2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

Efficient Distributed Range Query Processing in Apache Spark

Apostolos N. Papadopoulos
Dept. of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
papadopo@csd.auth.gr

Spyros Sioutas
Dept. of Computer Eng. & Informatics
University of Patras
Patras, Greece
sioutas@ceid.upatras.gr

Nikolaos Zacharatos
Dept. of Computer Eng. & Informatics
University of Patras
Patras, Greece
zacharato@ceid.upatras.gr

Christos Zaroliagis
CTI & Dept. of Computer Eng. & Informatics
University of Patras
Patras, Greece
zaro@ceid.upatras.gr

***Abstract*—Range queries are important in many diverse applications. In its simplest one-dimensional form, a range query is expressed by an interval $[a, b]$ on the real line, whereas the answer consists of all elements $e \in [a, b]$. In this work, we focus on efficient range query processing techniques in the Apache Spark engine, which is the state-of-the-art solution for big data management and analytics. We aim at developing a Spark-based indexing scheme that supports range queries in such large-scale decentralized environments and scale well w.r.t. the number of nodes and the data items stored. Towards this goal, there have been solutions in the last few years, which however turn out to be inadequate at the envisaged scale, since the classic linear or even the logarithmic complexity (for point queries) is still too expensive, whereas range query processing is even more demanding. In this paper, we go one step further and present a solution with sub-logarithmic complexity. In particular, we present SPIS (SPark-based Interpolation Search), a tree structure that outperforms the existing Spark built-in lookup techniques. We carry out an experimental evaluation by using synthetic data sets. Our experimental results demonstrate the efficiency and scalability of the proposed approach.**

***Index Terms*—Range queries, Indexing, Apache Spark, Performance evaluation**

I. Introduction

Range queries have been extensively studied in different disciplines, such as computational geometry, geographical information systems, multimedia databases, just to name a few. In its simplest one-dimensional form (also known as the *interval query*), this query involves searching for all elements e where $e \in [a, b]$. Thus, we are interested in detecting all available values that belong to a specific interval. Evidently, generalizations to higher dimensions are derived easily, since we just need to consider intervals in each axis of the coordinate system. Many applications require the management and the analysis of massive multi-dimensional datasets.

- For example, large-scale spatial systems contain massive datasets with location information about the underlying objects. Range queries are often executed to extract information from specific parts of the address space.
- As a second example, consider an e-shop data warehouse which contains information about products. A customer may require to focus on specific products that satisfy several restrictions on the product attributes, e.g., the price should be less than 100$, the screen size should be at least 21 inches. To enable this behavior, efficient algorithmic techniques are required to support range queries in potentially massive amounts of product data.

One of the directions to follow for efficient range query processing is the exploitation of multiple resources (i.e., CPUs and disks). Typically, modern infrastructures are composed of a set of machines organized in a shared-nothing architecture where machines communicate using a high-speed LAN. Distributed range queries were investigated in previous research mainly from the P2P point of view. In [4] *Skip Graphs* were introduced (an extension of skip lists) connecting the elements between the nodes forming a balanced tree providing great resilience, while supporting search/insertion/deletion algorithms. In [1], another alternative was implemented that uses hash tables, which are very efficient for exact-match searching, to perform range queries by using an order-preserving peer to peer indexing.

In addition to P2P systems, research in distributed data management involves the exploitation of multiple resources that form a cluster or data center. Platforms like Apache Hadoop [16] enable application development by hiding low-level operations such as fault tolerance. A rapidly evolving system, that complements the Hadoop ecosystem, is Apache Spark [17] which offers significant performance improvements in comparison to vanilla MapReduce.

A. The MapReduce Model

MapReduce [15] is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster (collection of compuservers or nodes), with a designated node as master

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00073

and the other nodes designated as workers. It can usually be divided into a five-step computation.

1) Partition: input is being split and assigned to each worker.
2) Map: each worker node applies the map function to its local data, and writes the output to a temporary storage.
3) Shuffle: worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node.
4) Reduce: worker nodes now process each group of output data, per key, in parallel.
5) Join Results: workers combine their local output data to create the final output result.

B. The Spark Environment

Apache Spark [3] is a fast and general engine for big-data processing. It is the evolution of the Hadoop framework [2]. Hadoop is the open-source implementation of the MapReduce model and is widely used for distributed processing among multiple nodes. Spark's in-memory data engine can perform tasks up to one hundred times faster than Hadoop, when compared with jobs that require the writing of their state back out to disk between stages. Even jobs where the data cannot be completely contained within memory tend to be around 10 times faster than their Hadoop counterpart. Spark also includes a stack of libraries that combine SQL, streaming, machine learning and graph processing in a single engine. Spark makes easy to build distributed applications in Java, Python, Scala and R. The applications are translated into MapReduce jobs and run in parallel. Furthermore, Spark can access different data sources, such as HDFS or HBase and can be used to process datasets bigger than the cluster's aggregate memory.

Spark uses *Resilient Distributed Datasets* (RDDs) as its fundamental data organization scheme. An RDD is an immutable (i.e., read-only) distributed collection of objects. The datasets are divided into partitions, which are further computed on different nodes of the cluster. However, *Data Frames* (DFs) are also supported that provide more rich smantics and also provide additional optimizations for running SQL queries over distributed data. In this paper, we focus on the RDD API since we are interested in providing faster search capabilities at the partition level.

C. Motivation and Contributions

In this paper, we develop an indexing scheme to organize the data in order to support efficient execution of range queries in Apache Spark. Our proposal is based on the *Interpolation Search Tree* (IST) [5] data structure which has proven to be extremely efficient. We show that by substituting the ordinary linear scan of each data partition with an IST-based organization, significant performance improvements may be achieved. The intuition behind the proposed approach is that in many realistic scenarios data need to be filtered during processing in order for the values to be contained in specific intervals (ranges). In such a case, by simply performing a linear scan is expected to bring unnecessary computational costs.

We have tested our solution against two of the baseline ones used by Spark in a cluster of 32 physical machines running Hadoop and Spark. Our results show that the IST-based scheme is significantly more efficient and also it scales well by increasing the number of Spark Executors.

An acceptable solution for processing range queries in large-scale decentralized environments must scale in terms of the number of nodes and the number of data items stored. The available solutions for architecting such large-scale cloud management systems are inadequate for our purposes, since at the envisaged scale (trillions of data items at millions of nodes) the classic linear or even the logarithmic complexity (for point queries) offered by these solutions is still too expensive. Also, for range queries, it is even more disappointing. Until now, all available Spark built-in solutions incur large overheads with respect to all operations (search, insertion/deletion of items). In this work, we aim at achieving sub-logarithmic complexity for all the above operations. However, since Big Data platforms are mainly used for analysis (and not for arbitrary insertions/deletions) we focus on the performance of range queries over a large collection of values.

Algorithm 1 Search(x,S)

```
1:  left = 1
2:  right = n
3:  next = k ∈ [left, right]
4:  while x ≠ S[next] and left < right do
5:      if x ≤ S[next] then
6:          right = next − 1
7:      else
8:          left = next + 1
9:      end if
10:     next = k ∈ [left, right]
11: end while
12: if x = S[next] then
13:     print('Success')
14: else
15:     print('Fail')
16: end if
```

Let $S = \{X_i, 1 \leq i \leq n\}$ be an ordered set of n elements. Algorithm 1 provides the pseudocode for (the very common in data processing) search operation.

If $next = left + 1$, then Algorithm 1 is a linear searching routine with $O(n)$ worst-case complexity. If $next = \left\lfloor \frac{right+left}{2} \right\rfloor$, then Algorithm 1 is a binary searching routine with $O(\log n)$ worst-case time. If $next = \left\lfloor \frac{x-S[left]}{S[right]-S[left]}(right - left) \right\rfloor + left$, then Algorithm 1 is an interpolation searching routine for which further time improvements can be obtained if the input elements follow some specific classes of distributions. In particular, for elements generated according to the uniform distribution, the interpolation searching routine achieves $\Theta(\log \log n)$ expected time,

and this time bound holds for the extended class of regular distributions [8]. A natural extension is to adapt interpolation search into dynamic data structures, that is, data structures which support insertion and deletion of elements in addition to interpolation search. In [5], the first such dynamic structure, called Interpolation Search Tree (IST), was presented.

In this paper, we present SPIS (SPark-based Interpolation Search), a dynamic tree structure that constitutes an implementation of IST in Spark's environment, aiming at rapid processing of range queries in large-scale data applications. The rest of the work is organized as follows. In the next section, we present some fundamental concepts related to our research. Section III present out main contribution whereas Section IV reports some representative experimental results. Finally, Section V concludes the paper and discusses briefly future work in the area.

II. Preliminaries

In this section, we present some important concepts related to our research. In particular, we explain the way the Interpolation Search Tree works and present its main characteristics. The Interpolation Search Tree (IST) is a multi-way tree, where the degree of a node depends on the cardinality of the set stored below it [5]. It requires $\mathcal{O}(n)$ space for an element set of cardinality n. More precisely, the degree of the root is $\Theta(n^a)$, for some constant $0 < a < 1$. The root splits the set into $\Theta(n^{1-a})$ subsets. The children of the root have degree equal to $\Theta(n^{(1-a)a})$.

Each node u in the IST is associated with a REP_u array, which contains a sample of the subset of elements that is stored below u, a variable S_u, which denotes the size of the subset, and a variable C_u which counts how many insertions/deletions have been performed since the last rebuilding (of the IST) that involved u.

The search algorithm (Algorithm 2) performs interpolation search (by using Algorithm 1 with the appropriate next value as described in Section 1) in the REP array of every node of the tree, starting from the root, in order to locate the subset in which the search should be continued. The expected search time is $\mathcal{O}(\log\log(n))$.

Algorithm 2 Search element into IST

```
procedure IST_SEARCH(Input : X)
    i = Search(X, REP_root)
    let node u be the i-th child of root
    while u ≠ leaf do
        i = Search(X, REP_u)
        let u be the i-th child of u
    end while
    return u
end procedure
```

The insertion algorithm (Algorithm 3) uses the search algorithm to locate the correct position and then adds a marked leaf. After inserting a marked leaf, the path from that marked leaf to the root is followed increasing each node's C variable by one. It finds the highest node, let it be r, that violates the rebuilding condition ($C_r > S_r/4$) and rebuilds the subtree rooted at r. Only after rebuilding, the marked leaf is unmarked.

Algorithm 3 Insert element into IST

```
procedure INSERT(Input : X)
    Leaf node u = IST_Search(X)
    Add new marked Leaf node to the parent of u
    Increase the counters from u to root
    Let r be the highest node such that C_r > Size_r/4
    if r exists then
        Rebuild the tree rooted at r
    end if
end procedure
```

The deletion algorithm (Algorithm 4) works in a similar manner. It uses the search algorithm to locate (if it exists) the leaf we want to delete, and then marks it. The path from that marked leaf to the root is followed, increasing each node's C variable by one. It finds the highest node, let it be r, that violates the rebuilding condition ($C_r > S_r/4$) and rebuilds the subtree rooted at r. Only after rebuilding, the marked leaf is removed.

Algorithm 4 Delete element from IST

```
procedure DELETE(Input : X)
    Leaf node u = IST_Search(X)
    if u_value = X then
        Mark leaf u
        Increase the counters from u to root
        Let r be the highest node such that C_r > Size_r/4
        if r exists then
            Rebuild the tree rooted at r
        end if
    end if
end procedure
```

Any rebuilt subtree is an ideal Interpolation Search subtree. A rebuilding that starts from the root is called global rebuilding and makes the whole IST ideal. The amortized insertion and deletion cost is $\mathcal{O}(\log n)$; the expected amortized insertion and deletion cost is $\mathcal{O}(\log\log(n))$ [5].

III. IST in Spark

In this section we present the implementation of IST in the Spark environment, using Scala.

Let A be our input file that contains our data set (for this example we will use random integer numbers). Suppose A has one number per line (if not, then we can transform it to the desired format).

We create an RDD named $inputRDD$ by referencing A (line 1 of algorithm 5). Then we transform our $inputRDD$ to $integerRDD$ using a map function to convert each element from string to integer (as shown in line 2 of algorithm 5). Using spark's built-in sortBy method we sort the elements of $integerRDD$ and transform it to T (line 3 of algorithm 5).

Algorithm 5 Simple RDD transformation example

1: $val\ inputRDD = sc.textFile(\text{"}hdfs://url/to/A\text{"})$
2: $val\ integerRDD = inputRDD.map(\_.toInt)$
3: $val\ T = integerRDD.sortBy[Int](x => x)$

After sorting is completed, each partition of T holds roughly the same number of sorted elements. Assume T is divided in K partitions.

Algorithm 6 IST building

1: **procedure** BUILD IST($Input$:Sorted RDD T)
2: **for all** ($T_i \in T$) **do**
3: new $IntepolationSearchTree$ object
4: new root node
5: **for all** ($x_i^a \in T_i$) **do**
6: new leaf node r
7: $r_{value} = x_i^a$
8: Insert r into the tree
9: Update counters
10: **end for**
11: Rebuild tree at the root
12: **end for**
13: **end procedure**

Let T_i denote the i-th partition of T, $|T_i|$ the size of the i-th partition, x_i^a the a-th element and let x_i^F, x_i^L denote the first and last element of the i-th partition respectively, where $1 \leq i \leq K$ and $1 \leq a \leq |T_i|$.

Then, $max\{T_i\} \leq min\{T_j\}$, $\forall\ 1 \leq i < j \leq K$, and $x_i^a \leq x_i^b$, $\forall\ 1 \leq a < b \leq |T_i|$. Thus, $x_i^F = min\{T_i\}$ and $x_i^L = max\{T_i\}$.

The next step is to create an IST on each partition. Instead of using the insertion algorithm to add the elements one by one in the tree, we use a bulk-insertion to insert all sorted elements in the tree, and globally rebuild it from the root. This way, we need only one rebuilding to make an ideal IST for each partition. The procedure of IST building is shown in Algorithm 6. Note that the IST object for the specific partition has to fit in the memory of each Executor. However, this issue can be resolved since we can split our input dataset in a larger number of partitions if necessary.

Algorithm 7 Searching element in Spark's IST

procedure DISTRIBUTED SEARCH($Input : X$)
for all ($T_i \in T$) **do**
if (($X \geq x_i^F$) AND ($X \leq x_i^L$)) **then**
$Y = IST\_Search(X)$)
else
$Y = null$
end if
$return\ Y$
end for
end procedure

We can now search/insert/delete elements in the IST of each partition. Searching is quite simple: we query each partition with a single value.

- If the value is inside the partition's interval, perform the searching algorithm, and return the corresponding leaf and the number of the partition.
- If the value isn't inside the partition's interval, return null.

Thus only one partition performs the searching algorithm, while the rest return null. Algorithm 7 shows the searching procedure.

The insertion algorithm works as follows. Distributed Search is performed, which returns the partition and the position the leaf has to be inserted. Then, the classic Insertion algorithm is performed. Once again, only one partition performs the insertion algorithm. Algorithm 8 shows the insertion procedure.

Algorithm 8 Inserting element in Spark's IST

procedure DISTRIBUTED INSERT($Input : X$)
$Y = Distributed\ Search(X)$
In the tree and position that Y denotes, perform:
$Insert(X)$
end procedure

Algorithm 9 Deleting element in Spark's IST

procedure DISTRIBUTED DELETE($Input : X$)
$Y = Distributed\ Search(X)$
In the tree and position that Y denotes, perform:
$Delete(X)$
end procedure

The deletion algorithm works in a similar manner. Distributed Search is performed which returns the partition and the leaf to be deleted. Then, the classic deletion algorithm is performed. Algorithm 9 shows the deletion procedure.

Now, we turn to the range query and explain how it works. Given two values min and max, we want to get all the elements that their key is inside the $[min, max]$ interval. We query each partition with min and max:

- If min is inside the i-th partition's interval [x_i^F,x_i^L], then the search algorithm is performed and we get the corresponding element B.
  - If max is inside the partition's interval, then the search algorithm is performed again and we get the corresponding element E. All elements in-between B and E are returned.
  - Else if max isn't inside the partition's interval, then E is assigned to the last element of the partition (x_i^L). All elements in-between B and E are returned.
- Else if max is inside the i-th partition's interval, then the search algorithm is performed and we get the corresponding element E. B is assigned to the first element

of the partition (x_i^F). All elements in-between B and E are returned.
- Else if $min < x_i^F$ and $max > x_i^L$, the whole partition is returned.
- Else if none of the above happens, zero is returned.

IV. Experimental Evaluation

In this section, we report on the experimental evaluation of the IST implementation in the Spark environment. All experiments have been conducted on a cluster with 32 physical machines running Hadoop 2.7 and Spark 2.1.0. We have used synthetic datasets for the experimentation with different cardinalities. The dataset contains one-dimensional values that were produced by a mixture of Gaussians. The selection of this dataset was based in the fact that many real-world datasets contain clusters and are frequently modeled as Gaussian mixtures. Figure 1 presents such a distribution in the two-dimensional space. In our experiments, we have used the projection in the x and y axis to construct the one-dimensional dataset used in our performance evaluation.

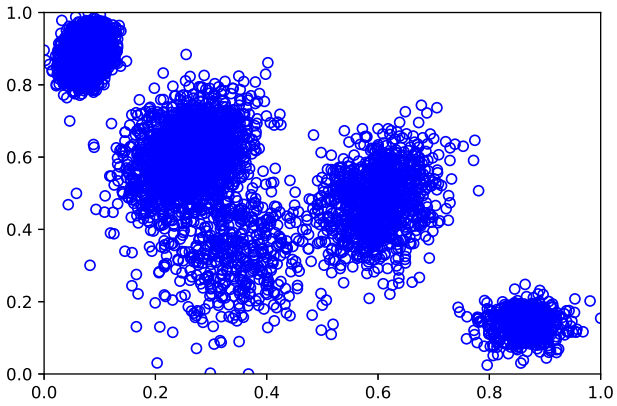

Fig. 1. Dataset distribution.

A. Runtime Performance Tests

First, we report our results for the evaluation of the runtime performance of the proposed approach. The following algorithmic techniques were carried out in conducting our experiments. Each technique corresponds to a different query mechanism. We start with our technique, called Algorithmic Technique A.

Algorithmic Technique A

- Create an RDD by referencing the dataset file. Map its contents to Float. Sort and partition the RDD to the Executors as shown in Algorithm 5.
- Generate an array of 5000 random float pairs in the interval $[0, 1]$ to perform range search queries. The array is created at the Driver node and broadcasted to all Executors in the cluster.
- Create an IST on each partition (using Algorithm 6).
- Execute 5000 Range Queries on the IST of each partition using the pairs of the array as input parameters, and monitor the total runtime.

Next we measure Spark's built-in features, called F and M Algorithmic Techniques. On the same datasets, we execute range queries using two approaches: the first approach (F) uses simple filtering on unsorted RDDs (using the $filter$ transformation), whereas the second one (M) uses $mapPartitions$ and $find$ transofmations on sorted RDDs.

The $filter$ function works as follows:

```
RDD2 = RDD1.filter((function p)=>Boolean)
```

An element of $RDD1$ is assigned to $RDD2$ only if it satisfies the predicate (function p). In our case each element has to be inside the $[min, max]$ interval.

These techniques are summarized below:

Algorithmic Technique F

- Create an RDD by referencing the dataset file.
- Generate an array of 100 random float pairs in the interval $[0, 1]$ (to perform range search queries).
- Filter the input RDD using the elements of the array as bounds.

The Algorithmic technique M is similar to A. The RDD is sorted and partitioned to the executors. Each query works like the IST Range Search, checking the partition's interval $[x_i^F, x_i^L]$ and performing the corresponding action. The difference is that instead of using the Search Algorithm on the IST, we apply linear search to determine the relevant elements.

Algorithmic Technique M

- Create an RDD by referencing the dataset file. Map its contents to Float. Sort and partition the RDD to the Executors (like shown in Algorithm 5).
- Generate an array of 5000 random float pairs in the interval $[0, 1$ (to perform range search queries).
- Using $mapPartition$ and $find$ function, perform 5000 range queries on the elements of each partition.

Again, we measure the total runtime of the whole processes. The difference between the algorithmic techniques F and M is that F's partition's dataset size can be larger than the Executor's memory since it is performed using iterators. M's partition dataset size on the other hand must be able to fit inside the Executor's memory in order to determine the first (x_i^F) and the last (x_i^L) element respectively. In case the size of a partition is larger than the size of the Executor memory, we can split the current partition into smaller parts and work in each part separately.

For comparison purposes, we have used the elapsed time per query for all algorithmic techniques. The corresponding

results are given in Table I. The runtime results correspond to 32 Spark Executors.

TABLE I
RUNTIME PERFORMANCE COMPARISON.

Input Size $\times 10^6$	Number of Partitions	A (ms)	F (s)	M (ms)
10	64	7.88	2.58	13.46
20	64	8.74	5.22	19.58
100	128	12.2	2.46	83.12
200	128	14.5	4.28	125.04
1000	512	56.54	15.10	507.14

We observe that algorithmic technique A is significantly faster than Spark's built-in techniques F and M. The difference is more evident for bigger datasets.

B. Scalability Results

The second set of experiments was carried out in order to test the scalability of our proposed organization scheme. Using an input dataset of ten million float numbers, we followed the procedure described below while gradually adding more Executors to our cluster. The following process has been followed:

- Create an RDD by referencing the dataset file. Map its contents to Float. Sort and partition the RDD to the Executors (like shown in Algorithm 5).
- Generate an array of 5000 random float pairs in the interval $[0, 1]$ (to perform range search queries). The array is created in the master node and broadcasted to the cluster.
- Create an IST on each partition (using Algorithm 6).
- Execute 5000 Range Queries on the IST of each partition using the pairs of the array as input parameters, and monitor the total runtime.

Time performance evaluation results (in seconds) reported in Fig. 2 correspond to average runtime for each experiment.

The total runtime also includes sorting time. Sorting is performed by three Executors (note that the file is stored in three partitions in the HDFS) before being split across the cluster. That is the reason behind the significant improvement in the first three tests. After that, the runtime is steadily decreases which shows the good scalability of our proposed approach.

V. CONCLUSIONS

In this work we have presented SPIS, a scalable interpolation search file structure for Spark, implemented in Scala. The main motivation behind this research is that the standard filtering mechanism that Spark provides is inefficient when specific queries must be executed. In particular, executing multi-dimensional range queries using the standard Spark mechanism, requires a large number of comparisons which leads to performance degradation. In this paper, we apply a simple IST-based data organization which leads to fast range query execution and allows scalable range query processing in large amounts of data.

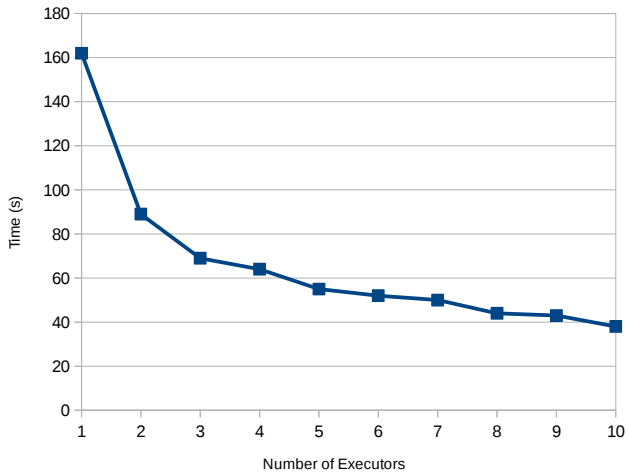

Fig. 2. Scalability performance.

An interesting direction for future work is the support of efficient range query processing mechanisms in SQL query execution over Spark data frames. Spark supports SQL queries through the SQL library. We believe that the incorporation of efficient range query processing techniques will contribute in reducing the overall cost of processing SQL queries. Another interesting direction is the performance evaluation of range query processing in high dimensional datasets. In that case, it is expected that data organization methods, like the one presented in this paper, are expected to improve performance significantly.

REFERENCES

[1] Maha Abdallah and Hung Cuong Le. Scalable Range Query Processing for Large-Scale Distributed Database Applications, *Proceedings of IASTED PDCS*, 2005.

[2] Apache Hadoop. *The apache software foundation: Hadoop homepage.* `http://hadoop.apache.org/`, 2015.

[3] Apache Spark. *The apache software foundation: Spark homepage.* `http://spark.apache.org/`, 2015.

[4] Aspnes J. and Shah G. *Skip Graphs", Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp.384-393, Baltimore, MD, 2003.*

[5] K. Mehlhorn and A. Tsakalidis. *Dynamic Interpolation Search. Journal of the ACM, 40(3):621-634, July 1993.*

[6] Pearl, Y., Itai, A., and Avni, H. *Interpolation Search-A* $\log \log N$ *search. Commun.ACM 21, 7, (1978), 550554.*

[7] Peterson W. W. *Addressing for random storage. IBM J Res. Develop. 1 (1957), 131-132*

[8] Willard. D. *Maintaining dense sequential files in a dynamic environment. In Proceedings of the 14th Symposium on Theory of Computing. (San Fransisco, Calif., May 5-7). ACM, New York, 1982. pp. 114-122.*

[9] Jagadish H.V., Ooi B.C. and Vu Q.H. *Baton: a Balanced Tree Structure for Peer-to-peer Networks, Proceedings 31st International Conference on Very Large Data Bases (VLDB), pp.661-672, Trondheim, Norway, 2005.*

[10] Jagadish H.V., Ooi B.C., Tan K.L., Vu Q.H. and Zhang R. *Speeding up Search in P2P Networks with a Multi-way Tree Structure", Proceedings ACM International Conference on Management of Data (SIGMOD), pp.1-12, Chicago, IL, 2006.*

[11] Karger D., Kaashoek F., Stoica I., Morris R. and Balakrishnan H. *Chord: a Scalable Peer-to-peer Lookup Service for Internet Applications", Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp.149-160, San Diego, CA, 2001.*

[12] Ratnasamy S., Francis P., Handley M., Karp R. Shenker S. *A Scalable Content addressable Network", Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp.161-172, San Diego, CA, 2001.*

[13] Goodrich M.T., Nelson M.J. and Sun J.Z. *The Rainbow Skip Graph: a Fault-Tolerant Constant-Degree Distributed Data Structure", Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp.384-393, Miami, FL, 2006.*

[14] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, Scott Shenker. *prefix hash tree", Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing table of contents (Brief announcement), pp.368-368, Newfoundland, Canada, 2004.*

[15] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters Commun. ACM 51(1): 107-113 (2008)*

[16] Tom White. *Hadoop: The Definitive Guide*, O'Reilly, 2015

[17] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica. *Apache Spark: A Unified Engine for Big Data Processing*, Commun. ACM 59(11), pp.56-65, 2016.