SinExTree : Scalable Multi-Attribute Queries through Distributed Spatial Partitioning

Mahdi Tayarani Najaran
Computer Science Department
University of British Columbia
tayarani@cs.ubc.ca

Charles Krasic
Computer Science Department
University of British Columbia
krasic@cs.ubc.ca

Norman C. Hutchinson
Computer Science Department
University of British Columbia
norm@cs.ubc.ca

Abstract

In this paper we present SinExTree, a spatial partitioning tree designed for scalable low-latency information storage and retrieval. SinExTree is built over a Sinfonia-like service that provides atomic access to distributed memory suitable for a cloud environment. An n-dimension SinExTree provides key/value storage, where each key has n attributes, and supports general application-defined queries over multiple attributes.

1 Introduction

Commercial distributed systems and services rely on distributed key/value storage systems, such as Amazon's S3 [1] and Cassandra [2], which can store massive amounts of data. However, data can only be retrieved by lookup using the exact same key. Enumeration primitives are not supported, nor are complex queries. While such properties fit nicely for many applications, other types of applications expect more from the underlying storage system.

We present *SinExTree*, a scalable distributed spatial partitioning tree. An n-dimension SinExTree provides a key/value storage system, where each key has n attributes. Attributes in a SinExTree can have any type, as long as a strict ordering can be defined over the possible values of each attribute. SinExTree also provides strong consistency with scalable key/value operations and enumeration primitives, along with general application-defined queries over keys stored in the system, and is built over SinfoniaEx, an extension to Sinfonia [4].

Sinfonia is a distributed memory service based on short-lived minitransactions. SinfoniaEx extends Sinfonia by providing a set of new transaction items which provides a cleaner and more natural interface for distributed systems, while allowing applications to share the memory nodes that host their data.

Unlike related work in this area, SinExTree is designed to scale for any type of workload, in terms of read/write intensity, and is not tuned for or limited to a specific type of workload. Such scalability, along with the rich set of enumeration primitives and application defined queries and low latency response times, well meets the requirements of real-time applications and distributed systems.

SinExTree serves as a building block for distributed systems, where various types of applications can benefit from it, but we limit our discussion to three different cases. First, SinExTree can be used to index various attributes of items stored in a distributed database, or any system needing to store multiple indices per data entry. Such applications require the indexing system to be able to answer complex queries across multiple attributes with strong consistency, but without requiring a priori data partitioning or knowledge of the queries. Second, SinExTree may be used to store metadata in a distributed file system. The workload of such applications involves a high volume of metadata reads, while requiring atomic write operations to prevent corrupting the file system. Third, SinExTree can be used as the back-end of a multi-player online game, which has the most challenging type of workload. The workload consists of a high volume of reads, writes and complex queries every second, where it is critical for operations to complete at most within tens of milliseconds.

We first begin by providing a background on spatial partitioning trees and SinfoniaEx in Sections 2 and 3, respectively. Next, we present details of SinExTree in Section 4. Section 5 presents related work and we conclude in Section 6.

2 Spatial Partitioning Trees

A spatial partitioning tree (SPT) is used to store keys with multiple attributes, along with a value per key. An n-dimensional SPT supports keys with n attributes, each with any data type, as long as a strict ordering can be defined over the data type. A node in an SPT encloses

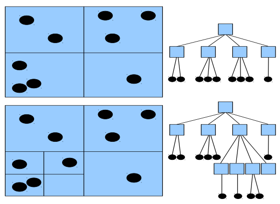

Figure 1: Spatial and internal structure of a Quadtree with maximum 3 keys per node: (top) before new key insert (bottom) after new key insert.

a specific region in an n-dimensional space of possible keys, defined by the min and max possible keys the node can hold. For example, in 2D and 3D trees, e.g., Quadtrees and Octrees, each node encloses a square and cubic region, respectively. The node is either a *leaf* node (holds key/value pairs) or a *branch* node (holds pointers to M other nodes referred to as its *children*). The children of a single node enclose non-overlapping regions which may have different sizes and their union is exactly equal to the space enclosed by their common parent. The choice of M, i.e., *fan-out*, directly impacts the depth of the tree, where 2^n is a common choice.

An SPT supports three basic key/value operations: *insert(key, value)*, *lookup(key)* and *remove(key)*. A key is inserted into a node if it falls within the range of min possible key (key_{min}) and max possible key (key_{max}) the node may store. Keys inserted in branch nodes are inserted in their appropriate child. A leaf node can hold a pre-defined maximum number of keys. If number of keys held by the leaf exceeds the maximum, it is *split*, i.e., transformed into a branch node and the keys are moved into its children. Figure 1 illustrates an insert operation in a 2D SPT where each child node can store a maximum of three key/value pairs.

Looking-up a key starts at the root of the tree. At each node, the operation is forwarded to the child node that encloses the given key, until a leaf node is reached and key/value pairs stored in the leaf are checked. Remove traverses the tree in a similar fashion, removing the key (if found) from the leaf. However, after a remove, parent nodes check the total number of keys stored in their children. If all are empty, a *merge* operation is done, where child nodes are destroyed and the node in transformed into a leaf node.

The spatial layout of SPTs allows various types of checks to be performed on keys in the tree, such as keys within a specific range, keys within a specific range of another key, and keys along any arbitrary direction. This has lead to wide spread use of 2D and 3D SPTs in applications interacting with virtual environments, where a key is the position of an object in the environment, and the SPT is used for ray tracing in graphics [11], collision detection in physics simulations [7] and scene management in 3D engines [8].

3 SinfoniaEx

This section describes our extensions to Sinfonia [4], creating a more general service we call *SinfoniaEx*. SinfoniaEx is based on the same design principles as Sinfonia, and is backward compatible (i.e., any application built over Sinfonia can be built over SinfoniaEx, hence, the name *SinfoniaEx*), and available with an open source implementation.

3.1 Sinfonia

Sinfonia is a distributed shared memory service allowing applications to share data in a fault-tolerant, scalable and consistent manner. It has been shown to provide an efficient programming model for classic distributed systems problems, such as file systems, group communications, and scalable storage [4, 3]. A set of Sinfonia *memory nodes* export an unstructured linear address space, where a data pointer consists of the tuple $\{$*memnode-id,offset*$\}$. Applications access the exported memory atomically via short-lived light-weight *minitransactions*. [1] Each minitransaction uses a *two-phase commit*, which is designed to provide high performance by using at most two network round-trips to complete (regardless of commit succeeding or not).

A transaction consists of three groups of items, categorized by the kind of operations they perform: *read items*, *compare items* and *write items*. Read items read and return data from the specified addresses, compare items compare user provided data with data stored in memory, and write items write provided data to specified memory locations. A transaction begins with empty items and the application adds items to each category before calling commit (note that each item can be pointing to data on different memory nodes). Once a transaction is committed, it either fails if and only if at least one compare item failed (i.e., the data in the compare buffer does not match data stored in the referenced memory location), or succeeds, in which case data in the write buffers are written to memory and read items contain buffers holding data read from memory.

Minitransactions provide consistency by locking memory locations they reference while committing is in progress. Unlike conventional full-featured transactions, minitransactions are meant to be light-weight. The two-phase commit ensures locks on memory locations are short-lived, lasting at most two network round-trips (one and a half to be accurate), providing high performance atomic transactions as the basic memory access primitive. Sinfonia also provides fault tolerance and some op-

[1]We use the terms transaction and minitransaction interchangeably in the course of this paper, not to be confused with traditional database transactions.

timization techniques discussed in [4] to further increase reliability and performance of Sinfonia.

Sinfonia has some short comings in memory management. A Sinfonia memory node's address space is raw and unstructured, allowing any arbitrary transaction to access any part of memory. This forces applications to keep track of memory maps themselves, i.e., what parts of the memory of each memory node are holding application data and what parts are un-allocated. This increases the complexity of the applications using Sinfonia by forcing them to implement their own memory allocation/deallocation functions, namely $malloc$ and $free$, that have long existed in all operating systems and are highly optimized.

Additionally, since applications handle their own memory mappings, different applications face difficulties sharing memory nodes. For example, consider application A which is a distributed B-tree, and application B which is a distributed file system, both built over Sinfonia. The memory nodes hosting the B-tree have to either be exclusive to the B-tree, i.e., no sharing between the applications which will result in over/under utilization of resources allocated to one application, or A and B have to reach an agreement on how to share the memory nodes to avoid data corruption. The agreement can either be to use a specific algorithm, like bit vectors stored at the memory nodes to mark fixed sized pages as allocated/free [3], or to rely on an external allocator. Using bit vectors is inefficient, as the applications should decide on the page size to use which will lead to fragmentation and inefficient use of memory. Using external allocators is also inefficient since transactions trying to allocate memory may fail due to concurrent allocations, which adds extra network round trip times to memory allocation. Finally, every transaction has to ensure the memory locations it references have not been freed. This adds to the weight of the transactions, causing contention on the memory maps, and violates the principle of lightweight minitransactions.

We aim to address these deficiencies of Sinfonia through SinfoniaEx. SinfoniaEx relies on the same fundamentals of two-phase short-lived minitransactions and fault tolerance as Sinfonia, while dividing the address space exported by a memory node into two spaces, each accessed via its own specific transaction item types. In the remainder of this section we focus on features and design of SinfoniaEx.

3.2 SinfoniaEx Address Spaces

SinfoniaEx memory nodes export two separate address spaces: *un-protected* and *protected*, each accessed through its own set of transaction items. The unprotected address space mostly resembles the address space of Sinfonia, i.e., any application can write, read or compare to any location. Applications sharing this address space require some sort of coordination and should use external memory mappings to ensure they don't corrupt each other's data. On the other hand, the protected address space handles memory mappings internally and only allows operations on pointers that have been malloced, but not yet freed. A transaction trying to access any other memory location is bound to fail. This address space allows applications to use it without knowledge of other potential applications sharing it.

```
//Protected Address Space Items
read(memnode-id, offset, len)
compare(memnode-id, offset, len, data)
write(memnode-id, offset, len, data)
malloc(memnode-id, pointer-id, len, data)
free(memnode-id, offset)
//Un-protected Address Space Items
lookup(memnode-id, offset)
compare_unprot(memnode-id, offset, buf, len)
put(memnode-id, offset, buf, len)
remove(memnode-id, offset)
compare_non_exist(memnode-id, offset)
//Access Macros
buf=get_read_item(memnode-id, offset)
buf,len=get_lookup_item(memnode-id, offset)
offset=get_malloc_offset(memnode-id, pointer-id)
```

Figure 2: List of SinfoniaEx items. The first five operate on the protected address space, while the next five use the unprotected address space. The last three macros are used to access data returned by the transaction.

Figure 2 lists SinfoniaEx items, and the macros used to access data returned by a transaction. For all items, the tuple $\{$*memnode-id, offset*$\}$ represents a memory pointer in SinfoniaEx and the type of the item specifies the address space (e.g., put is in the un-protected space while write is in the protected). Even though the address spaces are disjoint, a single transaction may use any mix of items regardless of the address space. We next delve into details of each item.

3.2.1 Protected Address Space Items

Transactions can allocate/de-allocate memory in this address space using `malloc` and `free`. A pointer in this address space must be malloced before any other operation may be performed on it. A `malloc` item takes the *memnode-id* of the memory node that should host the newly allocated location, along with a user provided handle *pointer-id* which should be unique within the transaction. Once the transaction commits successfully, the offset of the newly allocated pointer can be retrieved using the `get_malloc_offset` macro by providing the same handle. An optional data buffer may also be provided to `malloc` in order to initialize the newly allocated memory with user specified values. `free` items simply take *memnode-id* and *offset* of the memory location to release upon successful commit. A `read` item reads the amount of data specified by *len* from the given pointer, while a `compare` item compares given data of size *len* to what it stored at the specified pointer. Finally, a `write` item writes *data* to the specified location.

A transaction having an item belonging to this address space may fail due to two reasons. First, if there is a mismatch between a compare item and what is stored in the memory node. Second, if an item is referencing an illegal pointer. Having memory nodes handle memory allocations provides a safe environment for distributed data structures, ensuring no operations on the structure touch invalid memory locations. To our experience, using this address space moves the complexity of distributing an application down to the SinfoniaEx layer, and greatly simplifies the transition from a non-distributed application to a distributed one.

3.2.2 Un-protected Address Space Items

The `lookup`, `compare_unprot` and `put` items of this address space are identical to Sinfonia's read, compare and write items, and simply take a pointer and buffer holding data (if required). Based on their semantics, we decided to rename them to those used for key/value operations, where the key is the memory pointer and value is the buffer. Sinfonia provides no way for an application to distinguish whether a memory location is in use and holds valid data, or not. So to complete the set, we added `remove` and `compare_non_exist` items. `put` stores provided data to the given address, and marks the address as occupied. `lookup`, `compare_unprot` and `remove` only succeed on occupied locations, where remove marks the address as unoccupied. Finally, a `compare_non_exist` item ensures the address is unoccupied prior to committing the transaction.

The un-protected address space provides applications the freedom to decide where their data is stored, and is designed for backward compatibility with Sinfonia, while providing a richer set of primitives. It is suitable for applications that have some sort of coordination, e.g., use a secure hash function for mapping data to memory locations, and may itself be used for coordinating applications, e.g., storing metadata about a distributed data structure (see Table 1).

Figure 3 presents an example of creating a distributed tree using most of SinfoniaEx's items. The first transaction allocates a metadata object on a random memory node (line 1-4) and stores its address (line 5). The second transaction tries to store the address of the metadata at location 123456 on memory node 3 only if it doesn't exist (line 7-10). The transaction only fails if the tree al-

```
1  $tx \leftarrow$ new SinfoniaEx_Tx()
2  loc.memnode $\leftarrow$ rand() mod #memnodes //random node
3  $tx$.malloc( loc.memnode, 777, sizeof(tree_metadata), nil )
4  $tx$.commit()
5  loc.offset $\leftarrow$ $tx$.get_malloc_offset( loc.memnode, 777 )
6  $tx$.end()
7  $tx \leftarrow$ new SinfoniaEx_Tx()
8  $tx$.put( 3, 123456, loc, sizeof(loc) )
9  $tx$.compare_non_exist( 3, 123456 )
10 $tx$.commit()
11 if $tx$.success() then
12     //metadata has been allocated and its address stored
13 else
14     $tx$.end()
15     $tx \leftarrow$ new SinfoniaEx_Tx()
16     $tx$.lookup( 3, 123456 )
17     $tx$.free( loc.memnode, loc.offset )
18     $tx$.commit()
19     loc,len $\leftarrow$ tx.get_lookup_item( 3, 123456 )
20 $tx$.end()
```

Figure 3: SinfoniaEx transaction example.

ready exists, in which case the address of the metadata is looked up from node 3 and allocated memory is released (line 14-19). Returned data is retrieved using an access macro.

SinfoniaEx uses the same two-phase commit protocol as Sinfonia, with minor modifications. For details refer to [10].

3.3 Load Balancing & Security

Balancing the load of different memory nodes is an important feature of SinfoniaEx, which is desired by many applications. Each SinfoniaEx memory node may use a set of reserved keys to store metadata about its current load information, such as allocated and free memory byte counts, number of requests handled per time unit, etc, accessible through transactions. Considering applications decide which memory nodes to use when allocating memory locations, they can benefit from this information to select the most suitable memory nodes. Moreover, applications can use transactions to migrate data to and from memory nodes (see Section 4.2). By periodically reading this metadata, applications can shift around data to evenly spread their load between different memory nodes.

SinfoniaEx allows different applications to share the same memory nodes. Albeit, sharing memory nodes may pose security threats, since an application can virtually access any other application's private data. SinfoniaEx is based on the same assumptions as Sinfonia, in which applications operate in a data center and their designers are trustworthy, rather than malicious (in contrast to WANs and peer-to-peer systems) [4]. Nevertheless, security cre-

$Query_1$: "$x > 27$ and $x < 487$"
$Query_2$: "$x > 3 \times y$ and $z < 24$"
$Query_3$: "$d_0 < a \times x + b \times y + c \times z < d_1$"

Figure 4: Sample queries supported by SinExTree.

dentials can be added to transactions and memory nodes can authenticate applications while restricting their access to memory locations, a task left for future work.

4 SinExTree

In this section we present the design of SinExTree over SinfoniaEx. SinExTree is designed to be a distributed SPT, where all machines storing or using the tree are in the same data center: they have high bandwidth low-latency connectivity to each other. SinExTree is designed to distribute the spatial partitioning tree it represents when the tree is either too big to be stored on a single machine, or the workload on the tree is greater than what a single machine can handle.

SinExTree was designed to be an SPT in order to support multi-attribute queries. Figure 4 lists a few example queries in SinExTree. Notice a query can have constraints on any attribute, or even their combination. The goal is to provide a better abstraction able to support complex queries which cannot be done directly using other data structures or systems, such as B-trees or relational databases.

4.1 API

Applications using a SinExTree operate with the tree via an API listed in Table 1, which is very similar to that of data structures stored in local memory. A SinExTree is stored in a set of SinfoniaEx memory nodes, identified by a special key stored at a specific node in its unprotected address space. Applications using the SinExTree, referred to here as clients, get to know about the tree key via some external mechanism (e.g, offline agreement, external service, etc). The key holds a pointer to metadata stored of the tree, such as size/center of the tree, maximum capacity of each node, and the address of the root node, all stored in the protected address space.

The first client calling `Create` creates and stores the metadata, which is read by other clients calling `Create` on the same key. The implementation of `Create` is very similar to Figure 3. Calling `Destroy` on a tree frees all metadata and tree nodes associated with the tree. The rest of the API provides three basic operations, `Insert`, `Lookup` and `Remove` and `Query` function. In general, all operations use a single SinfoniaEx transaction, the operation's *uber transaction*, to update the tree atomically and consistently, and to validate result correctness of the operation.

Table 1: Basic API of SinExTree

Operation	Description
Create(*tree_key*)	Reads metadata of tree identified by *tree_key*. Creates new if none found.
Destroy(*tree_key*)	Destroys tree associated with *tree_key*.
Insert(*k*,*v*)	Inserts (*k*,*v*) into tree.
Lookup(*k*)	Returns *v* if (*k*,*v*) is found, else *nil*.
Remove(*k*)	Removes (*k*,*v*) if found.
Query(*q2n*,*q2k*,*con*)	Query defined by provided functions: *q2n*: tests each tree node with query. *q2k*: tests each key with query. *con*: requested consistency level.

A node in the tree is located by its address stored at its parent. Starting at the root of the tree, all nodes either required for or affected by an operation are read using separate transactions and appended to the uber transaction's compare items. Additionally, nodes affected by the operation are appended to the uber transaction's write items. An operation succeeds when its uber transaction is successfully committed. Appending nodes to the compare items ensures the nodes have not changed since they were read, preventing race conditions on parts of the data structure accessed concurrently. An uber transaction fails if some of the nodes change between when they were read and the time the uber transaction is committed. This requires the uber transaction to abort and be retried from the beginning. In the remainder of this section we present details of the implementation of operations and queries.

4.2 Operations

Figure 5 lists the pseudo code for the Insert operation in a SinExTree. The majority of the code is similar to a normal SPT. Lines marked with a (*) are lines added for the SinExTree. The `Insert` operation starts at the root, forwarding the key/value pair to the appropriate child node until a leaf node is reached, which then stores the data (lines 6-15). If a leaf node runs out of storing capacity, new nodes in the tree are allocated to store key/value pairs currently stored at the leaf node, which is now transformed into a branch node (lines 16-18, 20-25). Newly added lines 1, 3-5 create and commit the uber transaction, retrying until success, while lines 11, 14, 19, 26 ensure the uber transaction includes all nodes required for correctness. Other functions modified for SinExTree are `read_node` which reads the specified node using a separate transaction, and `malloc_nodes` that allocates new nodes, each on a random memory node, returning their addresses and a local copy of newly allocated nodes. The implementation of `Lookup` and `Remove` are not presented due to space limitations and redundancy.

Various features of SinExTree allow us to optimize performance. The compare items are used to validate the

```
//Inserts ($key$,$value$) into the tree
**Insert**($key$, $value$):
1*  $tx \leftarrow$ new minitransaction //the uber transaction
2   node_insert($root$, $tx$, $key$, $value$)
3*  $action \leftarrow$ commit($tx$) //commit retries tx until success or fail
4*  if $action$==COMMIT then return
5*  else reload failed items of $tx$; end $tx$; goto 1

//Either inserts ($key$,$value$) into $node$, or forwards operation to
//appropriate child of $node$. Adds nodes affected by operation to
//compare and write items of $tx$.
**node_insert**($node$, $tx$, $key$, $value$):
6   if $node.type$==BRANCH then
7       $index \leftarrow$ index of child which $key$ should be inserted into
8       $child \leftarrow$ read_node($node$.child_address($index$))
9       if $node.child\_empty[index]$ == $true$ then
10          $node.child\_empty[index] \leftarrow false$
11*         $tx$.add_write_item($node$)
12      node_insert($child$, $tx$, $key$, $value$)
13  else
14*     $tx$.add_compare_item($node$)
15      $node.data \leftarrow node.data \cup \{(key, value)\}$
16      if $node.capacity$==0 then
17          $node.type \leftarrow$ BRANCH
18          split_node($node$, $tx$)
19*     $tx$.add_write_item($node$)

//Transforms $node$ into branch node, shifting all key/value pairs
//to newly allocated child nodes
**split_node**($node$, $tx$):
20  $nodes \leftarrow$ malloc_nodes()
21  $node.child\_address \leftarrow$ address of all nodes in nodes
22  foreach (key,value) in $node.data$ do
23      $node.data \leftarrow node.data - \{(key, value)\}$
24      $index \leftarrow$ index of node which $key$ should be inserted into
25      node_insert($nodes[index], tx, key, value$)
26* $tx$.add_write_item($nodes$)
```

Figure 5: Pseudo-Code for Insert.

path an operation takes from the root to the resulting leaf. However, we observe that most operations usually modify a sub-tree of SinExTree, and the internal structure, especially the upper levels, tend to change less frequently than leaf nodes. This allows us to make two important optimizations. First, we can only validate the leaf node touched by the operation, not the entire path. This highly reduces contention on upper nodes in the tree since they no longer have to be locked by the uber transaction of each operation. Second, we can cache the inner structure of the tree by modifying the `read_node` function to store a local copy of nodes it reads, and return them on subsequent reads.

Caching improves performance by decreasing network round trips required to fetch different nodes, at the cost of the uber transaction facing a higher failure probability due to out-of-date cached items. Upon failure, all failed items of the uber transaction are identified, compare failed items are reloaded, while a reload is tried on parents of items referencing bad memory locations. In either case, if reloading a node fails, it is removed from the local cache and a reload is tried on its parent. If an uber transaction commits successfully, its cached write items are updated in the local cache.

The `malloc_nodes` function decides where to allocate memory for new nodes in the tree. An optimal decision at allocation time may be invalidated as the load of the memory node changes over time. SinExTree allows tree nodes to be migrated at any time. An application may use its own policy, with the aid of a single transaction, to move nodes between memory nodes and change parent/children references at once. Such a feature is much desirable and necessary for many applications.

4.3 Queries

Queries are an important feature of SinExTree, and unlike other distributed data structures, it supports general application-defined queries. Recall each node in an n-dimensional SinExTree encloses a sub-space with n dimensions. A query can be thought of a sub-space with n dimensions, looking for keys that fall on or within its enclosed sub-space. A query executes in two steps. First, all the nodes in the tree holding possible keys required by the query should be identified. This translates to finding all nodes that enclose a space intersecting the space of the query. Starting at the root, the region enclosed by each node in the tree is tested against the query. If they intersect, the query is forwarded to the node's children that intersect the query, which are read using read transactions, until all intersecting leaf nodes are found. Second, all keys in the found leaf nodes should be checked for membership in the query to produce the final result. At this point, each key in the found leaf nodes is tested against the query, and appended to the result of the query if the membership test passes, finalizing the result of the query.

A SinExTree query can provide one of two levels of consistency in its result: *full* and *none*. Consistency level full ensures the result is consistent since when the query started until it finished reading nodes, with the help of an uber transaction. The leaf nodes read for the result of the query are appended to its uber transaction, which commits before the second step is executed to ensure the nodes haven't changed. If the uber transaction fails, step one has to be retried until it succeeds. For consistency level none, there is no use for the uber transaction, and step two is performed right after step one. Fully consistent queries are very expensive compared to none consistent ones, since the uber transaction of a fully consistent query has to lock the query region.

Finding the intersection of each node with a query and testing for memberships of keys in a query are done using functions provided by the application, $q2n$ (query-to-node) and $q2k$ (query-to-key) in Table 1, respectively. This provides applications the flexibility to define any query without having to modify SinExTree. For each node tested against the query, $q2n$ is passed key_{min} and key_{max} defining the node, while $q2k$ is passed the key to be tested, where each returns either $true$ or $false$. We implement three types of queries in our 3D SinExTree: *range query*, *ray query* and *frustum query*. In a 3D tree, each node encloses an axis aligned cube which simplifies the implementation of the queries. A range query is an axis aligned box, while a ray query is an oriented cylinder and a frustum query is an oriented pyramid. Other types of queries may also be defined based on their application.

5 Related Work

Many distributed data structures have been proposed prior to this work [9, 5, 6], none of which have explicit notions of consistency. Like SinExTree, the distributed B-tree [3] is specifically designed for a data center, providing atomic operations with strong consistency, and was the main inspiration to our work. The B-tree is built over Sinfonia and memory allocations are done using bit vectors stored at memory nodes. However, the distributed B-tree performs poorly for workloads even with moderate write intensity, and lacks flexible queries. SinExTree scales for all types of workloads because SinExTree only validates leaf nodes affected by an operation, while the B-tree validates the entire path from root to leaf using compare items, which creates false sharing between operations touching different nodes. SinExTree is built over SinfoniaEx which handles memory maps internally, and with the help of SinfoniaEx's protected address space, error handling for invalid memory references is much simpler. Finally, a B-tree usually has a large fan-out, suitable for storing data, while SinExTree best fits applications requiring keys with multiple attributes and application-defined queries, and is unbalanced in nature.

Mercury [6] is designed to be a distributed data structure providing multi-attribute range queries. Mercury distributes key/value pairs in a p2p system across multiple ring overlays, each handling a specific key attribute, and each node in a ring handling a contiguous range of attribute values. Load balancing is achieved using random sampling techniques to relocate nodes in the ring, which required some nodes to gracefully leave one part of the ring and join another. SinExTree is specifically designed for a cloud environment, having operation latencies in orders of milliseconds. SinExTree supports general queries, unlike Mercury that only supports range queries, and provides strong consistency. Mercury has no notion of consistency. Load can be shifted in SinExTree between nodes much more efficiently, not requiring nodes to offload all their current load before being reassigned new load, a requisite in Mercury.

6 Conclusion & Future Work

We present SinExTree, a distributed SPT built over SinfoniaEx, which scales almost linearly with the number of machines hosting it. SinExTree provides strong consistency, while maintaining low latency for the majority of workloads, even in a virtualized commercial cloud environment (EC2).

7 Availability

A C implementation of SinfoniaEx and SinExTree is free software available at: `http://www.qstream.org`.

References

[1] Amazon s3. http://aws.amazon.com/s3.

[2] Apache cassandra. http://cassandra.apache.org.

[3] AGUILERA, M. K., GOLAB, W., AND SHAH, M. A. A practical scalable distributed b-tree. *Proc. VLDB Endow. 1* (Aug. 2008), 598–609.

[4] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07* (New York, NY, USA, 2007), ACM, pp. 159–174.

[5] ANDRZEJAK, A., AND XU, Z. Scalable, efficient range queries for grid information services. In *Proc. of the 2nd Int. Conf. on Peer-to-Peer Computing* (2002).

[6] BHARAMBE, A. R., AGRAWAL, M., AND SESHAN, S. Mercury: supporting scalable multi-attribute range queries. SIGCOMM '04, ACM, pp. 353–366.

[7] HAVOK PHYSICS. http://www.havok.com.

[8] OGRE. http://www.ogre3d.org.

[9] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11* (Feb. 2003), 17–32.

[10] TAYARANI NAJARAN, M., AND KRASIC, C. SinfoniaEx : Fault-Tolerant Distributed Transactional Memory. Tech. rep., University of British Columbia, Department of Computer Science, 03 2011.

[11] WHANG, K.-Y., SONG, J.-W., CHANG, J.-W., KIM, J.-Y., CHO, W.-S., PARK, C.-M., AND SONG, I.-Y. Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics 1* (1995), 343–349.