ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems [Innovative Systems Paper]

Peng Lu ‡1, Gang Chen #2, Beng Chin Ooi ‡3, Hoang Tam Vo ‡4 , Sai Wu #5

#*Zhejiang University*
2,5{cg, wusai}@zju.edu.cn

‡*National University of Singapore*
1,3,4{lupeng, ooibc, voht}@comp.nus.edu.sg

ABSTRACT

MapReduce has become the state-of-the-art for data parallel processing. Nevertheless, Hadoop, an open-source equivalent of MapReduce, has been noted to have sub-optimal performance in the database context since it is initially designed to operate on raw data without utilizing any type of indexes. To alleviate the problem, we present `ScalaGiST` – scalable generalized search tree that can be seamlessly integrated with Hadoop, together with a cost-based data access optimizer for efficient query processing at run-time. `ScalaGiST` provides extensibility in terms of data and query types, hence is able to support unconventional queries (e.g., multi-dimensional range and k-NN queries) in MapReduce systems, and can be dynamically deployed in large cluster environments for handling big users and data.

We have built `ScalaGiST` and demonstrated that it can be easily instantiated to common B$^+$-tree and R-tree indexes yet for dynamic distributed environments. Our extensive performance study shows that `ScalaGiST` can provide efficient write and read performance, elastic scaling property, as well as effective support for MapReduce execution of ad-hoc analytic queries. Performance comparisions with recent proposals of specialized distributed index structures, such as SpatialHadoop, Data Mapping, and RT-CAN further confirm its efficiency.

1. INTRODUCTION

By design, traditional parallel database systems are optimized for fairly static environments with a relatively small number of high-end machines. While this architecture provides the desired performance, its capability is limited in scaling dynamically with loads and needs. To take advantage of dynamic cluster environments comprising a large number of commodity machines, MapReduce was first introduced by Dean and Ghemawat [10] to simplify the building of web-scale inverted indexes, and the framework has gained fast popularity as the state-of-the-art of data parallel programming model.

The framework can be used to evaluate more complex data analytical tasks by executing a series of MapReduce jobs [24]. However, based on the evaluation of an open-source implementation of MapReduce, namely Hadoop[1], the framework has been noted to yield sub-optimal performance in the database context [24]. MapReduce does not exploit any type of indexes, and current state-of-the-art cloud storage systems such as Bigtable [5], Cassandra [18], and HBase[2] have not been designed to support general purpose indexes for MapReduce processing.

Consider the following example query which tries to generate the statistics about the regional behavior of users in a certain age group:

```
SELECT count(*)
FROM mobile m, user u
WHERE m.x< x0 + α and m.x> x0 − α and
m.y< y0 + β and m.y> y0 − β and m.uid=u.uid
m.dataUsage>3000MB and u.age>20 and u.age<30
GROUP BY u.age
```

Current way of processing the query is to assemble a MapReduce job comprises of two types of *mappers*. One type of *mapper* scans the mobile table and filters the tuples based on the data usage and location information. The other type scans through the user table to retrieve the users with ages within the query range. Both types of *mapper* shuffle the data to the *reducers* by using *uid* as the partition key. Then a *reducer* can generate the partial results for its designated *uid*. But to merge the users from the same age group, one additional MapReduce job is required to aggregate the results using *age* as the key .

In fact, the above MapReduce query processing strategy incurs unnecessary I/O overheads, which can be effectively avoided by using indexes. The challenge, however, is that we need to build various types of indexes to support the query. For example, an R-tree index can be employed to locate the mobile phones within the range $[(x_0 - \alpha, x_0 + \alpha), (y_0 - \beta, y_0 + \beta)]$. An un-clustered B$^+$-tree index can be built for attribute *dataUsage* to track user's data consumption. For the user table, a clustered B$^+$-tree index is preferred as it can efficiently prune the users that are not in the qualified age group. With those indexes available, the *mappers* can push down query predicates for the indexes to evaluate and scan only the tuples that contribute to the query result, in which way query performance can be significantly improved.

There have been several proposals on distributed index schemes in cloud environments. For example, a distributed B$^+$-tree-like index was proposed to support single-dimensional range queries [27]. To facilitate multi-dimensional queries, SpatialHadoop [14] realized multi-dimensional indexes in Hadoop using specialized index operators, and can support several types of spatial indexes, such as R-tree and grid files. Another distributed R-tree-like index

[1]http://hadoop.apache.org
[2]http://hbase.apache.org

to support multi-dimensional range and k-NN (k-nearest neighbors) queries was introduced in [26]. However, these index schemes are specialized in a certain type of index. In order to support multiple indexes of different types, which is a desirable feature for MapReduce-based query processing systems, these distributed index schemes need to be implemented and deployed separately on the same cluster. Such an approach results in high index maintenance overheads, and lacks code and interface reuse.

More recently, a generalized distributed indexing framework based on Cayley graph model has been proposed to address the scalability and performance issues of supporting a large number of indexes of different types such as hash, range, and multi-dimensional indexes in dynamic cluster environments [6]. Although this work provides a generic framework for the declaration and implementation of scalable distributed indexes, it requires users to define a data mapping function for each specific index (for example, mapping from multi-dimensional to single dimensional data). This could result in inefficiency of range query processing because such data mapping may incur redundant I/Os for the system having to scan false positive candidates.

In summary, the challenges of incorporating indexes for the *MapReduce* and other large-scale data processing systems are:

1. To support different types of applications and queries, a general indexing framework is required which can be used to build all popular indexes, such as B$^+$-tree index and R-tree index, for the distributed systems. It should also provide unified interfaces for users to implement new types of index.

2. The framework should work as a non-intrusive component for existing systems such as MapReduce so that the previous algorithms written for those systems do not need to be modified to exploit the benefit of index-base processing.

3. As an index service for parallel data processing, the design of index framework must consider the efficiency, reliability and scalability as its first class citizen.

Based on the above considerations, in this paper, we present an indexing framework, `ScalaGiST` – Scalable Generalized Search Tree – which is intrigued by classical Generalized Search Tree (GiST) [16]. GiST provides functionalities of various types of database search trees in a single package, while `ScalaGiST` is designed for dynamic distributed environments such as in-house clusters and public clouds so as to handle large-scale datasets and adapt to changes in the workload while leveraging commodity hardware. `ScalaGiST` is extensible in terms of both data and query in that it enables users to define indexes for new type of data and provides efficient lookup over the index data as built-in functions without the need of data mapping as being used in other distributed indexing frameworks [6, 22].

Indexes in `ScalaGiST` are distributed and replicated among index servers in the cluster for scalability, data availability and load balancing purposes. `ScalaGiST` develops a light-weight distributed processing service to process the request in parallel and effectively reduce the overhead of searching over a large index. `ScalaGiST` is designed as indexing service and can work with other systems in a non-intrusive way. In this paper, we show how to embed `ScalaGiST` into the execution of MapReduce-based systems by only launching appropriate map tasks on selected data chunks containing records (for primary indexes) that satisfy the query predicate. This strategy creates opportunities in reducing the startup cost of MapReduce jobs, and most importantly, avoids unnecessary I/Os and computation that do not eventually contribute to the query results. While secondary indexes facilitate a more direct location of data of interest, they may incur non-negligible cost due to random accesses to the base data. Therefore, `ScalaGiST` develops a data access optimizer to compare two possible query execution plans, namely index scan and full table scan, and choose the better plan before running the query.

In summary, the contributions of the paper are as follows.

- We introduce `ScalaGiST` – scalable generalized search tree for dynamic cluster environments such as the Cloud. It provides extensibility in terms of data and query types for supporting unconventional queries (e.g., multi-dimensional range and k-NN queries), and more importantly, can be dynamically deployed on large clusters for handling big users and data.

- We present an approach to integrating `ScalaGiST` seamlessly with Hadoop platform, coupled with a cost-based data access optimizer for improving the performance of MapReduce execution.

- We have built `ScalaGiST` and conducted an extensive performance study on an in-house cluster. We compare the R-tree and B$^+$-tree-like indexes implemented using `ScalaGiST` with recent indexes such as Data Mapping [6], RT-CAN[26] and SpatialHadoop[14]. The results confirm its efficiency and scalability in terms of write and read performance, as well as effective support of exact match, range, and similarity queries.

The remainder of the paper is organized as follows. We review related work in Section 2. We present the overall architecture and system implementation of `ScalaGiST` in Section 3 and Section 4 respectively. Its integration with Hadoop and data access optimization are described in Section 5, and an extensive performance evaluation is presented in Section 6. We conclude the paper in Section 7.

2. RELATED WORK

In this section, we review related work and highlight the contributions of our proposed scalable index structure – `ScalaGiST` – for cloud environments.

2.1 MapReduce-based Systems

MapReduce [10] has become the state-of-the-art data parallel processing framework for solving complex problems over large-scale datasets. Nevertheless, as shown by research in the database community, Hadoop – an open-source implementation of MapReduce – achieves sub-optimal performance when being used to process database analysis tasks [24]. As a result, many extensions and augmentations have been proposed [19]. In order to utilize sophisticated technologies from the database community, HadoopDB [2] introduces a hybrid architecture bring together the MapReduce runtime and database engine. It makes use of a cluster of local databases as the storage layer for MapReduce processing engine.

Another line of research focuses on tweaking the core of Hadoop. For example, Jiang et al. [17] have identified five design factors that affect the performance of Hadoop MapReduce including I/O modes, indexing utilization, record parsing, grouping algorithm and scheduling policy. Note that the index scheme considered in [17] is restricted to sorted files such as range-index and B$^+$-tree files, which have limited usage in a wider range of database applications that require multiple indexes of different types.

Similarly, other research proposals, Hadoop++ [12] and HAIL [13], also aim to optimize the performance of Hadoop via exploitation of indexes. These systems reduce overall I/O cost by utilizing local index on the input of map tasks. Split-oriented indexes for Hadoop was proposed in [15], and SpatialHadoop [14] was implemented as an extension of Hadoop to support multi-dimensional queries. However, unlike ScalaGiST, these indexes are not generic and cannot support user-defined index and unconventional queries (e.g., metric space query).

Other research in this trend build global B$^+$-tree-like indexes [29] or multi-dimensional indexes [20] on top of HDFS. These works all focus on building a specific type of index for an individual system.

2.2 Distributed Data Structures for Cluster Environments

A thorough survey and feature comparison of distributed data structures for cluster environments can be found in [4]. Bigtable [5] and its open-source implementation HBase provide record oriented access to very large tables which are distributed in commodity clusters consisting of thousands of machines. Dynamo [11] and Cassandra [18] are other popular key-value stores with some different features and design considerations.

Given a key, the key-value systems can efficiently locate the value associated to the key. These systems build a global index over the underlying storage layer, using key space as the partition factor. More complex indexes, such as multi-dimensional indexes [22], are also introduced to the key-value store (HBase).

Motivated by the fact that existing distributed indexing services are mainly designed for some specific type of index, which greatly limits the much needed versatility of supporting various types indexes, Chen et al. [6] proposed a comprehensive distributed indexing framework based on the generic Cayley graph model. It is designed to support key-based data retrieval, while ScalaGiST is tailored for the MapReduce system to process large analytic queries.

2.3 Generalized Search Trees (GiST)

Generalized Search Tree (GiST) [16] was proposed as a framework that aims to provide the functionalities of various types of indexing search trees. What makes the GiST index structure efficient and suited to broad applications is its extensibility in terms of both data and query types. That is, instead of implementing indexes from scratch, users are able to customize indexes over various type of data while still being guaranteed with the efficiency.

To customize the GiST as a search tree for a specific data type, users only need to implement four abstract methods defined in the GiST package, which determine the behavior of data keys handled in the tree. In this paper, we extend the idea of GiST to dynamic cluster environments and present ScalaGiST, scalable generalized search tree with the ability to distribute indexes across machines for scalability and load balancing, and demonstrate its usefulness for MapReduce execution.

3. ARCHITECTURE OVERVIEW

ScalaGiST is designed as a scalable and non-intrusive indexing framework for MapReduce systems as illustrated in Figure 1. The index is organized as a tree structure and stored as a sequential file in the DFS (HDFS in this paper). Similar to a normal DFS file, the index file is also partitioned into multiple chunks. Each chunk, in fact, contains the index data of one or multiple sub-trees of an index. Given limited memory, ScalaGiST selectively loads some index chunks into memory. More specifically, ScalaGiST employs a metastore to collect the query statistics, based on which a prediction model is applied to generate a caching strategy to maximize the performance (the details will be discussed in the next section.) If multiple indexes are created, the metastore also maintains the basic information of the index including:

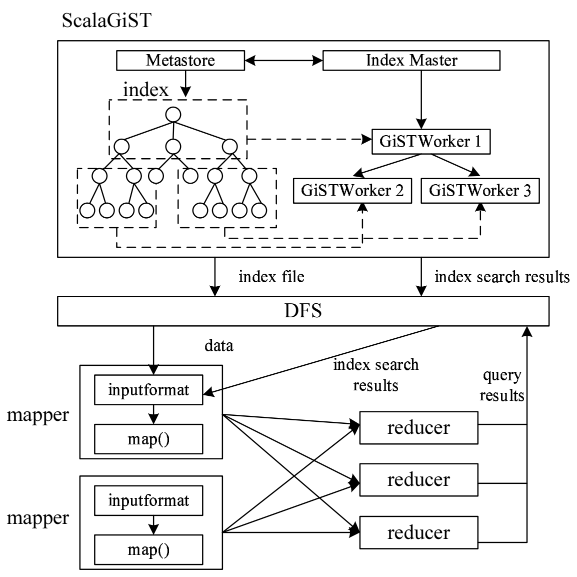


Figure 1: Overview of ScalaGiST.

1. The DFS file for which the index is constructed. By default, ScalaGiST considers all DFS files as an unstructured format, where each line contains a key and a value. If it is built for the relational data, the user should define a parser for the DFS file and specify the indexed column.

2. The role of the index, e.g. primary and secondary. Primary index is built for the sorted data and if the data are not sorted by the indexed column, ScalaGiST will invoke the tera-sort algorithm of MapReduce to do the sorting.

3. The type of indexes. ScalaGiST includes B-tree index and R-tree index in it implementations. For other customized indexes, the user can register their index types in the metastore via the interface provided by ScalaGiST.

To process index search requests, ScalaGiST develops a light-weighted distributed processing service which includes an index master and multiple index workers. As shown in Figure 1, each index worker handles one index chunk by scanning the file and materializing the index in memory. Among the workers, a specific worker (worker 1 in Figure 1) is responsible for the root node. Once receiving a search request, the index master forwards it to the worker hosting the root node, which progressively forwards the request to the other workers. When receiving the request, all workers start the search concurrently to exploit the parallelism to maximize the performance. To reduce maintenance overhead, when no query is being processed, the worker process releases all its resources.

The search results of ScalaGiST are offsets of the DFS that refer to the tuples that satisfy the predicates. The offsets are flushed back to the DFS as a temporary file. In ScalaGiST, we provide a specific IndexInputFormat for MapReduce runtime to read the temporary files. In the ideal case, if `ScalaGiST` finds that a data chunk cannot contribute to any query result, the corresponding *mapper* will not be scheduled. In most cases, the *mapper* will adopt the skip-and-scan strategy to read a few tuples that qualifies the predicates, which effectively reduces the processing cost.

4. SYSTEM IMPLEMENTATION

In this section, we list the interfaces of `ScalaGiST` and the techniques adopted in our implementation. More specifically, we discuss how multiple indexes are used together to facilitate the query processing and how the memory is exploited to reduce the index search overhead.

4.1 Interface of ScalaGiST

The essence of `ScalaGiST` is to provide template algorithms for traversal and modification of the tree structures distributed in dynamic environments. These algorithms are designed and implemented to work with a generic class of data key.

Unlike classical B-trees whose keys typically are numerical values or short strings, `ScalaGiST`'s keys are instances of a user-defined class overrides the abstraction key class. This capability allows users to define new types of indexes by customizing the key class.

As an illustration, `ScalaGiST` can be instantiated as a distributed B$^+$-tree-like index structure by defining keys as ranges of numbers, which means that all index entries descending from a certain index node have the values between the range. Similarly, `ScalaGiST` can be instantiated as a distributed R-tree-like index structure by defining keys as bounding boxes so that all index entries descending from a certain index node are bounded by the box – in this paper, we use the term "node" and "page" interchangeable.

Overall, in order to instantiate `ScalaGiST` as a specific type of search tree, the only thing that users are required to do is to define what represents a key, and implement, i.e., override, abstracted methods in the key class as discussed below. These methods will be invoked at runtime by the template algorithms implemented within the `ScalaGiST` framework to realize basic tree operations such as search and modification.

- **Consistent**$(N.p, q)$. This method provides the basis for guiding the search operation correctly. It takes as input two parameters, namely a key predicate p of a tree node N and a query predicate q. It returns true if both p and q are satisfied for a given data key, and returns false otherwise.

- **Penalty**(e, N). This method provides an indication of the cost if the new index entry e is inserted to the subtree rooted by node N. The path that has the least penalty in the tree is chosen for inserting the new entry.

- **Union**(S). This method defines how to merge a set S of index nodes. It returns a new key predicate p that evaluates to true for all the index entries contained in or reachable from the index nodes in S.

- **PickSplit**(N). This method is invoked when there is a node split upon the insertion of a new index entry. It decides which index entries stay on the old node, and which ones go to a newly allocated index node.

- **Parse**$(InputStream)$. This method reads the binary data from the DFS and parses it into a user-defined tree node.

- **Store**$(N, OutputStream)$. This method serializes node N into its binary representation and flushes it back to the DFS.

In `ScalaGiST`, users can define a customized node type, and thereby a new index type, by implementing the above interfaces. An abstract class, ***GiSTWorker***, is used as our index processing unit and users should pass the node definition as a template to the worker process. The declaration of ***GiSTWorker*** class is:
`abstract class GiSTWorker<AbstractNode>`

As shown in Figure 1, we actually maintain multiple nodes in one ***GiSTWorker***. The ***GiSTWorker*** loads a data chunk of the index file and adopts the `Parse` method to reconstruct the tree nodes. For each node, ***GiSTWorker*** invokes the user-defined function to process the request. In current implementation, `ScalaGiST` has created two sub-classes of the *AbstractNode*, namely the *BTreeNode* and *RTreeNode* for supporting the B-tree index and and R-tree index respectively. In the experiments, we also show that using `ScalaGiST`, we can build various types of indexes, such as a metric index MTree [8], by overriding the interface functions. The new indexes can provide a scalable performance as well. In the following discussion, we use the R-tree as our running example to demonstrate the index construction and search process in `ScalaGiST`.

4.2 Tree Methods

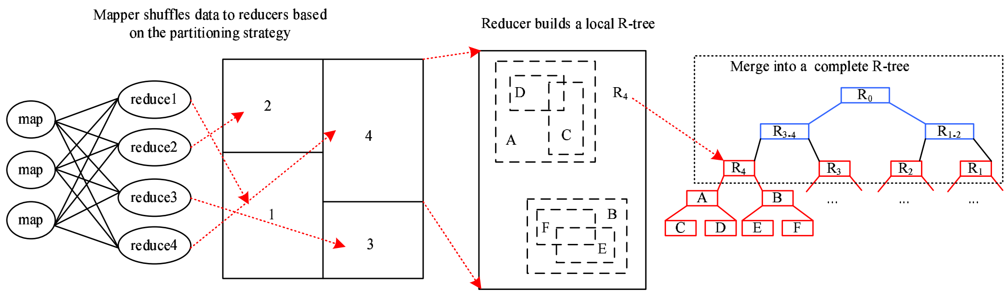
4.2.1 Index Construction

When a user requests to build an index using `ScalaGiST`, a new MapReduce job is submitted for the index construction. Figure 2 illustrates the idea of how an R-tree index is built. `ScalaGiST` first randomly picks K samples from the indexed attributes and then partitions the key space into W sub-spaces (W is the number of *reducers* used to construct the index), so that each sub-space has the same number of samples. For single dimension case, the partitioning process works as building a equal-depth histogram, while for multi-dimension case, it just simulates the KD-tree algorithm. After the partitioning, C *mappers* are started to scan the data where C is the total number of data chunks. Each *mapper* generates W intermediate files, recording how tuples are distributed to different sub-spaces. In the *reduce* phase, each *reducer* collects the intermediate files from the *mappers* for a specific sub-space and constructs a local R-tree using Algorithm 1. In the ChooseSubtree method, we apply the user-defined `Penalty` function to recursively select the subtree that the key should be inserted into until reaching the leaf node. After inserting the new key, we check whether the node needs to split. If so, a recursive split process is invoked using the `PickSplit` function.

After the MapReduce job completes, the index master reads the root nodes of all sub-trees (e.g., R_1, R_2, R_3 and R_4 in Figure 2)

Algorithm 1 Insert(Key key)

```
1: AbstractNode root = getRoot()
2: if root==null then
3:     createRootNode(key)
4: else
5:     AbstractNode node = ChooseSubtree(root, key)
6:     node.insert(key)
7:     if node.needSplit() then
8:         Key splitkey=PickSplit(node)
9:         create two new nodes based on the splitkey
10:        notify the parent node about the two new nodes and do
           the recursive split if necessary
```



Mapper shuffles data to reducers based on the partitioning strategy
Reducer builds a local R-tree
Merge into a complete R-tree
map
map
map
reduce1
reduce2
reduce3
reduce4
1
2
3
4
D
A
C
B
F
E
R_4
R_0
$R_{3\text{-}4}$
$R_{1\text{-}2}$
R_4
R_3
R_2
R_1
A
B
C
D
E
F

Figure 2: Building an R-tree Index

and builds a top R-tree by using those root nodes as its leaf nodes. The top R-tree is the merging result of the sub-trees. Both the top R-tree and the sub-trees are written back to the HDFS as index chunks. In particular, the sub-trees are serialized into a sequential file based on the in-order traversal. The sub-tree in Figure 2 is serialized as *C, D, A, E, F, B,* R_4. For a large sub-tree, it may be stored as multiple index chunks. Suppose *C, D, A* and *E, F, B,* R_4 are two index chunks. For the second chunk, we include a pointer to the offset of A in the first chunk to indicate the position of the left child of R_4.

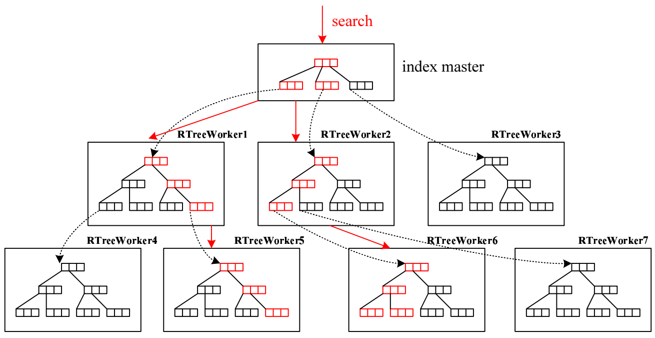


Figure 3: Search with R-tree

To give an detailed illustration of the index construction process, here we show the performance breakdown of indexing 10 GB 2-dimensional data. There are two strategies that are commonly used for R-Tree construction, namely sequential insertion (insert data points one by one sequentially) and bulk loading. We implement both methods and compare their efficiency. The data set contains approximately 10 million records.

	Sequential Insertion	Bulk Loading
No. of Mapper	20	20
Map Time (second)	151	151
No. of Reducer	40	40
Reduce Time (second)	1094.2	26.63
Sub-Tree Merging (millisecond)	178	178

Table 1: COMPARISON OF INDEX CONSTRUCTION STRATEGIES

As shown, bulk loading outperforms sequential insert by a large margin. There are in total 20 mappers launched. Each mapper reads in 512MB of data, maps them to sub-spaces and shuffles the data to 40 reducers. This process takes up to 2.5 minutes (151 seconds) for both methods, inclusive of instantiation time of the job. When a reducer receives the data, it performs R-Tree construction (locally). In this phase, sequential insertion spends 18 minutes due to large amount of keys (over 10 millions), whereas bulk loading is very efficient and costs only 27.19 seconds. The final merge phase reads in the root nodes of local R-Trees as leaf nodes and inserts them into a top layer R-Tree. This phase involves reading the 40 root nodes from DFS and inserting them into an in-memory R-Tree, which is rather fast and can be finished in 178 milliseconds. It is notable that sequential insertion is directly supported by the GiST interface for all types of indexes, while bulk loading requires some customized codes for each index.

4.2.2 Search

Figure 3 shows how the *GiSTWorker* processes a range query. It simulates the typical tree search algorithm. The search starts from the index master which maintains the top R-tree. Based on the search range, it sends the query to *GiSTWorker1* and *GiSTWorker2*. The two workers start the tree search in parallel. When reaching the leaf nodes, a worker checks whether it can return the result or it needs to forward the search message to other workers. Algorithm 2 illustrates the local search process inside each worker. The index master monitors the whole search process. Once it detects that all workers have finished their tasks, it notifies the MapReduce scheduler for further query processing.

Algorithm 2 GiSTWorker.Search(Query q)

```
1:  Set<AbstractNode> nodes = new Set($root$)
2:  while nodes.hasMoreElement() do
3:      AbstractNode $\bar{n}$ = nodes.next()
4:      if Consistent($\bar{n}$.key, $q$) then
5:          if $\bar{n}$.isLeaf() then
6:              result.add($\bar{n}$)
7:          else
8:              Set<AbstractNode> child = $\bar{n}$.getChild()
9:              while child.hasMoreElement() do
10:                 AbstractNode c = child.next()
11:                 nodes.add(c)
12: return result
```

On top of range search, we provide two k-NN algorithms, one is generic for all indexes defined by the `ScalaGiST` interface and one is specific for the R-tree index. The generic k-NN algorithm iteratively expands its search range until k results are obtained. Suppose the query point is $p = (v_1, ..., v_d)$. We submit an initial query as $Q_0 = ([v_1 - r_0, v_1 + r_0], ..., [v_d - r_0, v_d + r_0])$. If more than k results are obtained, the search completes. Otherwise, we enlarge the search range by θ. So the new query is $Q_1 = ([v_1 - r_0 - \theta, v_1 + r_0 + \theta], ..., [v_d - r_0 - \theta, v_d + r_0 + \theta])$. To avoid repeating the search of the initial query, we also include Q_0 when processing Q_1. The query will not be sent to the tree nodes that only overlaps with Q_0. r_0 and θ are two tunable parameters in the k-NN search which have been well studied [25]. Using their analysis, we set $r_0 = \theta$, and θ is estimated as:

$$\theta = \frac{D_k}{k} = \frac{2\sqrt[d]{\Gamma(\frac{d}{2}+1)}}{k\sqrt{\pi}(1 - \sqrt{1 - \sqrt[d]{\frac{K}{N}}})}$$

where D_k represents the distance between the kth nearest neighbor and the query point and N is the estimated number of data in the whole space. Γ is a function defined as: $\Gamma(x+1) = x\Gamma(x)$ with $\Gamma(1) = 1$ and $\Gamma(0.5) = \frac{\pi}{2}$. We start with an initial θ value and each time when we complete a range search, we will update our estimation for D_k, so the next range is more accurate and tight.

We also include the classic branch-and-bound k-NN algorithm for R-tree which provides a near optimal performance [23]. It first retrieves the nearest neighbor to the query point and then computes its minimal and maximal distances to the nearby bounding boxes. The distances are used to expand the search range and decide when to terminate. In fact, both the performances of the two k-NN algorithms differ marginally in both our experiments and the theoretic analysis [25].

It is also noted that different index construction strategies has substantial impact on search performance [7]. We use the two R-Trees populated by sequential insertion and bulk loading in the last benchmark, and compare their query performance using range search and k-NN search.

	Sequential Insertion	Bulk Loading
Range Query (second)	0.185	10.28
k-NN Query (second)	3.97	27.19

Table 2: COMPARISON OF QUERY PERFORMANCE

Table 2 shows that, although bulk loading speeds up index construction, it compromises the query performance to some extend comparing to that of sequential insertion, because one-by-one insertion allows the index to adaptively pick a better sub-tree in the R-tree to insert and reduces the size of bounding boxes. In the above evaluation, query performances for range query and k-NN query are about 55 times and 7 times worse when using bulk loading. Therefore, there is a trade-off between index construction cost and run-time performance. It is up to the users to decide which method to use.

4.2.3 *Insertion*

As an indexing service for the MapReduce system, ScalaGiST only supports batch insertion. For a new batch of data, instead of appending them to the existing DFS file, we import them as a new file under the same directory. ScalaGiST checks the metastore whether we need to build indexes for the new data and starts the index construction process if necessary. ScalaGiST creates a new index tree and registers it in the index master. Therefore, for an increasing dataset, we may have multiple index trees and we will route the query to all trees for processing.

Periodically, ScalaGiST merges the index trees to reduce the search cost. Let T_0 be the original tree and T_1,...,T_k be the new trees. T_1,..,T_k, in fact, are discarded and we build a new T_0 by inserting their data into T_0. We start a MapReduce job to perform the merging. Specifically, in the *map* phase, we scan data files of T_1,..,T_k and partition them based on the same partitioning strategy of T_0. In the *reduce* phase, the *reducer* loads its specific sub-tree of T_0 and inserts the new data into T_0. After all *reducers* complete their insertion, the index master will generate a new index tree for all existing data files. For example, in Figure 2, the new data are partitioned into four sub-spaces defined by the previous sampling process. In *reducer4*, we load R_4 from the DFS and insert the received data into R_4's sub-tree. The insertion process applies the user-defined **Consistent**, **Penalty** and **PickSplit** functions to guide the tree construction. After the MapReduce job, a new R_4 covers all existing data in the sub-space is built.

4.3 Search with Multiple Indexes

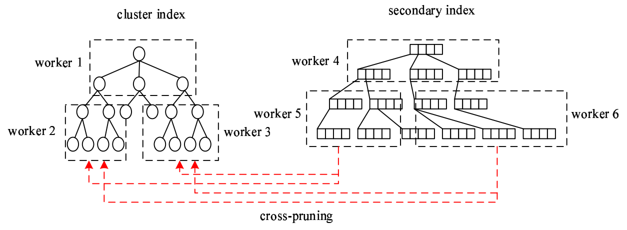


Figure 4: Search With Multiple Indexes

One of the most distinguished features of ScalaGiST is its capability of supporting various types of index. As shown in our example query in the introduction section, we can build both a B-tree index and an R-tree index for different attributes of a table. In ScalaGiST, only one clustered index can be built for a table, while the number of secondary indexes is not limited. Figure 4 shows how ScalaGiST exploits multiple indexes together to process the query.

Suppose we have a clustered B-tree index which is disseminated to worker 1, worker 2 and worker 3. We also have a secondary R-tree index which is maintained by worker 4, worker 5 and worker 6. Given a query with two predicates, p_0 and p_1, suppose p_0 is on the clustered attribute and p_1 is on the other attributes. ScalaGiST splits the search into two parts. p_0 is forwarded to worker 1 to worker 3, while p_1 is forwarded to the other 3 workers. All the workers start their search in parallel. When we complete the search of clustered index, suppose only worker 2 has the query results (DFS offsets that point to the corresponding tuples). Instead of returning the results to the applications, e.g., MapReduce jobs, worker 2 waits for the search results of the secondary index. Once worker 5 and worker 6 finish their search, they broadcast their results to worker 2 and worker 3. As a matter of fact, worker 3 will not be notified by ScalaGiST, as it does not have the query results for p_0. On the other hand, worker 2 will merge its results with the results from the other two workers. The final index search results are then returned to the users. The flexibility of ScalaGiST allows us to link the workers in an arbitrary way, simplifying the search algorithm design for the multiple indexes.

4.4 Memory Management

After a worker completes the job, we destroy its memory stacks and reclaim all the used memory. When the next query comes, ScalaGiST will wake up the worker and reconstruct its states. Such initialization cost and the cost of loading index nodes from the DFS into memory cannot be ignored. One way to address the problem is to maintain some workers and their states in memory. In other words, those workers are maintained as a "persistent worker" in ScalaGiST. They are always running, waiting for receiving the requests from the users. Their states, e.g., the tree structure, are also cached in memory. This is similar to using the RDD as the storage in Spark [28]. However, given limited memory, we must adaptively select the memory-resident workers to maximize the performance.

DEFINITION 1. **Benefit of A *GiSTWorker***
The benefit of a GiSTWorker *regarding to a query q is defined as the total size of index tree nodes (except the root nodes) that are required to read from the DFS to process q.*

In ScalaGiST, we record the last processed k queries in the metastore and use that statistics to measure the benefit of buffering each *GiSTWorker*. So the memory management problem is transferred into an optimization problem:

DEFINITION 2. **Optimal Buffering Strategy**
Given a query set Q and a GiSTWorker *set U, suppose we only have limited memory M, we want to select a subset* GiSTWorkers *$\bar{U}$ from U, so that:*

1. *The memory for buffering* GiSTWorkers *in $\bar{U}$ is less than M.*
2. *For any other subset $\widehat{U} \subseteq U$ satisfying the memory constraint, its benefit is less than that of $\bar{U}$.*

Note that although all *GiSTWorkers* handle the same-size index chunks, when materializing the tree nodes in memory, the *GiSTWorkers* require different sizes of memory because the index nodes may have different data structures. Therefore, the optimal buffering strategy is, in fact, a set-packing problem which is NP-hard. In `ScalaGiST`, we adopt a greedy-based heuristic approach as shown in Algorithm 3. The intuition is to compute a score for each worker as $\frac{benefit}{memory\_size}$ and rank workers based on the scores. The top ranked workers are set as "persistent workers" which are maintained in memory for speeding up the processing.

Algorithm 3 ManageBuffer()

```
1: for each GiSTWorker $u \in U$ do
2:     $u$.score = $u$.benefit/$u$.memory_size
3: Heap $H$ = sortByScore($U$)
4: while $\bar{U}.size < M$ and $H$.size>0 do
5:     Worker u = $H$.pop()
6:     $\bar{u}$.add(u)
7: return $\bar{u}$
```

In `ScalaGiST`, Algorithm 3 is invoked periodically to adjust the buffer strategy. We provide a parameter for users to tune the frequency. By default, Algorithm 3 is invoked only when new indexes have been created since the last adjustment.

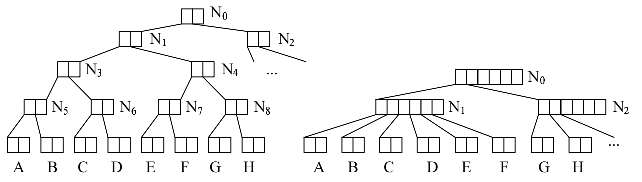


Figure 5: Effect of Fanout

4.5 Tuning the Fanout

For tree-based index, fanout F affects the search performance. We illustrate the problem using Figure 5. Suppose each index chunk can maintain three leaf nodes or 6 pointers of the internal nodes. For the left binary tree, the workers and their tree node assignment is: { *(W_1: A, B, N_5), (W_2: C, D, N_6, N_3), (W_3: E, F, N_7), (W_4: G, H, N_8, N_4, N_1)*, ...}. For the right tree, the tree node assignment is: {*(W_1: A, B, C), (W_2: D, E, F), (W_3: N_1)*,...}. Given a query that retrieves data from leaf nodes B, C, D and E, the left tree first forwards the query to W_4. W_4 then forwards the query to W_2 and W_3. W_2 further forwards the query to W_1. There are totally four workers involved in the processing and three workers (W_1, W_2 and W_3) perform their jobs concurrently. On the contrary, in the right tree, the query is first routed to W_3 and then forwarded to W_1 and W_2. Only two workers can run concurrently. However, the left binary tree requires a longer search path $W_4 \rightarrow W_2 \rightarrow W_1$, while the right tree has a much shorter path $W_3 \rightarrow W_1$. In summary, the fanout has the following two properties:

1. A small fanout can increase the level of parallelism by involving more workers in the processing. This can effectively improve the performance when the query needs to retrieve a large portion of data. But it also incurs more communication costs, when only a few results are required.
2. A large fanout can reduce the search path and hence, lead to a lower communication and I/O cost. However, it may result in load imbalance, as the query is processed by fewer workers.

In fact, most B$^+$-tree style hierarchies in production such as Bigtable [5] and HBase also use a small number of levels and very high fanouts. This is because they are targeting at high-selective queries (e.g., key-based retrieval). `ScalaGiST`, on the other hand, is designed for the MapReduce system. So we also want to benefit the large analytic queries. In `ScalaGiST`, we group the historical queries into two categories, high-selective queries and large analytic queries. Our purpose is to estimate a fanout F that can achieve a good performance for both types of queries.

We apply a coarse estimation and for space limitation, we use the single-dimension index to briefly demonstrate the idea. We assume that the index evenly partitions the key range. Therefore, at level l, we have F^l leaf nodes and the domain is partitioned into F^l sub-ranges. Let t be the size of a tree node and C be the size of DFS file chunk. Each worker handles $\frac{C}{t}$ index nodes. Using the partitioning strategy shown in Figure 1, we can estimate how the $F^0 + ... + F^l$ nodes are distributed to different workers. Given a list of historical queries $\{q_0, ..., q_k\}$, we can also estimate how many workers are involved for each query. Based on the query pattern, we set two selectivity thresholds θ_x and θ_y. For queries with selectivity smaller than θ_x, we want to set a F that only one worker is involved in the search concurrently. So we can get a lower bound ϵ_{low} for F. For queries with selectivity larger than θ_y, we want as many index workers as possible in the search process. Namely, the involved index workers are no less than W where W is the number of available cluster nodes that `ScalaGiST` is deployed on. This constraint can generate an upper bound ϵ_{up} for F. Let $\{\theta_0, ..., \theta_n\}$ denote the selectivities of historical queries on table T, and e as the size of each leaf index entry. The total cost can be estimated as (detailed cost model will be discussed in Section 5.2.2):

$$\sum_{i=0}^{n} \lceil \frac{\sum_{j=1}^{\log_F |T|} \theta_i F^j e^2}{W} \rceil$$

We then iterate F in the range of $[\epsilon_{low}, \epsilon_{up}]$ and compute the above equation respectively. `ScalaGiST` selects the F value that minimizes the cost estimation.

5. HADOOP INTEGRATION AND DATA ACCESS OPTIMIZATION

Unlike existing proposals [3, 6, 27, 26], our proposed `ScalaGiST` has been designed for seamless integration with Hadoop and its data access optimization algorithm helps MapReduce select an index scan versus a full table scan method depending on characteristics of queries.

5.1 Leveraging Indexes in Hadoop

In `ScalaGiST`, we implement an `IndexInputFormat` class for `ScalaGiST` so that its data can be accessed by MapReduce. This class overrides the required public methods such as `getSplits()` and `createRecordReader()`.

More specifically, the `getSplits()` routine will be called during the starting up of a MapReduce job to identify how the index data are split into chunks, which in turn will determine the number of map tasks that are required to execute the query processing job. Given a query, the system parses its range predicates and composes an appropriate `Scan` operator on the corresponding index. All index pages between the start and end keys of the range `Scan` operator are included for processing the query. `ScalaGiST` splits the index data at index pages' boundaries, and therefore the number of splits is essentially the number of index pages located within the query range.

During the execution of the MapReduce job, the framework iterates over the splits and calls the `createRecordReader()` for each split. Each calling of `createRecordReader()` with a split creates a new `IndexRecordReader` to access the corresponding index page. That is, each `IndexRecordReader` handles exactly one index page and process every index entry between the index page's start and end keys in two steps: (1) retrieving the base record referred by the index entry, and (2) mapping the record based on the map function customized for processing the query and shuffling intermediate data to the appropriate reduce task.

Overall, in this MapReduce execution with index scan, the selectivity of the query predicate determines the number of map tasks to be launched for processing the query. By utilizing the index, the number of needed maps is restricted to the minimal and only relevant records satisfying the query predicate are retrieved from the base table, thus reducing the task's startup and I/O cost significantly. However, the benefits of this index scan execution do not come for free. In fact, this index scan strategy introduces other overhead that does not exist in the full table scan approach. A cost model is therefore essential and proposed in the following section to estimate the performance of the two strategies so that the system is able to choose the optimal one for query execution.

5.2 Data Access Optimization Algorithm

To identify an optimal access method, we build histograms to collect statistics of data distribution and design a cost model to select the data access plan.

5.2.1 *Construction of Histograms*

At regular time, the system runs a background MapReduce job for constructing histograms of tables. Suppose $a_0, a_1, ..., a_{n-1}$ are columns of table T and $[l_i, u_i]$ is a_i's domain. We build an equal-width histogram for each column. That is, we split $[l_i, u_i]$ into B buckets, and for each bucket, we count the number of tuples whose attribute value falls within the bucket. In the map phase, we generate a composite key for each tuple. Key-value pairs follow the format of $<(columnID, bucketID), 1>$, where $columnID$ is the unique ID of the column and $bucketID$ is the bucket ID of the bucket containing the corresponding attribute value.

To reduce shuffling cost, we customize the combiner function to aggregate key-value pairs within the same bucket so that each mapper only generates at most one key-value pair for a bucket. In the reduce phase, we group key-value pairs by their $columnID$ and combine the results from multiple mappers. Finally, the metadata of a histogram bucket, including table name, column name, bucket range and bucket value, are written back to HDFS. To efficiently locate a histogram, histograms are maintained as a directory tree in HDFS, e.g., the histogram for column a_i of table T is stored in "$/histogram/T/a_i$".

5.2.2 *Selection of Optimal Data Access Plan*

After having constructed the histograms for selectivity estimation of range predicates, we proceed to design an algorithm for selection of optimal data access plan.

The base tables are comprised of equal-size (s_d) data chunks in the underlying distributed file system (e.g., HDFS). Consider a query Q, we use the function $f(Q)$ to denote the size of data involved in the processing of that query. For the full table scan, if the query Q involves multiple tables T_1,...,T_k, then $f(Q)$ is computed as $\sum_{i=1}^{k} |T_i| s_i$, where s_i denotes the average size of records in the table T_i. For the index scan approach, $f(Q)$ is estimated as $\sum_{i=1}^{k} g(T_i, Q) s_i$. $g(T_i, Q)$ denotes the number of tuples in the table T_i that satisfy the selection predicates of the query Q. In the following discussion, we estimate the cost of *map* phase for processing a table T_i, as index is mainly used by the *mappers* to reduce the I/O cost.

Full scan. The total number of data chunks in the base tables referred in the query is $\frac{|T_i| s_i}{s_d}$ We need the same number of *mappers* in our processing. The underlying distributed file system (HDFS) ensures that the data chunks are roughly distributed across machines in the cluster. Suppose we have N cluster nodes. Let c_s be the cost ratio of sequential scan. The cost of the slowest node is:

$$c_{pscan} = \lceil \frac{|T_i| s_i}{s_d N} \rceil \times s_d c_s = \lceil \frac{|T_i| s_i}{N} \rceil c_s \quad (1)$$

Index scan. If the query can be processed by the primary index of T_i, we can effectively reduce the number of data chunks in the MapReduce job. The scan cost of the slowest node is reduced to

$$c_{iscan} = \lceil \frac{|T_i| s_i g(T_i, Q)}{N} \rceil c_s \quad (2)$$

If the query only involves the secondary index, `ScalaGiST` groups the pointers that refer to the same data chunk and performs random accesses to the base records in sequential offsets. Let c_r denote the cost ratio of random read with sequential offsets. The cost of slowest node is

$$c_{iscan} = \lceil \frac{|T_i| s_i g(T_i, Q)}{N} \rceil c_r \quad (3)$$

For a query involving k tables, we normally generate $k - 1$ MapReduce jobs to perform the join. If we stick to the left-deep plan, except the first job, the rest jobs join a raw table with an intermediate result table. For an intermediate result table T_i, we consider it as a table without indexes (namely, $g(T_i, Q) = |T_i|$). Given two table T_i and T_j, the scan cost of slowest node in the *map* phase is:

$$\lceil \frac{|T_i| s_i + |T_j| s_j}{N} \rceil c_s$$

And the cost of primary index scan is:

$$c_{iscan} = \lceil \frac{|T_i| s_i g(T_i, Q) + |T_j| s_j g(T_j, Q)}{N} \rceil c_s \quad (4)$$

Similarly, the cost of secondary index scan can be estimated.

Another cost is the index lookup cost. As most internal tree nodes are buffered in memory, our model only computes the network communication cost and the scan cost of leaf nodes. Let L be the number of index workers in the longest search path of the index. So the maximal network cost is $L c_n$, where c_n is the network cost ratio. If the size of each leaf index entry is e, we can maintain approximately $\frac{s_d}{e}$ leaf nodes in one index chunk. Namely, each index worker can handle about $\frac{s_d}{e}$ leaf nodes. Suppose we have W index workers, the index search cost is estimated as:

$$c_{lookup} = Lc_n + \lceil \frac{g(T_i, Q)e^2}{W} \rceil c_s \quad (5)$$

The second term in above equation denotes the average cost of each index worker when processing the leaf node scan. For two table join, we need to add up the index search costs of both tables.

Data access optimizer. After estimating the cost of the two data accessing schemes (scan and index-based processing), we now present a data access optimization algorithm as a guiding principle for the system to dynamically choose the optimal data access plan for the execution of a specific query with MapReduce. Given a query, we split it into multiple MapReduce jobs $\{j_0, j_1, ..., j_k\}$. For each job j_i, we estimate the cost of c_{pscan} and $c_{iscan} + c_{lookup}$ and select the optimal strategy. At regular time, the system runs a background a micro-benchmark on the underlying distributed file system to measure the performance of raw random and sequential I/Os and update the values of c_s, c_r and c_n respectively.

6. PERFORMANCE EVALUATION

We have performed a series of experiments to evaluate the efficiency and scalability of ScalaGiST. First, we evaluate the performance of ScalaGiST using the YCSB benchmark [9]. Then, we compare the performance of ScalaGiST-integrated MapReduce with generic MapReduce in processing analytical queries. We also study the performance of ScalaGiST in terms of analytic query and multi-dimensional query, and compare its performance with other distributed indexing frameworks, namely Data Mapping [6], SpatialHadoop [14], and RT-CAN [26]. To show the flexibility of ScalaGiST, we implement a new index MTree [8] on top of ScalaGiST and evaluate its performance on processing multi-dimensional queries. Lastly, we show the effectiveness of ScalaGiST in an application scenario involving multiple indexes in a single query.

6.1 Experimental Setup

The experiments are conducted on an in-house cluster, which includes 64 commodity machines equipped with Intel X3430 2.4 GHz processors, 8 GB of memory, two 7200 RPM SATA disks with 500 GB capacity each, and 1 Gb ethernet. The machines in the cluster are connected via a flat network.

A Hadoop cluster is set up as the infrastructure system for index storage and query processing with ScalaGiST. We keep the settings of Hadoop as default. Each machine in the cluster runs three daemon processes and plays multiple roles as a data node for HDFS, a worker node for MapReduce, and an index worker for ScalaGiST. The index master process is configured to run on the same machine as Hadoop cluster's master node.

6.2 Micro-benchmarks

In this test, we study the performance of index construction and index lookup operations with ScalaGiST using the YCSB [9] benchmark. We build a B^+-tree index using ScalaGiST. We generate the following two workloads.

- *Insert.* New records (key-value pairs) are randomly generated and inserted into the system. Note that when working with MapReduce system, ScalaGiST only supports batch insertion, but for the indexing service itself, ScalaGiST can support realtime insertion. The master forwards the updates to the corresponding index workers who update their local sub-trees. This experiment shows the raw performance of ScalaGiST and also indicates the cost of batch insertion using MapReduce.
- *Lookup.* The previously inserted records are searched, with the keys are randomly chosen.

For each index server, we configure the YCSB runtime to instantiate 4 client threads to concurrently access the ScalaGiST tree. That is, client workloads submitted to the system scale up much faster than the system size and index servers will observe heavy loads when the system size is large. The aggregate throughput of the two workloads are measured as the system scales out from 10 to 60 index servers.

We also run the workload on a centralized system with a standalone B^+-tree implementation to show the advantages of scalable distributed search trees. The B^+-tree is deployed on a Dell PowerEdge R610 server (which has a much higher hardware configuration compared to other commodity machines in the cluster), and is configured to have a cache of 4 GB. The number of concurrent client threads submitting workloads to the standalone B^+-tree is configured to be equivalent with the above setting of ScalaGiST tree. Before running the insert workload, both the ScalaGiST tree and the standalone B^+-tree are pre-populated with 10,000 records.

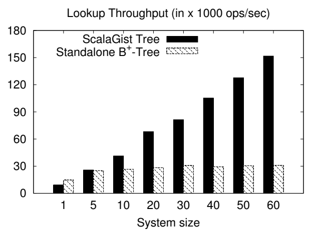


Figure 6: Aggregate throughput, lookup workload.

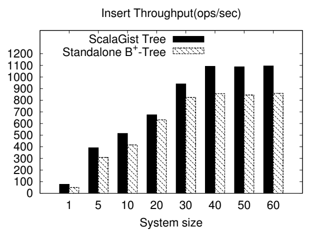


Figure 7: Aggregate throughput, insert workload.

As shown in Figure 6, the ScalaGiST tree scales almost linearly with the system size. On the contrary, the standalone B^+-tree achieves a much lower throughput and its capacity is saturated quite soon due to the lack of the ability to handle concurrent requests. The high throughput performance of ScalaGiST is attributed to its scalable architectural design. In addition, with the help of caching internal index nodes, most of the tree traversals along the read path can be finished within one network hop before reaching the appropriate index server to retrieve the desired index leaf page.

Figure 7 plots the system throughput for the insert workload. The aggregate throughput grows almost linearly with system sizes at low and medium scales (up to 40 machines). As the workload gets heavier (by increasing the number of machines and hence the number of client threads also scales up four times as much as described in the experiment settings), new insertions incur more network communication overhead and I/O contentions.

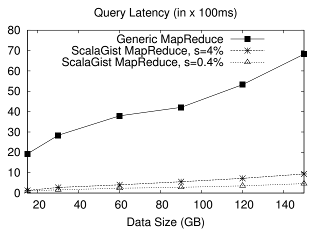
Overall, the above experimental results confirm the elastic performance of raw ScalaGiST framework. More index servers can be added into the system to serve the increasing workloads.

6.3 MapReduce Scan vs. Index Scan

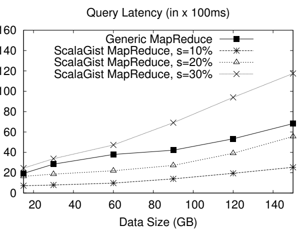
In this section, we compare the performance of ScalaGiST-integrated MapReduce with generic MapReduce in processing analytical queries. We conduct the experiment on TPC-H benchmark dataset [1] which models the workload of a decision support system.

```
Q1:SELECT custkey, count(orderkey)
   FROM Orders
   WHERE totalprice ≥ y and totalprice ≤ y + 100
   GROUP BY (custkey)
```

We consider a selective query above on the `Orders` table. For the data, we vary the scale factor of `Orders` table from 10 to 100. Under each scale factor, the workload generator produces 1.5 million records for the table. Each record has an average size of 1 KB. Thus, the total data size ranges from 15 GB to 150 GB. The data records are stored in the underlying HDFS and sorted by the selection key, i.e., the $totalprice$ attribute. A 20-machine cluster is set up for this experiment. The `ScalaGiST` index built on the data set is configured to instantiate a scalable B$^+$-tree-like index on the $totalprice$ column.



(a) High selectivity



(b) Low selectivity

Figure 8: MapReduce Scan vs. Index Scan.

By setting the value of y in the query predicate, we can define the selectivity of the query (denoted as s in Figure 8). Two sets of experiments were conducted to evaluate the query processing time of generic MapReduce and `ScalaGiST`-integrated MapReduce under different selectivity settings.

In the first experiment, we study performance characteristics of the systems when executing highly selective queries – the selectivity is set to 0.4% and 4%. The results plotted in Figure 8(a) confirm the effectiveness of `ScalaGiST` in supporting query processing over large scale data. Particularly, `ScalaGiST` helps to achieve a better performance by first querying the distributed index to identify the qualified data, then launching map tasks only on the data chunks hosting the target data.

Based on the insights of our cost model, `ScalaGiST`-integrated MapReduce underperforms in the case when index search cannot prune out enough data chunk. Hence, in the second experiment, we test the two approaches with non-selective queries to see the crossover point. As depicted in Figure 8(b), a selectivity of 30% is low enough for `ScalaGiST`-integrated MapReduce to perform worse than the generic MapReduce. Low query selectivity results in a larger result set that may span across more data chunks, and hence more map tasks have to be launched.

6.4 Multi-Dimensional Index Performance

In this section, we demonstrate the effectiveness of `ScalaGiST` in terms of its support for multi-dimensional data. We first compare the performances of `ScalaGiST` and three systems (namely, Data Mapping [6], SpatialHadoop [14] and RT-CAN [26]) on a 2-dimensional dataset. We then evaluate the performance of `ScalaGiST`'s M-tree implementation in higher dimensional (up to 10) settings.

For the first set of experiments, we construct a 2-dimensional table T with schema $T(a_1, a_2, p)$ where each attribute a_i uniformly generated from the domain of 10^9 integer values, and attribute p is a payload of 1 KB string data. The table is populated with 10 million to 100 million records, thus the size of the table varies from 10 GB to 100 GB. R-Tree indexes are built on (a_1, a_2) pair using the three systems, respectively. The number of index servers in the system is fixed to be 20.

We run both range queries and k-NN queries to evaluate the systems' performance. Specifically, range queries are run on (a_1, a_2) against the indexes with the following template:

```
Q2:SELECT p FROM T
   WHERE a1_l ≤ a1 ≤ a1_u and a2_l ≤ a2 ≤ a2_u
```

We define the selectivity as the percentage of searched space. By adjusting the lower bounds and upper bounds for both a_1 and a_2, we are able to control the query selectivity, which is set to 0.4% in this experiment. K-NN queries are processed via a set of range queries. For k-NN queries, k is set to be 16 in the experiments. The results are presented in the following figures.

6.4.1 *Generalized Search Tree vs. Data Mapping*

For multi-dimensional domains, data mapping approach partitions the original space into sub-spaces by different dimension iteratively, then links the partitions with adjacent identifiers to form the Z-ordering [21], which is a 1-dimensional representation of the original multi-dimensional domain, and thus range query in higher dimensional spaces could be transformed into querying intervals along the Z-ordering.

As can be seen in Figure 9, `ScalaGiST` gains a better performance over the 'Data Mapping' approach. As the data size increases, the latency of queries with 'Data Mapping' grows proportionally, while remaining stable with only a slight increment for `ScalaGiST`. This is because mapping multi-dimensional data into one single dimensional space results in information loss. It incurs many false positives in during the index lookup process.

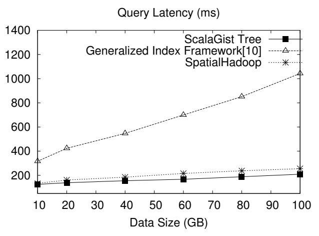


Figure 9: Range Query Performance.

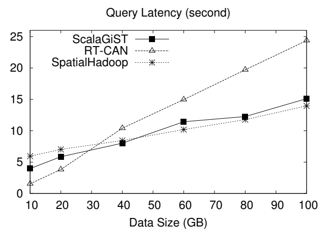


Figure 10: k-NN Query Performance.

6.4.2 *Generalized Search Tree vs. SpatialHadoop*

SpatialHadoop extents Hadoop to support spatial index operations. It organizes the spatial index in a layered structure, namely global partition index and local indexes. SpatialHadoop provides a layer of abstraction upon MapReduce by implementing its own multi-dimensional index operators, such as range operator and k-NN operator in order to facilitate multi-dimensional queries. In comparison, `ScalaGiST` adopts a different index processing mechanism in which the index operations are performed by index workers, and DFS data requests are handled by MapReduce(e.g. Hadoop) runtime.

Figure 9 and 10 compare the performance of `ScalaGiST` and SpatialHadoop in terms of range queries and k-NN queries. From the results, we observe a close performance for the two systems. In both systems, index operations are mostly done in memory, while the local index (in SpatialHadoop) or index workers (in `ScalaGiST`) take care of DFS I/Os. The slight difference comes from different implementation of index operators, in particular, the different instantiation cost for the two systems.

However, it is a promising result for `ScalaGiST` in that the performance of its generalized framework is comparable to that of SpatialHadoop's specially built and tuned index.

6.4.3 *Generalized Search Tree vs. RT-CAN*

RT-CAN is a multi-dimensional indexing framework for cloud environments. RT-CAN organizes the servers into an overlay based on an extended CAN routing protocol, and utilizes R-tree based index scheme at each server to support multi-dimensional query.

In the experiments, data are pre-partitioned into 5,000 grids and disseminated to the servers of RT-CAN. A local R-tree is built for the grids at each server with a page size of 4 KB.

The results plotted in Figure 10 confirm the extensibility of `ScalaGiST` to support complex multi-dimensional query. In particular, the overall performance of `ScalaGiST` is better than RT-CAN in terms of supporting k-NN query. Even though `ScalaGiST` incurs higher latency for k-NN queries at small data size, we observe a better performance of `ScalaGiST` as the data size increases. When data size is small, the start-up time for MapReduce tasks has more significant impact on the query efficiency with `ScalaGiST`, while RT-CAN does not suffer such overhead due to its different (peer-to-peer) processing model. However, at lager scales, the iterative overlay lookup and local R-tree search yield relatively high I/Os and computational cost.

6.4.4 *Scalagist in Multi-dimensional Metric Space*

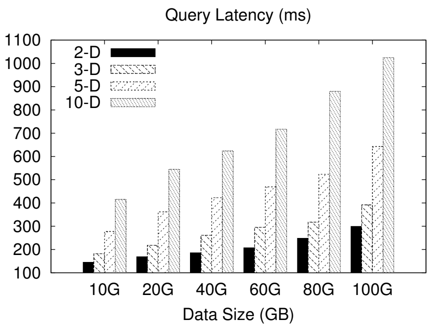


Figure 11: Effect of Dimensionality.

Using `ScalaGiST`, we can build new distributed indexes with ease by overriding the interface functions. In this experiment, we demonstrate the `ScalaGiST`'s M-tree implementation which is employed to index multi-dimensional data in a metric space. We use synthetic Random-Cluster (R-Cluster) data sets to evaluate the performance of `ScalaGiST`on varied dimensionalities (up to 10). The R-Cluster data sets consist of records with a tuple ID and d-dimensional coordinates. The IDs are 4-byte integers and the coordinates are 4-byte floating-point types. Distance between two records can be calculated using any user defined metric distance function. In this experiment, we adopt the L_∞ metric, i.e. $L_\infty(O_x, O_y) = max_{j=1}^{Dim}\{|O_x[j] - O_y[j]|\}$. System settings remain the same as in the previous experiments. A range query with selectivity of 0.4% in the d-dimensional space is run on the indexed data. Under each scale, we report the effect of varying dimensionality from 2 to 10 in Figure 11.

As depicted in Figure 11, with the increase of dimensionality and data size, the average time of running a range query also increases. This trend coincides with the typical performance of M-tree in the stand-alone setting. In addition, we are able to observe a good scalability both in terms of dimensionality and data size. The time-to-dimensionality and time-to-size pairs both scale nearly linearly. These results verify the functionality of `ScalaGiST` in supporting multi-dimensional data.

6.5 Multiple Indexes Performance

As mentioned in Section 4.3, one of the most distinguished features of `ScalaGiST` is its capability of supporting various types of indexes. In this experiment, we demonstrate this merit by incorporating multiple indexes in a single query. As most of the real data are business sensitive and are not publicly available, we synthetically construct our data to have multiple dimensional characteristics. Our purpose is to use this simple but straightforward example to exhibit how `ScalaGiST` benefits query using multiple indexes.

The schema $T\{orders, (a_1, a_2)\}$ is composed of the `Orders` table from TPC-H dataset, and the two-dimensional attribute (a_1, a_2) we generate in the last experiment. The table is sorted by *totalprice* column in `Orders` table. Given its characteristic, a B$^+$-tree can be built on the *totalprice* column, and the 2-dimensional column (a_1, a_2) can be indexed by an R-tree.

Note that when building multiple indexes on one table, factors such as the clustering of data, the choice of primary index, etc., would all have substantial influence on the performance. `ScalaGiST` is designed to provide flexible functionality and APIs, and leaves other decisions to the user.

The query used in this experiment is a simple extension of Q1 with a range search on the 2-dimensional column:

```
Q3:SELECT custkey, count(orderkey)
   FROM Orders
   WHERE totalprice $\geq y$ and totalprice $\leq y + 100$
   and $a_1\_l \leq a_1 \leq a_1\_u$ and $a_2\_l \leq a_2 \leq a_2\_u$
```

Q3 is evaluated in three execution modes. The first is `ScalaGiST` multiple indexes mode, who has both B$^+$-tree and R-tree built on the two columns respectively. The second mode only builds R-tree on the 2-dimensional column. And in the third mode we use SpatialHadoop. The size of data varies approximately from 15G to 150G (`Orders` table plus additional column). The results are plotted in Figure 12.

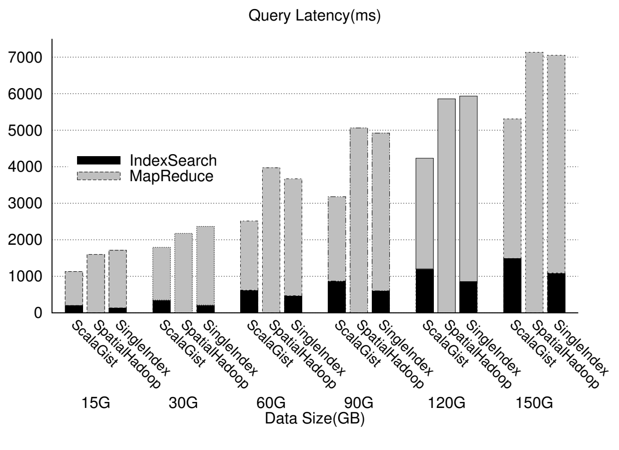


Figure 12: Multiple Index Performance.

To process Q3, two columns of the table are touched. The indexed column is searched via index workers first. With the knowledge of index search result, MapReduce jobs are launched on the chunks hosting the interested data. For the column without index, a MapReduce scan must be launched. Specifically, in the R-tree only case, index search on R-tree returns a super set of the accurate selection result. Then a "partial" MapReduce job scans through the chunks included in the search result to test against the *totalprice* column before generating the final results. For SpatialHadoop, two set of MapReduce jobs are launched. The first MapReduce job runs range search using original SpatialHadoop function. The second set of MapReduce jobs are used to scan the whole table for selective condition, and merge the scan result with result of range search.

In Figure 12, the processing time is broken down to highlight the index search phase and the MapReduce phase. As shown, the overall execution time of `ScalaGiST` is significantly reduced comparing to those of SpatialHadoop and single index. `ScalaGiST` has longer index processing time, since the runtime need to wait until all index workers complete the search and merge the results. However, with the benefit of more accurate index search result, the subsequent MapReduce job in `ScalaGiST` is able to avoid launching redundant mappers, and enjoys better performance.

7. CONCLUSION

In this paper, we have presented `ScalaGiST` – scalable generalized search tree – which provides the much desired extensibility in terms of data and query type. It supports multiple types of indexes, and can be dynamically deployed on large clusters while resilient to machine failures. We have implemented `ScalaGiST` and demonstrated that it can be easily instantiated as scalable B^+-tree and R-tree like indexes for dynamic cluster environments. More importantly, its seamless integration with Hadoop platform, coupled with a cost-based data access optimizer, provide promising opportunities for significant performance improvement on query processing in MapReduce-based systems. Our experiments on `ScalaGiST`'s performance with respect to multiple types of indexes confirmed the effectiveness and efficiency of our proposed indexing mechanism.

Acknowledgment

This work was supported by the National Research Foundation, Prime Minister's Office, Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). We would also like to thank anonymous reviewers for insightful comments.

8. REFERENCES

[1] TPC-H benchmark. [Online] http://www.tpc.org/tpch.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[3] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *PVLDB*, 1(1):598–609, 2008.

[4] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, 2011.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[6] G. Chen, H. T. Vo, S. Wu, B. C. Ooi, and M. T. Özsu. A framework for supporting dbms-like indexes in the cloud. *PVLDB*, 4(11):702–713, 2011.

[7] R. Choubey, L. Chen, and E. A. Rundensteiner. Gbi: A generalized r-tree bulk-insertion strategy. In *Advances in Spatial Databases*, pages 91–108. Springer, 1999.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of VLDB*, pages 426–435, 1997.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of SoCC*, pages 143–154, 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of SOSP*, pages 205–220, 2007.

[12] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1-2):515–529, 2010.

[13] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[14] A. Eldawy and M. F. Mokbel. A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. *PVLDB*, 6(12):1230–1233, 2013.

[15] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrak. Eagle-eyed elephant: Split-oriented indexing in hadoop. In *Proc. of EDBT*, pages 89–100, 2013.

[16] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of VLDB*, pages 562–573, 1995.

[17] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *PVLDB*, 3(1-2), 2010.

[18] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[19] F. Li, B. C. Ooi, M. Ozsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Survey*, 2014.

[20] H. Liao, J. Han, and J. Fang. Multi-dimensional index on hadoop distributed file system. In *Proc. of NAS*, pages 240–249, 2010.

[21] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ottawa, Canada, 1966.

[22] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Proc. of MDM*, 2011.

[23] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in r-trees. In *ICDT*, 1997.

[24] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of SIGMOD*, pages 165–178, 2009.

[25] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. on Knowl. and Data Eng.*, 16(10):1169–1184, Oct. 2004.

[26] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *Proc. of SIGMOD*, pages 591–602, 2010.

[27] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *PVLDB*, 3(1):1207–1218, 2010.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[29] H. Zhao, S. Yang, Z. Chen, S. Jin, H. Yin, and L. Li. Mapreduce model-based optimization of range queries. In *Proc. of FSKD*, pages 2478–2492, 2012.