Developing Complex Data Structures over Partitioned State Machine Replication

Mojtaba Eslahi-Kelorazi, Long Hoang Le, Fernando Pedone
Universita della Svizzera italiana (USI)
Lugano, Swtizerland

***Abstract*—Modern applications require replication for availability. State machine replication (SMR) is a standard way to replicate applications over a number of servers. In SMR, increasing the number of servers improves fault tolerance, but it does not increase performance, since each replica executes all the requests. Partitioned state machine replication seeks to increase performance by partitioning the application state. In this paper, we discuss challenges involved in developing complex applications over partitioned state machine replication. In particular, we develop a distributed B+tree whose nodes are distributed over a set of partitions, and each partition is replicated. B+tree is an important data structure employed in a number of well-known applications and database systems. Moreover, the techniques used in the paper can be easily extended to other data structures and applications.**

***Index Terms*—replication, sharding, distributed b+tree**

I. Introduction

State machine replication (SMR) is a classic approach to fault tolerance [1]. In this method, the application is replicated in a number of servers and each replica deterministically executes client requests in the same order. In SMR, increasing the number of replicas improves fault tolerance because the system can withstand additional failures. However, it does not increase performance since each replica executes all the requests. Several works have extended classic SMR in order to scale performance (e.g., [2]–[5]). One prominent idea is to partition (or shard) the application state and replicate each partition in a number of replicas (e.g., [5]–[11]). Partitioned state machine replication shares the basic characteristics of classic SMR, namely, requests are ordered and then deterministically executed. In order to achieve good performance, only the partitions involved in a request must order and execute the request. Consequently, the partitions involved in the execution of a request must be known *before* the request is ordered and executed. Prior work has considered classes of applications in which this information is readily available (e.g., key-value stores, file systems). In this paper, we share our experiences with developing complex applications with partitioned state machine replication. By complex we mean applications in which the partitions involved in a request cannot be easily identified a priori.

This paper presents DynaTree, a scalable and highly available B+tree over partitioned state machine replication. B+tree is a self-balancing tree data structure that preserves sorting order and guarantees lower bounds for accessing data. DynaTree employs the partitioned state machine replication model proposed by Le et al. [7]. This model partitions the application state and replicates each partition. Client requests are atomically multicast to the partitions involved in the request, and then executed. Atomic multicast ensures that requests are properly ordered across partitions [12]. This model provides scalability and fault tolerance at the cost of additional requirements: clients must identify the data and partitions accessed in a request before the request is executed. This poses some challenges in the case of complex data structures, such as B+trees. For example, to insert a key in the tree, a client must know in advance whether the insert will lead to a split and the nodes and partitions involved in the insert. To satisfy this requirement, clients in DynaTree lazily cache inner nodes of the tree. Therefore, clients first traverse the cached tree to find the appropriate nodes and then issue the request to the involved partitions for execution. This scheme, however, introduces additional complications. Since the client cache may be stale, partitions must verify the validity of the cached information before executing a request. In case a partition finds that the client cache is stale, it informs the client to update the cache and re-try the request.

The paper makes the following contributions: It discusses the challenges involved in designing a distributed data structure in partitioned state machine replication. It introduces a scalable and fault-tolerant distributed B+tree key-value store that scales update operations with the number of partitions. We implemented DynaTree and assessed its performance extensively. Our results show that DynaTree scales read and update requests with the number of partitions, and outperforms a well-established database that relies on a B-tree, deployed in high-availability mode. DynaTree's source code is publicly available.

The reminder of the paper is structured as follows. Section II presents the system model. Section III introduces DynaTree and explains the challenges involved. Section IV discusses implementation details and presents experimental results. Section V reviews related works and Section VI concludes the paper.

II. System Model

We consider a distributed system composed of a set of client and server processes, where servers are divided into groups. Clients submit requests to one or more server groups. Client and server processes are either *correct*, if they do not fail, or *faulty*, otherwise. In either case, processes do not experience arbitrary behavior (i.e., no Byzantine failures). We assume that

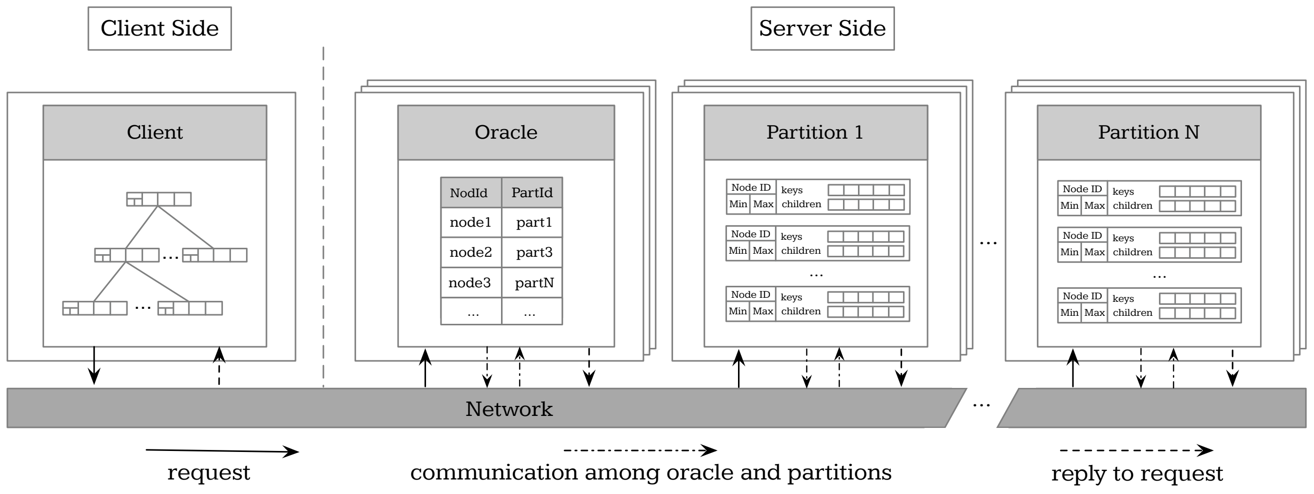

Fig. 1: The client cache, the oracle map, and the distributed tree nodes in DynaTree

there are enough correct server processes in a group so that each group is always operational, despite the failure of some of its members.

The system is *partially synchronous* [13]: it is initially asynchronous and eventually becomes synchronous. When the system is asynchronous, there are no bounds on the time it takes for messages to be transmitted and actions to be executed; when the system is synchronous, such bounds exist but are unknown to the processes. The partially synchronous assumption allows consensus, a fundamental problem at the core of replication [1], [14], and atomic multicast [12] to be implemented under realistic conditions [1], [15].

III. DYNATREE

DynaTree implements a distributed B+tree. In a B+tree, leaf nodes store key-value pairs and inner nodes store key-pointers. When a node is full, it is split into two half-size nodes. With the exception of the root node, which may contain a single key, all other nodes contain $k < n < 2k - 1$ keys, where n is the node size and k is the minimum size of a node [16].

DynaTree is composed of three main components: the client process, the server process, and the oracle process. Figure 1 shows different components and how they are connected.

The client provides an interface with search, update and insert operations. Clients cache internal B+tree nodes. Whether a client searches for a key or inserts a new key in the tree, it traverses the tree in its cache from the root down to the appropriate leaf. After finding the leaf, the client issues a request to the partitions to execute the request. While traversing the tree, the client either has the node in its cache or reads the required nodes from the partitions.

Each server process contains a subset of tree nodes. A partition receives client requests if it is in the destination of the request. A partition is in the destination of a request when the request involves nodes of the tree from that partition. The partitions deliver and execute requests respecting their realtime order so they contain the most updated version of the objects for each request execution to guarantee strong consistency (i.e., linearizability).

The oracle is responsible for storing a location map from tree nodes to partitions. The oracle plays the role of a directory to guide clients to find nodes in partitions. This allows clients to forward requests to the partitions that are involved in the execution of the request.

In the following, we first discuss client caches and their importance in the execution of requests. Then we discuss tree operations and explain the corresponding tree algorithms.

A. Client cache

In DynaTree, clients store internal tree nodes and thus are able to traverse the tree locally rather than reading nodes from the partitions each time. This cache becomes invalid when the structure of the tree changes (e.g., when a node splits). Upon executing a request, the partition notifies the client to invalidate its cache and try again when the client's cache is not valid.

We use fence keys to validate client requests at partitions [17]. Fence keys are essentially two integers determining the range of the keys a node is responsible for, even though the node may not contain all keys in the range (i.e. key never inserted). Fence keys optimize the cache invalidation by decreasing the number of unnecessary invalidations. While splitting nodes, their fence keys are changed. DynaTree guarantees continuous, non-overlapping ranges for the nodes in each tree level in the presence of concurrent node split requests.

A client reads the inner nodes from the partitions and stores them for traversing the tree. The lazy replicated nodes in the client cache may become stale. For example, when a client traverses its stale cache to find a key, it finds a leaf called T is the leaf that should be looked up for the key. So, the client issues a request for looking up in node T. The partition that receives the request will first check if the request is valid. The request is valid if the key is in the fence keys of node T. If node T has split and the key is not in the range of its fence

TABLE I: Tree node fields and methods

Fields/Methods	Description
nid	Node ID
size	Node size
keys	Keys stored in the node
values	Values associated with the keys
children	Pointer to child nodes (if any)
isLeaf()	Returns true if the node is leaf
isRoot()	Returns true if the node is root
lookup(key)	Looks up the key in node
update(key, $value$)	Updates the value of a key
insert(key, $value$)	Inserts the key/value pair
split($parent$, $node$)	Splits and move half of keys into $node$; updates the $parent$ node
isInFenceKeys(key)	true if key is in node's fence keys

TABLE II: Objects and methods in the algorithms

Object	Description
rootId	The root node of the tree
MAX-SIZE	Maximum size of a node
cache	Client cache of internal nodes
storage	List of tree nodes that a partition owns
childParent	A map of nodes to their parents
ancestorlist	List of ancestors of the node
numReservedObjIds	Number of requested object Ids asked by request
reservedObjIds	The oracle reserves a number of object Ids declared in numReservedObjIds and append to the request
result	Result of looking up a key in node; the object contains $value$ if the key is available
RootSplit()	Splits the root of the tree
NodeSplit()	Splits a tree node

keys anymore, the request is not valid. The partition then asks the client to invalidate its cache and try again.

Cache invalidation is optimized by invalidating one node at a time. When a request performing an operation on node T is asked to retry, the algorithm invalidates node T's parent, called T'. The retry asked by the partition implies that the node T' has changed. By reading the parent node again, the client will likely find the correct child next time. In case reading the parent node also results in a retry, the invalidation proceeds one level up. This can go up to the root node where the client clears its cache and starts reading the root node again.

Alg. 1 illustrates the steps executed by clients to read a tree node. The GetNode method starts by searching the client's cache for a node. In case the client does not contain a node, it issues a request to read the node from the partitions. We invoke a DynaStar command by specifying the operation and the arguments passed to the command (line 3). The request is later processed by the server-side logic in the destination partition (18-20). Here, the partition simply returns the node asked by the request. The client checks if the node is valid. Otherwise, the client invalidates its cache and tries again (5-12). Tables I and II summarize the variables and methods employed in the algorithms.

Algorithm 1 Reading a node from cache or partitions

```
 1: GetNode(nid, parentid, key)                              {Client side}
 2:   if $cache$.contains($nid$) then
 3:     return $cache$.get($nid$)                   {read node from local cache}
 4:   <nid, node> ← command(READ, $nid$)
 5:   if $node$.isInFenceKeys($key$) then
 6:     $cache$.add($nid$, $node$)                   {add node to client's cache}
 7:     $childParent$.add($nid$, $parentid$)
 8:     return $node$
 9:   else
10:     parent ← childParent.get($nid$)
11:     InvalidateNode($parent$.nid)                        {cache invalidation}
12:     return null
13: InvalidateNode(nid)                                      {Client side}
14:   $node$ ← $cache$.get($nid$)
15:   for $child$ in $node$.children do
16:     InvalidateNode($child$)
17:   $cache$.remove($node$)
18: Read(nid)                                                {Server side}
19:   $node$ ← $storage$.get($nid$)
20:   returns $node$
```

Algorithm 2 Searching for a key

```
 1: Search(key)                                              {Client Side}
 2:   $node$ ← SearchUtil(key)                       {find the appropriate leaf}
 3:   <RESPONSE,value> ← command(SEARCH, $node$.nid, $key$)
 4:   if RESPONSE = SEARCHFOUND then
 5:     return $value$                          {key is available in the tree}
 6:   else if RESPONSE = SEARCHNOTFOUND then
 7:     return null                                     {key is not available}
 8:   else if RESPONSE = SEARCHRETRY then
 9:     $parent$ ← childParent.get($node$.nid)
10:     InvalidateNode($parent$)                      {invalidate parent node}
11:     return Search($key$)
12: SearchUtil(key)                                          {Client Side}
13:   $node$ ← GetNode($rootId$, null, $key$)             {read the root node}
14:   while !$node$.isLeaf() do
15:     <$nid, result, childid$> ← $node$.lookup($key$)
16:     $node$ ← GetNode($childid$, $nid$, $key$)
17:     if $node$ = null then                        {retry traversing the tree}
18:       return SearchUtil($key$)
19:   return $node$
20: Search(nid, key)                                         {Server Side}
21:   $node$ ← $storage$.get($nid$)
22:   if !$node$.isInFenceKeys($key$) then                   {check fence keys}
23:     return <SEARCHRETRY>
24:   $result$ ← $node$.lookup($key$)
25:   if $result$.isAvailable() then
26:     return <SEARCHFOUND, $result$.value>
27:   else
28:     return <SEARCHNOTFOUND>
```

B. Search and update

Alg. 2 illustrates the steps to search for a key. Updating the value of a node follows a similar execution path, where the client provides the new value for the node. The client starts searching for a key by reading the root node (Line 13) and looks up this node to find the next node to read (15). It continues reading nodes and looking them up down to the leaf (14-18).

After finding the appropriate leaf, the client asks the corresponding partition to search for the key (3). The partition validates the request before execution (22-23). The validation is necessary since the client cache may be stale. The partition uses the leaf's fence keys to find out if the request is valid. If so, the partition is able to search the leaf and return the result to the client (25-28).

Algorithm 3 Inserting a key-value pair

```
 1: Insert(key, value)                                    {Client side}
 2:   $node \leftarrow$ SearchUtil($key$)
 3:   <RESPONSE, $nid$, $key$, $value$, $ancestorlist$> $\leftarrow$
 4:       command(INSERT, $node$.nid, $key$, $value$)
 5:   InsertRespone(RESPONSE,$nid$,$key$,$value$,$ancestorlist$)

 6: Insert(nid, key, value)                               {Server side}
 7:   $node \leftarrow storage$.get($nid$)
 8:   if !$node$.isInFenceKeys($key$) then
 9:     return <INSERTRETRY>
10:   $result \leftarrow node$.lookup($key$)
11:   if $result$.isAvailable() then                     {update the value}
12:     $node$.update($key$, $value$)
13:     return <UPDATED>
14:   else if $node$.size $<$ MAX-SIZE then                {insert new key}
15:     $node$.insert($key$, $value$)
16:     return <INSERTED>
17:   else if $node$.size $=$ MAX-SIZE then                 {split needed}
18:     return <RETRYSPLIT,$nid$,$key$,$value$,null>

19: InsertResponse(RESPONSE, nid, key, value, ancestorlist)
20:   if RESPONSE $\in$ {INSERTED,UPDATED} then               {Client side}
21:     return true                                           {success}
22:   else if RESPONSE = INSERTRETRY then                     {stale cache}
23:     $parent \leftarrow$ childParent.get($nid$)
24:     InvalidateNode($parent$)
25:     return Insert($key$, $value$)
26:   else if RESPONSE = RETRYSPLIT then                     {split needed}
27:     if $ancestorlist$.isEmpty() then
28:       $parent \leftarrow$ childParent.get($nid$)
29:       $ancestorlist$.add($parent$)
30:       $numReservedObjIds \leftarrow 1$
31:     else if !$ancestorlist$.last().isRoot() then
32:       $ancestor \leftarrow$ childParent.get($ancestorlist$.last())
33:       $ancestorlist$.add($ancestor$)
34:       $numReservedObjIds \leftarrow 1$
35:     else
36:       $numReservedObjIds \leftarrow 2$
37:     command.allocate($numReservedObjIds$)
38:     <RESPONSE, $nid$, $key$, $value$, $ancestorlist$> $\leftarrow$
39:         command(INSERTSPLIT,$nid$,$key$,$value$,$ancestorlist$)
40:     InsertRespone(RESPONSE,$nid$,$key$,$value$,$ancestorlist$)
```

C. Insert

Alg. 3 illustrates the insert operation. The client starts the insert request by traversing the tree in its cache. This leads to a leaf which is the node for inserting the key. Next, the client issues a request for inserting the key (lines 4-5). When a partition delivers the request, it checks if the insertion is valid. In case the request is not valid due to stale client cache, the client is asked to invalidate its cache and try again (9-10). The client cache is stale when it asks to insert a key in a node out of the node's fence keys.

In case the insert operation is valid, the node is looked up to see if it already contains the key. In this case, the algorithm only needs to update the value of the key (12-14). If the node does not contain the key and the node is not full, the partition inserts the key in the tree (15-17). In the last case where the node is full, the partition needs to split the node before inserting the key. Therefore, the partition notifies the client that the node is full (18-19). This case involves additional communication steps and is discussed in the next section.

Algorithm 4 Splitting a node

```
 1: Split (nid, key, value, ancestorlist)                  {Server side}
 2:   $node \leftarrow storage$.get($nid$)

 3:   if !$node$.isInFenceKeys($key$) then
 4:     $invalidNode \leftarrow node$.nid
 5:     return <SPLITRETRY, $invalidNode$>

 6:   $result \leftarrow node$.lookup($key$)
 7:   if $result$.isAvailable() then
 8:     $node$.update($key$, $value$)                    {update the value}
 9:     return <UPDATED>

10:   if $node$.size $<$ MAX-SIZE then
11:     $node$.insert($key$, $value$)          {insert the new key-value pair}
12:     return <INSERTED>

13:   for $i = ancestorlist$.size$-1$; $i >= 0$; $i--$ do
14:     $ancestor \leftarrow ancestorlist[i]$
15:     if !$ancestor$.isInFenceKeys($key$) then
16:       $invalidNode \leftarrow ancestor$           {return the invalid node}
17:       return <SPLITRETRY, $invalidNode$>

18:   for $i = 0$; $i < ancestorlist$.size; $i{+}{+}$ do
19:     $ancestor \leftarrow ancestorlist[i]$    {check if ancestors need split}
20:     if $ancestor$.size $<$ MAX-SIZE then
21:       for $j = ancestorlist.size-1$; $j > i$; $j--$ do
22:         $ancestorlist$.remove($j$)

23:   $reservedObjIds \leftarrow$ command.getReservedObjIds()

24:   $lastNode \leftarrow nodes$.get($ancestorlist$.last())
25:   if $lastNode$.size $<$ MAX-SIZE or $lastNode$.isRoot() then
26:     if $lastNode$.isRoot then
27:       RootSplit($lastNode$, $reservedObjIds$)
28:     $pos \leftarrow ancestorlist$.size$-1$
29:     for $pos$; $pos >= 0$; $pos--$ do
30:       NodeSplit($pos$, $node$, $ancestorlist$, $reservedObjIds$)
31:     $node$.insert($key$, $value$)              {insert the new key-value}
32:     return <INSERTED>
33:   else                                          {upmost parent is full}
34:     return <RETRYSPLIT,$nid$,$key$,$value$,$ancestorlist$>
```

D. Splitting nodes

In the B+tree algorithm, a tree node splits when its size hits node's maximum size. While splitting, the node is divided into two nodes, each containing half of the key-value pairs. To split a node, the request needs to involve the node and its parent. The parent node is necessary because a separator key for the new node is inserted in the parent node.

Splitting a node is a more complex operation than the other operations. There are three main reasons for this complexity. First, splitting a node involves creating a new node. The oracle has to be aware of all objects in the system. So, the oracle has to be involved in the request. Second, the split operation can lead to further splits. When a node needs to split while its parent node is full, the operation needs to cascade the split up to the parent node. Third, clients may try to split a node concurrently. The algorithm needs to check if concurrent requests have changed the node before applying changes. In the following, we explain how we deal with these cases.

The split operation starts when a client receives RETRYSPLIT as the result of an insert request (lines 39-40 in Alg. 3). The client starts a new request by adding the node id, the key and the value. There is one more field appended to the arguments list called ancestorlist, which contains the ancestors of the node. Ancestor nodes are added to the list one at a time. When a partition asks the client to retry an insertion with split, the client adds its parent node to the list. If the parent node is also full and needs to split, the client retries by adding the parent of the parent node to the list. The list contains all ancestors of the leaf when the root needs to split. DynaStar ensures that all those nodes are gathered in one partition before the execution of the request.

The Split operation is shown in Alg. 4. The algorithm starts by checking whether the request is valid (lines 3-5). Next, it checks whether the key has been inserted concurrently. In this case, it is enough to update the key's value (6-9). The node has split concurrently with a split request from another client. In this case, the key can be inserted into the node without an additional split (10-12).

For the other cases, we ensure that modifications to the tree deal with concurrent requests to maintain a consistent tree. Starting from line 13, the algorithm verifies that the insertion is valid. Next, the algorithm verifies whether ancestor nodes need to split. We clarify this verification with an example. Assume that there are two splits needed before insertion, which means that there are two items in the ancestorlist. However, it is possible that the immediate parent of the node has split with concurrent requests from a different client. In this case, we avoid splitting the parent node again.

In the last step, the algorithm splits the nodes. The invocation of the split method finishes by moving half of the key-value pairs to the new node and by adding the first element of the new node to the parent node. Now, the algorithm is able to insert the key in the leaf node (31-32).

IV. Evaluation

In this section, we evaluate the performance of DynaTree. In particular, we investigate the scalability of tree operations with the growing number of partitions. Our prototype is written in Java. The source code is publicly available[1].

Experimental environment: We conducted all experiments on a cluster with two types of nodes: (a) Forty nodes (HP SE1102), equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Forty eight nodes (Dell SC1435), equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve 2920-48G gigabit network switch, and the Dell nodes were connected to another, identical switch. Those switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 7.1 with kernel 3.10 and had the OpenJDK Runtime Environment 8 with the 64-Bit Server VM.

A. BerkeleyDB High Availability

BerkeleyDB is a well-known embedded key-value store that provides high performance data management service to applications. BerkeleyDB Java Edition is a native Java implementation that we use in our evaluation as a baseline. BerkeleyDB JE uses a B-tree as its underlying data structure. We use BerkeleyDB with high availability mode enabled. High availability mode adds replication with master-slave model. It provides fault-tolerance and increases the performance of BerkeleyDB under certain workloads. All changes in data have to be performed by the single read-write replica and then propagated to the read-only replicas. We configure the read-write replica to wait for acknowledgements from all read-only replicas to ensure strong consistency (i.e., linearizability) and make it comparable to the strong properties offered by DynaTree.

B. Workloads

In all experiments, unless stated otherwise, the tree is populated with 100K key-values. Tree node's minimum size is 100 key-value pairs. This results in a tree of height 4. The tree is configured to maintain three replicas per partition. Keys and values are 4-byte integers and taken from integer's positive range, chosen randomly from a uniform distribution. Each experiment lasts two minutes; we ignore the first and last 15 seconds. We report peak throughput, achieved by increasing the number of clients to saturate the system, and contention-free latency, by configuring the experiments with a single client. In both setups, DynaTree and BerkeleyDB, we configured the system so that there is enough memory to keep all data in memory.

C. Search scalability

In this experiment we investigate the ability of DynaTree to scale with the increasing number of partitions. Fig. 2 shows the throughput and latency of DynaTree and the BerkeleyDB-HA for 1 to 16 partitions, the maximum number of partitions we can accommodate in our experimental environment.

Since BerkeleyDB is fully replicated, for a workload with only search requests, replicas can individually respond the requests. Therefore, BerkeleyDB scales almost linearly with the number of replicas for search requests. DynaTree also scales linearly for search requests, though it incurs higher overhead than BerkeleyDB, due to coordination introduced by DynaStar which explains its lower peak throughput than BerkeleyDB. Moreover, since BerkeleyDB replicas have a full copy of the data, the execution of a search in BerkeleyDB is local to a replica, which results in lower latency than DynaTree.

These performance advantages come with a cost though. In the BerkeleyDB setup, each replica must contain the entire data. In particular, for the configuration with 16 partitions, BerkeleyDB demands 16 times memory in each partition/replica in comparison to DynaTree. In case servers do not have enough resources to keep all data in memory, they will rely on expensive I/O to read data from disk. DynaTree does not suffer from this drawback as it partitions the data.

D. Update scalability

We now examine the throughput under an update-intensive workload. In the beginning of each experiment, 100K keys,

[1]https://github.com/meslahik/dynatree

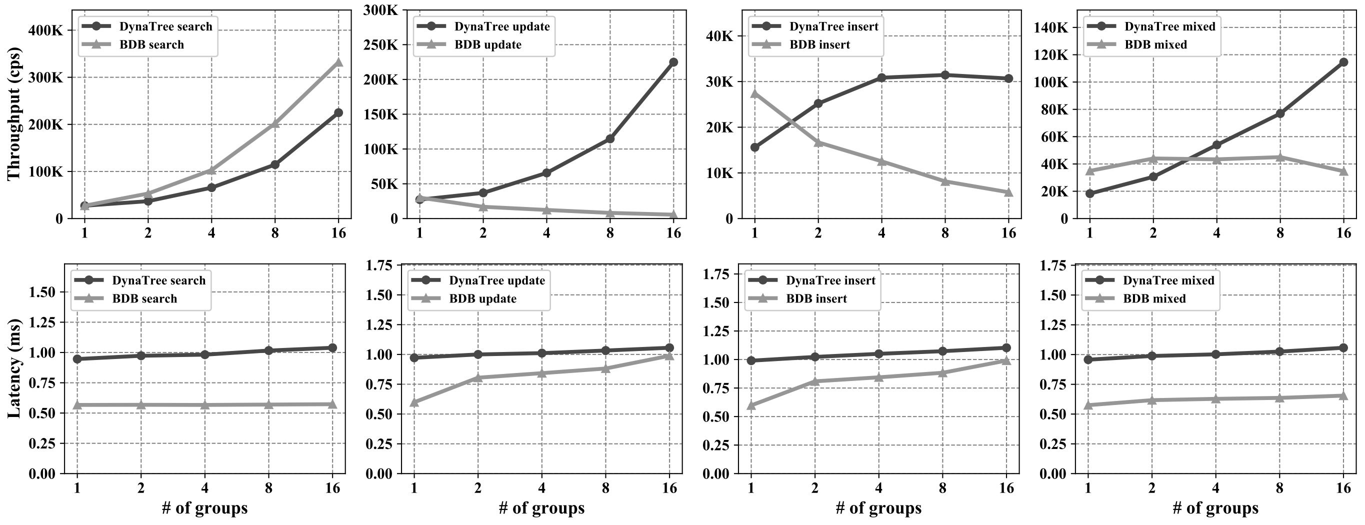

Fig. 2: Peak throughput and contention-free latency for different workloads in DynaTree and BrekeleyDB.

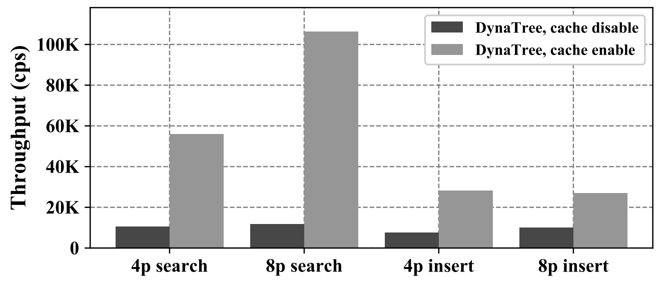

Fig. 3: Effect of client cache on throughput.

chosen randomly from a uniform distribution, have been inserted in the tree. Then, clients in a closed loop choose a key from the inserted keys and update their value. The number of keys show the system behavior in the presence of contention.

BerkeleyDB scales poorly with updates as each update involves all replicas. In DynaTree, however, updates scale linearly with the number of partitions. This is due to the fact that updates do not change the structure of the tree. Thus, updates are single-partition requests to update the key's value.

Latency graph in Fig. 2 shows how the two systems behave when the number of partitions increases. The difference between the latency of requests in one partition/replica configuration is due to the fact that in DynaTree, the requests pass the ordering process. This implies some delays even for requests submitted to one partition. However the latency does not increase with the number of partitions since update operation is a single-partition request. In BerkeleyDB, the latency increases when there are more replicas. The master replica has to wait for acknowledgements from all replicas. Therefore, the more the replicas in the system, the more it has to wait for the completion of updates.

TABLE III: Peak throughput versus tree node size

Node min size	10	50	100	200	1000
Throughput (cps)	7858	19077	31443	20108	15866

E. Insert scalability

In order to show how DynaTree scales out for the 100% insert workload, we conducted experiments where we insert keys randomly, chosen from a uniform distribution. Fig. 2 shows the peak throughput and contention-free latency of both DynaTree and BerkeleyDB as we vary the number of partitions and replicas, respectively.

Insertions scale well up to four partitions in DynaTree. Performance stops growing when the oracle is saturated with the high number of node creations. Each node, when it is full, does not accept a new key and asks the client to retry the request by splitting the node. Each split request involves the oracle for the creation of the new node. When the oracle gets saturated, it cannot handle more requests and the throughput stops growing. However, performance does not decrease with the number of partitions.

BerkeleyDB has good performance with one partition but loses performance with the growing number of partitions. Each replica added to the system has to receive all insertions, and the read-write replica waits for acknowledgements from all other replicas before responding to the client. This explains why the throughput for insertion decreases with the number of replicas.

F. Mixed workload

In this experiment, we assess the performance of DynaTree and BerkeleyDB with a mixed workload consisting of 80% of search requests, 15% of update requests, and 5% of insert requests.

In the mixed workload, BerkeleyDB's performance increases from one to two partitions, then remains stable up to eight partitions, after which it decreases. The expensive updates and inserts affect the mixed workload even with 5 percent of inserts. DynaTree scales well up to 16 partitions. The reason is that, from the previous experiments, DynaTree scales linearly with searches and updates, and the rate of inserts in the mixed workload is within the range that the oracle can handle. Both BerkeleyDB and DynaTree experience little variation in latency with the number of replicas and partitions, respectively.

G. Client cache impact

The client cache plays an important role in DynaTree. Thanks to the cached data, clients can identify the partitions involved in the execution of a request before the request is executed by the servers. However, requests based on outdated cached data may lead to expensive retries. One question that arises is whether the client cache indeed improves performance.

When the tree structure changes, cached data must be invalidated and clients may need to retry requests based on stale data. Retries are expensive since clients need to interactively rebuild their cached tree, starting from an up-to-date tree node, or the tree root, in the worst case. Notice that while the client cache reduces the probability of retrying a request, it does not eliminate retries due to concurrent accesses to common parts of the tree.

Finally, we try to mitigate the effect of cache invalidation by invalidating the client cache step-by-step rather than invalidating the whole cache. Since the changes in the tree are propagated bottom-up, it is probable that only a small part of a branch has been changed.

Fig. 3 shows the performance of DynaTree with and without the client cache. We conducted experiments with search-only and insert-only requests in deployments with 4 and 8 partitions. When the clients do not have a tree node in their cache, they ask for that from the partitions. This decreases the performance of both search and insert requests considerably, with direct implications on performance. The insertion workloads, which expects high frequency of cache invalidation, gain performance improvement from client cache. The improvement is more significant in search workloads due to no cache invalidation.

H. Node size variation

We assess the effect of tree node size on the peak throughput. There is a performance tradeoff involving the tree node size. A large node results in fewer splits, which can improve the throughput of insertions. However, a large node leads to expensive marshaling and unmarshaling costs when nodes are moved from one partition to another. The nodes are moved between partitions while splitting a node or when a client asks for an inner nodes to traverse the tree. The results reveal the tradeoff between the node size and the throughput. Table III shows the peak throughput for 8 partitions versus tree node minimum size with insert workload. Motivated by these results, we have configured all other experiments in the paper with a tree node minimum size equal to 100.

V. Related Work

The organization and maintenance of large ordered indexes based on B-tree date back to the 70s. [16]. This was later followed by efforts to introduce concurrency in the execution of B-tree operations. Investigation into the performance of concurrent tree algorithms [18], [19] showed that B-link trees [17] provide the best performance for most operations. In a B-link tree, nodes own a pointer to their right sibling. The algorithm follows the links to find the correct tree node when the node has changed concurrently. This limits the number of locks for any operation to one. A distributed dictionary based on a distributed B-link tree was introduced later in [20]. An extension of B+Tree in P2P approaches for multi-dimensional information is proposed in [21].

Designing linearizable systems that scale has been considered in the past (e.g., [5]–[8], [22], [23]). S-SMR [5] ensures strong consistency for multi-partition requests by synchronizing partitions. It relies on a static partitioning of state to partitions. DynaStar [7] improves on S-SMR by migrating state variables across partitions in order to execute multi-partition requests. DynaStar employs well-known graph partitioning techniques [24] to decide where each object should be. As a result, it reduces the number of multi-partition requests. Scatter [22] introduces a distributed hash table in a P2P environment and provides linearizable consistency semantics but for operations on a single key/value pair. Calvin [25] is a transaction scheduling layer over non-transactional storage systems that provides strong consistency and ACID transactions. Its deterministic locking order allows high throughput for multi-partition requests. Sarek [26] provides parallel ordering in byzantine fault tolerant environments. It partitions data in a number of partitions and uses predictions to learn which partition is involved in the execution of a request. It provides mechanisms to handle mis-predictions and therefore avoid rollbacks. Consistent hashing [27] is an approach to partition data through mapping keys to partitions. It is particularly useful when the number of partitions may change. While the mapping of keys to partitions is intended not to change, unless machines are added or removed, DynaTree benefits from dynamic partitioning that adapts to workload changes over time. Dynamic partitioning minimizes the number of state moves and results in a higher number of single-partition requests.

There are several studies investigating a distributed B-tree. Boxwood [28] studies the possibility of having a high-level scalable data structure as the fundamental storage infrastructure. It is shown that there is no universal abstraction that fits all needs. Spanner [8] uses Paxos to support replication for its global-scale distributed database. Spanner ensures strong consistency across partitions with synchronized clocks in each partition. Mitchel et al. [29] introduced a cell distributed B-tree store. Their model explores the possibility of using the potential network capabilities when the processor becomes the bottleneck. HyperDex [30] is another distributed data store that provides a new search primitive for retrieving objects by secondary attributes. It deterministically maps objects to servers according to object values. Objects are duplicated to increase the performance of searching for an index with the cost of slower update operations. Aguilera et al. [31] implemented a distributed B-tree using Sinfonia [32], a distributed data sharing service. They use distributed transactions to make changes to B-tree nodes. The overall throughput of the proposed system is limited due to large number of aborts in their model. Sowell et al. [33] extended the previous work to unify operational and analytics systems. Their model is a multi-version tree with snapshots to increase the performance at the cost of weaker consistency guarantees. Aguilera et al. [34] introduced a distributed balanced tree in the core of their storage engine for Yesquel. A balanced tree differs from a B-tree to some extent, for example, by providing load balance rather than size balance. Yesquel does not scale for insert operations. Even though Yesquel's design supports replication, the available implementation does not include replication. Thus, we do not experimentally compare Yesquel to DynaTree.

VI. Conclusion

In this paper, we presented DynaTree, a distributed B+tree that is both scalable and fault-tolerant. DynaTree provides an architectural design for a B+tree whose nodes are distributed among a number of partitions in a partitioned state machine replication system. It is shown that building a complex data structure such as a B+tree in partitioned state machine replication system implies a number of challenges. We discussed those challenges and presented distributed algorithms for tree operations which deal with concurrent tree modifications while ensuring strong consistency. The results show that both read and update operations scale linearly with the number of partitions. Moreover, insert operations do not lose performance with the growing number of partitions.

Acknowledgements

We wish to thank the anonymous reviewers for helping us improve the paper. This work was partially supported by the Swiss National Science Foundation (project number 175717).

References

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[2] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *DSN 2016*.

[3] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *ICDCS 2013*.

[4] M. Kapritsos and F. P. Junqueira, "Scalable agreement: Toward ordering as a service." in *HotDep 2010*.

[5] C. E. B. Bezerra, F. Pedone, and R. van Renesse, "Scalable state-machine replication," in *DSN 2014, Atlanta, GA, USA, June 23-26*.

[6] L. L. Hoang, C. E. B. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *DSN 2016, Toulouse, France, June 28 - July 1*.

[7] L. L. Hoang, E. Fynn, M. Eslahi-Kelorazi, R. Soule, and F. Pedone, "Dynastar: Optimized dynamic partitioning for scalable state machine replication," in *ICDCS 2019*.

[8] J. C. Corbett and et al., "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

[9] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, "Clay: fine-grained adaptive partitioning for general database schemas," *VLDB 2016*.

[10] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi, "Squall: Fine-grained live reconfiguration for partitioned main memory databases," in *2015 ACM SIGMOD International Conference on Management of Data*.

[11] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *VLDB 2010*.

[12] P. R. Coelho, N. Schiper, and F. Pedone, "Fast atomic multicast," in *DSN 2017, Denver, CO, USA, June 26-29*.

[13] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[14] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[16] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, Sep 1972.

[17] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on b-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, 1981.

[18] T. Johnson and D. E. Shasha, "A framework for the performance analysis of concurrent b-tree algorithms," in *Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*.

[19] V. Srinivasan and M. J. Carey, "Performance of B+ tree concurrency algorithms," *VLDB J.*, vol. 2, no. 4, pp. 361–406, 1993.

[20] T. Johnson and A. Colbrook, "A distributed data-balanced dictionary based on the b-link tree," in *Proceedings of the 6th International Parallel Processing Symposium, Beverly Hills, CA, USA, March 1992*.

[21] S. Bianchi, P. Felber, and M. G. Potop-Butucaru, "Stabilizing distributed r-trees for peer-to-peer content routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1175–1187, 2009.

[22] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. E. Anderson, "Scalable consistency in scatter," in *SOSP 2011*.

[23] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *DSN 2011*.

[24] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *IPDPS 2006, 25-29 April 2006, Rhodes Island, Greece*.

[25] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 1–12.

[26] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "Sarek: Optimistic parallel ordering in byzantine fault tolerance," in *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016, pp. 77–88.

[27] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.

[28] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *OSDI 2004, San Francisco, California, USA, December 6-8*.

[29] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing CPU and network in the cell distributed b-tree store," in *USENIX ATC 2016, Denver, CO, USA, June 22-24*.

[30] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: a distributed, searchable key-value store," in *SIGCOMM '12*.

[31] M. K. Aguilera, W. M. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *VLDB 2008*.

[32] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP 2007, Stevenson, Washington, USA, October 14-17*.

[33] B. Sowell, W. M. Golab, and M. A. Shah, "Minuet: A scalable distributed multiversion b-tree," *VLDB 2012*.

[34] M. K. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: scalable sql storage for web applications," in *SOSP 2015, Monterey, CA, USA, October 4-7*.