PAPER Special Section on Data Engineering and Information Management

The Efficient Algorithms for Constructing Enhanced Quadtrees Using MapReduce*

Hongyeon KIM[†a)], **Sungmin KANG**[†b)], **Seokjoo LEE**[†c)], ***Nonmembers***, ***and*** **Jun-Ki MIN**[†d)], ***Member***

SUMMARY MapReduce is considered as the *de facto* framework for storing and processing massive data due to its fascinating features: simplicity, flexibility, fault tolerance and scalability. However, since the MapReduce framework does not provide an efficient access method to data (i.e., an index), whole data should be retrieved even though a user wants to access a small portion of data. Thus, in this paper, we devise an efficient algorithm constructing quadtrees with MapReduce. Our proposed algorithms reduce the index construction time by utilizing a sampling technique to partition a data set. To improve the query performance, we extend the quadtree construction algorithm in which the adjacent nodes of a quadtree are integrated when the number of points located in the nodes is less than the predefined threshold. Furthermore, we present an effective algorithm for incremental update. Our experimental results show the efficiency of our proposed algorithms in diverse environments.

key words: *index, quadtree, range query, MapReduce*

1. Introduction

MapReduce proposed by Google is becoming the *de facto* framework for storing and processing massive data due to its fascinating properties: simplicity, flexibility, fault tolerance and scalability [7]. Efficient data processing in MapReduce has received a lot of attention since its debut. MapReduce is a distributed parallel processing model and execution environment that allows easy development of scalable parallel applications to process big data on large clusters of commodity machines. Google's MapReduce or its open-source equivalent Hadoop [2] is a powerful tool for building such applications. Of particular, the development of variants of Hadoop has been active, e.g., Hadoop++ [9], MapReduce Online [6], Pig [15], Hive [18] and others. Much work has been conducted also to make efficient parallel algorithms running on the MapReduce framework [11], [16], [23].

The commonly argued issue of the above work is that the MapReduce framework does not provided efficient data access methods such as spatial indexes. Thus, even though a user want to retrieve a subset of data, the whole data should be accessed in the MapReduce framework. To alleviate this problem, we propose a parallel algorithm constructing quadtrees running on the MapReduce framework. By utilizing the constructed quadtree, we can rapidly reduce the search space using and, in nature, apply the MapReduce framework to process data in parallel. Thus, we can enhance the performance of big data processing. In our work, we utilize a sampling technique to split a data set into partitions in order to reduce the quadtree construction time.

Each set of points belonging to each leaf node of the constructed quadtree is stored into a separate file. Many files tend to contain only a few points due to the skewness of data distribution. Thus, when we retrieve some points using a range query, the query performance may be degraded due to a large number of files overlapped with the query range. To improve the query performance, we extend the quadtree construction algorithm in which the adjacent leaf nodes of a quadtree are consolidated when the number of points located in the nodes is less than the predefined threshold.

Organization. The remainder of the paper is organized as follows. In Sect. 2, we present the background of our work. Section 3 discusses related work. Section 4 presents our proposed algorithms for quadtree construction on the MapReduce framework. Section 5 presents an empirical evaluation. Finally, Sect. 6 summarizes our work.

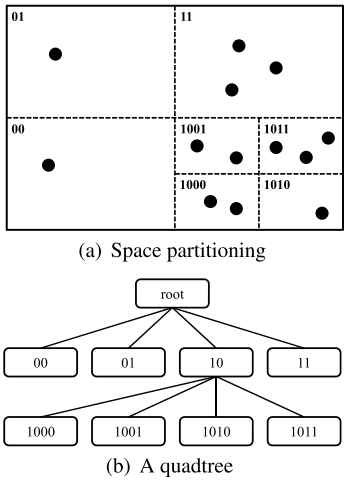
2. Preliminary

2.1 Quadtrees

For databases, several index structures such as B+-tree [5] and R*-tree [3] have been proposed. As one of the prominent index structures for multi-dimensional data, a *quadtree* [10] is used in diverse areas such as spatial data management and image processing since it is conceptually simple and easy to maintain.

Given a set D of d-dimensional points, the quadtree [10] subdivides the d-dimensional space recursively into subregions. In the quadtree, each internal node has exactly 2^d children and each leaf node has at most a predefined number of points c, denoted as *capacity*. Every node n in the quadtree represents a d-dimensional region, denoted as $n.region = \langle [n(1)^-, n(1)^+), \ldots, [n(d)^-, n(d)^+) \rangle$ where $[n(k)^-, n(k)^+)$ is the range of the k-th dimension of n's covering region. When a quadtree is constructed, if the number of points located in $n.region$ is greater than the capacity c, $n.region$ is divided into equi-sized 2^d subspaces each of which is associated with n's child node. Thus, every intermediate node of the quadtree has 2^d child nodes and every leaf node n has a subset of points which are located in the

Manuscript received June 10, 2015.
Manuscript publicized January 14, 2016.
†The authors are with Korea Univ. of Tech. & Edu., Korea.
*The preliminary version of this paper was published in the first International DASFAA workshop on BDMA 2013.
a) E-mail: zenweird@koreatech.ac.kr
b) E-mail: chaoslove@koreatech.ac.kr
c) E-mail: gcl8775@koreatech.ac.kr
d) E-mail: jkmin@koreatech.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2015DAP0005



(a) Space partitioning

(b) A quadtree

Fig. 1 An example of a quadtree

region presented by n. Figure 1 shows a quadtree for a 2-dimensional data set with the capacity $c = 3$.

As shown in Fig. 1, an id is assigned to each node n of the quadtree based on its location. We represent the node with id i as $node(i)$. In a d-dimensional space, the id of a node n with depth e is represented by $id(n) = a_1 a_2 \cdots a_{e \cdot d}$ which consists of the first $(e-1) \cdot d$ bits coming from its parent node and the remaining d bits $a_{(e-1) \cdot d+1} a_{(e-1) \cdot d+2} \cdots a_{e \cdot d}$ where $a_{(e-1) \cdot d+i} = 0$ (or $a_{(e-1) \cdot d+i} = 1$) if the i-th dimensional range of the region represented by the node is the first half (or the second half) of its parent's i-th dimensional range. For instance, in Fig. 1 the first child node of the node with the id '10' has '1000' as its id. In the id '1000', the first two digits '10' come from its parent node. Since the region corresponding to the node with the id '1000' locates in the first half part on each dimension, the remained 2 digits are '00'.

2.2 MapReduce

Inspired by the map and reduce primitives present in functional languages, Google developed the MapReduce [7] framework that enables the users to easily develop large scale distributed applications. *MapReduce* is a distributed as well as parallel processing model and execution environment in the shared-nothing clusters of commodity machines. Hadoop [2] is implemented in the OpenSource community as the MapReduce framework. In Hadoop, using the Hadoop Distributed File System (HDFS), a large sized file is initially partitioned into several fragments, called *chunk*, and stored in several machines redundantly for reliability. The size of a chunk is typically 64 MByte.

In MapReduce, a program consists of a map function and a reduce function which are user-defined functions. The associated implementation parallelizes large computations easily as each map function invocation is independent and uses re-execution as the primary mechanism of fault tolerance. Basically, the MapReduce framework consists of one *job tracker* and several *task trackers*. Each task tracker is

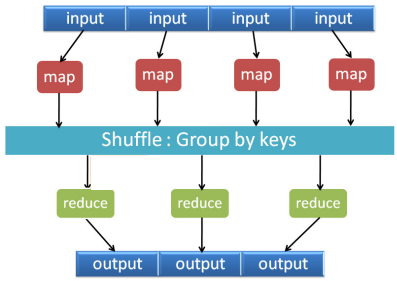


Fig. 2 The data flow in MapReduce

running on a commodity machine, called *slave*. A slave processes data using a map function and a reduce function each of which is invoked by the task tracker. The job tracker running on a single machine, called *master*, takes responsibility for error detection, load balancing and so on.

Conceptually, the map and reduce functions implemented by the user have the following types:

$$map(key_1, value_1) \rightarrow list(key_2, value_2)$$
$$reduce(key_2, list(value_2)) \rightarrow (key_3, list(values_3))$$

In the MapReduce framework, the computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The data processing in MapReduce is composed of three phases: *map phase*, *shuffle phase* and *reduce phase*. Figure 2 illustrates the data flow in the MapReduce framework.

Map phase: A map function takes a key-value pair $(key_1, value_1)$ as input, executes some computation and may output a set of intermediate key-value pairs $(key_2, value_2)$. In addition, by applying a combine function, additional computation to intermediate results can be executed.

Shuffle phase: In this phase, intermediate key-value pairs are grouped with respect to the key key_2. Thus, a reduce function to be executed in the next phase can obtain a list of values having the same key. Therefore, through this phase, each work is assigned the data lists as $(key_2, list(value_2))$.

Reduce phase: Each reduce function executes computation for the value list and emits a key-value list pair $(key_3, list(value_3))$ as a final result. The intermediate values are supplied to the reduce function via an iterator. This allows MapReduce to handle lists of values that are too large to fit in main memory.

3. Related Work

Recently, many techniques using the MapReduce framework have been proposed. In [22], the raster image sorting technique using the road information in satellite images was proposed. In [13], the parallel algorithm for constructing suffix array for DNA sequence analysis was proposed. In [12], for IR systems, four parallel algorithms to construct inverted index were proposed and evaluated their efficiencies on the MapReduce framework. In [1], k-Nearest

Neighbor (k-NN) and reverse nearest neighbor problems are solved using MapReduce based on the Voronoi partitioning. In [11], an efficient k-NN join technique using MapReduce was proposed. In [21], the query for ALL Nearest Neighbor (ANN) was processed on the MapReduce framework. Of particular, similar to our work, the data space is divided into several tiles for load balancing and the tiles are integrated according to Z-order in [21]. However, if this technique is applied to the quadtree construction, it is hard to consolidate the local quadtrees generated for tiles into the global quadtree.

In [4], R-tree was constructed using MapReduce. All objects are sorted with respect to a space filling curve such as Z-order and partitioned into the subsets. For each subset, a local R-tree is constructed in each machine. Then, the local R-trees are combined into a final R-tree. However, in this work, the maintenance of the constructed R-tree is not mentioned and the access method using the R-tree on the MapReduce framework is not introduced.

The most related work to ours is [17]. To perform the earthquake simulation, a quadtree for 3-dimensional points is constructed on the MapReduce framework. In [17], the quadtree is constructed in a bottom-up fashion. In the first MapReduce pahse, each point is transformed to a node of a quadtree and the nodes are merged into a partial quadtree. In the next MapReduce phase, the partial quadtrees generated in the previous phase are merged. This job is iteratively conducted until a single quadtree is generated. Therefore, since several MapReduce phases are required to construct the final quadtree, overall performance is degraded.

Recently, Park et al. [16] presented an efficient parallel algorithm for skyline processing on MapReduce. In this work, to prune out non-skyline points eagerly, a variant of the quadtrees is used. However, the performance can be degraded since the variant of the quadtrees is constructed using a sample data only on a single machine.

4. Quadtree Construction on MapReduce

In this section, we first present an efficient parallel quadtree construction algorithm proposed in [14] briefly and next propose an enhanced quadtree construction algorithm to improve the query performance.

4.1 Efficient Quadtree Construction

In [14], we proposed an efficient quadtree construction algorithm on the MapReduce framework, called *SQMR* (Sample based Quadtree with MapReduce), which utilizes a sampling technique in order to identify the data distribution approximately. We present some notations of quadtree used in our work.

Definition 1: Given d-dimensional point set D whose size is $|D|$, the *global quadtree* is the quadtree Q_c^D constructed with D where the capacity is c.

Definition 2: Given d-dimensional point set D, the *base quadtree* is the quadtree $Q^D_{|D|/t}$ constructed with D where the capacity is $|D|/t$ and t is the number of machines.

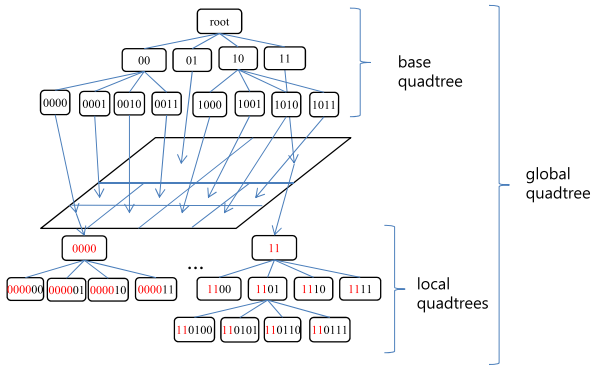


Fig. 3 The hierarchical structure of the global quadtree

Function SQMR(D, c, d, t, f)
D: a data set, S: a sample set, c: capacity, d: the dimension
t: the number of machines, f: sampling ratio
begin
//Base Quadtree Construction Step
1. S = ReservoirSampling(D, f)
2. $Q^S_{|S|/t}$ = ConQuadTree(S, $|S|/t$)
//Local Quadtree Construction Step on MapReduce
3. Broadcast $Q^S_{|S|/t}$ and c
4. $LocalTrees$ = RunMapReduce(LOCALQ.map, LOCALQ.reduce)
//Consolidation Step
5. $QuadTree$ = GLOBALQ($Q^S_{|S|/t}$, $LocalTrees$)
6. **return** $QuadTree$
end

Fig. 4 The algorithm of SQMR

Definition 3: Given a leaf node n of the base quadtree and the d-dimensional point set D, the *local quadtree* is the quadtree Q_c^P constructed with a set of points $P \subseteq D$ which are located in $n.region$ with the capacity c.

Figure 3 illustrates the hierarchical structure of the global quadtree which is composed of a base quadtree and a set of local quadtrees. The pseudocode of *SQMR* is presented in Fig. 4. SQMR is composed of three steps: *base quadtree construction step*, *local quadtree construction step* and *consolidation step*.

In the *base quadtree construction step*, to split d-dimensional data space into the subspaces, we build a base quadtree (lines 1-2 in Fig. 4). To improve the performance, we first generate a sample S from the data D using the reservoir sampling [20] (line 1 of SQMR in Fig. 4). In the reservoir sampling, instead of flipping a coin for each point $p \in D$ to add p into S, the number of skipped points before the next point to be added to S is determined with respect to the sampling ratio $f(= |S|/|D|)$. Thus, the reservoir sampling is adequate for a large sized data set since some points in D are skipped without retrieval.

Using a sample S, we construct an approximate base quadtree $Q^S_{|S|/t}$ where the capacity is $|S|/t$ (line 2 of SQMR). In the statistical view, when $|S| > 30$, by the central limit theorem, the expected sample mean is population mean and the expected sample variance is equal to the population variance divided by $|S|$-1 [8]. Thus, the variance of sample

points in each dimension is an unbiased estimator for the variance of D in each dimension. In addition, the $d \times d$ covariance matrix of S is also an unbiased estimator for the covariance matrix of D [19]. Thus, the shape of an approximate base quadtree will be similar to that of a base quadtree.

In the *local quadtree construction step*, the data set D is split into partitions based on the regions divided by an approximate base quadtree $Q^S_{|S|/t}$ and a local quadtree for each partition is computed in parallel on the MapReduce framework (lines 3-4 in Fig. 4). To do so, the approximate base quadtree $Q^S_{|S|/t}$ and the capacity c are broadcast to all machines. Then, each map function (i.e., LOCALQ.map) invoked with a point $p \in D$ finds the leaf node n of $Q^S_{|S|/t}$ where p is located in $n.region$ and emits the key-value pair (n, p). During the shuffle phase, the key-value pairs generated by all map function are sorted and grouped by key. The reduce function (i.e., LOCALQ.reduce) is called with each key n and the corresponding point list P. Then, the reduce function constructs the local quadtree Q^P_c with the capacity c and outputs the local quadtree Q^P_c with n (i.e, a leaf node of the approximate base quadtree $Q^S_{|S|/t}$).

Given a leaf node n_ℓ of a local quadtree Q^P_c, let $P(n_\ell)$ be the set of points located in $n_\ell.region$ (i.e., $P(n_\ell) = \{p \in D | p$ is located in $n_\ell.region\}$). The set $P(n_\ell)$ of every n_ℓ of a local quadtree Q^P_c is stored into a separated file and each leaf node only keeps the name of the corresponding file where the file name storing $P(n_\ell)$ is the id of n_ℓ (i.e., $id(n_\ell)$).

Finally, we construct the global quadtree using the local quadtrees for all partitions in the *consolidation step* (lines 5-6 in Fig. 4). Since each partition associates with each leaf node of an approximate base quadtree, by replacing the leaf node of an approximate base quadtree to the proper local quadtree's root node, a global quadtree is easily constructed.

4.2 Enhanced Quadtrees

To process a range query q retrieving a small portion of data with the MapReduce framework only, whole points should be retrieved. In contrast, if we utilize a quadtree, we can reduce the search space significantly.

Let $q.region$ be the query region of a query q. Then, by traversal of the constructed global quadtree Q^D_c, we can collect a set of leaf nodes whose regions are overlapped with $q.region$. We refer to such set of leaf nodes in Q^D_c with respect to $q.region$ as $N_q = \{n_\ell | n_\ell$ is a leaf node of Q^D_c and $n_\ell.region$ overlaps with $q.region\}$. Recall that, as mentioned in Sect. 4.1, each leaf node n_ℓ keeps the name (i.e., $id(n_\ell)$) of the file for the set of all points $P(n_\ell)$. Thus, we can generate the query result A_q of q in parallel by retrieving $P(n_\ell)$ independently where n_ℓ is in N_q such that, for each leaf node $n_\ell \in N_q$, if a point $p \in P(n_\ell)$ is in $q.region$, we put p into A_q.

Typically, in the MapReduce framework, the task tracker called *mapper* takes responsibility for invoking every map function with each key-value pair in each chunk. Thus, we set the capacity c 64Mbyte/sizeof(p) where

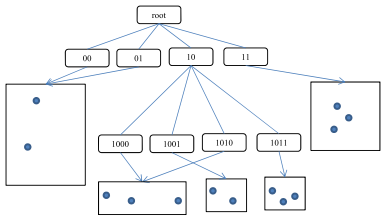


Fig. 5 An example of enhanced quadtrees

sizeof(p) returns the size of a point p so that the size of each file is *at most* 64Mbyte (i.e., chunk size), and hence, each file is processed by each distinct mapper. However, as the number of files whose corresponding leaf nodes' regions are overlapped with the query range increases, the query performance tends to decrease since the number of mappers increases. Thus, to improve the query performance, we devise an *enhanced quadtree* by merging the files. An example of an enhanced quadtree is shown in Fig. 5.

Note that when we construct a local quadtree in SQMR, the region in which the number of points exceeds the capacity c is split into 2^d subregions. Thus, the numbers of points in leaf nodes may not be evenly distributed as shown in Fig. 1. In other words, some leaf nodes could have very small number of points. Thus, by merging the files having small number of points into ones whose sizes are at most c, we can reduce the number of generated files as well as every file can be handled by each mapper. Furthermore, to improve the query performance, we merge the files whose corresponding leaf nodes are adjacent. Before defining the adjacency of nodes, we first address the containment relationship between a pair of ranges.

Definition 4: Given a pair of ranges $r = [r_{min}, r_{max})$ and $r' = [r'_{min}, r'_{max})$, if $r_{min} \le r'_{min}$ and $r'_{max} \le r_{max}$, we say that r contains r' which is denoted as $r' \subseteq r$.

Definition 5: Let i-th range of $n.region$ be $n.region(i) = [n(i)^-, n(i)^+)$. Given a pair of leaf nodes n, n' of a quadtree for d-dimensional data set, we say that n and n' are *adjacent* when $n.region$ and $n'.region$ satisfy the following two conditions:
(1) In a single i-th dimension ($1 \le i \le d$), $n(i)^+ = n'(i)^-$ or $n(i)^- = n'(i)+$
(2) In every other j-th dimension ($1 \le j \le d$ and $j \ne i$), $n.region(j) \subseteq n'.region(j)$ or $n'.region(j) \subseteq n.region(j)$

Informally, we say that a pair of leaf nodes n and n' are adjacent when $n.region$ and $n'.region$ meet in $d-1$ dimensional space. In order that a pair of regions R and R' meet in $d-1$ dimensions, the first condition of Definition 5 addresses that boundaries of R and R' should be shared in a single dimension and the second condition illustrates that the ranges of the other dimensions should have containment relationships.

The id of a leaf node n with depth e (i.e., $id(n) = a_1, \ldots, a_{ed}$) can be decomposed into d bit strings $sub_i(id(n))$ for $1 \le i \le d$ such that $sub_i(id(n)) = a_i a_{i+d} a_{i+2d} \ldots a_{i+(e-1)d}$. Then, by utilizing the ids of leaf nodes, we can check effi-

```
Function LOCALQ.map(key, p)
key: null, p: a point
begin
1. Q^S_{|S|/t} = LoadBaseQuadTree()
2. n = FindLeafNode(p, Q^S_{|S|/t})
3. emit(n, p)
end

Function LOCALQ.reduce( key, P )
key: a node, P: a list of point
begin
1. c = LoadCapacity()
2. Q^P_c = ConQuadTree(P, c)
3. Q^P_c = MergingEQMR(Q^P_c, c)
4. output(key, Q^P_c)
end

Procedure MergingEQMR( Q, c )
Q: local quadtree, c: capacity
begin
1. Leafnodes = LoadLeaves(Q);
2. for each leaf node n ∈ Leafnodes do {
3.     Leafnodes = Leafnodes - {n}
4.     Target = {n}, MergedSize = n.size
5.     AdjLeafnodes = LookupAdjLeafnode(n)
6.     for each leafnode n_adj ∈ AdjLeafnodes do {
7.         if MergedSize + n_adj.size < c then {
8.             Target = Target ∪ {n_adj}
9.             MergedSize += n_adj.size
10.            Leafnodes = Leafnodes - {n_adj}
11.            AdjLeafnodes = AdjLeafnodes ∪ LookupAdjLeafnode(n_adj)
12.        }
13.    }
14.    MergeFiles(Target)
15. }
16. return Q
end
```

Fig. 6 The algorithm for an enhanced quadtree construction

ciently whether a pair of leaf nodes n and n' are adjacent or not based on the following proposition.

Proposition 1: For a pair of leaf nodes n and n', consider a pair of $n.region(i) = [n(i)^-, n(i)^+)$ and $n'.region(i) = [n'(i)^-, n'(i)^+)$ as well as a pair of i-th substrings $sub_i(id(n)) = a_1 a_2 \ldots a_p$ and $sub_i(id(n')) = b_1 b_2 \ldots b_q$.
(1) When $n(i)^+ = n'(i)^-$, there exists an integer k with $1 \leq k \leq min(p, q)$ such that $a_j = b_j$ for $j < k$, $a_k = 0$, $b_k = 1$ and $a_{k+1} = \cdots = a_p = 1$ (if $k < p$) as well as $b_{k+1} = \cdots = b_q = 0$ (if $k < q$).
(2) When $n.region(i) \subseteq n'.region(i)$, $p \leq q$ and $a_k = b_k$ for all $k = 1, \ldots, p$.

For instance, as shown in Fig. 1, since $sub_1(00) = 0$ and $sub_1(1001) = 10$, the boundaries of $node(00).region$ and $node(1001).region$ are shared in x-coordination. In addition, since $sub_2(00) = 0$ and $sub_2(1001) = 01$ satisfy the condition (2) of Proposition 1, the range of $node(00).region$ contains that of $node(00).region$ in y-coordination. Therefore, $node(00)$ and $node(1001)$ are adjacent. However, $node(00)$ and $node(11)$ are not adjacent since the boundaries of $node(00).region$ and $node(11).region$ are shared in both coordinations.

To construct an enhanced quadtree, we present the enhanced quadtree construction algorithm, referred to as EQMR (Enhanced Quadtree with MapReduce) as shown in Fig. 6. Note that, in enhanced quadtrees, the quadtree structure itself is not changed. Instead, as shown in Fig. 5, leaf nodes of an enhanced quadtree point to an identical file. Thus, the procedure of EQMR is similar to that of SQML except the reduce phase of local quadtree construction.

As mentioned in Sect. 4, each map function invoked with a point p finds the leaf node n of $Q^S_{|S|/t}$ where p is in $n.region$ (line 2 of LOCALQ.map in Fig. 6) and emits (n, p) (line 3 of LOCALQ.map). After the shuffle phase, the reduce function LOCALQ.reduce with a node *key* and a list of points P is invoked. When the construction of local quadtrees Q^P_c (line 2 of LOCALQ.reduce in Fig. 6) is finished, the procedure MergingEQMR is invoked to merge the files corresponding to some leaf nodes (line 3 of LOCALQ.reduce).

In the procedure MergingEQMR, all leaf nodes of a local quadtree Q are collected into the set $Leafnodes$ (line 1 of MergingEQMR in Fig. 6).

For each leaf node n in $Leafnodes$, the leaf nodes which can be merged with n are computed and consolidated (lines 2-15). To avoid computing n redundantly, n is removed from $Leafnodes$ (line 3). A set $Target$ is used to keep the leaf nodes to be merged with n and $MergedSize$ keeps the sum of sizes of files whose corresponding nodes are in the set $Target$. Thus, $MergedSize$ is initially set to the file size (i.e., $n.size$) of the leaf node n (line 4). For each leaf node n in $Leafnodes$, we collect n's adjacent leaf nodes from $Leafnodes$ to a set $AdjLeafnodes$ by using Proposition 1 (line 5) and checks whether file of each adjacent leaf node n_{adj} in $AdjLeafnodes$ can be merged with n's file (lines 6-13). If the sum of $MergedSize$ and $n_{adj}.size$ is less than c, n_{adj} is inserted into $Target$ and $MergedSize$ increases by $n_{adj}.size$. Then, n_{adj} is removed from $Leafnode$ (lines 8-10). Since the file of n_{adj} is merged, the adjacent nodes of n_{adj} should be investigated. Thus, n_{adj}'s adjacent nodes are appended to $AdjLeafnodes$ (line 11). Finally, by invoking MergeFile(), all files which can be merged with n are consolidated into a single file and the leaf nodes in $Target$ record the name of the merged file (line 14). The name of the file generated by merging the files in $Target$ is assigned to the concatenation of the ids of the leaf nodes in $Target$ with a delimiter '|'. For instance, the name of the file pointed by $node(00)$ and $node(01)$ in Fig. 7 (c) is "00|01".

The following example illustrates the behavior of MergingEQMR.

Example 1: Let us assume that the quadtree in Fig. 1 be a local quadtree with the capacity $c = 3$. As shown in Fig. 7-(a), the procedure MergingEQMR collects the list $AdjLeafnodes$ of the adjacent leaf nodes for the first leaf node with the id '00' (i.e., $node(00)$). Firstly, since $node(01)$ can be merged with $node(00)$ among the adjacent leaf nodes, it is put into $Target$. In addition, since $node(11)$ is an adjacent node of $node(01)$, $node(11)$ is newly inserted into $AdjLeafnodes$ (Fig. 7-(b)). Now, if an adjacent node is merged additional, the size of the merged file exceeds c. Thus, the files of $node(00)$ and $node(01)$ are merged by invoking $MergingEQMR$ (Fig. 7-(c)). Next, for $node(1000)$, the adjacent nodes are collected and the nodes whose files

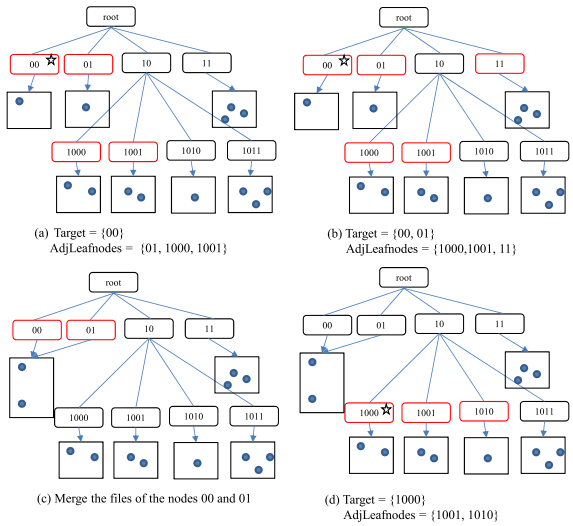


Fig. 7 The behavior of EQMR

can be merged with the file of *node*(1000) are selected. Consequently, the enhanced quadtree shown in Fig. 5 is generated.

4.3 Incremental Update

In this section, we present how to support incremental update to quadtrees. A naive algorithm to support incremental update is that a quadtree is newly constructed with whole data points whenever a new data point is inserted into the data set or a data point is removed from the data set. It results in the performance degradation. To alleviate this overhead, we use a temporary file T that keeps newly inserted points until the size of the temporary file is at most the capacity c. Thus, we do not need to construct a quadtree whenever a newly point occurs. Instead, when a query is posed, we retrieve the temporal file T as well as a quadtree simultaneously to compute the query result. Since the size of T is at most c, a single mapper can handle T to processing a query. Furthermore, when the size of T exceeds c, we do not reconstruct a quadtree. Instead, every point p in T is inserted into the file of the leaf node whose corresponding region contains p. Then, when the size of the file pointed by the leaf node exceeds c, the leaf node is split.

Recall that, in an enhanced quadtree, several leaf nodes may point to an identical file. As mentioned in Sect. 4.2, since the name of the merged file is composed of ids of the leaf nodes and delimiters '|', we can find the number of leaf nodes pointing to the file by performing tokenization to the file name. Thus, when the file size exceeds c by inserting a point p, if the number of the leaf nodes pointing to the file is larger than 1, the file is divided. Otherwise (i.e., the number of the leaf nodes is 1), the leaf node is split. We omit the pseudocode for the data insertion since it is straightforward.

When a point p is removed from the data set, we remove p from the leaf node n whose region contains p. If the size of the leaf node n becomes zero, we simply remove n from a quadtree. Since it is trivial, we also omit the details.

Table 1 Parameters

Parameter	Default value	Comments	Range
dist	skewed	Data distribution	(uniform, skewed)
$\|D\|$	5.0	# of points ($\times 10^8$)	1.0, 2.5, 5.0,7.5, 10.0
d	4	# of dimensions	2 ~ 6
c	64MByte	The capacity	64MByte

5. Performance Study

In this section, we verify the efficiency and effectiveness of EQMR by comparing it with the other algorithms.

5.1 Experimental Environments

To perform the experiment, we ran our implemented algorithms on a cluster of 31 commodity machines. One of machines acts as the master and the others act as slaves. The master has 3.1 GHz Intel Xeon E3-1220 CPU, 16GByte memory and 500GByte hard-disk. Each slave has 3.2 GHz Intel Core i5 CPU, 4GByte memory and 1TByte hard-disk. All machines are connected through a 1Gbps Ethernet switch. Every machine is running on Linux (Ubuntu 10.04 Lucid). We used Hadoop 2.0.0 for the MapReduce framework implementation obtained from [2].

We implemented three algorithms using JDK 1.6: EQMR (in Sect. 4.2), SQMR (in Sect. 4) and QMR. QMR partitions the data space into equi-sized subspaces without considering the data distribution.

To make diverse environments, we used some parameters, as summarized in Table 1. The domain of each dimension is [0.0, 1000.0]. We generated two types of synthetic data sets: uniform and skewed. To make the skewed data set, we used the normal distribution N(300, 50). We also varied the number of points from 10^8 to 10^9. The sizes for a set of 1.0×10^8 2-dimensional points, a set of 5.0×10^8 4-dimensional points and a set of 10.0×10^8 6-dimensional points are 1.68Gbyte, 15.83Gbyte and 46.56Gbyte, respectively. As presented in Sect. 4, we set the capacity (c) 64Mbyte. For SQMR and EQMR, we chose 10,000 points as a sample.

5.2 Performance Analysis

We ran each algorithm five times and report the average execution time of each algorithm in this section.

Data distribution: Fig. 8 (a) illustrates the performance of each algorithm with respect to the data distribution. The quadtree construction with the skewed data set takes more time compared to that with the uniform data set since the node partitioning occurs severely in the skewed data set. Since SQMR and EQMR identify the data distribution using sampling while QMR divides the data space into the equi-sized subspaces blindly, SQMR and EQMR show a better performance compared to QMR for the skewed data

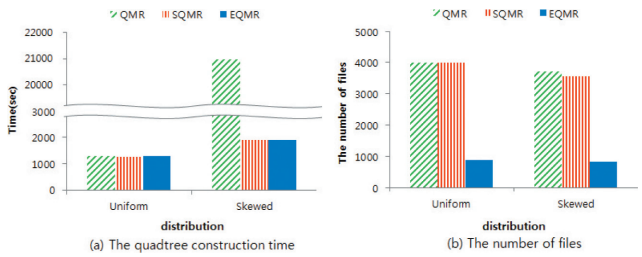


Fig. 8 The experimental result with varying data set

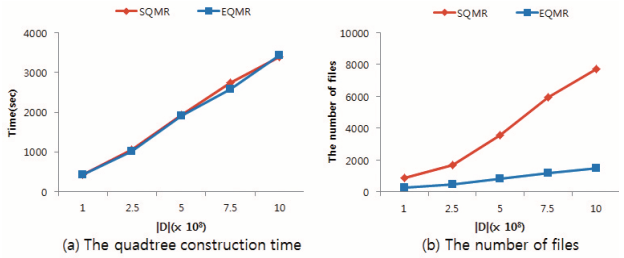


Fig. 9 The experimental result with varying $|D|$

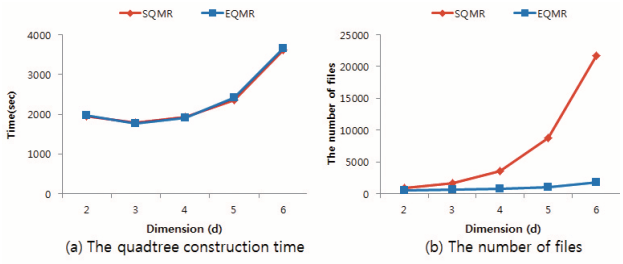


Fig. 10 The experimental result with varying dimensions

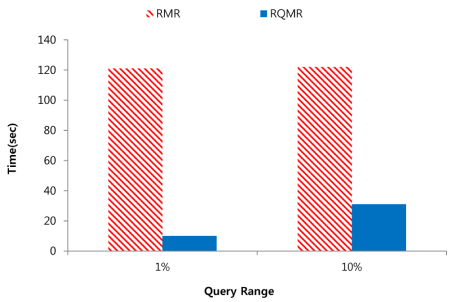


Fig. 11 The range query performance of *RMR* and *RQMR*

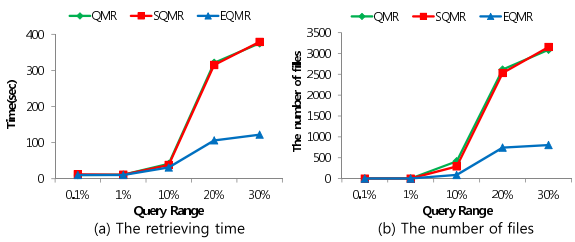


Fig. 12 The performances of *RQMR* utilizing QMR, SQMR, and EQMR

set. Furthermore, the performance of EQMR is similar to that of SQMR although EQMR has an addition procedure to merge files. This result indicates that the processing overhead of this procedure in EQMR is quite small compared to the entire quadtree construction time.

Figure 8 (b) shows the number of files associated with the leaf nodes of the constructed quadtree. As shown in Fig. 8 (b), the numbers of files are similar in the uniform and the skewed data set. The remarkable result is that the number of files in EQMR is quite smaller than those of QMR and SQMR since EQMR has the additional procedure to merge the files of leaf nodes which are adjacent to each other and the total size of them does not exceed the capacity.

Varying the number of points ($|D|$) and the number of dimensions (d): Fig. 9 (a) and Fig. 10 (a) show the quadtree construction times of SQMR and EQMR with varying the number of points and varying the number of dimensions, respectively, with the skewed data set. As shown in Fig. 8, since QMR shows the worst performance, we do not report the performance of QMR in these experiments.

As the number of points ($|D|$) as well as the number of dimensions (d) increase, the data size increases. Thus, the execution times of SQMR and EQMR increase as $|D|$ and d increase. However, as shown in Figs. 9 (a) and 10 (a), the quadtree construction time of SQMR is similar to that of EQMR like the other experiments although EQMR has an additional procedure for merging files.

Figure 9 (b) and Fig. 10 (b), show the numbers of files of the quadtrees constructed by SQMR and EQMR, respectively. As the number of points $|D|$ as well as the number of dimensions d increase, the number of files of SQMR increases rapidly since the region in which the number of points exceeds the capacity c is split into 2^d equi-sized subregions. Contrarily, in EQMR, since the files are merged, the number of files increases slowly.

Data Access: In this experiment, we show the performance of the range query processing using the quadtree with MapReduce, denoted as *RQMR*, compared with the range query processing using MapReduce without a quadtree, referred to *RMR*. The left lower corner of the query range is fixed at (200, 200, 200, 200) in 4-dimensional space.

For a range query, Figure 11 shows the performance of *RQMR* using an enhanced quadtree and *RMR*. Since *RMR* should retrieve whole points to process the range query, the performance of *RMR* does not affect the size of the query range as shown in Fig. 11. Meanwhile *RQMR* is better than *RMR* since *RQMR* reduces the search space efficiently using the quadtree.

Figure 12 presents the performance of *RQMR* utilizing the quadtrees generated by QMR, SQMR and EQMR, respectively, in the skewed data set. To make diverse range queries, we varied the size of the query range from 0.1% to 30% of the data space. As shown in Fig. 12 (a), as the size of query range increases, the query performance is degraded. The range query processing using an enhanced quadtree generated by EQMR shows the best performance

of the range query since the number of files to be retrieved is smaller than those of the other quadtrees as shown in Fig. 12 (b). However, the range query processing time using a quadtree generated by QMR and that using a quadtree by SQMR are similar to each other since the numbers of files to be retrieved of both quadtrees are similar as shown in Fig. 12 (b).

6. Conclusion

In this paper, we present an effective quadtree construction algorithm with MapReduce to improve the performance of data access. To evenly divide the data set and assign each partition to each machine, we build an approximate base quadtree using sample points. For each partition, we construct local quadtrees in parallel using MapReduce. Then, we build a global quadtree by integrating an approximate base quadtree and local quadtrees where each leaf node is associated with a data file containing the points.

In our experiment, we show the effectiveness to process range queries utilizing the constructed quadtree. To improve the performance of range queries, in *EQMR*, the data files correspond to each leaf nodes are merged into a single file when the merged file size does not exceed the threshold. By reducing the number of files, EQMR shows the best performance compared to the other algorithms. In addition, we present an effective algorithm supporting incremental update.

Acknowledgements

The research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2015R1D1A1A01058909).

References

[1] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Voronoi-based geospatial query processing with mapreduce," Proceedings of IEEE CloudCom, pp.9–6, 2010.
[2] Apache: Apache hadoop. http://hadoop.apache.org, 2010.
[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," Proceedings of ACM SIGMOD, vol.19, no.2, pp.322–331, 1990.
[4] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," Proceedings of SSDBM, vol.5566, pp.302–319, 2009.
[5] D. Comer, "ubiquitous b-tree," ACM Comput. Surv., vol.11, no.2, pp.121–137, 1979.
[6] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," Proceedings of NSDI, vol.10, p.21, 2010.
[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Communication of the ACM, vol.51, no.1, pp.107–113, 2008.
[8] J.L. Devore, Probability and statistics for engineering and the science, 4th ed., Duxbury Press, 1995.
[9] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," Proceedings of the VLDB Endowment, vol.3, no.1-2, pp.515–529, 2010.
[10] R.A. Finkel and J.L. Bentley, "Quad trees a data structure for retrieval on composite keys," Acta informatica, vol.4, no.1, pp.1–9, 1974.
[11] W. Lu, Y. Shen, S. Chen, and B.C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," Proceedings of VLDB, vol.5, no.10, pp.1016–1027, 2012.
[12] R. McCreadie, C. Macdonald, and I. Ounis, "Mapreduce indexing strategies: Studying scalability and efficiency," Information Processing & Maanagement, vol.48, no.5, pp.873–888, 2012.
[13] R.K. Menon, G.P. Bhat, and M.C. Schatz, "Rapid parallel genome indexing with mapreduce," Proceedings of MapReduce, pp.51–58, 2011.
[14] H. Noh and J.-K. Min, "An efficient data access method exploiting quadtrees on mapreduce frameworks," Proceedings of 1st international DASFAA workshop on BDMA, vol.7827, pp.86–100, 2013.
[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," Proceedings of ACM SIGMOD, pp.1099–1110, 2008.
[16] Y. Park, J.-K. Min, and K. Shim, "Parallel computation of skyline and reverse skyline queries using mapreduce," Proceedings of the VLDB Endowment vol.6, no.14, pp.2002–2013, 2013.
[17] S.W. Schlosser, M.P. Ryan, R. Taborda, J. Lopez, D.R. O'Hallaron, and J. Bielak, "Materialized community ground models for large-scale earthquake simulation," Proceedings of ACM/IEEE conference on Supercomputing, pp.1–12, 2008.
[18] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," Proceedings of the VLDB Endowment, vol.2, no.2, 1626–1629, 2009.
[19] R. Vershynin, "How close is the sample covariance matrix to the actual covariance matrix?," Journal of Theoretical Probability, vol.25, no.3, pp.655–686, 2012.
[20] J.S. Vitter, "Random sampling with a reservoir," ACM Transactions on Mathematical Software, vol.11, no.1, pp.37–57, 1985.
[21] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song, "Accelerating spatial data processing with mapreduce," Proceedings of IEEE ICPADS, pp.229–236, 2010.
[22] X. Wu, R. Carceroni, H. Fang, S. Zelinka, and A. Kirmse, "Automatic alignment of large-scale aerial rasters to road-maps," Proceedings of ACM GIS, pp.17:1–17:8, 2007.
[23] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using mapreduce," Proceedings of VLDB, vol.5, no.11, pp.1184–1195, 2012.

Hongyeon Kim was born in 1986. He is currently pursuing the Ph.D. degree in computer science and engineering from Korea University of Technology and Education, Republic of Korea. He received the M.S. degree from the School of Computer Science and Engineering, Korea University of Technology and Education, in 2013. His main research interests include indexing, data clustering, Big data, and MapReduce.

Sungmin Kang was born in 1990. He is currently pursuing the M.S. degree in computer science and engineering from Korea University of Technology and Education, Republic of Korea. He received the bachelor's degree from the School of Computer Science and Engineering, Korea University of Technology and Education, in 2014. His main research interests include database, data mining, Big data, and MapReduce.

Seokjoo Lee was born in 1990. He is currently pursuing the M.S. degree in computer science and engineering from Korea University of Technology and Education, Republic of Korea. He received the bachelor's degree from the School of Computer Science and Engineering, Korea University of Technology and Education, in 2014. His main research interests include database, data mining, Big data, and MapReduce.

Jun-Ki Min was born in 1972. He is a Professor with the School of Computer Science and Engineering, Korea University of Technology and Education, Republic of Korea. He received the Ph.D. degree from KAIST, Republic of Korea, in 2002. Then, he continued his research work as a Post-Doctoral Researcher with the School of Computing, KAIST, from 2003 to 2004. In 2004, he worked as a Senior Researcher in ETRI, Republic of Korea. His main research interests include XML, spatial-temporal DB, stream data, sensor data, Big data, and MapReduce. Tel.) 82-41-560-1494, Fax.) 82-41-560-1462.