Indexing Cached Multidimensional Objects in Large Main Memory Systems *

Beomseok Nam and Alan Sussman
UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742
{bsnam,als}@cs.umd.edu

Abstract

Semantic caches allow queries into large datasets to leverage cached results either directly or through transformations, using semantic information about the data objects in the cache. As the price of main memory continues to drop and its size increases, the size of semantic caches grows proportionately, and it is becoming expensive to compare the semantic information for each data object in the cache against a query predicate. Instead, we propose to create an index for cached objects. Unlike straightforward linear scanning, indexing cached objects creates additional overhead for cache replacement. Since the contents of a semantic cache may change dynamically at a high rate, the cache index must support fast inserts and deletes as well as fast search. In this paper, we show that multidimensional indexing helps navigate efficiently through a large semantic cache in spite of the additional overhead and overall is considerably less expensive than linear scanning. Little emphasis has been laid upon the performance of multidimensional index inserts and deletes, as opposed to search performance. We compare the performance of a few widely used multidimensional indexing structures with our SH-tree*, looking at insert, delete, and search operations, and show that SH-trees overall perform better for large semantic caches than the widely used indexing techniques.*

1 Introduction

Multiple query optimization has been extensively studied in various contexts, including relational databases and data analysis applications [7, 8, 11, 17, 18]. The objective is to exploit subexpression commonality across multiple queries on a set of concurrently executing queries to reduce execution time by reusing cached output data objects. Finding a globally optimal query plan has been shown to be an NP-complete problem [17], so a good solution can only be achieved using heuristics or probabilistic techniques, but multiple query optimization has still been shown to be useful in the context of scientific data analysis applications. Over the last few years, we have developed a distributed multiple query optimization framework (MQO) for scientific data analysis applications [1, 2]. MQO stores query results in distributed semantic caches in order to reuse them for the incoming queries. In order to find out which intermediate results can be reused for a new incoming query, the MQO framework scans the semantic information for all the cached results, which is inefficient when the cache has a large number of results. As the price of main memory drops, it is not uncommon to find machines with many gigabytes of memory that can hold a large number of cached objects. Scanning all the objects in such a large main memory cache is an expensive operation, thus it is imperative to provide a faster cache look-up mechanism.

Since many scientific datasets are multidimensional (i.e. in space and time), they can be indexed using multidimensional spatial tree structures in order to avoid linear scanning. We refer to the multidimensional index for a main memory semantic cache as the *cache index*. The cache index should make semantic cache look-up operations faster, at the cost of making index updates (insertion and deletion of objects) more expensive, because of the overhead for updating the index. Whenever a data object is stored or replaced in the cache, the index needs to be updated to reflect the change. If the index maintenance overhead for updates negates the benefits of the index compared to linear scanning, cache indexing would be of no use. Hence fast index update is as important as fast index search, when cache replacement occurs frequently as in semantic caches.

In the past two decades, a large amount of research has been done to create efficient multidimensional index-

*This research was supported by the National Science Foundation under Grant #EIA-0121161 and NASA under Grant #NAG512652.

ing data structures, including R-trees [9], R*-trees [3], and Hybrid-trees [5]. However, most of that work has focused on search performance for multidimensional indexing structures, while index update performance has been neglected and sacrificed for better search performance. In this paper, we compare the performance of a few widely used multidimensional indexing structures with our *SH-trees*, looking at inserts, deletes, and searches in the context of the cache index. To the best of our knowledge, this is the first study that proposes using a multidimensional index to speed up semantic cache look-up performance.

The rest of this paper is organized as follows. In Section 2 we briefly discuss the MQO semantic cache model. In Section 3 we discuss SH-trees, which support both fast index updates and searches. In Section 4 we present performance results for SH-trees, R-trees, and R*-trees, measuring insertion, deletion, and search performance for various experimental parameters. We conclude in Section 5.

2 Multiple Query Processing Middleware

The Multi-Query Optimization framework (MQO) is a distributed query processing middleware that we have designed and built to support large scale data analysis applications [1, 2]. MQO provides an environment based on C++ abstract operators that are customized when new applications are developed and implemented, or when existing applications are ported. MQO targets several types of computational platforms, transparently employing platform-specific optimizations. From large SMP machines, to clusters of homogeneous nodes, to a distributed heterogeneous Grid environment, MQO is able to use the application-customized operators for efficient query planning and scheduling. MQO offers three main features to improve query processing performance: load balancing, parallel sub-query execution, and semantic caching. In this section, we focus on MQO's semantic cache infrastructure. Figure 1 shows a simplified view of the overall architecture of MQO.

The frontend server interacts with clients to receive queries and return query results. When a query is submitted, the frontend instantiates the corresponding query object and spawns a query thread to execute the query. The query thread searches for cached results in the frontend server that can be reused to either completely or partially answer a query. If the frontend cache does not have complete intermediate or final results for the query, the frontend generates sub-queries for the partial region that was not in the cache, as shown in Figure 2. The current version of MQO stores a query result as a single entry, however this causes redundant objects to be cached. Therefore, we plan to partition the multidimensional space into small chunks as proposed in [6] in order to avoid redundant cached results. Chunk-

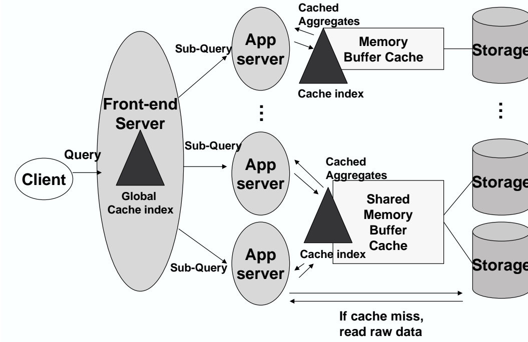


Figure 1. *Overview of MQO framework*

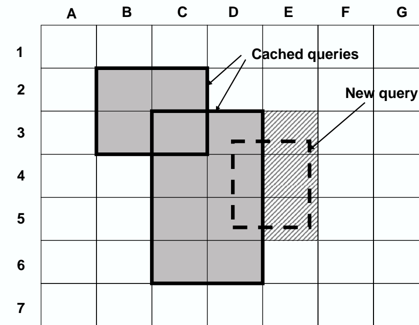


Figure 2. *Sub-queries generated from reusing cached results*

based caching will allow for finer-grained caching, and also helps avoids fragmentation problems due to storing large numbers of very small different sized objects in the cache.

The sub-queries are processed by different application servers in parallel. The application servers also have their own semantic caches, hence partial intermediate results from prior processing are stored in the caches along with semantic information about those intermediate data structures. This makes it possible to use intermediate results to answer queries submitted later. The current version of MQO scans the semantic information for all cached data objects to find out whether they overlap a submitted query. As the size of the cache increases, the amount of semantic information about the cached objects that has to be maintained increases so the time to scan such information also increases. In order to avoid scanning all the semantic information, multidimensional indexing structures can be used to speed up cache look-ups.

Another important reason why MQO needs multidimen-

sional indexing structures is because it is possible to generate better query plans using distributed indexes. The semantic caches in MQO are independent and evict content as needed according to their local cache replacement policies without any global coordination. Therefore, it is very expensive to keep track of the up-to-date contents of remote semantic caches in distributed systems. However, we have shown that a hierarchical distributed multidimensional indexing scheme can help generate better query plans leveraging information about the contents of remote semantic caches [13, 14]. In hierarchical distributed indexing, each server has an independent multidimensional index for its local semantic cache, and the frontend server maintains a global index that stores minimum bounding boxes (MBRs) for local indexes. Hierarchical distributed indexing helps improve performance both for load-balancing and for clustering similar queries near in time to obtain a higher cache hit ratio. In the rest of this paper, we focus on the performance of multidimensional indexing structures as the cache index for a single semantic cache.

3 Multidimensional Indexing Trees

We discuss a few widely used multidimensional indexing tree structures, concentrating on issues related to performance for indexing the semantic cache. Multidimensional indexing structures can be classified into two categories. One category is space partitioning methods, where internal tree nodes are represented by a split dimension and one or two split positions, partitioning the multidimensional space from the root to the leaves of the tree, and putting objects into the right partition in a top-down manner. The second category is data partitioning methods, where internal tree nodes are represented by a list of multidimensional bounding rectangles (MBRs), building up the bounding rectangles by grouping objects from the leaves of the tree in a bottom-up manner.

3.1 Space partitioning methods

A **KDB-tree** is a balanced B-tree version of the binary KD-tree [16]. The main problem with a KDB-tree is that minimum utilization of nodes is not guaranteed because of the *downward cascading split problem*. Moreover, KDB-trees can only index point data as for most of other space partitioning methods, while a cache index may need to store rectangular range query results.

The **Hybrid-tree** solves the downward cascading split problem by allowing overlap of the two sub-regions after a node is split, as for data partitioning methods [5]. An internal node in a Hybrid-tree is a binary KD-tree, and a node contains a splitting dimension and two splitting positions in that dimension. By having two split positions, a Hybrid-tree node can allow an overlapping area that is shared by both child nodes.

3.2 Data partitioning methods

The **R-tree** and the **R*-tree** are the most well-known disk-based data partitioning methods that can index non-point data. Unlike space partitioning methods, an internal tree node for a data partitioning method contains a list of minimum bounding rectangles (MBRs) that can overlap [3, 9]. In data partitioning methods, a large amount of overlap between internal nodes leads to increased search times. To reduce the amount of overlap, R*-trees performs an expensive *forced reinsertion* operation. When a node overflows during an insertion operation the R*-tree algorithm reinserts a user-specified fraction of the child nodes instead of splitting the node, which sacrifices insertion performance for search performance.

3.3 Spatial Hybrid Tree

The Spatial Hybrid tree (SH-tree) is a disk-based space partitioning indexing structure that supports efficient range queries on non-point data objects, especially for high dimensional spaces [12]. The SH-tree combines the properties of the SKD-tree [15] and the Hybrid tree [5], both of which are based on space partitioning methods, and allows overlapping sub-regions by having two split positions. The SKD-tree allows overlapping sub-regions when a mutually disjoint partition is not possible due to the volumes occupied by the data objects, whereas the Hybrid-tree allows overlapping sub-regions when a downward cascading split is unavoidable [5]. In other words, the Hybrid-tree may create a new overlapping region when a node that overflows must be split, while the SKD-tree adjusts overlapping regions so that one region will fully contain a new object that is to be inserted. The SH-tree employs the node splitting algorithm of the Hybrid-tree and the insertion algorithm of the SKD-tree. We now describe the node split and object insertion and deletion algorithms in more detail.

3.3.1 Object Insertion

Figure 3 depicts an internal node of an SH-tree. An internal node for a disk-based space partitioning method such as a KDB-tree [4] is represented as a binary KD-tree, not a list of bounding rectangles as for R-trees. In SH-trees, one split dimension and two split positions are required for each child node in order to allow overlapping regions between child nodes. One split position represents the minimum boundary of the upper (right) region ($minU$) and the other the maximum boundary of the lower (left) region ($maxL$) in the split dimension.

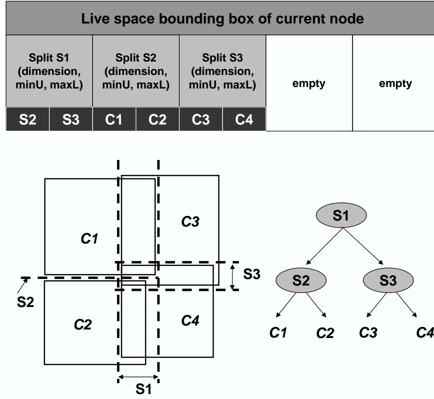


Figure 3. *KD-tree representation of an internal node of an SH-tree*

When a new data object is inserted into a node in the SH-tree, the insertion algorithm compares the MBR of the object with the split information in the root node of the internal KD-tree. If the object is completely inside one of two sub partitions in the root level, the algorithm repeats the same comparison in the next lower level in the KD-tree of the node until the object reaches a leaf node, which points to a child node in the SH-tree. However if the object does not fit completely inside either of the two sub-partitions, either $minU$ or $maxL$ for the node must be adjusted to include the object. Which one is adjusted is determined based on which sub-partition causes less enlargement of the region, to minimize the size of the overlapping region ($maxL - minU$).

One of the benefits of the KD-tree internal node representation is reduced insertion algorithm complexity. R-tree based indexing structures use a list of bounding rectangles in an internal node. Therefore, in order to determine which child node should be assigned a newly inserted object, the R-tree insertion algorithm must compare the query with the MBRs of all child nodes, which requires $\#dimensions \times n$ comparisons of real numbers, where n is the node capacity (number of children). On the other hand, the the SH-tree insertion algorithm performs only $\log n$ comparisons when the internal KD-tree is balanced, but n comparisons in the worst case (when the tree is highly skewed), which is still faster than R-tree insertion algorithm.

3.3.2 Node Split

The $minU$ and $maxL$ values, and the split dimension, are locally optimized to reduce the overlap when a node that overflows must be split. The goal of the node split algorithm for SH-trees is to minimize the distance between the two split positions ($maxL - minU$). For an N-dimensional dataset, only one of the dimensions is used as a split dimension. For each dimension, the bounding boxes of the child sub-regions of the node to be split are sorted twice, based on their lower and upper boundaries in the split dimension. The sub-region with the lowest upper bound and the sub-region with the highest lower bound are selected and put into the lower and upper resulting regions respectively, until the minimum required node utilization is reached. When the minimum required node utilization for both regions is reached, it must be determined which region will increase in size the least if each remaining sub-region is inserted into that region. In this way, all the children are placed into the two resulting regions to achieve minimal overlap in the split dimension. This process is performed for each dimension, and the dimension that causes the smallest overlapping region is chosen to be split. After $minU$ and $maxL$ values and the split dimension are chosen, the split information is stored in the parent node of the node to be split. The complexity of the split algorithm of SH-trees is proportional to the cost of the sorting algorithm, $O(\#dimensions \cdot n \log n)$, where n is the node capacity (maximum number of child nodes). The goal of the R-tree node split algorithm is to minimize the volumes of the resulting MBRs. While an exhaustive algorithm generates all possible splits, that is too expensive in general, so most R-tree implementations employ one of two heuristics. Quadratic split selects the next child entry to assign to one of the two new nodes by selecting the child node that requires the minimum expansion of a current new node MBR, and linear split simply chooses the next child node in the node list to place into one of the two new nodes. The SH-tree node split algorithm has lower complexity than the quadratic split policy used for R-trees ($O(n^2)$).

3.3.3 Object Deletion: Live Space Bounding Box

In both SKD-trees and Hybrid-trees, deletion is a problem because of the overlapping regions between nodes. When a data object that caused the creation of an overlapping region is deleted from the tree, and if the overlapping region is not necessary for other data objects in a node, the overlapping region should be removed in order to make index search faster. However, no such mechanism exists for either SKD-trees or Hybrid-trees. In SKD-trees, the overlapping regions only grows, and in Hybrid-trees the overlapping regions do not change once they are created, which is possible because hybrid trees do not support non-point data. This unnecessary overlap problem is mainly because splitting positions are shared by multiple child nodes. The shared split positions generate approximate (not tight) bounding boxes for child nodes, and there is no way of knowing the precise occupied regions within child nodes unless all sub-trees are searched. In R-trees, condensing bounding boxes is not a problem, because the bounding box information in an inter-

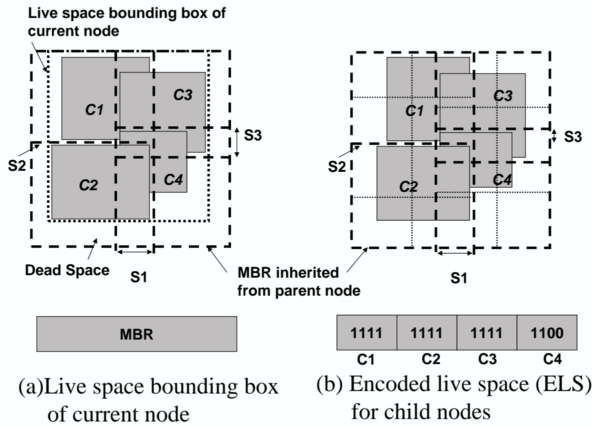


Figure 4. *Dead space elimination: live space bounding box vs. live space encoding*

nal node is the precise information for all its sub-trees.

In order to solve this problem, SH-trees store the minimum bounding box information in the node itself instead of in the parent, as shown in Figure 3. With this additional bounding box information, which we refer to as a *live space bounding box*, SH-trees can avoid searching all sub-trees in order to condense overlapping regions. Instead, the deletion algorithm needs to access a small number of child nodes to determine the actual overlap. The live space bounding box also solves the *dead space problem* of space partitioning methods (i.e. the regions in the MBR of an internal node where no actual data objects are located.).

There have been some previous efforts to solve the dead space problem, such as the *ELS* (Encoding Live Space) data structure used for Hybrid-trees. ELS divides the MBR of a child node into a regular grid and encodes an occupancy map using a small number of bits, as shown in Figure 4. ELS helps improve search performance, but it is not sufficient to condense overlapping regions. ELS gives an approximate hint for the bounding boxes of the child nodes. Contrary to ELS, the algorithm using the live space bounding box must access the child nodes to get precise bounding box information so that it can condense the overlapping region appropriately. When precise bounding boxes for child nodes are known, it is simple to remove unnecessary overlap. First start from the leaf node whose minimum bounding box was condensed from deleting the object. The algorithm proceeds to the parent node and compares the parent node MBR with the split information in the parent node. In order to check whether the split position can be shifted to reduce the overlap, the algorithm must visit the child nodes of the parent that caused the overlap, in order to get live space bounding boxes for those nodes. After accessing the live space bounding boxes for the children, if the live space bounding box of the parent node can be condensed, then this process is performed recursively up the tree until the root node is reached.

Although both the ELS and live space bounding box data structures improve range query performance, they make the number of fan-outs (number of child nodes) for a tree node dependent on the number of dimensions of the data. Higher fan-out is better because it makes the tree height smaller, which makes the paths through the tree for search and insertion shorter. The number of fan-outs for R-trees, Hybrid trees with ELS, and SH-trees with the live space bounding box are as follows: [1]

1. R-trees:

$$\frac{PageSize}{\#dimensions \cdot (S(lowerBound) + S(upperBound)) + S(ChildPointer)}$$

2. Hybrid trees:

$$\frac{PageSize}{S(ELS) + S(minU) + S(maxL) + S(splitDim) + S(ChildPointer)}$$

3. SH-trees:

$$\frac{PageSize - \#dimensions \cdot (S(lowerBound) + S(upperBound))}{S(minU) + S(maxL) + S(splitDim) + S(ChildPointer)}$$

For R-trees, the node fan-out is inversely proportional to the number of dimensions, and similarly for Hybrid trees, because the amount of space for ELS encoding is proportional to the number of dimensions. For SH-trees with the live space bounding box, the number of dimensions only decreases the numerator in the formula, so the number of fanouts for SH-trees decreases linearly with the number of dimensions. Hence, for high dimensional data SH-trees have a larger number of child nodes for a given node compared to R-tree based structures.

3.3.4 Object Deletion: Merging

The deletion algorithm for SH-trees is similar to the one for R-trees. When a node is underutilized, its child nodes are reinserted from the root node. However, we expect that reinsertion might be expensive for the cache index, although it improves the tree structure for searches. Hence we designed an alternative reinsertion strategy. Instead of reinserting from the root node, the child nodes of the underutilized node are reinserted from the parent node of the underutilized node. This alternative reinsertion strategy is similar to merging the underutilized node with its sibling nodes. Figure 5 shows an example. Suppose C1 is to be deleted since it does not have enough child nodes, denoted by the gray boxes. Although C1 is deleted, its dangling child nodes must be reinserted somewhere in the tree. Thus, we need to determine which sibling node will contain each

[1] S() in the formulas denotes the number of bytes needed to represent the value

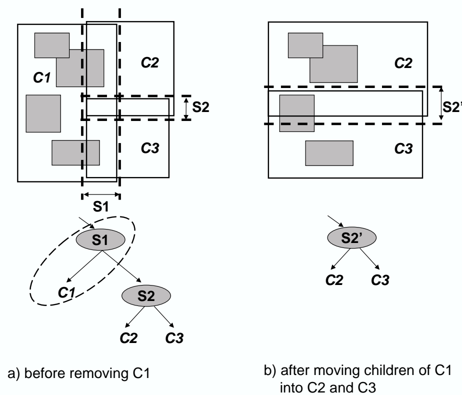


Figure 5. *Merging an underutilized node*

dangling node, and the split positions of the affected nodes must change accordingly. However this merging process is not as simple as it seems. If a sibling node is full, the merging process will make the sibling split, and the parent node may also split recursively. If the parent node splits, there is a problem: which parent should be used for the rest of the dangling child nodes? Our answer to this question is that we do not split the parent node. For each dangling node, if the chosen sibling node is full, the algorithm puts the dangling child node into a reinsertion queue. After all the dangling child nodes are merged into the sibling nodes or put in the reinsertion queue, the algorithm reinserts the child nodes in the reinsertion queue from the root node, as for R-trees.

4 Experiments

4.1 Case Study Application - Kronos

Remote sensing has become a powerful tool for geographical, meteorological, and environmental studies. Systems processing remotely sensed data often provide on-demand access to raw data and user-specified data product generation. Kronos is an example of such a class of applications. It targets datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer - Global Area Coverage) orbit data. The raw data is continuously collected by multiple satellites and the volume of data for a single day is about 1 GB. Each sensor reading is associated with a position (longitude and latitude) and the time the reading was recorded.

4.2 Query Workload Generator

In order to generate query workloads, we employed a variation of the Customer Behavior Model Graph (CBMG) technique. CBMG has been utilized by researchers analyzing performance aspects of e-business applications and website capacity planning [10]. A CBMG can be characterized by a set of $n \times n$ matrix, $P = [p_{i,j}]$, of transition probabilities between the n states.

In our model, the first query in a batch specifies a geographical region and a set of temporal coordinates (a continuous period of days). The subsequent queries in the batch are generated based on the following operations: a *new point of interest*, *spatial movement*, *temporal movement*, *resolution increase or decrease*. In the batch query workload, there are 200 hot points of interest; for example, New Orleans on 29 August 2005. In this way, subsequent queries after the first one in the batch may either remain around that point (moving around its neighborhood) or move on to a different point. These transitions are controlled according to the transition probabilities.

For the experiments, we generated 2,000 3-dimensional queries (latitude, longitude, and time) with various different probabilities, but we only show the results using one of the transition probabilities due to space limitations. Results for other transition probabilities that we experimented with were similar to those shown. We also do not show performance results for high dimensional datasets because scientific datasets usually have four or fewer dimensions (i.e., space and time). Instead of running the real MQO middleware, we modeled the semantic cache behavior of MQO and measured only the cache index performance, without including the time to read the raw AVHRR data from disk.

We partitioned the AVHRR data for one year (365GB) into approximately six million chunks. The size of each chunk is 2 degrees in both latitude and longitude, and 24 hours in time ($\frac{180}{2} \times \frac{360}{2} \times 365$). Since the size of the dataset for one day is about 1 GB, the size of each chunk is about 61 KB. When the cache filled, we employed a least recently used (LRU) cache replacement policy. When the cache size is smaller than 1 GB, more than 1 million chunks are replaced. The overhead to update the cache index for such a large number of chunks is significant. However the experiments we present show that this high overhead is minimal compared to the cost of linear scanning.

Chunk size is an important performance factor for the cache index. For smaller chunk sizes, the overhead for the index increases because more chunks are inserted and deleted for a query. If chunk size is equal to or larger than the query range size, the maximum number of cache replacements will be $2^{\#dimensions}$, which is the number of chunks that share a corner in the multidimensional space. However if the chunk size is larger than the average query range size, the overhead for a cache miss will be significant and the cache hit ratio also decreases. If it is possible to predict the average volume of query ranges (query selectivity), it would be best to make the chunk size the same as the

average query volume, to minimize index update overhead.

Since it is very difficult to predict average query volume, for the rest of experiments, except for the ones shown in Figure 7, based on our user model we set the query volume to span about 30 chunks on average with a standard deviation of about 33. Hence, cache replacement happens hundred of times more frequently than search operations in the cache. While fine-grained chunking increases the overhead to update the cache index, it improves the overall system performance by increasing the cache hit ratio. In this paper, we show the results from the experiments with fine-grained workloads that have large replacement overheads, and the cache index still performs better overall than linear scanning. Experiments with other workloads that have smaller overheads, not shown in this paper, show that the cache index has much better performance than linear scanning in those cases.

4.3 Experimental Setup

We measured the performance of both index creation and search using the SH-tree, R-tree, and R*-tree algorithms. Since the insertion performance is an important performance factor in our experiments, we looked at the linear split and quadratic split heuristics for the R-tree node split algorithm, since they are much faster than the exhaustive split algorithm.

The experiments were run on a Linux machine with a 2.4GHz Intel Pentium 4 processor and 512 MB memory. Although we simulate 12GB of memory for the experiments, that much physical memory is not needed because the experiments do not store the actual raw datasets in the cache. We used open source R-tree and R*-tree implementations developed by Marios Hadjieleftheriou at the University of California at Riverside. We fixed the node utilization factor (the minimum number of child nodes of a valid non-root node divided by the node capacity) to 40% (which is a common value used in many R-tree implementations), and the page size to 1 KB, except for the experiments shown in Figure 6 that vary the page size.

4.4 Experimental Results

Figure 6 illustrates how node fan-out for the cache index trees affects performance. While R-trees can hold 35 3-dimensional child pointers in an internal node for a page size of 1 KB, SH-trees can hold 38 child pointers. Note that the index tree node size does not have to be the same as the disk page size since the cache index resides in main memory. Thus we measured the performance of the cache index as a function of tree node size. Contrary to search performance, insertion and deletion time increases as node size increases. For an 8 KB node size, each node has 291 child pointers for R-trees while an SH-tree node has 313 child pointers. With fewer child pointers, the insertion and deletion algorithms require less computation time to split and recalculate the bounding boxes for tree nodes. However, the longer search paths caused by smaller fan-out increase search time, especially for SH-trees. Because the internal tree nodes for SH-trees share split positions, reducing the node fan-out leads to larger overlapping regions for SH-trees.

With all the disk-based balanced tree structures, the node fan-out must be larger than 3, so that we can split the node without violating the minimum node utilization constraint (40% for these experiments). Thus we could not run the experiments for node size smaller than 256 bytes. The total execution time slightly increases when the node size is smaller than 512 bytes. The deletion time for R*-trees also grows rapidly when the node size is smaller than 512 bytes. Figure 6 shows that SH-trees with the merging optimization shows the best performance in most cases. Also, most indexing structures show good performance when the node size is 512 or 1024 bytes. Thus, for the rest of our experiments, we fixed node size to 1KB.

Figure 7 shows performance results with various chunk sizes. The size of a chunk determines the number of cache replacements. For smaller chunk sizes, more cache replacements occur so the overhead for updating the cache index increases. In order to reduce the cache index overhead, the chunk size should be chosen as big as possible, but not so big that it significantly decreases the cache hit ratio.

Figure 8(a) shows the wall clock time to insert metadata for the new chunks into the cache index. We increased the size of the cache from 1.2GB to 12GB. For smaller caches more cache misses occur, so that more chunks are inserted and deleted. For insertions, SH-trees show the best performance and R-trees using the linear split algorithm show the next best performance. Note that SH-trees with the merging deletion algorithm have no performance difference from SH-trees with the reinsertion algorithm for insertions. R*-trees shows the worst performance for insertions because of forced reinsertion for nodes that overflow.

The deletion performance results are not much different from insertion performance. R*-trees again suffer from forced reinsertion. When a node is underutilized, its child nodes must be reinserted from the root node, as for R-trees. But the objects that are reinserted may cause other node overflows, again causing the child nodes from those overflowing node to be reinserted from the root again. Note that R-trees do not reinsert child nodes from an overflowing node. Thus the forced reinsertion algorithm of R*-trees affects insertion performance as well as deletion performance. Thus, the overhead for deletion in R*-trees is much higher than that of R-trees, although both trees perform reinsertion operations for underutilized nodes.

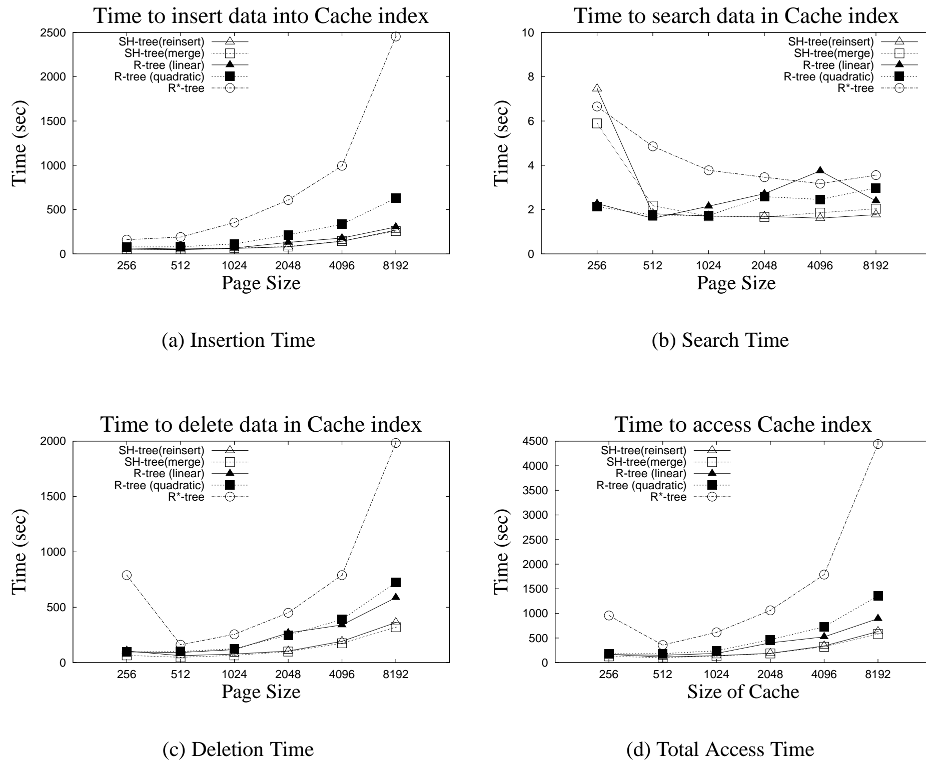
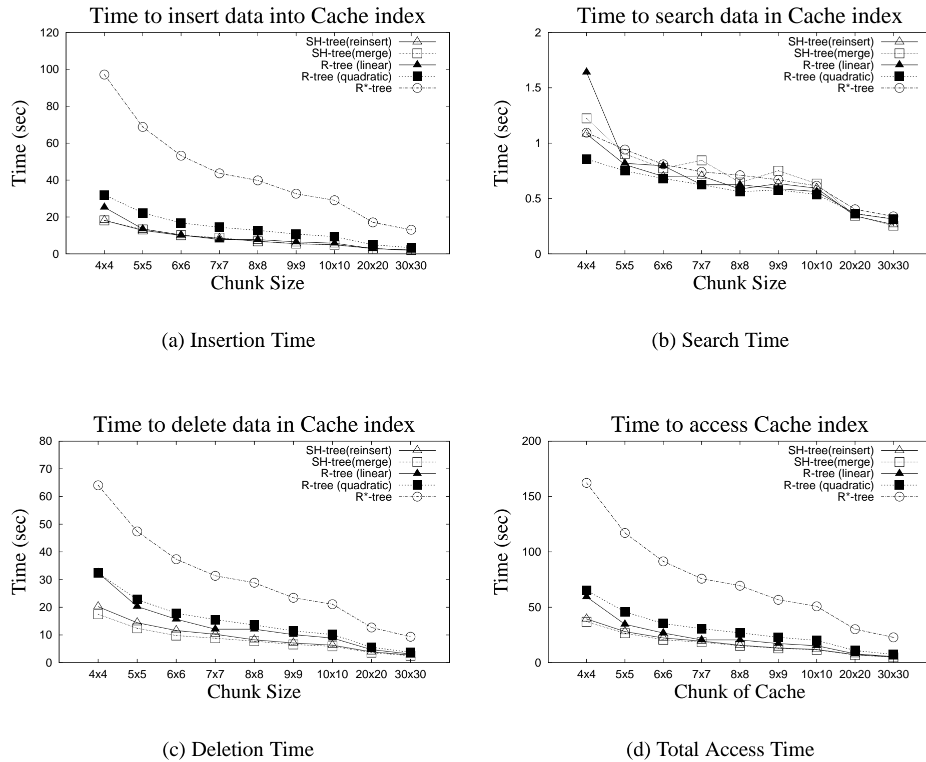


Figure 6. *Total Cache Index Access Time for 2,000 Queries with Various Page Size*

All the indexing structures have the same deletion algorithm – reinsertion from underutilized nodes – except SH-trees with the merging algorithm. Therefore, deletion performance is dependent on the insertion algorithm, i.e. how fast they can perform reinsertion. As shown in Figure 8(c), the merging algorithm for SH-trees helps improve deletion performance. The deletion performance of SH-trees with the merging algorithm was up to 16% faster than that of the SH-tree with reinsertion. This merging algorithm can be applied to any other R-tree based indexing structure without much difficulty, but it will hurt search performance by increasing overlapping regions in child nodes. The search time for SH-trees with the merging algorithm was about 8% slower than that of SH-trees with the reinsertion algorithm. Since the node fan-out is about the same for all the indexing algorithms when the number of dimensions is 3 as in our experimental dataset, search performance for SH-trees is not better than for the other indexing trees.

It is a surprising result that the search performance of R*-trees is not better than that of R-trees. In experiments not shown in the paper, the search performance of R*-trees was shown to be several times slower than that of R-trees in some cases. The main performance improvement for R*-trees over R-trees is from restructuring the internal nodes via forced reinsertion. However, in our experiments R-trees also perform many reinsertions due to frequent deletions. Thus the internal tree structure of R-trees becomes as good as that of R*-trees due to the large number of reinsertions. The idea of forced reinsertion in R*-trees first came from the observation that R-tree search performance significantly improves after many deletions.

SH-trees show better performance when counting tree node accesses as well overall execution time for all operations (search, insert, and delete). SH-trees with the merging algorithm access 8-10% fewer tree nodes than do R-trees and 22-29% fewer than R*-trees. Although SH-trees do not perform best for searching, search time does not take a big portion of the overall execution time. Figure 8(d) shows total execution time, and shows that search time is less than 1% of the total execution time for all the index trees. Insertion time takes from 43-62% of the total execution time, and deletion takes from 37-56%. Figure 8(d) also shows the performance of linear scanning without a cache index. The search performance of linear scan increases as the cache size grows, because the large number of objects in the cache makes linear scanning expensive. Linear scan does not have the overhead of insertion or deletion, but scanning itself is expensive enough that when the cache has more than 40,000



(a) Insertion Time

(b) Search Time

(c) Deletion Time

(d) Total Access Time

Figure 7. *Total Cache Index Access Time for 2,000 Queries with Various Chunk Sizes*

chunks linear scan performs worse than even R*-trees. Using various other workloads that we generated, not shown in the paper due to space limitations, we observed that the performance of linear scanning becomes worse than using the indexes as the cache hit ratio increases, which reduces the overhead of cache replacement. From these results, we claim that cache indexing is worthwhile for caching objects in large main memory systems.

5 Conclusion

In this paper, we have discussed how multidimensional indexing structures can help search for cached objects in a large semantic cache. Experimental results show that even R*-trees, which do not perform particularly well, perform better than linear scanning and the performance benefits grow as the size of the semantic cache increases. Compared to several widely used multidimensional indexing structures, we have shown that SH-trees perform best because of more efficient insertion and deletion algorithms.

A future direction of this work is to integrate a cache replacement policy with the cache index so that deletion overhead can be avoided. We plan to evaluate the cache index with the real working multiple query optimization middleware (MQO), and to extend this work to additional data analysis applications as well as different workload profiles.

References

[1] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient execution of multiple query workloads in data analysis applications. In *Proceedings of the ACM/IEEE SC1001 Conference*, Nov. 2001.

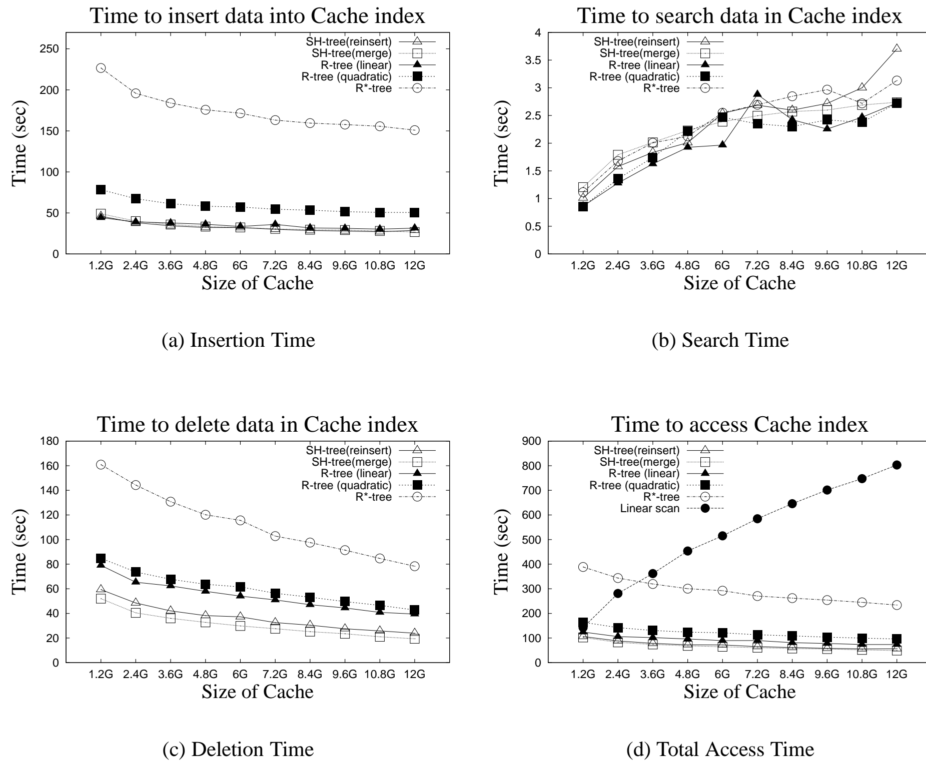
[2] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the ACM/IEEE SC2002 Conference*, Nov. 2002.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD90)*, pages 322–331, May 1990.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM 18(9)*, 1975.

[5] K. Chakrabarti and S. Mehrotra. The Hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering (ICDE99)*, pages 440–447, 1999.

[6] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton:. Caching multidimensional queries using chunks. In *Proceed-*



(a) Insertion Time

(b) Search Time

(c) Deletion Time

(d) Total Access Time

Figure 8. *Total Cache Index Access Time for 2,000 Kronos Queries*

ings of the ACM SIGMOD International Conference on Management of Data (SIGMOD98), 1998.

[7] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proceedings of the 9th International Conference on Extending Databases Technology (EDBT)*, 2004.

[8] P. Godfrey and J. Gryz. Answering queries by semantic caches. In *DEXA '99: Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pages 485–498, 1999.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD84)*, pages 47–57, 1984.

[10] D. Menasce and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall PTR, 2000.

[11] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in PLACE. In *Proceedings of the 2nd Workshop on Spatio-temporal Databases Management (STDBM)*, 2004.

[12] B. Nam and A. Sussman. A comparative study of spatial indexing techniques for multidimensional scientifi c datasets. In *Proceedings of 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, June 2004.

[13] B. Nam and A. Sussman. Spatial indexing of distributed multidimensional datasets. In *Proceedings of CCGrid2005: IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2005.

[14] B. Nam and A. Sussman. DiST: Fully decentralized indexing for querying distributed multidimensional datasets. In *Proceedings of IPDPS2006: IEEE International Parallel and Distributed Processing Symposium*, 2006.

[15] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC Conference*, 1987.

[16] J. T. Robinson. The K-D-B tree: A search structure for large multi-dimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD81)*, 1981.

[17] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[18] X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Scalable spatio-temporal continuous query processing for location-aware services. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2004.