DART: Distributed Adaptive Radix Tree for Efficient Affix-based Keyword Search on HPC Systems

Wei Zhang
Texas Tech University
Lubbock, Texas
X-Spirit.zhang@ttu.edu

Houjun Tang
Lawrence Berkeley National Laboratory
Berkeley, California
htang4@lbl.gov

Suren Byna
Lawrence Berkeley National Laboratory
Berkeley, California
sbyna@lbl.gov

Yong Chen
Texas Tech University
Lubbock, Texas
yong.chen@ttu.edu

ABSTRACT

Affix-based search is a fundamental functionality for storage systems. It allows users to find desired datasets, where attributes of a dataset match an affix. While building inverted index to facilitate efficient affix-based keyword search is a common practice for standalone databases and for desktop file systems, building local indexes or adopting indexing techniques used in a standalone data store is insufficient for high-performance computing (HPC) systems due to the massive amount of data and distributed nature of the storage devices within a system. In this paper, we propose Distributed Adaptive Radix Tree (DART), to address the challenge of distributed affix-based keyword search on HPC systems. This trie-based approach is scalable in achieving efficient affix-based search and alleviating imbalanced keyword distribution and excessive requests on keywords at scale. Our evaluation at different scales shows that, comparing with the "full string hashing" use case of the most popular distributed indexing technique - Distributed Hash Table (DHT), DART achieves up to 55× better throughput with prefix search and with suffix search, while achieving comparable throughput with exact and infix searches. Also, comparing to the "initial hashing" use case of DHT, DART maintains a balanced keyword distribution on distributed nodes and alleviates excessive query workload against popular keywords.

CCS CONCEPTS

• **Computing methodologies** → *Parallel algorithms*;

KEYWORDS

distributed search, distributed affix search, distributed inverted index

ACM Reference Format:
Wei Zhang, Houjun Tang, Suren Byna, and Yong Chen. 2018. DART: Distributed Adaptive Radix Tree for Efficient Affix-based Keyword Search on HPC Systems. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3243176.3243207

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243207

1 INTRODUCTION

Affix-based keyword search is a typical search problem, where resulting data records match a given prefix, suffix, or infix. For contemporary parallel and distributed storage systems in HPC environments, searching string-based metadata is a frequent use case for users and applications to find desired information. For instance, the affix-based keyword search on string-based identifiers, such as "`name=chem*`" and "`date=*/2017`", is a common requirement in searching through the metadata among terabytes to petabytes of data generated by various HPC applications [1, 7, 22, 23, 27, 32].

It is a common practice to build trie-based inverted index to accelerate affix-based keyword search. There are several standalone trie-based indexing data structure, such as prefix B-tree [8], Patricia tree [19], or adaptive radix tree (ART) [21]. But none of these standalone data structures can address the metadata search problem of an HPC system. The massive amount of data on HPC system will cause leaf node expansion in these trie-based data structures and hence exhausts limited memory resources while diminishing the overall efficiency. Thus, it is preferable to distribute such trie-based inverted index onto multiple nodes to utilize the data parallelism and to reduce tree traversal overhead.

However, it is challenging to build distributed inverted indexes on HPC systems. One challenge comes from the distributed nature of HPC storage systems which involves communication overhead. Some recent research efforts (e.g. [40]) explored the practice of indexing metadata by storing metadata into multiple distributed instances of database management systems, such as SQLite[42], MySQL [31], PostgreSQL [33] or MongoDB [30]. With such approach, queries have to be sent to all participating database instances and the results have to be collected from all these instances. Such broadcasting and reduction operations cause heavy communication overhead and hence reduce search efficiency. The same problem can also be found in some cloud computing solutions, such as Apache SolrCloud [2] and ElasticSearch [11], in building and using distributed inverted indexes. Moreover, these approaches do not scale well in HPC environments and it is difficult to ensemble them into a HPC storage system as a pluggable component. Also, as a common practice for managing distributed index in many HPC storage systems (such as SoMeta [43], IndexFS [35], FusionFS [49], and DeltaFS [50]), distributed hash table (DHT) [47] has been used to build inverted index for metadata [48]. However, when it comes

to affix-based keyword search, the application of DHT on each entire keyword may still result in query broadcasting for prefix or suffix searches.

Another challenge in building distributed inverted indexes comes from the emerging trend of enabling user-defined tags in storage systems (e.g., SoMeta [43] and TagIt [40]). As compared to traditional file systems where only a fixed number of attributes are available in the metadata, such practice can introduce unlimited number of attributes and user-defined tags. As the number of keywords to be indexed increases, the keyword distribution lead (or ended) by different prefixes (or suffixes) and the keyword popularity can be imbalanced. Such imbalance would cause query contention and poor resource utilization in a distributed system, which may discount the efficiency of affix-based keyword search. Thus, it is necessary to consider load balancing issue when creating such distributed trie-based inverted index.

Given the challenges of distributed affix-based keyword search, in this paper, we propose a distributed trie-based inverted index for efficient affix-based keyword search. Our technique can be easily applied on HPC metadata management systems that facilitate parallel client-server communication model. Inspired by Adaptive Radix Tree (ART) [21], we name our proposed approach as DART - Distributed Adaptive Radix Tree. We have designed DART to provide simplicity and efficiency for distributed affix-based keyword search.

The major contributions of this paper are:

- We designed the DART partition tree and its initializing process in which the height of the DART partition tree and the number of DART leaf nodes is determined by the number of servers, which ensures the scalability of DART.
- We composed the index creation process with a series of node selection procedures and replication mechanism on top of DART partition tree so that the query workload on different index node can be balanced.
- We devised the query routing procedure on DART to achieve efficient query response. Our evaluation demonstrates that DART outperforms DHT by up to 55× in executing prefix and suffix keyword searches and performs similar to DHT for exact and infix searches.

The rest of the paper is organized as follows: In Section 2, we review existing solutions and challenges on distributed inverted index. Then, we analyze the requirements of an efficient distributed inverted index in Section 3. We propose our Distributed Adaptive Radix Tree (DART) method in Section 4 and evaluate DART in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

There are two paradigms of distributed index construction: document-partitioned approach and term-partitioned approach [24, 25].

The document-partitioned approach is considered to be "local index organization", where each server stores a subset of all documents or data items while maintaining index for the documents or data items on that server locally. In this case, many data structures can be used for local indexing, such as B+ Tree [8], Patricia tree [19], ART [21], and many software libraries and tools including SQLite [42], ElasticSearch [11], and Apache Lucene [13] can also be used for such purpose. In addition, each server leverages locality between the index and the actual documents or data items being indexed (TagIt [40] is a typical example of such approach). However, a keyword query must be broadcasted to all servers in order to retrieve all the related documents or data items, and requires all servers to participate in the process of serving a query. Such query broadcasting can lead to intense network traffic and exhausts limited computing resources when query frequency rises. In addition, load balance of inverted index on different servers highly relies on the partitioning strategy of documents or data items. In this case, it can be difficult to manage the distribution of indexed keywords on each server, and hence may result in imbalanced workload. **In this study, we do not consider document-partitioned approach nor compare our work with any document-partitioned approach like TagIt[40]** because the load imbalance of this approach prevents itself from supporting efficient affix-based keyword search.

The term-partitioned approach takes each indexed keyword as the cue for index partitioning. The typical case of such approach designed for HPC systems is the use of DHT, which can be found in many recent distributed metadata and key value store solutions, such as SoMeta [43], IndexFS [35], FusionFS [49], and DeltaFS [50]. However, the application of DHT still remains inappropriate for affix-based keyword search in two-fold.

First, when applying hashing algorithm against each keyword, it may result in a well balanced keyword distribution with careful selection of hash function. However, this DHT approach only supports efficient exact keyword search. When it comes to prefix search or suffix search, the query still has to be sent to all the servers and hence the search performance is far from optimal. In this paper, we call this approach as "**full string hashing**", in the sense that the hashing algorithm is applied on the entire keyword.

Second, to support efficient prefix search, a common approach to use DHT is the *n-gram* technique [14, 15, 37, 46]. This approach iterates over all prefixes of each keyword (including the keyword itself), and creates index for each individual prefix of the keyword. However, since each keyword may have multiple prefixes, and such approach leads to significant space consumption for storing the index, which can be many times larger than only indexing the keyword itself. To avoid this drawback, it is preferable to adopt 1-gram approach rather than the n-gram approach, by which only the first character of a keyword is considered in the partitioning, and we call this approach "**initial hashing**". However, due to the natural imbalanced keyword distribution led by different leading characters, such approach may fail to maintain balanced keyword distribution and hence unable to achieve decent load balance.

Building distributed index may not be rewarding for all affix-based keyword search. For infix search, since the presence and the position of an infix may vary from keyword to keyword, it is common to go through all index records to achieve infix search, which can also lead to query broadcasting among all servers.

Other than DHT, several research efforts have explored the distributed keyword search problem [17], [39], [34], [36], [18], [6], [3], [4]. However, all of them attempt to address the keyword search problem in peer-to-peer systems, where performance requirements

are more relaxed than in HPC systems. Thus, it is necessary to develop a distributed indexing methodology for efficient affix-based keyword search on HPC system.

3 KEY REQUIREMENTS

We summarize and formally define four requirements of distributed keyword search in HPC systems: functionality, efficiency, load balance, and scalability, in building efficient distributed inverted index. We discuss them in details below.

3.1 Functionality

Affix-based keyword search is fundamental in data discovery in many application scenarios. Typical examples can be seen in search engines, file systems, database systems, etc. In such systems, the total amount of data can be tremendous and hence the amount of information which needs to be indexed in metadata is also huge. Hence, it can be much more often to perform affix-based search in order to find multiple data items instead of a particular dataset using an exact search. While there could be other forms of the affixes, in this paper, we focus on four basic types of affix-based keyword search: 1) **exact search**, 2) **prefix search**, 3) **suffix search**, and 4) **infix search**. We consider them to be essential for the metadata search and distributed key value store searches on HPC systems.

An exact search finds an indexed key that matches with the given keyword in its entirety. A typical example of this can be "`name=experiment001.dat`". A prefix search matches the first few characters in the key of an index record, i.e., "`name=experiment*`". Likewise, a suffix search matches the last few characters in the key of an index record, such as "`name=*.dat`", and an infix search finds the index record where the indexed key contains the characters in the middle of the query string, like "`name=*001*`".

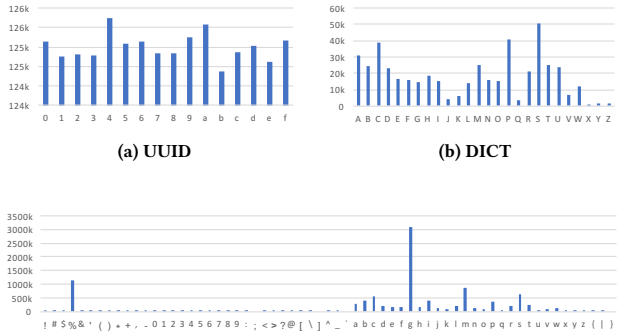
It is noteworthy that an exact search can be transformed into a prefix search since the longest prefix of a string is itself. Another note is that a suffix search can be accomplished by indexing the inverse of each keyword and search through such inverted index. In addition, an infix search can be performed by traversing through each keyword in the index and collecting the result from the index records where the indexed keyword contains the given infix. Thus, it is possible to achieve four different types of affix-based keyword search by designing one indexing methodology.

3.2 Efficiency

The main goal of building distributed inverted index is to accelerate affix-based search. Thus, the efficiency of affix-based keyword search is crucial. For HPC environment which prefers on-core computation but still holds distributed nature, the communication overhead has a major impact on the overall efficiency of affix-based keyword search when the index is created across different cores. Thus, we focus on minimizing the communication cost in affix-based keyword search, especially prefix search, suffix search, exact search. However, for infix search, since we cannot predict where the given infix may appear in a keyword, our approach should not degrade the infix search performance as compared to the existing DHT approaches. Also, the procedure of index creation and index deletion should also be efficient.

3.3 Load Balance

For a distributed index, imbalanced workload distribution may lead to overloaded nodes, which in turn damage the overall efficiency. Thus, it is important to ensure a balanced index partition on each server. However, in practice, keywords that need to be indexed often follow an uneven distribution, which may lead to imbalanced workload.



(a) UUID (b) DICT

(c) WIKI

Figure 1: Uneven keyword distribution under different leading characters in different datasets. The leading letters are listed along the horizontal axis, and the number of keywords led by different leading letters are shown vertically.

To demonstrate the presence of uneven keyword distribution, we have done a preliminary study with three different datasets. We provide the histogram of each dataset to demonstrate the number of keywords led by different leading characters (see Figure 1). The first dataset is a set of 2 million UUIDs that are randomly generated by Linux *libuuid*[5]. We consider the **UUID** dataset represents unique identifiers of some scientific datasets, for example, the identifier of datasets in the HDF5 file of BOSS data[38]. The second dataset is a list of 479k English words [28], labeled as DICT in Figure 1. We consider this dataset represents a more generalized keyword set. The last dataset contains all the keywords extracted from the Wikipedia access log during September, 2007 [45], along with the number of requests on each keyword (labeled as WIKI). We consider such dataset to be a representative of real-life keyword search scenario, and the skewness in the number of requests on each keyword also gives a close approximation on the request distribution pattern of different keywords in real practice. From Figure 1, we can observe that the distribution of keywords led by different leading character is imbalanced, and similar observation is present in all three datasets. Particularly, Figure 1(a) only shows the keyword distribution of UUIDs that are randomly generated in one batch. Although the keyword distribution of UUIDs may vary from batch to batch, the randomness of the UUIDs can also sufficiently support the validity of our claim that uneven keyword distribution is ubiquitous.

The imbalance of workload does not only come from the uneven index distribution, but also comes from the skewness of requests on different keywords. We plot the number of requests of different keywords in dataset WIKI, and as it can be observed from Figure

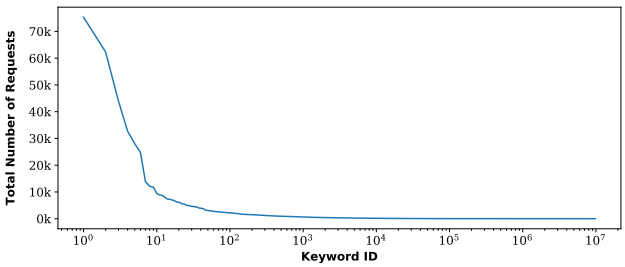


Figure 2: Highly skewed requests for different keywords.

2, the number of requests on keywords during September, 2007 follows a skewed distribution. Such skewness also leads to uneven workloads of all servers.

3.4 Scalability

For a distributed inverted index, especially when used on HPC systems, scalability is an important requirement. User applications are often running on hundreds to thousands of compute nodes. The strategies and optimizations introduced and evaluated at smaller scale need to be valid at larger scales as well. Thus, it is crucial for an indexing and keyword-based querying to provide efficient performance at scale. The querying strategy should also remain efficient regardless of the dataset size and data type.

4 DISTRIBUTED ADAPTIVE RADIX TREE

To address the challenges of distributed affix-based keyword search, we now present the design of DART - the Distributed Adaptive Radix Tree.

4.1 Terminology

We first define the terminology and notations used throughout this paper.

We denote the **character set** which DART has to work on as $\mathcal{A}$, thus, the number of all characters in $\mathcal{A}$ can be denoted as $k = |\mathcal{A}|$, which we call **the size of the character set**. For the sequences of characters from $\mathcal{A}$, we call them **terms of** $\mathcal{A}$, or simply **terms**. A term $T = (t_1 t_2 ... t_l)$ of length l comprises $l = |T|$ characters from $\mathcal{A}$. We define a **prefix** of terms T to be the first n characters of T, denoted as $p(n, T)$, where $n \leq l$. A prefix $p(n, T)$ can also be a prefix of another prefix $p(m, T)$, as long as $n \leq m$ and $m \leq l$. If $n < m \leq l$, we call $p(n, T)$ a **shorter prefix** of T, as compared to the **longer prefix** $p(m, T)$. If $n = m \leq l$, we say both prefix $p(n, T)$ and $p(m, T)$ are **identical**. For the term for which we want to build index, we call it an **indexing term**. For the term of which an index is built on, we call it an **indexed term**. For the term a query is looking for, we call it a **search term**. Also, in the discussion of distributed index, we use "**term**" and "**keyword**" interchangeably. Note that, every character in $\mathcal{A}$ can be encoded into a string of one or more binary digits, hence all characters in $\mathcal{A}$ can be compared and sorted.

4.2 Overview of DART

DART is designed to meet the goals of functionality, efficiency, load balance, and scalability discussed above. DART is composed of two major components:

- **DART partition tree**, which virtually exists on all clients and divides the entire search space into several partitions with all its leaf nodes. Note that when we project the entire search space onto a polar coordinate system, all the nodes of DART partition tree actually form a ring, which we call **DART hash ring**.
- **The server-side local inverted index** on each physical node. We do not limit the actual data structure on the server side for local inverted index. In this paper, we select a trie-based data structure - adaptive radix tree [21] - as our prototype of server-side local inverted index.

In the rest of this section, we discuss each component and explain how they are constructed and utilized in detail, as well as the description of different operations on DART.

4.3 DART Initialization

During the initialization of DART, we follow the trie-based approach to initialize the DART partition tree, which divides the entire search space into several partitions. The root node of the DART partition tree is empty and it does not count towards the height of the DART partition tree. For a partition tree of height d, at each level $i \in \{1, 2, ..., d\}$ of the partition tree, each node branches out to its succeeding level (namely level $i + 1$) by iterating each character in the character set $\mathcal{A}$ in order. Thus, at level d, the partition tree will have $N_{leaf} = (k)^d$ leaf nodes, where $k = |\mathcal{A}|$, which we call ***the radix of DART***.

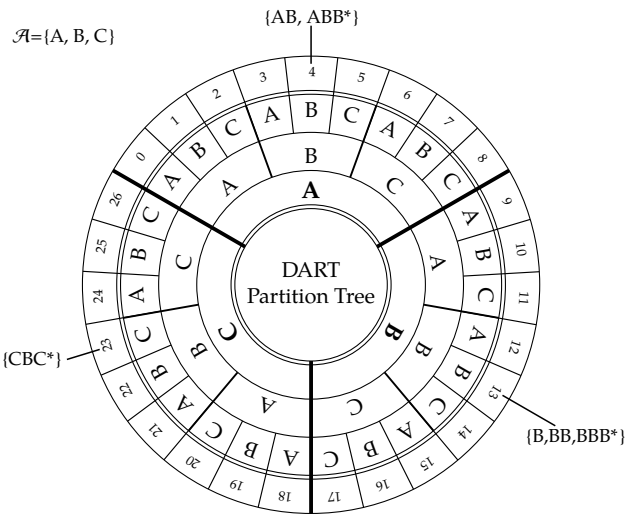


Figure 3: DART partition tree of height $d = 3$ on the basis of character set $\mathcal{A}$ ={A, B, C}. The numbers on the outermost ring are DART virtual leaf node IDs.

Figure 3 shows an example of a DART partition tree of height $d = 3$ over an character set $\mathcal{A} = \{A, B, C\}$ such that $k = 3$, the total number of leaf nodes in this partition tree is $N_{leaf} = 3^3$.

Each leaf node is regarded as a virtual node in the entire search space. And such virtual node can be mapped to an actual physical node with a modulus function $I_{physical} = I_{leaf}\%M$, where $I_{physical}$ denotes the ID of the physical node and I_{leaf} denotes the sequence number of a virtual leaf node on DART partition tree.

To ensure that the entire search space, namely M physical compute nodes, can be fully covered by all the leaf nodes of the partition tree, the number of virtual leaf nodes N_{leaf} should be larger than the number of physical nodes M. To achieve this, we use the relationship between the height of the tree d, the number of all leaf nodes in the partition tree N_{leaf}, and the radix of the tree $k = |\mathcal{A}|$ to calculate a minimum value of d such that $N_{leaf} > M$, namely $(k)^d > M$. Thus, we have:

$$d = \lceil log_k M \rceil + 1 \tag{1}$$

By initializing the height of DART partition tree using the above equation, we can guarantee that the number of leaf nodes is larger than the number of all physical nodes. And such computation is lightweight with a computational complexity of $O(1)$. In addition, as the DART partition tree only virtually exists, **this computation does neither require much memory at the client side nor require any synchronization among all clients**, which is ideal for a fast initialization in a distributed system.

4.3.1 *Root Regions and Subregions*.

Having DART partition tree initialized, we now define some important terminologies about DART. Recall what we have introduced in 4.2 that all leaf nodes of the DART partition tree forms the DART hash ring. It is evenly divided by the radix of DART into $k = |\mathcal{A}|$ root regions. Because the total number of leaf nodes in the DART partition tree is a multiple of the radix k, the number of virtual leaf nodes in each root region must be equal to $\mathcal{D}_{root} = N_{leaf}/k$, and $\mathcal{D}_{root}$ is the number of virtual nodes in each root region, which we call **root region distance**. For example, in Figure 3, the DART hash ring virtually consists of 27 leaf nodes in the sample DART partition tree, and these virtual leaf nodes are evenly divided into 3 root regions, and each root region contains 9 virtual nodes, namely, the root region distance is 9.

Also, in DART partition tree, each root region is further divided into k subregions, and each subregion contains $\mathcal{D}_{sub} = N_{leaf}/k^2$ virtual nodes, which we call **subregion distance**. For example, root region A in Figure 3 is further divided into 3 subregions, and the distance of each subregion is 3.

4.4 Index Creation

In this section, we introduce the index creation process. In order to know where to create the index for a given keyword, we must select a virtual node. In DART, load balance issue is one of our key concerns when performing virtual node selection.

As discussed in Section 3.3, the workload of a physical node depends on two major factors. On the one hand, the uneven keyword distribution over different physical nodes may lead to uneven workload. Thus, in DART, one way of achieving load balance is to make the number of indexed terms on each leaf node roughly equal. For such purpose, we first select both base virtual node and alternative virtual node. Afterwards, we take one of them which has fewer indexed keywords to be the eventual virtual node. Finally, we create index on the eventual virtual node.

On the other hand, in Section 3.3, we also mentioned that the workload of a physical node also depends on the popularity of the keywords on it. In order to cope with such issue, in DART, we also perform index replication to create a number of replicas for each indexed keyword.

Note that we perform index creation on each keyword as well as its inverse.

4.4.1 *Base Virtual Node Selection*.

We first need to select a base virtual node. For a given term $T = (t_1 t_2 ... t_l)$ of $\mathcal{A}$, let i_{t_n} denote the index of character t_n in character set $\mathcal{A}$. For example, in term "CAB" on character set $\mathcal{A} = \{A, B, C\}$, t_1="C", and hence $i_{t_1} = i_C = 2$. Likewise, we have $i_{t_2} = i_A = 0$ and $i_{t_3} = i_B = 1$.

We use the DART partition tree of height d to select the base virtual node. When the length of term T is greater than or equals to d, namely $l \geq d$, we calculate the base virtual node location I_v by the following equation:

$$I_v = \sum_{n=1}^{d} i_{t_n} \times k^{d-n} \tag{2}$$

When $l < d$, we first pad the term with its ending character until its length reaches to d, and then perform the above calculation to determine the virtual node to start from.

Note that the base virtual node of term $T = (t_1 t_2 ... t_l)$ must reside in the root region labeled with character t_1. For example, given the partition tree in Figure 3, the set of keywords {B, BB, BBB*} goes to virtual node 13, which resides in root region B. Similarly, the set of keywords {AB, ABB*} goes to virtual node 4 in root region A, and the set of keywords {CBC*} goes to virtual node 23 in root region C.

4.4.2 *Alternative Virtual Node Selection*.

Now we introduce how we select alternative virtual node. First, according to the first character of a keyword, we select an alternative root region for the keyword by performing the following calculation:

$$I_{alter\_region\_start} = [(i_{t_1} + \lceil k/2 \rceil)\%N_{leaf}] \times \mathcal{D}_{root} \tag{3}$$

This calculation returns the offset of the starting virtual node in the alternative root region. This is to make sure the alternative root region of term T is different than its base root region. For example, as shown in Table 1, the alternative root region starts from virtual node 18 in region C for the set of keywords {AB, ABB*}. Similarly, 0 in region A for {B, BB, BBB*} and 9 in region B for {CBC*}.

Afterwards, we further calculate the offset of alternative subregion, which we define as the **major offset**, denoted as w_1. For a DART partition tree of height d, given a keyword $T = (t_1 t_2 ... t_l)$, we perform such calculation following the equation below:

$$w_1 = (i_{t_{d-1}} + i_{t_d} + i_{t_{d+1}})\%k \tag{4}$$

where t_d is the character on the leaf level of the partition tree, which we call **on-leaf character**. For t_{d-1}, we call it the **pre-leaf character** and t_{d+1} the **post-leaf character**. However, when $d = 1$, there is no pre-leaf character, and we let $i_{t_{d-1}} = 0$ in this case. When $l = d$, there is no post-leaf character, so we just let $i_{t_{d+1}} = i_{t_l}$ and let $i_{t_d} = i_{t_l}$ in this case, where t_l is the very last character

in term $T = (t_1 \ldots t_l)$. When $l < d$, we pad term T with its ending character until its length reaches to d, and we follow the same procedure as when $l = d$.

Keyword	Base Root Region	Base Virtual Node	Alternative Root Region	Alternative Virtual Node
AB	A (begins at 0)	4	C (begins at 18)	21
ABB	A (begins at 0)	4	C (begins at 18)	21
ABBC	A (begins at 0)	4	C (begins at 18)	25
B	B (begins at 9)	13	A (begins at 0)	3
BB	B (begins at 9)	13	A (begins at 0)	3
BBBA	B (begins at 9)	13	A (begins at 0)	3
CBCBA	C (begins at 18)	23	B (begins at 9)	10

Table 1: Virtual node selection on concrete examples of set of keywords {AB, ABB*}, {B, BB, BBB*} and {CBC*} on character set $\mathcal{A}$ = {A,B,C}

After getting the offset for selecting an alternative subregion, we finally calculate the offset for selecting a virtual node in the subregion, which we call **minor offset**, denoted as w_2. We perform the following calculation:

$$w_2 = |(i_{t_{d+1}} - i_{t_d} - i_{t_{d-1}})| \% k \quad (5)$$

Now, with the aforementioned base virtual node ID I_v, the starting virtual node ID of the alternative root region $I_{alter\_region\_start}$, the root region distance $\mathcal{D}_{root}$ and subregion distance $\mathcal{D}_{sub}$, we can finally calculate alternative virtual node ID $I_{v'}$:

$$I_{v'} = I_{alter\_region\_start} + (I_v + w_1 \times \mathcal{D}_{sub} + w_2) \% \mathcal{D}_{root} \quad (6)$$

In Table 1, we also show the alternative root region and alternative virtual node of each concrete keyword example. It can be seen from the table that, for any keyword, the alternative root region is different from the base root region. And inside either of the root region, the offset of the virtual node is also different. This actually ensures sufficient randomness when it comes to eventual node selection.

4.4.3 ***Eventual Virtual Node Selection.*** After getting the base virtual node I_v and the alternative virtual node I_v' for term T, the client will retrieve the number of indexed keywords on corresponding physical nodes of both virtual nodes. Since such operation only take few microseconds to finish and can be performed in parallel, it does not introduce much communication overhead. Afterwards, the client will pick the virtual node with less indexed keywords as the eventual virtual node E_v. Then, following the eventual virtual node, the index for term T will be created on the corresponding physical node. By following such node selection procedure, DART is expected to maintain a balanced keyword distribution among all physical nodes. For a randomized load balancing scheme[29], the choice of two nodes are randomly made. But in DART, the node selection is performed by mathematical operations over known parameters, so the ID of the physical node can be easily calculated, which is helpful for achieving efficient deterministic query routing during the query responding process. But at the same time, DART makes an effort in maintaining sufficient randomness in terms of the distance between base virtual node and alternative node.

4.4.4 ***Index Replication.*** As discussed in Section 3, another cause for an imbalanced workload is the skewness in the number of requests on different terms. Since the popularity of a term is uncertain when the index is created, we can only alleviate the excessive requests by having more replicas of the index. In DART, we define a replication factor r, by which the index of any term will be replicated for r times, and these r replicas are created by selecting r virtual nodes along the DART hash ring using the following calculation:

$$R_i = E_v + \frac{N_{leaf}}{k} \times i \quad (7)$$

where i denotes the ID of replica and $i \in [1, r]$ and E_v is the eventual virtual node selected from the based virtual node and the alternative virtual node. By doing so, each query request will be sent to one replica at a time in a round-robin fashion, thus the r replicas of each indexed keyword will share the workload caused by the query requests. Therefore, the excessive requests against some popular keywords can be alleviated. Since such replication is done in an asynchronous way, the index creation latency is not bound to the creation of replicas.

4.5 Query Response

As discussed in Section 3.1, DART supports four types of affix-based keyword search, including exact keyword search, prefix search, suffix search and infix search, and both exact keyword search and suffix search can be transformed into the problem of solving prefix search. Thus, in this section we only discuss how DART fulfills prefix search and infix search.

4.5.1 ***Prefix Search using DART.***
For a given prefix p of length $n = |p|$, there can be multiple terms T that match with p. We first discuss the case when the length of prefix p is greater than the height of the DART partition tree d, i.e., $n > d$. In this case, the client can follow the same procedure as described in Sections 4.4.1 and 4.4.2 to select the base virtual node and the alternative virtual node. Subsequently, the query will be sent to the corresponding physical nodes. The client takes the result from the node whichever returns a non-empty result. If both physical nodes return empty result, we consider the query does not hit any index record, meaning, there is no relevant data matching the given query. The communication cost of such operation remains as constant of $O(1)$ since only two physical nodes are contacted.

When the length of given prefix is less than or equal to the height of DART partition tree, i.e., when $n \leq d$, we perform prefix search only according to the leading character of the search term. For example, if the search term is "AB*" or "ABB*", we actually perform prefix search "A*" instead. And since all the keywords sharing the same leading character must reside on the same base root region or the same alternative root region, we send this query to all virtual nodes in both root regions to collect the result. For character set $\mathcal{A}$ of size k, 2/k of the virtual nodes will be queried. In real practice, the size of the character set can be much larger. In the case of ASCII, there are 128 characters in the character set. Thus, only 2/128 of the virtual nodes will be selected for responding the prefix query. This is a huge deduction in the communication cost as compared to query broadcasting on all physical nodes.

4.5.2 ***Infix Search with DART.*** Since the position of a given infix is uncertain in a keyword, to guarantee that no relevant keyword will be omitted in the query responding process, we still have to perform query broadcasting to all physical nodes. To the best of our knowledge, there is no indexing technique that can avoid full scan on the indexed keywords when it comes to infix query. Also, it is difficult to predict where a given infix will appear in a keyword. Thus, in DART, when an infix query is requested, the client sends request to all physical nodes, and on each physical node, a full scan on the local index is performed to collect all matching results.

4.5.3 ***Alleviating Excessive Requests.*** In DART, we maintain a request counter at each client, and the request counter will be increased by one each time when a request is sent. When there are r replicas, each time the client will do the following calculation to determine the ID of the replica where the request should be sent to:

$$R_v = (I_v + C_{request}) \% r \quad (8)$$

where R_v represents the ID of the replica, I_v represents the ID of the initial virtual node, and $C_{request}$ is the value of request counter. Such mechanism enables the round-robin access pattern among all replicas, and helps to alleviate the excessive requests against some extremely popular keywords.

4.6 Index Update and Index Deletion

In DART, index update is done by deleting the original index record and rebuild a new index record. To delete an index, we still perform the same calculation described in Sections 4.4.1 and 4.4.2 to select the base virtual node and alternative virtual node. And then we send the index deletion request to both virtual nodes and hence their corresponding physical nodes. Whichever the physical node with the specified index record will actually perform the index deletion operation while the other ignores the operation once it confirms there is no specified index record. The communication cost of this operation is still $O(1)$, which is constant.

4.7 Complexity Analysis on DART

All operations in DART, including DART initialization, index creation, query response, index update and index deletion, all involve one or more of the 8 equations from (1) to (8). It is noteworthy that all calculations of these equations are arithmetical calculations and can be performed very quickly at the client side with $O(1)$ time.

Also, for conducting these calculations, only a few of scalar values are required to be maintained in the memory, including the number of physical nodes, the size of the character set, the index of each character in the character set, and some intermediate results like N_{leaf}, $I_{alter\_region\_start}$, I_v, w_1, w_2, $\mathcal{D}_{root}$ and $\mathcal{D}_{sub}$. And since DART partition tree only virtually exists, there is no need to maintain DART partition tree in memory. So the memory footprint of DART is also very low.

As for communication cost, the selection of eventual virtual node only involves access of two physical nodes, in addition to the r requests for actually creating the index replicas, the communication cost is $O(r)$, which is constant. When multiple replicas are created in parallel, the index creation overhead can be considered to be even lower. Also, since each client actually follow the same procedures to conduct DART operations independently, there is no need to perform any synchronization between clients, which saves huge amount communications in the network as well.

Operations	Computation Complexity of Locating Procedure (Worst Case)	Communication Complexity (Worst Case)
Insertion	$O(1)$	$O(1)$
Deletion	$O(1)$	$O(1)$
Exact Search	$O(1)$	$O(1)$
Prefix Search	$O(1)$	$O(\frac{M}{k})$
Suffix Search	$O(1)$	$O(\frac{M}{k})$
Infix Search	$O(1)$	$O(M)$

Table 2: Complexity of Different DART Operations

We provide worst case complexity of different DART operations in Table 2.

5 EVALUATION

5.1 Experimental Setup

We have evaluated DART on Cori, a Cray XC40 supercomputing system, located at the National Energy Research Scientific Computing Center (NERSC). It has 2,388 Haswell compute nodes, with each node featuring two 16-core Intel® Xeon™ processors E5-2698 v3 ("Haswell") at 2.3 GHz and 128GB memory. The interconnect of Cori is Cray Aries with Dragonfly topology and 5.625 TB/s global bandwidth.

We implemented DART on the same software platform that has been used in SoMeta [43] and PDC [44], which is built upon MPI and Mercury [41] RPC framework. The software platform already contains DHT implementation which already facilitated the experiments in both aforementioned research works. To be coherent with the trie-based design that DART is following, we use ART [21] as the server-side indexing data structure for DART. We use hash table as the server-side local data structure for two DHT-based approaches, "Full String Hashing" and "Initial Hashing", since such combination is prevalently used in many existing studies, such as FusionFS [49], DeltaFS [50], SoMeta [43], etc. For all experiments, we set up an equal number of client instances and server instances. Each client instance can access all server instances.

We choose three datasets, **UUID**, **DICT** [28], and **WIKI** [45] (discussed in Section 3.3), for our evaluation. We use dataset UUID for our performance test since it can be generated by Linux *libuuid* [5] in memory, which avoids the overhead of disk read operations. For load balance test, we use all three datasets since we would like to observe whether DART is capable of balancing the keyword distribution among various sets of keywords. Since every keyword in all three datasets consists of only basic ASCII characters, so we set up the radix of DART as 128.

5.2 Efficiency

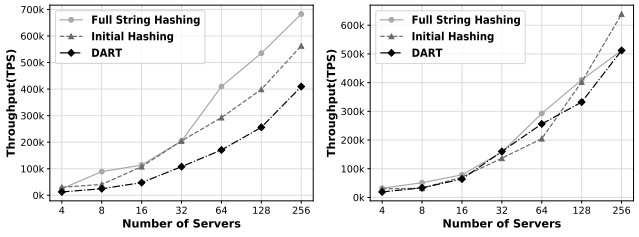
We compare the efficiency of DART against full string hashing and initial hashing in terms of the throughput of various operations at different cluster scales, ranging from 4 nodes to 256 nodes. Two million UUIDs were inserted each time to all servers to perform index creation. Afterwards, we took the 4-letter prefix, 4-letter

suffix and 4-letter infix of each UUID to conduct prefix search, suffix search and infix search respectively. Finally, we performed deletion operation for each UUID. We measure the throughput in thousands of transactions per second (TPS).

Please note that DART works for any prefix/suffix/infix length (denoted as n). The reason we choose $n = 4$ is that the DART partition tree height ranges from 2 to 3 when the number of servers varies from 4 to 256. In our evaluation, we emulated a predictably more common use case where some of the characters in the query keyword will be addressed by the DART partition tree and the remaining characters would be addressed by a local index on a given server.

5.2.1 *Index Creation and Deletion.*

In the index creation process, DART needs to perform load detection operations for determining the workload between base virtual node and alternative virtual node. Then, DART needs to send index creation request to the selected server, which introduces communication overhead. In contrast, both full string hashing and initial hashing are conventional hashing algorithms and do not include any sophisticated operations. For delete operation, DART only needs to send delete requests to both base virtual node and alternative virtual node simultaneously, which does not introduce much overhead.



(a) Index Creation Throughput (b) Index Deletion Throughput

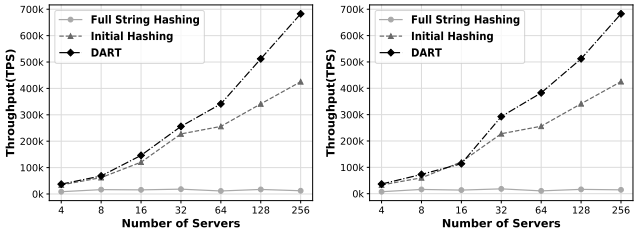
Figure 4: Throughput of Index Creation and Index Deletion

As shown in Figure 4(a), for different cluster scales, the insert throughput of full string hashing and initial hashing are slightly higher than that of DART. Such a trend can be observed at different cluster scales. Considering that DART accomplishes much more in load balance, such performance gap between DART and the other two DHT cases is reasonable and acceptable. As for the throughput of index deletion, as it can be seen from Figure 4(b), DART maintains comparable performance with these two DHT-base approaches.

5.2.2 *Prefix Search and Suffix Search.*

DART is designed to support efficient affix-based search. In terms of prefix search and suffix search, the advantage of DART is more obvious.

As shown in Figure 5(a) and Figure 5(b), both initial hashing and DART achieve better throughput than full string hashing. This is because, in full string hashing, the query broadcasting is required when performing prefix search and suffix search, while in initial hashing there is no need to perform query broadcasting and in DART the query broadcasting is eliminated as well in most cases. Particularly, as compared to full string hashing, at different scales the throughput of prefix search and suffix search on DART is 4×∼55× higher.

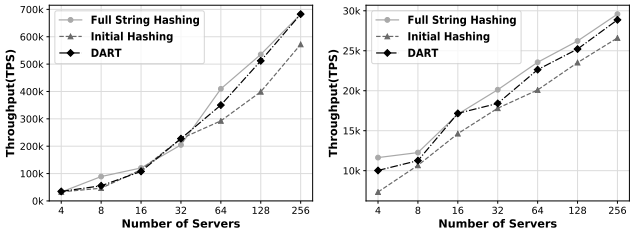


(a) Prefix Search Throughput (b) Suffix Search Throughput

Figure 5: Throughput of Prefix Search and Suffix Search

It is noteworthy that DART even performs better than initial hashing. This is because DART achieves a more balanced keyword distribution which help with avoiding overloaded servers. For example, in the case of 256 servers, the standard deviation of the DART keyword distribution is 2877.208 while the same index in initial hashing is 15128.953, which means, with initial hashing, there can be more overloaded servers and each of them will damage the overall throughput.

5.2.3 *Exact Search and Infix Search.*



(a) Exact Search Throughput (b) Infix Search Throughput

Figure 6: Throughput of Exact Search and Infix Search

For exact keyword search, full string hashing is expected to deliver the best performance, since each keyword can be efficiently found via the hashing function. The actual results, plotted in Figure 6(a), show that the exact search performance of DART and full string hashing are almost identical, while the throughput of initial hashing slightly falls behind the others, when the system size scales to 64 servers and beyond. This is because the keyword distribution of initial hashing is imbalanced, and the server hosting the most frequent keywords becomes the bottleneck. We show that the keyword distribution of initial hashing later in Section 5.4 and further justify our explanation on this.

In the case of infix search, each query has to go through all indexed keywords since the position of the infix in a keyword is uncertain. As shown in Figure 6(b), the throughput of infix search with three different indexing methods are very close. These results show that DART achieves comparable efficiency as DHT for infix searches.

5.2.4 *Latency of DART Operations.*

We show the latency of different DART operations on different number of physical nodes in Figure 7 with box plot. As shown in the figure, most of the insertion and deletion operations can be finished within 400 microseconds (0.4 milliseconds), and on average, the insertion operation latency and the deletion operation latency are around 300 microseconds and 200 microseconds respectively.

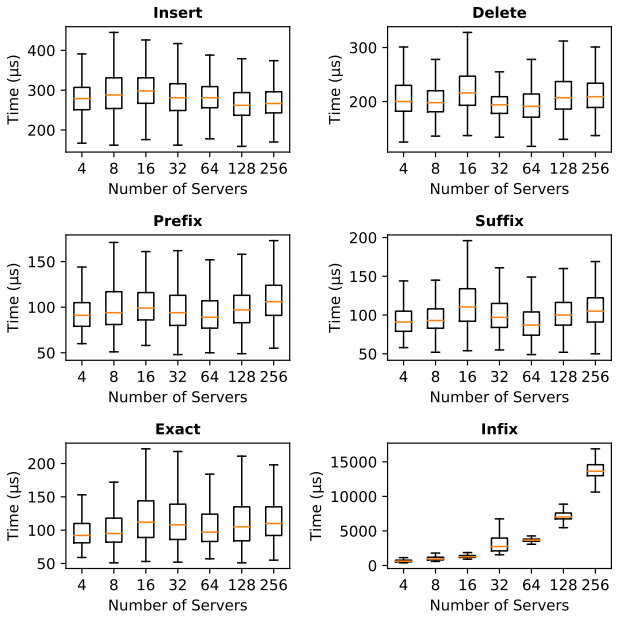


Figure 7: Latency of DART operations

The average latency of prefix search, suffix search and exact search is around 100 microseconds, with no search latency exceeding 250 microseconds. As for infix search, as the number of server grows exponentially, the query latency increases in the same pace since an increasing amount of communications was performed due to query broadcasting. At maximum, the infix query latency on 256 physical nodes is around 17000 microseconds, namely, 17 milliseconds. When consider the natural complexity of infix search, we consider such latency to be acceptable.

Note that there is some fluctuation in terms of latency over different number of servers. We took multiple times of our evaluation, and all of them show different fluctuating pattern. We believe this may relate to the fluctuating network condition at the time when we conducted our evaluation. We did not try to even out such dynamics by taking results from multiple runs, since we believe such dynamics actually makes our result valid in reflecting the actual variations in query latency. So we only demonstrate the latency result from one single run, and this shows the actual performance limit that our DART can reach to in the real case, which turns out to be decent.

5.3 Load Balance

In this series of tests, we inserted three different datasets - UUID, DICT, and WIKI - onto 16 servers via full string hashing, initial hashing, and DART. The hashing algorithm we used for both DHT cases are `djb2` hash, which is known to achieve load balance well.

For each keyword, the full string hashing takes all the characters in a keyword as a whole, computes the hash value accordingly and use the modulus hashing function to uniformly select the server where the keyword should be indexed. Thus, the full string hashing is able to generate balanced keyword distribution. However, for each keyword, initial hashing will only take the first character into account, which leads to aggregation of keywords sharing common leading character. As we analyzed in Section 2, the distribution of keywords led by different characters is imbalanced, thus, the initial hashing is expected to generate imbalanced keyword distribution. DART is designed to alleviate imbalanced keyword distribution, and we should see such effect in the result of DART.

In order to compare the dispersion of keyword distribution on different dataset in an intuitive way, we introduce **coefficient of variation** (abbv. CV) as a metric. The coefficient of variation C_v is defined as the ratio of the standard deviation σ to the mean μ, i.e. $C_v = \frac{\sigma}{\mu}$ [12]. It is a standardized measure of dispersion of probability distribution. A smaller CV means smaller deviation to the average value and hence a more balanced distribution.

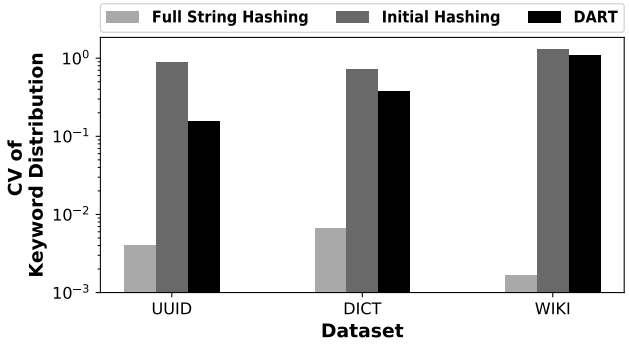


Figure 8: CV of keyword distribution by 3 different hashing policies on 3 different datasets over 16 servers.

As shown in Figure 8, on 16 servers, the CV value by full string hashing algorithm is always the smallest on our three different datasets - UUID, DICT and WIKI. Such result shows that the full string hashing algorithm always leads to a perfectly balanced keyword distribution regardless the variance of datasets. Similarly, the initial hashing policy always leads to the highest CV value over 3 different datasets, which means the keyword distribution rooted from this algorithm is the most imbalanced. However, our DART algorithm always leads to a smaller CV value than the initial hashing algorithm on different datasets, which means it is able to generate a more balance keyword distribution than what initial hashing algorithm can do, even though it does not achieve the balanced keyword distribution as perfectly as the full string hashing does. Overall, such result shows the universality of DART in maintaining balanced keyword distribution over different datasets, which meets our expectation.

In DART, we also design the replication strategy to replicate the index record of a certain keyword and hence reduce the excessive

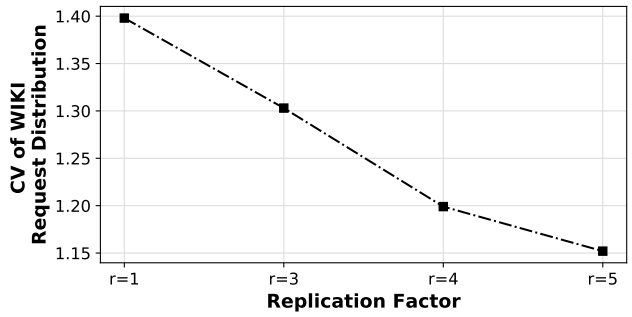


Figure 9: Coefficient of variation on the request distribution while replaying WIKI requests with DART under different replication factors.

workload of some extremely popular keywords. We used the WIKI dataset to test the effectiveness. Recall that in the WIKI dataset, each keyword is recorded along with the number of requests that has been queried during September, 2007. We have shown the request distribution on different WIKI keywords in Section 2, Figure 2. It can be seen that the request distribution is highly skewed. We replayed these 48M requests recorded in the WIKI dataset on 16 servers, after the index was created for all keywords.

As reported in Figure 9, as the replication factor increases, the CV value of the request distribution decreases, which means the index replication mechanism in DART is able to mitigate skewness of the request distribution as well, which can be another performance perk when it comes to real use cases.

As discussed in Section 2, the UUID dataset can be considered as a representative of digital identifiers of scientific dataset, while DICT can be viewed as a representative of a more general keyword set in natural languages. We also consider the WIKI dataset represents a realistic workload. These load balance tests and results demonstrate that the DART algorithm is not sensitive to the type of keyword set, which makes it a suitable choice for various scenarios and capable of mitigating imbalanced keyword distribution on servers, not to mention HPC applications. We further report the load balancing capability at different scales in the next section.

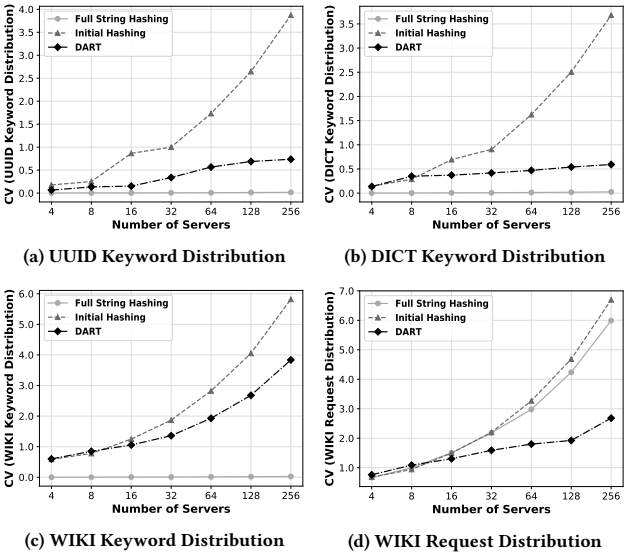
5.4 Scalability

5.4.1 *Performance at scale.*

As discussed in Section 5.2, DART is able to maintain its efficiency at scale for different operations, which already demonstrates the scalability of DART in terms of efficiency. The achieved scalable performance confirms DART as an efficient indexing method for affix-based searches on HPC systems.

5.4.2 *Load balance at scale.*

As a distributed inverted index, DART is designed to mitigate imbalanced keyword distribution on different nodes. We repeated the series of tests discussed in Section 5.3 at different cluster scales. When testing the key distribution, we set the replication factor of DART to be 1. For testing the request distribution of the WIKI dataset, we set the replication factor of DART to be 3. In order to compare the dispersion of keyword distribution of different datasets on different number of servers fairly, we still use coefficient of variation as our metric (as introduced in Section 5.3), since it removes the impact of the changes in the number of servers and the size of the dataset.



(a) UUID Keyword Distribution (b) DICT Keyword Distribution

(c) WIKI Keyword Distribution (d) WIKI Request Distribution

Figure 10: Comparison of load balance at scale

As shown in Figure 10 (a), (b), and (c), full string hashing has smallest CV at different scales. However, the CV value of initial hashing remains the largest throughout all different scales with an increasing trend. This behavior indicates that, as the number of server grows, the imbalance of the keyword distribution under initial hashing becomes more significant.

On all three datasets, the CV value of DART always remains the second when the number of server increases. As it can be seen in the figure, in both UUID and DICT datasets, DART is able to maintain a small value of CV (under 0.6 at all scales). For WIKI dataset, although the CV value of DART grows as the number of server grows, it is still smaller than that of initial hashing. We consider that DART is able to maintain better balanced keyword distribution than initial hashing.

Figure 10 (d) demonstrates the coefficient of variation for the request of WIKI at different scales. As it can be seen from the figure, DART is able to maintain a more balanced request distribution even though the actual requests on different keywords is highly skewed as we have shown in Figure 2 at Section 2, while full string hashing and initial hashing perform poorly for such highly uneven workload. It is noteworthy that, for achieving this, we only set the replication factor of DART to be 3. If the replication factor increases, DART is expected to achieve even better load balance at scale.

6 RELATED WORK

Many solutions exist for affix-based keyword search. A straightforward approach is to perform string comparison of each data record. Methods including Brute-Force string comparison, Rabin-Karp[10], Knuth-Morris-Pratt[20], finite state automaton for regular

expressions[16] are well studied. However, such approach cannot efficiently handle affix-based keyword search, as they do not support pattern matching directly.

To address efficient affix-based keyword search, various indexing techniques are proposed and shown to be effective. Tree-based indexing techniques are integrated into RDBMS for keyword search. B-Trees, including the original B-Tree and its variants like B+ Tree, B* Tree, etc. [8] are widely used. They are efficient to search through a large number of records on disk and are ideal solutions for many relational databases where disk seeking is frequent and time-consuming. For keyword search, a prefix B+ tree stores the indexed keywords on its leaf nodes and uses the prefixes as the separator keys on the non-leaf nodes. However, to keep all branches balanced, complicated operations are required to adjust the tree structure with each tree modification and has a significant overhead.

Another approach to create index for keywords is the trie data structure, or prefix tree, such as Patricia Tree [19], radix tree [9], ART (Adaptive Radix Tree) [21], etc. These approaches do not need to balance the branches, and hence do not require significant structural changes. An intuitive model of the prefix tree uses each node to represent a character, such that a query on a string or its prefix can be performed by traversing through the path starting from the leading character and ending at the very last end of the string or prefix. Patricia tree [19], a variant of trie, compresses the paths by collapsing all interior vertices that have only one child into one single path and thus reduces the tree height. Radix tree [9] makes brunch comparisons based on its radix, which is 2^s (where the span $s \geq 1$). The increase of the span s will result in the decrease of the tree height, which helps reducing the number of node comparisons and the total search time. While a large span of the radix tree may cause degradation in space utilization, Adaptive Radix Tree [21] is proposed to provide better space utilization and optimized tree height for search efficiency by having different size of tree nodes. The radix of different tree nodes may vary according to the content the node stores. However, these data structures are all designed to address keyword search problem on a single machine. They cannot be directly applied to distributed systems, as node selection and load balance must be taken into consideration for optimized performance.

In cloud computing environment, popular services such as ElasticSearch [11] and SolrCloud [2] can address various text-related queries efficiently. However, these approaches require a number of dedicated servers running services constantly, making them undesirable for HPC applications as maintaining such an indexing cluster can be very expensive. Moreover, such solution follows the document-partitioned approach, query broadcasting is necessary and limits their performance.

A recent HPC-oriented work, TagIt [40], attempted to integrate embedded database like SQLite [42] to achieve efficient keyword search on metadata. However, it requires specific runtime support and also introduces extra cost and overhead to maintain. In addition, it follows the document-partitioned approach which leaves many challenges of affix-based keyword search in a distributed environment unsolved, such as high network traffic due to query broadcasting and load imbalance among indexing nodes. In comparison, DART is designed as a light-weight distributed indexing method which can be easily incorporated into any platform that follows the client-server model, and does not require any other runtime support. Additionally, DART keeps low network traffic by avoid query broadcasting, and the load balancing feature meets the requirement of distributed affix-based keyword search.

7 CONCLUSION

With the goal of developing a distributed affix-based keyword search on HPC systems, we have developed a trie-based inverted indexing technique, called DART (Distributed Adaptive Radix Tree). DART can efficiently locate a keyword based on its partition tree and node selection procedures. Our evaluation results show that DART can achieve efficient affix-based keyword search while maintaining load balance at large scale. Particularly, when comparing with full string hashing DHT, the throughput of prefix search and suffix search using DART was up to 55× faster than full string hashing over different number of servers. The throughput of exact search and infix search on DART remains comparable to that of full string hashing. Compared with initial hashing DHT, DART is able to maintain balanced keyword distribution and request distribution while initial hashing fails to achieve. These two advantages of DART are also independent on dataset, making DART an ideal inverted indexing technique for distributed affix-based keyword search. While DHT is prevalently used for data placement in many applications, in scenarios where affix-based search is required or frequent on HPC systems, DART is a competitive replacement of DHT since it provides scalable performance and achieves load balance at scale. In our future work, we plan to apply DART to in-memory metadata object management systems in searching metadata objects using affixes of user-defined tags or other object attributes.

ACKNOWLEDGMENT

This research is supported in part by the National Science Foundation under grant CNS-1338078, IIP-1362134, CCF-1409946, and CCF-1718336. This work is supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. (Project: Proactive Data Containers, Program manager: Dr. Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility.

REFERENCES

[1] MG Aartsen, K Abraham, M Ackermann, J Adams, JA Aguilar, M Ahlers, M Ahrens, D Altmann, K Andeen, T Anderson, et al. 2016. Search for sources of High-Energy neutrons with four years of data from the Icetop Detector. *The Astrophysical Journal* 830, 2 (2016), 129.

[2] apache.org. 2014. SolrCloud. https://wiki.apache.org/solr/SolrCloud.

[3] Baruch Awerbuch and Christian Scheideler. 2003. Peer-to-peer systems for prefix search. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 123–132.

[4] Dirk Bradler, Jussi Kangasharju, and Max Mühlhäuser. 2008. Optimally Efficient Prefix Search and Multicast in Structured P2P Networks. *CoRR* abs/0808.1207 (2008). http://arxiv.org/abs/0808.1207

[5] Ralph Böhme. 2013. libuuid. https://sourceforge.net/projects/libuuid/

[6] Hailong Cai and Jun Wang. 2004. Foreseer: a novel, locality-aware peer-to-peer system architecture for keyword searches. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., 38–58.

[7] Chi Chen, Zhi Deng, Richard Tran, Hanmei Tang, Iek-Heng Chu, and Shyue Ping Ong. 2017. Accurate force field for molybdenum by machine learning large materials data. *Physical Review Materials* 1, 4 (2017), 043603.

[8] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[9] J. Corbet. 2006. Trees I: Radix trees. http://lwn.net/Articles/175432/

[10] T Cormen, C Leiserson, R Rivest, and Clifford Stein. 2001. The rabin–karp algorithm. *Introduction to Algorithms* (2001), 911–916.

[11] elastic.co. 2017. Distributed Search Execution. https://www.elastic.co/guide/en/elasticsearch/guide/current/distributed-search.html.

[12] Brian Everitt and Anders Skrondal. 2002. *The Cambridge dictionary of statistics*. Vol. 106. Cambridge University Press Cambridge.

[13] Apache Software Fundation. 2017. Apache Lucene. https://lucene.apache.org.

[14] Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. 1992. New Indices for Text: Pat Trees and Pat Arrays. *Information Retrieval: Data Structures & Algorithms* 66 (1992), 82.

[15] Matthew Harren, Joseph Hellerstein, Ryan Huebsch, Boon Loo, Scott Shenker, and Ion Stoica. 2002. Complex queries in DHT-based peer-to-peer networks. *Peer-to-peer systems* (2002), 242–250.

[16] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2006. Automata theory, languages, and computation. *International Edition* 24 (2006).

[17] Yuh-Jzer Joung and Li-Wei Yang. 2006. KISS: A simple prefix search scheme in P2P networks. In *Proc. of the WebDB Workshop*. 56–61.

[18] Yuh-Jzer Joung, Li-Wei Yang, and Chien-Tse Fang. 2007. Keyword search in dht-based peer-to-peer networks. *IEEE Journal on Selected Areas in Communications* 25, 1 (2007).

[19] Donald Knuth. 1997. 6.3: Digital Searching. *The Art of Computer Programming Volume 3: Sorting and Searching* (1997), 492.

[20] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. https://doi.org/10.1137/0206024 arXiv:https://doi.org/10.1137/0206024

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[22] J. Liu, D. Bard, Q. Koziol, S. Bailey, and Prabhat. 2017. Searching for millions of objects in the BOSS spectroscopic survey data with H5Boss. In *2017 New York Scientific Data Summit (NYSDS)*. 1–9. https://doi.org/10.1109/NYSDS.2017.8085044

[23] Yaning Liu, George Shu Heng Pau, and Stefan Finsterle. 2017. Implicit sampling combined with reduced order modeling for the inversion of vadose zone hydrological data. *Computers & Geosciences* (2017).

[24] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. 2008. *Introduction to information retrieval*, Chapter 20.3 Distributing indexes, 415–416. Volume 1 of [26].

[25] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. 2008. *Introduction to information retrieval*, Chapter 4.4 Distributed indexing, 68–71. Volume 1 of [26].

[26] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. 2008. *Introduction to information retrieval*. Vol. 1. Cambridge university press Cambridge.

[27] Arun Mannodi-Kanakkithodi, Tran Doan Huan, and Rampi Ramprasad. 2017. Mining materials design rules from data: The example of polymer dielectrics. *Chemistry of Materials* 29, 21 (2017), 9001–9010.

[28] Marius. 2017. English Words. https://github.com/dwyl/english-words.

[29] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[30] Inc. MongoDB. 2018. MongoDB. https://www.mongodb.com/

[31] Oracle. 2017. MySQL. https://www.mysql.com

[32] David Paez-Espino, I Chen, A Min, Krishna Palaniappan, Anna Ratner, Ken Chu, Ernest Szeto, Manoj Pillay, Jinghua Huang, Victor M Markowitz, et al. 2017. IMG/VR: a database of cultured and uncultured DNA Viruses and retroviruses. *Nucleic acids research* 45, D1 (2017), D457–D465.

[33] PostgreSQL. 2018. PostgreSQL. https://www.postgresql.org/

[34] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M Hellerstein, and Scott Shenker. 2004. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM symposium on principles of distributed computing*, Vol. 37.

[35] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 237–248.

[36] Patrick Reynolds and Amin Vahdat. 2003. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Springer-Verlag New York, Inc., 21–40.

[37] Gerard Salton. 1989. Automatic text processing: The transformation, analysis, and retrieval of. *Reading: Addison-Wesley* (1989).

[38] David J Schlegel, M Blanton, D Eisenstein, B Gillespie, J Gunn, P Harding, P McDonald, R Nichol, N Padmanabhan, W Percival, et al. 2007. SDSS-III: The Baryon Oscillation Spectroscopic Survey (BOSS). In *Bulletin of the American Astronomical Society*, Vol. 39. 966.

[39] Shuming Shi, Guangwen Yang, Dingxing Wang, Jin Yu, Shaogang Qu, and Ming Chen. 2004. Making Peer-to-Peer Keyword Searching Feasible Using Multi-level Partitioning.. In *IPTPS*. Springer, 151–161.

[40] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Geoffroy R Vallée, Seung-Hwan Lim, and Ali R Butt. 2017. TagIt: An Integrated Indexing and Search Service for File Systems. (2017).

[41] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. 2013. Mercury: Enabling remote procedure call for high-performance computing. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 1–8.

[42] sqlite.org. 2017. SQLite. https://sqlite.org.

[43] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. 2017. SoMeta: Scalable Object-Centric Metadata Management for High Performance Computing. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 359–369.

[44] Houjun Tang, Suren Byna, François Tessier, Teng Wang, Bin Dong, Jingqing Mu, Quincey Koziol, Jerome Soumagne, Venkatram Vishwanath, Jialin Liu, et al. 2018. Toward Scalable and Asynchronous Object-centric Data Management for HPC. In *Proceedings of The 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2018*. 113–122.

[45] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks* 53, 11 (July 2009), 1830–1845. http://www.globule.org/publi/WWADH_comnet2009.html.

[46] Brent Welch and John Ousterhout. 1985. *Prefix tables: A simple mechanism for locating files in a distributed system*. Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.

[47] Wikipedia. 2018. Distributed Hash Table. https://en.wikipedia.org/wiki/Distributed_hash_table. Accessed: 2018-04-15.

[48] Dongfang Zhao, Kan Qiao, Zhou Zhou, Tonglin Li, Zhihan Lu, and Xiaohua Xu. 2017. Toward Efficient and Flexible Metadata Indexing of Big Data Systems. *IEEE Trans. Big Data* 3, 1 (2017), 107–117.

[49] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. 2014. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 61–70.

[50] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW '15)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2834976.2834984