HCTrie: A Structure for Indexing Hundreds of Dimensions for Use in File Systems Search

Yasuhiro Ohara
Storage Systems Research Center & Computer Science Department
University of California, Santa Cruz
Santa Cruz, California 95064
Email: yasu@cs.ucsc.edu

***Abstract*—Data management in large-scale storage systems involves indexing and search for data objects (e.g., files). There are hundreds of types of metadata attributed to the data objects: examples include environmental settings of photograph files and simulation configurations for simulation output files. To provide intelligent file search that uses file metadata, we introduce a novel search structure called Hyper-Cube Trie (HCTrie), that can handle a few hundred dimensions of data attributes. HCTrie can utilize the differences in many dimensions effectively: candidates can be pruned based on differences in all dimensions. To the best of our knowledge, this is the first approach to restrain the memory growth to a linear scale against the number of dimensions, when multiple dimensions are indexed at the same time. Our prototype has successfully indexed five million data entries with one hundred attributes in a single data structure. We show that HCTrie can outperform MySQL in range search where ranges for less than 100 dimensions are specified in the search query.**

I. INTRODUCTION

Data management is becoming a challenging problem, especially in High Performance Computing (HPC) where huge amounts of scientific data must be handled. The number of files that must be handled reaches billions [16], making data management difficult. Data are stored in file systems, but the users cannot always remember the file path name [22]. Consequently, HPC scientists are required to manage their simulation results using external notes, including long complex directory and file names [14]. This is becoming a serious issue as we move toward exascale computing where hundreds of exabytes of data and billions of files exist in the file system.

To tackle this, there is a growing demand to search by metadata, rather than by hierarchical file name [17]. Scientific data typically has many metadata fields, such as configuration parameters of the measurement and/or the simulation that derived the data. For example, the WISE All-Sky Data Release includes 285 fields (columns) in the data set [18]. There is a demand to search a data entry or file using this *scientific metadata*.

There are past proposals to overcome this issue by the use of a database management system (DBMS) [1], [22]. DBMSes can construct an index on each column independently. Typically, some of the columns are selected and indexed based on the specific query trend for the target application. However, since using a DBMS to index file systems is not a widely used technique, such a query trend is not known. Without the search query trend, a DBMS that constructs indices only on some columns cannot perform all search functions efficiently. As an example of inefficient DBMS performance, we note that a query involving join operations may take 400–1000 seconds [9].

In this paper we tackle the problem of finding an efficient data structure for file search using scientific metadata and propose a novel search structure, called *HCTrie*. Our goal is an environment where there are a few hundred attributes for every few billion (10^{11}) files in the file system. We will use the attributes for the purpose of file search; this is in general a multidimensional information search. These few hundred attributes correspond to the number of dimensions in search. We assume that the data structure as a whole would not be able to fit into the primary memory, thus we consider the use of the memory in secondary storage.

Intuitively, there seems to be no reason to limit the number of dimensions d to a certain (low) number, even when we require no performance degradation. No matter how many dimensions, the complexity (both in time and space) should be bounded by the number of data entries n, if each internal branch node in the structure divides the n data in subgroups. Here, we want to remove the limitation on the number of dimensions and at the same time, leverage the locality and the differences in the rich attributes.

Our primary goal is to provide a data structure with feasible growth in both time and space complexity against the number of dimensions, d. It is desirable to have a data structure that is not affected by the number of dimensions. Our secondary goal is efficient interaction between dimensions. We want to filter out the candidates in one dimension and at the same time effectively filter out that same candidates in other dimensions as well. Also, if we query entries in a combination of ranges in a few dimensions, we do not want entries outside the range to be returned, as much as possible.

HCTrie scales well in the number of dimensions, utilizes the differences in the combination of dimensions, and is efficient in multi-dimensional range search in that the range can be handled in an aggregate unit of search space hypercube. The prototype implementation in this paper shows results for range search on one hundred dimensions, and the implementation supports up to 512 dimensions.

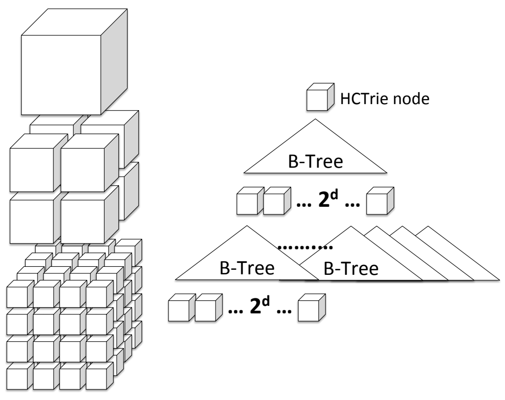

Fig. 1. HCTrie: the search space is divided in each dimension to construct the hypercube search spaces in each level of the tree. Every hypercube space is registered in the upper level hypercube's B-Tree, enabling a space efficient descendant link array.

II. HCTrie

HCTrie can be thought of as a simple extension of Quadtree [21]. The major difference is that the descendant pointer array is implemented in a B-Tree [7], [15]. This way, only the necessary descendant pointers in the 2^d space are held, although there is the regular B-Tree memory overhead[1]. Furthermore, efficient utilization of the secondary memory storage device (e.g., HDDs) can be achieved with the notion of disk pages.

HCTrie divides the search space into d-dimensional hypercubes. Each hypercube at depth i corresponds to the 2^i-divided interval in each of the d-dimensions. Each hypercube is labeled with a number derived by each decomposition in dimension, whether it is HIGH(1) or LOW(0) within the parent hypercube. If it has only a single dimension, it is equivalent to Trie [10], so HCTrie can be thought of as a Trie on d-dimensional hypercubes.

The notion of 3-D HCTrie is illustrated in Figure 1. The left side of the figure depicts the division of the search space into halves in all dimensions, producing smaller hypercubes in each level of HCTrie. Only the lowest hypercubes in HCTrie (called leaf hypercubes) have pointers to the data entries.

B-Trees serve as the link array from the ascendent hypercube to the descendent hypercubes. This enables efficient memory space usage, without spending the memory for a large number of empty pointers, even with high dimensionality. For example, despite the fact that there are 2^{100} subspaces when $d = 100$, at most n subspaces need to be stored in the B-Tree. (Note, we assume $n \ll 2^d$.) If the division in a level of HCTrie does not sufficiently distinguish the n data entries, the number of subspaces registered in that level's B-Tree is smaller, and the difference is treated in a deeper level of HCTrie.

In order to put a hypercube in the B-Tree, it is necessary to have an order relation between hypercubes. HCTrie simply extracts the i-th bit from keys in all dimensions. Figure 2 illustrates the makelabel() function. The makelabel() function appears in Algorithm 1 in Line 4.

[1]Each storage space overhead within B-Tree nodes is guaranteed to be less than half of the disk page size, except the root node [15].

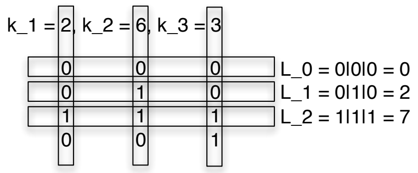

Fig. 2. Makelabel() in HCTrie. Each bit in the dimension keys (i.e., $k_1 = 2, k_2 = 6$, and $k_3 = 3$) constitutes a label of a hypercube search space. For example, L_0 is the label of the corresponding hypercube in the level 0, L_1 is the one in the level 1, and so on.

TABLE I. Notations and Descriptions

Notation	Description
d	the number of dimensions.
w	the maximum length of key in all d dimensions. $w = \max_{i=1}^{d} length(k_i)$
B	the order of B-Tree. (i.e., the number of branches.)
n	the number of data entries.
K, k_i	the search query, i.e., $K = (k_1, k_2, ..., k_d)$, where k_i is the search key in the i-th dimension.
$R = (K^s, K^e)$	the range of the search query. K^s and K^e are the beginning and the end of the range.

A. Algorithms and complexity

Table I describes the notation used in this paper.

Algorithm 1 HCTrie Lookup.

```
1: procedure HCTrieLookup($K = (k_1, k_2, ..., k_d)$)
2:     $i \leftarrow 0$, $h_0 \leftarrow$ HCTrie.root
3:     for ($i < w \wedge h_i$ is not leaf) do
4:         $L_i \leftarrow$ makelabel ($K, i$)
5:         $h_{i+1} \leftarrow h_i$.B-Tree lookup ($L_i$)
6:         $i \leftarrow i + 1$
7:     end for
8:     return $h_i$
9: end procedure
```

Algorithm 2 HCTrie Range Search.

```
1: procedure HCTrieRangeSearch($R = (K^s, K^e)$)
2:     $S \leftarrow \varnothing, Q \leftarrow Q \cup$ HCTrie.root
3:     for all $h_i \in Q$ do
4:         $L_i^s \leftarrow$ makelabel ($K^s$, $h_i$.level)
5:         $L_i^e \leftarrow$ makelabel ($K^e$, $h_i$.level)
6:         $b \leftarrow h_i$.BTreeLeafHead ($L_i^s$)
7:         do
8:             if ($b$.data is a hypercube) then
9:                 $h_{i+1} \leftarrow b$.data
10:                $Q \leftarrow Q \cup h_{i+1}$
11:            else
12:                $S \leftarrow S \cup b$.data
13:            end if
14:            $b \leftarrow h_i$.BTreeLeafNext ($b$)
15:        while $b$.key $\leq L_i^e$
16:    end for
17:    return $S$
18: end procedure
```

Algorithm 1 is the function to look up the corresponding hypercube. It inputs d-dimensional keys $K = (k_1, k_2, ..., k_d)$, and returns the corresponding hypercube, h_i. The algorithm starts at level 0, taking the root hypercube of HCTrie. The maximum depth is bounded by the number of bits in the dimension keys, w. For example, if the "long long unsigned int" is the longest type of field in the d keys, then $w = 64$. At each level i, the i-th bit in the keys are extracted to compose a corresponding hypercube label L_i, as illustrated in Figure 2.

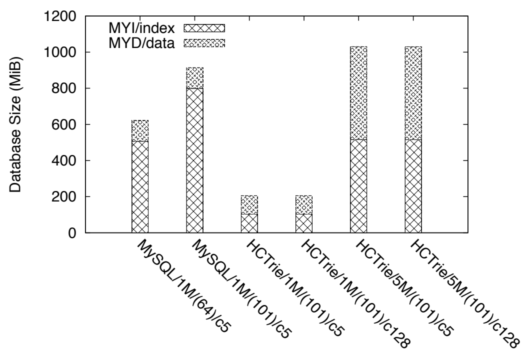

Fig. 3. Database size, including the index and data file size. "MYI/index" shows the index, and "MYD/data" shows the data. The MyISAM engine was used for MySQL. The labels are formatted as "name/# of data entries/(# of columns)/cardinality". "MySQL/1M/(101)/c5" was calculated from "MySQL/1M/(64)/c5", since MySQL supports only a maximum of 64 indices. The difference between the index size of MySQL and HCTrie can be attributed to the number of trees: MySQL has larger number of relatively small trees, and HCTrie has one larger tree. With five million dataset with cardinality 5 and 128, the index file and the data file sizes in HCTrie were both approximately 515 MiB, 1 GiB in total.

L_i is then used to search the B-Tree of the i-th hypercube to find the descendant hypercube (at level $i+1$).

Algorithm 2 shows the range search function. The input R stores the beginning of the range, K^s, and the end of the range, K^e. The range search function conducts a breadth first search from the HCTrie root using the queue Q. Within the breadth first search, the B-Tree in the current hypercube h_i is searched for the label L_i^s that corresponds to the start of the range K^s in the appropriate level (h_i.level). From L_i^s through L_i^e, the B-Tree is traversed while the B-Tree key is in the range. The collected data entries are returned through S.

The worst-case scenario for the space complexity in HCTrie is that every difference in data entries makes a branch of factor 2, and all of these branches are realized by B-Trees which consist of only the root node in all hypercubes. The number of branches (and, equivalently, the number of nodes) in the binary tree that classifies n entries is $(2n-1)$. A disk page is consumed for each of these branches, and the size of the disk page can be derived as Bd, since B d-dimensional keys must be held in a disk page. Hence, the space complexity is $O(Bdn)$. This means that HCTrie scales linearly in growth of both d and n.

Note that HCTrie can benefit from the use of improved versions of B-Trees. For example, a B*-Tree [7] can be used to improve the memory efficiency guarantee, or a Streaming B-Tree [5] can be used to speed up the update and range search performance.

III. Evaluation

A. Random Synthetic Data

We evaluate our prototype of HCTrie against MySQL, on a random synthetic dataset. Figure 4 illustrates the sample random data entry. The random synthetic data consists of a 64-bit unsigned "id" attribute and one hundred 8-bit "ti" attributes

```
id (uint64, 8B): 999
ti1 (int8, 1B): 1 ti2 (int8, 1B): 4 ti3 (int8, 1B): 1 ti4 (int8, 1B): 2
ti5 (int8, 1B): 3 ti6 (int8, 1B): 0 ti7 (int8, 1B): 3 ti8 (int8, 1B): 4
:
ti97 (int8, 1B): 1 ti98 (int8, 1B): 1 ti99 (int8, 1B): 2 ti100 (int8, 1B): 1
```

Fig. 4. A sample data entry from the random synthetic data. Each attribute is shown as a list of attribute name, type, length in bytes, and the value. The value takes a random value from the range [0..4] to simulate the low cardinality of scientific metadata.

("ti" stands for tiny integer). The size of a data entry is thus 108 bytes.

The "id" attribute is a sequentially assigned number, and serves also to indicate the location of data in the data file. For example, the data entry with "id" 100 can be found at the file offset of 10800 bytes. We assume that the "id" is not used in querying a data entry, since to find the location of the data is the major purpose of the index.

Each of the "ti" attributes can take a value from -127 to 127. However in this comparison, we suppress the number of distinct value to 5, simulating the low cardinality of the scientific metadata. The value is random from 0 to 4.

B. Range Search Performance

Figure 5 shows the performance measurement against MySQL, on one million random synthetic data. It was conducted on a host with Intel Xeon E3-1230 (3.20GHz) with 16GB memory and with the Intel SSD SSDSC2CT12. The MySQL version was 5.5.29, using the default configuration. We used the mysql command to measure the query execution time in MySQL, which includes an SQL parse time and communication time. Since we assumed out of index queries, and because MySQL limits the number of indices to 64, we did not create a MySQL index. The InnoDB MySQL engine was used.

In the evaluation, range query searches were issued over one hundred dimensions. Our prototype implementation supports only search ranges that correspond to the binary divided search space boundaries. The range of the query for each dimension was [0..3] when specified, and [0..127] when unspecified.

The x-axis of Figure 5 describes the number of unspecified dimensions. For example, the point $x = 20$ means that the first 20 dimensions are unspecified (i.e., [0..127]), and the latter 80 dimensions are specified with the range [0..3].

With up to 70 unspecified dimensions, HCTrie outperformed MySQL. MySQL took about 1.5 seconds, and HCTrie took less than 0.5 seconds. The line labeled "HCTrie-First1" is the time taken to search the index structure. The line labeled "HCTrie-First2" includes the time taken by the range match check. Since HCTrie may still return candidates that do not fall into the query range, it is necessary to load the entire data entry and check the matching status. In our prototype implementation, when the query range is larger and many candidates are in the range, the loading of data entries impacted the overall performance. For example, the line labeled "HCTrie-First2" in Figure 5 took more time than MySQL at 80 unspecified dimensions with 11,452 candidates. In our case of cardinality

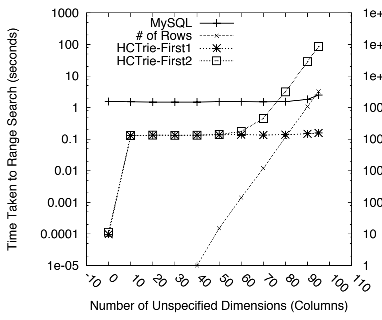

Fig. 5. Range query performance evaluation on one million 100-dimensional data entries with the cardinality 5. x-axis is the number of unspecified dimensions from the first in the dimension order. The rest of the dimensions are specified with the range [0..3]. "HCTrie-First1" shows the time taken to return candidates, and "HCTrie-First2" includes, in addition, the time taken to load the entire data and check the matching status. "# of Rows" indicates the number of results using the right y-axis. MySQL performed at around 1.5 seconds, while HCTrie returned candidates ("HCTrie-First1") in around 0.14 seconds.

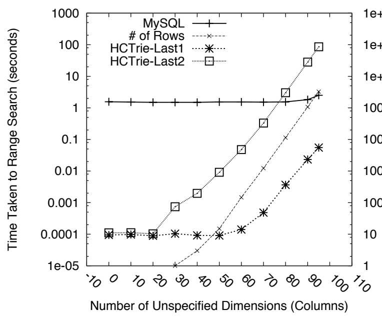

Fig. 6. Range query performance evaluation on one million 100-dimensional data entries with the cardinality 5. x-axis is the number of unspecified dimensions from the last (not first, compared to Figure 5) in the dimension order. The performance is slightly better than Figure 5, because specifying in the first dimensions makes the most significant bits in hypercube label specified, thus enables the fast B-Tree leaf traversal.

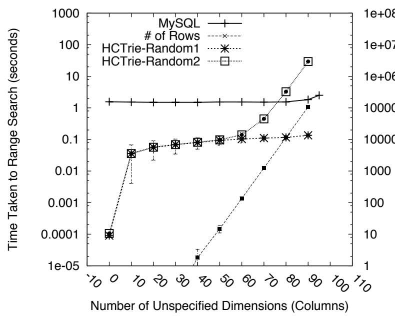

Fig. 7. Range query performance evaluation on one million 100-dimensional data entries with the cardinality 5. x-axis is the number of unspecified dimensions randomly chosen in the dimension order. The averages of 100 times are shown with standard deviations. MySQL data is just projected from the previous figure for comparison. The result is similar and only slightly better than Figure 5.

5 in each dimension, all the candidates found were in the query range.

With 90 unspecified dimensions, only 10 dimensions are specified for the range search query, thus the number of matching data entries increases significantly. The line with the label "# of Rows" indicates the number of matching data entries as the result of the query, using the right y-axis. As the query range becomes larger and the matching entries increase, the performance of HCTrie degrades, due to the aforementioned data loading and the last check. Our prototype implementation took around 29 and 87 seconds for 90 and 95 unspecified dimensions, respectively.

Line "HCTrie-First1" in Figure 5 remains constant at around 0.15 seconds. This shows that for more than 10 unspecified dimensions, HCTrie had to check all the B-Tree leaf slots. In other words, HCTrie only took around 0.15 seconds to check one million data entries to see if the differentiating part indexed for the data entry matched the query range. Performance is affected only by the number of candidates found. Note that candidates out of the query range may be returned, depending on the characteristics of the dataset.

Overall, if more than 30 out of 100 dimensions can be specified, HCTrie is more beneficial to use than MySQL, in our search case. Our query was loose: the range of [0..3] is specified where only [0..4] is possible. Still, differences in different dimensions can effectively be used to prune the candidates quickly, and thus contributes to the search performance.

Furthermore, the line "HCTrie-First1" shows that the index structure returns candidates quickly and indicates that what takes time is the data loading and the last check. We may be able to use HCTrie similar to Bloom filter [13], where candidates can be narrowed down quickly.

Figure 6 illustrates an experiment with the unspecified dimensions at the end. For example, $x = 40$ means that the first 60 dimensions are specified (i.e., [0..3]) and the last 40 dimensions are unspecified (i.e., [0..127]). Since the first dimensions are specified, the most significant bits of the hypercube labels are specified, thus contributing to the B-Tree lookup speed. This, in turn, also contributes to the overall performance.

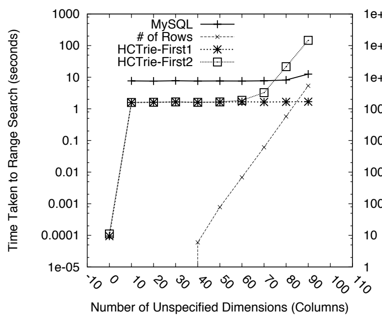

Fig. 8. Range query performance evaluation on five million (not one million, compared to previous experiments) 100-dimensional data entries with the cardinality 5. x-axis is the number of unspecified dimensions from the first in the dimension order. MySQL performed at around 7.6 seconds, while HCTrie returned candidates (HCTrie-First1) in around 1.7 seconds.

Figure 7 shows the randomly chosen unspecified dimension case. The results were only slightly better than the worst case (i.e., Figure 5 where first portion was unspecified) in our experiment.

Figure 8 illustrates the case of Figure 5 with five million data entries. It shows that the advantage of HCTrie is approximately 6 seconds in most cases, which is a further improvement compared to Figure 5.

IV. Related Work

An extensive list of access methods is summarized by Gaede and Günther [11]. Past studies on multidimensional search structure mainly focus on only a few dimensions, typically two or three, and generally less than ten. They typically have 2^d growth in space for d-dimensions, like a natural extension of Quadtree would, because of their need to provision a pointer array for all possible numbers of d-dimensional spaces.

B-Trees are a one-dimensional access method that is widely used today, both in DBMSes (e.g., MySQL) and in file systems (e.g., Btrfs [20]). The fact that only one dimension can be indexed leads to the problem of composite indexing in a DBMS, where the combinations of columns need to be considered.

Binary Space Partitioning (BSP), a family of structures or a notion of dividing a space by hyperplanes, is widely used in computer graphics [19]. Also, families of Quadtree [21] and R-Tree [3], [4], [12] handle multidimensional keys. However, they target only two or three dimensions, and do not suit indexing one hundred dimensions.

While a family of Quadtree [21] conducts a simple and intuitive division of multi-dimensional space, it supports only a low number of dimensions for the multi-dimensional search, such as two or three dimensions. If the number of dimensions, d, increases by hundreds, the size of the pointer array in Quadtree grows with 2^d (e.g., $2^{100} \approx 10^{30}$), which does not scale.

K-d-Trees [6] have a good scalability against the number of dimensions d: they have the advantage that the number of dimensions does not impact scalability. However, they lose the data entry's locality, hence it deteriorates on range search performance. The average cost of range search in a K-d-Tree is dominated by the overwork (i.e., a redundant and theoretically avoidable cost), and the complexity has been given by Duch and Martínez [8].

B-Trie [2] combines B-Trees and Trie. However, it assumes only a single dimensional string key, and does not support multidimensional keys.

V. Conclusion

We have presented a novel multidimensional search structure called HCTrie. HCTrie can support a hundred of dimensions, which is progress compared to today's support of only several dimensions in multidimensional search structures.

On a 100-dimensional random synthetic dataset which simulates the low cardinality of scientific metadata, HCTrie showed better performance than MySQL, when a range is specified in more than 30 dimensions.

This novel structure is expected to be beneficial in file system search, in which we will combine many types of functions to provide a new type of file search.

Acknowledgment

This research was supported in part by the NSF under awards IIP-0934401 and CCF-0937938, and by the Dept. of Energy under awards DE-FC02-10ER26017/DE-SC0005417. We also thank the industrial sponsors of the Storage Systems Research Center for their generous support.

We also thank the faculty and students of SSRC for their help and guidance, especially Darrell Long, Ethan Miller, Aleatha Parker-Wood, Brian Madden, and Christina Strong, for their valuable comments.

References

[1] S. Ames, M. Gokhale, and C. Maltzahn, "QMDS: A file system metadata management service supporting a graph data model-based query language," in *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, Jul. 2011, pp. 268–277.

[2] N. Askitis and J. Zobel, "B-tries for disk-based string management," *The VLDB Journal*, vol. 18, no. 1, pp. 157–179, Jan. 2009.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *ACM SIGMOD*, 1990, pp. 322–331.

[4] N. Beckmann and B. Seeger, "A revised R*-tree in comparison with related index structures," in *ACM SIGMOD*, 2009, pp. 799–812.

[5] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming b-trees," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '07, 2007, pp. 81–92.

[6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, September 1975.

[7] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[8] A. Duch and C. Martínez, "Improving the performance of multidimensional search using fingers," *J. Exp. Algorithmics*, vol. 10, Dec. 2005.

[9] A. Floratou, J. M. Patel, W. Lang, and A. Halverson, "When free is not really free: What does it cost to run a database workload in the cloud?" in *TPCTC*, ser. Lecture Notes in Computer Science, vol. 7144. Springer, 2011, pp. 163–179.

[10] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960.

[11] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, Jun. 1998.

[12] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984, pp. 47–57.

[13] F. Hao, M. Kodialam, and T. V. Lakshman, "Building high accuracy bloom filters using partitioned hashing," in *ACM SIGMETRICS*, 2007, pp. 277–288.

[14] S. Jones, C. Strong, A. Parker-Wood, A. Holloway, and D. D. E. Long, "Easing the Burdens of HPC File Management," in *Proceedings of the 6th Parallel Data Storage Workshop (PDSW '11)*, Nov. 2011.

[15] D. E. Knuth, *The art of computer programming, volume 3: sorting and searching*. Addison-Wesley-Longman, 1973.

[16] P. Kogge *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascal e Systems," DARPA / IPTO, Tech. Rep., 2008.

[17] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009, pp. 153–166.

[18] NASA, "WISE All-Sky Release Explanatory Supplement: Data Products," http://wise2.ipac.caltech.edu/docs/release/allsky/expsup/sec2_2a.html, 2012.

[19] M. S. Paterson and F. F. Yao, "Optimal binary space partitions for orthogonal objects," *Journal of Algorithms*, vol. 13, no. 1, pp. 99–113, 1992.

[20] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree Filesystem," IBM, Tech. Rep., Jul. 2012.

[21] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, Jun. 1984.

[22] M. Seltzer and N. Murphy, "Hierarchical file systems are dead," in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, 2009.