Replicated Indexes for Distributed Data

David Lomet
One Microsoft Way, Bldg. 9
Redmond, WA 98052
lomet@microsoft.com

Abstract

We describe a distributed index structure, in which data is distributed among multiple sites and indexes to the data are replicated over multiple sites. This permits good scalability as storage and accessing load are distributed over the sites and each site with an index replica has fast local access to the index structure, making remote requests at most for data at the leaves of the index tree. We call our method the dPi-tree because it is based on the Pi-tree. We replicate the index without the need for coherence messages. This works whether the index replica is persistent or a transient cached copy. We generalize a technique first used to provide recovery for Pi-tree indexes to independently and lazily maintain the index replicas. A further result is that each index replica is fully recoverable, an area not treated previously in replication schemes. We also show how the data in the leaves of the index can be distributed and re-distributed at very low cost.

1 Introduction

1.1 Scalability

Data mining, data warehousing, and the general escalation of the complexity of queries and the size of the underlying data put an increased premium on scaleable database systems. One aspect of that scalability is providing very efficient access to data as the size of the data explodes and the number of processors involved in accessing the data increases. This leads to an interest in distributed search structures in which the data is spread over multiple sites. This also leads to an interest in replicated search structures so that a substantial part of the computational cost of an access can be done locally, also reducing latency.

There are additional payoffs from data distribution and search structure replication. Data distribution enhances one's ability to access data in parallel, exploiting the set of processors at which the data resides. Index structure replication can, if done with real independence of index maintenance, greatly increase concurrency of access. Further, redundant index structures also enhance data read availability, as does replicated data in general. Availability of the distributed data does not become dependent on the availability of some single copy of the index stored on a possibly unavailable processor.

Consider two limited forms of distributed systems, client/server systems and cluster based systems.

Client/Server: Servers are typically very heavily used, and can easily become the bottleneck in system performance when accessing shared data. Replicated indexes permit all index traversal to be off-loaded to clients, with only the data leaf accesses remaining at the server. This improves response time and reduces network traffic as well.

Clusters: In clusters, any processor can play the role of both client and server. It then becomes possible to distribute the data over members of the cluster, while replicating the index at all members of the cluster. This improves upon client/server by spreading the data access load and permitting the accesses to proceed in parallel.

In both forms of distributed system above, caching has frequently been used to increase performance. We view cached index structures as transient replicas. Such transient replicas increase the desirability of our techniques for incrementally building replicas, as cached versions are continually being rebuilt.

The difficulty in exploiting distributed search structures is the update burden imposed. What search structure permits low cost maintenance when distributed over several processors? What search structure permits several replicas of its index to be easily and inexpensively maintained?

There has been recent interesting work on distributed and/or replicated search structures [2, 6, 7, 9, 10, 13]. This work attacks the cost of maintaining distributed and replicated search structures. The work of [9, 2, 13] is targeted at hashing structures. The DRT of [7] is a distributed (not replicated) binary tree.

1.2 Our Approach

Our system and communication models are similar to [6], and our search structure shares some structural and operation characteristics with their proposed dB-tree which is based on the B-link tree of [8]. Like the distributed DRT tree and RPs* of [10], we employ a form of "correction message". DRT, RPs*, and dB-tree all use lazy techniques to maintain distributed index trees over multiple processors.

We call our search structure the dPi-tree as it is a derivative of the Pi-tree of [11]. There, a correction mechanism was used to enable high concurrency and robust recovery for a centrally stored tree. When dealing with distributed and replicated trees, the correction mechanism requires messages, which we send lazily, and only when other information is requested. Correction is triggered in the same way as index recovery is triggered in [11], via detection of a misdirected search.

What distinguishes our techniques are their great simplicity and low cost, enabled by the *extremely* lazy way that we handle index structure maintenance. Unlike the dB-tree or an RP* tree, the dPi-tree does not require convergence of index replica node structure. No effort is made to keep the "replicated" index structures coherent- only the index content, i.e. the set of index terms, is important. All replica indexes must continue to effectively index and provide access to the underlying non-replicated data at the leaves of the index tree. Thus, the dPi-tree does not need to provide coherence messages in which one replica informs another of an index node split. The specific pagination of each index replica is managed solely by the replica itself, without reference to other replicas. Further, when data nodes of the index tree split, the posting of the index term for the new node in any of the index replicas can be very lazy since access to the data is always possible and the index term to be used can always be found via side traversals.

An additional bonus of our approach is that system failures can interrupt operation at arbitrary moments and the search correctness of the dPi-tree is not compromised. This is a natural consequence of the robustness of the Pi-tree, in which a system crash between the time a node split occurs and the time when its parent index node is updated does not effect search correctness, and the parent node is eventually updated during subsequent search operations. The dPi-tree inherits this robustness directly from the Pi-tree.

The fundamental insight that we exploit is that information about changes to the organization of data, its node splitting and its distribution, is propagated only on messages in which the exchange of data is required in any event. This is the same strategy as used with Pi-trees to ensure that interrupted splits eventually post an index term to the parent node. It is not unlike the correction messages of [7], but here such messages are always in response to explicit access requests. Each index replica can maintain itself based only on this information, and hence index replicas need not exchange messages to ensure structure convergence. Such convergence is not required. Our approach depends fundamentally on the unique structure provided by the Pi-tree, in particular, the presence of additional information on side links between sibling nodes.

The rest of the paper is organized as follows. In section 2, we describe the Pi-tree, emphasizing what it is that makes the Pi-tree unique, and sketching how that is exploited for concurrency control and recovery. Based on its very lazy concurrency and recovery technique, we describe in section 3 an index replication protocol that very lazily maintains dPi-tree replicas. How data can be distributed in a low cost way is described in section 4. The more complex re-balancing of data among processors of a cluster is also described there. It also exploits a very lazy approach to information propagation. We conclude in section 5 with a discussion of what has been accomplished and what might further be done.

2 Pi-Trees

2.1 Pi-tree Structure

The Pi-tree was introduced in [11] to provide a high concurrency index tree that also supported recovery. It was described then as a generalization of the B-link-tree [8] in that both search structures have side pointers connecting nodes at the same level on the search tree that are used to permit tree re-structuring that separate node splitting from the posting of the index term for the new node (see Figure 1.). The Pi-tree generalized B-link trees by permitting multi-dimensional index trees to exploit side pointers.

In one important sense, however, Pi-trees are not generalizations of B-link trees. It is that the links to sibling nodes of a Pi-tree are not purely pointers. Rather, they are the index terms that are to be posted to the parent node of the new sibling in directing the search to that node. This "indexed" side link is necessitated by the intent to support multi-dimensional searching. When traversing among the data nodes for data that is in some region of interest, a single dimensional search space trivially (by the nature of the search space) provides a search direction. With multi-dimensional index structures, the search direction of a

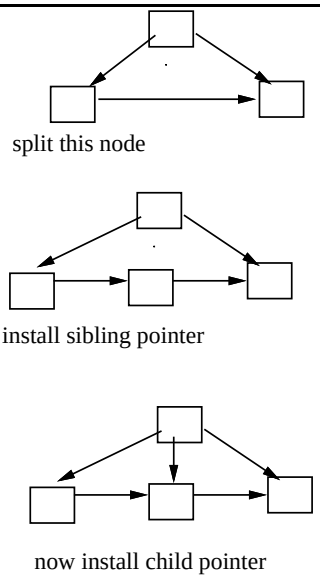

Figure 1: Side links permit node splits to be separated from index term postings because the new node is reachable via the side link without using the new index term. For Pi-trees, the side pointer comes with an index term describing the search space of the sibling.

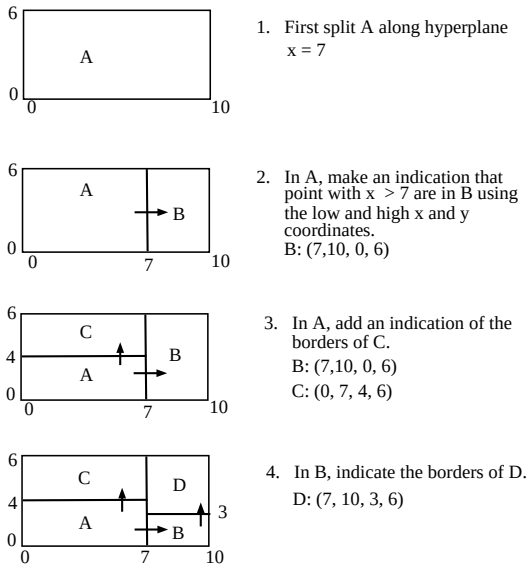

Figure 2: Space division and side pointers for a two dimensional search space are shown. Splits are by hyperplane with node space described via border coordinates. In a Pi-tree, nodes contain sibling terms for siblings split from them.

side link needs to be supplied explicitly by the index structure. One needs to know, e.g., that proceeding in the X direction exits a query range, but that additional items of the range exist in the Y direction. Hence, distinguishing X from Y in the links (i.e. having the links describe the space to which they refer) then becomes important. This division of the search space is illustrated in Figure 2.

More precisely, each node (within a single level of the Pi-tree) is given responsibility for a well defined part of the search space when it is created. The invariant that is preserved within the Pi-tree is that the entire space for which a node is responsible is reachable from that node. A node delegates responsibility for part of its search space to a new node during a node split. We call the original node the *container*, and the new node to which it has delegated responsibility for part of the search space the *extracted* node. To maintain the invariant that the search space for which it is responsible be reachable, the container must maintain an indexed side pointer that refers to the extracted node and identifies the space for which the extracted node is responsible.

An index node referencing a set of child nodes retains within its information the containment ordering among its children. This is trivial in the case of a one-dimensional B-link tree where simply ordering the index terms by the single attribute is sufficient. With multi-dimensional searching, more complex containment can arise. With the hB-Pi-tree, such containment is reflected in the k-d tree that comprises the index node. Containment order is essential in managing node deletion.

2.2 Concurrency Control and Recovery

While side pointers are important for range searching, their intent in the B-link-tree is to provide high concurrency during tree structure modifications. The Pi-tree extends this to also ensure that recovery of the search tree is possible should the system crash at any arbitrary time. The foundation for this is that side pointers give you multiple paths to the data nodes at the leaves. In particular, during a node split, an updater can gain access to a new node via a side pointer, even before its index term is posted in its parent node. This means that, e.g. a Pi tree, does not need to lock (latch) an index node (so as to permit the immediate posting of the index term for a new child) while its children nodes are being split.

The same kind of reasoning that permits concurrency enables one to separate the structure modification into two atomic parts for recovery. Atomic action #1 splits a full node and provides an indexed side link

to the new node in the original over-full node. Atomic action #2 posts the index term to the new node's parent. A system crash between the time a node has split and the time the index term for the new node is posted to its parent nonetheless leaves the tree well-formed, i.e., all data remains accessible and the structure continues to permit correct search by means of side index terms. The only thing that remains to be done is to ensure that this index term is eventually posted so that performance remains logarithmic in the size of the data. And this is exactly what the Pi-tree indexed side pointers permit.

For multi-dimensional search spaces, space decomposition can be quite complex. During a side traversal, one needs to know which of the possible directions a side pointer is taking the search. For "lazy" posting of an index term, one needs the exact description of the search space handled by the new node so that the index structure remains search correct. The Pi-tree provides this with indexed side pointers. When a side pointer is traversed, at any level of the Pi-tree, this occasion is used to post the missing index term. The sibling term's presence provides the information needed to update the parent. That is, the sibling term becomes the index (child) term posted to the parent node. Its posting is not an issue of search correctness, but solely of search performance.

Crashes may be quite frequent, index posting may be long delayed, and the Pi-tree index remains search correct because of the indexed side pointers. Indeed, the entire Pi-tree index between the root and the data leaves can be reconstructed via posting during side traversals. All the information needed to index the leaves is present in the leaves. This is not the case when the side pointers do not describe the search space of the new node. It is these indexed side pointers and this lazy index updating that we exploit in order to construct and maintain replicated and distributed indexes to the same data. And we do this with no need to keep the index replicas coherent, even eventually, and hence we can dispense with messages used to maintain coherent index structures [6].

3 Lazy Update of Replicated Indexes

This section describes how we deal with data node splitting which results, of course, from data being inserted into a leaf of the index structure. We begin with a primitive technique and proceed to a more effective but still very simple technique. We do not treat node consolidation (deletion) here, which we see as decidedly secondary. This is left to section 4.

We assume that the data that is being indexed is not itself replicated. We are only concerned, as was [6], with index replication, by which we mean "interior nodes", not leaf nodes. How data (leaf nodes) might be distributed among several processors of a distributed system is also discussed in section 4. Data distribution is orthogonal to the handling of the indexing structure and its replication. Replication of the index has an excellent cost/benefit ratio. The cost is low because the index is typically less than one percent of the file size. The performance benefit is large because the server is very effectively off-loaded. Only the data node search remains for the server, the client taking responsibility for the entire index portion of the search.

In the subsequent discussion, we are indifferent as to whether the replicated index is a persistent replica or simply represents a cached version of the interior nodes of the index.

3.1 The Primitive Technique

We describe initially a primitive technique that needs an absolute minimum of information from which to start and that maintains an index replica without knowledge of any other index. We describe this technique not because it is a practical method but to illustrates how little information is needed in order to maintain index replicas. Our pragmatic techniques exploit the existence of other indexes to more rapidly construct a replica with logarithmic search performance.

We assume that data has been entered into a collection of one or more data nodes and that these nodes are connected by indexed side pointers. We further assume that there is a pointer, which we call the *prime pointer(PP)* stored in a known location within our distributed system. The PP references the *prime node(PN)* which is responsible for the entire search space and which is connected to all other data nodes via side pointers, either directly or indirectly. There will usually be an existing index structure for accessing these data nodes as well, but our primitive index creation and maintenance method does not require or exploit it.

We pass the prime pointer to the site at which an index replica is desired. Search requests that arrive at that site trigger a traversal of the data nodes via side index pointers, starting with the data node referenced by the PP. Each traversal results in the posting of an additional index term to the index nodes maintained at the site. The first index term posted triggers the creation of an index root. Subsequent postings result in the index growing. Index nodes split at each site, completely independently of how they may be splitting at any other site. No coordination between the sites

is required.

Replicated Pi-tree indexes need never be coherent. They can contain different nodes, with different index terms and even have different heights. The completeness of a Pi-tree index "replica" reflects how many side traversals have been done via the Pi-tree rooted at a site. These side traversals include the original sequence whose update necessitated the splitting of a data node. Importantly, here, even the splitting of a data node does not result in a broadcast to all index replicas. Only the replica whose traversal resulted in the update and data node split is notified, and even that notification may fail. Information about splits or index traversals is never broadcast to all replicas.

3.2 Index Sharing Technique

The primitive technique above stubbornly refuses to learn from others. It insists on discovering, in its entirety, all the index terms that it will need for each new index replica. We can do much better by permitting a new replica to learn from a prior replica. This is particularly important when we are dealing with a cached index, as we then will be starting from an "empty" replica. The idea is to gain access to the index terms that have already been discovered by another index replica that we call the *basis*. The basis could be located at a data server.

Our sharing technique is initiated as follows. Instead of passing to the new replica a copy of the prime pointer, we rather pass to it a copy of the root of the basis index. In this way, the new replica receives a batch of information about accessing the underlying data. The new replica is a partial one, sharing some of the index structure of the basis. This is similar to the technique used in Exodus [1] for creating a new version, but in this case, the intent is to optimize creation of the index replica. Thus, instead of cloning the path to an updated data node, as done in Exodus, we clone paths as we proceed through the tree doing searches. This incremental replica creation is is shown in Figure 3.

When the replica starts processing search requests, it traverses its own private nodes as it would were it a completely independent replica. When a search comes to an index term that references an *index node* of the basis (or some other replica, since sharing can be recursive), it requests the acquisition of that node of the shared index. When the node is received, it is replicated locally and the pointer that originally referred to the remote shared node is made to reference the local copy of the node. In this way, a new replica acquires index terms in node size batches, permitting it to provide efficient logarithmic access to the data

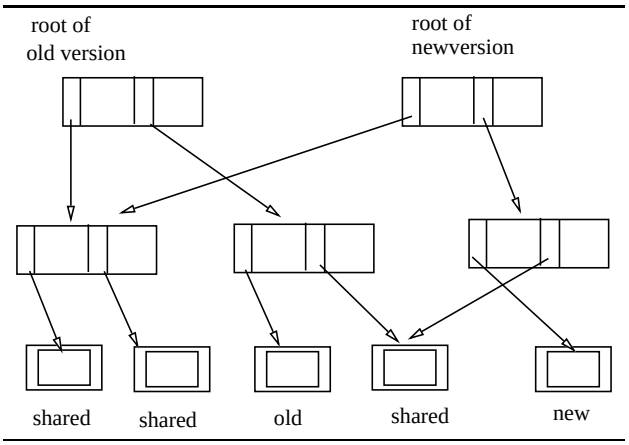

Figure 3: Exodus tree versioning has versions with unique roots but shared subtrees. For our distributed indexes, these versions are the local replicas. Each local replica starts with a copy of the root of a basis replica and initially shares the rest of the index with the basis.

immediately, assuming that its basis replica had such logarithmic access. This is a form of bulk correction message, i.e., where the index is grown a Pi-tree node at a time.

Within a cloned index node, the index terms carry with them a way of identifying the site at which the index nodes they reference are instantiated. A node that is returned to a requesting site and that will be replicated at that site will be in a form that is appropriate for the basis site. That is, all pointers to nodes in the basis site will be indicated as private pointers. But the replicated version of the node at the requesting site must treat these pointers as remote pointers to the basis site. Thus, our index pointers can no longer be simple local disk addresses. Note, however, that pointers to data nodes are unchanged by this optimization. These pointers, which are stored at the tree level immediately above the data level, constitute the vast majority of index terms, and they are unaffected by index replication. Only higher levels are impacted, which is much less than one per cent of the size of the replicated index.

Once again, there is no need for coherence messages. The index sharing technique is a performance enhancement that permits a more rapid construction of the replica. At any time after the replica has acquired a complete path to the prime node (PN), all index terms pointing to shared index nodes can be dropped without sacrificing search correctness or the ability to eventually complete the replica. However, shared index pointers continue to optimize the con-

struction of every unfinished part of the new replica. So retention of the shared index is of great value.

What we are describing is an optimization: whenever index node sharing is not involved, our new technique can default to the primitive behavior. In particular, whenever a side index term is traversed, a new index term is posted in the parent node. This includes side index terms within an index replica as well as for the data nodes. In this case, in addition to doing the traversal, we also copy the node encountered. The index term posted is thus to the copied node.

3.3 Message Minimization

The protocol above requires request/response messages every time a single dPi-tree node is transferred. Essentially, nodes are transferred one by one as the Pi-tree node search path is traversed. There is leverage in terms of reduced message traffic if multiple nodes can be transferred in a single message. The procedure below returns all the nodes and their associated index terms for the search path that is traversed at a shared replica. Indeed, we accumulate the search path as we return from lower parts of the tree and pass it to sharing replicas that requested data indexed in the dPi-tree. Since we know the search request, we can in fact do this. We proceed as indicated in Figure 4.

Again, what we have described is a form of correction message. But now, the entire Pi-tree path from an initial local replicated node to the data is corrected in one large batch operation. Thus, it only requires modest access activity from an index site for the index maintained there to provide very effective access to the underlying data. Note also, that unlike [7], a tree merge is not required. Rather, the path returned is appended to the existing replica by replacing a remote reference to a reference to a local copy of the path. Also, we deal easily with the paths that are missing from the replica as the replica is built. The index information in such missing paths is reached via either child pointers or via side pointers to nodes in the basis or to nodes shared by the basis.

Requests from a client site to a site with a shared index do not require re-traversal of the shared index starting from the root. Rather, the search simply continues down the shared index from the node identified by the requesting replica. This avoids unnecessary work and reduces the access burden placed on shared sites. Thus, if a requesting replica has reached level n of the tree (as measured by distance from the leaves), the shared replica proceeds from level $n-1$ onward. The search is not re-initiated at the root.

A shared replica, in trying to complete a search for a remote node, itself may become the remote node in

1. Replica A encounters a reference to a shared node N while processing a search request. Replica A sends a message to B (the owner of N) that identifies N and the search request itself.
2. Replica B receives the request from A and accesses N. It uses the search request from A to continue the search in N.
   - If the search leads to a data node reference, replica B returns node N's contents to replica A, where the node is replicated. In the replica at A, pointers in node N which had been to nodes local to B, become pointers to shared nodes that refer to replica B.
   - If the search leads to a node M at replica B, then the search is continued at B, adding the local nodes that it encounters in the search to the set of nodes that it will return to A. These include both child nodes and sibling nodes reached via side traversals.
   - If the search leads to a shared node M at yet another replica C, then B requests M from C and when C has returned the appropriate node or nodes that are further down the index tree, B concatenates those nodes with the nodes that it has traversed locally and returns the entire concatenated sequence to A. B also adds the nodes from C to its replica of the index.
3. Replica A replicates the returned sequence of index nodes in its local index replica, changing shared pointers to private pointers as appropriate. This conversion applies both to child pointers and to sibling pointers. It converts the remote pointer that triggered the remote request in step 1 to a local pointer to the first of the replicated nodes. It then continues the search to the data node P where one expects to find the data. This involves sending the search request and the address of P to the site where P resides.
4. Eventually (see the next section on data level organization), the data level responds with a list of side pointers traversed and the ultimate node Q that contains the data space specified in the request. Note that the data level does not send the contents of the traversed nodes, as these are not useful for maintaining the index replica.
5. Replica A adds the index terms returned to the appropriate node(s) of its index, performing structure modifications (i.e. node splits) as needed to store these new index terms.
6. Finally, the node returned (Q) is searched for the data requested in the search.

Figure 4: The procedure to build a replica index by adding to it all dPi-tree nodes in the path to the data as a correction message during an index search.

further processing of the search. This recursive process ensures that the shared replica has a complete path to the data prior to passing the path to its remote client replica. Note also that the data itself is not accessed by remote replicas holding the shared index nodes since the shared replica does not need the data itself. This saves a data access by the shared replica, putting that burden on the requesting replica.

3.4 "Seriously" Incomplete Replicas

A replica that is seriously incomplete will access multiple new nodes on the path to the data. This poses two problems.

- How do we keep the correction messages reasonable in size when multiple new nodes need to be returned?

  If the path traversed has too many nodes to comfortably fit into a single return message, this is not a large difficulty. Whenever results are returned that do not include the data requested, a pointer is returned to the node where the search may be continued. In the full optimization case, this will be to a data node. But should there be a large number of side traversals, this could be to a node at some remote index, either the basis replica returning the path, or to another. In all cases, the strategy is to include what is returned in one's own replica and to request a search continuation from the node referenced by this final pointer. Thus, how many nodes in the path to the data are returned to a requester can be very flexibly determined.

- How do we avoid long side traversal searches, which are a form of linear search, when a great deal of update activity may have moved the data of interest to a relatively remote node?

  We need a short circuiting mechanism that prevents such long side traversals if we make more than n side traversals in a search. A value of n that is approximately the height of the index tree should be reasonable. At that point, a particularly simple approach is to re-start the search at the root of the basis replica. Should we have the same problem with the basis, we re-apply the short-circuiting technique recursively. The result is to limit the number of node traversals to no more than $2 * height-of-tree * number-of-replicas$, though in practice, the number of node traversals should be very close to the height of the tree.

3.5 Shrinking the Replica

Not only can one grow the size of an index replica, but one can shrink it as well. This is the case for the same reason that we can be lazy in posting updates to the index. Data remains accessible via side pointers. Index size reduction is particularly important when dealing with transient (cached) replicas. This permits us to retain only the part of the tree that is active, and hence to maximize the effectiveness of the cache in reducing remote accesses. And this ensures that the prefix of the entire path to the data is always present at the replica.

It is almost always possible to delete an index term from a local replica, whether or not that index term points to a local node of the replica, a data (leaf) level node, or to a node that exists in the basis replica. Deletion of an index term must leave the containment structure of the index node well formed. The result is that the container for a node X whose index term we delete will appear to contain the space previously contained by X. Subsequent searches to the search space of X will be directed to X's container. Since the container has delegated responsibility for the space to X, sibling traversal from the container will lead to X. Such sibling traversals can be used to re-post the index term for X. Thus, deleting X's index term leaves the parent node of the dPi-tree in a state as if the container for X has been split with X being extracted, but before the index term for X has been posted.

We can delete an entire dPi-tree node N by dropping all its child index terms except for the index term to the child that serves as the containing node for the other children. At any point we can try to merge a node I with its container node. (And N will always contain at least this one child index term.) We insert the remaining contents of N into N's container node, including N's children as children of the container and N's siblings as siblings of the container. Sibling terms are never directly deleted. A sibling term is only removed when the node to which it refers is merged into the container. At this point the sibling index term referring to the node from the container is dropped.

Such deletion of index terms permits us to free space in a dPi-tree index node for more active parts of the tree. Dropping an entire index node permits us to reclaim space in the cache for other nodes. All information so dropped can be easily recovered should it be needed again. No coordination/coherence with other replicas is required for this. Subsequent reference to data in subtrees that are not represented in the index will result in the reconstruction in the local replica of the path to the data requested.

When we are dealing with transient replicas (cached copies of the dPi-tree index), a normal assumption is that no other replica is using it as a basis for replication. Thus, there are no other replicas with pointers to dPi-tree nodes of the transient replica. Hence, for these replicas, there is no need for coherence messages to cope with node deletion. Note here, however, that deleting nodes from an index replica is not simply a matter of dropping them and reaccessing them as needed, which is the way that caching is normally accomplished. Such simple node dropping is only possible if one were to retain an index term to a node in the basis. This is possible, but does not provide for the incremental construction of the replica and the complete independence among replicas that is so effective at reducing the search and update cost.

4 Data Level Organization

We have not yet addressed how the data itself might be managed by a data server, and how its updating and data node splitting is handled. That is the topic of this section. We describe two approaches for assigning data nodes to servers, one involving centralizing all data at a single server, the second involving distributing it among multiple sites.

4.1 Centrally Stored Data

The simplest way to manage the data, which is not replicated in our system model, is to assign all of it to a single data server. The data server itself should probably maintain an index to the data, from which future index replicas can be initialized. Our centralized data server maintains the data and the index for it very much as it would were there no index replicas involved. We describe here only the incremental difference that is made by the data server's need to cope with the index replicas and update requests that can come from multiple clients with replicas.

As with a search request, the data server receives an update request from a (index replica maintaining) client that identifies the data node at the server where the request activity is expected to occur. For a search or an update, this is the node that the client believes directly contains the portion of the search space desired. An update request is also accompanied by the new data and an indication of whether this is an update, insert, or delete.

The identified data node may contain the search space desired, or it may have delegated it to another node which can be accessed by side traversal from the originally identified node. Each side traversal results in a side index term being added to the message that will be sent back to the requester. These side pointers represent index terms that should be added to the client's index replica to make its index more complete. A data node split in order to provide storage to accommodate the update is simply a particular instance of this process. (The only additional work is that the data server itself will probably want to update its own index to the data.)

Our central server needs to perform concurrency control and recovery on the data that it manages. This can most easily be handled by the Pi-tree technique itself [11] but other techniques are possible so long as side index terms are maintained [4, 11, 12]. There is no guarantee to the clients that any data node that it may have read will remain valid while the client caches it (in the traditional way). The client must take out explicit locks, on either some of the records of the data node, or on the node itself, at the server. An obvious optimization is to piggyback lock requests on the messages that request the data nodes themselves.

4.2 Distributing Data Among Sites

4.2.1 Distribution Strategy

Supporting index replicas to permit the index search to be off-loaded to clients permits a substantial scale-up in the number of clients that can be served with the indexed data. However, the single central data server remains a potential bottleneck. To scale further, the data itself needs to be distributed among multiple servers. A few of the several ways the distribution might be done are:

Opportunistic: Choose the updating site as the site for the new node that is created by the split. This reduces communication costs because the updating site is already a participant.

Randomized: Choose the site for the new node based on a randomization process that uniformly distributes the load, such as via a hash function applied to the key. While not precluding the need for re-balancing, it is much reduced.

Range: Assign particular sites ranges of key values to store. The new node goes to the site handling the key range involved. This localizes range searching to a smaller number of sites.

Where the new node in a node splitting index modification resides at the same site as the original node, the techniques used for centrally managed data suffice. However, redistribution of data among the sites will undoubtedly be required at times. System configurations change, update activity skews the load, too many keys fall into one range. Indeed, unbalanced load is possible with essentially all data placement

policies without resorting to very substantial coordination message overhead. Re-balancing data between sites can be accomplished in terms of node deletion. To migrate a node, we perform a data node split", and move the entire contents of the old data node to the new node at a different site. This uses the node split protocol described below. Then we delete the old node. Clearly, the magic of this is in the handling of node deletes, the subject of section 4.3.

4.2.2 Distributed Node Splitting

For the tree restructuring resulting from a split to correctly survive a system crash, the action of splitting a node must be atomic. That is, within the same atomic action, two actions need to occur.

- The new node must be allocated and initialized with about half the contents of the old node.
- The old node must be updated so as to remove the contents now stored in the new node and have a sibling index term that now identifies the new node as the place where a search is to continue.

The new node and the old node may be co-located or at separate sites. In either event, we require atomicity. When co-located, such atomicity is easy to provide. When at separate sites, we require atomicity via a distributed commit protocol.

Two site atomicity can be achieved very inexpensively in this case. Site N writes the new node and prepares its part of the atomic action. N notifies site O that serves the old node that the old node has been split, the key value used in accomplishing the split, and the address of the new node. In the same message, N transfers commit coordination to O. O removes the data now in the new node from the old node, and inserts a side index link in the old node that refers to the new node at site N. O then commits the atomic action locally. N waits to receive a separate access request for the new node before committing its creation of the new node, thus using this access message as a lazy commit message. Should N wait too long, it asks O whether the action committed. It may need to do that in any event as commit messages may be lost. Site O remembers the commit status and can answer such inquiries by simply checking whether the old node has been updated with a side index link to the new node. This results in a normal case with only one message to both transfer information and to coordinate the distributed atomic action.

There is no commit processing overhead for site O. The overhead involves only site N's need for a durable "prepared" state which permits the new node to be deleted if site O does not commit. For index replica updating, a lazy message informs the updater of the node split trigged by its update. Other replicas will be notified during subsequent accesses. None of the index updating need be transactional as coherence is not required. The sibling index term will keep moved data accessible for search as well as providing the information needed for subsequent updating of the indexes.

4.3 Handling Deletes

4.3.1 The Deletion Structures

In order to keep the Pi-tree data level search correct when a data node N is deleted, we require the following actions to be atomically performed.

- Empty the deleted node N by updating N's "containing" node, i.e. the one with sibling index term referring to N, with the data and sibling terms of N so that all data for which the container is responsible remains accessible.
- Delete N and establish a tombstone for it containing a forwarding address to N's containing node. (We eventually garbage collect this tombstone.)

The strategy we pursue will preserve index correctness in a very lazy fashion. No index replica need be involved in a delete atomic action. A data node delete only requires an atomic action involving the site D with the node to be deleted and the site C with its container. These sites may have to handle accesses related to the deleted node for some time after the delete has been accomplished.

Site D keeps a tombstone for the deleted node N for as long as any index replica may have a pointer to N. The tombstone identifies the search space of N and contains a forwarding address to N's container, now responsible for the search space of N. This permits the system to distinguish invalid references to N from valid references to the same node when N is reused. All incoming accesses must check the tombstone table before accessing data nodes. This tombstone table must be persistent, but should be small enough to reside entirely in main memory while the index is being used. To keep it small, we garbage collect tombstones as described next.

4.3.2 Garbage Collecting the Tombstone

There is control information in each node describing every deleted node that the node has absorbed and that has an outstanding tombstone. This control information consists of a description of the search space

of the deleted node together with a bit vector with a bit for each index replica. The bit is *on* if that index replica may have a dangling pointer to the deleted node, and *off* otherwise. When all the bits have been turned off, site C tells site D that the tombstone can be garbage collected.

There will usually be zero, sometimes one, rarely more than one such deleted node whose tombstone a container node needs cope with. Further, we assume that the number of index replicas is modest, say fewer than 64, perhaps fewer than 16. Thus the control information for a deleted node might be no more than a word per deleted node with tombstone, and a few words at most. Also, since deletes are optional and require that the container accept data as well as control information from the deleted node, the container node can refuse to participate in a delete to limit the number of such deleted nodes.

We distinguish two kinds of data access to the search space that had been handled by the deleted node.

Accesses by way of the tombstone When an access request comes to site D, the request is forwarded to C (as if it were a side traversal) together with an indication that this request comes via the tombstone. Site C satisfies the request, and includes in its response not only the data requested, but the container node address and an indication that the deleted node has in fact been deleted. The requester then removes its index term referring to the deleted node. This is analogous to the lazy way in which node splitting and index maintenance is handled. (Note: Since the index term is not removed within a coordinated atomic action with the request, we can only say that we expect it to be removed.)

Accesses directly to the container When an access specifying a search space handled by the deleted node comes directly to the container, it is clear that the requestng replica does not have a dangling pointer to the deleted node. The bit associated with that index replica in the container node is turned off (if not already off). The request is then answered in the normal fashion. [1]

When the bits representing the index replicas have all been turned off, site C notifies site D that the tombstone can be garbage collected. This does not require a two site atomic action. Rather, site D simply drops the tombstone in a local atomic action. It then notifies C that the tombstone is gone. Should either message be lost, C will eventually ask D to delete the tombstone again and D will comply or simply report that this has been done already. [This exploits tombstone existence as testable state for the atomic action.] Once the tombstone has been confirmed as garbage collected, the information in the container node for the tombstone can be discarded as well.

[1] This assumes that once an index replica indicates that it knows about a node deletion, that no delayed request for the deleted node ever is made again from that replica. This requires some care by the replicas.

4.3.3 Additional Comments

The deleted node can serve as its own tombstone. The cost of this is that the node itself is not garbage collected until all replicas have referenced the container node directly. The benefit is that no separate tombstone table is needed, and there is no tombstone "look-aside" needed before accessing data nodes at a site. If deletion is sufficiently uncommon, this is a good strategy. If deletion becomes more common, then the need to reclaim space becomes more urgent, justifying a separate tombstone table.

Index node deletion for other than transient replicas(as opposed to data node deletion) does not appear to have much of a payoff. An index node will rarely become sufficiently empty because many data nodes must first be deleted. Further, each index is less than one percent of the size of the data so most of the gain from node deletion involves deleting data nodes. Index node deletion is a trivial task when no index sharing is involved (see section 3.4). However, supporting deletes with index sharing requires greater effort, which we do not describe, and complicates the sharing process itself.

5 Discussion

We have shown how to generalize an existing index structure, the Pi-tree, for index replication in a cluster-based or client/server system. The advantage of index replication is to off-load index traversals from the primary data server to the client or node performing the access. This also avoids the message overhead of a distributed search.

5.1 Very Lazy Replica Updating

The advantage of using dPi-trees over other index organizations is that dPi-trees enable a very lazy strategy for index maintenance. This exploits Pi-tree sibling index terms which were initially used to keep the Pi-tree recoverable while enabling high concurrency. No explicit (and separate) index coherence messages were required, and indeed, the node structure of the index replicas need not be coherent for the search to

remain correct. It is the redundancy of paths through Pi-tree indexes (both child and sibling index terms) that makes this possible. Thus, our replication strategy has less overhead than any of its predecessors.

If other methods can refer to their update strategy as "lazy", then we can justifiably describe ours as "very lazy". Separate coherence messages are never used in index maintenance for node splitting. A "correction message" is always piggybacked on the response to an access request, and we describe how to return the entire shared path from the current point in the local index to the remote data. This permits a local replica to update its index with large batch operations. The result is that no tree is likely to be more than a few accesses away from providing excellent indexing performance. Even when dealing with deletes, we take a lazy approach of using tombstones and notifying an index replica that a delete has occurred as part of a response to an access request. Only data level restructuring involving splitting or deletion requires coordination via atomic actions. And these actions involve only sites that host data nodes. Index only sites need never be involved.

Messages to inform replicas of new data nodes are required in all approaches. However, with our approach, the messages needed to provide index replica coherence are never needed as replica coherence is itself unnecessary. Delays introduced by the need to synchronize for index coherence are thus completely avoided. Finally, the simplicity of the strategy makes its implementation cost modest compared with methods requiring more strenuous coherence measures.

We suggest that node deletion only occur within the index level of the dPi-tree for transient replicas. Such deletes are strictly local and have no impact on other replicas when we preclude cached (transient) replicas as the basis for other replicas. Hence, no coherence strategy of any sort is required for them.

5.2 Concurrency and Recovery

Each replica supports high concurrency and recovery at its site, an issue not addressed previously. Index concurrency and recovery can be handled at a single site by means of the Pi-tree algorithm [11]. Multi-site coordination is not needed. Index node locking is needed only to provide synchronization among local updates to a replica. When a response to a remote requests needs to include a version of an index node, it is always acceptable to deliver a copy of an earlier version of the node since whether the node is out-of-date is not critical to our algorithms.

Data (as opposed to index) concurrency and recovery can be handled by a variant of the Pi-tree strategy. Typically, this would exploit low level physical redo and high level logical undo. Because data nodes can be split between sites, recovery needs to cope with this distribution. But this is a standard part of distributed actions and poses no additional difficulties.

5.3 Range Searching

When a search request is for a range of values, a distributed search structure can execute the request in parallel. This is a very important aspect of index tree performance. During a range search, all children nodes of an index node that are within the range can be accessed in parallel. When the index is to distributed data, different nodes can return the data within the range while executing in parallel.

The search requesting node then needs to determine when the range search is complete. The existence of side pointers for dPi-trees makes this easy. We need compute twi sets as we perform the search.

nodes needed Addresses for data nodes within the range that are at other processes. This is initialized to the data nodes that are immediate children of the index nodes in the range.

nodes searched Addresses for all nodes that are within the range and that have been searched already.

Each process involved in the search returns (1) its data in the range, together with (2) the list of data node addresses in which the data was found, which is added to "nodes searched", and (3) the list of side pointers to nodes that are part of the range but that are located elsewhere, which are added to "nodes needed". When nodes searched equals nodes needed, the range search is complete.

5.4 Data Nodes

How to deal with the data at the leaf level of an index tree is largely orthogonal to index replication. We demonstrated lazy methods that are in the same spirit as our lazy index replication, to deal with distributing the leaves across multiple sites. We also showed how to lazily handle node deletes. These methods avoid the introduction of coherence messages. However, there is more complexity involved as splitting and deleting nodes requires atomicity, and distribution increases the cost and complexity of atomicity. We use a tombstone technique for coping with dangling references to deleted data nodes. Hence, lock coupling is not required when accessing the dPi-tree since its only function is to provide assurance that the reference found at one level of the tree is not dangling when used to access the next level.

5.5 Generality

The laziness of our method is enabled by the fact that Pi-trees maintain not only a side pointer to sibling nodes, but an entire side index term with both space description and pointer [11]. It is this index term that provides the source for lazy updating of index replicas as all replicas eventually store this index term. This is a very powerful paradigm for lazy distributed search structure maintenance that should find use in other distributed and/or replicated search structures, e.g. hashing or grid like structures. What we do require is that the search space for which a node is responsible not increase since we deal with node deletes in a rather special and less efficient fashion. This precludes the direct application of our techniques to spatial search using the R-tree [5]. We feel, however, that the hB-Pi-tree [3] is a more robust search structure in any event, and our techniques apply immediately to it.

Acknowledgments

Witold Litwin provided a valuable critical reading of an earlier version of this paper. In addition to correcting misunderstandings about existing distributed indexing methods, he raised two issues: bounding the worst case number of side traversals (see section 3.4) and parallel search for range queries, with its requirement to determine when such a query is complete (see section 5.3).

References

[1] Carey,M., DeWitt, D., Richardson, J., and Shekita, E. Object and File Management in the EXODUS Extensible Database System. *Proc. Very Large Databases Conf.*(Sept. 1986) 91-100.

[2] Devine, R. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. 4th Int'l Conf. on Foundations of Data Organization and Algorithms. (Oct. 1993) Evanston, IL

[3] Evangelidis, G., Lomet, D., and Salzberg, B. The hB-Pi-Tree: A Modified hB-tree Supporting Concurrency, Recovery, and Node Consolidation. *Proc. Very Large Databases Conf.*(Sept. 1995) Zurich, Switz. 551-561.

[4] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques* Morgan Kaufmann (1993) San Mateo, CA

[5] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *Proc. ACM SIGMOD Conf.*(May 1984) Boston, MA 47-57.

[6] Johnson, T., and Krishna, P. Lazy Updates for Distributed Search Structure. *Proc. ACM SIGMOD Conf.*(May 1993) Washington, D.C. 337-346.

[7] Kroll, B. and Widmayer, P. Distributing a Search Tree Among a Growing Number of Processors. *Proc. ACM SIGMOD Conf.*(May, 1994) Minneapolis, MN 265-276.

[8] Lehman, P., and Yao, B. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. on Database Systems* 6,4 (Dec. 1981) 650-670.

[9] Litwin, W., Neimat, M-A, and Schneider, D. Linear Hashing for Distributed Files. *Proc. ACM SIGMOD Conf.*(May 1993) Washington, D.C. 327-336.

[10] Litwin, W., Neimat, M-A, and Schneider, D. RP*: A Family of Order-Preserving Scaleable Distributed Data Structures. *Proc. Very Large Databases Conf.*(Sept. 1994) Santiago, Chile

[11] Lomet, D. and Salzberg, B. Access Method Concurrency with Recovery. *Proc. ACM SIGMOD Conf.*(May 1992) San Diego, CA 351-360.

[12] Mohan, C. and Levine, F. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. *Proc. ACM SIGMOD Conf.*(May 1992) San Diego, CA 371-380.

[13] Vingralek, R., Breitbart, Y., and Weikum, G. Distributed File Organization with Scaleable Cost/Performance. *Proc. ACM SIGMOD Conf.*(May, 1994) Minneapolis, MN 253-264.