A parallel index for semistructured data

Brian F. Cooper*
Dept. of Computer Science
Stanford University
Stanford, CA 94305 USA
cooperb@stanford.edu

Neal Sample†
Dept. of Computer Science
Stanford University
Stanford, CA 94305 USA
nsample@stanford.edu

Moshe Shadmon
RightOrder Inc.
3850 N. First St.
San Jose, CA 95134 USA
moshes@rightorder.com

ABSTRACT

Database systems are increasingly being used to manage semistructured data, which may not have a fixed structure or set of relationships between data items. Indexes which use tree structures to manage semistructured data become unbalanced and difficult to parallelize due to the complex nature of the data. We propose a mechanism by which an unbalanced *vertical* tree is managed in a balanced way by additional layers of *horizontal* index. Then, the vertical tree can be partitioned among parallel computing nodes in a balanced fashion. We discuss how to construct, search and update such a horizontal structure using the example of a Patricia trie index. We also present simulation results that demonstrate the speedup offered by such parallelism; for example, with three-way parallelism, our techniques can provide almost a factor of three speedup.

1. INTRODUCTION

Data is rarely flat. Instead, a database can contain many relationships between individual data elements, and these relationships can be as important as the data itself. Increasingly, organizations must deal with data that is *semistructured*: neither the structure of data elements nor the set of relationships between them is fixed in advance. Thus, a database must allow users to efficiently access data despite irregularities in the structure. One solution is to represent relationships between data items in a hierarchical manner, using a tree structure. This tree can allow users to find objects, as well as objects related to them. Instead of having to join together separate datasets, we can conduct a top down traversal of the tree. Subordinate objects are stored in a subtree, such that all the leaves of the subtree represent all of the related, subordinate items.

We could partition portions of such a relationship tree between parallel computing nodes so that the index is distributed and can answer many queries simultaneously. For example, Figure 1(a) shows a relationship tree, where each vertex represents an object and each edge represents a relationship. As with any tree, the upper layers near the root (labeled α in the figure) are likely to be small compared to the lower layers near the leaves (labeled β). Therefore, we can adopt the partitioning scheme shown in Figure 1(b), where the upper layers are replicated to every computing node and the lower layers are partitioned between nodes.

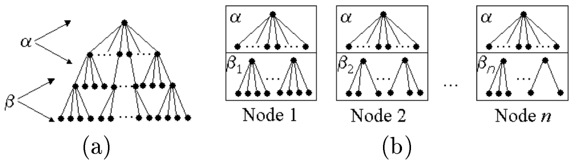

Figure 1: Partitioning a relationship tree.

In this configuration, a user can submit a query to any node i. The i node will traverse the copy of the upper layers (α) of the index that it holds locally until it reaches the lower layers ($\beta_1, \beta_2 ... \beta_n$). Then, i will forward the query to node j that holds the lower layer partition (β_j) necessary to answer the query. If $i = j$, then no forwarding will be necessary and i will answer the query directly. This approach is called the *forwarding* strategy. Alternatively, we can submit the query simultaneously to all nodes $(1, 2 ... n)$. Each node will traverse α to determine if it locally holds the correct β_j. If so, that node will answer the query. If not, that node will simply discard the query. This approach is called the *parallel issue* strategy.

Once we have achieved an even partitioning of the tree, we can use traditional techniques to manage parallelism. For example, we can utilize replication and voting schemes to ensure consistency after updates, and provide failover and availability (e.g. [3, 11, 7]). The benefits of partitioning go beyond the speedup offered by parallelizing computation. In addition, the partitioning allows us to construct large indexes while keeping the resource requirements for any one computing node small; for example, we can build a terabyte-sized index from a collection of commodity parts such as 50 gigabyte disks distributed among 20 off-the-shelf computers.

Unfortunately, with semistructured data the relationship tree structure can become unbalanced. Data elements with many complex relationships cause portions of the tree to become deep and branchy, while elements with simpler relationships reside in shallower, more linear portions of the tree.

*Work done while author was at RightOrder Inc.
†Work done while author was at RightOrder Inc.

SAC 2002, Madrid, Spain

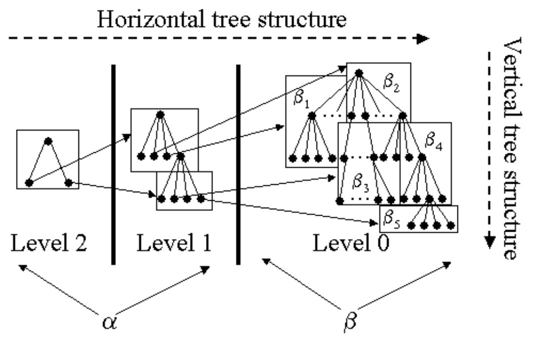

Figure 2: A layered index tree.

This makes it difficult to partition the tree evenly among parallel computing nodes to support parallel execution of queries. As a result, nodes assigned to complex portions of the tree become heavily loaded, while nodes with simpler subtrees are lightly loaded and underutilized.

1.1 Balancing an unbalanced index

To achieve parallelism, we require an algorithm to introduce properties of balance into an unbalanced data set. We have created an indexing mechanism, the Index Fabric structure, that deals with the problem of balancing an uneven structure to allow for partitioning among parallel nodes. The Index Fabric structure is an instance of a *layered index*: the tree is divided into equally sized blocks, and then extra levels of index are used to address these blocks. These extra levels are *horizontal levels* in contrast with the *vertical* structure of the normal relationship tree, as shown in Figure 2. Searches proceed along the horizontal structure to a localized portion of the vertical tree. The leftmost levels (e.g. level 1, 2 ...) act as indexes over the blocks in level 0, and even if the number of level 0 blocks is large the leftmost levels will be relatively small. Thus, we can replicate the leftmost levels to every computing node while partitioning the rightmost level, much as we did with the α and β in Figure 1. A query can be submitted to any node, which navigates the leftmost levels to find the correct level 0 partition, and then forwards the query to be answered by a single node in the parallel network. Similarly, queries can be issued to every node, and the node with the correct β_j will answer the query.

The challenge in constructing such an index is to build a balanced horizontal structure, and this is the issue we confront in this paper. As a concrete example, we will discuss how to partition a Patricia trie [13], a compact but potentially unbalanced tree structure for indexing string data. The Patricia trie allows us to represent relationships between data items by concatenating strings. For example, to indicate that $item_1$ is related to $item_2$, we insert the key $item_1 item_2$ into the trie. The result is that all items $item_2, item_3 ... item_n$ related to $item_1$ form a subtrie within the Patricia trie. More details of this representation can be found elsewhere, in [5]. This relationship structure is a natural fit for semistructured data, which often contains complex relationships between data items. Querying paths in semistructured data, such as XML, requires traversal of this relationship structure, and this traversal is supported by our index.

1.2 Contributions and overview

We have implemented the Index Fabric, and have used it to answer queries over semistructured data, particularly XML. The encoding and query processing strategies are discussed elsewhere [9]. Here, we focus on the balancing, partitioning and parallelization of the index, and present experimental results that examine the speedup offered by our techniques.

Specifically, we make the following contributions:

- We present a balancing strategy for an unbalanced index over semistructured data.
- We examine a partitioning strategy whereby some levels of the balanced tree are replicated to all parallel nodes, and some are partitioned among nodes.
- We present simulation results that illustrate the speedup available under different parallelization strategies. For example, our techniques can provide a factor of three speedup for a three-way parallel system.

This paper is organized as follows. In Section 2 we describe how an essentially unbalanced structure (such as a Patricia trie) can be balanced using a layered index. Then, in Section 3 we examine searching this layered index structure. Section 4 discusses updating the index, and in Section 5 we examine simulation results. In Section 6 we examine related work and Section 7 summarizes our conclusions.

2. BALANCING AND PARTITIONING TRIES USING LAYERED INDEXES

Tries are data structures for storing string data. A vertex in the tree corresponds to a position in a stored string, and edges emanating from the vertex are assigned to the possible characters in that position in the string. Tries are searched by comparing the appropriate character in the search key with the outgoing edges at each vertex, and following the proper edge to the next vertex until a leaf containing a pointer to a data record are found. *Patricia* tries are a compressed form of tries, and contain only the vertices that have two or more children. In this way, differences between inserted keys are represented instead of the whole keys. An example is shown in Figure 3(b), which is a Patricia version of the full trie in Figure 3(a). The numbers in the vertices (the *depth* of the vertex) indicate the character position in the string to compare to the labels on the outgoing edges. For example, if we are searching for the string *key*, when we reach the vertex labeled "2," the character at *key*[2] should be compared to the outgoing edges. A Patricia achieves compression at the cost of no longer storing the complete keys.

We can divide a Patricia trie into roughly equally sized subtries, and store each subtrie in its own block. For example, we can take the index in Figure 4(a) and divide it. Thus, in Figure 4(b), the original index still exists (at level 0), but has been split into two blocks. The block boundaries are indicated by dotted lines ········. The vertex at level 1 now acts as an index over the blocks in level 0; specifically, the trie in level 1 indexes the *common prefixes* of each subtrie stored in the blocks in level 0. The common prefix is the prefix of all vertices in the subtrie of a particular block, and

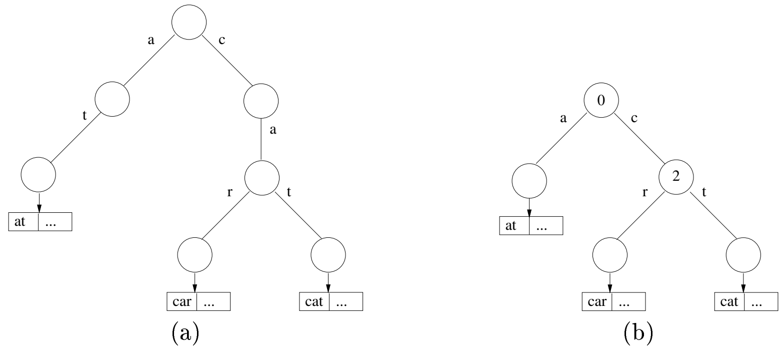

Figure 3: Tries.

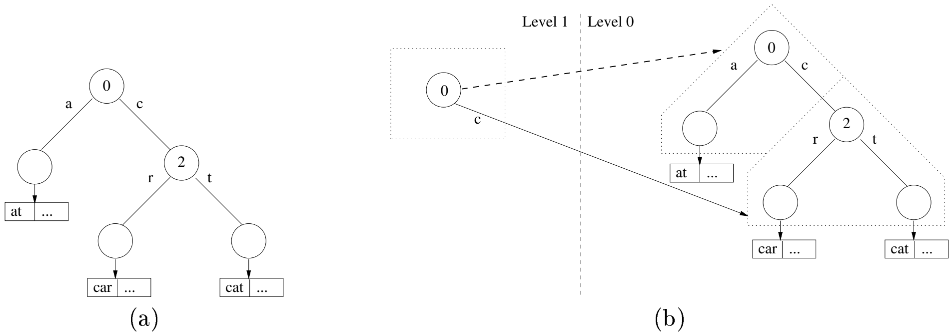

Figure 4: Partitioning tries.

is the portion of the string represented by the root vertex in the block. In Figure 4(b), the upper block has a common prefix that is empty, while the lower block has a common prefix that is "ca". For the rest of this paper, we will refer to the leftmost horizontal layers (e.g. layers 1, 2 ...) as "upper levels" and the rightmost layer (e.g. the basic partitioned index structure in level 0) as the "lower level."

An upper level index refers to the lower level in two different ways. The first way is called a *far* link, and is shown Figure 4(b) as a regular arrow $\longrightarrow$. This link is the same as an edge between a parent and a child in a normal trie, except that the parent is in level $i+1$ and the child is in level i. Thus, a far link has a label; in the figure, the label is "c". In contrast, the other type of link, called a *direct link*, has no label. This kind of link is represented as a dashed arrow ⇢ in Figure 4(b). Direct links connect vertices that represent the same key prefix but are in different levels. In the figure, the prefix represented by both vertices labeled "0" is the empty prefix (although any prefix can be represented.) Direct and far links are necessary for searching the layered index trie structure as described below. We can refer to "normal" edges between a parent and a child within the same block, as *near links*. Edges that cross block boundaries within the same level are called *split links*.

Tries can have more than two levels. In fact, this structure can be generalized to as many levels as needed. Each level i contains a complete trie divided into blocks. Level i is indexed by level $i+1$ such that the trie in level $i+1$ indexes the common prefixes of level i. In Figures 4 we illustrate increasing levels $i, i+1 \ldots n$ from right to left. The leftmost or "highest" level contains the root of the horizontal index, and has only one block. This root block contains a complete Patricia trie and therefore may contain several trie vertices in order to index several blocks at the next level.

In practice, the number of horizontal levels in the tree remains small, even as the number of indexed keys becomes very large. The Patricia requires very little space per indexed key, leaving a great deal of room in a block for pointers. Consequently, the horizontal tree has a very high branching factor and a correspondingly low height. For illustration, we can make simple assumptions about the sizes of blocks and pointers (e.g. 8 Kb blocks and 32 bit pointers) and store 1,000 pointers per block. The root block (level n) can then index 1,000 blocks at level $n-1$, and each $n-1$ block can index 1,000 blocks at level $n-2$, and so on. Thus, a three level tree can index a billion items. The uppermost levels (level $n > 0$) are very small compared to the leaf blocks (level 0). In fact, the upper two levels of a three level tree occupy less than 10 Mb of space, and can easily be stored in main memory. As a result, the upper levels can be replicated at all computing nodes, and only the blocks in the lower layers must be partitioned among nodes.

The number of blocks in the lowest layer of the index is determined by the size of the trie and the assigned block size. We could make blocks large, so that each computing node is assigned a single block. Another possibility is to make blocks small, and assign multiple blocks to each node. This latter approach has the benefit that adding or removing computing nodes to the parallel network requires transferring a few small blocks as opposed to repartitioning the entire index with a different block size.

```
address SEARCH(key x, starting block B) {
    while B is not a pointer to data do
        B =SEARCH_BLOCK(x, B);
        if B = NULL then
          return NULL;
        if B is a pointer to a block not cached locally then
            Invoke SEARCH(x,B) at node i holding B;
            Return marker indicating node i will answer query;

    if key(B) = x then
      return B;
    else
      return NULL;
}
```

Figure 5: Procedure SEARCH.

```
address SEARCH_BLOCK(key x, block B) {
    n =root vertex of B;
    i = depth(n); /* The depth of the vertex */
    B' = NULL; /* This will be the block in the next level */

    while B' = NULL and i < length(x) and
      a near or far edge n → n' labeled x[i] exists {
        if n → n' is a far link then {
            /* n' is in the next lower level */
            B'=address of block pointed to by edge n → n';
        } else { /* n → n' is a near link */
            n = n';
            i = depth(n);
        }
    }

    if B' = NULL and there exists a direct edge n → n' then
      B' =address of block pointed to by direct edge n → n';

    /*at this point, B' may be a pointer to another block,
    a pointer to data, or NULL */
    return B';
}
```

Figure 6: Procedure SEARCH_BLOCK.

3. SEARCHING A LAYERED INDEX

The search procedure for the layered index index closely parallels the search process for a simple trie. Figure 5 shows procedure SEARCH, which returns the disk address of the data matching the search key, or NULL if no such data exists. When a query is issued, the starting block argument B is set to the root block of the layered index trie (e.g., the block in the leftmost horizontal level). Basically, the search starts at the top level and proceeds both vertically (within the same level) and horizontally (between levels). A search at level i proceeds until it is appropriate to follow a far or direct link to the next level (level $i-1$). Then the search resumes at the trie in level $i-1$. For all levels except the leaf level ($i=0$), procedure SEARCH calls procedure SEARCH_BLOCK (Figure 6) to search within a particular block. The result of SEARCH_BLOCK is either a pointer to another block, a pointer to data, or NULL.

We can illustrate the search algorithm using Figure 4(b). Imagine that we are searching for the key "cat." Procedure SEARCH begins by finding the root block, which is the block in level 1, and searching this block with procedure SEARCH_BLOCK. The root vertex in the block is examined, and has a depth of "0." Thus, $key[0]$, or "c," is compared to the outgoing edges. The far link from the root vertex has the label "c", and the search then proceeds to the block pointed to by the far link. This block is the level 0 block in Figure 4(b) containing a trie rooted at a vertex of depth "2." Procedure SEARCH therefore invokes SEARCH_BLOCK again. This procedure begins by examining $key[2]$, or "t," and follows the near link labeled "t." This takes the search to its goal, the vertex which points directly to the data with key "cat."

The effect is that the search starts in the high horizontal levels, and then proceeds directly to a localized portion of the vertical trie (in level 0) that contains the desired data. Every search accesses the same number of levels; moreover, it is only necessary to enter a single block per level. As a result, any node containing a replicate of the upper levels can perform the first portion of the search. However, once it becomes appropriate to enter the lowest layer, the query must be answered by the specific computing node assigned to that block. This computing node can answer the query completely, since a single block is needed in level 0. (See Section 3.1.)

Note that because of the compression achieved by the Patricia trie, procedure SEARCH may return data for a key that does not actually exist in the index. For example, in the index in Figure 4, the key "cut" does not exist. However, if "cut" is specified as a search key to procedure SEARCH, then the return value will be the address of the data with key "cat." This is an unavoidable consequence of the fact that Patricia does not store full keys. Therefore, the found data should be checked to verify that the search key matches the data key; this is done at the bottom of procedure SEARCH. There are other complications introduced by the compression that we do not address here due to space limitations, see [5].

3.1 Searching a parallel layered index

There are two strategies for searching a layered index in parallel. In the *forwarding strategy*, procedure SEARCH is executed at any compute node holding a cached copy of the upper levels of the layered index trie. Queries can be issued to nodes in any reasonable way, e.g. randomly or in round-robin order. The node i receiving the query traverses as much of the layered trie as it can until it reaches the partitioned lower layers. The i node checks to see if the required block is available locally. If so, the i node answers the query. If not, the query is forwarded to the compute node j that does have the block. The j node executes procedure SEARCH, but starts at the appropriate lower layer block instead of the root. The j node can then answer the query.

In the *parallel issue* strategy, queries are issued to all nodes simultaneously. All nodes traverse the upper layers of the layered trie until they reach block B_L in the leaf layer. One node has been assigned the partition containing B_L; this node continues the traversal in B_L until it answers the query. All other nodes simply discard the query.

4. INSERTING DATA

Keys are inserted into the layered index trie using a three step process. First, a search is performed to locate the level 0 block that must be updated. Next, the key is inserted in this block using the normal Patricia insertion algorithm.

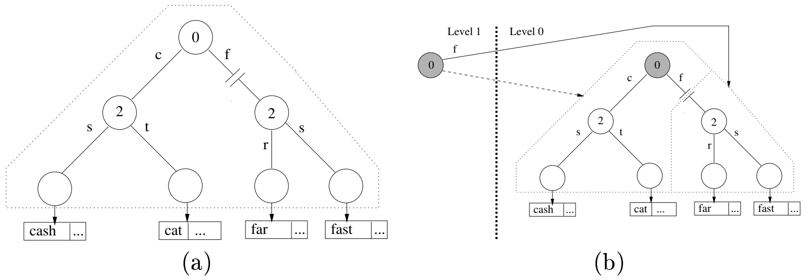

Figure 7: Splitting a block.

This may involve creating one or more vertices in an existing block. Third, if after creating new vertices there are now too many vertices to fit in the block, the block must be split. Most insertions will not require a split and can be wholly handled by the computing node assigned to the affected block. It is only when a split occurs are the upper levels affected and all nodes must be notified.

Every block in the index contains a connected subtrie. The splitting operation must preserve this property. Therefore, in order to split we must select a *split edge*, such that splitting the block's subtrie on this edge produces two connected subtries. One subtrie remains in the existing block, while the new subtrie must be placed in a newly created block. Both subtries resulting from the split should be of approximately equal size to maintain good space utilization of the disk blocks. It is often impossible to select a split edge that produces two subtries of exactly equal size. In this case, a split should be selected which is closest to this ideal. Figure 7 illustrates the splitting process. Figure 7(a) shows a trie, with the split edge marked. After the split, this trie occupies two blocks, as shown in Figure 7(b).

After the split is completed, a pointer to the new block must be added to the level 1 trie. We take the *split vertex*, which is the parent vertex on the split edge, and copy it to the level 1 trie. This vertex retains its *depth*. The copy of the split vertex in level 1 is given a *far* pointer, with the same label as the split edge, pointing to the new block. The copy of the split vertex is also given a *direct* (unlabeled) pointer referring to the old block. The result is shown in Figure 7(b), where the shaded vertex, labeled "0," has been copied to level 1. Finally, the newly copied split vertex must be connected to any existing vertices in the level 1 block to preserve the property that each block has a connected subtrie. This may involve converting a far link to a near link and copying additional nodes from level 0 to level 1. The complete process is described in [5].

If copying the vertices to level 1 causes a level 1 block to overflow, that block must be split as well. This is done in the same way as the split at level 0, and this process can contine at higher levels in the index. Eventually, it may be necessary to split the block at the root level (level n). If so, then a new level (level $n+1$) is created. This is how the layered index grows horizontally.

5. PERFORMANCE RESULTS

We have examined the performance of our partitioned index by running simulation experiments. We took a block trace from an implemented, single-node version of the Index Fabric, and fed the trace to a simulated parallelized version of the index. In this way, we can examine the effect of partitioning and parallelization on a real workload.

5.1 Experimental setup

The Index Fabric was used to index XML documents from the DBLP, the popular computer science bibliography [1]. Each XML document corresponds to a single publication. The database contained over 180,000 documents, totaling 72 Mb of data, grouped into eight classes (journal article, book, etc.) The documents were stored in a popular commercial relational database system, and the Index Fabric was used instead of the database system's native query processor. We ran a series of 10,000 queries over this database; each query required one index key lookup. The details of the indexing strategy, as well as performance numbers in the non-parallel case, can be found in [9]. Here, it is sufficient to note that the Index Fabric provided up to an order of magnitude speedup versus the database system's native query processor.

From these experiments, we collected a block trace which indicated the order of index block requests, and whether those blocks corresponded to leftmost levels (e.g. α in Figure 2) or the rightmost level (e.g. β in the same figure). This block trace was fed to a simulator. The simulator constructed a scenario where the index was partitioned among multiple nodes, and reported results about the number of queries that could be answered by each node that was issued a query. The partitioning was performed by randomly assigning blocks to nodes. In this way, we could compare the benefit of the *forwarding* (with queries issued in round-robin fashion) and *parallel issue* parallelization strategies (see Section 1). We compared against the base case of no parallelization (e.g., one node).

We assumed the following parameters in our simulation:

- The time to process the α layer (in memory) for a query is 1 unit.
- The network cost to forward a query to another node is 4 units.
- The time to read a β block from disk to answer the query was 10 units.
- If the β block was in cache, then it could be processed in memory without a disk read, and thus the cost was 1 unit.

These costs reflect standard assumptions about latencies, e.g. network latencies are about 4 times memory latency, and disks are about an order of magnitude slower than memory.

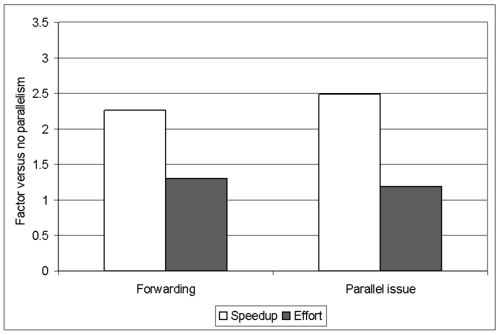

Figure 8: Performance for parallel index with no cache, 3 nodes.

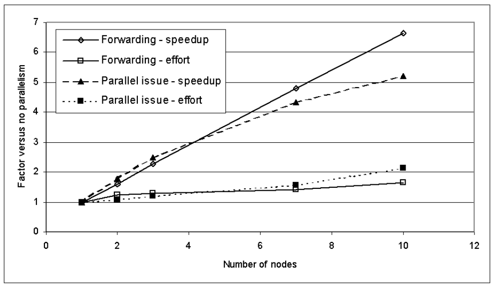

Figure 9: Performance for parallel index with no cache versus number of parallel nodes.

5.2 Results

The first experiment we ran examined the impact due to partitioning the index among parallel nodes. In this experiment, cache sizes were set to zero to eliminate cache effects. We compared the forwarding and parallel issue strategies against the base case of no parallelism. In the forwarding strategy, a node must be chosen to receive each query; we issued queries to nodes in round-robin fashion. The results are shown in Figure 8 for the case of 3 nodes. This figure illustrates two metrics. *Speedup* is the ratio of the time to answer queries for the parallel index versus the time for the non-parallel index. *Effort* is the ratio of the total units of work done by the parallel index to the total units of work done by the non-parallel index. Speedup thus represents the increase in the throughput due to parallelism, while effort reflects cycles "wasted." In the forwarding strategy, cycles are wasted by the forwarding of queries, and in the parallel issue strategy, cycles are wasted by nodes that are issued a query but discard it because they do not hold the correct β_j. An effort value of 1 indicates the parallel index did as much work as the non-parallel index, while any value over 1 indicates the proportion of wasted work.

Figure 8 illustrates that both strategies offer more than 2× speedup. However, it is clear that the parallel issue strategy offers the best throughput. The effort "wasted" by a node that examines a query but does not answer it is less than the effort "wasted" by forwarding queries. This makes sense, given that the cost of a forward is 4 units, but the cost for two nodes to examine their local α copy is only two units.

Next, we examined the speedup and effort as the number of parallel nodes changed. Figure 9 shows the results, with the number of nodes along the horizontal axis. This figure shows the speedup and effort for both the forwarding and parallel issue strategy. As the number of nodes increases, the speedup under the forwarding strategy increases linearly, while the speedup under the parallel issue strategy increases less quickly. In fact, when there are four nodes, both strategies are equally fast, but beyond four nodes, the forwarding strategy is best. Recall that in our cost model, the cost to forward a query is 4 units, while the cost to process an α layer in memory is 1 unit. Under forwarding, the only "wasted effort" is work done to forward a query, while under parallel issue, the wasted effort is done when nodes process the α layer but then discard the query. Thus, at four nodes, both strategies are doing roughly equal amounts of "wasted" work, while at more than four nodes, forwarding does less wasted work than parallel issue. The result is better use of computing resources under the forwarding strategy, and thus more speedup versus parallel issue, when there are more than four nodes. This result depends on the ratio R of the cost to forward a query versus the cost to process an α layer. However, the general result remains the same: forwarding provides more speedup when there are at least R nodes.

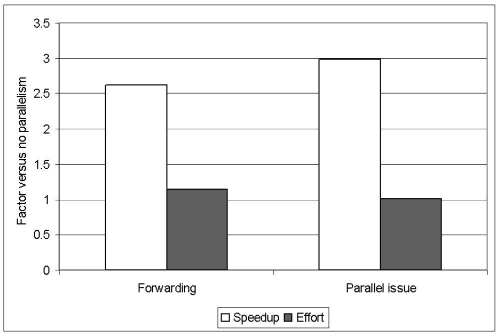

Figure 10: Performance for parallel index with caching, 3 nodes.

Another benefit of parallelism is the ability to construct systems that have a large amount of resources from smaller, cheaper components. In this context we can construct a system that has a large amount of aggregate cache from components that themselves have a small cache. To examine this effect, we ran another experiment, where we assumed that an individual node could devote enough memory to cache one percent of the total index size. The results for three nodes are shown in Figure 10. This figure demonstrates that the ability to build a larger aggregate cache from multiple individual small caches improves performance significantly. Both strategies have at least 2.5× speedup; in fact, the parallel issue strategy approaches 3× speedup, the theoretical maximum for three-way parallelism. (The actual speedup is

2.99.) This is because the cost of accessing the disk dominates the cost to process queries, and caching can reduce this dominant cost. By partitioning the index among three nodes, we have effectively tripled the aggregate cache size. In fact, our experiments indicate that the average hit rate in each node's cache is the same (39 percent) as if there was a single-node index with three times as much cache. Even though each node has a small individual cache (only one percent of the index size), the aggregate effect is that of a large global cache, because each node need only cache blocks from its local partition of the β layer. The increase in effective caching reduces number of disk reads, and this benefit compensates for the "wasted" cycles needed to conduct queries in parallel. The end result is that the system can approach $3\times$ speedup.

We ran another experiment where we varied the number of nodes in a situation where nodes had caches, and measured the speedup and effort. The results (not shown) are similar to those of Figure 9. Specifically, forwarding provides better speedup (up to 8.3 times speedup for ten nodes) than parallel issue when there are more than four nodes. The best speedup observed for parallel issue was 5.9 times speedup for ten nodes.

6. RELATED WORK

Many investigators have examined the problem of distributing and replicating databases to provide load balancing while protecting database consistency [7, 14]. Traditionally, the problem is confined to managing structured data that can be easily partitioned. Semistructured data has received much attention recently [6, 10, 2], although the authors know of no work focused on parallelizing and distributing semistructured indexes.

The most common indexing technology is the B-tree and its variants [4]. The size of B-trees is sensitive to the length of keys as well as the number of keys inserted. As a result, B-trees tend to have many levels, reducing the search efficiency. Another type of index is the hash table, which computes a *hash* function over keys to place keys in separate buckets [13]. Although the hash table can be support efficient querying, it requires some knowledge about the nature of the indexed data so that an appropriate hash function can be selected. Both B-trees and hash tables index homogeneous sets of keys. The Index Fabric operates the same way regardless of the nature of the data. Moreover, a hash table does not support range queries, while the Index Fabric can. Other researchers have examined more flexible tree indexes (e.g. generalized search trees [12]) and further work needs to be done to see if our techniques can be applied to such indexes.

Other researchers have investigated partitioning unbalanced trees. For example, Diwan et al [8] examine a way to cluster vertices of a tree into blocks to minimize external access time. The Index Fabric is similar to these previous efforts in the attempt to achieve balance through tree partitioning. However, our approach is more directly focused on partitioning the tree for use with the parallel mechanism shown in Figure 1.

7. CONCLUSION

We have described how to transform an unbalanced tree into a balanced, partitionable structure. The layered index structure allows a query processor to proceed directly to the relevant portion of the tree, allowing the query to be answered independent of other nodes. This structure allows us to exploit parallel computing to answer multiple queries and to build large indexes by distributing them over commodity hardware. The ability to build and parallelize such a tree structure gives us the ability to represent and search semistructured data relationships effectively.

We have implemented a partitionable, layered index, and conducted experiments that measured the speedup offered by parallelism. For example, our results indicate that three parallel nodes can achieve a factor of 2.5 speedup simply by partitioning the index, and almost a factor of 3 speedup if each node has even a small local cache. Clearly, the Index Fabric is a natural structure for parallel execution of queries over semistructured data.

8. REFERENCES

[1] DBLP computer science bibliography. http://www.informatik.uni-trier.de/~ley/db/.

[2] Serge Abiteboul. Querying semi-structured data. In *Proc. ICDT*, 1997.

[3] Barbara T. Blaustein and Charles W. Kaufman. Updating replicated data during communications failures. In *Proc. VLDB*, pages 49–58, 1985.

[4] D. Comer. The ubiquitous b-tree. *Computing Surveys*, 11(2):121–137, 1979.

[5] Brian Cooper and Moshe Shadmon. The Index Fabric: A mechanism for indexing and querying the same data in many different ways, 2000. RightOrder Incorporated Technical Report.

[6] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proc. SIGMOD*, 1999.

[7] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[8] A. A. Diwan, Sanjeeva Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proc. VLDB*, pages 342–353, 1996.

[9] Brian Cooper et al. A fast index for semistructured data. In *Proc. VLDB*, September 2001.

[10] Jason McHugh et al. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[11] D. Gifford. Weighted voting for replicated data. In *Proc. SOSP*, pages 49–58, 1979.

[12] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proc. VLDB*, pages 562–573, September 1995.

[13] Donald Knuth. *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition.* Addison Wesley, Reading, MA, 1998.

[14] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems (Second Edition).* Prentice Hall, Upper Saddle River, New Jersey, 1999.