R9 which is inside of both of the region R3 and R1 is an example of this. One-dimensional data indices cannot handle these situations because if you look at the R-Tree shown in Figure

Multi-dimensionality also effects it when it comes to catenating or splitting the nodes for balancing. In one-dimensional data, no matter which nodes are catenated together or split apart, we had the same efficiency since the number of nodes that are searched stayed the same. But now, splitting the nodes in a wrong way highly affects the overall efficiency of the system. Figure 8 shows an example of good and bad splitting. The reason behind such a consideration is, because if we allow for the two new regions to occuppy more area than a lot of smaller reigons will cluster under it and therefore negatively effects the balance of the tree and therefore the efficiency of the indexing mechanism. For this reason, in every splitting, R-Tree indices tries to find a splitting that occupies as least amount of area as possible. However, as the authors pointed out (Guttman, 1984), exhaustively finding the best possible splitting is not feasible, since it has 2^{50} possible combinations. Instead, the authors propose a greedy approach for finding a solution.

IR-Trees

With the rise of search engines, spatial searches with an associated text searches became an important part of queries, making the R-Trees which is an index mechanism on spatial searches alone, obsolete. IR-Trees are proposed in order to allow users to query spatiotextual data.

Imagine a user uses the phrase “Izmir's Kebab Restaurants” in a search engine. We have to find the location Izmir in our queries, but we have to do it with respect to the text “Kebab Restaurant” associated with it. These type searches has a few problems with it as noted by the authors. First of all spatial information and textual information may not be equally important to the user. One user may also be interested in Brasseries, while another user might even consider leaving the city as long as it is a kebab restaurant. ny type of indexing algorithm should handle the weights for textual and spatial references. Second problem is, each location or text might have different forms, representation or relevence. For example, a person may not be willing to go far away for a restaurant but more willing to do it for a concert. Also, the region might be a city, a country, a district and so on. Finally, the index mechanism must respond very fast to the user with top-k result since the online user may not be willing to spend their entire night for query result to a restaurant.

Although one can use a spatial index first and a textual one second then join them, it would be inefficient, and therefore we need an index to deal with both of these aspects of the data concurrently. An IR-Tree in Figure 9 is an example structure to do it within a timely manner. Each node contains hierarchical infromation about regions. On each node top N_i represents a region and each node N_i in the bottom represents a region encapsulated by the top region. Also, different from the R-Tree, each nodes holds a document summary, a DF/TF block that looks like the one in Figure 9. Using this

Conclusion

References

Patrick O'Neil and Dallan Quass. 1997. Improved query performance with variant indices. SIGMOD Rec. 26, 2 (June 1997), 38-49

R. Bayer and E. McCreight. 1970. Organization and maintenance of large ordered indices. In Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data De-

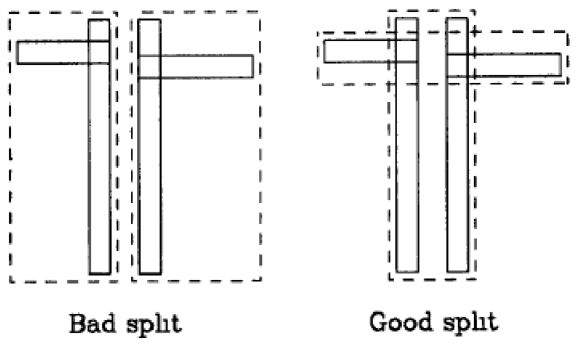


Figure 8: Splitting of an R-Tree (Taken from (Guttman, 1984))

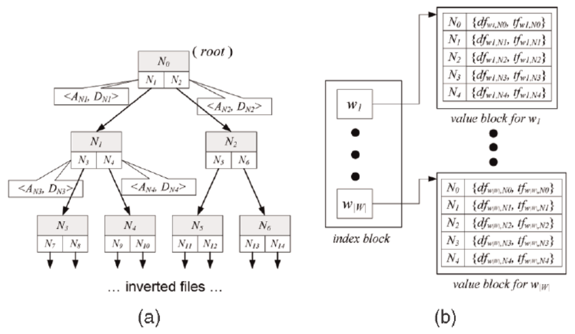


Figure 9: An example IR-Tree (Taken from (Li et al, 2011))

scription, Access and Control (SIGFIDET '70). ACM, New York, NY, USA, 107-141

Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84). ACM, New York, NY, USA, 47-57

Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. 2011. IR-Tree: An Efficient Index for Geographic Document Search. IEEE Trans. on Knowl. and Data Eng. 23, 4 (April 2011), 585-599

Y.Zhou, X.Xie, C.Wang, Y.Gong, and Y.Y.Ma. Hybrid index structures for location-based web search. Proc. 14th ACM int'l Conf. Information and Knowledge Management (CIKM '05), pp.155-162, 2005.